CHAPTER 14

Blocks and Statements

He was not merely a chip of the old block, but the old block itself.
—Edmund Burke, *On Pitt's First Speech*

THE sequence of execution of a program is controlled by *statements*, which are executed for their effect and do not have values.

Some statements *contain* other statements as part of their structure; such other statements are substatements of the statement. We say that statement *S* *immediately contains* statement *U* if there is no statement *T* different from *S* and *U* such that *S* contains *T* and *T* contains *U*. In the same manner, some statements contain expressions (§15) as part of their structure.

The first section of this chapter discusses the distinction between normal and abrupt completion of statements (§14.1). Most of the remaining sections explain the various kinds of statements, describing in detail both their normal behavior and any special treatment of abrupt completion.

Blocks are explained first (§14.2), followed by local class declarations (§14.3) and local variable declaration statements (§14.4).

Next a grammatical maneuver that sidesteps the familiar "dangling `else`" problem (§14.5) is explained.

The last section (§14.21) of this chapter addresses the requirement that every statement be *reachable* in a certain technical sense.

14.1 Normal and Abrupt Completion of Statements

Poirot's abrupt departure had intrigued us all greatly.
—Agatha Christie, *The Mysterious Affair at Styles* (1920), Chapter 12

Every statement has a normal mode of execution in which certain computational steps are carried out. The following sections describe the normal mode of execution for each kind of statement.

If all the steps are carried out as described, with no indication of abrupt completion, the statement is said to *complete normally*. However, certain events may prevent a statement from completing normally:

- The `break` (§14.15), `continue` (§14.16), and `return` (§14.17) statements cause a transfer of control that may prevent normal completion of statements that contain them.
- Evaluation of certain expressions may throw exceptions from the Java virtual machine; these expressions are summarized in §15.6. An explicit `throw` (§14.18) statement also results in an exception. An exception causes a transfer of control that may prevent normal completion of statements.

If such an event occurs, then execution of one or more statements may be terminated before all steps of their normal mode of execution have completed; such statements are said to *complete abruptly*.

An abrupt completion always has an associated *reason*, which is one of the following:

- A `break` with no label
- A `break` with a given label
- A `continue` with no label
- A `continue` with a given label
- A `return` with no value
- A `return` with a given value
- A `throw` with a given value, including exceptions thrown by the Java virtual machine

The terms "complete normally" and "complete abruptly" also apply to the evaluation of expressions (§15.6). The only reason an expression can complete abruptly is that an exception is thrown, because of either a `throw` with a given value (§14.18) or a run-time exception or error (§11, §15.6).

If a statement evaluates an expression, abrupt completion of the expression always causes the immediate abrupt completion of the statement, with the same reason. All succeeding steps in the normal mode of execution are not performed.

Unless otherwise specified in this chapter, abrupt completion of a substatement causes the immediate abrupt completion of the statement itself, with the same reason, and all succeeding steps in the normal mode of execution of the statement are not performed.

Unless otherwise specified, a statement completes normally if all expressions it evaluates and all substatements it executes complete normally.

14.2 Blocks

He wears his faith but as the fashion of his hat;
it ever changes with the next block.
—William Shakespeare, *Much Ado about Nothing* (1623), Act I, scene i

A *block* is a sequence of statements, local class declarations and local variable declaration statements within braces.

Block:
  `{` *BlockStatements*$_{opt}$ `}`

BlockStatements:
  BlockStatement
  BlockStatements BlockStatement

BlockStatement:
  LocalVariableDeclarationStatement
  ClassDeclaration
  Statement

A block is executed by executing each of the local variable declaration statements and other statements in order from first to last (left to right). If all of these block statements complete normally, then the block completes normally. If any of these block statements complete abruptly for any reason, then the block completes abruptly for the same reason.

14.3 Local Class Declarations

A *local class* is a nested class (§8) that is not a member of any class and that has a name. All local classes are inner classes (§8.1.2). Every local class declaration statement is immediately contained by a block. Local class declaration statements may be intermixed freely with other kinds of statements in the block.

The scope of a local class immediately enclosed by a block (§14.2) is the rest of the immediately enclosing block, including its own class declaration. The scope of a local class immediately enclosed by in a switch block statement group

(§14.11)is the rest of the immediately enclosing switch block statement group, including its own class declaration.

The name of a local class *C* may not be redeclared as a local class of the directly enclosing method, constructor, or initializer block within the scope of *C*, or a compile-time error occurs. However, a local class declaration may be shadowed (§6.3.1) anywhere inside a class declaration nested within the local class declaration's scope. A local class does not have a canonical name, nor does it have a fully qualified name.

It is a compile-time error if a local class declaration contains any one of the following access modifiers: `public`, `protected`, `private`, or `static`.

Here is an example that illustrates several aspects of the rules given above:

```
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
}
```

The first statement of method `foo` creates an instance of the member class `Global.Cyclic` rather than an instance of the local class `Cyclic`, because the local class declaration is not yet in scope.

The fact that the scope of a local class encompasses its own declaration (not only its body) means that the definition of the local class `Cyclic` is indeed cyclic because it extends itself rather than `Global.Cyclic`. Consequently, the declaration of the local class `Cyclic` will be rejected at compile time.

Since local class names cannot be redeclared within the same method (or constructor or initializer, as the case may be), the second and third declarations of `Local` result in compile-time errors. However, `Local` can be redeclared in the context of another, more deeply nested, class such as `AnotherLocal`.

The fourth and last declaration of `Local` is legal, since it occurs outside the scope of any prior declaration of `Local`.

14.4 Local Variable Declaration Statements

A *local variable declaration statement* declares one or more local variable names.

LocalVariableDeclarationStatement:
  LocalVariableDeclaration `;`

LocalVariableDeclaration:
  VariableModifiers Type VariableDeclarators

The following are repeated from §8.3 to make the presentation here clearer:

VariableDeclarators:
  VariableDeclarator
  VariableDeclarators `,` *VariableDeclarator*

VariableDeclarator:
  VariableDeclaratorId
  VariableDeclaratorId `=` *VariableInitializer*

VariableDeclaratorId:
  Identifier
  VariableDeclaratorId `[ ]`

VariableInitializer:
  Expression
  ArrayInitializer

Every local variable declaration statement is immediately contained by a block. Local variable declaration statements may be intermixed freely with other kinds of statements in the block.

A local variable declaration can also appear in the header of a `for` statement (§14.14). In this case it is executed in the same manner as if it were part of a local variable declaration statement.

14.4.1 Local Variable Declarators and Types

Each *declarator* in a local variable declaration declares one local variable, whose name is the *Identifier* that appears in the declarator.

If the optional keyword `final` appears at the start of the declarator, the variable being declared is a final variable(§4.5.4).

If an annotation *a* on a local variable declaration corresponds to an annotation type *T*, and *T* has a (meta-)annotation *m* that corresponds to `annotation.Target`, then *m* must have an element whose value is `annotation.Element-`

`Type.LOCAL_VARIABLE`, or a compile-time error occurs. Annotation modifiers are described further in (§9.7).

The type of the variable is denoted by the *Type* that appears in the local variable declaration, followed by any bracket pairs that follow the *Identifier* in the declarator.

Thus, the local variable declaration:

```
int a, b[], c[][];
```

is equivalent to the series of declarations:

```
int a;
int[] b;
int[][] c;
```

Brackets are allowed in declarators as a nod to the tradition of C and C++. The general rule, however, also means that the local variable declaration:

```
float[][] f[][], g[][][], h[];// Yechh!
```

is equivalent to the series of declarations:

```
float[][][][] f;
float[][][][][] g;
float[][][] h;
```

We do not recommend such “mixed notation” for array declarations.

A local variable of type `float` always contains a value that is an element of the float value set (§4.2.3); similarly, a local variable of type `double` always contains a value that is an element of the double value set. It is not permitted for a local variable of type `float` to contain an element of the float-extended-exponent value set that is not also an element of the float value set, nor for a local variable of type `double` to contain an element of the double-extended-exponent value set that is not also an element of the double value set.

14.4.2 Scope of Local Variable Declarations

The scope of a local variable declaration in a block (§14.4.2) is the rest of the block in which the declaration appears, starting with its own initializer (§14.4) and including any further declarators to the right in the local variable declaration statement.

