Chapter 19

Recursion

Goals

- Trace recursive algorithms
- Implement recursive algorithms

19.1 Simple Recursion

One day, an instructor was having difficulties with a classroom's multimedia equipment. The bell rang, and still there was no way to project the lecture notes for the day. To keep her students occupied while waiting for the AV folks, she asked one of her favorite students, Kelly, to take attendance. Since the topic of the day was recursion, the instructor proposed a recursive solution: Instead of counting each student in the class, Kelly could count the number of students in her row and remember that count. The instructor asked Kelly to ask another student in the row behind her to do the same thing—count the number of students in their row, remember the count, and ask the same question of the next row.

By the time the question reached the last row of seats in the room, there was one person in each row who knew the number of students in that particular row. Andy, who had just counted eight students in the last row, asked his instructor what to do since there were no more rows of seats behind him. The teacher responded that all he had to do was return the number of students in his row to the person who asked him the question moments ago. So, Andy gave his answer of eight to the person in the row in front of him.

The student in the second to last row added the number of people in her row (12) to the number of students in the row behind her, which Andy had just told her (8). She returned the sum of 20 to the person in the row in front of her.

At that point the AV folks arrived and discovered a bent pin in a video jack. As they were fixing this, the students continued to return the number of students in their row plus the number of students behind them, until Kelly was informed that there were 50 students in all the rows behind her. At that point, the lecture notes, entitled “Recursion”, were visible on the screen. Kelly told her teacher that there were 50 students behind her, plus 12 students in her first row, for a total of 62 students present.

The teacher adapted her lecture. She began by writing the algorithm for the head count problem. Every row got this same algorithm.

```
if you have rows behind you
    return the number of students in your row plus the number behind you
otherwise
    return the number of students in your row
```

Andy asked why Kelly couldn’t have just counted the students one by one. The teacher responded, “That would be an *iterative* solution. Instead, you just solved a problem using a *recursive* solution. This is precisely how I intend to introduce recursion to you, by comparing recursive solutions to problems that could also be solved with iteration. I will suggest to you that some problems are better handled by a recursive solution.”

Recursive solutions have a final situation when nothing more needs to be done—this is the base case—and situations when the same thing needs to be done again while bringing the problem closer to a base case.

Recursion involves partitioning problems into simpler subproblems. It requires that each subproblem be identical in structure to the original problem.

Before looking at some recursive Java methods, consider a few more examples of recursive solutions and definitions. Recursive definitions define something by using that something as part of the definition.

Recursion Example 1

Look up a word in a dictionary:

```
find the word in the dictionary
if there is a word in the definition that you do not understand
    look up that word in the dictionary
```

Example: Look up the term **object**

Look up **object**, which is defined as “an instance of a **class**.”

What is a class? *Look up* **class** to find “a collection of **method**s and data.”

What is a method? *Look up* **method** to find “a **method heading** followed by a collection of programming statements.”

Example: Look up the term **method heading**

What is a method heading? *Look up* **method heading** to find “the name of a method, its **return type**, followed by a **parameter list** in parentheses.”

What is a parameter list? *Look up* **parameter list** to find “a **list** of **parameters**.”
Look up **list**, *look up* **parameters**, and *look up* **return type**, and you finally get a definition of all of the terms using the same method you used to *look up* the original term. And then, when all new terms are defined, you have a definition for **object**.

Recursion Example 2

A definition of a *queue*:

empty
or has one thing at the front of the queue followed by a *queue*

Recursion Example 3

An arithmetic expression is defined as one of these:

a numeric constant such as 123 or –0.001
or a numeric variable that stores a numeric constant
or an *arithmetic expression* enclosed in parentheses
or an *arithmetic expression* followed by a binary operator (`+`, `−`, `/`, `%`, or `*`)
followed by an *arithmetic expression*

Characteristics of Recursion

A **recursive definition** is a definition that includes a simpler version of itself. One example of a recursive definition is given next: the power method that raises an integer (x) to an integer power (n). This definition is recursive because x^{n-1} is part of the definition itself. For example,

$4^3 = 4 \times 4^{(n-1)} = 4 \times 4^{(3-1)} = 4 \times 4^2$

What is 4^2? Using the recursive definition above, 4^2 is defined as:

$4^2 = 4 \times 4^{(n-1)} = 4 \times 4^{(2-1)} = 4 \times 4^1$

and 4^1 is defined as

$$4^1 = 4 \times 4^{(n-1)} = 4 \times 4^{(1-1)} = 4 \times 4^0$$

and 4^0 is a base case defined as

$$4^0 = 1$$

The recursive definition of 4^3 includes 3 recursive definitions. The base case is `n==0`:
$x^n = 1$ if $n = 0$

To get the actual value of 4^3, work backward and let 1 replace 4^0, 4 * 1 replace 4^1, 4 * 4^1 replace 4^2, and 4 * 4^2 replace 4^3. Therefore, 4^3 is defined as 64.

To be recursive, an algorithm or method requires at least one recursive case and at least one base case. The recursive algorithm for power illustrates the characteristics of a recursive solution to a problem.

