.

第十四章

块、语句和模式

程序的执行顺序是由语句控制的，语句的执行是为了它们的效果，没有值。

有些语句包含其他语句作为其结构的一部分;这样的其他语句是语句的子语句。我们称语句 S 直接包含语句 U，如果没有与 S 和 U 不同的语句 T，使得 S 包含 T，T 包含 U。同样，有些语句包含表达式(§15(表达式))作为其结构的一部分。

本章第一节讨论了语句的正常完成和突然完成之间的区别(第 14.1 节)。剩下的大部分章节解释了各种各样的语句，详细描述了它们的正常行为和任何对突然完成的特殊处理。

首先解释块(§14.2)，一方面是因为它们可以出现在某些语句不允许的地方，另一方面是因为一种语句-局部变量声明语句(§14.4.2)，必须立即包含在一个块中。局部类和接口声明(§14.3)不是语句，但也必须立即包含在一个块中。

接下来，解释了一种回避常见的“悬空 else”问题（§14.5）的语法策略。

每个语句必须在某种技术意义上是可达到的（§14.22）。

第 14.23-14.29 节未使用，以便将来引入新的语句类型。

本章最后一节（§14.30）描述了在语句和表达式中用于有条件地声明和初始化局部变量的模式。模式对一个值（如对象）如何由一个或多个其他值（由变量声明表示）组成进行了简要描述。模式匹配试图从一个给定值中提取一个或多个值，就像对其进行分解一样，并使用提取的值初始化模式声明的变量。

14.1 语句的正常和突然完成

每个语句都有一个正常的执行模式，其中执行某些计算步骤。以下各节描述了每种语句的正常执行模式。

如果所有步骤都按所述执行，没有突然完成的迹象，则该语句称为正常完成。但是，某些事件可能会阻止语句正常完成:

- break, yield, continue, 和 return 语句(§14.15, §14.21, §14.16, §14.17) 导致控制权转移，可能会阻止表达式、语句和包含它们的块的正常完成。
- 某些表达式的计算可能会从 Java 虚拟机引发异常（§15.6）。显式抛出（§14.18）语句

也会导致异常。异常会导致控制权转移，这可能会阻止语句的正常完成。

如果发生这样的事件，则一个或多个语句的执行可能在其正常执行模式的所有步骤完成之前终止；这样的语句被称为突然完成。

突然完成总是有关联的原因，可以是以下原因之一：

- 没有标签的 break
- 有给定标签的 break
- 没有标签的 continue
- 有给定标签的 continue
- 不带值的 return
- 带给定值的 return
- 具有给定值的抛出，包括 Java 虚拟机抛出的异常
- 带给定值的 yield

术语“正常完成”和“突然完成”也适用于对表达式的计算(§15.6)。表达式可以突然完成的唯一原因是抛出异常，原因要么是具有给定值的抛出(§14.18)，要么是运行时异常或错误(§11，§15.6)。

如果语句计算表达式，则出于相同的原因，表达式的突然结束总是导致语句立即突然结束。不执行正常执行模式中的所有后续步骤。

除非本章另有说明，否则突然完成子语句会导致语句本身立即突然完成，原因相同，并且不会执行语句正常执行模式中的所有后续步骤。

除非另有指定，否则如果语句计算的所有表达式和执行的所有子语句都正常完成，则该语句正常完成。

14.2 块

块是一系列语句、局部变量声明语句以及大括号内的局部类和接口声明。

Block:
{ *[BlockStatements]* }

BlockStatements:
BlockStatement {BlockStatement}

BlockStatement:
LocalClassOrInterfaceDeclaration
LocalVariableDeclarationStatement
Statement

一个块是通过按照从第一到最后（从左到右）的顺序执行每个局部变量声明语句和其他语句来执行的。如果所有这些块语句都正常完成，则块正常完成。如果这些块语句中的任何一个由于任何原因而突然完成，则该块由于相同的原因而突然结束。

14.3 局部类和接口声明

局部类是嵌套类（§8），其声明立即包含在块（§14.2）中。

局部接口是嵌套接口（§9），其声明立即包含在块中。

LocalClassOrInterfaceDeclaration:
ClassDeclaration
NormalInterfaceDeclaration

为了方便起见，此处显示了以下产品：

ClassDeclaration:
NormalClassDeclaration
EnumDeclaration
RecordDeclaration

NormalClassDeclaration:
{ClassModifier} `class` *TypeIdentifier [TypeParameters] [ClassExtends] [ClassImplements] [ClassPermits] ClassBody*

EnumDeclaration:
{ClassModifier} `enum` *TypeIdentifier [ClassImplements] EnumBody*

NormalInterfaceDeclaration:
{InterfaceModifier} `interface` *TypeIdentifier [TypeParameters] [InterfaceExtends] [InterfacePermits] InterfaceBody*

局部类和接口声明可以与包含块中的语句（包括局部变量声明语句）自由混合。

如果局部类或接口声明具有访问修饰符 public, protected,或 private 中的任何一个，则它是编译时错误(§6.6)。

如果局部类或接口声明具有修饰符 static (§8.1.1.4)、sealed 或 non-sealed (§8.1.1.2、§9.1.1.4)，则为编译时错误。

如果局部类的直接超类或直接超接口是密封的，则它是编译时错误。

如果局部接口的直接超接口是密封的，则它是编译时错误。

局部类可以是普通类(§8.1)、枚举类(§8.9)或记录类(§8.10)。每个局部正态类都是一个内部类(§8.1.3)。每个局部普通类都是一个内部类(§8.1.3)。每个局部枚举类和局部记录类都是隐式静态的(§8.1.1.4)，因此不是内部类。

局部接口可以是普通接口(§9.1)，但不是注解接口(§9.6)。每个局部接口都是隐式静态的(§9.1.1.3)。

与匿名类(§15.9.5)一样，局部类或接口不是任何包、类或接口的成员(§7.1、§8.5)。与匿名类不同，局部类或接口有一个简单的名称(§6.2、§6.7)。

局部类或接口声明的作用域和遮蔽在§6.3 和§6.4 中有规定。

例子 14.3-1. 局部类声明

以下是一个示例，说明了上面给出的规则的几个方面：

```
class Global {
    class Cyclic {}

    void foo() {
        new Cyclic(); // create a Global.Cyclic

        class Cyclic extends Cyclic {} // circular definition

        {
            class Local {}
            {
                class Local {} // compile-time error
            }
            class Local {} // compile-time error
            class AnotherLocal {
               void bar() {
                    class Local {} // ok
               }
            }
        }
        class Local {} // ok, not in scope of prior Local
   }
}
```

方法 foo 的第一个语句创建成员类 Global.Cyclic 的一个实例，而不是局部类 Cyclic 的实例，因为语句出现在局部类声明的作用域之前。

局部类声明的作用域包含它的整个声明(不仅仅是它的主体)这一事实意味着局部类 Cyclic 的定义确实是循环的，因为它扩展了自己而不是 Global.Cyclic。因此，局部类 Cyclic 的声明在编译时被拒绝。

由于局部类名不能在同一方法（或构造函数或初始化器，视情况而定）中重新声明，Local 的第二次和第三次声明会导致编译时错误。然而，Local 可以在另一个嵌套更深的类（如 AnotherLocal 类）的上下文中重新声明。

Local 的最终声明是合法的，因为它发生在任何 Local 事先声明的作用域之外。

14.4 局部变量声明

局部变量声明声明并可选地初始化一个或多个局部变量（§4.12.3）。

LocalVariableDeclaration:
{VariableModifier} LocalVariableType VariableDeclaratorList

LocalVariableType:
UnannType
`var`

参见§8.3 了解 Unnantype。为方便起见，此处显示了§4.3、§8.3 和§8.4.1 中的以下产品：

VariableModifier:
Annotation
`final`

VariableDeclaratorList:
VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
VariableDeclaratorId [= VariableInitializer]

VariableDeclaratorId:
Identifier [Dims]

Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

VariableInitializer:
Expression
ArrayInitializer

局部变量声明可以出现在以下位置：

- 块中的局部变量声明语句(§14.4.2)
- 基本 for 语句的头(§14.14.1)
- 增强 for 语句的头 (§14.14.2)
- try-with-resources 语句的资源规范 (§14.20.3)
- 模式 (§14.30.1)

关于局部变量声明的注解修饰符的规则在§9.7.4 和§9.7.5 中规定。

如果关键字 final 作为局部变量声明的修饰符出现，则局部变量是 final 变量(§4.12.4)。

如果 final 作为局部变量声明的修饰符多次出现，则为编译时错误。

如果 LocalVariableType 为 var 并且满足以下任何一个条件，则为编译时错误：

- 列出了多个 VariableDeclarator。
- VariableDeclaratorId 具有一个或多个方括号对。
- VariableDeclarator 缺少初始化器。
- VariableDeclarator 的初始化器是一个 ArrayInitializer。
- VariableDeclarator 的初始化器包含对变量的引用。

例子 14.4-1. 用 var 声明的局部变量

以下代码说明了这些限制使用 var 的规则:

```
var a = 1;            //  Legal
var b = 2, c = 3.0;   //  Illegal: multiple declarators
var d[] = new int[4]; //  Illegal: extra bracket pairs
var e;                //  Illegal: no initializer
var f = { 6 };        //  Illegal: array initializer
var g = (g = 7);      //  Illegal: self reference in initializer
```

这些限制有助于避免混淆 var 表示的类型。

14.4.1 局部变量声明和类型

局部变量声明中的每个声明符声明一个局部变量，其名称是出现在声明符中的标识符。

如果可选关键字 final 出现在声明的开头，那么被声明的变量就是 final 变量(§4.12.4)。

局部变量声明的类型如下所示:

- 如果 LocalVariableType 为 UnannType，并且 UnannType 或 VariableDeclaratorId 中没有出现括号对，则局部变量的类型由 UnannType 表示。
- 如果 LocalVariableType 为 UnannType，并且括号对出现在 UnannType 或 VariableDeclaratorId 中，则局部变量的类型由§10.2 指定。
- 如果 LocalVariableType 是 var，则当 T 被视为不出现在赋值上下文中时，T 就是初始化表达式的类型，因此是一个独立的表达式(§15.2)。局部变量的类型是 T 相对于 T 提到的所有合成类型变量的向上投影(§4.10.5)。

  如果 T 为空类型，则为编译时错误。

  因为初始化器被视为没有出现在赋值上下文中，所以如果它是 lambda 表达式(§15.27)或方法引用表达式(§15.13)，则会出现错误。

局部变量声明的作用域和遮蔽见§6.3 和§6.4。

对嵌套类或接口或 lambda 表达式的局部变量的引用是受限制的，如§6.5.6.1 中规定。

例子 14.4.1-1. 用 var 声明的局部变量的类型

下面的代码演示了用 var 声明的变量的类型:

```
var a = 1;                // a has type 'int'
var b = java.util.List.of(1, 2);  // b has type 'List<Integer>'
var c = "x".getClass(); // c has type 'Class<? extends String>'
                          // (see JLS 15.12.2.6)
var d = new Object() {};  // d has the type of the anonymous class
var e = (CharSequence & Comparable<String>) "x";
                          // e has type CharSequence & Comparable<String>
var f = () -> "hello"; // Illegal: lambda not in an assignment context var
g = null;                 // Illegal: null type
```

注意，有些用 var 声明的变量不能用显式类型声明，因为变量的类型是不可表示的。

在确定变量类型时，向上投影应用于初始化器的类型。如果初始化器的类型包含捕获变量，则此投影将初始化器的类型映射到不包含捕获变量的超类型。

虽然允许变量的类型提到捕获变量是可能的，但通过将它们投影出去，我们强制一个吸引人的不变量，即捕获变量的作用域永远不会大于包含被捕获类型的表达式的语句。非正式地，捕获变量不能“泄露”到后续语句中。

14.4.2 局部变量声明语句

局部变量声明语句由局部变量声明组成。

LocalVariableDeclarationStatement:

LocalVariableDeclaration ;

