Recursive Definition

Recursion is the process of solving a problem by reducing it to successively smaller versions of itself. Java uses this technique, in which a method or function calls itself to solve some problem. A function or method that uses this technique is called **recursive** or **recursive method**. A recursive method is one (1) whose definition or body contains an invocation of itself. Many programming problems can be solved only by recursion, and some problems that can be solved by other techniques are betters solved by recursion.

The following program introduces an example of a recursive method that returns a value. This method implements the factorial definition using recursion instead of using a `for` loop:

```
public static int factorial (int num) {
        if (num == 0) {
                return 1;
        } else {
                return num * factorial(num - 1); //recursive call
        }
}
```

When the following statements of the `factorial` function is invoked in the main method, the return value will be 24.

```
int x = factorial(4);
System.out.println(x);
```

Figure 1 illustrates the execution of the recursive calls for the statement, `factorial(4);` and shows how the parameter, `num`, is passed from one (1) method to the next.

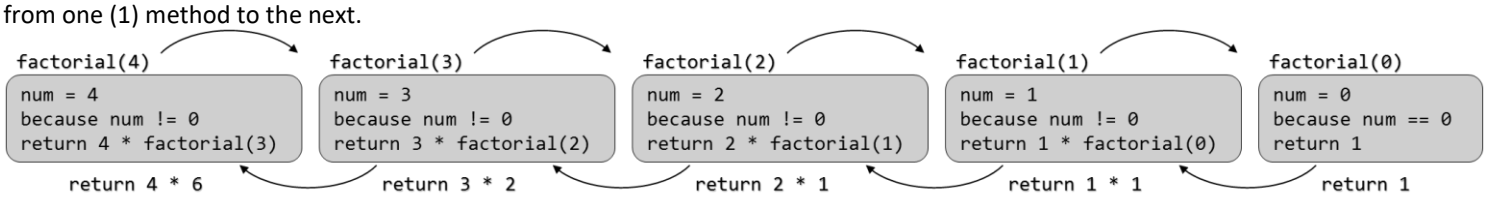

Figure 1. Execution of factorial(4); (Malik, 2012)

Recursion is possible because every call to `factorial` function results in a distinct variable `num` that is local to the function for which it is invoked. When an invocation of a recursive method executes, Java does nothing special to handle this. The argument values are copied to parameters, and the code in the method executes. For example, in the statement `factorial(3)`, the value of the argument is copied into the method's `num` parameter and the resulting code executes.

The following example program defines a void function with a recursive call. This program prints numbers from *n* to 1:

```
public static void countDown (int num) {
        if (num <= 0) {
                System.out.println("End");
        } else {
                System.out.print(num + " ");
                countDown(num - 1); //recursive call
        }
}
```

The following is the output, when the statement, `countDown(4);` is invoked in the main method:

```
4 3 2 1 End
```

Figure 2 illustrates the execution of the recursive calls for the statement, `countDown(4);` and shows how the `num` parameter is passed from one (1) method to the next.

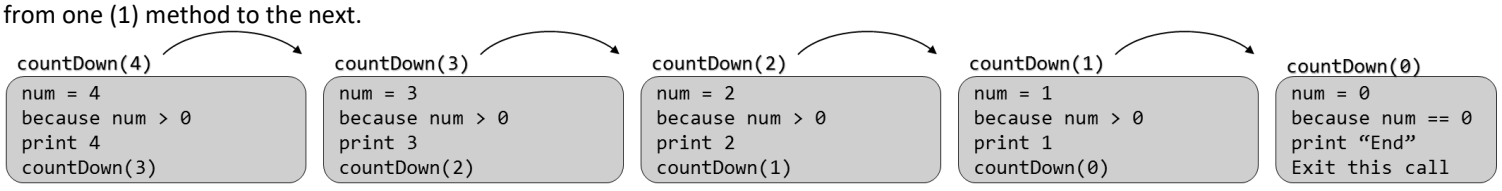

Figure 2. Recursive calls for the countDown() function

The following example program shows the other version of the void `countDown()` function, but this program prints numbers from 1 to *n*. Notice that the recursive call executes first before printing the result:

```
public static void counting (int num) {
        if (num <= 0) {
                System.out.println("Start");
        } else {
                counting(num - 1); //execute recursive call first before printing
                System.out.print(num + " ");
        }
}
```

The following is the output when the statements of `counting(4);` function is invoked in the main method:

```
Start
1 2 3 4
```

Figure 3 illustrates the execution of the recursive calls for the statement `counting(4);`.

