JAVA CONTROL STATEMENTS

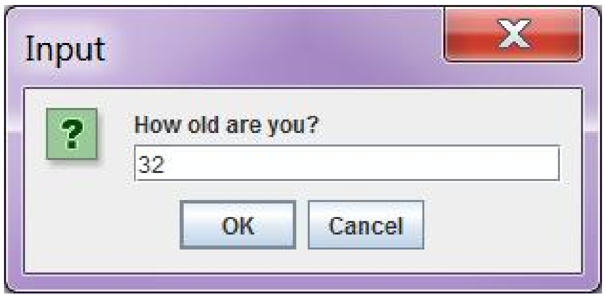
An ***execution path*** is a sequence of statements that the computer executes as it runs a program. A ***control statement*** tells the computer to divert onto a different path. If a program has no control statements, the JVM executes it, statement by statement, in sequential order. Often programmers refer to an execution path as a ***flow of control***.

Example

This application has one execution path through lines 7, 8, 9 and 10.

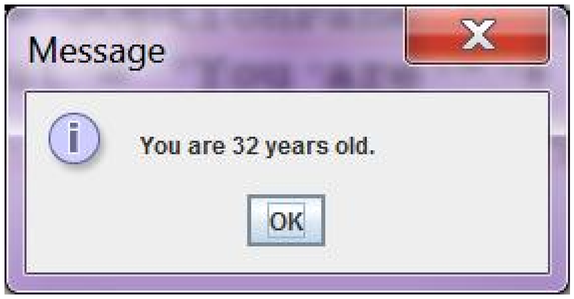
```
 1  import static javax.swing.JOptionPane.*;
 2
 3  public class HowOldAreYou
 4  {
 5     public static void main( String args [ ] )
 6     {
 7        String prompt = "How old are you?";
 8        String age = showInputDialog( prompt );
 9        String output = "You are " + age + " years old.";
10        showMessageDialog( null, output );
11     }
12  }
```

The application uses an input dialog to read the user's age and an output dialog to display it.



How old are you?

32



You are 32 years old.

Method Call

A method call is considered a control statement because it diverts the flow of control to the method's block, even though it doesn't introduce any additional paths in the program.

Example

By following one execution path through lines 5, 13, 6, 13, 7, 18, 8 and 13, the application below outputs:

```
Happy birthday to you
Happy birthday to you
Happy birthday dear, Tom
Happy birthday to you
```

```
 1  public class HappyBirthday
 2  {
 3     public static void main( String args [ ] )
 4     {
 5        printTo( );
 6        printTo( );
 7        printDear( "Tom" );
 8        printTo( );
 9     }
10
11     static void printTo( )
12     {
13        System.out.println( "Happy birthday to you" );
14     }
15
16     static void printDear( String name )
17     {
18        System.out.println( "Happy birthday dear, " + name );
19     }
20  }
```

Loops

A major category of control statements are the ***loops***, which repeatedly execute a group of statements some number of times.

Example

The following code shows a `while` statement, which repeatedly executes lines 2, 4 and 5, stopping when `c` equals 3 (making `c < 3` false). Its output is:

```
c = 1
c = 2
c = 3
```

```
1  int c = 0;
2  while ( c < 3 )
3  {
4     c++;
5     System.out.println( "c = " + c );
6  }
```

Here's a list of Java's loops.

Loop	**Behavior**
`while`	repeat statements 0 or more times until some truth value is false
`do-while`	repeat statements 1 or more times until some truth value is false
`for`	repeat statements and an update clause until some truth value is false

Selection Statements

Another major category of control statements are the ***selection statements***, which choose between groups of statements, deciding which group to execute.

Example

The **`if`** statement, shown on lines 2–9 below, introduces a second execution path into the code. If the value of **`age`** is 21 or more, the computer executes lines 1, 2, 4 and 10. If age is less than 21, the computer executes lines 1, 2, 8 and 10.

```
 1  System.out.print( "You " );
 2  if ( age >= 21 )
 3  {
 4     System.out.print( "can " );
 5  }
 6  else
 7  {
 8     System.out.print( "can't " );
 9  }
10  System.out.println( "purchase liquor!" );
```

Here's a list of Java's selection statements.

