THE DO..WHILE STATEMENT IN JAVA

Sometimes it is more convenient to test the continuation condition at the end of a loop, instead of at the beginning, as is done in the `while` loop. The `do..while` statement is very similar to the `while` statement, except that the word "while," along with the condition that it tests, has been moved to the end. The word "do" is added to mark the beginning of the loop. A `do..while` statement has the form

```
do
    statement
while ( boolean-expression );
```

or, since, as usual, the **statement** can be a block,

```
do {
    statements
} while ( boolean-expression );
```

Note the semicolon, ';', at the very end. This semicolon is part of the statement, just as the semicolon at the end of an assignment statement or declaration is part of the statement. Omitting it is a syntax error. (More generally, **every** statement in Java ends either with a semicolon or a right brace, '}'.)

To execute a `do` loop, the computer first executes the body of the loop -- that is, the statement or statements inside the loop -- and then it evaluates the boolean expression. If the value of the expression is `true`, the computer returns to the beginning of the `do` loop and repeats the process; if the value is `false`, it ends the

loop and continues with the next part of the program. Since the condition is not tested until the end of the loop, the body of a `do` loop is always executed at least once.

For example, consider the following pseudocode for a game-playing program. The `do` loop makes sense here instead of a `while` loop because with the `do` loop, you know there will be at least one game. Also, the test that is used at the end of the loop wouldn't even make sense at the beginning:

```
do {
   Play a Game
   Ask user if he wants to play another game
   Read the user's response
} while ( the user's response is yes );
```

Let's convert this into proper Java code. Since I don't want to talk about game playing at the moment, let's say that we have a class named `Checkers`, and that the `Checkers` class contains a static member subroutine named `playGame()` that plays one game of checkers against the user. Then, the pseudocode "Play a game" can be expressed as the subroutine call statement "`Checkers.playGame();`". We need a variable to store the user's response. The *TextIO* class makes it convenient to use a boolean variable to store the answer to a yes/no question. The input function `TextIO.getlnBoolean()` allows the user to enter the value as "yes" or "no". "Yes" is considered to be `true`, and "no" is considered to be `false`. So, the algorithm can be coded as

```
boolean wantsToContinue;  // True if user wants to play
again.
do {
   Checkers.playGame();
```

```
        TextIO.put("Do you want to play again? ");
        wantsToContinue = TextIO.getlnBoolean();
    } while (wantsToContinue == true);
```

When the value of the boolean variable is set to `false`, it is a signal that the loop should end. When a boolean variable is used in this way -- as a signal that is set in one part of the program and tested in another part -- it is sometimes called a flag or flag variable (in the sense of a signal flag).

By the way, a more-than-usually-pedantic programmer would sneer at the test "`while (wantsToContinue == true)`". This test is exactly equivalent to "`while (wantsToContinue)`". Testing whether "`wantsToContinue == true`" is true amounts to the same thing as testing whether "`wantsToContinue`" is true. A little less offensive is an expression of the form "`flag == false`", where `flag` is a boolean variable. The value of "`flag == false`" is exactly the same as the value of "`!flag`", where `!` is the boolean negation operator. So you can write "`while (!flag)`" instead of "`while (flag == false)`", and you can write "`if (!flag)`" instead of "`if (flag == false)`".

Although a `do..while` statement is sometimes more convenient than a `while` statement, having two kinds of loops does not make the language more powerful. Any problem that can be solved using `do..while`loops can also be solved using only `while` statements, and vice versa. In fact, if **doSomething** represents any block of program code, then

```
do {
    doSomething
} while ( boolean-expression );
```

has exactly the same effect as

```
doSomething
while ( boolean-expression ) {
    doSomething
}
```

Similarly,

```
while ( boolean-expression ) {
    doSomething
}
```

can be replaced by

```
if ( boolean-expression ) {
   do {
       doSomething
   } while ( boolean-expression );
}
```

without changing the meaning of the program in any way.

Source : http://math.hws.edu/javanotes/c3/s3.html