
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS390 Fundamental Programming Practices (FPP)
Professor Paul Corazza

Lecture 7:
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 and two examples: Factorial and Fibonacci
- Recursive Utility Functions
  - Reversing characters in a string
  - Merging sorted Strings
  - Finding the minimum element in an array

Recursion – Basic Idea

- **Recursion**: The definition of an operation in terms of itself.
  - Solving a problem using recursion depends on solving smaller occurrences of the same problem.

- A Java method is **recursive**, or exhibits recursion, if in its body it calls itself.
  - It is a powerful substitute for *iteration* (loops)
  - Particularly well-suited to solving certain types of problems (problems which can naturally be broken down into sub-problems).

Why Recursion?

- A different way of thinking of problems
- Can solve some kinds of problems better than iteration
- Leads to elegant and simple code (when used well)
- Many programming languages ("functional" languages such as Scheme, ML, and Haskell) use recursion exclusively (no loops)

First Example: factorial

- Definition of factorial:

  For any positive integer n:

  $n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$

  $0! = 1$

- Another way of defining factorial using (mathematical) recursion.

  $n! = n \times (n-1)!$

  $0! = 1.$

- In Java:

```
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 12. See package `lesson7.fibfactorial`.

Recursion in Java

- In order for a self-calling method 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.

Exercise 7.1

- You are standing in line, waiting to get into an event (like a movie or a concert). You are at the front of the line. You wonder "How many people are in line behind me?" The line is long, so there is no way for you to count the number of people in line yourself. You decide to use recursion.
- For this task, we make the following rule: Each person in line is allowed to ask the person directly behind him one question.
- What question should you, and the person behind you, and the person behind him,… ask? How can the answers lead to a solution to the problem?

The idea

- With recursion, you break up a big problem into smaller occurrences of that same problem.
  - In this problem, each person can solve a small part of the problem.
    - What is a small version of the problem that would be easy to answer?
    - What information from a neighbor might help?

Recursive Solution

The question to ask: "How many people are behind you?"

Suppose Person C is behind Person B and person B is behind Person A. If Person C answers "There are N people behind me", then Person B knows there are N+1 persons behind *him*. Then, when person A asks the question to Person B, Person B will answer "N+1".

The last person in line has no one behind him, so he will answer "0".

Recursive Solution In Code

```
public class Person {
    String name;
    Person behind;
    Person(String n) {
        name = n;
    }
    int answerToQuestion() {
        if(behind == null) return 0 ;
        else return behind.answerToQuestion() + 1;
    }
}
```

```
public class PersonLine {
    private Person[] personLine;
    PersonLine(Person[] persons) {
        personLine = persons;
    }
    public int howManyInLine() {
        return personLine[0].answerToQuestion() + 1;
    }
}
```

Another Example of Recursion

The Fibonacci numbers are defined as follows:

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

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 `lesson7.fibfactorial`

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 7.2

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

```
public class Fib {

    public int fib(int n) {
        //implement
        return 0;

    }

    public static void main(String[] args) {
        Fib f = new Fib();
        System.out.println(f.fib(10));

    }
}
```

Solution

```
public class Fib {
    int[] store;

    public int fib(int n) {
        store = new int[n+1];
        store[0] = 0;
        store[1] = 1;

        for(int i = 2; i <= n; i++) {
            store[i] = store[i-1] + store[i-2];
        }
        return store[n];
    }
}
```

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).

Outline of Topics

- Recursion Defined and two examples: Factorial and Fibonacci
- Recursive Utility Functions
  - Reversing characters in a string
  - Merging sorted Strings
  - Finding the minimum element in an array

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: 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. Assume the recursive produces the expected output, and find a way to combine the base case output with the output from the recursive step.

Implementation

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

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 containing all characters from s and t and which is 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` in a buffer, otherwise place `cht` in a buffer; 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

```
public class MergeStrings {
     StringBuilder ret = new StringBuilder();
     public String merge(String s, String t) {
       if(s == null && t == null) return null;
       if(s == null || s.isEmpty()) {
          ret.append(t);
          return ret.toString();
       }
       if(t == null || 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"));
“abcde” //output
```

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, using the following recursive strategy:

- Remove the 0th character `ch` from the string and call the resulting string `t`.
- Find (recursively) the minimum character in `t` -- call it `min` .
- 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);
  }
}
```

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.