The name of a local variable *v* may not be redeclared as a local variable of the directly enclosing method, constructor or initializer block within the scope of *v*, or a compile-time error occurs. The name of a local variable *v* may not be redeclared as an exception parameter of a catch clause in a try statement of the directly enclosing method, constructor or initializer block within the scope of *v*, or a compile-time error occurs. However, a local variable of a method or initializer block may be shadowed (§6.3.1) anywhere inside a class declaration nested within the scope of the local variable.

A local variable cannot be referred to using a qualified name (§6.6), only a simple name.

The example:

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

causes a compile-time error because the initialization of x is within the scope of the declaration of x as a local variable, and the local x does not yet have a value and cannot be used.

The following program does compile:

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

because the local variable x is definitely assigned (§16) before it is used. It prints:

```
4
```

Here is another example:

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

which compiles correctly and produces the output:

```
2+1=3
```

The initializer for `three` can correctly refer to the variable `two` declared in an earlier declarator, and the method invocation in the next line can correctly refer to the variable `three` declared earlier in the block.

The scope of a local variable declared in a `for` statement is the rest of the `for` statement, including its own initializer.

If a declaration of an identifier as a local variable of the same method, constructor, or initializer block appears within the scope of a parameter or local variable of the same name, a compile-time error occurs.

Thus the following example does not compile:

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

This restriction helps to detect some otherwise very obscure bugs. A similar restriction on shadowing of members by local variables was judged impractical, because the addition of a member in a superclass could cause subclasses to have to rename local variables. Related considerations make restrictions on shadowing of local variables by members of nested classes, or on shadowing of local variables by local variables declared within nested classes unattractive as well. Hence, the following example compiles without error:

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

On the other hand, local variables with the same name may be declared in two separate blocks or `for` statements neither of which contains the other. Thus:

```
    class Test {
        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();
        }
    }
```

compiles without error and, when executed, produces the output:

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

14.4.3 Shadowing of Names by Local Variables

If a name declared as a local variable is already declared as a field name, then that outer declaration is shadowed (§6.3.1) throughout the scope of the local variable.

Similarly, if a name is already declared as a variable or parameter name, then that outer declaration is shadowed throughout the scope of the local variable (provided that the shadowing does not cause a compile-time error under the rules of §14.4.2). The shadowed name can sometimes be accessed using an appropriately qualified name.

For example, the keyword `this` can be used to access a shadowed field `x`, using the form `this.x`. Indeed, this idiom typically appears in constructors (§8.8):

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

In this example, the constructor takes parameters having the same names as the fields to be initialized. This is simpler than having to invent different names for the parameters and is not too confusing in this stylized context. In general, however, it is considered poor style to have local variables with the same names as fields.

14.4.4 Execution of Local Variable Declarations

A local variable declaration statement is an executable statement. Every time it is executed, the declarators are processed in order from left to right. If a declarator has an initialization expression, the expression is evaluated and its value is assigned to the variable. If a declarator does not have an initialization expression, then a Java compiler must prove, using exactly the algorithm given in §16, that every reference to the variable is necessarily preceded by execution of an assignment to the variable. If this is not the case, then a compile-time error occurs.

Each initialization (except the first) is executed only if the evaluation of the preceding initialization expression completes normally. Execution of the local variable declaration completes normally only if evaluation of the last initialization expression completes normally; if the local variable declaration contains no initialization expressions, then executing it always completes normally.

14.5 Statements

There are many kinds of statements in the Java programming language. Most correspond to statements in the C and C++ languages, but some are unique.

As in C and C++, the `if` statement of the Java programming language suffers from the so-called “dangling `else` problem,” illustrated by this misleadingly formatted example:

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

The problem is that both the outer `if` statement and the inner `if` statement might conceivably own the `else` clause. In this example, one might surmise that the programmer intended the `else` clause to belong to the outer `if` statement. The Java programming language, like C and C++ and many programming languages before them, arbitrarily decree that an `else` clause belongs to the innermost `if` to which it might possibly belong. This rule is captured by the following grammar:

Statement:
    StatementWithoutTrailingSubstatement
    LabeledStatement
    IfThenStatement
    IfThenElseStatement
    WhileStatement
    ForStatement

StatementWithoutTrailingSubstatement:
    Block
    EmptyStatement
    ExpressionStatement

    AssertStatement
    SwitchStatement
    DoStatement
    BreakStatement
    ContinueStatement
    ReturnStatement
    SynchronizedStatement
    ThrowStatement
    TryStatement

StatementNoShortIf:
    StatementWithoutTrailingSubstatement
    LabeledStatementNoShortIf
    IfThenElseStatementNoShortIf
    WhileStatementNoShortIf
    ForStatementNoShortIf

The following are repeated from §14.9 to make the presentation here clearer:

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

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

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

Statements are thus grammatically divided into two categories: those that might end in an `if` statement that has no `else` clause (a "short `if` statement") and those that definitely do not. Only statements that definitely do not end in a short `if` statement may appear as an immediate substatement before the keyword `else` in an `if` statement that does have an `else` clause.

This simple rule prevents the "dangling `else`" problem. The execution behavior of a statement with the "no short `if`" restriction is identical to the execution behavior of the same kind of statement without the "no short `if`" restriction; the distinction is drawn purely to resolve the syntactic difficulty.

14.6 The Empty Statement

I did never know so full a voice issue from so empty a heart:
but the saying is true 'The empty vessel makes the greatest sound.'
—William Shakespeare, *Henry V* (1623), Act IV, scene iv

An *empty statement* does nothing.

EmptyStatement:
  `;`

Execution of an empty statement always completes normally.

14.7 Labeled Statements

Inside of five minutes I was mounted, and perfectly satisfied
with my outfit. I had no time to label him "This is a horse,"
and so if the public took him for a sheep I cannot help it.
—Mark Twain, *Roughing It* (1871)

Statements may have *label* prefixes.

LabeledStatement:
  Identifier **:** *Statement*

LabeledStatementNoShortIf:
  Identifier **:** *StatementNoShortIf*

The *Identifier* is declared to be the label of the immediately contained *Statement*.

Unlike C and C++, the Java programming language has no `goto` statement; identifier statement labels are used with `break` (§14.15) or `continue` (§14.16) statements appearing anywhere within the labeled statement.

The scope of a label declared by a labeled statement is the statement immediately enclosed by the labeled statement.

Let *`l`* be a label, and let *`m`* be the immediately enclosing method, constructor, instance initializer or static initializer. It is a compile-time error if *`l`* shadows (§6.3.1) the declaration of another label immediately enclosed in *`m`*.

There is no restriction against using the same identifier as a label and as the name of a package, class, interface, method, field, parameter, or local variable. Use of an identifier to label a statement does not obscure (§6.3.2) a package, class, interface, method, field, parameter, or local variable with the same name. Use of an identifier as a class, interface, method, field, local variable or as the parameter of an exception handler (§14.20) does not obscure a statement label with the same name.

A labeled statement is executed by executing the immediately contained *Statement*. If the statement is labeled by an *Identifier* and the contained *Statement* completes abruptly because of a `break` with the same *Identifier*, then the labeled statement completes normally. In all other cases of abrupt completion of the *Statement*, the labeled statement completes abruptly for the same reason.

14.8 Expression Statements

Certain kinds of expressions may be used as statements by following them with semicolons:

ExpressionStatement:
  StatementExpression **;**

StatementExpression:
  Assignment
  PreIncrementExpression
  PreDecrementExpression
  PostIncrementExpression
  PostDecrementExpression
  MethodInvocation
  ClassInstanceCreationExpression

An *expression statement* is executed by evaluating the expression; if the expression has a value, the value is discarded. Execution of the expression statement completes normally if and only if evaluation of the expression completes normally.

Unlike C and C++, the Java programming language allows only certain forms of expressions to be used as expression statements. Note that the Java programming language does not allow a "cast to `void`"—`void` is not a type—so the traditional C trick of writing an expression statement such as:

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

does not work. On the other hand, the language allows all the most useful kinds of expressions in expressions statements, and it does not require a method invocation used as an expression statement to invoke a `void` method, so such a trick is almost never needed. If a trick is needed, either an assignment statement (§15.26) or a local variable declaration statement (§14.4) can be used instead.

14.9 The `if` Statement

The `if` statement allows conditional execution of a statement or a conditional choice of two statements, executing one or the other but not both.

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

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

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

The *Expression* must have type `boolean` or `Boolean`, or a compile-time error occurs.

