

A large, two-story, light-colored building with a red-tiled roof and a central tower, surrounded by green grass and trees under a clear blue sky.

MAHARISHI UNIVERSITY of MANAGEMENT

Engaging the Managing Intelligence of Nature

Computer Science Department

**CS390 Fundamental Programming
Practices (FPP)
Professor Paul Corazza**



© 2016 Maharishi University of Management, Fairfield, Iowa

All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi University of Management.

Lecture 4: Recursion in Java

Creation Through Self-Referral Dynamics

Wholeness of the Lesson

Computation of a function by recursion involves repeated self-calls of the function. Recursion is implicit also at the design level when a reflexive association is present. Recursion mirrors the self-referral dynamics of consciousness, the unified field, on the basis of which all creation emerges.

Outline of Topics

- Recursion Defined
- Examples: Factorial and Fibonacci
- Sometimes Iteration Is More Efficient Than Recursion
- Recursive Utility Functions
 - Reversing characters in a string
 - Finding the minimum element in an array
 - Searching for a file in a directory system

Recursion in Java

- A Java method is recursive, or exhibits recursion, if in its body it calls itself.
- In order to be a *valid recursion* (one that eventually terminates), the following criteria must be met:
 - *Base Case Exists.* The method must have a base case which returns a value without making a self-call.
 - *Self-calls Lead to Base Case.* For every input to the method, the sequence of self-calls must eventually lead to a self-call in which the base case is accessed.

Example

The factorial function on input n computes the product of the positive integers less than or equal to n .

```
int factorial(int n) {  
    //base case  
    if(n == 0 || n == 1) {  
        return 1;  
    }  
    return n * factorial(n-1);  
}
```

Demo: Observe the sequence of self-calls in executing `factorial` on input 5. See package `lesson4.fibfactorial`.

Example

The Fibonacci numbers are defined as follows:

$$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, \dots,$$
$$F_n = F_{n-1} + F_{n-2}, \dots$$

The nth Fibonacci number can be computed by:

```
int fib(int n) {  
    //base case  
    if(n == 0 || n == 1) {  
        return n;  
    }  
  
    return fib(n-1) + fib(n-2);  
}
```

Demo: Observe the sequence of self-calls – package `lesson4.fibfactorial`

Main Point

Java supports the creation of recursive methods, characterized by the fact that they call themselves in their method body. A self-calling method is a *valid* recursive function if it contains a *base case* – a branch of code that exits the method under certain conditions but does not involve a self-call – and if the sequence of self-calls, on any input to the method, always converges to the base case. Likewise, a quest for self-knowledge not based in the direct experience of the "Self" is endless (and baseless).

Design Guideline

No recursion should involve a large amount of redundant computation.

Usually, if a recursion *does* involve redundant computations, it can be rewritten as a loop or by using a more efficient recursive strategy.

Example: We have seen how there is a great deal of redundant computation in the recursive computation of Fibonacci numbers.

Example: Implementing factorial Iteratively

```
int factorial(int n) {  
    if(n == 0 || n == 1) {  
        return 1;  
    }  
  
    int result = 1;  
    for(int i = 1; i <= n; ++i ) {  
        result *= i;  
    }  
    return result;  
}
```

Exercise: Rewrite the function that computes the nth fibonacci number iteratively (using a loop instead of recursion).

Main Point

When a recursion involves many redundant computations, we try to write an iterative version of the method (using loops). Likewise, the ancient perspective on gaining knowledge is that deepest knowledge is available within self-referral dynamics of consciousness, but when this way of knowing is not available, learning through steps becomes necessary.

Highest form of gaining knowledge: *Immediate knowing*

- *Knowing through Prathibha (intuition) everything becomes known - Yoga Sutras, chap 3 (Vedic)*
- *Through use of the noetic faculty (intuition), the ground of all knowledge becomes known. - Plato, the Republic (Western)*
- *Without going outside, you may know the whole world. - Laozi, Daodejing, v. 47 (Taoist)*

不出戶知天下

bù chū hù zhī tiān xià

Intellect-based way of getting knowledge: *Go through logical steps till the goal is reached.*

Recursive Implementations of a Utility Method

- Sorting, searching, and other manipulations of characters in a string or elements in arrays or lists are often done recursively. Sometimes (but not always), an implementation of such a utility provides a public method

```
public <return-value-type> thePublicMethod(params)
```

whose signature and return type make sense to potential users, and a private recursive method

```
private <ret-value-type> privateRecurMethod(otherParams)
```

which does the real work and is designed to call itself.

We first consider simpler examples that do not require this separation

Example of a Recursive Utility: Merging Sorted Char Arrays (Strings)

Problem: Merge two strings s , t consisting of characters in the range a-z that are each in sorted order, to produce a new string whose characters are also in sorted order. Try the following recursive strategy:

1. Let chs be the 0th character of s and cht be the 0th character of t .
2. If chs comes before cht alphabetically, store chs , otherwise place cht in a buffer and remove the stored character from its original string
3. Merge the remaining strings (recursively) and insert the result in the buffer, placed after the stored character.