每一个局部变量声明语句都立即包含在一个块中，而其他类型的语句(§14.5)既可以立即包含在一个块中，也可以立即包含在另一个语句中。

在包含块中，局部变量声明语句可以与其他类型的语句以及局部类和接口声明自由混合。

局部变量声明语句是一种可执行语句。每次执行时，声明符都按从左到右的顺序处理。如果声明符有初始化器，则对初始化器求值并将其值赋给变量。

如果声明符没有初始化器，那么每次对变量的引用都必须在执行对变量的赋值操作之前，否则就会出现§16(明确赋值)中的编译时错误。

每个初始化器(第一个除外)只有在前一个初始化式的计算正常完成时才会被求值。

只有当最后一个初始化器的求值正常完成时，局部变量声明语句的执行才正常完成。

如果局部变量声明语句中的声明符都没有初始化器，那么执行语句总是正常完成。

14.5 语句

Java 编程语言中有许多种语句。大多数与 C 和 C++语言中的语句相对应，但也有一些是独特的。

与 C 和 C++一样，Java 编程语言的 if 语句也存在所谓的“悬空 else 问题”，下面这个格式误导的示例说明了这一点:

```
if (door.isOpen())
    if (resident.isVisible())
      resident.greet("Hello!");
else door.bell.ring(); // A "dangling else"
```

问题是外部的 if 语句和内部的 if 语句都可能拥有 else 子句。在这个例子中，人们可能会猜测程序员想让 else 子句属于外部 if 语句。

Java 编程语言，像 C 和 C++以及之前的许多编程语言一样，武断地将 else 子句归属于它可能所属的最里面的 if 子句。该规则由以下语法捕获:

Statement:
StatementWithoutTrailingSubstatement
LabeledStatement
IfThenStatement
IfThenElseStatement
WhileStatement
ForStatement

StatementNoShortIf:
StatementWithoutTrailingSubstatement
LabeledStatementNoShortIf
IfThenElseStatementNoShortIf
WhileStatementNoShortIf
ForStatementNoShortIf

StatementWithoutTrailingSubstatement:
Block
EmptyStatement
ExpressionStatement
AssertStatement
SwitchStatement
DoStatement
BreakStatement
ContinueStement
ReturnStatement
SynchronizedStatement
ThrowStatement
TryStatement
YieldStatement

为了方便起见，以下是§14.9 的产品:

IfThenStatement:
`if ( `*Expression*` ) `*Statement*

IfThenElseStatement:
`if ( Expression ) StatementNoShortIf else Statement`

IfThenElseStatementNoShortIf:
`if ( `*Expression*` ) `*StatementNoShortIf* `else` *StatementNoShortIf*

因此，语句在语法上被分为两类:那些可能以没有 else 子句的 if 语句结尾的语句(“简短的 if 语句”)和那些肯定没有 else 子句的语句。

只有绝对不以短 if 语句结尾的语句才可能作为直接子语句出现在具有 else 子句的 if 语句中的关键字 else 之前。

这个简单的规则防止了“悬空 else”问题。具有“不是短 if”限制的语句的执行行为与不具有“不是短 if”限制的同类语句的执行行为是相同的;这种区分纯粹是为了解决语法上的困难。

14.6 空语句

空语句什么都不做。

EmptyStatement:
;

空语句的执行总是正常完成。

14.7 标签语句

语句可以有标签前缀。

LabeledStatement:
Identifier : *Statement*

LabeledStatementNoShortIf:
Identifier : *StatementNoShortIf*

标识符被声明为立即包含的语句的标签。

与 C 和 C++不同，Java 编程语言没有 goto 语句；标识符语句标签用于标记语句内任何位置出现的 break 或 continue 语句（§14.15，§14.16）。

带标签语句的标签作用域是立即包含的语句。

如果标签语句的标签名称在标签作用域内用作另一个标签语句的标签，则是编译时错误。

使用同一标识符作为标签和包、类、接口、方法、字段、参数或局部变量的名称没有限制。使用标识符标记语句不会遮掩（§6.4.2）具有相同名称的包、类、接口、方法、字段、参数或局部变量。将标识符用作类、接口、方法、字段、局部变量或异常处理程序的参数（§14.20）不会遮掩具有相同名称的语句标签。

通过执行立即包含的语句来执行标记语句。

如果语句由标识符标记，并且包含的语句由于具有相同标识符的 break 而突然完成，则标记的语句正常完成。在所有其他突然完成语句的情况下，带标签的语句出于相同的原因突然完成。

例子 14.7-1. 标签和标识符

下面的代码取自类 String 及其方法 indexOf 的一个版本，其中标签最初称为 test。将标签更改为具有与局部变量 i 相同的名称不会在 i 的声明范围内遮掩标签。因此，代码是有效的。

```
class Test {
    char[] value;
    int offset, count;
    int indexOf(TestString str, int fromIndex) {
       char[] v1 = value, v2 = str.value;
```

```
            int max = offset + (count - str.count);
            int start = offset + ((fromIndex < 0) ? 0 : fromIndex);
        i:
            for (int i = start; i <= max; i++) {
                int n = str.count, j = i, k = str.offset;
                while (n-- != 0) {
                    if (v1[j++] != v2[k++])
                    continue i;
                }
                return i - offset;
            }
            return -1;
        }
    }
```

标识符 max 也可以用作语句标签；标签不会遮掩带标签语句中的局部变量 max。

14.8 表达式语句

某些类型的表达式可以通过在它们后面加上分号来作为语句使用。

ExpressionStatement:
StatementExpression ;

StatementExpression:
Assignment
PreIncrementExpression
PreDecrementExpression
PostIncrementExpression
PostDecrementExpression
MethodInvocation
ClassInstanceCreationExpression

表达式语句通过对表达式求值来执行;如果表达式有值，则丢弃该值。

当且仅当表达式的求值正常完成时，表达式语句的执行正常完成。

与 C 和 C++不同，Java 编程语言只允许特定形式的表达式作为表达式语句使用。例如，使用方法调用表达式是合法的(§15.12):

```
System.out.println("Hello world"); // OK
```

但使用括号表达式（§15.8.5）是不合法的:

```
(System.out.println("Hello world")); // illegal
```

请注意，Java 编程语言不允许“强制转换为 void”-void 不是一种类型-因此编写表达式语句的传统 C 技巧如下:

```
(void)... ;       // incorrect!
```

这不起作用。另一方面，Java 编程语言允许在表达式语句中使用所有最有用的表达式，并且不需要将方法调用用作表达式语句来调用 void 方法，因此几乎不需要这种技巧。如果需要技巧，可以使用赋值语句（§15.26）或局部变量声明语句（§14.4）。

14.9 if 语句

if 语句允许有条件地执行一条语句或有条件地选择两条语句，执行一条或另一条语句，但不能同时执行两条语句。

IfThenStatement:
`if` `(` *Expression* `)` *Statement*

IfThenElseStatement:
`if` `(` *Expression* `)` *StatementNoShortIf* `else` *Statement*

IfThenElseStatementNoShortIf:
`if` `(` *Expression* `)` *StatementNoShortIf* `else` *StatementNoShortIf*

Expression 的类型必须为 boolean 或 Boolean，否则会发生编译时错误。

14.9.1 if-then 语句

if-then 语句通过首先计算 Expression 来执行。如果结果为 Boolean 类型，则对其进行拆箱转换（§5.1.8）。

如果 Expression 的求值或随后的拆箱转换（如果有的话）由于某种原因突然完成，则 if-then 语句出于同样的原因突然完成。

否则，通过基于结果值进行选择继续执行：

- 如果该值为真，则执行包含的 Statement；当且仅当 Statement 的执行正常完成时，if-then 语句才正常完成。
- 如果该值为 false，则不采取进一步操作，if-then 语句正常完成。

14.9.2 if-then-else 语句

通过首先计算 Expression 来执行 if-then-else 语句。如果结果为 Boolean 类型，则对其进行拆箱转换（§5.1.8）。

如果 Expression 的求值或随后的拆箱转换（如果有的话）由于某种原因突然完成，那么 if-then-else 语句由于同样的原因突然完成。

否则，通过基于结果值进行选择继续执行：

- 如果该值为 true，则执行第一个包含的语句（else 关键字之前的语句）；当且仅当该语句的执行正常完成时，if-then-else 语句才正常完成。
- 如果该值为 false，则执行第二个包含的语句（else 关键字之后的语句）；当且仅当该

语句的执行正常完成时，if-then-else 语句才正常完成。

14.10 assert 语句

断言是包含布尔表达式的断言语句。断言要么被启用要么被禁用。 如果启用了断言，则断言的执行将导致布尔表达式的计算，如果表达式的计算结果为 false，则会报告错误。如果禁用了断言，则断言的执行不会产生任何影响。

AssertStatement:
  `assert` *Expression* `;`
  `assert` *Expression* `:` *Expression* `;`

为了简化表示，两种形式的断言语句的第一个 Expression 称为 Expression1。在断言语句的第二种形式中，第二个 Expression 称为 Expression2。

如果 Expression1 没有 boolean 或 Boolean 类型，则是编译时错误。

如果在断言语句的第二种形式中，Expression2 为 void（§15.1），则为编译时错误。

在其类或接口完成初始化后执行的断言语句，当且仅当主机系统已确定词法上包含该断言语句的顶级类或接口启用断言时，才启用。

顶级类或接口是否启用断言取决于不迟于最早的(i) 顶级类或接口的初始化, 以及 (ii) 嵌套在顶级类或接口中的任何类或接口的初始化。在确定顶级类或接口是否启用断言之后，不能更改它。

启用在其类或接口完成初始化之前执行的断言语句。

这一规定的动机是一个需要特殊对待的案件。回想一下，类的断言状态的设置不迟于它初始化的时间。在初始化之前执行方法或构造函数是可能的，尽管通常不可取。当类层次结构的静态初始化中包含循环时，就会发生这种情况，如下所示:

```
public class Foo {
    public static void main(String[] args) {
        Baz.testAsserts();
        // Will execute after Baz is initialized.
    }
}
class Bar { static {
        Baz.testAsserts();
        // Will execute before Baz is initialized!
    }
}
class Baz extends Bar {
    static void testAsserts() {
        boolean enabled = false;
        assert enabled = true;
        System.out.println("Asserts " +
      (enabled ? "enabled" : "disabled"));
    }
```

```
}
```

调用 Baz.testAsserts()导致 Baz 被初始化。在此之前，必须初始化 Bar。Bar 的静态初始化器再次调用 Baz.testAsserts()。由于当前线程已经在进行 Baz 的初始化，所以第二次调用会立即执行，虽然 Baz 没有初始化(§12.4.2)。

根据上面的规则，如果上面的程序在没有启用断言的情况下执行，它必须输出:

```
Asserts enabled
Asserts disabled
```

禁用的断言语句不执行任何操作。特别地，无论是 Expression1 还是 Expression2(如果存在的话)都不会被求值。禁用的断言语句的执行总是正常完成。

启用的断言语句是通过第一次计算 Expression1 来执行的。如果结果是 Boolean 类型，则进行拆箱转换(§5.1.8)。

如果 Expression1 的求值或随后的拆箱转换(如果有的话)由于某种原因突然结束，那么断言语句也会因为同样的原因突然结束。

否则，执行将继续根据 Expression1 的值进行选择:

- 如果值为 true，则不再执行任何操作，断言语句正常完成。
- 如果值为 false，执行行为取决于是否存在 Expression2:
  - 如果存在 Expression2，则对其求值。然后:

    > 如果求值因为某种原因突然结束，断言语句也会因为相同的原因突然结束。

    > 如果计算正常完成，则创建一个 AssertionError 实例，其“详细消息”是 Expression2 的结果值。然后:

    » 如果实例的创建由于某种原因突然结束，那么断言语句也会因为同样的原因突然结束。

    » 如果实例创建正常完成，断言语句会突然结束，抛出新创建的 AssertionError 对象。

  - 如果 Expression2 不存在，则创建一个没有“细节消息”的 AssertionError 实例。然后:

    > 如果实例的创建由于某种原因突然结束，那么断言语句也会因为同样的原因突然结束。

    > 如果实例创建正常完成，assert 语句会突然结束，抛出新创建的 AssertionError 对象。

    通常，在程序开发和测试期间启用断言检查，在部署时禁用断言检查，以提高性能。

    因为可以禁用断言，所以程序不能假定断言中包含的表达式会被求值。因此，这些布尔表达式

通常应该没有副作用。对这种布尔表达式的求值不应影响求值完成后可见的任何状态。包含在断言中的布尔表达式产生副作用并不违法，但这通常是不合适的，因为它可能会导致程序行为根据是否启用断言而变化。