- The problem can be decomposed into a simpler version of itself in order to bring the problem closer to a base case.
- There is at least one base case that does not make a recursive call.
- The partial solutions are managed in such a way that all occurrences of the recursive and base cases can communicate their partial solutions to the proper locations (values are returned).

Comparing Iterative and Recursive Solutions

For many problems involving repetition, a recursive solution exists. For example, an iterative solution is shown below along with a recursive solution in the `TestPowFunctions` class, with the methods `powLoop` and `powRecurse`, respectively. First, a unit test shows calls to both methods, with the same arguments and same expected results.

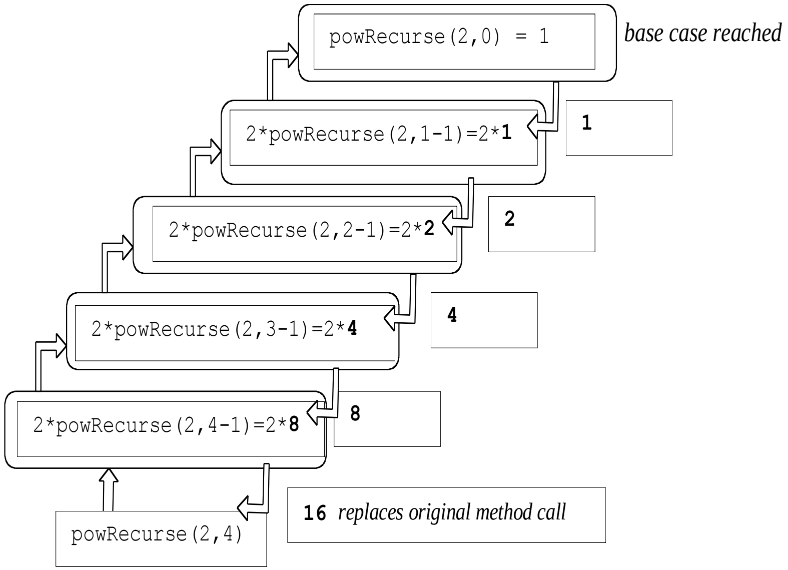
```java
// File RecursiveMethodsTest.java
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class RecursiveMethodsTest {

  @Test
  public void testPowLoop() {
    RecursiveMethods rf = new RecursiveMethods();
    assertEquals(1, rf.powLoop(4, 0));
    assertEquals(1, rf.powRecurse(4, 0));
    assertEquals(4, rf.powLoop(4, 1));
    assertEquals(4, rf.powRecurse(4, 1));
    assertEquals(16, rf.powLoop(2, 4));
    assertEquals(16, rf.powRecurse(2, 4));
  }
}

// File RecursiveMethods.java
public class RecursiveMethods {

  public int powLoop(int base, int power) {
    int result;
    if (power == 0)
      result = 1;
    else {
      result = base;
      for (int j = 2; j <= power; j++)
        result = result * base;
    }
    return result;
  }
  public int powRecurse(int base, int power) {
    if (power == 0)
      return 1;
    else
                       // Make a recursive call \\
      return base * powRecurse(base, power - 1);
  }
}
```

In `powRecurse`, if n is 0—the base case—the method call evaluates to 1. When n > 0—the recursive case—the method is invoked again with the argument reduced by one. For example, `powRecurse(4,1)` calls `powRecurse(4, 1-1)`, which immediately returns 1. For another example, the original call `powRecurse(2,4)` calls `powRecurse(2,3)`, which then calls `powRecurse(2,2)`, which then calls `powRecurse(2,1)`, which then calls `powRecurse(2,0)`, which returns 1. Then, `2*powRecurse(2,0)` evaluates to 2*1, or 2, so `2*powRecurse(2,1)` evaluates to 4, `2*powRecurse(2,2)` evaluates to 8, `2*powRecurse(2,3)` evaluates to 16, and `2*powRecurse(2,4)` evaluates to 32.

Tracing recursive methods requires diligence, but it can help you understand what is going on. Consider tracing a call to the recursive power method to get 2^4.

```
  assertEquals(16, rf.powRecurse(2, 4));
```

After the initial call to `powRecurse(2,4)`, `powRecurse` calls another instance of itself until the base case of `power==0` is reached. The following picture illustrates a method that calls instances of the same method. The arrows that go up indicate this. When an instance of the method can return something, it returns that value to the method that called it. The arrows that go down with the return values written to the right indicate this.



The final value of 16 is returned to the `main` method, where the arguments of 2 and 4 were passed to the first instance of `powRecurse`.

Self-Check

19-1 What is the value of `rf.powRecurse(3, 0)`?

19-2 What is the value of `rf.powRecurse(3, 1)`?