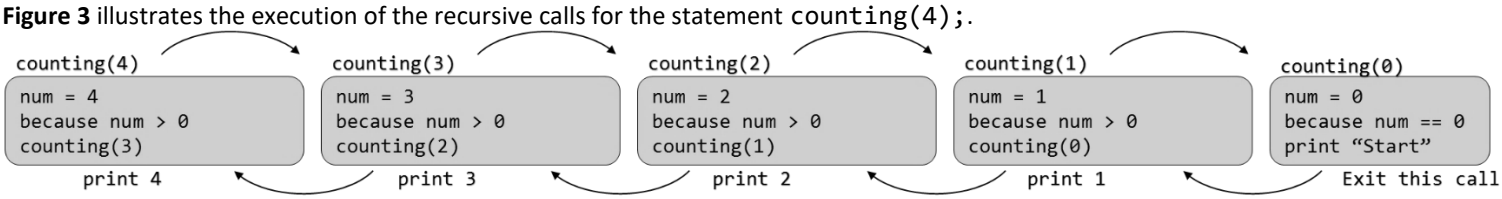

Figure 3. Recursive calls for the counting() function

Typically, all recursive methods must use a selection control structure statement that leads to different cases: the statements with the recursive call is called **general case** and the statements that stop the recursion to avoid infinite recursion is called **base case**. Each recursive call reduces the problem to a smaller version of itself. Every recursive definition must have one (1) or more base cases.

Problem Solving Using Recursion

Recursive algorithms are implemented in Java to develop recursive methods to solve problems. When designing recursive methods, you must make sure that every recursive call eventually reduces to a base case.

The following should be considered when designing a recursive method:

1. Understand the problem requirements.
2. Determine the limiting conditions.
3. Identify the base cases and provide a direct solution to each base case.
4. Identify the general cases and provide a solution to each general case in terms of a smaller version of itself.

The following examples illustrate how recursive algorithms are developed and implemented in Java using recursive methods:

Example 1:

This example uses a recursive algorithm to find the largest element in an array. Consider the following statement:

```
int[] intArray = { 5, 10, 12, 8 };
```

The following Java function finds the largest element in an array and return it:

```
public static int largest (int[] list, int startIndex, int lastIndex) {
	int max;
	if (startIndex == lastIndex) {
		return list[startIndex];
	} else {
		max = largest(list, startIndex + 1, lastIndex);
		if (list[startIndex] >= max) {
			return list[startIndex];
		} else {
			return max;
		}
	}
}
```

When the `largest()` function is invoked in the main method, the return value will be 12.

```
int num = largest(intArray, 0, 3);
System.out.println("The largest element in the array is " + num);
```

Example 2:

This example converts decimal numbers to binary:

```
public static void decToBin (int num) {
	if (num == 0) {
		System.out.print(0);
	} else if (num > 0) {
		decToBin(num / 2);
		System.out.print(num % 2);
	}
}
```

The following is the output when the statement, `decToBin(13);` is invoked in the main method:

```
01101
```

Recursion versus Iteration

Often there are two (2) ways to solve a particular problem: **recursion** or **iteration**, and the key factor in determining the best solution is efficiency. Recursive methods are methods with a recursive call. A method that implements iteration is called an **iterative method**.

The following program is an example of an iterative method version of the function `factorial()` to find the largest element in an array:

```
public static int factorial (int num) {
       int fact = 1;
       for (int i = 0; i < num; i++) {
             fact = fact * i;
       }
       return fact;
}
```

A recursive method uses more storage space than an iterative version of the method because of the overhead to the system that results from tracking the recursive calls and suspended computations. This overhead can make a recursive method run slower than a corresponding iterative one.

When a non-recursive method is called, memory space for its formal parameters and local variables are allocated. When the method terminates, that memory space is then deallocated.

In a recursive method, every call also had its own set of parameters and local variables. That is, every call required that the system allocate memory space for its formal parameters and local variables and then deallocate the memory space when the method exited. Thus overhead is associated with executing a recursive method, both in terms of memory space and computer time. On slower computers, specifically those with limited memory space, the slow execution of a recursive method would be noticeable.

In other situations, recursion is perfectly acceptable and can make a program easier to understand. There are times when recursion can be a big aid in terms of program clarity.

The choice between iteration or recursion depends on the nature of the problem. For problems such as control systems, efficiency is absolutely critical, therefore, the efficiency factor dictates the solution method.

REFERENCES:

Baesens, B., Backiel, A., & Broucke, S. (2015). *Beginning java programming: The object-oriented approach*. Indiana: John Wiley & Sons, Inc.

Farrell, J. (2014). *Java programming, 7th edition*. Boston: Course Technology, Cengage Learning.

Malik, D.S. (2012). *Java programming: from problem solving to program design, 5th edition*. Boston: Cengage Learning.

Savitch, W. (2014). *Java: An introduction to problem solving and programming, 7th edition*. California: Pearson Education, Inc.