有鉴于此，断言不应用于公共方法中的参数检查。参数检查通常是方法契约的一部分，无论启用还是禁用断言，都必须维护此契约。

使用断言进行参数检查的第二个问题是，错误的参数应导致适当的运行时异常(例如 IllegalArgumentException, ArrayIndexOutOfBoundsException, 或 者 NullPointerException)。断言失败不会引发适当的异常。同样，在公共方法上使用断言进行参数检查并不违法，但通常是不适当的。它的意图是永远不会捕获 AssertionError，但这是可能的，因此 try 语句的规则应该将出现在 try 块中的断言视为与当前对 throw 语句的处理类似。

14.11 switch 语句

switch 语句根据表达式的值将控制传递给多个语句或表达式之一。

SwitchStatement:
`switch` (*Expression*) *SwitchBlock*

Expression 称为选择器表达式。选择器表达式的类型必须是 char, byte, short, int, Character, Byte, Short, Integer, String 或枚举类型（§8.9），否则会发生编译时错误。

14.11.1 Switch 块

switch 语句和 switch 表达式（§15.28）的主体称为 switch 块。本小节介绍适用于所有 switch 块的一般规则，无论它们出现在 switch 语句或 switch 表达式中。其他小节提供了适用于 switch 语句中 switch（§14.11.2）或 switch 表达式中 switch 块的附加规则（§15.28.1）。

SwitchBlock:
{ *SwitchRule {SwitchRule}* }
{ *{SwitchBlockStatementGroup} {SwitchLabel* :*}* }

SwitchRule:
SwitchLabel -> *Expression* ;
SwitchLabel -> *Block*
SwitchLabel -> *ThrowStatement*

SwitchBlockStatementGroup:
SwitchLabel : *{SwitchLabel* :*} BlockStatements*

SwitchLabel:
`case` *CaseConstant {*, *CaseConstant}*
`default`

CaseConstant:
ConditionalExpression

switch 块可以由以下任一部分组成：

- switch 规则，使用->引入 switch 规则表达式、switch 规则块或 switch 规则抛出语句；或
- switch 标记的语句组，使用:引入 switch 标记的块语句。

每个 switch 规则和 switch 标签语句组都以 switch 标签开始，switch 标签可以是 case 标签，也可以是 default 标签。对于带有 switch 标签的语句组，允许使用多个 switch 标签。

一个 case 标签有一个或多个 case 常量。每个 case 常量必须是常量表达式（§15.29）或枚举常量的名称（§8.9.1），否则会发生编译时错误。

switch 标签及其 case 常量被称为与 switch 块相关联。与 switch 块关联的两个 case 常量可能没有相同的值，或者出现编译时错误。

如果以下两项均为真，则 switch 语句或 switch 表达式的 switch 块与选择器表达式 T 的类型兼容：

- 如果T不是枚举类型，则与switch块关联的每个case常量都与T的赋值兼容（§5.2）。
- 如果 T 是枚举类型，则与 switch 块关联的每个 case 常量都是类型为 T 的枚举常量。

switch 语句或 switch 表达式的 switch 块必须与选择器表达式的类型兼容，否则会发生编译时错误。

执行 switch 语句（§14.11.3）和计算 switch 表达式（§15.28.2）都需要确定 switch 标签是否与选择器表达式的值匹配。为了确定 switch 块中的 switch 标签是否与给定值匹配，将该值与和 switch 块关联的 case 常量进行比较。然后：

- 如果其中一个 case 常量等于该值，那么我们说包含 case 常量的 case 标签匹配。

  等式是根据==运算符（§15.21）定义的，除非值是字符串，在这种情况下，等式是根据 String 类的 equals 方法定义的。

- 如果没有 case 标签匹配，但有一个默认标签，那么我们说默认标签匹配。

  一个 case 标签可以包含几个 case 常量。如果标签的任何一个常量与选择器表达式的值匹配，则标签将与选择器表达式值匹配。例如，在如下代码中，如果枚举变量 day 是以下所示枚举常量之一，则 case 标签匹配。

  ```
  switch (day) {
      ...
      case SATURDAY, SUNDAY :
          System.out.println("It's the weekend!"); break; ...
  }
  ```

  null 不能作为 case 常量因为它不是常量表达式。即使允许 case null，这也是不可取的，因为这种情况下的代码永远不会被执行。这是因为，给定引用类型的选择器表达式（即，字符串或装箱原生类型或枚举类型），如果选择器表达式在运行时计算为 null，则会发生异常。根据 Java 编程语言设计者的判断，传播异常比没有 case 标签匹配或具有 default 标签匹配更好。

如果带有枚举类型选择器表达式的switch语句缺少默认default并且缺少一个或多个枚举常量的case标签，则鼓励（但不是必需的）Java 编译器提供警告。如果表达式的计算结果是缺少的常量之一，那么这样的switch语句将不起任何作用。

在 C 和 C++中，switch 语句的主体可以是语句，带有 case 标签的语句不必立即包含在该语句中。考虑简单的循环：

```
for (i = 0; i < n; ++i) foo();
```

其中 n 已知为正。在 C 或 C++中可以使用一种称为 Duff 的设备的技巧来展开循环，但这在 Java 编程语言中不是有效的代码：

```
int q = (n+7)/8;
switch (n%8) {
    case 0: do { foo();       // Great C hack, Tom,
    case 7:         foo();    // but it's not valid here.
    case 6:         foo();
    case 5:         foo();
    case 4:         foo();
    case 3:         foo();
    case 2:         foo();
    case 1:         foo();
          } while (--q > 0) ;
}
```

幸运的是，这一技巧似乎没有广为人知或被广泛使用。此外，现在不太需要它；这种代码转换完全属于最先进的优化编译器领域。

14.11.2 switch 语句的 switch 块

除了 switch 块的一般规则（§14.11.1），switch 语句中还有 switch 块的其他规则。即，switch 语句的 switch 块必须满足以下所有条件，否则将发生编译时错误：

- 与 switch 块关联的 default 标签不超过一个。
- switch 块中的每个 switch 规则表达式都是一个语句表达式（§14.8）。

  switch 语句与 switch 表达式的不同之处在于，哪些表达式可能出现在 switch 块中箭头（->）的右侧，即哪些表达式可以用作 switch 规则表达式。在 switch 语句中，只有语句表达式可用作 switch 规则表达式，但在 switch 表达式中，可以使用任何表达式（§15.28.1）。

14.11.3 执行 switch 语句

通过首先计算选择器表达式来执行 switch 语句。然后：

- 如果选择器表达式的求值突然完成，则由于相同的原因，整个 switch 语句也会突然完成。
- 否则，如果对选择器表达式求值的结果为 null，则会引发 NullPointerException，并且由于该原因，整个 switch 语句会突然完成。
- 否则，如果计算选择器表达式的结果为 Character, Byte, Short 或 Integer 类型，则对其进行拆箱转换（§5.1.8）。如果此转换突然完成，则由于相同的原因，整个 switch 语句也会突然完成。

如果选择器表达式的计算正常完成，且结果为非空，且随后的拆箱转换（如有）正常完成，则通过确定与 switch 块关联的 switch 标签是否与选择器表达式的值匹配来继续执行 switch 语句（§14.11.1）。然后：

- 如果没有 switch 标签匹配，整个 switch 语句正常完成。
- 如果一个 switch 标签匹配，然后，下列其中一项适用：
  - 如果是 switch 规则表达式的 switch 标签，则 switch 规则表达式必须是语句表达式（§14.11.2）。 如果计算的结果为值，则将其丢弃。
  - 如果是 switch 规则块的 switch 标签，则执行该块。如果该块正常完成，则 switch 语句正常完成。
  - 如果它是 switch 规则 throw 语句的 switch 标签，则执行 throw 语句。
  - 如果它是带有 switch 标签的语句组的 switch 标签，则按顺序执行 switch 块中跟随 switch 标签的所有语句。如果这些语句正常完成，则 switch 语句正常完成。
  - 否则，在 switch 块中，没有跟随匹配的 switch 标签的语句，switch 语句正常完成。

如果 switch 块中任何语句或表达式的执行突然完成，则按如下方式处理：

- 如果语句的执行由于没有标签的 break 而突然完成，则不采取进一步的操作，switch 语句正常完成。

  标签语句的一般规则（§14.7）处理因带标签的 break 而导致的突然完成。

- 如果语句或表达式的执行由于任何其他原因而突然完成，则 switch 语句由于相同的原因而突然结束。

  由 yield 语句引起的突然完成由 switch 表达式的一般规则处理(§15.28.2)。

  例子 14.11.3-1. switch 语句中的 fall-through

  当选择器表达式与 switch 规则的 switch 标签匹配时，只执行由 switch 标签引入的 switch 规则表达式或语句，而不执行其他操作。如果语句组中有一个 switch 标签，则执行 switch 块中在 switch 标签之后的所有块语句，包括出现在后续 switch 标签之后的语句。其结果是，就像在 C 和 C++中一样，语句的执行可以“通过标签进行”。

  例如，程序:

```
class TooMany {
    static void howMany(int k) {
       switch (k) {
            case 1:  System.out.print("one ");
            case 2:  System.out.print("too ");
            case 3:  System.out.println("many");
        }
    }
    public static void main(String[] args) {
        howMany(3);
```

```
            howMany(2);
            howMany(1);
        }
    }
```

包含一个 switch 块，其中每一个 case 的代码 fall-through 进入下一个 case 的代码。因此，程序打印:

```
many too many one too many
```

Fall through 可能是微小 bug 的原因。如果代码不能以这种方式逐项下降，那么可以使用 break 语句来指示何时应该转移控制，或者可以使用 switch 规则，如程序中所示:

```
class TwoMany {
    static void howMany(int k) {
       switch (k) {
            case 1:  System.out.println("one");
                     break; // exit the switch
            case 2:  System.out.println("two");
                     break; // exit the switch
            case 3:  System.out.println("many");
                     break; // not needed, but good style
        }
    }
    static void howManyAgain(int k) {
       switch (k) {
            case 1  -> System.out.println("one");
            case 2  -> System.out.println("two");
            case 3  -> System.out.println("many");
       }
    }
    public static void main(String[] args) {
       howMany(1);
       howMany(2);
       howMany(3);
       howManyAgain(1);
       howManyAgain(2);
       howManyAgain(3);
    }
}
```

程序打印:

```
one
two
many
one
two
many
```

14.12 while 语句

while 语句重复执行一个 Expression 和一个 Statement，直到 Expression 的值为 false。

WhileStatement:
`while` (*Expression*) *Statement*

WhileStatementNoShortIf:
`while` (*Expression*) *StatementNoShortIf*

Expression 的类型必须为 boolean 或 Boolean, 否则会发生编译时错误。

执行 while 语句时，首先对 Expression 求值。如果结果类型为 Boolean，它要进行拆箱转换(§5.1.8)。

如果 Expression 的求值或随后的拆箱转换(如果有的话)由于某种原因突然完成，while 语句也会由于相同的原因突然完成。

否则，执行将根据结果值继续进行选择:

- 如果值为 true，则执行包含的 Statement。那么就有一个选择:
  - 如果 Statement 的执行正常完成，那么整个 while 语句将再次执行，从重新计算 Expression 开始。
  - 如果 Statement 的执行突然结束，见§14.12.1。
- 如果 Expression 的值(可能是未装箱的)为 false，则不再执行任何操作，while 语句正常完成。

  如果 Expression 第一次被求值时(可能是未装箱的)的值为 false，则不执行 Statement。

14.12.1 while 语句的突然完成

突然完成所包含的 Statement 的处理方式如下:

- 如果由于没有标签的 break，Statement 突然结束，则不会采取进一步的操作，while 语句正常完成。
- 如果 Statement 的执行由于没有标签的 continue 而突然结束，那么整个 while 语句将再次执行。
- 如果 Statement 的执行突然结束，因为有一个标签为 L 的 continue，那么有一个选择:
  - 如果 while 语句有标签 L，那么整个 while 语句将再次执行。
  - 如果 while 语句没有标签 L，则 while 语句会突然结束，因为有一个带标签 L 的 continue。
- 如果由于任何其他原因，Statement 的执行突然结束，while 语句也会因为同样的原因突然结束。

