In Java programming, not all static methods return a value. It is sometimes the case that our purpose in invoking or applying a method is to cause something to occur while execution is in progress rather than to compute a value for the method to return.

Such incidental occurrences are called *side-effects*. A very common side-effect is for a message to be printed in the output window. To tell the Java compiler that a method does not return a value, we use the keyword void where we would usually announce the return value data type, as in the following example:

 public static void selfish( int n )

{

  for ( int i = 0 ; i < n ; i++ )

    System.out.print( "Me " );

}

    
    selfish( 25 );

Me Me Me Me Me Me Me Me Me Me Me Me Me Me Me Me Me Me Me Me Me Me Me Me Me

Since the selfish method is designed not to return a value, it is no surprise that its definition includes no return statement. It is perfectly acceptable, though, for the definition of a static method with a void return value data type to contain one or more return statements, provided that those statements have nothing following the return keyword apart from a statement-ending semicolon.

The purpose for including such "empty" return statements is to make use of the fact that execution of a method terminates as soon as the Java compiler encounters a return statement. The inclusion of such statements therefore provides a form of program control that we can sometimes turn to our advantage. The following code fragment provides an example:

public static void mySqrt( double d )

{

  if ( d < 0 )

  {

    System.out.println( "d must be non-negative!" );

    return;

  }

  System.out.println( "sqrt(d) is " + Math.sqrt( d ) );

  return;

}

    
    mySqrt(  );

The following static method definition uses the fact that a return statement causes immediate termination of the method to convert an ordinal number to the corresponding counting number. The for loop is interrupted and the method terminates as soon as the input is matched to one of the ordinals.

public static int ordinalToInt( String s )

{

  String[] ordinals = { "first", "second", "third", "fourth", "fifth" };

  for ( int i = 0 ; i < ordinals.length ; i++ )

    if ( s.equals( ordinals[ i ] ) )

      return (i + 1);

  return 0;

}

    
    int t = ordinalToInt(  );   
  
    System.out.println( "t is " + t );

t is 0

In case you were wondering, we inform you that it is possible to define static methods that accept arrays as their arguments and/or that generate an array as their return value. The code fragment below includes an example of a static method strLengths that accepts an array of Strings as its argument and returns an array of ints. (Recall that these data types are specified as String[] and int[], respectively.)

Another point to be noted in this code is the dual use of length. Where we need the length of the argument s — in the initialization of the array lngArray and in the for loop condition — we use the expression s.length because s is an array and the length of an array is a [*property*](http://pages.eimacs.com/eimacs/mainpage?epid=E2267316730&cid=162149) of that array. On the other hand, where we need the length of one of the elements of s — in the body of the for loop that initializes the elements of the array lngArray — we use the expression s[ i ].length() because, for each valid index i, s[ i ] is a string and we obtain the length of a string by invoking its length [*method*](http://pages.eimacs.com/eimacs/mainpage?epid=E204487707&cid=162149).

public static int[] strLengths( String[] s )   
  {   
    int[] lngArray = new int[ s.length ];   
    for ( int i = 0 ; i < s.length ; i++ )   
      lngArray[ i ] = s[ i ].length();   
  
      return lngArray;   
  }   
    
  String[] str = {  };   
  
  int[] a = strLengths( str );   
  
  System.out.println( a[  ] );

4

  public static int leastOf3( int a, int b, int c )   
  {   
    if ( a < b )   
      if ( a < c )   
        return a;   
      else   
        return c;   
    else if ( b < c )   
      return b;   
    else   
      return c;   
  }   
    
  int m = leastOf3(  );   
  
  System.out.println( "The least of these three is " + m );

The least of these three is -1

We can tell that it takes three inputs by observing that its *parameter list*

int a, int b, int c

consists of three data type / formal parameter pairs separated by commas. (This is our first example in which the use of the word "list" in the phrase "parameter list" reflects its everyday meaning.) The order in which the data type / formal parameter pairs appear determines the order in which arguments must be supplied when the method is applied to actual values. In this case, the first argument will be used to replace all the as throughout the method body, the second will replace all the bs, and so on. It is especially important to pay close attention to the order of the arguments when — unlike the situation in this example — they have different data types.

We mention in passing that it is possible to define the leastOf3 method in a simpler fashion using a feature of Java that is not part of the Advanced Placement Computer Science Java subset. Click [here](javascript:secWindow('mainpage?epid=E2153922584&cid=162149&s=2','PopUp',480,420,50,50,'menubar,scrollbars,resizable')) to find out more.

**An Alternative Definition**

In addition to the methods Math.abs, Math.pow, and Math.sqrt that are included in the Advanced Placement Computer Science Java subset, Java also provides the methods Math.max and Math.min. Each of these takes two numerical arguments. Math.max returns the greater of its two arguments, while Math.min returns the lesser of the two. Using Math.min, it is possible to define leastOf3 rather more succinctly, as follows:

  public static int leastOf3( int a, int b, int c )  
  {  
    return Math.min( Math.min( a, b ), c );  
  }