Selection Statement	**Behavior**
`if`	choose whether or not to execute a group of statements
`if-else`	choose one of two groups of statements to execute
`switch`	branch to a match point and begin executing from there

Structured Control Statements

You can look at a loop or selection statement in your program and pretty much tell at which points the flow of control enters and exits it. Some programmers refer to them as being ***structured***; others say that they have ***one entrance and one exit***.

Example

Assuming there are no unstructured control statements between lines 2 and 55, execution enters this **`while`** loop at line 1 when **`x`** is positive and exits at line 1 when **`x`** equals 0, after which it proceeds to line 56.

```
 1  while ( x > 0 )
 2  {
 ~     . . .
55  }
56
```

Unstructured Control Statements

Java has several statements that allow you to create more than one exit in your control structures, which earns them the title of ***unstructured control statements***. Here's a list.

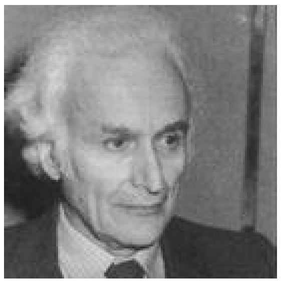
Unstructured Control Statement	**Behavior (roughly described)**
`break`	stop what you're doing and continue after the current loop
`continue`	stop what you're doing and continue with the next loop cycle
`return`	stop what you're doing and go back to calling method
`throw`	stop what you're doing and generate a run-time exception

Example

This loop has two exits: (1) From line 1 to line 56 when **`x`** equals 0. (2) From line 31 to line 56 if **`y`** equals 100.

```
  1  while ( x > 0 )
  2  {
  ~     . . .
 31     if ( y == 100 ) break;
  ~     . . .
 55  }
 56
```

Structured Programming


Corrado Böhm

In 1966, Corrado Böhm and Giuseppe Jacopini published a paper showing that any computer program containing unstructured control statements can be transformed into an equivalent program using only structured control statements. Specifically, any unstructured control statement can be replaced by combinations of the **`while`** and **`if-else`** statements.[1]

Of course, just because you *can* use only two control structures doesn't mean that you *should* do so. Thus, during the decade following Böhm's and Jacopini's paper, computer programmers argued the meaning and merits of ***structured programming*** – What exactly is it? How do you do it? Why should you do it?

Most of this debate has been made irrelevant by the variety of control statements in modern-day programming languages. However, the essential idea of structured programming survives – although a conscientious computer programmer strives to use structured control statements and one-entrance-one-exit code, he or she gives preference to code that (1) is clear to the human reader and (2) reflects the structure of the problem being solved.

Example

We want an application that asks the user multiplication questions – what's 2 X 3, what's 5 X 4, etc. The program must print a question, read the user's answer and check it. If correct, the program must give the user a point and encourage him or her to continue. If the answer is wrong, the program must quit. The user can also signal that he or she wishes to quit by entering -1 for the answer. The complete application is shown below.

The looping construct is not, in the strictest sense, "structured" because it has two exits (from lines 24 and 29) neither of which occurs from line 38, which is where the exit ordinarily should be. Nevertheless, the code clearly expresses the intent of the programmer and the flow of control closely resembles the interaction required by the situation.

```
 1  import java.util.Scanner;
 2
 3  public class TimesDrill
 4  {
 5     public static void main( String args [ ] )
```

[1] Böhm and Jacopini, "Flow diagrams, Turing machines and languages with only two formation rules," *Communications of the ACM*, 9:5 (May 1966), pages 366-371.

```
   {
      int a, b;  // times operands
      int ansR;  // right answer
      int ansU;  // user's answer
      int score; // user's score
      String s;  // output string
      Scanner scanner = new Scanner( System.in );
      System.out.println( "Practice multiplication" );
      System.out.println( "To quit, enter -1" );
      score = 0; // initialize score
      do
      {  // generate question
         a = (int)( Math.random( )*11 );
         b = (int)( Math.random( )*11 );
         s = a + " X " + b + " = ";
         System.out.print( s + "? " );
         ansR = a * b; // compute right answer
         ansU = scanner.nextInt( ); // get user's answer
         if ( ansU == -1 ) // user wants to quit
         {
            System.out.println( "Goodbye" );
            break;  // quit
         }
         if ( ansU != ansR ) // user answered wrong
         {
            System.out.print( "Sorry, " + s + ansR );
            break;  // quit
         }
         // user entered a correct answer so add one to
         // score and encourage continuing
         score++;
         System.out.println(score+" right. Keep going!");
      } while ( true );
   }
}
```