19-3 Fill in the blanks with a trace of the call `rf.powRecurse(3, 4).`

Tracing Recursive Methods

In order to fully understand how recursive tracing is done, consider a series of method calls given the following method headers and the simple `main` method:

```
// A program to call some recursive methods
public class Call2RecursiveMethods {

  public static void main(String[] args) {
    Methods m = new Methods();
    System.out.println ("Hello");
    m.methodOne(3);
    m.methodTwo(6);
  }
}

// A class to help demonstrate recursive method calls
public class Methods {
  public void methodOne(int x) {
    System.out.println("In methodOne, argument is " + x);
  }

  public void methodTwo(int z) {
    System.out.println("In methodTwo, argument is " + z);
  }
}
```

Output

```
Hello
In methodOne, argument is 3
In methodTwo, argument is 6
```

This program begins by printing out `"Hello"`. Then a method call is made to `methodOne`. Program control transfers to `methodOne`, but not before remembering where to return to. After pausing execution of `main`, it begins to execute the body of the `methodOne` method. After `methodOne` has finished, the program flow of control goes back to the last place it was and starts executing where it left off—in this case, in the `main` method, just after the call to `methodOne` and just before the call to `methodTwo`. Similarly, the computer continues executing main and again transfers control to another method: `methodTwo`. After it completes execution of `methodTwo`, the program terminates.

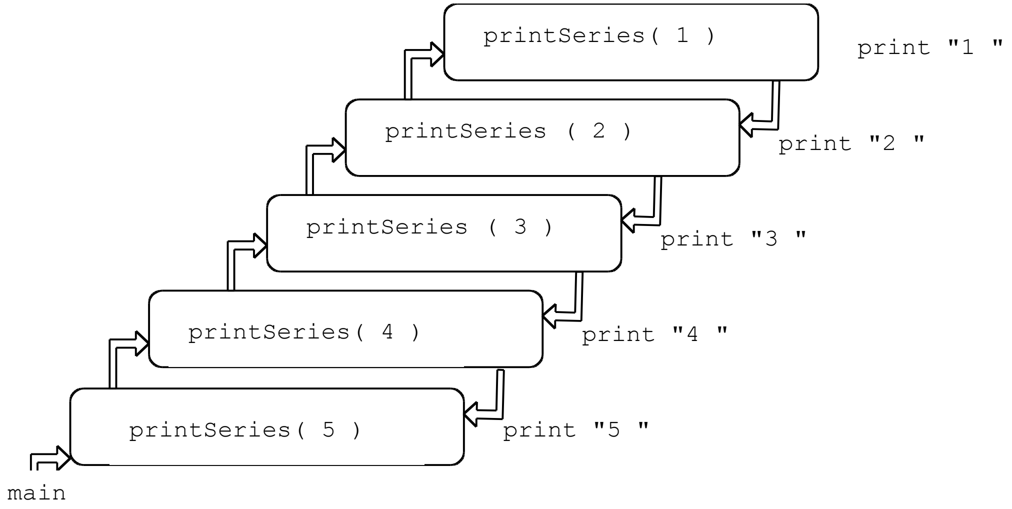
The above example shows that program control can go off to execute code in other methods and then know the place to come back to. It is relatively easy to follow control if no recursion is involved. However, it can be difficult to trace through code with recursive methods. Without recursion, you can follow the code from the beginning of the method to the end. In a recursive method, you must trace through the same method while trying to remember how many times the method was called, where to continue tracing, and the values of the local variables (such as the parameter values). Take for example the following code to print out a list of numbers from 1 to n, given n as an input parameter.

```
public void printSeries(int n) {
  if (n == 1)
    System.out.print(n + " ");
  else  {
    printSeries(n - 1);
    // after recursive call \\
    System.out.print(n + " ");
  }
}
```

A call to `printSeries(5)` generates this output:

```
1 2 3 4 5
```

Let’s examine step by step how the result is printed. Each time the method is called, it is stacked with its argument. For each recursive case, the argument is an integer one less than the previous call. This brings the method one step closer to the base case. When the base case is reached (`n==1`) the value of `n` is printed. Then the previous method finishes up by returning to the last line of code below `/* after recursive call */`.



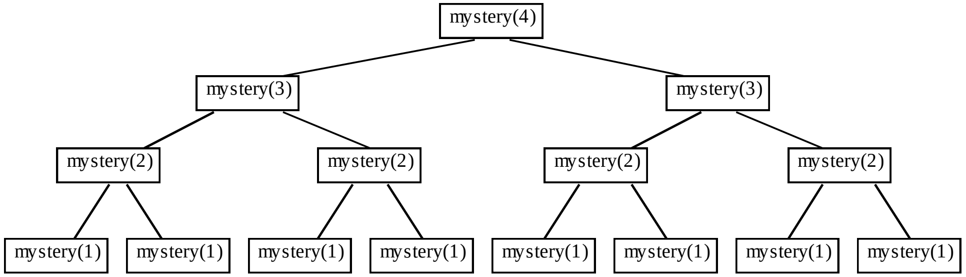
Recursive execution of printSeries(5)

Notice that when a new recursive call is made, the current invocation of the method `printSeries(5)` starts a completely new invocation of the same method `printSeries(4)`. The system pushes the new method invocation on the top of a stack. Method invocations are pushed until the method finds a base case and finishes.

Control returns back to previous invocation `printSeries(2)` by popping the stack, and the value of `n` (`2`) prints.   Now consider a method that has multiple recursive calls within the recursive case.

