**UNIVERSIDAD DE LAS FUERZAS ARMADAS – ESPE**

**NAME:** Miquely Calvopiña

**RESUME: CONTROLLING EXECUTION**

Java uses all of C’s execution control statements, most procedural programming languages have some kind of control statements, and there is often overlap among languages. In Java, the keywords include if-else, while, do-while, for, return, break, and a selection statement called switch. Java does not, however, support the much-maligned goto (which can still be the most expedient way to solve certain types of problems). You can still do a goto-like jump, but it is much more constrained than a typical goto.

**true and false**

All conditional statements use the truth or falsehood of a conditional expression to determine the execution path. An example of a conditional expression is a == b. This uses the conditional operator == to see if the value of a is equivalent to the value of b. The expression returns true or false. Any of the relational operators you’ve seen in the previous chapter can be used to produce a conditional statement. Note that Java doesn’t allow you to use a number as a boolean, even though it’s allowed in C and C++ (where truth is nonzero and falsehood is zero). If you want to use a non-boolean in a boolean test, such as if(a), you must first convert it to a boolean value by using a conditional expression, such as if(a != 0).

**if-else**

The if-else statement is the most basic way to control program flow. The else is optional, so you can use if in two forms: if and if-else.

The Boolean-expression must produce a boolean result. The statement is either a simple statement terminated by a semicolon, or a compound statement, which is a group of simple statements enclosed in braces.

Whenever the word “statement” is used, it always implies that the statement can be simple or compound.

**Iteration**

Looping is controlled by while, do-while and for, which are sometimes classified as iteration statements. A statement repeats until the controlling Boolean-expression evaluates to false. The Boolean-expression is evaluated once at the beginning of the loop and again before each further iteration of the statement.

If you print a boolean value, you automatically get the appropriate string “true” or “false.”

**do-while**

The sole difference between while and do-while is that the statement of the do-while always executes at least once, even if the expression evaluates to false the first time. In a while, if the conditional is false the first time the statement never executes. In practice, dowhile is less common than while.

**for**

A for loop is perhaps the most commonly used form of iteration. This loop performs initialization before the first iteration. Then it performs conditional testing and, at the end of each iteration, some form of “stepping.” The expression is tested before each iteration, and as soon as it evaluates to false, execution will continue at the line following the for statement. At the end of each loop, the step executes.

**The comma operator**

It has only one use in Java: in the control expression of a for loop. In both the initialization and step portions of the control expression, you can have a number of statements separated by commas, and those statements will be evaluated sequentially.

The int definition in the for statement covers both i and j. The initialization portion can have any number of definitions of one type. The ability to define variables in a control expression is limited to the for loop. You cannot use this approach with any of the other selection or iteration statements

**Foreach syntax**

Java SE5 introduces a new and more succinct for syntax, for use with arrays and containers (you’ll learn more about these in the Arrays and Containers in Depth chapter). This is often called the foreach syntax, and it means that you don’t have to create an int to count through a sequence of items—the foreach produces each item for you, automatically.

*for(float x : f) {*

This defines a variable x of type float and sequentially assigns each element of f to x.

The foreach syntax not only saves time when typing in code. More importantly, it is far easier to read and says what you are trying to do (get each element of the array) rather than giving the details of how you are doing it.

**return**

Several keywords represent unconditional branching, which simply means that the branch happens without any test. These include return, break, continue, and a way to jump to a labeled statement which is similar to the goto in other languages. The return keyword has two purposes: It specifies what value a method will return (if it doesn’t have a void return value) and it causes the current method to exit, returning that value.

If you do not have a return statement in a method that returns void, there’s an implicit return at the end of that method, so it’s not always necessary to include a return statement. However, if your method states it will return anything other than void, you must ensure every code path will return a value.

**break and continue**

You can also control the flow of the loop inside the body of any of the iteration statements by using break and continue. break quits the loop without executing the rest of the statements in the loop. continue stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration.

**The infamous “goto”**

The goto keyword has been present in programming languages from the beginning. Indeed, goto was the genesis of program control in assembly language: “If condition A, then jump here; otherwise, jump there.” If you read the assembly code that is ultimately generated by virtually any compiler, you’ll see that program control contains many jumps (the Java compiler produces its own “assembly code,” but this code is run by the Java Virtual Machine rather than directly on a hardware CPU). A goto is a jump at the source-code level, and that’s what brought it into disrepute. If a program will always jump from one point to another, isn’t there some way to reorganize the code so the flow of control is not so jumpy? goto fell into true disfavor with the publication of the famous “Goto considered harmful” paper by Edsger Dijkstra, and since then gotobashing has been a popular sport, with advocates of the cast-out keyword scurrying for cover. As is typical in situations like this, the middle ground is the most fruitful. The problem is not the use of goto, but the overuse of goto; in rare situations goto is actually the best way to structure control flow. Although goto is a reserved word in Java, it is not used in the language; Java has no goto. However, it does have something that looks a bit like a jump tied in with the break and continue keywords. It’s not a jump but rather a way to break out of an iteration statement. The reason it’s often thrown in with discussions of goto is because it uses the same mechanism: a label.