14.9.1 The `if–then` Statement

I took an early opportunity of testing that statement . . .
—Agatha Christie, *The Mysterious Affair at Styles* (1920), Chapter 12

An `if–then` statement is executed by first evaluating the *Expression*. If the result is of type Boolean, it is subject to unboxing conversion (§5.1.8). If evaluation of the *Expression* or the subsequent unboxing conversion (if any) completes abruptly for some reason, the `if–then` statement completes abruptly for the same reason. Otherwise, execution continues by making a choice based on the resulting value:

- If the value is `true`, then the contained *Statement* is executed; the `if–then` statement completes normally if and only if execution of the *Statement* completes normally.
- If the value is `false`, no further action is taken and the `if–then` statement completes normally.

14.9.2 The `if–then–else` Statement

Did you ever have to finally decide—
To say yes to one, and let the other one ride?
—John Sebastian, *Did You Ever Have to Make Up Your Mind?*

An `if–then–else` statement is executed by first evaluating the *Expression*. If the result is of type Boolean, it is subject to unboxing conversion (§5.1.8). If evaluation of the *Expression* or the subsequent unboxing conversion (if any) completes abruptly for some reason, then the `if–then–else` statement completes abruptly for the same reason. Otherwise, execution continues by making a choice based on the resulting value:

- If the value is `true`, then the first contained *Statement* (the one before the `else` keyword) is executed; the `if–then–else` statement completes normally if and only if execution of that statement completes normally.
- If the value is `false`, then the second contained *Statement* (the one after the `else` keyword) is executed; the `if–then–else` statement completes normally if and only if execution of that statement completes normally.

14.10 The `assert` Statement

An *assertion* is a statement containing a boolean expression. An assertion is either *enabled* or *disabled*. If the assertion is enabled, evaluation of the assertion causes evaluation of the boolean expression and an error is reported if the expression evaluates to false. If the assertion is disabled, evaluation of the assertion has no effect whatsoever.

AssertStatement:

`assert` *Expression1 ;*

`assert` *Expression1 : Expression2 ;*

It is a compile-time error if *Expression1* does not have type `boolean` or `Boolean`. In the second form of the `assert` statement, it is a compile-time error if *Expression2* is void (§15.1).

Assertions may be enabled or disabled on a per-class basis. At the time a class is initialized, prior to the execution of any field initializers for class variables (§8.3.2.1) and static initializers (§8.7), the class's class loader determines whether assertions are enabled or disabled as described below. Once a class has been initialized, its assertion status (enabled or disabled) does not change.

DISCUSSION

There is one case that demands special treatment. Recall that the assertion status of a class is set at the time it is initialized. It is possible, though generally not desirable, to execute methods or constructors prior to initialization. This can happen when a class hierarchy contains a circularity in its static initialization, as in the following example:

```
public class Foo {
    public static void main(String[] args) {
         Baz.testAsserts(); // Will execute after Baz is ini-
tialized.
    }
}

class Bar {
    static {
        Baz.testAsserts(); // Will execute before Baz is ini-
tialized!
    }
}

class Baz extends Bar {
    static void testAsserts(){
        boolean enabled = false;
        assert enabled = true;
         System.out.println("Asserts " + (enabled ? "enabled"
: "disabled"));
    }
    }
```

Invoking `Baz.testAsserts()` causes `Baz` to get initialized. Before this can happen, `Bar` must get initialized. `Bar`'s static initializer again invokes `Baz.testAsserts()`. Because initialization of `Baz` is already in progress by the current thread, the second invocation executes immediately, though `Baz` is not initialized (JLS 12.4.2).

If an assert statement executes before its class is initialized, as in the above example, the execution must behave as if assertions were enabled in the class.

DISCUSSION

In other words, if the program above is executed without enabling assertions, it must print:

```
    Asserts enabled
    Asserts disabled
```

An `assert` statement is enabled if and only if the top-level class (§8) that lexically contains it enables assertions. Whether or not a top-level class enables assertions is determined by its defining class loader before the class is initialized (§12.4.2), and cannot be changed thereafter.

An `assert` statement causes the enclosing top level class (if it exists) to be initialized, if it has not already been initialized (§12.4.1).

DISCUSSION

Note that an assertion that is enclosed by a top-level interface does not cause initialization.

Usually, the top level class enclosing an assertion will already be initialized. However, if the assertionis located within a static nested class, it may be that the initialization has nottaken place.

A disabled `assert` statement does nothing. In particular neither *Expression1* nor *Expression2* (if it is present) are evaluated. Execution of a disabled `assert` statement always completes normally.

An enabled `assert` statement is executed by first evaluating *Expression1*. If the result is of type `Boolean`, it is subject to unboxing conversion (§5.1.8). If evaluation of *Expression1* or the subsequent unboxing conversion (if any) completes abruptly for some reason, the `assert` statement completes abruptly for the same reason. Otherwise, execution continues by making a choice based on the value of *Expression1* :

- If the value is `true`, no further action is taken and the assert statement completes normally.
- If the value is `false`, the execution behavior depends on whether *Expression2* is present:
  - If *Expression2* is present, it is evaluated.
    - If the evaluation completes abruptly for some reason, the `assert` statement completes abruptly for the same reason.
    - If the evaluation completes normally, the resulting value is converted to a `String` using string conversion (§15.18.1.1).
      - If the string conversion completes abruptly for some reason, the `assert` statement completes abruptly for the same reason.
      - If the string conversion completes normally, an `AssertionError` instance whose "detail message" is the result of the string conversion is created.
        - If the instance creation completes abruptly for some reason, the `assert` statement completes abruptly for the same reason.
        - If the instance creation completes normally, the `assert` statement completes abruptly by throwing the newly created `AssertionError` object.
  - If *Expression2* is not present, an `AssertionError` instance with no "detail message" is created.
    - If the instance creation completes abruptly for some reason, the `assert` statement completes abruptly for the same reason.
    - If the instance creation completes normally, the `assert` statement completes abruptly by throwing the newly created `AssertionError` object.

DISCUSSION

For example, after unmarshalling all of the arguments from a data buffer, a programmer might assert that the number of bytes of data remaining in the buffer is zero. By verifying that the boolean expression is indeed true, the system corroborates the programmer's knowledge of the program and increases one's confidence that the program is free of bugs.

Typically, assertion-checking is enabled during program development and testing, and disabled for deployment, to improve performance.

Because assertions may be disabled, programs must not assume that the expressions contained in assertions will be evaluated. Thus, these boolean expressions should generally be free of side effects:

Evaluating such a boolean expression should not affect any state that is visible after the evaluation is complete. It is not illegal for a boolean expression contained in an assertion to have a side effect, but it is generally inappropriate, as it could cause program behavior to vary depending on whether assertions were enabled or disabled.

Along similar lines, assertions should not be used for argument-checking in public methods. Argument-checking is typically part of the contract of a method, and this contract must be upheld whether assertions are enabled or disabled.

Another problem with using assertions for argument checking is that erroneous arguments should result in an appropriate runtime exception (such as `IllegalArgumentException`, `IndexOutOfBoundsException` or `NullPointerException`). An assertion failure will not throw an appropriate exception. Again, it is not illegal to use assertions for argument checking on public methods, but it is generally inappropriate. It is intended that AssertionError never be caught, but it is possible to do so, thus the rules for try statements should treat assertions appearing in a try block similarly to the current treatment of throw statements.

14.11 The `switch` Statement

Fetch me a dozen crab-tree staves, and strong ones: these are but switches . . .
—William Shakespeare, *Henry VIII* (1623), Act V, scene iv

The `switch` statement transfers control to one of several statements depending on the value of an expression.

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

SwitchBlock:
    `{` *SwitchBlockStatementGroups*$_{opt}$ *SwitchLabels*$_{opt}$ `}`

SwitchBlockStatementGroups:
    SwitchBlockStatementGroup
    SwitchBlockStatementGroups SwitchBlockStatementGroup

SwitchBlockStatementGroup:
  SwitchLabels BlockStatements

SwitchLabels:
  SwitchLabel
  SwitchLabels SwitchLabel

SwitchLabel:
  `case` *ConstantExpression* `:`
  `case` *EnumConstant* `:`
  `default :`

EnumConstant:
  Identifier

The type of the *Expression* must be `char`, `byte`, `short`, `int`, `Character`, `Byte`, `Short`, `Integer`, or an enum type (§8.9), or a compile-time error occurs.

The body of a `switch` statement is known as a *switch block*. Any statement immediately contained by the switch block may be labeled with one or more `case` or `default` labels. These labels are said to be *associated* with the `switch` statement, as are the values of the constant expressions (§15.28) in the `case` labels.