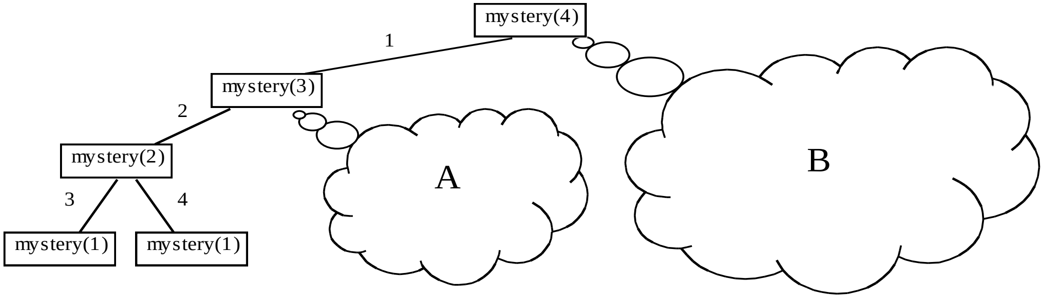
```
public void mystery(int n) {
  if (n == 1)
    System.out.print(" 1 ");
  else {
    mystery(n - 1);
    System.out.print("<" + n + ">");
    mystery(n - 1);
  }
}
```

When the base case has not yet been reached, there is a recursive call, then a print statement, and then another recursive call. With two recursive calls, it proves more insightful to approach a trace from a graphical perspective. A method call tree for `mystery(4)` looks like this.



Recursive execution of mystery(4)

As you can see, when there are multiple recursive calls in the same method, the number of calls increases exponentially — there are eight calls to `mystery(1)`. The recursion reaches the base case when at the lowest level of the structure (at the many calls to `mystery(1)`). At that point `" 1 "` is printed out and control returns to the calling method. When the recursive call returns, `"<" + n + ">"` is printed and the next recursive call is called. First consider the left side of this tree. The branches that are numbered 1, 2, 3, and 4 represent the method calls after `mystery(4)` is called. After the call #3 to `mystery`, `n` is `1` and the first output `" 1 "` occurs.



The first part of mystery

Control returns to the previous call when `n` was 2. `<2>` is printed. The output so far:

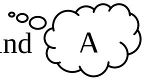
```
1 <2>
```

Then a recursive call is a made as `mystery(2-1)` and `" 1 "` is printed again. The output so far:

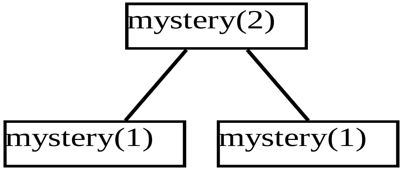
```
1 <2> 1
```

Control then returns to the first call to mystery(3) and <3> is printed. The output so far:

```
1 <2> 1 <3>
```

Then these method calls behind 

occur.



With n == 2, the base case is skipped and the recursive case is called once again. This means another call to `mystery(2-1)`, which is `" 1 "`, a printing of `<2>`, followed by another call to `mystery(2-1)`, which is yet another `" 1 "`. Add these three prints to the previous output and the output so far is:

```
1 <2> 1 <3> 1 <2> 1
```

This represents the output from `mystery(3)`. Control then returns to the original call `mystery(4)` when `<4>` is printed. Then the cloud behind B prints the same output as `mystery(3)`, the output shown immediately above. The final output is `mystery(3)`, followed by printing `n` when `n` is 4, followed by another mystery(3).

```
1 <2> 1 <3> 1 <2> 1 <4> 1 <2> 1 <3> 1 <2> 1
```

Self-Check

19-4 Describe the output that would be generated by the message `mystery(5);` .

19-5 What does `mystery2(4)` return?

```
public void mystery2(int n)  {
  if (n > 1)
    mystery2(n - 1);
  System.out.print(n + " ");
}
```

Infinite Recursion

Infinite recursion occurs when a method keeps calling other instances of the same method without making progress towards the base case or when the base case can never be met.

```
public int sum(int n) {
  if (n  ==  0)
    return 0;
  else
    return n + sum(n + 1);
}
```

In this example, no progress is made towards the base case when n is a positive integer because every time sum is called, it is called with a larger value, so the base condition will never be met.

Recursion and Method Calls

Recursion can be implemented on computer systems by allowing each method call to create a **stack frame** (also known as an activation record). This stack frame contains the information necessary for the proper execution of the many methods that are active while programs run. Stack frames contain information about the values of local variables, parameters, the return value (for non-void methods), and the return address where the program should continue executing after the method completes. This approach to handling recursive method calls applies to all methods. A recursive method does not call itself; instead, a recursive call creates an instance of a method that just happens to have the same name.

With or without recursion, there may be one too many stack frames on the stack. Each time a method is called, memory is allocated to create the stack frame. If there are many method calls, the computer may not have enough memory. Your program could throw a `StackOverflowError`. In fact you will get a `StackOverflowError` if your recursive case does not get you closer to a base case.

```
// Recursive case does not bring the problem closer to the base case.
public int pow(int base, int power) {
  if (power == 0)
    return 1;
  else
    return base * pow(base, power + 1); // <- should be power - 1
}
```

Output

```
java.lang.StackOverflowError
```

The exception name hints at the fact that the method calls are being pushed onto a stack (as stack frames). At some point, the capacity of the stack used to store stack frames was exceeded. `pow,` as written above, will never stop on its own.

