Overview

Recursion allows us to achieve a form of de facto looping by exploiting the program stack. Since stacks are naturally recursive data structures, we can use this to our advantage to utilize recursion.

Simply put, recursion is when a method calls itself. All forms of looping, iterative or recursive, must contain three properties to behave properly:

- Starting condition
- Ending condition
- Step condition

Recursion is no different. The initial call of a recursive method is the starting condition. The recursive call of the method within itself is the step condition.

The ending condition is a conditional statement in the recursive method that prevents a Stack Overflow (correction).

Stack Overflow vs. Infinite Loop

I had not made this clear before but I had it pointed out to me by a student that this was confusing.

An infinite loop happens when an iterative style of looping has no step condition. Consider the following code:

```
int i = 0;

while(i < 10){

	System.out.println(i);

}
```

This is an "infinite loop" because it can go on forever. Now, consider a recursive example:

```
// Inside main method...

int max = 10;

recurse(max);

// The recurse wrapper method

public static void recurse(int max){

	recurse_helper(0, max);

}

// The recurse helper method

public static void recurse_helper(int i, int max){

	if(i > max)

		return;

	System.out.println(i);

	recurse_helper(i, max); // <- This does NOT add a step condition (or reach an ending condition)

	// Should be recurse_helper(++i, max);

}
```

The difference is that the recursive call continues to put more and more methods on the program stack (stack space/stack memory) without removing (returning) anything. This will grow the program stack until space runs out causing a Stack Overflow. This is the stack equivalent to the Heap Space Exception (Out of Memory).

This should make everything clear.

Example

Let's look at a trivial example of a recursive method that prints from starting "i" to "n":

```
public static void recursivePrint(int i, int n){
    if(i > n)
        return;
    System.out.println(i);
    recursivePrint(i+1, n);
}
```

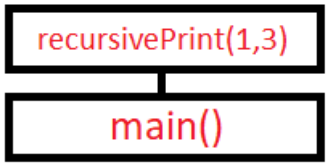
Assume we called this method initially as follows:

```
public static void main(String[] args){
    recursivePrint(1, 3);
}
```

Program Stack View

Looking at the program stack for the previous example, we start with the main() method:

Once the "recursivePrint" method is called initially, the program stack looks as follows:



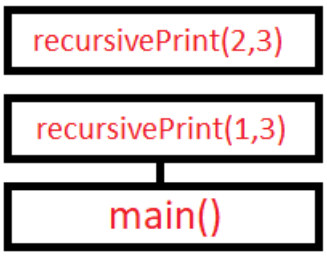
Once called, the recursivePrint method executes by checking for the ending condition (i > n) but since 1 is not greater than 3, the code skips the return statement.

The method then prints the current value of "i" (1) and calls another copy of itself. This is the step condition. What makes it a step is the fact that we are incrementing the value of "i" inside of the method call.

This accomplishes the same objective as a while loop that does "i++;"

Program Stack View II

After the recursive call, the program stack looks as follows:



This pattern continues until recursivePrint(4, 3) is called. Then the ending condition (i > n) is triggered and all of the recursivePrint methods are popped off of the program stack.

We return to the main method at the point after the recursivePrint method call.

Considerations

Due to the overhead of method calls on the program stack, iterative versions of loops (while, do/while, for) are often faster. That being said, recursion can make sense logically for certain situations.

Binary tree traversal, for example, is easier to accomplish with recursion due to the recursive structure of the tree (left - middle - right for inorder traversal).

Efficient vs Inefficient

Even with recursion, not all recursive algorithms are created equal. When solving the Fibonacci problem recursively, the classic method works but is inefficient:

```
public static int fib(int n){
    if(n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

This version does redundant work and is an exponential time algorithm. If we were to solve this iteratively, we could do it in linear time:

```
public static int fib(int n){
    int n1 = 0;
    int n2 = 0;
    int n3 = 1;
    for(int i = 1; i < n; i++){
        n1 = n2;
        n2 = n3;
        n3 = n1 + n2;
    }
    return n3;
}
```

Therefore, the iterative solution is preferable to the recursive version.

Converting Iterative Solutions to Efficient Recursive Solutions

Just because the iterative version is better does not mean that we cannot convert this concept over to recursion. We can think of each recursive call as a single iteration in a iterative loop. Therefore, we could write a recursive method to run in linear time as such:

```
public static int fib(int n, int n1, int n2, int n3, int i){
    if(i == n)
        return n3;
    return fib(n, n2, n3, n2+n3, i+1);
}
```

The problem with this algorithm is that it is ugly and overly cumbersome for a user who wants the simplicity of the previous versions. We can solve this easily using overloaded methods and the concept of wrapper/helper methods...

Wrapper / Helper Methods

Using the previous method for the efficient recursive call, we can make that method a "helper method" (also referred to as a "workhorse method") and make the privacy level "private" so the implementation details are hidden from the user (e.g. "Abstraction")

```
private static int fib(int n, int n1, int n2, int n3, int i){
    if(i == n)
        return n3;
    return fib(n, n2, n3, n2+n3, i+1);
}
```

To still give our users the simplicity of other implementations, we can create another version of the method called a "wrapper method". In the context of recursion, these can be referred to as "recursion initiator methods". Here is our public wrapper method that gives our users an easy way to implement our routines:

```
public static int fib(int n){
    return fib(n, 0, 0, 1, 1);
}
```

Now our users have the best of all worlds...a recursive solution that is efficient and easy to use. Thanks to Abstraction and Method Overloading, they are none the wiser of the implementation details behind the scenes.