All of the following must be true, or a compile-time error will result:

- Every `case` constant expression associated with a `switch` statement must be assignable (§5.2) to the type of the `switch` *Expression*.
- No switch label is `null`.
- No two of the `case` constant expressions associated with a `switch` statement may have the same value.
- At most one `default` label may be associated with the same `switch` statement.

The prohibition against using null as a switch label prevents one from writing code that can never be executed. If the switch expression is of a reference type, such as a boxed primitive type or an enum, a run-time error will occur if the expression evaluates to null at run-time.

It follows that if the switch expression is of an enum type, the possible values of the switch labels must all be enum constants of that type.

Compilers are encouraged (but not required) to provide a warning if a switch on an enum-valued expression lacks a default case and lacks cases for one or more of the enum type's constants. (Such a statement will silently do nothing if the expression evaluates to one of the missing constants.)

DISCUSSION

In C and C++ the body of a `switch` statement can be a statement and statements with `case` labels do not have to be immediately contained by that statement. Consider the simple loop:

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

where `n` is known to be positive. A trick known as *Duff's device* can be used in C or C++ to unroll the loop, but this is not valid code in the Java programming language:

```
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);
}
```

Fortunately, this trick does not seem to be widely known or used. Moreover, it is less needed nowadays; this sort of code transformation is properly in the province of state-of-the-art optimizing compilers.

When the `switch` statement is executed, first the *Expression* is evaluated. If the *Expression* evaluates to `null`, a `NullPointerException` is thrown an dthe entire `switch` statement completes abruptly for that reason. Otherwise, if the result is of a reference type, it is subject to unboxing conversion (§5.1.8). If evaluation of the *Expression* or the subsequent unboxing conversion (if any) completes abruptly for some reason, the `switch` statement completes abruptly for the same reason. Otherwise, execution continues by comparing the value of the *Expression* with each `case` constant. Then there is a choice:

- If one of the `case` constants is equal to the value of the expression, then we say that the `case` matches, and all statements after the matching `case` label in the switch block, if any, are executed in sequence. If all these statements complete normally, or if there are no statements after the matching `case` label, then the entire `switch` statement completes normally.
- If no `case` matches but there is a `default` label, then all statements after the matching `default` label in the switch block, if any, are executed in sequence.

If all these statements complete normally, or if there are no statements after the `default` label, then the entire `switch` statement completes normally.

- If no `case` matches and there is no `default` label, then no further action is taken and the `switch` statement completes normally.

If any statement immediately contained by the *Block* body of the `switch` statement completes abruptly, it is handled as follows:

- If execution of the *Statement* completes abruptly because of a `break` with no label, no further action is taken and the `switch` statement completes normally.
- If execution of the *Statement* completes abruptly for any other reason, the `switch` statement completes abruptly for the same reason. The case of abrupt completion because of a `break` with a label is handled by the general rule for labeled statements (§14.7).

As in C and C++, execution of statements in a switch block "falls through labels."

For example, the program:

```
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);
    }
}
```

contains a switch block in which the code for each case falls through into the code for the next case. As a result, the program prints:

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

If code is not to fall through case to case in this manner, then `break` statements should be used, as in this example:

```
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
            }
        }
        public static void main(String[] args) {
            howMany(1);
            howMany(2);
            howMany(3);
        }
    }
```

This program prints:

```
one
two
many
```

14.12 The `while` Statement

The `while` statement executes an *Expression* and a *Statement* repeatedly until the value of the *Expression* is `false`.

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

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

The *Expression* must have type `boolean` or `Boolean`, or a compile-time error occurs.

A `while` statement is executed by first evaluating the *Expression*. If the result is of type `Boolean`, it is subject to unboxing conversion (§5.1.8). If evaluation of the *Expression* or the subsequent unboxing conversion (if any) completes abruptly for some reason, the `while` statement completes abruptly for the same reason. Otherwise, execution continues by making a choice based on the resulting value:

- If the value is `true`, then the contained *Statement* is executed. Then there is a choice:
  - If execution of the *Statement* completes normally, then the entire `while` statement is executed again, beginning by re-evaluating the *Expression*.
  - If execution of the *Statement* completes abruptly, see §14.12.1 below.
- If the (possibly unboxed) value of the *Expression* is `false`, no further action is taken and the `while` statement completes normally.

If the (possibly unboxed) value of the *Expression* is `false` the first time it is evaluated, then the *Statement* is not executed.

14.12.1 Abrupt Completion

Abrupt completion of the contained *Statement* is handled in the following manner:

- If execution of the *Statement* completes abruptly because of a `break` with no label, no further action is taken and the `while` statement completes normally.
  - If execution of the *Statement* completes abruptly because of a `continue` with no label, then the entire `while` statement is executed again.
  - If execution of the *Statement* completes abruptly because of a `continue` with label `L`, then there is a choice:
    - If the `while` statement has label `L`, then the entire `while` statement is executed again.
    - If the `while` statement does not have label `L`, the `while` statement completes abruptly because of a `continue` with label `L`.
  - If execution of the *Statement* completes abruptly for any other reason, the `while` statement completes abruptly for the same reason. Note that the case of abrupt completion because of a `break` with a label is handled by the general rule for labeled statements (§14.7).

14.13 The do Statement

"She would not see it," he said at last, curtly,
feeling at first that this statement must do without explanation.
—George Eliot, *Middlemarch* (1871), Chapter 76

The `do` statement executes a *Statement* and an *Expression* repeatedly until the value of the *Expression* is `false`.

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

The *Expression* must have type `boolean` or `Boolean`, or a compile-time error occurs.

A `do` statement is executed by first executing the *Statement*. Then there is a choice:

- If execution of the *Statement* completes normally, then the *Expression* is evaluated. If the result is of type `Boolean`, it is subject to unboxing conversion (§5.1.8). If evaluation of the *Expression* or the subsequent unboxing conversion (if any) completes abruptly for some reason, the `do` statement completes abruptly for the same reason. Otherwise, there is a choice based on the resulting value:
  - If the value is `true`, then the entire `do` statement is executed again.
  - If the value is `false`, no further action is taken and the `do` statement completes normally.
- If execution of the *Statement* completes abruptly, see §14.13.1 below.

Executing a `do` statement always executes the contained *Statement* at least once.

14.13.1 Abrupt Completion

Abrupt completion of the contained *Statement* is handled in the following manner:

- If execution of the *Statement* completes abruptly because of a `break` with no label, then no further action is taken and the `do` statement completes normally.
- If execution of the *Statement* completes abruptly because of a `continue` with no label, then the *Expression* is evaluated. Then there is a choice based on the resulting value:
  - If the value is `true`, then the entire `do` statement is executed again.
  - If the value is `false`, no further action is taken and the `do` statement completes normally.
- If execution of the *Statement* completes abruptly because of a `continue` with label *`L`*, then there is a choice:

  - If the `do` statement has label *L*, then the *Expression* is evaluated. Then there is a choice:
    - If the value of the *Expression* is `true`, then the entire `do` statement is executed again.
    - If the value of the *Expression* is `false`, no further action is taken and the `do` statement completes normally.
  - If the `do` statement does not have label *L*, the `do` statement completes abruptly because of a `continue` with label *L*.
- If execution of the *Statement* completes abruptly for any other reason, the `do` statement completes abruptly for the same reason. The case of abrupt completion because of a `break` with a label is handled by the general rule (§14.7).

14.13.2 Example of do statement

The following code is one possible implementation of the `toHexString` method of class `Integer`:

```
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();
}
```

Because at least one digit must be generated, the `do` statement is an appropriate control structure.

14.14 The for Statement

ForStatement:
    BasicForStatement
    EnhancedForStatement

The `for` statement has two forms:

- The basic `for` statement.
- The enhanced `for` statement

14.14.1 The basic for Statement

The basic for statement executes some initialization code, then executes an *Expression*, a *Statement*, and some update code repeatedly until the value of the *Expression* is `false`.

BasicForStatement:
  `for ( ` *ForInit*$_{opt}$ `;` *Expression*$_{opt}$ `;` *ForUpdate*$_{opt}$ `)` *Statement*

ForStatementNoShortIf:
  `for ( ` *ForInit*$_{opt}$ `;` *Expression*$_{opt}$ `;` *ForUpdate*$_{opt}$ `)` *StatementNoShortIf*

ForInit:
  StatementExpressionList
  LocalVariableDeclaration

ForUpdate:
  StatementExpressionList

StatementExpressionList:
  StatementExpression
  StatementExpressionList `,` *StatementExpression*

The *Expression* must have type `boolean` or `Boolean`, or a compile-time error occurs.