Self-Check

19-6 Write the return value of each.

a. ____ `mystery6(-5)`
b. ____ `mystery6(1)`
c. ____ `mystery6(2)`
d. ____ `mystery6(3)`
e. ____ `mystery6(4)`

```
public int mystery6(int n) {
  if (n < 1)
    return 0;
  else if (n == 1)
    return 1;
  else
    return 2 * mystery6(n - 1);
}
```

19-7 Write the return value of each.

a. ____ `mystery7(14, 7)`
b. ____ `mystery7(3, 6)`
c. ____ `mystery7(4, 8)`

```
 public boolean mystery7(int a, int b) {
   if (a >= 10 || b <= 3)
     return false;
   if (a == b)
     return true;
   else
     return mystery7(a + 2, b - 2) || mystery7(a + 3, b - 4);
}
```

19-8 Given the following definition of the Fibonacci sequence, write a recursive method to compute the nth term in the sequence.

```
fibonacci(0) = 1;
fibonacci(1) = 1;
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2); when n >= 2
```

19-9 Write recursive method `howOften` as if it were n class RecursiveMethods that will compute how often a substring occurs in a string. Do not use a loop. Use recursion.   These assertions must pass:

```
@Test
public void testSequentialSearchWhenNotHere() {
  RecursiveMethods rm = new RecursiveMethods();
  assertEquals(5, rm.howOften("AAAAA", "A")) ;
  assertEquals(0, rm.howOften("AAAAA", "B")) ;
  assertEquals(2, rm.howOften("catdogcat", "cat")) ;
  assertEquals(1, rm.howOften("catdogcat", "dog")) ;
  assertEquals(2, rm.howOften("AAAAA", "AA")) ;
}
```

19.2 Palindrome

Suppose that you had a word and you wanted the computer to check whether or not it was a palindrome. A **palindrome** is a word that is the same whether read forward or backward; radar, madam, and racecar, for example. To determine if a word is a palindrome, you could put one finger under the first letter, and one finger under the last letter. If those letters match, move your fingers one letter closer to each other, and check those letters. Repeat this until two letters do not match or your fingers touch because there are no more letters to consider.

The recursive solution is similar to this. To solve the problem using a simpler version of the problem, you can check the two letters on the end. If they match, ask whether the `String` with the end letters removed is a palindrome.

The base case occurs when the method finds a `String` of length two with the same two letters. A simpler case would be a `String` with only one letter, or a `String` with no letters. Checking for a `String` with 0 or 1 letters is easier than comparing the ends of a `String` with the same two letters. When thinking about a base case, ask yourself, “Is this the simplest case? Or can I get anything simpler?” Two base cases (the number of characters is 0 or 1) can be handled like this (assume `str` is the `String` object being checked).

```
if (str.length() <= 1)
  return true;
```

Another base case is the discovery that the two end letters are different when `str` has two or more letters.

```
else if (str.charAt(0) != str.charAt(str.length() - 1))
  return false; // The end characters do not match
```

So now the method can handle the base cases with `String`s such as `""`, `"A"`, and `"no"`. The first two are palindromes; `"no"` is not.

If a `String` is two or more characters in length and the end characters match, no decision can be made other than to keep trying. The same method can now be asked to solve a simpler version of the problem. Take off the end characters and check to see if the smaller string is a palindrome. `String`’s `substring` method will take the `substring` of a `String` like `"abba"` to get `"bb"`.

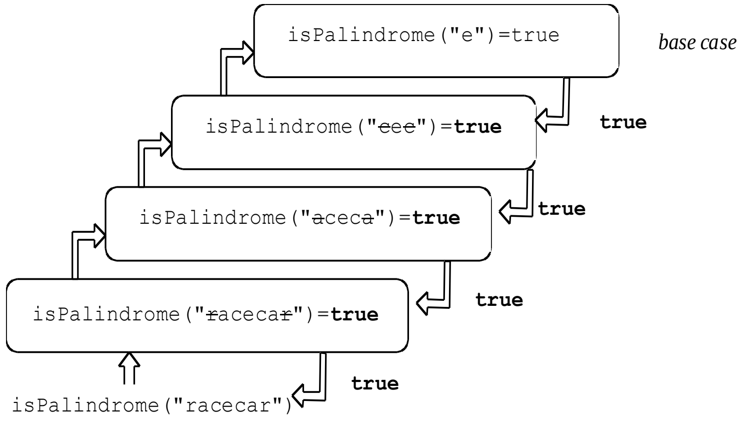
```
// This is a substring of the original string
// with both end characters removed.
return isPalindrome(str.substring(1, str.length() - 1));
```

This message will not resolve on the next call. When `str` is `"bb"`, the next call is `isPalindrome("")`, which returns `true`. It has reached a base case—length is 0. Here is a complete recursive palindrome method.

```
// Return true if str is a palindrome or false if it is not
public boolean isPalindrome(String str) {
  if (str.length() <= 1) {
    // Base case when this method knows to return true.
    return true;
  }
  else if (str.charAt(0) != str.charAt(str.length() - 1)) {
    // Base case when this method knows to return false
    // because the first and last characters do not match.
    return false;
  }
  else {
    // The first and last characters are equal so check if the shorter
    // string--a simpler version of this problem--is a palindrome.
    return isPalindrome(str.substring(1, str.length() - 1));
  }
}
```

