Chapter 11
Recursion

Recursive void Methods

• A recursive method is a method that includes a call to itself
• Recursion is based on the general problem solving technique of breaking down a task into subtasks
  – In particular, recursion can be used whenever one subtask is a smaller version of the original task

Vertical Numbers

• The (static) recursive method writeVertical takes one (nonnegative) int argument, and writes that int with the digits going down the screen one per line
  – Note: Recursive methods need not be static
• This task may be broken down into the following two subtasks
  – Simple case: If n<10, then write the number n to the screen
  – Recursive Case: If n>=10, then do two subtasks:
    • Output all the digits except the last digit
    • Output the last digit

• Given the argument 1234, the output of the first subtask would be:
  1
  2
  3
• The output of the second part would be:
  4

Algorithm for Vertical Numbers

• Given parameter n:
  if (n<10)
    System.out.println(n);
  else
    { writeVertical
      // (the number n with the last digit removed)
      System.out.println(n%10); } — Note: n/10 is the number n with the last digit removed, and n%10 is the last digit of n}
A Recursive **void** Method (Part 1 of 2)

```java
public class Recursive
{
    public static void writeVertical(int arg)
    {
        System.out.print("writeVertical("
        System.out.print(arg/10);
        System.out.print(");
        System.out.print("\n");
    }
}
```

Tracing a Recursive Call

- Recursive methods are processed in the same way as any method call
  - `writeVertical(123)`
  - When this call is executed, the argument 123 is substituted for the parameter n, and the body of the method is executed
  - Since 123 is not less than 10, the else part is executed

Refresher on Function/Method Calls

- Calling functions “suspend” until the called function returns.
- How does “return” know where to go?

```java
public static void main(String[] args) {
    System.out.println("\n");
    System.out.println("\n");
}
```

Evaluating `writeVertical(123)`

- Calling a function recursively is just like calling a non-recursive function
- The current invocation of `writeVertical` “suspends” until the called invocation returns
Tracing a Recursive Call

`writeVertical(12);`
- When this call is executed, the argument 12 is substituted for the parameter `n`, and the body of the method is executed
- Since 12 is not less than 10, the `else` part is executed
- The else part begins with the method call:
  `writeVertical(n/10);`
- Substituting `n` equal to 12 produces:
  `writeVertical(12/10);`
- Which evaluates to
  `writeVertical(1);`

So this second computation of `writeVertical` is suspended, leaving two computations waiting to resume, as the computer begins to execute another recursive call.
- When this recursive call is finished, the execution of the second suspended computation will return and continue from the point above.

Execution of `writeVertical(12)`

```java
if (12 < 10) {
    System.out.println(12);
} else { // n is two or more digits long:
    System.out.println(n/10);
    writeVertical(n/10);
}
```

Execution of `writeVertical(1)`

```java
if (1 < 10) {
    System.out.println(1);
} else { // n is two or more digits long:
    System.out.println(12/10);
    writeVertical(12/10);
}
```

When the call `writeVertical(1)` ends, the suspended computation that was waiting for it to end (the one that was initiated by the call `writeVertical(12)`) resumes execution where it left off.
- It outputs the value `12%10`, which is 2
- This ends the method
- Now the first suspended computation can resume execution.
Completion of `writeVertical(12)`

```java
if (123 % 10) {
    System.out.println(123);
} else if (n is two or more digits long:)
    writeVertical(123/10);
    System.out.println(123%10);
```

**Tracing a Recursive Call**

- The first suspended method was the one that was initiated by the call `writeVertical(123)`
- It resumes execution where it left off
- It outputs the value `123%10`, which is `3`
- The execution of the original method call ends
- As a result, the digits 1, 2, and 3 have been written to the screen one per line, in that order

Completion of `writeVertical(123)`

```java
if (123 % 10) {
    System.out.println(123);
} else if (n is two or more digits long:)
    writeVertical(123/10);
    System.out.println(123%10);
```

**A Closer Look at Recursion**

- The computer keeps track of recursive calls as follows:
  - When a method is called, the computer plugs in the arguments for the parameter(s), and starts executing the code
  - If it encounters a recursive call, it temporarily stops its computation
  - When the recursive call is completed, the computer returns to finish the outer computation

**A Closer Look at Recursion**

- When the computer encounters a recursive call, it must temporarily suspend its execution of a method
  - It does this because it must know the result of the recursive call before it can proceed
  - It saves all the information it needs to continue the computation later on, when it returns from the recursive call
  - Ultimately, this entire process terminates when one of the recursive calls does not depend upon recursion to return