  由于一个带标签的 break 而导致的突然完成的情况，由标记语句的一般规则来处理(§14.7)。

14.13 do 语句

do 语句重复执行一个 Statement 和一个 Expression，直到 Expression 的值为 false。

DoStatement:
`do` *Statement* `while (` *Expression* `) ;`

Expression 必须具有 boolean 或 Boolean 类型，否则将发生编译时错误。

do 语句是通过首先执行 Statement 来执行的。那么就有一个选择:

- 如果 Statement 正常执行完成，则对 Expression 进行求值。如果结果是 Boolean 类型，则进行拆箱转换(§5.1.8)。

  如果 Expression 的求值或随后的拆箱转换(如果有的话)由于某种原因突然完成，那么 do 语句也会出于同样的原因突然完成。

  否则，将根据结果值进行选择:

  - 如果值为 true，则再次执行整个 do 语句。
  - 如果值为 false，则不采取进一步的操作，do 语句正常完成。
- 如果 Statement 的执行突然完成，见§14.13.1。

  执行 do 语句总是至少执行一次所包含的 Statement。

14.13.1 do 语句的突然完成

包含 Statement 的突然完成的处理方式如下:

- 如果由于没有标签的 break，Statement 突然执行完毕，则不采取进一步的操作，do 语句正常完成。
- 如果 Statement 的执行突然完成，因为一个没有标签的 continue，那么 Expression 将被求值。然后就有了基于结果值的选择:
  - 如果值为 true，则再次执行整个 do 语句。
  - 如果值为 false，则不采取进一步的操作，do 语句正常完成。
- 如果 Statement 的执行突然完成，因为有一个标签为 L 的 continue，那么有一个选择:
  - 如果 do 语句有标签 L，则 Expression 被求值。那么就有一个选择:
    - ➢如果 Expression 的值为 true，则再次执行整个 do 语句。
    - ➢如果 Expression 的值为 false，则不再执行任何操作，do 语句正常完成。
  - 如果 do 语句没有标签 L，那么 do 语句会突然完成，因为有一个带标签 L 的

continue。

- 如果由于任何其他原因，Statement 的执行突然结束，那么 do 语句也会因为同样的原因突然结束。

由于一个带标签的 break 而导致的突然完成的情况，由标记语句的一般规则来处理(§14.7)。

例子 14.13-1. do 语句

以下代码是类 Integer 的 toHexString 方法的一个可能实现：

```
public static String toHexString(int i) {
   StringBuffer buf = new StringBuffer(8);
   do {
        buf.append(Character.forDigit(i & 0xF, 16));
        i >>>= 4;
    } while (i != 0);
   return buf.reverse().toString();
}
```

因为必须生成至少一个数字，所以 do 语句是一种适当的控制结构。

14.14 for 语句

for 语句有两种形式：

- 基础的 for 语句
- 增强的 for 语句

ForStatement:
BasicForStatement
EnhancedForStatement

ForStatementNoShortIf:
BasicForStatementNoShortIf
EnhancedForStatementNoShortIf

14.14.1 基础的 for 语句

基本的 for 语句先执行一些初始化代码，然后重复执行 Expression、Statement 和一些更新代码，直到 Expression 的值为 false。

BasicForStatement:
`for` (*[ForInit]* ; *[Expression]* ; *[ForUpdate]*) *Statement*

BasicForStatementNoShortIf:
`for` (*[ForInit]* ; *[Expression]* ; *[ForUpdate]*) *StatementNoShortIf*

ForInit:

StatementExpressionList
LocalVariableDeclaration

ForUpdate:
StatementExpressionList

StatementExpressionList:
StatementExpression {, StatementExpression}

Expression 的类型必须是 boolean 或 Boolean，否则将发生编译时错误。

在基本 for 语句的 ForInit 部分声明的局部变量的作用域和遮蔽在§6.3 和§6.4 中有明确的说明。

从嵌套类、接口或 lambda 表达式中引用基本 for 语句的 ForInit 部分声明的局部变量是受限制的，如§6.5.6.1 所述。

14.14.1.1 for 语句的初始化

执行 for 语句时，首先执行 ForInit 代码:

- 如果 ForInit 代码是一个语句表达式列表(§14.8)，则表达式的求值顺序为从左到右;如果有，则丢弃它们的值。

  如果由于某种原因，任何表达式的求值突然完成，那么 for 语句也会因为同样的原因突然完成; 突然完成的语句表达式右边的任何 ForInit 语句表达式都不计算。

- 如果 ForInit 代码是一个局部变量声明(§14.4)，它就像一个出现在块中的局部变量声明语句一样被执行(§14.4.2)。

  如果局部变量声明的执行因为任何原因突然结束，for 语句也会因为同样的原因突然结束。

- 如果不存在 ForInit 部分，则不采取任何操作。

14.14.1.2 for 语句的迭代

接下来，执行一个 for 迭代步骤，如下所示:

- 如果 Expression 存在，则对其求值。如果结果是 Boolean 类型，则进行拆箱转换(§5.1.8)。

  如果 Expression 的求值或随后的拆箱转换(如果有的话)突然完成，for 语句也会因为同样的原因突然完成。

  否则，根据表达式的存在或不存在以及表达式存在时的结果值进行选择；请参阅下一个项目符号。

- 如果 Expression 不存在，或者它存在，并且其求值(包括任何可能的拆箱)得到的值为真，则执行所包含的 Statement。然后就有了一个选择:

- 如果 Statement 的执行正常完成，则按顺序执行以下两个步骤：

  1. 首先，如果存在 ForUpdate 部分，则按从左到右的顺序计算表达式；它们的值，如果有，则丢弃。如果任何表达式的求值由于某种原因而突然完成，则 for 语句也会因同样的原因突然完成；不计算突然完成的语句右侧的任何 ForUpdate 语句表达式。

     如果 ForUpdate 部分不存在，则不会执行任何操作。

  2. 其次，执行另一个 for 迭代步骤。

- 如果 Statement 执行突然完成，请参见第 14.14.1.3 节。

- 如果 Expression 存在，并且其求值(包括任何可能的拆箱)得到的值为 false，则不会采取进一步操作，并且 for 语句将正常完成。

  如果第一次计算 Expression 的值(可能未装箱)为 false，则不会执行 Statement。

如果 Expression 不存在，则 for 语句正常结束的唯一方法是使用 break 语句。

14.14.1.3 for 语句的突然完成

对包含的 Statement 的突然完成按以下方式处理：

- 如果由于没有标签的 break 而导致 Statement 的执行突然完成，则不会采取进一步的操作，并且 for 语句将正常完成。

- 如果由于没有标签的 continue 语句而导致 Statement 执行突然完成，则按顺序执行以下两个步骤：

  1. 首先，如果存在 ForUpdate 部分，则按从左到右的顺序计算表达式；它们的值，如果有，则丢弃。

     如果 ForUpdate 部分不存在，则不会执行任何操作。

  2. 其次，执行另一个 for 迭代步骤。

- 如果由于标签为 L 的 continue 而突然完成 Statement 的执行，则有一个选择：

  - 如果 for 语句具有标签 L，则按顺序执行以下两个步骤：

    1. 首先，如果存在 ForUpdate 部分，则按从左到右的顺序计算表达式；它们的值，如果有，则丢弃。

       如果 ForUpdate 不存在，则不会执行任何操作。

    2. 其次，执行另一个 for 迭代步骤。

  - 如果 for 语句没有标签 L，则 for 语句会因为带有标签 L 的 continue 而突然结束。

- 如果 Statement 的执行因任何其他原因而突然完成，则 for 语句也会因同样的原因而突然完成。

请注意，由于带标签的 break 而突然完成的情况由标签语句的一般规则处理(§14.7)。

14.14.2 增强的 for 语句

增强的 for 语句的形式为：

EnhancedForStatement:
`for` `(` *LocalVariableDeclaration* `:` *Expression* `)`
Statement

EnhancedForStatementNoShortIf:
`for` `(` *LocalVariableDeclaration* `:` *Expression* `)`
StatementNoShortIf

为方便起见，此处显示了§4.3、§8.3、§8.4.1 和§14.4 中的以下内容：

LocalVariableDeclaration:
{VariableModifier} LocalVariableType VariableDeclaratorList

VariableModifier:
Annotation
`final`

LocalVariableType:
UnannType `var`

VariableDeclaratorList:
VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
VariableDeclaratorId [= VariableInitializer]

VariableDeclaratorId:
Identifier [Dims]

Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

Expression 的类型必须是数组类型（§10.1）或原始类型 Iterable 的子类型，否则会发生编译时错误。

增强型 for 语句的头声明了一个局部变量，其名称是 VariableDeclaratorId 给出的标识符。当执行增强型 for 语句时，在循环的每次迭代中，局部变量被初始化为可迭代的连续元素或表达式生成的数组。

在增强的 for 语句的头中声明的局部变量的规则在§14.4 中指定，而不考虑该部分中当 LocalVariableType 为 var 时适用的任何规则。此外，以下所有条件都必须为真，否则会发生编译时错误：

- VariableDeclaratorList 由一个 VariableDeclarator 组成。

- VariableDeclarator 没有初始化器。
- 如果 LocalVariableType 为 var，则 VariableDeclaratorID 没有括号对。

在增强的 for 语句的头中声明的局部变量的作用域和遮蔽在§6.3 和§6.4 中指定。

根据§6.5.6.1 的规定，对嵌套类或接口或 lambda 表达式中的局部变量的引用受到限制。

增强型 for 语句头中声明的局部变量的类型 T 确定如下：

- 如果 LocalVariableType 是 UnannType, 并且在 UnnanType 或 VariableDeclaratorId 中没有出现括号对，那么 T 是 UnannType 表示的类型。
- 如果 LocalVariableType 是 UnannType，并且括号对出现在 UnnanType 或 VariableDeclaratorId 中，则 T 由§10.2 规定。
- 如果 LocalVariableType 是 var, 然后，让 R 从 Expression 的类型中派生出,如下：
  - 如果表达式具有数组类型，则 r 是该数组类型的组件类型。
  - 否则，如果 Expression 的类型是 Iterable<X>的子类型，那么对于某个类型 X，R 就是 X。
  - 否则，Expression 的类型是原始类型 Iterable 的子类型，并且 R 是 Object。

    T 是 R 相对于 R 提到的所有合成类型变量的向上投影(§4.10.5)。

增强的 for 语句的确切含义是通过转换为基本 for 语句来给出的，如下所示：

- 如果 Expression 的类型是 Iterable 的子类型，则基本的 for 语句具有以下形式：

```
for (I #i = Expression.iterator(); #i.hasNext(); ){
  {VariableModifier} T Identifier = (TargetType) #i.next();
  Statement
}
```

其中:

- 如果对于某个类型参数 X，Expression 的类型是 Iterable<X>的子类型，那么 I 就是类型 java.util.Iterator<X>。否则, I 是原始类型 java.util.Iterator。
- #i 是一个自动生成的标识符，它不同于在增强的 for 语句发生时在作用域(§6.3)内的任何其他标识符(自动生成或以其他方式生成的)。
- {VariableModifier}与增强的 for 语句的头中给出的一样。
- T 是上面确定的局部变量的类型。
- 如果 T 是一个引用类型, 那么 TargetType 就是 T。否则, TargetType 是 I 的类型参数的捕获转换的上限(§5.1.10)，或者如果 I 是原始类型，则 TargetType 是 Object。

- 否则，Expression 必须具有数组类型 S[]，并且基本的 for 语句具有以下形式：

```
S[] #a = Expression;
L1: L2: ... Lm:
for (int #i = 0; #i < #a.length; #i++){
  {VariableModifier} T Identifier = #a[#i];
  Statement
}
```

其中:

- L_1 ... L_m是紧接在增强的 for 语句之前的标签序列(可能为空)。

- #a 和#i 是自动生成的标识符，它们不同于在增强的 for 语句发生时在作用域中的任何其他标识符(自动生成或以其他方式生成)。

- {VariableModifier}与增强的 for 语句的头中给出的一样。

- T 是上面确定的局部变量的类型。

例如，此代码:

```
List<? extends Integer> l = ...
for (float i : l) ...
```

将被翻译为:

```
for (Iterator<Integer> #i = l.iterator(); #i.hasNext(); ) {
  float #i0 = (Integer)#i.next();
  …
```