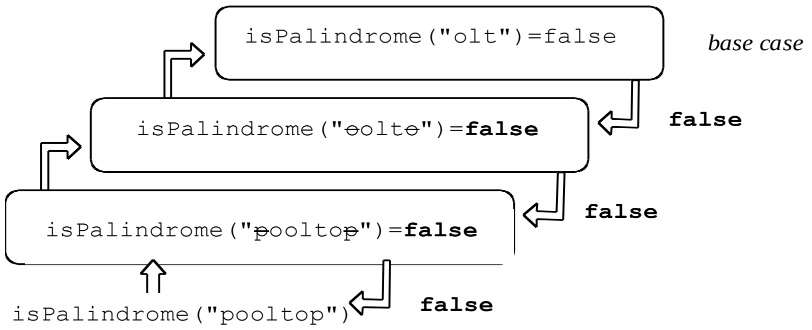
If the length of the string is greater than 1 and the end characters match, `isPalindrome` calls another instance of `isPalindrome` with smaller and smaller `String` arguments until one base case is reached. Either a `String` is found that has a length less than or equal to 1, or the characters on the ends are not the same. The following trace of `isPalindrome("racecar")` visualizes the calls that are made in this way.

`isPalindrome` Recursive Calls (true result)


isPalindrome("e")=true
base case
true
isPalindrome("~~c~~e~~c~~")=true
true
isPalindrome("~~a~~cec~~a~~")=true
true
isPalindrome("~~r~~aceca~~r~~")=true
true
isPalindrome("racecar")

Since the fourth (topmost) call to `isPalindrome` is called with the `String` `"e"`, a base case is found—a `String` with length 1. This `true` value gets returned to its caller (argument was `"e"`), which in turn returns `true` back to its caller (the argument was `"cec"`), until `true` gets passed back to the first caller of `isPalindrome`, the method call with the original argument of `"racecar"`, which returns the value `true`. Now consider tracing the recursive calls for the `String` `"pooltop"`.

`isPalindrome` Recursive Calls (false result)



isPalindrome("olt")=false
base case
isPalindrome("~~o~~olt~~o~~")=**false**
false
isPalindrome("~~p~~oolto~~p~~")=**false**
false
isPalindrome("pooltop")
false

The base case is reached when the method compares the letters at the ends—`"o"` and `"t"` do not match. That particular method call returns `false` back to its caller (whose argument was `"oolto"`), which returns `false` to its caller. The original call to `isPalindrome("pooltop"`) is replaced with `false` to the method that originally asked if `"pooltop"` was a palindrome.

Self-Check

19-10 What value is returned from `isPalindrome("yoy")?`

19-11 What value is returned from `isPalindrome("yoyo")?`

19-12 Write the return value of each method call

a. ________ huh("+abc+"));
b. ________ huh("-abc-"));
c. ________ huh("-a-b-c-"));
d. ________ huh("------abc-----"));

```
public String huh(String str) {
  if (str.charAt(0) == '-')
    return huh(str.substring(1, str.length()));
  else if (str.charAt(str.length() - 1) == '-')
    return huh(str.substring(0, str.length() - 1));
  else
    return str;
}
```

19.3 Recursion with Arrays

The *sequential search* algorithm uses an integer subscript that increases if an element is not found and the index is still in the range (meaning that there are more elements to compare). This test method demonstrates the desired behavior.

```
@Test
public void testSequentialSearchWhenHere() {
  RecursiveMethods rm = new RecursiveMethods();
  String[] array = { "Kelly", "Mike", "Jen", "Marty", "Grant" };
  int lastIndex = array.length - 1;

  assertTrue(rm.exists(array, lastIndex, "Kelly"));
  assertTrue(rm.exists(array, lastIndex, "Mike"));
```

```
    assertTrue(rm.exists(array, lastIndex, "Jen"));
    assertTrue(rm.exists(array, lastIndex, "Marty"));
    assertTrue(rm.exists(array, lastIndex, "Grant"));
  }
```

The same algorithm can be implemented in a recursive fashion. The two base cases are:

1. If the element is found, return true.
2. If the index is out of range, terminate the search by returning false.

The recursive case looks in the portion of the array that has not been searched. With sequential search, it does not matter if the array is searched from the smallest index to the largest or the largest index to the smallest. The `exists` message compares the search element with the largest valid array index. If it does not match, the next call narrows the search. This happens when the recursive call simplifies the problem by decreasing the index. If the element does not exist in the array, eventually the index goes to -1 and the method returns `false` to the preceding call, which returns `false` to the preceding call, until the original method call to `exists` returns `false` to the point of the call.

```
  // This is the only example of a parameterized method.
  // The extra <T>s allow any type of arguments.
  public <T> boolean exists(T[] array, int lastIndex, T target) {
    if (lastIndex < 0) {
      // Base case 1: Nothing left to search
      return false;
    } else if (array[lastIndex].equals(target)) { // Base case 2: Found it
      return true;
    } else { // Recursive case
      return exists(array, lastIndex - 1, target);
    }
  }
```

