If you have ever learned a foreign language, you know that there comes a point in time when you no longer mimic or consciously translate word for word; your brain just subconsciously gets it and you begin developing communication fluency in the new language. You are on the verge of developing that same capability as a programmer. As you learn more about analyzing problems, designing algorithms, and writing code, you will develop a programmer's perspective about even the most common tasks. Once that happens, the process of programming will become second nature to you and you will gain fluency by attempting more difficult assignments. You may not recognize the emergence of this new perspective until you are doing some routine task that you have done over and over again. Then, out of the blue you realize that you are analyzing the task from a programmer's perspective, designing an algorithm, and trying to write code to solve the problem! For example, **while** loops occur in some of the most unexpected places, as the following example illustrates.

Imagine that you are driving around in your car and notice that it is time to fill up; however, you only have enough money for $10 worth of gas. (You can't get far with that these days!)

The computer operating the pump's shut off switch is broken, consequently you have to watch the meter very carefully to avoid going over the $10 amount.

The meter is initially set at $0.00 and the cost increments by $0.01 while you pump the gas.

The numbers whiz by so you decide to stop when the cost reaches $9.99 in the display.

The accompanying flowchart segment illustrates this situation from a programmer's perspective.

Moving from the flow chart to code, the following program is an example of a **while** loop that stops based on comparing a counter to a terminal value (e.g., $9.99). Enter the program and observe the output.

public class GasPump  
{  
   public static void main(String [] args)  
   {  
     double costOfGas = 0.0;

     while(costOfGas <=9.99)  
     {  
        costOfGas += .01;  
        System.out.println("Cost: " + ((int)(costOfGas \* 100)) / 100.0);  
     }  
   }  
}

Every while loop has three components, as follows:

    while ( condition )  
      body

|  |  |
| --- | --- |
| while | The Java keyword that identifies this piece of code as a while loop. |
| body | **The *loop body*:** a statement or block of statements. This code is executed once during each iteration of the loop. |
| condition | **The *loop condition*:** a boolean expression. When condition is true, the loop body is executed; otherwise it is not. |

Int[] a = { -1, 7, 987, -768, 5, 1024, 15, 0 }

 Drawing inspiration from the code in part (a), we see that we have only to change the inequality in the test used by the conditional statement from < to >. The code to be entered in the space provided is therefore the following:

  int i = 0;  
  int m = a[ 0 ];  
  
  while ( i < a.length )  
  {  
    if ( a[ i ] > m )  
      m = a[ i ];  
    i++;  
  }  
  
  System.out.println( m );

  int[] a = { 8, 5, 9, 1, 8, 2 };    
    int i = 0;   
    int m = a[ 0 ];   
  
    while ( i < a.length )    
    {    
      if ( a[ i ] < m )    
        m = a[ i ];   
      i++;   
    }   
  
    System.out.println( m );

1

i is the place in a.

m is the value that is being hunted for, smallest or biggest.

A positive integer is *prime* if and only if it is greater than 1 and it has exactly two divisors, namely, itself and 1. A positive integer greater than 1 that is not prime is said to be *composite*. The integer 1 is neither prime nor composite.

1. The following code tests whether a positive integer n is prime. Study the code carefully, paying particular attention to the test employed by the conditional statement inside the while loop (this checks whether m is a divisor of n by seeing if the remainder when n is divided by m is 0). Then use this code to determine which of the following positive integers are prime: 5, 93, 101.

    int n = ;   
    boolean isPrime = (n > 1);   
    int m = n - 1;   
  
    while ( isPrime && m > 1 )    
    {   
      if ( n % m == 0 )   
        isPrime = false;   
      m--;   
    }   
  
    if ( isPrime )   
      System.out.println( n + " is prime!" );   
    else if ( n > 1 )   
      System.out.println( n + " is composite!" );   
    else   
      System.out.println( n + " is neither prime nor composite!" );