例子 14.14.1.增强 for 语句和数组

下面的程序计算整数数组的和，它显示了增强的 for 是如何处理数组的:

```
int sum(int[] a) {
    int sum = 0;
    for (int i : a) sum += i;
    return sum;
}
```

例子 14.14.2.增强 for 语句和拆箱转换

以下程序将增强的 for 语句与自动取消装箱相结合，以将直方图转换为频率表:

```
Map<String, Integer> histogram = ...;
double total = 0;
for (int i : histogram.values())
    total += i;
for (Map.Entry<String, Integer> e : histogram.entrySet())
    System.out.println(e.getKey() + " " + e.getValue() / total);
}
```

14.15 break 语句

break 语句将控制权从封闭语句中转移出来。

BreakStatement:
`break` *[Identifier]* `;`

break 语句有两种:

- 没有标签的 break 语句
- 带有标签标识符的 break 语句

没有标签的 break 语句试图将控制转移到最里面的封闭 switch、while、do 或 for 语句； 这个封闭的语句称为 break 目标，然后立即正常完成。

带有标签标识符的 break 语句试图将控制转移到与其标签具有相同标识符的封闭标签语句(§14.7)；这个封闭的语句称为 break 目标，然后立即正常完成。在这种情况下，break 目标不需要是 switch、while、do 或 for 语句。

如果 break 语句没有 break 目标，则为编译时错误。

如果 break 目标包含包含 break 语句的任何方法、构造函数、实例初始化器、静态初始化器、lambda 表达式或 switch 表达式，则为编译时错误。也就是说，不存在非局部跳转。

没有标签的 break 语句的执行总是突然完成，原因是没有标签的 break。

使用标签标识符执行 break 语句总是突然完成，原因是使用标签标识符的 break。

由此可见，break 语句总是突然结束。

前面的描述说“尝试转移控制”，而不仅仅是“转移控制”，因为如果 break 目标内有任何 try 语句(§14.20)，其 try 块或 catch 子句包含 break 语句，则在将控制权转移到 break 目标之前，这些 try 语句的所有 finally 子句将按从内到外的顺序执行。finally 子句的突然完成可能会中断 break 语句启动的控制转移。

例子 14.15.1. break 语句

在下面的示例中，数学图由数组的数组表示。图由一组节点和一组边组成；每条边都是一个箭头，它从某个节点指向另一个节点，或从一个节点指向它自己。在该示例中，假设没有冗余边；即，对于任意两个节点 P 和 Q，其中 Q 可以与 P 相同，从 P 到 Q 至多有一条边。

节点由整数表示，对于每个 i 和 j，存在从节点 i 到节点 edges[i][j]的边，数组引用 edges [i][j]不会引发 ArrayIndexOutOfBoundsException。

在给定整数 i 和 j 的情况下，方法 loseEdges 的任务是通过复制给定图但省略从结点 i 到结点 j 的边(如果有)和从结点 j 到结点 i 的边(如果有)来构造新图:

```
class Graph {
    int[][] edges;
    public Graph(int[][] edges) { this.edges = edges; }
    public Graph loseEdges(int i, int j) {
```

```
        int n = edges.length;
        int[][] newedges = new int[n][];
        for (int k = 0; k < n; ++k) {
edgelist:
{
            int z;
search:
{
            if (k == i) {
                for (z = 0; z < edges[k].length; ++z) {
                  if (edges[k][z] == j) break search;
                }
            } else if (k == j) {
                for (z = 0; z < edges[k].length; ++z) {
                  if (edges[k][z] == i) break search;
                }
            }

            // No edge to be deleted; share this list.
            newedges[k] = edges[k];
            break edgelist;
} //search

            // Copy the list, omitting the edge at position z.
            int m = edges[k].length - 1;
            int[] ne = new int[m];
            System.arraycopy(edges[k], 0, ne, 0, z);
            System.arraycopy(edges[k], z+1, ne, z, m-z);
            newedges[k] = ne;
} //edgelist
        }
        return new Graph(newedges);
    }
}
```

注意两个语句标签的使用，edgelist 和 search，以及 break 语句的使用。这允许在两个单独的测试之间共享复制列表(省略一个边)的代码，即从节点 i 到节点 j 的边的测试，以及从节点 j 到节点 i 的边的测试。

14.16 continue 语句

continue 语句只能出现在 while、do 或 for 语句中；这三种语句称为迭代语句。控制传递到迭代语句的循环延续点。

ContinueStatement:
`continue` *[Identifier]* `;`

有两种 continue 语句：

- 不带标签的 continue 语句。
- 带有标签标识符的 continue 语句。

没有标签的 continue 语句试图将控制转移到最里面的封闭的 while、do 或 for 语句；这个封闭语句称为 continue 目标，然后立即结束当前迭代并开始新的迭代。

带有标签标识符的 continue 语句试图将控制转移到与其标签具有相同标识符的封闭标签语句(§14.7)；这个封闭语句称为 continue 目标，然后立即结束当前迭代并开始新的迭代。 在这种情况下，continue 目标必须是 while、do 或 for 语句，否则会发生编译时错误。

如果 continue 语句没有 continue 目标，则它是编译时错误。

如果 continue 目标包含任何包含 continue 语句的方法、构造函数、实例初始化器、静态初始化器、lambda 表达式或 switch 表达式，则为编译时错误。也就是说，没有非本地跳跃。

没有标签的 continue 语句的执行总是突然完成，原因是没有标签的 continue。

带有标签标识符的 continue 语句的执行总是突然完成，原因是带有标签标识符的 continue。

由此可见，continue 语句总是突然结束。

有关因 continue 而突然终止的处理的讨论，请参阅 while 语句(§14.12)、do 语句(§14.13)和 for 语句(§14.14)的说明。

前面的描述说“尝试转移控制”，而不仅仅是“转移控制”，因为如果在 continue 目标中有任何 try 语句(§14.20)，其 try 块或 catch 子句包含 continue 语句，则在将控制权转移到 continue 目标之前，这些 try 语句的任何 finally 子句将按从内到外的顺序执行。finally 子句的突然完成可能会中断由 continue 语句启动的控制转移。

例子 14.16-1. `continue` **语句**

在§14.15 的 Graph 类中，一个 break 语句用来完成最外层 for 循环的整个循环体的执行。这个 break 语句可以被 continue 语句替换，如果 for 循环本身打了标签的话：

```
class Graph {
    int[][] edges;
    public Graph(int[][] edges) { this.edges = edges; }

    public Graph loseEdges(int i, int j) {
        int n = edges.length;
        int[][] newedges = new int[n][];
edgelists:
        for (int k = 0; k < n; ++k) {
          int z;
search:
{
          if (k == i) {
               for (z = 0; z < edges[k].length; ++z) {
                 if (edges[k][z] == j) break search;
               }
           } else if (k == j) {
               for (z = 0; z < edges[k].length; ++z) {
                 if (edges[k][z] == i) break search;
               }
           }
```

```
                // No edge to be deleted; share this list.
                newedges[k] = edges[k];
                continue edgelists;
    } //search

                // Copy the list, omitting the edge at position z.
                int m = edges[k].length - 1;
                int[] ne = new int[m];
                System.arraycopy(edges[k], 0, ne, 0, z);
                System.arraycopy(edges[k], z+1, ne, z, m-z);
                newedges[k] = ne;
            } //edgelists
            return new Graph(newedges);
        }
    }
```

使用哪一个，如果有的话，很大程度上是一个编程风格的问题。

14.17 return 语句

return 语句将控制权返回给方法的调用者(§8.4，§15.12)或构造器(§8.8，§15.9)。

ReturnStatement:
`return` *[Expression]* `;`

有两种返回语句:

- 不带值的返回语句。
- 带有 Expression 值的返回语句。

return 语句试图将控制权转移给最内部封闭的构造函数、方法或 lambda 表达式的调用者;这个封闭的声明或表达式称为返回目标。在带有 Expression 值的 return 语句的情况下，Expression 的值成为调用的值。

如果返回语句没有返回目标，则为编译时错误。

如果返回目标包含(i)包含 return 语句的实例或静态初始化器，或(ii)包含 return 语句的 switch 表达式，则它是编译时错误。

如果没有值的 return 语句的返回目标是一个方法，并且该方法没有被声明为 void，则为编译时错误。

如果带有 Expression 值的 return 语句的返回目标是构造函数或声明为 void 的方法，则为编译时错误。

如果带有 Expression 值的 return 语句的返回目标是声明了返回类型 T 的方法，并且 Expression 的类型与 T 不可赋值兼容(§5.2)，则为编译时错误。

没有值的 return 语句的执行总是突然完成，原因是没有值的 return。

执行带有 Expression 值的 return 语句首先计算 Expression。如果由于某种原因，Expression 的计算突然结束，则 return 语句也会因此突然结束。如果 Expression 的计算正常完成，并产生值 V，则 Return 语句突然结束，原因是带返回值 V 的 return。

因此，可以看出，返回语句总是突然结束。

前面的描述说"尝试转移控制"，而不仅仅是"转移控制"，因为如果在方法或构造函数中有任何 try 语句(§14.20)，其 try 块或 catch 子句包含 return 语句，那么在将控制权转移到方法或构造函数的调用者之前，将按从内到外的顺序执行那些 try 语句的所有 finally 子句。finally 子句的突然完成可能会中断由 return 语句启动的控制转移。

14.18 throw 语句

throw 语句导致抛出异常(§11)。结果是立即转移控制(§11.3)，这可能会退出多个语句和多个构造函数、实例初始化器、静态初始化器和字段初始化器求值，以及方法调用，直到找到捕获抛出的值的 try 语句(§14.20)。如果找不到这样的 try 语句，则在调用线程所属的线程组的 uncaughtException 方法后，终止执行抛出的线程(§17(线程和锁))的执行(§11.3)。

ThrowStatement:
`throw` *Expression* `;`

throw 语句中的 Expression 必须表示可赋给 Throwable 类型(§5.2)的引用类型的变量或值，或者表示空引用，否则会发生编译时错误。

Expression 的引用类型将始终是类类型(因为没有接口类型可分配给 Throwable)，它不是参数化的(因为 Throwable 的子类不能是泛型的(§8.1.2))。

以下三个条件中必须至少有一个为真，否则会发生编译时错误：

- Expression 的类型是未检查的异常类(§11.1.1)或 null 类型(§4.1)。
- throw 语句包含在 try 语句的 try 块中(§14.20)，而 try 语句不能抛出 Expression 类型的异常。(在这种情况下我们说抛出的值被 try 语句捕获。)
- throw 语句包含在方法或构造函数声明中，并且 Expression 的类型可分配给声明的它 throws 子句(§8.4.6、§8.8.5)中列出的至少一种类型(§5.2)。

throw 语句可以抛出的异常类型在§11.2.2 中指定。

throw 语句首先计算 Expression 的值。然后：

- 如果 Expression 的计算由于某种原因而突然完成，则 throw 也会因此而突然完成。
- 如果 Expression 的计算正常完成，并生成一个非空值 V，则 throw 语句突然结束，原因是带有值 V 的 throw。
- 如果 Expression 的求值正常完成，并产生 null 值，则创建并抛出类 NullPointerException 的实例 V'，而不是 null 值。然后，throw 语句突然结束，原因是

带有值为 V'的 throw。

因此，可以看出，throw 语句总是突然结束。

如果有任何封闭的 try 语句(§14.20)，其 try 块包含 throw 语句，则在向外转移控制时执行这些 try 语句的所有 finally 子句，直到捕捉到抛出的值。请注意，finally 子句的突然完成可能会中断由 throw 语句启动的控制转移。

如果 throw 语句包含在方法声明或 lambda 表达式中，但它的值没有被包含它的某个 try 语句捕获，那么方法的调用就会因为抛出而突然完成。

如果 throw 语句包含在构造函数声明中，但它的值没有被包含它的某个 try 语句捕获，则调用构造函数的类实例创建表达式将因抛出而突然完成(§15.9.4)。

如果 throw 语句包含在静态初始化器中(§8.7)，则编译时检查(§11.2.3)可确保其值始终是未检查的异常，或者其值始终被包含它的某个 try 语句捕获如果在运行时，尽管进行了此检查，但包含 throw 语句的某些 try 语句不会捕获该值，那么，如果该值是 Error 类的实例或其子类之一，则重新抛出该值；否则，它被包装在 ExceptionInInitializerError 对象中，然后抛出该对象(§12.4.2)。