A test should also be made to ensure exists returns false when the target is not in the array.

```
@Test
public void testSequentialSearchWhenNotHere() {
  RecursiveMethods rm = new RecursiveMethods();
  Integer[] array = { 1, 2, 3, 4, 5 };
  int lastIndex = array.length - 1;
  assertFalse(rm.exists(array, lastIndex, -123));
  assertFalse(rm.exists(array, lastIndex, 999));
}
```

Self-Check

19-13 What would happen when `lastIndex` is not less than the array's capacity as in this assertion?

```
assertFalse(rm.exists(array, array.length + 1, "Kelly"));
```

19-14 What would happen when `lastIndex` is less than 0 as in this assertion?

```
assertFalse(rm.exists(array, -1, "Mike"));
```

19-15 Write a method `printForward` that prints all objects referenced by the array named `x` (that has `n` elements) from the first element to the last. Use recursion. Do not use any loops. Use this method heading:

```
public void printForward(Object[] array, int n)
```

19-16 Complete this `testReverse` method so a method named reverse in class RecursiveMethods will reverse the order of all elements in an array of Objects that has `n` elements. Use this heading:

```
public void printReverse(Object[] array, int leftIndex, int rightIndex)

        @Test
        public void testReverse() {
          RecursiveMethods rm = new RecursiveMethods();

            String[] array =  { "A", "B", "C" };
            rm.reverse(array, 0, 2);
            assertEquals(                        );
            assertEquals(                        );
            assertEquals(                        );
          }
```

19-17 Write the recursive method reverse is if it were in class RecursiveMethods. Use recursion. Do not use any loops.

19.4 Recursion with a Linked Structure

This section considers a problem that you have previously resolved using a loop — searching for an object reference from within a linked structure. Consider the base cases first.

The simplest base case occurs with an empty list. In the code shown below, this occurs when there are no more elements to compare. A recursive find method returns `null` to indicate that the object being searched for did not equal any in the list. The other base case is when the object is found. A recursive method then returns a reference to the element in the node.

The recursive case is also relatively simple: If there is some portion of the list to search (not yet at the end), and the element is not yet found, search the remainder of the list. This is a simpler version of the same problem – search the list that does not have the element that was just compared. In summary, there are two base cases and one recursive case that will search the list beginning at the next node in the list:

Base cases:

If there is no list to search, return `null`.
If the current node equals the object, return the reference to the data.

Recursive case:

Search the remainder of the list – from the next node to the end of the list

The code for a recursive search method is shown next as part of `class SimpleLinkedList`. Notice that there are two `findRecursively` methods — one public and one private. (Two methods of the same name are allowed in one class if the number of parameters differs.) This allows users to search without knowing the internal implementation of the class. The public method requires the object being searched for, but not the private instance variable named `front`, or any knowledge of the `Node` class. The public method calls the private method with the element to search for along with the first node to compare — `front`.

```
public class SimpleLinkedList<E> {

  private class Node  {
    private E data;
    private Node next;

    public Node(E objectReference, Node nextReference) {
      data = objectReference;
      next = nextReference;
    }
  } // end class Node

  private Node front;

  public SimpleLinkedList() {
    front = null;
  }
```

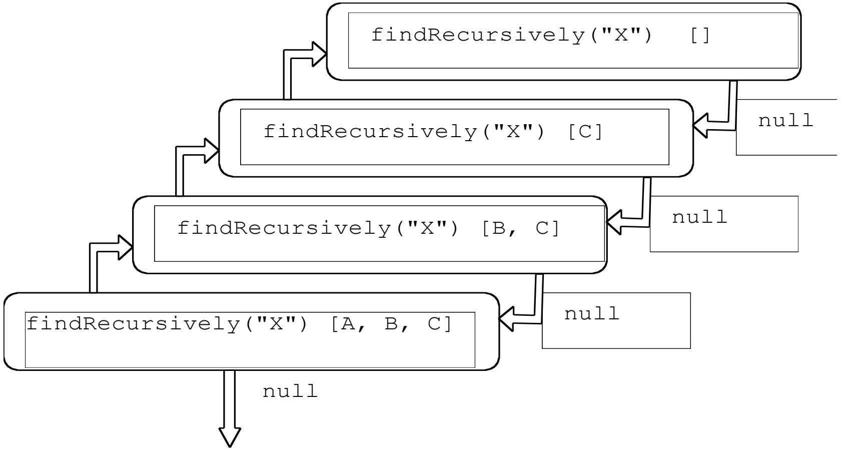
```
   public void addFirst(E element) {
     front = new Node(element, front);
  }

  // Return a reference to the element in the list that "equals" target
  // Precondition: target's type overrides "equals" to compare state
  public E findRecursively(E target) {
    // This public method hides internal implementation details
    // such as the name of the reference to the first node to compare.
    //
    // The private recursive find, with two arguments, will do the work.
    // We don't want the programmer to reference first (it's private).
    // Begin the search at the front, even if front is null.
    return findRecursively(target, front);
  }

  private E findRecursively(E target, Node currentNode) {
    if (currentNode == null) // Base case--nothing to search for
      return null;
    else if (target.equals(currentNode.data)) // Base case -- element found
      return currentNode.data;
    else
      // Must be more nodes to search, and still haven't found it;
      // try to find from the next to the last. This could return null.
      return findRecursively(target, currentNode.next);
  }
}
```

Each time the `public findRecursively` method is called with the object to find, the `private findRecursively` method is called. This private method takes two arguments: the object being searched for and `front` — the reference to the first node in the linked structure. If `front` is `null`, the private `findRecursively` method returns `null` back to the public method `findRecursively` , which in turn returns `null` back to main (where it is printed).

