4.4 Control Structures

Normally, statements in a program are executed one after the other in the order in which they're written. This process is called **sequential execution**. Various Java statements, which we'll soon discuss, enable you to specify that the next statement to execute is *not* necessarily the *next* one in sequence. This is called **transfer of control**.

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of much difficulty experienced by software development groups. The blame was pointed at the **goto statement** (used in most programming languages of the time), which allows you to specify a transfer of control to one of a wide range of destinations in a program. [*Note:* Java does *not* have a `goto` statement; however, the word `goto` is *reserved* by Java and should *not* be used as an identifier in programs.]

The research of Bohm and Jacopini[1] demonstrated that programs could be written *without* any `goto` statements. The challenge of the era for programmers was to shift their styles to "`goto`-less programming." The term **structured programming** became almost synonymous with "`goto` elimination." Not until the 1970s did most programmers start taking structured programming seriously. The results were impressive. Software development groups reported shorter development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. The key to these successes was that structured programs were clearer, easier to debug and modify, and more likely to be bug free in the first place.

1. C. Bohm and G. Jacopini, “Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules,” Communications of the ACM, Vol. 9, No. 5, May 1966, pp. 336–371.

Bohm and Jacopini’s work demonstrated that all programs could be written in terms of only three control structures—the **sequence structure**, the **selection structure** and the **iteration structure**. When we introduce Java’s control-structure implementations, we’ll refer to them in the terminology of the *Java Language Specification* as “control statements.”

4.4.1 Sequence Structure in Java

The sequence structure is built into Java. Unless directed otherwise, the computer executes Java statements one after the other in the order in which they’re written—that is, in sequence. The UML **activity diagram** in Fig. 4.1 illustrates a typical sequence structure in which two calculations are performed in order. Java lets you have as many actions as you want in sequence. As we’ll soon see, anywhere a single action may be placed, we may place several actions in sequence.

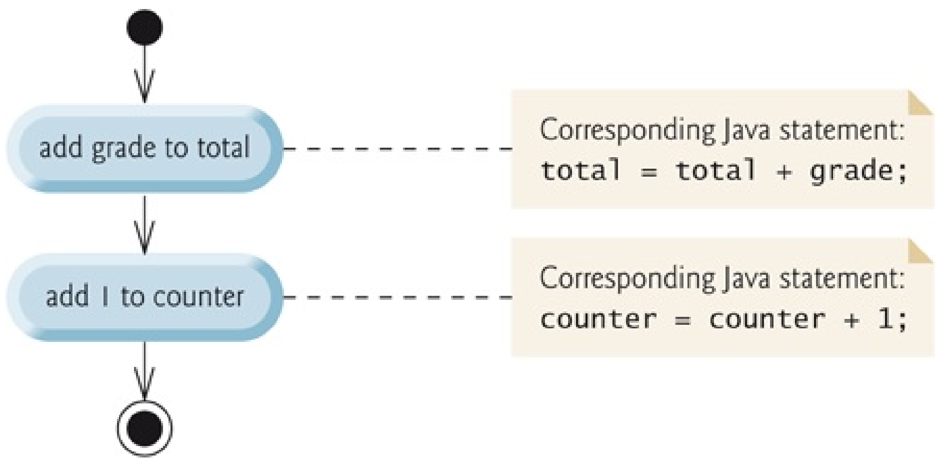

Fig. 4.1

Sequence-structure activity diagram.

A UML activity diagram models the **workflow** (also called the **activity**) of a portion of a software system. Such workflows may include a portion of an algorithm, like the sequence structure in Fig. 4.1. Activity diagrams are composed of symbols, such as **action-state symbols** (rectangles with their left and right sides replaced with outward arcs), **diamonds** and **small circles**. These symbols are connected by **transition arrows**, which represent the *flow of the activity*—that is, the *order* in which the actions should occur.

Like pseudocode, activity diagrams help you develop and represent algorithms. Activity diagrams clearly show how control structures operate. We use the UML in this chapter and Chapter 5 to show control flow in control statements. Online Chapters 33–34 use the UML in a real-world automated-teller-machine case study.

Consider the activity diagram in Fig. 4.1. It contains two **action states**, each containing an **action expression**—“add grade to total” or “add 1 to counter”—that specifies a particular action to perform. Other actions might include calculations or input/output operations. The arrows represent **transitions**, which indicate the order in which the actions represented by the action states occur. The program that implements the activities illustrated by the diagram in Fig. 4.1 first adds `grade` to `total`, then adds `1` to `counter`.

The **solid circle** at the top of the activity diagram represents the **initial state**—the *beginning* of the workflow *before* the program performs the modeled actions. The **solid circle surrounded by a hollow circle** at the bottom of the diagram represents the **final state**—the *end* of the workflow *after* the program performs its actions.

Figure 4.1 also includes rectangles with the upper-right corners folded over. These are UML **notes** (like comments in Java)—explanatory remarks that describe the purpose of symbols in the diagram. Figure 4.1 uses notes to show the Java code associated with each action state. A **dotted line** connects each note with the element it describes. Activity diagrams normally do *not* show the corresponding Java code. We do this here to illustrate how the diagram relates to Java code. For more information on the UML, see our optional online object-oriented design case study (Chapters 33–34) or visit `http://www.uml.org`.

4.4.2 Selection Statements in Java

Java has three types of **selection statements** (discussed in this chapter and Chapter 5). The `if` *statement* either performs (selects) an action, if a condition is *true*, or skips it, if the condition is *false*. The `if…else` *statement* performs an action if a condition is *true* and performs a different action if the condition is *false*. The `switch` *statement* (Chapter 5) performs one of *many* different actions, depending on the value of an expression.

The `if` statement is a **single-selection statement** because it selects or ignores a *single* action (or, as we'll soon see, a *single group of actions*). The `if…else` statement is called a **double-selection statement** because it selects between *two different actions* (or *groups of actions*). The `switch` statement is called a **multiple-selection statement** because it selects among *many different actions* (or *groups of actions*).

4.4.3 Iteration Statements in Java

Java provides four **iteration statements** (also called **repetition statements** or **looping statements**) that enable programs to perform statements repeatedly as long as a condition (called the **loop-continuation condition**) remains *true*. The iteration statements are `while`, `do…while`, `for` and enhanced `for`. (Chapter 5 presents the `do…while` and `for` statements and Chapter 7 presents the enhanced `for` statement.) The `while` and `for` statements perform the action (or group of actions) in their bodies zero or more times —if the loop-continuation condition is initially *false*, the action (or group of actions) will *not* execute. The `do…while` statement performs the action (or group of actions) in its body *one or more* times. The words `if`, `else`, `switch`, `while`, `do` and `for` are Java keywords. A complete list of Java keywords appears in Appendix C.

4.4.4 Summary of Control Statements in Java

Java has only three kinds of control structures, which from this point forward we refer to as *control statements*: the *sequence statement*, *selection statements* (three types) and *iteration statements* (four types). Every program is formed by combining as many of these statements as is appropriate for the algorithm the program implements. We can model each control statement as an activity diagram. Like Fig. 4.1, each diagram contains an initial state and a final state that represent a control statement's entry point and exit point, respectively. **Single-entry/single-exit control statements** make it easy to build programs—we simply connect the exit point of one to the entry point of the next. We call this **control-statement stacking**. We'll learn that there's only one other way in which control statements may be connected—**control-statement**

nesting—in which one control statement appears *inside* another. Thus, algorithms in Java programs are constructed from only three kinds of control statements, combined in only two ways. This is the essence of simplicity.