**General Form of a Recursive Method Definition**

- The general outline of a successful recursive method definition is as follows:
  - One or more cases that include one or more recursive calls to the method being defined
    - These recursive calls should solve "smaller" versions of the task performed by the method being defined
  - One or more cases that include no recursive calls: base cases or stopping cases
Pitfall: Infinite Recursion

- In the `writeVertical` example, the series of recursive calls eventually reached a call of the method that did not involve recursion (a stopping/base case).
- If, instead, every recursive call had produced another recursive call, then a call to that method would, in theory, run forever.
  - This is called infinite recursion.
  - In practice, such a method runs until the computer runs out of resources, and the program terminates abnormally.

An alternative version of `writeVertical`:

```java
public static void newWriteVertical(int n)
{
    newWriteVertical(n/10);
    System.out.println(n%10);
}
```

A program with this method will compile and run:

- Calling `newWriteVertical(12)` causes that execution to stop to execute the recursive call `newWriteVertical(12/10)`.
  - Which is equivalent to `newWriteVertical(1)`.
- Calling `newWriteVertical(1)` causes that execution to stop to execute the recursive call `newWriteVertical(1/10)`.
  - Which is equivalent to `newWriteVertical(0)`.

Calling `newWriteVertical(0)` causes that execution to stop to execute the recursive call `newWriteVertical(0/10)`.
- Which is equivalent to `newWriteVertical(0)`.
- ... And so on, forever!

Since the definition of `newWriteVertical` has no stopping case, the process will proceed forever (or until the computer runs out of resources).

Stacks for Recursion:

- To keep track of recursion (and other things), most computer systems use a stack.
  - A stack is a specialized kind of memory structure analogous to a stack of paper.
  - As an analogy, there is also an inexhaustible supply of extra blank sheets of paper.
  - Information is placed on the stack by writing on one of these sheets, and placing it on top of the stack (becoming the new top of the stack).
  - More information is placed on the stack by writing on another one of these sheets, placing it on top of the stack, and so on.

Stacks for Recursion:

- To get information out of the stack, the top paper can be read, but only the top paper.
  - To get more information, the top paper can be thrown away, and then the new top paper can be read, and so on.
- Since the last sheet put on the stack is the first sheet taken off the stack, a stack is called a last-in/first-out memory structure (LIFO).
Stacks for Recursion

• To keep track of recursion, whenever a method is called, a new "sheet of paper" is taken
  – The method definition is copied onto this sheet, and the arguments are plugged in for the method parameters
  – The computer starts to execute the method body
  – When it encounters a recursive call, it stops the computation in order to make the recursive call
  – It writes information about the current method on the sheet of paper, and places it on the stack

Stacks for Recursion

• A new sheet of paper is used for the recursive call
  – The computer writes a second copy of the method, plugs in the arguments, and starts to execute its body
  – When this copy gets to a recursive call, its information is saved on the stack also, and a new sheet of paper is used for the new recursive call

Stacks for Recursion

• This process continues until some recursive call to the method completes its computation without producing any more recursive calls
  • Then the computer goes to the top sheet of paper on the stack
    – This sheet contains the partially completed computation that is waiting for the recursive computation that just ended
    – Now it is possible to proceed with that suspended computation

Stacks for Recursion

• After the suspended computation ends, the computer discards its corresponding sheet of paper (the one on top)
• The suspended computation that is below it on the stack now becomes the computation on top of the stack
• This process continues until the computation on the bottom sheet is completed

Stacks for Recursion

• Depending on how many recursive calls are made, and how the method definition is written, the stack may grow and shrink in any fashion
• The stack of paper analogy has its counterpart in the computer
  – The contents of one of the sheets of paper is called a stack frame or activation record
  – The stack frames don’t actually contain a complete copy of the method definition, but reference a single copy instead

Pitfall: Stack Overflow

• There is always some limit to the size of the stack
  – If there is a long chain in which a method makes a call to itself, and that call makes another recursive call, and so forth, there will be many suspended computations placed on the stack
  – If there are too many, then the stack will attempt to grow beyond its limit, resulting in an error condition known as a stack overflow
• A common cause of stack overflow is infinite recursion
Recursion Versus Iteration

- Recursion is not absolutely necessary
  - Any task that can be done using recursion can also be done in a nonrecursive manner
  - A nonrecursive version of a method is called an iterative version
- An iteratively written method will typically use loops of some sort in place of recursion
- A recursively written method can be simpler, but will usually run slower and use more storage than an equivalent iterative version