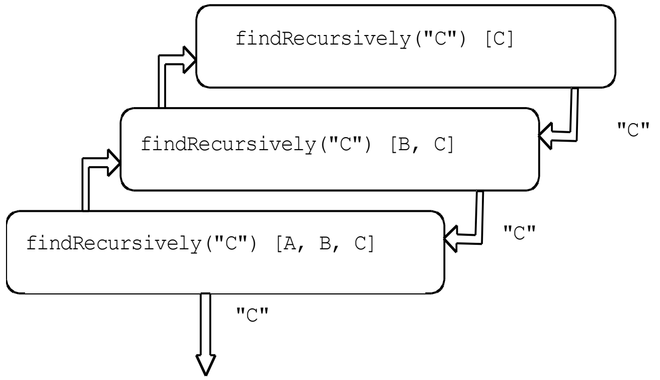
In a non-empty list, `currentNode` refers to a node containing the first element. If the first node's data does not equal the search target, a recursive call is made. The second argument would be a reference to the second node in the list. (The method still needs to pass the object being searched). This time, the problem is simpler because there is one less element in the list to search. Each recursive call to `findRecursively` has a list with one less node to consider. The code is effectively "shrinking" the search area.

The recursive method keeps making recursive calls until there is either nothing left of the list to search (return `null`), or the element being searched for is found in the smaller portion of the list. In this latter case, the method returns the reference back to the public method, which in turn returns the reference back to main (where it is printed).

At some point, the method will eventually reach a base case. There will be one method call on the stack for each method invocation. The worst case for `findRecursively` is the same for sequential search: O(n). This means that a value must be returned to the calling function, perhaps thousands of times. A trace of looking for an element that is not in the list could look like this. The portion of the list being searched is shown to the right of the call (`[]` is an empty list):



And here is a trace of a successful search for "C". If "C" were at the end of the list with size() == 975, there would have been 975 method calls on the stack.



Self-Check

19-18 Add a recursive method `toString` to the `SimpleLinkedList` class above that returns a string with the `toString` of all elements in the linked list separated by spaces. Use recursion, do not use a loop.

Answers to Self-Checks

19-1 powRecurse (3, 0) == 1

19-2 powRecurse (3, 1) == 3

19-3 filled in from top to bottom

3*(3, 0) = 1
3*(3, 1) = 3*1 = 3
3*(3, 2) = 3*3 = 9
3*(3, 3) = 3*9 = 27
3*(3, 4) = 3*27 = 81

19-4 result mystery(5)
1 <2> 1 <3> 1 <2> 1 <4> 1 <2> 1 <3> 1 <2> 1 <5> 1 <2> 1 <3> 1 <2> 1 <4> 1 <2> 1 <3> 1 <2> 1
fence post pattern - the brackets follow the numbers being recursed back into the method

19-5 mystery2(4) result: 1 2 3 4

19-6
a. __0__ `mystery6(-5)`
b. __1__ `mystery6(1)`
c. __2__ `mystery6(2)`
d. __4__ `mystery6(3)`
e. __8__ `mystery6(4)`

19-7 a. false b. false c. true

19-8
```
public int fibonacci(int n){
  if(n == 0)
    return 1;
  else if(n ==1)
    return 1;
  else if(n >= 2)
    return fibonacci(n-1) + fibonacci(n-2);
  else
   return -1;
}
```

19-9

```java
  public int howOften(String str, String sub) {
    int subsStart = str.indexOf(sub);
    if (subsStart < 0)
      return 0;
    else
      return 1 + howOften(str.substring(subsStart + sub.length()), sub);
  }
```

19-10 isPalindrome ("yoy") == true

19-11 isPalindrome("yoyo") == false

19-12 return values for huh, in order

a. +abc+
b. abc
c. a-b-c
d. abc

19-13 - if "Kelly" is not found at the first index, it will throw an arrayIndexOutOfBounds exception

19-14 - it will immediately return false without searching

19-15

```java
  public void printForward(Object[] array, int n) {
    if (n > 0) {
      printForward(array, n - 1);
      System.out.println(array[n-1].toString());
    }
  }
```

19-16

```
assertEquals("A", array[2]);
assertEquals("B", array[1]);
assertEquals("C", array[0]);
```

19-17

```
public void reverse(Object[] array, int leftIndex, int rightIndex) {
  if (leftIndex < rightIndex) {
    Object temp = array[leftIndex];
    array[leftIndex] = array[rightIndex];
    array[rightIndex] = temp;
    reverse(array, leftIndex + 1, rightIndex - 1);
  }
}
```

19-18

```
public String toString (){
  return toStringHelper(front);
}

private String toStringHelper(Node ref) {
  if(ref == null)
    return "";
  else
    return ref.data.toString() + " " + toStringHelper(ref.next);
}
```