The program in part (a) can be made more efficient if we use the fact that the only even prime is 2. This implies that, if n is not even, then we do not need to test for even divisors. If n is 15, for example, then we need only test whether any of 13, 11, 9, 7, 5, and 3 is a divisor. Complete the following code to implement this new algorithm:

    int n = ;

boolean isPrime = (n > 1);

if ( n % 2 == 0 )

{

  isPrime = (n == 2);

}

else

{

  int m = n - 2 ;

  while ( isPrime && m > 1 )

  {

    if ( n % m == 0 )

      isPrime = false;

    m -=2;

  }

}

if ( isPrime )

  System.out.println( n + " is prime!" );

else if ( n > 1 )

  System.out.println( n + " is composite!" );

else

  System.out.println( n + " is neither prime nor composite!" );

Every counting loop must include the same three elements.

1. Initialization of the counter variable.
2. Evaluation of a boolean condition.
3. Incrementation of the counter variable.

The while loop places these three elements in separate statements.

In contrast, a **for** loop designed to count from 0 to 99 is illustrated below.

for(int counter = 0; counter <= 99; counter++)  
   System.out.println(counter);

The performance of a counting loop implemented with **while** or **for** is exactly the same, so the user would not be able to tell the difference. However, the syntax of a for loop is much more concise because the counter initialization, the **boolean** test, and the counter incrementation are all combined in a single statement separated by semicolons. In effect, the **for** loop automates the process of counting.

Since the loop only repeats one statement, no curly braces are required. If the block of code within a **for** loop consists of more than one line, it must be enclosed within a pair of opening and closing curly braces.

There are two types of for loop. We deal with the first of these now, and then later discuss a variation called the *for*-each loop.

The program in the Single Stepper below contains an example of a for loop that outputs the integers from 1 through 10. It consists of the keyword for followed by three things, separated by semicolons and enclosed in parentheses, followed in turn by the statement that forms the loop body. The three things that are separated by semicolons are as follows:

* the loop initialization code int i = 1. This is a statement expression, that is, a statement without its terminating semicolon. The corresponding statement (obtained by attaching a semicolon at the end) is executed once as execution of the for loop begins.
* the loop condition i <= 10. This is a boolean expression. If its value is false, then execution of the for loop stops. If the value is true, however, execution of the for loop continues.
* the loop update code i++. This is a statement expression. The corresponding statement (obtained by attaching a semicolon at the end) is executed once after each execution of the loop body.

There are two types of for loop. We deal with the first of these now, and then later discuss a variation called the *for*-each loop.

The program in the Single Stepper below contains an example of a for loop that outputs the integers from 1 through 10. It consists of the keyword for followed by three things, separated by semicolons and enclosed in parentheses, followed in turn by the statement that forms the loop body. The three things that are separated by semicolons are as follows:

* the loop initialization code int i = 1. This is a statement expression, that is, a statement without its terminating semicolon. The corresponding statement (obtained by attaching a semicolon at the end) is executed once as execution of the for loop begins.
* the loop condition i <= 10. This is a boolean expression. If its value is false, then execution of the for loop stops. If the value is true, however, execution of the for loop continues.
* the loop update code i++. This is a statement expression. The corresponding statement (obtained by attaching a semicolon at the end) is executed once after each execution of the loop body.

Warning: Variables that are declared within the initialization code or the update code of a for loop are not visible once execution of the loop has terminated. Thus the following code produces an error; click the **Run** button to see how the error is worded.

* for ( int i = 0 ; i < 3 ; i++ )   
        System.out.println( "Hello!" );   
    
      System.out.println( "i is " + i );
* TC1.java:11: error: cannot find symbol   
    
      System.out.println( "i is " + i );   
    
                                     ^
* To overcome this problem, if you want to refer outside a for loop to a variable that appears in the loop's initialization or update code, then make sure that you declare the variable in question somewhere prior to the loop. The following code, for example, achieves what the code above was intended to do, but without generating an error:
* int i;   
    
      for ( i = 0 ; i < 3 ; i++ )   
        System.out.println( "Hello!" );   
    
      System.out.println( "i is " + i );