It’s important to remember that the only reason to use labels in Java is when you have nested loops and you want to break or continue through more than one nested level.

**switch**

The switch is sometimes called a selection statement. The switch statement selects from among pieces of code based on the value of an integral expression. Integral-selector is an expression that produces an integral value. The switch compares the result of integral-selector to each integral-value. If it finds a match, the corresponding statement (a single statement or multiple statements; braces are not required) executes. If no match occurs, the default statement executes. You will notice in the preceding definition that each case ends with a break, which causes execution to jump to the end of the switch body. This is the conventional way to build a switch statement, but the break is optional. If it is missing, the code for the following case statements executes until a break is encountered. Although you don’t usually want this kind of behavior, it can be useful to an experienced programmer. Note that the last statement, following the default, doesn’t have a break because the execution just falls through to where the break would have taken it anyway. You could put a break at the end of the default statement with no harm if you considered it important for style’s sake.

**RESUME: ACCESS CONTROL**

If you leave a piece of code in a drawer for a while and come back to it, you may see a much better way to do it. This is one of the prime motivations for refactoring, which rewrites working code in order to make it more readable, understandable, and thus maintainable.

There is a tension, however, in this desire to change and improve your code. There are often consumers (client programmers) who rely on some aspect of your code staying the same. So you want to change it; they want it to stay the same. Thus a primary consideration in objectoriented design is to “separate the things that change from the things that stay the same.” This is particularly important for libraries.

Consumers of that library must rely on the part they use, and know that they won’t need to rewrite code if a new version of the library comes out. On the flip side, the library creator must have the freedom to make modifications and improvements with the certainty that the client code won’t be affected by those changes. This can be achieved through convention. For example, the library programmer must agree not to remove existing methods when modifying a class in the library, since that would break the client programmer’s code.

The reverse situation is thornier, however. In the case of a field, how can the library creator know which fields have been accessed by client programmers? This is also true with methods that are only part of the implementation of a class, and not meant to be used directly by the client programmer. What if the library creator wants to rip out an old implementation and put in a new one? Changing any of those members might break a client programmer’s code. Thus the library creator is in a strait jacket and can’t change anything.

To solve this problem, Java provides access specifiers to allow the library creator to say what is available to the client programmer and what is not. The levels of access control from “most access” to “least access” are public, protected, package access (which has no keyword), and private. From the previous paragraph you might think that, as a library designer, you’ll want to keep everything as “private” as possible, and expose only the methods that you want the client programmer to use.

**package: the library unit**

A package contains a group of classes, organized together under a single namespace. For example, there’s a utility library that’s part of the standard Java distribution, organized under the namespace java.util. One of the classes in java.util is called ArrayList.

One way to use an ArrayList is to specify the full name java.util.ArrayList.

When you create a source-code file for Java, it’s commonly called a compilation unit (sometimes a translation unit). Each compilation unit must have a name ending in .java, and inside the compilation unit there can be a public class that must have the same name as the file (including capitalization, but excluding the .java file name extension). There can be only one public class in each compilation unit; otherwise, the compiler will complain. If there are additional classes in that compilation unit, they are hidden from the world outside that package because they’re not public, and they comprise “support” classes for the main public class.

**Code organization**

When you compile a .java file, you get an output file for each class in the .java file. Each output file has the name of a class in the .java file, but with an extension of .class. Thus you can end up with quite a few .class files from a small number of .java files. If you’ve programmed with a compiled language, you might be used to the compiler spitting out an intermediate form (usually an “obj” file) that is then packaged together with others of its kind using a linker (to create an executable file) or a librarian (to create a library). That’s not how Java works. A working program is a bunch of .class files, which can be packaged and compressed into a Java ARchive (JAR) file (using Java’s jar archiver). The Java interpreter is responsible for finding, loading, and interpreting2 these files.

A library is a group of these class files. Each source file usually has a public class and any number of non-public classes, so there’s one public component for each source file. If you want to say that all these components (each in its own separate .java and .class files) belong together, that’s where the package keyword comes in.

**Creating unique package names**

You might observe that, since a package never really gets “packaged” into a single file, a package can be made up of many .class files, and things could get a bit cluttered. To prevent this, a logical thing to do is to place all the .class files for a particular package into a single directory; that is, use the hierarchical file structure of the operating system to your advantage. This is one way that Java references the problem of clutter; you’ll see the other way later when the jar utility is introduced.