14.14.1.1 *Initialization of `for` statement*

A `for` statement is executed by first executing the *ForInit* code:

- If the *ForInit* code is a list of statement expressions (§14.8), the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If evaluation of any expression completes abruptly for some reason, the `for` statement completes abruptly for the same reason; any *ForInit* statement expressions to the right of the one that completed abruptly are not evaluated.

If the *ForInit* code is a local variable declaration, it is executed as if it were a local variable declaration statement (§14.4) appearing in a block. The scope of a local variable declared in the *ForInit* part of a basic `for` statement (§14.13) includes all of the following:

- Its own initializer
- Any further declarators to the right in the *ForInit* part of the `for` statement
- The *Expression* and *ForUpdate* parts of the `for` statement
- The contained *Statement*

If execution of the local variable declaration completes abruptly for any reason, the `for` statement completes abruptly for the same reason.

- If the *ForInit* part is not present, no action is taken.

14.14.1.2 *Iteration of `for` statement*

Next, a `for` iteration step is performed, as follows:

- If the *Expression* is present, it is evaluated. If the result is of type `Boolean`, it is subject to unboxing conversion (§5.1.8). If evaluation of the *Expression* or the subsequent unboxing conversion (if any) completes abruptly, the `for` statement completes abruptly for the same reason. Otherwise, there is then a choice based on the presence or absence of the *Expression* and the resulting value if the *Expression* is present:
  - If the *Expression* is not present, or it is present and the value resulting from its evaluation (including any possible unboxing) is `true`, then the contained *Statement* is executed. Then there is a choice:
    - If execution of the *Statement* completes normally, then the following two steps are performed in sequence:
      - First, if the *ForUpdate* part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If evaluation of any expression completes abruptly for some reason, the `for` statement completes abruptly for the same reason; any *ForUpdate* statement expressions to the right of the one that completed abruptly are not evaluated. If the *ForUpdate* part is not present, no action is taken.
      - Second, another `for` iteration step is performed.
    - If execution of the *Statement* completes abruptly, see §14.14.1.3 below.

  - If the *Expression* is present and the value resulting from its evaluation (including any possible unboxing) is `false`, no further action is taken and the `for` statement completes normally.

If the (possibly unboxed) value of the *Expression* is `false` the first time it is evaluated, then the *Statement* is not executed.

If the *Expression* is not present, then the only way a `for` statement can complete normally is by use of a `break` statement.

14.14.1.3 *Abrupt Completion of `for` statement*

Abrupt completion of the contained *Statement* is handled in the following manner:

- If execution of the *Statement* completes abruptly because of a `break` with no label, no further action is taken and the `for` statement completes normally.
- If execution of the *Statement* completes abruptly because of a `continue` with no label, then the following two steps are performed in sequence:
  - First, if the *ForUpdate* part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If the *ForUpdate* part is not present, no action is taken.
  - Second, another `for` iteration step is performed.
- If execution of the *Statement* completes abruptly because of a `continue` with label *`L`*, then there is a choice:
  - If the `for` statement has label *`L`*, then the following two steps are performed in sequence:
    - First, if the *ForUpdate* part is present, the expressions are evaluated in sequence from left to right; their values, if any, are discarded. If the *ForUpdate* is not present, no action is taken.
    - Second, another `for` iteration step is performed.
  - If the `for` statement does not have label *`L`*, the `for` statement completes abruptly because of a `continue` with label *`L`*.
- If execution of the *Statement* completes abruptly for any other reason, the `for` statement completes abruptly for the same reason. Note that the case of abrupt completion because of a `break` with a label is handled by the general rule for labeled statements (§14.7).

14.14.2 The enhanced for statement

The enhanced `for` statement has the form:

EnhancedForStatement:
`for (` *VariableModifiers$_{opt}$ Type Identifier*`:` *Expression*`)` *Statement*

The *Expression* must either have type `Iterable` or else it must be of an array type (§10.1), or a compile-time error occurs.

The scope of a local variable declared in the *FormalParameter* part of an enhanced `for` statement (§14.13) is the contained *Statement*

The meaning of the enhanced `for` statement is given by translation into a basic `for` statement.

If the type of *Expression* is a subtype of `Iterable`, then let I be the type of the expression *Expression*`.iterator()`. The enhanced `for` statement is equivalent to a basic `for` statement of the form:

```
for (I #i = Expression.iterator(); #i.hasNext(); ) {
    VariableModifiersopt Type Identifier = #i.next();
    Statement;
}
```

Where *#i* is a compiler-generated identifier that is distinct from any other identifiers (compiler-generated or otherwise) that are in scope (§6.3) at the point where the enhanced `for` statement occurs.

Otherwise, the *Expression* necessarily has an array type, *T[]*. Let L_1 ... L_m be the (possibly empty) sequence of labels immediately preceding the enhanced `for` statement. Then the meaning of the enhanced `for` statement is given by the following basic `for` statement:

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

Where *#a* and *#i* are compiler-generated identifiers that are distinct from any other identifiers (compiler-generated or otherwise) that are in scope at the point where the enhanced `for` statement occurs.

DISCUSSION

The following example, which calculates the sum of an int array, shows how enhanced for works for arrays:

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

Here is an example that combines the enhanced for statement with auto-unboxing to translate a histogram into a frequency table:

```
      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 The `break` Statement

A break statement transfers control out of an enclosing statement.

BreakStatement:
`break` *Identifier*$_{opt}$ `;`

A `break` statement with no label attempts to transfer control to the innermost enclosing `switch`, `while`, `do`, or `for` statement of the immediately enclosing method or initializer block; this statement, which is called the *break target*, then immediately completes normally.

To be precise, a `break` statement with no label always completes abruptly, the reason being a `break` with no label. If no `switch`, `while`, `do`, or `for` statement in the immediately enclosing method, constructor or initializer encloses the `break` statement, a compile-time error occurs.

A `break` statement with label *Identifier* attempts to transfer control to the enclosing labeled statement (§14.7) that has the same *Identifier* as its label; this statement, which is called the *break target*, then immediately completes normally. In this case, the `break` target need not be a `while`, `do`, `for`, or `switch` statement. A `break` statement must refer to a label within the immediately enclosing method or initializer block. There are no non-local jumps.

To be precise, a `break` statement with label *Identifier* always completes abruptly, the reason being a `break` with label *Identifier*. If no labeled statement with *Identifier* as its label encloses the `break` statement, a compile-time error occurs.

It can be seen, then, that a `break` statement always completes abruptly.

The preceding descriptions say "attempts to transfer control" rather than just "transfers control" because if there are any `try` statements (§14.20) within the break target whose `try` blocks contain the `break` statement, then any `finally` clauses of those `try` statements are executed, in order, innermost to outermost, before control is transferred to the break target. Abrupt completion of a `finally` clause can disrupt the transfer of control initiated by a `break` statement.

In the following example, a mathematical graph is represented by an array of arrays. A graph consists of a set of nodes and a set of edges; each edge is an arrow that points from some node to some other node, or from a node to itself. In this example it is assumed that there are no redundant edges; that is, for any two nodes *`P`* and *`Q`*, where *`Q`* may be the same as *`P`*, there is at most one edge from *`P`* to *`Q`*. Nodes are represented by integers, and there is an edge from node *`i`* to node `edges[`*`i`*`][`*`j`*`]` for every *`i`* and *`j`* for which the array reference `edges[`*`i`*`][`*`j`*`]` does not throw an `IndexOutOfBoundsException`.

The task of the method `loseEdges`, given integers *`i`* and *`j`*, is to construct a new graph by copying a given graph but omitting the edge from node *`i`* to node *`j`*, if any, and the edge from node *`j`* to node *`i`*, if any:

```
    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);
        }

    }
```

Note the use of two statement labels, `edgelist` and `search`, and the use of `break` statements. This allows the code that copies a list, omitting one edge, to be shared between two separate tests, the test for an edge from node `i` to node `j`, and the test for an edge from node `j` to node `i`.

14.16 The `continue` Statement

"Your experience has been a most entertaining one," remarked Holmes as his client paused and refreshed his memory with a huge pinch of snuff. "Pray continue your very interesting statement."
—Sir Arthur Conan Doyle, *The Red-headed League* (1891)

A `continue` statement may occur only in a `while`, `do`, or `for` statement; statements of these three kinds are called *iteration statements*. Control passes to the loop-continuation point of an iteration statement.

ContinueStatement:
    `continue` *Identifier*$_{opt}$ `;`

A `continue` statement with no label attempts to transfer control to the innermost enclosing `while`, `do`, or `for` statement of the immediately enclosing method *or initializer block*; this statement, which is called the *continue target*, then immediately ends the current iteration and begins a new one.

To be precise, such a `continue` statement always completes abruptly, the reason being a `continue` with no label. If no `while`, `do`, or `for` statement of the immediately enclosing method or initializer block encloses the `continue` statement, a compile-time error occurs.

A `continue` statement with label *Identifier* attempts to transfer control to the enclosing labeled statement (§14.7) that has the same *Identifier* as its label; that statement, which is called the *continue target*, then immediately ends the current iteration and begins a new one. The continue target must be a `while`, `do`, or `for` statement or a compile-time error occurs. A `continue` statement must refer to a label within the immediately enclosing method or initializer block. There are no non-local jumps.

More precisely, a `continue` statement with label *Identifier* always completes abruptly, the reason being a `continue` with label *Identifier*. If no labeled statement with *Identifier* as its label contains the `continue` statement, a compile-time error occurs.

It can be seen, then, that a `continue` statement always completes abruptly.

See the descriptions of the `while` statement (§14.12), `do` statement (§14.13), and `for` statement (§14.14) for a discussion of the handling of abrupt termination because of `continue`.

The preceding descriptions say "attempts to transfer control" rather than just "transfers control" because if there are any `try` statements (§14.20) within the continue target whose `try` blocks contain the `continue` statement, then any `finally` clauses of those `try` statements are executed, in order, innermost to outermost, before control is transferred to the continue target. Abrupt completion of a `finally` clause can disrupt the transfer of control initiated by a `continue` statement.

In the `Graph` example in the preceding section, one of the `break` statements is used to finish execution of the entire body of the outermost `for` loop. This `break` can be replaced by a `continue` if the `for` loop itself is labeled:

```
class Graph {
    ...
    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) {
                ...
              } else if (k == j) {
                ...
              }
```

```
                newedges[k] = edges[k];
                continue edgelists;
            } // search
            ...
        } // edgelists
        return new Graph(newedges);
    }
}
```

Which to use, if either, is largely a matter of programming style.

14.17 The `return` Statement

"Know you, O judges and people of Helium," he said, "that John Carter, one time Prince of Helium, has returned by his own statement from the Valley Dor . . ."
—Edgar Rice Burroughs, *The Gods of Mars* (1913)

A `return` statement returns control to the invoker of a method (§8.4, §15.12) or constructor (§8.8, §15.9).

ReturnStatement:
  `return` *Expression*$_{opt}$ `;`

A `return` statement with no *Expression* must be contained in the body of a method that is declared, using the keyword `void`, not to return any value (§8.4), or in the body of a constructor (§8.8). A compile-time error occurs if a `return` statement appears within an instance initializer or a static initializer (§8.7). A `return` statement with no *Expression* attempts to transfer control to the invoker of the method or constructor that contains it.

To be precise, a `return` statement with no *Expression* always completes abruptly, the reason being a `return` with no value.

A `return` statement with an *Expression* must be contained in a method declaration that is declared to return a value (§8.4) or a compile-time error occurs. The *Expression* must denote a variable or value of some type `T`, or a compile-time error occurs. The type `T` must be assignable (§5.2) to the declared result type of the method, or a compile-time error occurs.

A `return` statement with an *Expression* attempts to transfer control to the invoker of the method that contains it; the value of the *Expression* becomes the value of the method invocation. More precisely, execution of such a `return` statement first evaluates the *Expression*. If the evaluation of the *Expression* completes abruptly for some reason, then the `return` statement completes abruptly for that reason. If evaluation of the *Expression* completes normally, producing a value `V`, then the `return` statement completes abruptly, the reason being a `return` with value *V*. If the expression is of type `float` and is not FP-strict (§15.4), then the value may be an element of either the float value set or the float-extended-exponent value set (§4.2.3). If the expression is of type `double` and is not FP-strict, then the value may be an element of either the double value set or the double-extended-exponent value set.

It can be seen, then, that a `return` statement always completes abruptly.

The preceding descriptions say "attempts to transfer control" rather than just "transfers control" because if there are any `try` statements (§14.20) within the method or constructor whose `try` blocks contain the `return` statement, then any `finally` clauses of those `try` statements will be executed, in order, innermost to outermost, before control is transferred to the invoker of the method or constructor. Abrupt completion of a `finally` clause can disrupt the transfer of control initiated by a `return` statement.

14.18 The `throw` Statement

A `throw` statement causes an exception (§11) to be thrown. The result is an immediate transfer of control (§11.3) that may exit multiple statements and multiple constructor, instance initializer, static initializer and field initializer evaluations, and method invocations until a `try` statement (§14.20) is found that catches the thrown value. If no such `try` statement is found, then execution of the thread (§17) that executed the `throw` is terminated (§11.3) after invocation of the `uncaughtException` method for the thread group to which the thread belongs.

ThrowStatement:
  `throw` *Expression* `;`

A throw statement can throw an exception type *E* iff the static type of the throw expression is *E* or a subtype of *E*, or the thrown expression can throw *E*.

The *Expression* in a throw statement must denote a variable or value of a reference type which is assignable (§5.2) to the type `Throwable`, or a compile-time error occurs. Moreover, at least one of the following three conditions must be true, or a compile-time error occurs:

- The exception is not a checked exception (§11.2)—specifically, one of the following situations is true:
  - The type of the *Expression* is the class `RuntimeException` or a subclass of `RuntimeException`.
  - The type of the *Expression* is the class `Error` or a subclass of `Error`.

- The `throw` statement is contained in the `try` block of a `try` statement (§14.20) and the type of the *Expression* is assignable (§5.2) to the type of the parameter of at least one `catch` clause of the `try` statement. (In this case we say the thrown value is *caught* by the `try` statement.)
- The `throw` statement is contained in a method or constructor declaration and the type of the *Expression* is assignable (§5.2) to at least one type listed in the `throws` clause (§8.4.4, §8.8.4) of the declaration.

A `throw` statement first evaluates the *Expression*. If the evaluation of the *Expression* completes abruptly for some reason, then the `throw` completes abruptly for that reason. If evaluation of the *Expression* completes normally, producing a non-`null` value *V*, then the `throw` statement completes abruptly, the reason being a `throw` with value *V*. If evaluation of the *Expression* completes normally, producing a `null` value, then an instance *V'* of class `NullPointerException` is created and thrown instead of `null`. The `throw` statement then completes abruptly, the reason being a `throw` with value *V'*.

It can be seen, then, that a `throw` statement always completes abruptly.

If there are any enclosing `try` statements (§14.20) whose `try` blocks contain the `throw` statement, then any `finally` clauses of those `try` statements are executed as control is transferred outward, until the thrown value is caught. Note that abrupt completion of a `finally` clause can disrupt the transfer of control initiated by a `throw` statement.

If a `throw` statement is contained in a method declaration, but its value is not caught by some `try` statement that contains it, then the invocation of the method completes abruptly because of the `throw`.

If a `throw` statement is contained in a constructor declaration, but its value is not caught by some `try` statement that contains it, then the class instance creation expression that invoked the constructor will complete abruptly because of the `throw`.

If a `throw` statement is contained in a static initializer (§8.7), then a compile-time check ensures that either its value is always an unchecked exception or its value is always caught by some `try` statement that contains it. If at run-time, despite this check, the value is not caught by some `try` statement that contains the `throw` statement, then the value is rethrown if it is an instance of class `Error` or one of its subclasses; otherwise, it is wrapped in an `ExceptionInInitializerError` object, which is then thrown (§12.4.2).

If a `throw` statement is contained in an instance initializer (§8.6), then a compile-time check ensures that either its value is always an unchecked exception or its value is always caught by some `try` statement that contains it, or the type of the thrown exception (or one of its superclasses) occurs in the `throws` *clause of every* constructor of the class.

By convention, user-declared throwable types should usually be declared to be subclasses of class `Exception`, which is a subclass of class `Throwable` (§11.5).

14.19 The `synchronized` Statement

A `synchronized` statement acquires a mutual-exclusion lock (§17.13) on behalf of the executing thread, executes a block, then releases the lock. While the executing thread owns the lock, no other thread may acquire the lock.

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

The type of *Expression* must be a reference type, or a compile-time error occurs.

A `synchronized` statement is executed by first evaluating the *Expression*.

If evaluation of the *Expression* completes abruptly for some reason, then the `synchronized` statement completes abruptly for the same reason.

Otherwise, if the value of the *Expression* is `null`, a `NullPointerException` is thrown.

Otherwise, let the non-`null` value of the *Expression* be *V*. The executing thread locks the lock associated with *V*. Then the *Block* is executed. If execution of the *Block* completes normally, then the lock is unlocked and the `synchronized` statement completes normally. If execution of the *Block* completes abruptly for any reason, then the lock is unlocked and the `synchronized` statement then completes abruptly for the same reason.

Acquiring the lock associated with an object does not of itself prevent other threads from accessing fields of the object or invoking unsynchronized methods on the object. Other threads can also use `synchronized` methods or the `synchronized` statement in a conventional manner to achieve mutual exclusion.

The locks acquired by `synchronized` statements are the same as the locks that are acquired implicitly by `synchronized` methods; see §8.4.3.6. A single thread may hold a lock more than once.

The example:

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

prints:

```
made it!
```

This example would deadlock if a single thread were not permitted to lock a lock more than once.

14.20 The `try` statement

These are the times that try men's souls.
—Thomas Paine, *The American Crisis* (1780)

. . . and they all fell to playing the game of catch as catch can,
till the gunpowder ran out at the heels of their boots.
—Samuel Foote

A `try` statement executes a block. If a value is thrown and the `try` statement has one or more `catch` clauses that can catch it, then control will be transferred to the first such `catch` clause. If the `try` statement has a `finally` clause, then another block of code is executed, no matter whether the `try` block completes normally or abruptly, and no matter whether a `catch` clause is first given control.

TryStatement:
  `try` *Block Catches*
  `try` *Block* $Catches_{opt}$ *Finally*

Catches:
  CatchClause
  Catches CatchClause

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

Finally:
  `finally` *Block*

The following is repeated from §8.4.1 to make the presentation here clearer:

FormalParameter:
  VariableModifiers Type VariableDeclaratorId

The following is repeated from §8.3 to make the presentation here clearer:

VariableDeclaratorId:
  Identifier
  VariableDeclaratorId `[ ]`

The *Block* immediately after the keyword `try` is called the `try` block of the `try` statement. The *Block* immediately after the keyword `finally` is called the `finally` block of the `try` statement.

A `try` statement may have `catch` clauses (also called *exception handlers*). A `catch` clause must have exactly one parameter (which is called an *exception parameter*); the declared type of the exception parameter must be the class `Throwable` or a subclass (not just a subtype) of `Throwable`, or a compile-time error occurs.In particular, it is a compile-time error if the declared type of the exception parameter is a type variable (§4.4). The scope of the parameter variable is the *Block* of the `catch` clause.

An exception parameter of a catch clause must not have the same name as a local variable or parameter of the method or initializer block immediately enclosing the catch clause, or a compile-time error occurs.

The scope of a parameter of an exception handler that is declared in a `catch` clause of a `try` statement (§14.19) is the entire block associated with the `catch`. Within the *Block* of the `catch` clause, the name of the parameter may not be redeclared as a local variable of the directly enclosing method or initializer block, nor may it be redeclared as an exception parameter of a catch clause in a try statement of the directly enclosing method or initializer block, or a compile-time error occurs. However, an exception parameter may be shadowed (§6.3.1) anywhere inside a class declaration nested within the *Block* of the `catch` clause.

A `try` statement can throw an exception type *E* iff either:

- The `try` block can throw *E* and *E* is not assignable to any `catch` parameter of the `try` statement and either no `finally` block is present or the `finally` block can complete normally; or
- Some `catch` block of the `try` statement can throw E and either no `finally` block is present or the `finally` block can complete normally; or
- A `finally` block is present and can throw *E*.

It is a compile-time error if an exception parameter that is declared `final` *is* assigned to within the body of the catch clause.

It is a compile-time error if a `catch` clause catches checked exception type *E1* but there exists no checked exception type *E2* such that all of the following hold:

- *E2 <: E1*
- The `try` block corresponding to the `catch` clause can throw *E2*

- No preceding `catch` block of the immediately enclosing `try` statement catches *E2* or a supertype of *E2*.

Exception parameters cannot be referred to using qualified names (§6.6), only by simple names.

Exception handlers are considered in left-to-right order: the earliest possible `catch` clause accepts the exception, receiving as its actual argument the thrown exception object.

A `finally` clause ensures that the `finally` block is executed after the `try` block and any `catch` block that might be executed, no matter how control leaves the `try` block or `catch` block.

Handling of the `finally` block is rather complex, so the two cases of a `try` statement with and without a `finally` block are described separately.

14.20.1 Execution of `try`–`catch`

Our supreme task is the resumption of our onward, normal way.
—Warren G. Harding, Inaugural Address (1921)

A `try` statement without a `finally` block is executed by first executing the `try` block. Then there is a choice:

- If execution of the `try` block completes normally, then no further action is taken and the `try` statement completes normally.
- If execution of the `try` block completes abruptly because of a `throw` of a value *V*, then there is a choice:
  - If the run-time type of *V* is assignable (§5.2) to the *Parameter* of any `catch` clause of the `try` statement, then the first (leftmost) such `catch` clause is selected. The value *V* is assigned to the parameter of the selected `catch` clause, and the *Block* of that `catch` clause is executed. If that block completes normally, then the `try` statement completes normally; if that block completes abruptly for any reason, then the `try` statement completes abruptly for the same reason.
  - If the run-time type of *V* is not assignable to the parameter of any `catch` clause of the `try` statement, then the `try` statement completes abruptly because of a `throw` of the value *V*.
- If execution of the `try` block completes abruptly for any other reason, then the `try` statement completes abruptly for the same reason.

In the example:

```
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("RuntimeException:" + r);
        } catch (BlewIt b) {
            System.out.println("BlewIt");
        }
    }
}
```

the exception `BlewIt` is thrown by the method `blowUp`. The `try—catch` statement in the body of `main` has two `catch` clauses. The run-time type of the exception is `BlewIt` which is not assignable to a variable of type `RuntimeException`, but is assignable to a variable of type `BlewIt`, so the output of the example is:

```
BlewIt
```

14.20.2 Execution of `try-catch-finally`

After the great captains and engineers have accomplish'd their work,
After the noble inventors—after the scientists, the chemist,
the geologist, ethnologist,
Finally shall come the Poet . . .
—Walt Whitman, *Passage to India* (1870)

A `try` statement with a `finally` block is executed by first executing the `try` block. Then there is a choice:

- If execution of the `try` block completes normally, then the `finally` block is executed, and then there is a choice:
  - If the `finally` block completes normally, then the `try` statement completes normally.
  - If the `finally` block completes abruptly for reason *S*, then the `try` statement completes abruptly for reason *S*.

- If execution of the `try` block completes abruptly because of a `throw` of a value *V*, then there is a choice:
  - If the run-time type of *V* is assignable to the parameter of any `catch` clause of the `try` statement, then the first (leftmost) such `catch` clause is selected. The value *V* is assigned to the parameter of the selected `catch` clause, and the *Block* of that `catch` clause is executed. Then there is a choice:
    - If the `catch` block completes normally, then the `finally` block is executed. Then there is a choice:
      - If the `finally` block completes normally, then the `try` statement completes normally.
      - If the `finally` block completes abruptly for any reason, then the `try` statement completes abruptly for the same reason.
    - If the `catch` block completes abruptly for reason *R*, then the `finally` block is executed. Then there is a choice:
      - If the `finally` block completes normally, then the `try` statement completes abruptly for reason *R*.
      - If the `finally` block completes abruptly for reason *S*, then the `try` statement completes abruptly for reason *S* (and reason *R* is discarded).
  - If the run-time type of *V* is not assignable to the parameter of any `catch` clause of the `try` statement, then the `finally` block is executed. Then there is a choice:
    - If the `finally` block completes normally, then the `try` statement completes abruptly because of a `throw` of the value *V*.
    - If the `finally` block completes abruptly for reason *S*, then the `try` statement completes abruptly for reason *S* (and the `throw` of value *V* is discarded and forgotten).
- If execution of the `try` block completes abruptly for any other reason *R*, then the `finally` block is executed. Then there is a choice:
  - If the `finally` block completes normally, then the `try` statement completes abruptly for reason *R*.
  - If the `finally` block completes abruptly for reason *S*, then the `try` statement completes abruptly for reason *S* (and reason *R* is discarded).

The example:

```
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("BlewIt");
      } finally {
         System.out.println("Uncaught Exception");
      }
   }

