Lecture 35
Sometimes you can find a recursive solution by slightly changing the original problem. This new method is called a recursive helper method.

The recursive `isPalindrome` method in Listing 20.3 is not efficient, because it creates a new string for every recursive call. To avoid creating new strings, you can use the low and high indices to indicate the range of the substring. These two indices must be passed to the recursive method. Since the original method is `isPalindrome(String s)`, you have to create the new method `isPalindrome(String s, int low, int high)` to accept additional information on the string, as shown in Listing 20.4.
```java
public class RecursivePalindrome {

    public static boolean isPalindrome(String s) {
        return isPalindrome(s, 0, s.length() - 1);
    }

    private static boolean isPalindrome(String s, int low, int high) {
        if (high <= low) // Base case
            return true;
        else if (s.charAt(low) != s.charAt(high)) // Base case
            return false;
        else
            return isPalindrome(s, low + 1, high - 1);
    }

    public static void main(String[] args) {
        System.out.println("Is moon a palindrome? " + isPalindrome("moon"));
        System.out.println("Is noon a palindrome? " + isPalindrome("noon"));
        System.out.println("Is a a palindrome? " + isPalindrome("a"));
        System.out.println("Is aba a palindrome? " + isPalindrome("aba"));
        System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
    }
}
```
Recursion vs. Iteration

Recursion is an alternative form of program control. It is essentially repetition without a loop.

When you use loops, you specify a loop body. The repetition of the loop body is controlled by the loop control structure. In recursion, the method itself is called repeatedly. A selection statement must be used to control whether to call the method recursively or not.

Recursion bears substantial overhead. Each time the program calls a method, the system must allocate memory for all of the method’s local variables and parameters. This can consume considerable memory and requires extra time to manage the memory.

Any problem that can be solved recursively can be solved nonrecursively with iterations. Recursion has some negative aspects: it uses up too much time and too much memory. Why, then, should you use it? In some cases, using recursion enables you to specify a clear, simple solution for an inherently recursive problem that would otherwise be difficult to obtain. Examples are the directory-size problem, the Towers of Hanoi problem, and the fractal problem, which are rather difficult to solve without using recursion.

The decision whether to use recursion or iteration should be based on the nature of, and your understanding of, the problem you are trying to solve. The rule of thumb is to use whichever approach can best develop an intuitive solution that naturally mirrors the problem. If an iterative solution is obvious, use it. It will generally be more efficient than the recursive option.
**Note**  
Recursive programs can run out of memory, causing a `StackOverflowError`.

**Tip**  
If you are concerned about your program’s performance, avoid using recursion, because it takes more time and consumes more memory than iteration. In general, recursion can be used to solve the inherent recursive problems such as Towers of Hanoi, recursive directories, and Sierpinski triangles.
Tail Recursion

A tail recursive method is efficient for reducing stack size.

A recursive method is said to be *tail recursive* if there are no pending operations to be performed on return from a recursive call, as illustrated in Figure 20.11a. However, method B in Figure 20.11b is not tail recursive because there are pending operations after a method call is returned.

![Diagram](image)

**Figure 20.11** A tail-recursive method has no pending operations after a recursive call.
Tail recursion may be desirable: because the method ends when the last recursive call ends, there is no need to store the intermediate calls in the stack. Some compilers can optimize tail recursion to reduce stack size.

```
Listing 20.10  ComputeFactorialTailRecursion.java

1    public class ComputeFactorialTailRecursion {
2        /** Return the factorial for a specified number */
3        public static long factorial(int n) {
4            return factorial(n, 1);  // Call auxiliary method
5        }
6
7        /** Auxiliary tail-recursive method for factorial */
8        private static long factorial(int n, int result) {
9            if (n == 0)
10                return result;
11            else
12                return factorial(n - 1, n * result);  // Recursive call
13        }
14    }
```