Thinking recursively: We are specifying a Merge procedure, yet in the third step we are calling that procedure. The way to think about it is: After you have done something with the 0th character of one of the two Strings, *assume* the recursive step works as specified.

Implementation: (see package lesson4)

```
public class MergeStrings {
    StringBuilder ret = new StringBuilder();
    public String merge(String s, String t) {
        if(s.isEmpty() && t.isEmpty()) return ret.toString();
        if(s.isEmpty()) {
            ret.append(t);
            return ret.toString();
        }
        if(t.isEmpty()) {
            ret.append(s);
            return ret.toString();
        }
        if(s.charAt(0) <= t.charAt(0)) {
            ret.append(s.charAt(0));
            return merge(s.substring(1), t);
        } else {
            ret.append(t.charAt(0));
            return merge(s, t.substring(1));
        }
    }
}

//sample usage
MergeStrings ms = new MergeStrings();
System.out.println(ms.merge("ace", "bd"));
// output:
"abcde"
```


Example of a Recursive Utility:

Reversing a String

Attempt to reverse the order of the characters in an input String by using the following strategy:

- Remove the 0th character `ch` from the input string and name the modified string `t`.
- Reverse `t` and append `ch`.

Thinking recursively. After processing the 0th character, assume the reverse method works on the remaining elements.

Implementation

```
static String reverse(String s) {  
    if(s.length() == 0) return s;  
    String first = "" + s.charAt(0);  
    return reverse(s.substring(1)) + first;  
}
```

Example of a Recursive Utility:

Finding the Minimum Value

Attempt to find the minimum (alphabetically) character in a String of characters in the range a-z, but using the following strategy:

- Remove the 0th character `ch` from the string, call the resulting string `t`.
- Find the minimum character `min` in `t`.
- If `min < ch`, return `min`; otherwise, return `ch`.

Thinking recursively. After processing the 0th character, assume the `findMin` operation works correctly on the remaining elements.

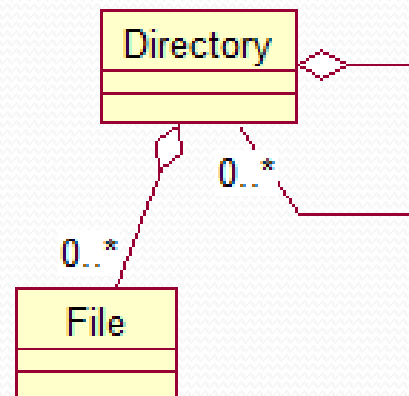
Implementation

```
public class RecursiveMin {  
    public Character rmin(String str) {  
        if(str == null || str.length() == 0) {  
            return null;  
        }  
        char ch = str.charAt(0);  
        if(str.length() == 1) return ch;  
        char c = rmin(str.substring(1));  
        return (ch < c ? ch : c);  
    }  
}
```

Object-based Recursion

- In Java, one can work with files and directories – each is represented by a particular Java class (to be discussed later). Suppose we want to write a Java method that searches for a particular file. This task will require recursion. To see what is involved, we represent the structure of a directory in the following class diagram:

Here there is a *reflexive association* from Directory to itself. This relationship at the design level suggests recursion at implementation level



Strategy

To search for a given file *file* in a given directory *dir*, the recursive strategy is:

- Get all the files and other directories that lie in the given directory *dir*
- For each of these files, compare with the given file *file* – if the same, return true
- For each directory *d* among the directories found in *dir*, recursively search for *file*
- Return false

Implementation

Rather than discuss the implementation of Directory and File in Java, we give pseudo-code to show how such a search is to be done. (See Lesson 13 for more details on File.)

```
//this is pseudo-code - not Java code
boolean searchForFile(Object file, Object startDir) {

    Object[] fileSystemObjects = startDir.getContents();

    for(Object o: fileSystemObjects) {
        //base case
        if(isFile(o) && isSameFile(o,f)) {
            return true;
        }

        if(isDirectory(o)) {
            searchForFile(file, o);
        }
    }
    //file not found in startDir
    return false;
}
```


Summary

- A Java method is *recursive*, or exhibits recursion, if in its body it calls itself.
- A recursion is *valid* if the following criteria are met:
 - The method must have a base case which returns a value without making a self-call.
 - For every input to the method, the sequence of self-calls eventually leads to a self-call in which the base case is accessed.
- Sometimes recursion leads to redundant computations, which lead to slow running times (like Fibonacci). In such cases, an implementation using iteration instead of recursion should be done.
- When recursion is used to provide utility function support, the public method signature that is exposed to the client reveals only the parameters that are relevant for the client – not the special parameters that may be needed to implement the recursion.

-

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Recursion creates from self-referral activity

1. In Java, it is possible for a method to call itself.
 2. For a self-calling method to be a legitimate recursion, it must have a base case, and whenever the method is called, the sequence of self-calls must converge to the base case.
-
3. **Transcendental Consciousness:** TC is the self-referral field of existence, at the basis of all manifest existence.
 4. **Wholeness moving within itself:** In Unity Consciousness, one sees that all activity in the universe springs from the self-referral dynamics of wholeness. The "base case" – the reference point – is always the Self, realized as Brahman.