}
```

produces the output:

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

The `NullPointerException` (which is a kind of `RuntimeException`) that is thrown by method `blowUp` is not caught by the `try` statement in `main`, because a `NullPointerException` is not assignable to a variable of type `BlewIt`. This causes the `finally` clause to execute, after which the thread executing `main`, which is the only thread of the test program, terminates because of an uncaught exception, which typically results in printing the exception name and a simple backtrace. However, a backtrace is not required by this pecification.

DISCUSSION

The problem with mandating a backtrace is that an exception can be created at one point in the program and thrown at a later one. It is prohibitively expensive to store a stack trace in an exception unless it is actually thrown (in which case the trace may be generated while unwinding the stack). Hence we do not mandate a back trace in every exception.

14.21 Unreachable Statements

That looks like a path.
Is that the way to reach the top from here?
—Robert Frost, *The Mountain* (1915)

It is a compile-time error if a statement cannot be executed because it is *unreachable*. Every Java compiler must carry out the conservative flow analysis specified here to make sure all statements are reachable.

This section is devoted to a precise explanation of the word “reachable.” The idea is that there must be some possible execution path from the beginning of the constructor, method, instance initializer or static initializer that contains the statement to the statement itself. The analysis takes into account the structure of statements. Except for the special treatment of `while`, `do`, and `for` statements whose condition expression has the constant value `true`, the values of expressions are not taken into account in the flow analysis.

For example, a Java compiler will accept the code:

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

even though the value of `n` is known at compile time and in principle it can be known at compile time that the assignment to `k` can never be executed.

A Java compiler must operate according to the rules laid out in this section.

The rules in this section define two technical terms:

- whether a statement is *reachable*
- whether a statement *can complete normally*

The definitions here allow a statement to complete normally only if it is reachable.

To shorten the description of the rules, the customary abbreviation “iff” is used to mean “if and only if.”

The rules are as follows:

- The block that is the body of a constructor, method, instance initializer or static initializer is reachable.
- An empty block that is not a switch block can complete normally iff it is reachable. A nonempty block that is not a switch block can complete normally iff the last statement in it can complete normally. The first statement in a nonempty block that is not a switch block is reachable iff the block is reach-

able. Every other statement *S* in a nonempty block that is not a switch block is reachable iff the statement preceding *S* can complete normally.

- A local class declaration statement can complete normally iff it is reachable.
- A local variable declaration statement can complete normally iff it is reachable.
- An empty statement can complete normally iff it is reachable.
- A labeled statement can complete normally if at least one of the following is true:
  - The contained statement can complete normally.
  - There is a reachable `break` statement that exits the labeled statement.

  The contained statement is reachable iff the labeled statement is reachable.
- An expression statement can complete normally iff it is reachable.
- The `if` statement, whether or not it has an `else` part, is handled in an unusual manner. For this reason, it is discussed separately at the end of this section.
- An `assert` statement can complete normally iff it is reachable.
- A `switch` statement can complete normally iff at least one of the following is true:
  - The last statement in the switch block can complete normally.
  - The switch block is empty or contains only switch labels.
  - There is at least one switch label after the last switch block statement group.
  - The switch block does not contain a `default` label.
  - There is a reachable `break` statement that exits the `switch` statement.
- A switch block is reachable iff its `switch` statement is reachable.
- A statement in a switch block is reachable iff its `switch` statement is reachable and at least one of the following is true:
  - It bears a `case` or `default` label.
  - There is a statement preceding it in the `switch` block and that preceding statement can complete normally.
- A `while` statement can complete normally iff at least one of the following is true:

- ◆ The `while` statement is reachable and the condition expression is not a constant expression with value `true`.
- ◆ There is a reachable `break` statement that exits the `while` statement.

The contained statement is reachable iff the `while` statement is reachable and the condition expression is not a constant expression whose value is `false`.

- A `do` statement can complete normally iff at least one of the following is true:
  - ◆ The contained statement can complete normally and the condition expression is not a constant expression with value `true`.
  - ◆ The `do` statement contains a reachable `continue` statement with no label, and the `do` statement is the innermost `while`, `do`, or `for` statement that contains that `continue` statement, and the condition expression is not a constant expression with value `true`.
  - ◆ The `do` statement contains a reachable `continue` statement with a label *`L`*, and the `do` statement has label *`L`*, and the condition expression is not a constant expression with value `true`.
  - ◆ There is a reachable `break` statement that exits the `do` statement.

  The contained statement is reachable iff the `do` statement is reachable.
- A basic `for` statement can complete normally iff at least one of the following is true:
  - ◆ The `for` statement is reachable, there is a condition expression, and the condition expression is not a constant expression with value `true`.
  - ◆ There is a reachable `break` statement that exits the `for` statement.

  The contained statement is reachable iff the `for` statement is reachable and the condition expression is not a constant expression whose value is `false`.
- An enhanced `for` statement can complete normally iff it is reachable.
- A `break`, `continue`, `return`, or `throw` statement cannot complete normally.
- A `synchronized` statement can complete normally iff the contained statement can complete normally. The contained statement is reachable iff the `synchronized` statement is reachable.
- A `try` statement can complete normally iff both of the following are true:
  - ◆ The `try` block can complete normally or any `catch` block can complete `normally`.

    - If the `try` statement has a `finally` block, then the `finally` block can complete normally.
- The `try` block is reachable iff the `try` statement is reachable.
- A `catch` block *`C`* is reachable iff both of the following are true:
    - Some expression or `throw` statement in the `try` block is reachable and can throw an exception whose type is assignable to the parameter of the `catch` clause *`C`*. (An expression is considered reachable iff the innermost statement containing it is reachable.)
    - There is no earlier `catch` block *`A`* in the `try` statement such that the type of *`C`*'s parameter is the same as or a subclass of the type of *`A`*'s parameter.
- If a `finally` block is present, it is reachable iff the `try` statement is reachable.

One might expect the `if` statement to be handled in the following manner, but these are not the rules that the Java programming language actually uses:

- HYPOTHETICAL: An `if-then` statement can complete normally iff at least one of the following is `true`:
    - The `if–then` statement is reachable and the condition expression is not a constant expression whose value is `true`.
    - The `then`–statement can complete normally.
- The `then`–statement is reachable iff the `if–then` statement is reachable and the condition expression is not a constant expression whose value is `false`.
- HYPOTHETICAL: An `if–then–else` statement can complete normally iff the `then`–statement can complete normally or the `else`–statement can complete normally. The `then`-statement is reachable iff the `if–then–else` statement is reachable and the condition expression is not a constant expression whose value is `false`. The `else` statement is reachable iff the `if–then–else` statement is reachable and the condition expression is not a constant expression whose value is `true`.

This approach would be consistent with the treatment of other control structures. However, in order to allow the if statement to be used conveniently for "conditional compilation" purposes, the actual rules differ.

The actual rules for the if statement are as follows:

- ACTUAL: An `if–then` statement can complete normally iff it is reachable. The `then`–statement is reachable iff the `if–then` statement is reachable.
- ACTUAL: An `if–then–else` statement can complete normally iff the `then`–statement can complete normally or the `else`–statement can complete normally. The `then`-statement is reachable iff the `if–then–else` statement is reachable. The `else`-statement is reachable iff the `if–then–else` statement is reachable.

As an example, the following statement results in a compile-time error:

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

because the statement `x=3;` is not reachable; but the superficially similar case:

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

does not result in a compile-time error. An optimizing compiler may realize that the statement `x=3;` will never be executed and may choose to omit the code for that statement from the generated `class` file, but the statement `x=3;` is not regarded as “unreachable” in the technical sense specified here.

The rationale for this differing treatment is to allow programmers to define “flag variables” such as:

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

and then write code such as:

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

The idea is that it should be possible to change the value of `DEBUG` from `false` to `true` or from `true` to `false` and then compile the code correctly with no other changes to the program text.

This ability to “conditionally compile” has a significant impact on, and relationship to, binary compatibility (§13). If a set of classes that use such a “flag” variable are compiled and conditional code is omitted, it does not suffice later to distribute just a new version of the class or interface that contains the definition of the flag. A change to the value of a flag is, therefore, not binary compatible with preexisting binaries (§13.4.8). (There are other reasons for such incompatibility as well, such as the use of constants in `case` labels in `switch` statements; see §13.4.8.)

One ought not to be thrown into confusion
By a plain statement of relationship . . .
—Robert Frost, *The Generations of Men* (1914)