Iterative version of writeVertical

```java
Display n as a Maxwidth Version of the Method in Display 11.2
1 public static void writeVertical(char ch, int n)
2 { int leftMaxplace = 1;
3     int leftMinplace = 1;
4     while (leftMaxplace < 10)
5     { leftMaxplace = leftMaxplace * 10;
6         if (n < leftMaxplace)
7         { ch = leftMinplace;
8             leftMinplace = leftMinplace / 10;
9             leftMinplace = leftMinplace + 1;
10             continue;
11         }
12         System.out.println(ch + leftMinplace);
13         leftMinplace = leftMaxplace;
14         n = n + leftMaxplace;
15     }
16 }
17
18 //mp101.wav is a file that has the same number
19 //of digits as n. For example, if n is 249, then
20 //mp101.wav is 1999.
21
22 for (int i = 0; i < n;
23     i++)
24     System.out.println(power(i));
25 }
26 } //mp101.wav
```

Recursive Methods that Return a Value

- Recursion is not limited to `void` methods
- A recursive method can return a value of any type
- An outline for a successful recursive method that returns a value is as follows:
  - One or more cases in which the value returned is computed in terms of calls to the same method
  - The arguments for the recursive calls should be intuitively “smaller” or “simpler”
  - One or more cases in which the value returned is computed without the use of any recursive calls (the base or stopping cases)

Another Powers Method

- The method `pow` from the Math class computes powers
  - It takes two arguments of type `double` and returns a value of type `double`
- The recursive method `power` takes two arguments of type `int` and returns a value of type `int`
  - The definition of `power` is based on the following formula:
    \[ x^n = \begin{cases} 
    1 & \text{if } n = 0 \\
    x^{n-1} \cdot x & \text{otherwise}
    \end{cases} \]

Another Powers Method

- In terms of Java, the value returned by `public int power(int x, int n)` for \( n > 0 \) should be the same as
  \[ x^{n-1} \cdot x \]
- When \( n = 0 \), then `power(x, n)` should return 1
  - This is the stopping case

The Recursive Method `power` (Part 1 of 2)

```java
public class RecursionEx02
1 { 
2 public static void main(String[] args)
3 { 
4     for (int n = 0; n < 4; n++)
5         System.out.print("" + to the power + n + \" + x + power(3, n));
6     }
7 }
8 
9 
10 }
```

© 2006 Pearson Addison-Wesley. All rights reserved
The Recursive Method \textit{power} (Part 1 of 2)

Another way to think of the recursion

- Calculating \( \text{power}(2,3) \)
  - \( \text{Power}(2,3) \) is \( \text{power}(2,2) \times 2 \)
  - \( \text{Power}(2,2) \) is \( \text{power}(2,1) \times 2 \)
  - \( \text{Power}(2,1) \) is \( \text{power}(2,0) \times 2 \)
  - \( \text{Power}(2,0) \) is 1 – The Base Case

Evaluating the Recursive Method Call \textit{power}(2,3)

Thinking Recursively

- If a problem lends itself to recursion, it is more important to think of it in recursive terms, rather than concentrating on the stack and the suspended computations

  \( \text{power}(x, n) \) returns \( \text{power}(x, n-1) \times x \)

- In the case of methods that return a value, there are three properties that must be satisfied, as follows:

  1. There is no infinite recursion
     - Every chain of recursive calls must reach a stopping case
  2. Each stopping case returns the correct value for that case
  3. For the cases that involve recursion: \textit{assuming} all recursive calls return the correct value, then the final value returned by the method is the correct value
     1. It helps to define the relationship between input and output to the function as accurately as possible
     1. These properties follow a technique also known as mathematical induction

Thinking Recursively

1. There is no infinite recursion
   - Every chain of recursive calls must reach a stopping case
2. Each stopping case returns the correct value for that case
3. For the cases that involve recursion: \textit{assuming} all recursive calls return the correct value, then the final value returned by the method is the correct value
   1. It helps to define the relationship between input and output to the function as accurately as possible
   1. These properties follow a technique also known as mathematical induction

Recursive Design Techniques

- The same rules can be applied to a recursive \texttt{void} method:
  1. There is no infinite recursion
  2. Each stopping case performs the correct action for that case
  3. For each of the cases that involve recursion: if all recursive calls perform their actions correctly, then the entire case performs correctly
     1. This is the tricky part for newbies: “how can I *assume* the recursive calls work?!!??
Binary Search

- Binary search uses a recursive method to search an array to find a specified value.
- The array must be a sorted array:
  \[ a[0] \leq a[1] \leq a[2] \leq \ldots \leq a[\text{finalIndex}] \]
- If the value is found, its index is returned.
- If the value is not found, -1 is returned.
- Note: Each execution of the recursive method reduces the search space by about a half.