Collecting the package files into a single subdirectory solves two other problems: creating unique package names, and finding those classes that might be buried in a directory structure someplace. This is accomplished by encoding the path of the location of the .class file into the name of the package. By convention, the first part of the package name is the reversed Internet domain name of the creator of the class. Since Internet domain names are guaranteed to be unique, if you follow this convention, your package name will be unique and you’ll never have a name clash.

When the compiler encounters the import statement for the simple library, it begins searching at the directories specified by CLASSPATH, looking for subdirectory net/mindview/simple, then seeking the compiled files of the appropriate names (Vector.class for Vector, and List.class for List). Note that both the classes and the desired methods in Vector and List must be public. Setting the CLASSPATH has been such a trial for beginning Java users (it was for me, when I started) that Sun made the JDK in later versions of Java a bit smarter. You’ll find that when you install it, even if you don’t set the CLASSPATH, you’ll be able to compile and run basic Java programs

**Collisions**

What happens if two libraries are imported via ‘\*’ and they include the same names? Since java.util.\* also contains a Vector class, this causes a potential collision. However, as long as you don’t write the code that actually causes the collision, everything is OK—this is good, because otherwise you might end up doing a lot of typing to prevent collisions that would never happen. The collision does occur if you now try to make a Vector: Vector v = new Vector(); Which Vector class does this refer to? The compiler can’t know, and the reader can’t know either. So the compiler complains and forces you to be explicit. If I want the standard Java Vector, for example, I must say: java.util.Vector v = new java.util.Vector(); Since this (along with the CLASSPATH) completely specifies the location of that Vector, there’s no need for the import java.util.\* statement unless I’m using something else from java.util.

**Using imports to change behavior**

A feature that is missing from Java is C’s conditional compilation, which allows you to change a switch and get different behavior without changing any other code. The reason such a feature was left out of Java is probably because it is most often used in C to solve crossplatform issues: Different portions of the code are compiled depending on the target platform. Since Java is intended to be automatically cross-platform, such a feature should not be necessary.

Java access specifiers The Java access specifiers public, protected, and private are placed in front of each definition for each member in your class, whether it’s a field or a method. Each access specifier only controls the access for that particular definition

**Package access**

All the examples before this chapter used no access specifiers. The default access has no keyword, but it is commonly referred to as package access (and sometimes “friendly”). It means that all the other classes in the current package have access to that member, but to all the classes outside of this package, the member appears to be private. Since a compilation unit—a file—can belong only to a single package, all the classes within a single compilation unit are automatically available to each other via package access. Package access allows you to group related classes together in a package so that they can easily interact with each other. When you put classes together in a package, thus granting mutual access to their package-access members, you “own” the code in that package. It makes sense that only code that you own should have package access to other code that you own.

**public: interface access**

When you use the public keyword, it means that the member declaration that immediately follows public is available to everyone, in particular to the client programmer who uses the library.

**private: you can’t touch that!**

The private keyword means that no one can access that member except the class that contains that member, inside methods of that class. Other classes in the same package cannot access private members, so it’s as if you’re even insulating the class against yourself. On the other hand, it’s not unlikely that a package might be created by several people collaborating together, so private allows you to freely change that member without concern that it will affect another class in the same package. The default package access often provides an adequate amount of hiding; remember, a packageaccess member is inaccessible to the client programmer using the class. This is nice, since the default access is the one that you normally use (and the one that you’ll get if you forget to add any access control). Thus, you’ll typically think about access for the members that you explicitly want to make public for the client programmer, and as a result, you might initially think that you won’t use the private keyword very often, since it’s tolerable to get away without it. However, it turns out that the consistent use of private is very important, especially where multithreading is concerned.

**protected: inheritance access**

Understanding the protected access specifier requires a jump ahead. First, you should be aware that you don’t need to understand this section to continue through this book up through inheritance (the Reusing Classes chapter). But for completeness, here is a brief description and example using protected. The protected keyword deals with a concept called inheritance, which takes an existing class— which we refer to as the base class—and adds new members to that class without touching the existing class. You can also change the behavior of existing members of the class.

Access control is often referred to as implementation hiding. Wrapping data and methods within classes in combination with implementation hiding is often called encapsulation.5 The result is a data type with characteristics and behaviors. Access control puts boundaries within a data type for two important reasons. The first is to establish what the client programmers can and can’t use. You can build your internal mechanisms into the structure without worrying that the client programmers will accidentally treat the internals as part of the interface that they should be using. This feeds directly into the second reason, which is to separate the interface from the implementation. If the structure is used in a set of programs, but client programmers can’t do anything but send messages to the public interface, then you are free to change anything that’s not public (e.g., package access, protected, or private) without breaking client code.

**Class access**

In Java, the access specifiers can also be used to determine which classes within a library will be available to the users of that library. If you want a class to be available to a client programmer, you use the public keyword on the entire class definition. This controls whether the client programmer can even create an object of the class.