PROGRAM CONTROL STATEMENTS

Asmaliza Ahzan
IVERSON ASSOCIATES SDN BHD

Program Control Statements

Decision Constructs

Java supports two selection statements: if and switch. These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

The if-else Statement

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement:

if (*condition*) *statement1*;
else *statement2*;

Here, each statement may be a single statement, or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else clause is optional.

The if works like this: If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed.

```
int a, b;
//...
if(a < b) a = 0;
else b = 0;
```

Nested ifs

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else. Here is an example:

```
if(i == 10) {
if(j < 20) a = b;
if(k > 100) c = d; // this if is
else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

As the comments indicate, the final else is not associated with if(j<20) because it is not in the same block (even though it is the nearest if without an else). Rather, the final else is associated with if(i==10). The inner else refers to if(k>100) because it is the closest if within the same block.

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested ifs is the if-elseif ladder. It looks like this:

```
if(condition)
statement;
else if(condition)
statement;
else if(condition)
statement;
.
.
.
else
statement;
```

The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. The final else acts as a default condition; that is, if all other conditional tests fail, then the last else statement is performed. If there is no final else and all other conditions are false, then no action will take place.

The switch Statement

The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements. Here is the general form of a switch statement:

```
switch (expression) {
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
.
.
.
case valueN :
// statement sequence
break;
default:
// default statement sequence
}
```

Nested switch Statements

You can use a switch as part of the statement sequence of an outer switch. This is called a nested switch. Since a switch statement defines its own block, no conflicts arise between the case constants in the inner switch and those in the outer switch. For example, the following fragment is perfectly valid:

```
switch(count) {
      case 1:
            switch(target) { // nested switch
                  case 0:
                        System.out.println("target is zero");
                        break;
                  case 1: // no conflicts with outer switch
                        System.out.println("target is one");
                        break;
            }
            break;
      case 2: // ...
```

Iteration

- Java's iteration statements are for, while, and do-while.
- A loop repeatedly executes the same set of instructions until a termination condition is met.

while

The while loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

while(*condition*) {
// body of loop
}

The condition can be any Boolean expression. The body of the loop will be executed if the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

do-while

The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is:

do {
// body of loop
} while (*condition*);

for

Beginning with JDK 5, there are two forms of the for loop. The first is the traditional form that has been in use since the original version of Java. The second is the newer "for-each" form.

Both types of for loops are discussed here, beginning with the traditional form. Here is the general form of the traditional for statement:

```
for(initialization; condition; iteration) {
// body
}
```

foreach

Unlike some languages, such as C#, that implement a for-each loop by using the keyword foreach, Java adds the for-each capability by enhancing the for statement. The advantage of this approach is that no new keyword is required, and no preexisting code is broken. The for-each style of for is also referred to as the enhanced for loop.

The general form of the for-each version of the for is shown here:

```
for(type itr-var : collection) statement-block
```

Nested Loops

Like all other programming languages, Java allows loops to be nested. That is, one loop may be inside another. For example, here is a program that nests for loops:

```
// Loops may be nested.
class Nested {
	public static void main(String args[]) {
		int i, j;
		for(i=0; i<10; i++) {
			for(j=i; j<10; j++)
				System.out.print(".");
				System.out.println();
		}
	}
}
```

Branching

Java supports three jump statements: break, continue, and return. These statements transfer control to another part of your program.

Use break to exit a loop

By using break, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a break statement is encountered inside a loop, the loop is terminated, and program control resumes at the next statement following the loop. Here is a simple example:

```
// Using break to exit a loop.
class BreakLoop {
	public static void main(String args[]) {
		for(int i=0; i<100; i++) {
			if(i == 10) break; // terminate loop if i is 10
			System.out.println("i: " + i);
		}
		System.out.println("Loop complete.");
	}
}
```

Use break as a form of goto

Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. This usually makes goto-ridden code hard to understand and hard to maintain. There are, however, a few places where the goto is a valuable and legitimate construct for flow control. For example, the goto can be useful when you are exiting from a deeply nested set of loops.

```
// Using break as a civilized form of goto.
class Break {
	public static void main(String args[]) {
		boolean t = true;
		first: {
			second: {
				third: {
					System.out.println("Before the break.");
					if(t) break second; // break out of second
					block
					System.out.println("This won't execute");
				}
				System.out.println("This won't execute");
			}
			System.out.println("This is after second block.");
		}
	}
}
```

Use continue

Sometimes it is useful to force an early iteration of a loop. The continue statement performs such an action.

In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop.

In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

```
// Demonstrate continue.
class Continue {
      public static void main(String args[]) {
            for(int i=0; i<10; i++) {
                  System.out.print(i + " ");
                  if (i%2 == 0) continue;
                  System.out.println("");
            }
      }
}
```

Use return

The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

```
// Demonstrate return.
class Return {
      public static void main(String args[]) {
            boolean t = true;
            System.out.println("Before the return.");
            if(t) return; // return to caller
            System.out.println("This won't execute.");
      }
}
```