如果 throw 语句包含在实例初始化器中(§8.6)，则编译时检查(§11.2.3)确保要么它的值始终是未检查的异常，要么它的值总是被包含它的某个 try 语句捕获，或者被抛出的异常的类型(或它的一个超类)出现在类的每个构造函数的 throws 子句中。

按照惯例，用户声明的可抛出类型通常应该声明为类 Exception 的子类，后者是 Throwable 类的子类(§11.1.1)。

14.19 synchronized 语句

同步语句代表执行线程获取互斥锁(§17.1)，执行一个块，然后释放锁。当执行线程拥有该锁时，任何其他线程都不能获取该锁。

SynchronizedStatement:
`synchronized` `(` *Expression* `)` *Block*

Expression 的类型必须是引用类型，否则会发生编译时错误。

通过首先计算 Expression 来执行同步语句。然后：

- 如果 Expression 的计算由于某种原因而突然完成，则同步语句也会因同样的原因突然完成。
- 否则，如果 Expression 的值为空，则会引发 NullPointerException 异常。
- 否则，让 Expression 的非空值为 V。执行线程锁定与 V 关联的监视器。然后执行 Block，然后有一个选择：

- 如果 Block 的执行正常完成，则监视器解锁，同步语句正常完成。
- 如果 Block 的执行由于任何原因而突然完成，则监视器被解锁，并且同步的语句出于相同的原因突然完成。

同步语句获取的锁与同步方法隐式获取的锁相同(§8.4.3.6)。单个线程可以多次获取一个锁。

获取与对象相关联的锁本身并不会阻止其他线程访问该对象的字段或调用该对象上的非同步方法。其他线程也可以以常规方式使用同步方法或同步语句来实现互斥。

例子 14.19-1. `synchronized` **语句**

```
class Test {
     public static void main(String[] args) {
        Test t = new Test();
        synchronized(t) {
            synchronized(t) {
                System.out.println("made it!");
            }
        }
     }
}
```

此程序生成以下输出：

```
made it!
```

请注意，如果不允许单个线程多次锁定一个监视器，则该程序将死锁。

14.20 try 语句

一条 try 语句执行一个块。如果抛出一个值，并且 try 语句具有一个或多个可以捕获该值的 catch 子句，则控制权将转移到第一个这样的 catch 子句。 如果 try 语句有 finally 子句，则无论 try 块是正常完成还是突然完成，也不管 catch 子句是否首先获得控制，都会执行另一个代码块。

TryStatement:
`try` *Block Catches*
`try` *Block [Catches] Finally*
TryWithResourcesStatement

Catches:
CatchClause {CatchClause}

CatchClause:
`catch` (*CatchFormalParameter*) *Block*

CatchFormalParameter:
{VariableModifier} CatchType VariableDeclaratorId

CatchType:
UnannClassType {`|` *ClassType*}

Finally:
`finally` *Block*

有关 UnannClassType 的信息，请参阅§8.3。为方便起见，此处列出了 §4.3, §8.3 和§8.4.1 的以下内容：

VariableModifier:
Annotation
`final`

VariableDeclaratorId:
Identifier [*Dims*]

Dims:
{*Annotation*} `[ ]` {{*Annotation*} `[ ]`}

紧跟在关键字 try 之后的 Block 称为 try 语句的 try 块。

紧跟在关键字 finally 之后的 Block 称为 try 语句的 finally 块。

try 语句可能有 catch 子句，也称为异常处理程序。

catch 子句只声明一个参数，该参数称为异常参数。

如果 final 作为异常参数声明的修饰符多次出现，则为编译时错误。

§6.3 和§6.4 规定了异常参数的作用域和遮蔽。

对嵌套类或接口或 lambda 表达式的异常参数的引用是受限制的，如§6.5.6.1 中规定的。

异常参数可以将其类型表示为单个类类型或两个或多个类类型的联合(称为替代类)。联合的备选方案在语法上由|分隔。

异常参数表示为单个类类型的 catch 子句称为单捕获子句。

异常参数表示为类型联合的 catch 子句称为多捕获子句。

异常参数的类型表示中使用的每个类类型必须是 Throwable 类或 Throwable 的子类，否则将发生编译时错误。

如果类型变量用于表示异常参数的类型，则这是编译时错误。

如果类型的联合包含两个替换 D_i 和 $D_j (i \neq j)$，其中 D_i 是 D_j 的子类型(§4.10.2)，则这是编译时错误。

用单个类类型表示其类型的异常参数的声明类型是该类类型。

异常参数的声明类型是 lub(D_1, D_2, ..., D_n)，它将其类型表示为与替换项 $D_1 \mid D_2 \mid \ldots \mid D_n$ 的并集。

如果多捕获子句的异常形参没有显式声明为 final，则隐式声明其为 final。

如果在 catch 子句的主体内对隐式或显式声明为 final 的异常参数赋值，则为编译时错误。

单捕获子句的异常参数永远不会隐式声明为 final，但可以显式声明为 final 或实际上为 final(§4.12.4)。

隐式的 final 异常参数因其声明而为 final，而实际上的 final 异常参数因其使用方式而为 final。多捕获子句的异常参数隐式声明为 final，因此永远不会作为赋值操作符的左操作数出现，但它实际上不被认为是 final 的。

如果异常参数是实际上的 final(在单捕获子句中)或隐式的 final(在多捕获子句中)，那么在其声明中添加显式的 final 修饰符将不会引入任何编译时错误。另一方面，如果单捕获子句的异常参数显式声明为 final，那么删除 final 修饰符可能会引入编译时错误，因为异常参数现在被认为是实际的 final，不再被 catch 子句体中的匿名和局部类声明引用。如果没有编译时错误，则可以进一步更改程序，以便在 catch 子句体中重新赋值异常参数，从而不再被认为是实际的 final。

try 语句可以抛出的异常类型在§11.2.2 中指定。

try 语句的 try 块抛出的异常与 try 语句的 catch 子句(如果有的话)捕获的异常之间的关系见§11.2.3。

异常处理程序按从左到右的顺序考虑: 最早的可能的 catch 子句接受异常，接收抛出的异常对象作为其参数，如§11.3 所述。

一个多捕获子句可以被认为是一个单捕获子句的序列。也就是说，其中异常参数的类型被表示为并集 $D_1|D_2|...|D_n$ 的 catch 子句等价于 n 个 catch 子句的序列，其中异常参数的类型分别是类类型 D_1, D_2, ..., D_n。在 n 个 catch 子句的每个块中，声明的异常参数类型是 lub(D_1, D_2, ..., D_n)。例如，以下代码:

```
try {
     ... throws ReflectiveOperationException ...
}
catch (ClassNotFoundException | IllegalAccessException ex) {
     ... body ...
}
```

在语义上等价于以下代码:

```
try {
   ... throws ReflectiveOperationException ...
}
catch (final ClassNotFoundException ex1) {
    final ReflectiveOperationException ex = ex1;
    ... body ...
}
catch (final IllegalAccessException ex2) {
    final ReflectiveOperationException ex = ex2;
    ... body ...
}
```

其中，具有两个可选方案的多捕获子句已被翻译为两个单捕获子句，每个可选方案一个。Java 编译器既不需要也不建议通过以这种方式复制代码来编译多捕获子句，因为可以在 class 文件中表示多捕捉子句不用复制。

finally 子句确保在 try 块和可能执行的任何 catch 块之后执行 finally 块，无论控制如何离开 try 块或 catch 块。finally 块的处理相当复杂，因此分别描述了带有和不带有 finally 块的 try 语句的两种情况(§14.20.1、§14.20.2)。

如果 try 语句是 try-with-resources 语句，则允许它省略 catch 子句和 finally 子句(§14.20.3)。

14.20.1 try-catch 的执行

通过首先执行 try 块来执行不带 finally 块的 try 语句。然后就有了一个选择：

- 如果 try 块的执行正常完成，则不会采取进一步操作，而 try 语句将正常完成。
- 如果由于抛出值 V 而导致 try 块的执行突然完成，则有一个选择：
  - 如果 v 的运行时类型与 try 语句的任何 catch 子句的可捕获异常类赋值兼容(§5.2)，则选择第一个(最左边)这样的 catch 子句。将值 V 赋给所选 catch 子句的参数，并执行该 catch 子句的 Block，然后有一个选择：
    - > 如果该块正常完成，则 try 语句也正常完成。
    - > 如果该块由于任何原因而突然完成，则 try 语句也会因同样的原因突然完成。
  - 如果 v 的运行时类型与 try 语句的任何 catch 子句的可捕获异常类赋值不兼容，则 try 语句会因抛出值 v 而突然结束。
- 如果由于任何其他原因，try 块的执行突然完成，则由于同样的原因，try 语句也会突然完成。

例子 14.20.1-1. 捕获异常

```
class BlewIt extends Exception {
    BlewIt() { }
    BlewIt(String s) { super(s); }
}
class Test {
    static void blowUp() throws BlewIt { throw new BlewIt(); }

    public static void main(String[] args) {
       try {
            blowUp();
       } catch (RuntimeException r) {
            System.out.println("Caught RuntimeException");
       } catch (BlewIt b) {
            System.out.println("Caught BlewIt");
       }
    }
}
```

在这里，方法 blowUp 抛出异常 BlewIt。main 主体中的 try-catch 语句有两个 catch 子句。异常的运行时类型是 BlewIt，它不能赋值给类型为 RuntimeException 的变量，但可以赋值给类型为 BlewIt 的变量，因此该示例的输出为：

```
Caught BlewIt
```

14.20.2 try-finally 和 try-catch-finally 的执行

通过首先执行 try 块来执行带有 finally 块的 try 语句。然后就有了一个选择:

- 如果 try 块的执行正常完成，则执行 finally 块，然后有一个选择:
  - 如果 finally 块正常完成，则 try 语句也正常完成。
  - 如果 finally 块因为 S 原因突然结束，那么 try 语句因为 S 原因突然结束。
- 如果 try 块的执行突然结束，因为抛出了值 V，那么有一个选择:
  - 如果 v 的运行时类型与 try 语句的任何 catch 子句的可捕获异常类赋值兼容，那么将选择第一个(最左边)这样的 catch 子句。将值 v 赋给所选 catch 子句的参数，并执行该 catch 子句的 Block。那么就有一个选择:
    > 如果 catch 块正常完成，则执行 finally 块。那么就有一个选择:
    » 如果 finally 块正常完成，那么 try 语句也会正常完成。
    » 如果 finally 块因为某种原因突然结束，那么 try 语句也会因为同样的原因突然结束。
    > 如果 catch 块由于 R 原因突然完成，则执行 finally 块。那么就有一个选择:
    » 如果 finally 块正常完成，那么 try 语句由于 R 原因突然完成。
    » 如果 finally 块因为 S 原因突然结束，那么 try 语句因为 S 原因突然结束(并且原因 R 被丢弃)。
  - 如果 v 的运行时类型与 try 语句的任何 catch 子句的可捕获异常类不赋值兼容，则执行 finally 块。然后有一个选择:
    > 如果 finally 块正常完成，那么 try 语句由于抛出值 v 而突然完成。
    > 如果 finally 块由于原因 S 而突然完成，那么 try 语句由于原因 S 突然完成（并且丢弃并忘记了值 v 的抛出）。
- 如果 try 块的执行由于任何其他原因 R 突然完成，则执行 finally 块，然后有一个选择:
  - 如果 finally 块正常完成，那么 try 语句由于原因 R 而突然完成。
  - 如果 finally 块由于原因 S 突然完成，那么 try 语句由于原因 S 而突然完成（并且原因 R 被丢弃）。

例子 14.20.2-1. 用 finally 处理未捕获异常

```
class BlewIt extends Exception {
    BlewIt() { }
    BlewIt(String s) { super(s); }
}
class Test {
```

```
    static void blowUp() throws BlewIt {
      throw new NullPointerException();
    }
    public static void main(String[] args) {
        try {
            blowUp();
        } catch (BlewIt b) {
            System.out.println("Caught BlewIt");
        } finally {
            System.out.println("Uncaught Exception");
        }
    }
}
```

该程序产生以下输出：

```
Uncaught Exception
Exception in thread "main" java.lang.NullPointerException
        at Test.blowUp(Test.java:7)
        at Test.main(Test.java:11)
```

main 中的 try 语句不会捕获由方法 blowUp 引发的 NullPointerException(这是一种 RuntimeException)，因为 NullPointerException 不能赋给 BlewIt 类型的变量。这会导致执行 finally 子句，之后执行 main 的线程(测试程序的唯一线程)会因为未捕获的异常而终止，这通常会导致打印异常名称和简单的回溯。但是，本规范并不需要回溯。

强制回溯的问题是，可以在程序中的某个点创建异常，然后在以后的某个点引发异常。在异常中存储堆栈跟踪的开销高得令人望而却步，除非它实际被抛出(在这种情况下，跟踪可能是在展开堆栈时生成的)。因此，我们不会强制在每个异常中进行回溯。

14.20.3 try-with-resources

try-with-resources 语句使用变量(称为资源)进行参数化，这些变量在 try 块执行之前被初始化，并在 try 块执行后以它们被初始化的相反顺序自动关闭。当资源自动关闭时，catch 子句和 finally 子句通常是不必要的。

TryWithResourcesStatement:
`try` *ResourceSpecification Block [Catches] [Finally]*

ResourceSpecification:
`(` *ResourceList [;]* `)`

ResourceList:
Resource {; Resource}

Resource:
LocalVariableDeclaration
VariableAccess

VariableAccess:
ExpressionName

FieldAccess

为方便起见，此处列出了 §4.3, §8.3, §8.4.1, 和 §14.4 中的以下内容:

LocalVariableDeclaration:
{VariableModifier} LocalVariableType VariableDeclaratorList

VariableModifier: Annotation `final`

LocalVariableType: UnannType `var`

VariableDeclaratorList:
VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
VariableDeclaratorId [= VariableInitializer]

VariableDeclaratorId:
Identifier [Dims]

Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

VariableInitializer:
Expression
ArrayInitializer

See §8.3 for *UnannType*.