Binary Search

- An algorithm to solve this task looks at the middle of the array or array segment first.
- If the value looked for is smaller than the value in the middle of the array:
  - Then the second half of the array or array segment can be ignored.
  - This strategy is then applied to the first half of the array or array segment.
- If the value looked for is at the middle of the array or array segment, then it has been found.
- If the entire array (or array segment) has been searched in this way without finding the value, then it is not in the array.

Pseudocode for Binary Search

**Algorithm to search a[first] through a[last]**

```
if (first > last) return -1;
if (key < a[mid])
    return search(a[first] through a[mid - 1], key);
else if (key > a[mid])
    return search(a[mid + 1] through a[last], key);
else return mid;
```

Recursive Method for Binary Search

```java
public class BinarySearch {
  public int search(int[] a, int first, int last, int key) {
    if (first > last) return -1;
    int mid = (first + last) / 2;
    if (key < a[mid])
      return search(a, first, mid - 1, key);
    else if (key > a[mid])
      return search(a, mid + 1, last, key);
    else return mid;
  }
  public int search(int[] a, int first, int last, int key) {
    return search(a, first, last, key);
  }
}
```

Execution of the Method `search` (Part 1 of 2)

```
<table>
<thead>
<tr>
<th>a[]</th>
<th>15</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>c[]</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>a[3]</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>a[2]</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>a[1]</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>a[4]</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>a[5]</td>
<td>83</td>
<td>83</td>
</tr>
<tr>
<td>mid = (0 + 5)/2</td>
<td>mid = (1 + 5)/2</td>
<td></td>
</tr>
<tr>
<td>first = 0</td>
<td>last = 5</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
```
Execution of the Method \texttt{search} (Part 1 of 2)

Checking the \texttt{search} Method

1. There is no infinite recursion
   - On each recursive call, the value of \texttt{first} is increased, or the value of \texttt{last} is decreased
   - If the chain of recursive calls does not end in some other way, then eventually the method will be called with \texttt{first} larger than \texttt{last}

Checking the \texttt{search} Method

2. Each stopping case performs the correct action for that case
   - If \texttt{first} > \texttt{last}, there are no array elements between \texttt{a[first]} and \texttt{a[last]}, so \texttt{key} is not in this segment of the array, and \texttt{result} is correctly set to -1
   - If \texttt{key} == \texttt{a[mid]}, \texttt{result} is correctly set to \texttt{mid}

Checking the \texttt{search} Method

3. For each of the cases that involve recursion, if all recursive calls perform their actions correctly, then the entire case performs correctly
   - If \texttt{key} < \texttt{a[mid]}, then \texttt{key} must be one of the elements \texttt{a[mid+1]} through \texttt{a[last]}, or it is not in the array
   - The method should then search only those elements, which it does
   - The recursive call is correct, therefore the entire action is correct

The method \texttt{search} passes all three tests:
Therefore, it is a good recursive method definition

Efficiency of Binary Search

- The binary search algorithm is extremely fast compared to an algorithm that tries all array elements in order
  - About half the array is eliminated from consideration right at the start
  - Then a quarter of the array, then an eighth of the array, and so forth
Efficiency of Binary Search

- Given an array with 1,000 elements, the binary search will only need to compare about 10 array elements to the key value, as compared to an average of 500 for a serial search algorithm.
- The binary search algorithm has a worst-case running time that is logarithmic: $O(\log n)$.
- A serial search algorithm is linear: $O(n)$.
- If desired, the recursive version of the method `search` can be converted to an iterative version that will run more efficiently.

Iterative Version of Binary Search

(Part 1 of 2)

```java
// Iterative Version of Binary Search

int search(int[] a, int low, int high, int key)
{
    int mid;
    while (low <= high)
    {
        mid = (low + high) / 2;
        if (key == a[mid])
            return mid;
        else if (key < a[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }
    return -1;
}
```

(Part 2 of 2)

```java
// Iterative Version of Binary Search

int search(int[] a, int low, int high, int key)
{
    int mid;
    while (low <= high)
    {
        mid = (low + high) / 2;
        if (key == a[mid])
            return mid;
        else if (key < a[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }
    return -1;
}
```