资源规范通过使用初始化器声明局部变量或引用现有变量来表示 try-with-resources 语句的资源。现有变量由表达式名称(§6.5.6)或字段访问表达式(§15.11)引用。

在资源规范中声明的局部变量的规则在§14.4 中指定。此外，以下所有条件都必须为真，否则会发生编译时错误:

- VariableDeclaratorList 由一个 VariableDeclarator 组成。
- VariableDeclarator 有一个初始化器。
- VariableDeclaratorId 没有括号对。

§6.3 和§6.4 规定了资源规范中声明的局部变量的作用域和遮蔽。

对嵌套类或接口或 lambda 表达式的局部变量的引用是受限制的，如§6.5.6.1 中规定的。

资源规范中声明的局部变量的类型见§14.4.1。

资源规范中声明的局部变量的类型，或资源规范中引用的现有变量的类型，必须是 AutoCloseable 的子类型，否则将发生编译时错误。

如果资源规范声明两个具有相同名称的局部变量，则会导致编译时错误。

资源是 final 的，因为:

- 在资源规范中声明的局部变量如果没有显式声明为 final，则隐式声明为 final(§4.12.4)。

- 资源规范中引用的现有变量必须是在 try-with- resources 语句之前明确赋值(§16)的 final 变量或实际的 final 变量，否则会发生编译时错误。

资源按照从左到右的顺序初始化。如果一个资源初始化失败(也就是说，它的初始化表达式抛出异常)，那么目前为止通过 try-with-resources 语句初始化的所有资源都将关闭。如果所有资源初始化成功，try 块将正常执行，然后关闭 try-with- resources 语句中的所有非空资源。

资源关闭的顺序与它们初始化时的顺序相反。只有初始化为非空值的资源才会关闭。关闭一个资源的异常并不会阻止其他资源的关闭。 如果先前由初始化器、try 块或关闭资源引发异常，则会抑制此类异常。

如果 try-with-resources 语句的资源规范表明有多个资源，那么它就会被当作多个 try-with-resources 语句来处理，每个 try-with-resources 语句都有一个资源规范，表明单个资源。当一个包含 n 个资源(n>1)的 try-with-resources 语句被转换时，结果是一个包含 n-1 个资源的 try-with-resources 语句。在 n 次这样的转换之后，有 n 个嵌套的 try-catch-finally 语句，整个转换就完成了。

14.20.3.1 基本的 try-with-resources

没有 catch 子句或 finally 子句的 try-with-resources 语句被称为基本的 try-with-resources 语句。

如果基本的 try-with-resources 语句是这样的:

```
try (VariableAccess ...)
    Block
```

然后，资源首先通过以下转换转换为局部变量声明:

```
try (T #r = VariableAccess ...) {
    Block
}
```

T 是 VariableAccess 所表示的变量类型，而#r 是一个自动生成的标识符，它不同于发生 try-with-resources 语句时作用域中的任何其他标识符(自动生成或其他方式)。然后，根据本节的其余部分对 try-with-resources 语句进行转换。

基本的 try-with-resources 语句的含义如下:

```
try ({VariableModifier} R Identifier = Expression ...)
    Block
```

通过以下转换为局部变量声明和 try-catch-finally 语句给出:

```
{
    final {VariableModifierNoFinal} R Identifier = Expression;
    Throwable #primaryExc = null;

    try ResourceSpecification_tail
        Block
```

```
        catch (Throwable #t) {
            #primaryExc = #t;
            throw #t;
        } finally {
            if (Identifier != null) {
                if (#primaryExc != null) {
                    try {
                        Identifier.close();
                    } catch (Throwable #suppressedExc) {
                        #primaryExc.addSuppressed(#suppressedExc);
                    }
                } else {
                    Identifier.close();
                }
            }
        }
    }
```

{VariableModifierNoFinal} 被定义为不带 final 的 {VariableModifier} （如果存在）。

t, #primaryExc,和#suppressedExc 是自动生成的标识符，它们不同于 try-with-resources 语句发生时作用域中的任何其他标识符(自动生成或以其他方式生成)。

如果资源规范指示一个资源，则 ResourceSpecification_tail 为空(且 try-catch-finally 语句本身不是 try-with-resources 语句)。

如果资源规范指示 n>1 个资源，则 ResourceSpecification_tail 由资源规范中指示的第 2、3、…、n 个资源以相同的顺序组成(而 try-catch-finally 语句本身就是 try-with-resources 语句)。

上面的转换隐含地指定了基本 try-with-resources 语句的可达性和明确的赋值规则。

在管理单个资源的基本 try-with-resources 语句中：

- 如果资源的初始化因抛出值 V 而突然完成，则 try-with-resources 语句因抛出值 V 而突然完成。
- 如果资源的初始化正常完成，而 try 块由于抛出值 V 而突然完成，则：
  - 如果资源的自动关闭正常完成，则 try-with-resources 语句会因为抛出值 V 而突然完成。
  - 如果资源的自动关闭因抛出值 V2 而突然完成，则 try-with-resources 语句因抛出值 V 而突然完成，并将 V2 添加到受抑制的异常列表 V 中。
- 如果资源的初始化正常完成，try 块正常完成，并且资源的自动关闭因抛出值 V 而突然完成，则 try-with-resources 语句会因抛出值 V 而突然完成。

在管理多个资源的基本 try-with-resources 语句中：

- 如果资源的初始化因抛出值 V 而突然完成，则：

- 如果所有成功初始化的资源(可能为零)的自动关闭正常完成，则 try-with-resources 语句因抛出值 V 而突然完成。
- 如果所有成功初始化的资源(可能为零)的自动关闭由于抛出值 V_1...V_n 而突然完成，则 try-with-resources 语句由于抛出值 V 而突然完成，并且将任何剩余的值 V_1...V_n 添加到 V 的已抑制异常列表中。

- 如果所有资源的初始化正常完成，而 try 块由于抛出值 V 而突然完成，则:
  - 如果所有初始化资源的自动关闭正常完成，则 try-with-resources 语句会因为抛出值 V 而突然结束。
  - 如果一个或多个初始化资源的自动关闭由于抛出值 V_1...V_n 而突然完成，则 try-with-resources 语句由于抛出值 V 而突然完成，并且任何剩余的值 V_1...V_n 被添加到 V 的已抑制异常列表中。
- 如果每个资源的初始化都正常完成，并且 try 块也正常完成，则:
  - 如果初始化资源的一个自动关闭因抛出值 V 而突然完成，并且初始化资源的所有其他自动关闭正常完成，则 try-with-resources 语句因抛出值 V 而突然完成。
  - 如果由于抛出值 V_1...V_n(其中 V_1 是来自最右侧资源的未能关闭的异常，而 V_n 是来自最左侧的资源未能关闭的异常)而突然完成一个以上的已初始化资源的自动关闭，则 try-with-resources 语句因抛出值 V_1 而突然完成，并且将任何剩余的值 V_2...V_n 添加到 V_1 的被抑制的异常列表中。

14.20.3.2 扩展的 try-with-resources

至少具有一个 catch 子句和/或 finally 子句的 try-with-resources 语句称为扩展的 try-with-resources 语句。

扩展的 try-with-resources 语句的含义:

```
try ResourceSpecification
    Block
[Catches]
[Finally]
```

由以下转换提供给嵌套在 try-catch、try-finally 或 try-catch-finally 语句中的基本 try-with-resources 语句:

```
try {
    try ResourceSpecification
        Block
}
[Catches]
[Finally]
```

转换的效果是将资源规范“放入”try 语句中。这允许扩展的 try-with-resources 语句的 catch 子句捕获由于自动初始化或关闭任何资源而导致的异常。

此外，在执行 finally 块时，所有资源都将关闭(或尝试关闭)，这与 finally 关键字的意图一致。

14.21 yield 语句

yield 语句通过使封闭的 switch 表达式(§15.28)产生指定值来转移控制。

YieldStatement: `yield` *Expression* `;`

yield 语句试图将控制转移到最里面的封闭 switch 表达式；这个封闭表达式称为 yield 目标，然后立即正常完成，并且 Expression 的值成为 switch 表达式的值。

如果 yield 语句没有 yield 目标，则它是编译时错误。

如果 yield 目标包含包含 yield 语句的任何方法、构造函数、实例初始化器、静态初始化器或 lambda 表达式，则为编译时错误。也就是说，没有非本地跳跃。

如果 yield 语句的 Expression 为 void(§15.1)，则为编译时错误。

执行 yield 语句首先对 Expression 求值。如果 Expression 的计算由于某种原因而突然完成，则 yield 语句也会因此原因而突然完成。如果 Expression 的计算正常完成，并产生值 V，则 yield 语句突然结束，原因是带有值 V 的 yield。

因此，可以看出，yield 语句总是突然结束。

例子 14.21-1. yield 语句

在下面的示例中，使用 yield 语句为封闭的 switch 表达式生成值。

```
class Test {
   enum Day {
      MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
      SATURDAY, SUNDAY
   }

   public int calculate(Day d) {
      return switch (d) {
         case SATURDAY, SUNDAY -> d.ordinal();
         default -> {
            int len = d.toString().length();
            yield len*len;
         }
      };
   }
}
```

14.22 无法访问的语句

如果语句因为无法访问而无法执行，则为编译时错误。

这一部分致力于对“可访问”一词的准确解释。其思想是，从包含语句的构造函数、方法、实例初始化器或静态初始化器的开头到语句本身，必须有一些可能的执行路径。这一分析考虑到了语句的结构。

除了对 while、do 和条件表达式的常量值为 true 的 for 语句进行特殊处理外，在流分析中不考虑表达式的值。

例如，Java 编译器将接受以下代码:

```
{
 int n = 5;
 while (n > 7) k = 2;
}
```

即使 n 的值在编译时是已知的，并且原则上可以在编译时知道对 k 的赋值永远不能被执行。

本节中的规则定义了两个技术术语:

- 语句是否可访问
- 语句是否可以正常完成

规则允许语句仅在可访问时才能正常完成。

还使用了另外两个技术术语:

- 如果在 break 目标内没有 try 语句的 try 块包含 break 语句，或者存在 try 块包含 break 语句并且这些 try 语句的所有 finally 子句都可以正常完成的 try 语句，则可访问的 break 语句退出语句。

  这个定义是基于§14.15 中关于“试图转移控制”的逻辑。

- 如果在 do 语句中没有 try 语句的 try 块包含 continue 语句，或者有 try 语句的 try 块包含 continue 语句，并且这些 try 语句的所有 finally 子句都可以正常完成，则 continue 语句继续 do 语句。

规则如下:

- 构造函数、方法、实例初始化器、静态初始化器、lambda 表达式或 switch 表达式的主体块是可访问的。
- 非 switch 块的空块可以正常完成当且仅当它是可访问的。

  非 switch 块的非空块可以正常完成当且仅当它的最后一条语句可以正常完成。

  非 switch 块的非空块的第一条语句是可访问的当且仅当块是可访问的。

  非 switch 块的非空块的每一个其他语句 S 是可访问的当且仅当在 S 之前的语句可以正常完成。
- 局部类声明语句可以正常完成当且仅当它是可达的。
- 局部变量声明语句可以正常完成当且仅当它是可达的。
- 空语句可以正常完成当且仅当它是可达的。
- 标签语句可以正常完成如果以下至少有一个是正确的:

- 包含的语句可以正常完成。
- 有一个可访问的 break 语句，它退出标记语句。

包含的语句是可访问的当且仅当标签语句是可访问的。

- 表达式语句可以正常完成当且仅当它是可访问的。
- 一个 if-then 语句可以正常完成当且仅当它是可访问的。

  then 语句是可访问的当且仅当 if-then 语句是可访问的。

  if-then-else 语句可以正常完成当且仅当 then 语句可以正常完成或者 else 语句可以正产完成。

  then 语句是可访问的当且仅当 if-then-else 语句是可访问的。

  else 语句是可访问的当且仅当 if-then-else 语句是可访问的。

  > 对于 if 语句的这种处理，无论它是否有 else 部分，都是相当不寻常的。基本原理在本节的最后给出。

- assert 语句可以正常完成当且仅当它是可访问的。
- 空 switch 块的 switch 语句，或者只包含 switch 标签的 switch 语句，可以正常完成。
- 如果 switch 语句块由带有 switch 标签的语句组组成，则 switch 语句可以正常完成当且仅当以下至少有一个为真:
  - switch 块的最后一个语句可以正常完成。
  - 在最后一个 switch 块语句组之后至少有一个 switch 标签。
  - 有一个可访问的 break 语句退出 switch 语句。
  - switch 块不包含 default 标签。
- 由 switch 规则组成 switch 块的 switch 语句可以正常完成当且仅当以下至少一个为真:
  - 其中一个 switch 则引入了 switch 规则表达式（它必须是语句表达式）。
  - 其中一个 switch 规则引入了可以正常完成的 switch 规则块。
  - 其中一条 switch 规则引入了一个 switch 规则块，该块包含退出 switch 语句的可访问 break 语句。
  - switch 块不包含 default 标签。
- switch 块可访问当且仅当它的 switch 语句是可访问的。
- switch 块中由 switch 标记的语句组组成的语句是可访问的当且仅当 switch 块是可访问的并且以下至少一个为真:
  - 它带有 case 或 default 标签。

- 在 switch 块中有一条语句在它前面，并且前面的语句可以正常完成。

- switch 块中的 switch 规则块是可访问的当且仅当 switch 块是可访问的。
- switch 块中的 switch 规则抛出语句是可访问的 当且仅当 switch 块是可访问的。
- while 语句可以正常完成当且仅当以下至少一个为真：
  - while 语句是可访问的，条件表达式不是值为 true 的常量表达式（§15.29）。
  - 有一个可访问的 break 语句退出 while 语句。

  包含的语句是可访问的当且仅当 while 语句是可访问的，并且条件表达式不是值为 false 的常量表达式。
- do 语句可以正常完成当且仅当以下至少一个为真：
  - 包含的语句可以正常完成，条件表达式不是值为 true 的常量表达式（§15.29）。
  - do 语句包含一个不带标签的可访问的 continue 语句，do 语句是包含该 continue 的最里面的 while、do 或 for 语句，continue 语句继续该 do 语句，并且条件表达式不是值为 true 的常量表达式。
  - do 语句包含标签为 L 的可访问 continue 语句，do 语句具有标签 L，continue 语句继续该 do 语句，并且条件表达式不是值为 true 的常量表达式。
  - 存在退出 do 语句的可访问的 break 语句。

  包含的语句是可访问的当且仅当 do 语句是可访问的
- 基本的 for 语句可以正常完成当且仅当以下至少一个为真：
  - for 语句是可访问的，有一个条件表达式，并且条件表达式不是值为 true 的常量表达式（§15.29）。
  - 有一个可访问的 break 语句退出 for 语句。

  包含的语句是可访问的当且仅当 for 语句是可访问的，并且条件表达式不是值为 false 的常量表达式。
- 增强的 for 语句可以正常完成当且仅当它是可访问的。
- break, continue, return, throw, 或 yield 语句不能正常完成。
- 同步语句可以正常完成当且仅当包含的语句可以正常完成。

  包含的语句是可以访问的当且仅当同步语句是可以访问的。
- try 语句可以正常完成当且仅当以下所有都为真：
  - try 块可以正常完成或任何 catch 块可以正常完成。
  - 如果 try 语句有一个 finally 块，那么 finally 块可以正常完成。

- try 块是可访问的当且仅当 try 语句是可访问的。
- catch 块 C 是可访问的当且仅当以下所有都为真：
  - C 参数的类型是未检查的异常类型或 Exception 或 Exception 的超类，或者 try 块中的某个表达式或 throw 语句是可访问的，并且可以抛出类型与 C 参数类型赋值兼容（§5.2）的已检查异常。(表达式是可访问的当且仅当包含它的最内部的语句是可访问的。)

    表达式的正常和突然完成见§15.6。
  - try 语句中没有较早的 catch 块 A，因此 C 的参数类型与 A 的参数类型相同，或是其子类。
- catch 块的 Block 是可访问的当且仅当 catch 块是可访问的。
- 如果存在 finally 块，当且仅当 try 语句是可访问的，它也是可访问的。

人们可能希望用以下方式处理 if 语句：

• if-then 语句可以正常完成当且仅当以下至少一个为真：

- if-then 语句是可访问的，并且条件表达式不是值为 true 的常量表达式。
- then 语句可以正常完成。

then 语句是可访问的当且仅当 if-then 语句是可访问的并且条件表达式不是值为 false 的常量表达式。

• if-then-else 语句可以正常完成当且仅当 then 语句可以正常完成或 else 语句可以正常完成。

then 语句是可访问的当且仅当 if-then-else 语句是可访问的并且条件表达式不是值为 false 的常量表达式。

else 语句是可访问的当且仅当 if-then-else 语句是可访问的并且条件表达式不是值为 true 的常量表达式。

这种方法将与其他控制结构的处理一致。然而，为了方便地将 if 语句用于“条件编译”目的，实际规则有所不同。

例如，以下语句会导致编译时错误：

```
while (false) { x=3; }
```

因为语句 x=3; 不可访问的；但表面上类似的情况：

```
if (false) { x=3; }
```

不会导致编译时错误。优化编译器可以实现语句 x = 3；将永远不会执行，并且可以选择从生成的 class 文件中省略该语句的代码，但语句 x=3,在此处规定的技术意义上，不被视为“无法访问”。

这种不同处理的基本原理是允许程序员定义“标志”变量，例如：

```
static final boolean DEBUG = false;
```

然后编写代码，例如：

```
if (DEBUG) { x=3; }
```

其思想是，应该可以将 DEBUG 的值从 false 更改为 true 或从 true 更改为 false，然后正确编译代码，而无需对程序文本进行其他更改。

条件编译附带一个警告。如果编译了一组使用“标志”变量（或更准确地说，任何静态常量变量（§4.12.4））的类，并省略了条件代码，则仅分发包含标志定义的类或接口的新版本是不够的。使用该标志的类将看不到它的新值，因此它们的行为可能会令人惊讶。本质上，对标志值的更改与已存在的二进制文件是二进制兼容的(没有发生 LinkageError)，但在行为上不兼容。

“内联”静态常量变量值的另一个原因是 switch 语句。它们是唯一一种依赖于常量表达式的语句，即 switch 语句的每个 case 标签必须是一个常量表达式，其值必须与其他 case 标签不同。case 标签通常是对静态常量变量的引用，所以可能不会立即看出所有标签都有不同的值。如果证明在编译时没有重复标签，那么将值内联到 class 文件中也可以确保在运行时没有重复标签—这是一个非常理想的属性。

例子 14.22-1. 条件编译

如果这个例子：

```
class Flags { static final boolean DEBUG = true; }
class Test {
    public static void main(String[] args) {
      if (Flags.DEBUG)
              System.out.println("DEBUG is true");
    }
}
```

编译并执行时，它生成输出：

```
DEBUG is true
```

假设产生了一个新版本的类 Flags:

```
class Flags { static final boolean DEBUG = false; }
```

如果 Flags 被重编但 Test 没有被重编，那么用现有的 Test 的二进制文件运行新的二进制文件产生输出：

```
DEBUG is true
```

因为 DEBUG 是一个静态常量变量，因此，它的值可以在编译 Test 时使用，而无需引用类 Flags。

如果 Flags 是接口，也会发生这种行为，如修改后的示例中所示：

```
interface Flags { boolean DEBUG = true; }
class Test {
    public static void main(String[] args) {
        if (Flags.DEBUG)
              System.out.println("DEBUG is true");
    }
}
```

事实上，由于接口的字段始终是静态的和 final 的，我们建议只将常量表达式赋值给接口的字段。我们注意到，但不建议，如果接口的原生类型字段可能更改，则其值可以按惯用方式表示为：

```
interface Flags {
    boolean debug = Boolean.valueOf(true).booleanValue();
}
```

确保该值不是常量表达式。其他原生类型也有类似的习惯用法。

14.30 模式

模式描述了可以对值执行的测试。模式显示为语句和表达式的操作数，提供要测试的值。模式声明局部变量，称为模式变量。

根据模式测试值的过程称为模式匹配。如果一个值成功地匹配了一个模式，那么模式匹配过程将初始化该模式声明的模式变量。

模式变量只在模式匹配成功的范围内(§6.3)，因此模式变量将被初始化。不可能使用未初始化的模式变量。

14.30.1 模式的类型

类型模式用于测试一个值是否是模式中出现的类型的实例。

Pattern:
  TypePattern

TypePattern:
  LocalVariableDeclaration

为方便起见，此处显示了§4.3、§8.3、§8.4.1 和§14.4 中的以下内容：

LocalVariableDeclaration:
  {VariableModifier} LocalVariableType VariableDeclaratorList

VariableModifier:
  Annotation `final`

LocalVariableType:
  UnannType `var`

VariableDeclaratorList:
  VariableDeclarator {, VariableDeclarator}

VariableDeclarator:

VariableDeclaratorId [= VariableInitializer]

VariableDeclaratorId:
Identifier [Dims]

Dims:
{Annotation} [ ] {{Annotation} [ ]}

参见§8.3 了解 Unnantype。

类型模式声明一个局部变量，称为模式变量。局部变量声明中的标识符指定模式变量的名称。

类型模式中声明的局部变量的规则在§14.4 中规定。此外，以下所有条件必须为真，否则将发生编译时错误:

- LocalVariableType 表示引用类型 (而且，这不是 var)。
- VariableDeclaratorList 由一个 VariableDeclarator 组成。
- VariableDeclarator 没有初始化器。
- VariableDeclaratorId 没有括号对。

模式变量的类型是由 LocalVariableType 表示的引用类型。

类型模式的类型是其模式变量的类型。

如果表达式 e 与 T 向下转换兼容，则表达式 e 与类型 T 的模式兼容(§5.5)。

表达式与模式的兼容性由 instanceof 模式匹配运算符使用(§15.20.2)。

14.30.2 模式匹配

模式匹配是在运行时根据模式测试值的过程。模式匹配不同于语句执行(§14.1)和表达式求值(§15.1)。

确定一个值是否匹配一个模式，以及初始化模式变量的规则如下:

- 如果值 v 可以强制转换为类型 T 而不引发 ClassCastException，则非空引用的值 v 匹配类型 T 的类型模式;否则就不匹配了。

  如果 v 匹配，则类型模式声明的模式变量初始化为 v。

  如果 v 不匹配，则类型模式声明的模式变量不初始化。

  没有规则可以覆盖空引用的值。这是因为执行模式匹配的孤立结构，即模式匹配操作符 instanceof(§15.20.2)，只在值不是空引用的时候才这样做。Java 编程语言的未来版本可能允许在其他表达式和语句中进行模式匹配。