Methods

Dr. H. B. Prajapati

Associate Professor
Department of Information Technology
Dharmsinh Desai University

4 July '20

Core Java Technology

Table of contents

Introduction to Methods

- Earlier, we have used methods such as
  - println()
  - printf()
- These methods are already available in Java classes
- We defined main() method in our own Java class.
- What is a method in programming?
  - A method is an ordered collection of statements that are grouped together to perform an operation
- Why to use method?
  - It is used for reusability of code or logic
- What about procedures and functions?
  - Procedure and function terms are used in procedural languages, e.g., in C, Pascal
  - Method term is used in object oriented languages, e.g., Java, C++

Procedure/function versus Method

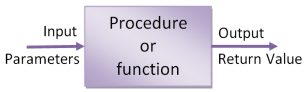


Figure : Representation of Function

- When we want to use a function, we need to pass all required data (through arguments)
- Some languages distinguish between procedure and function:
- Procedure returns nothing
- Function returns some value

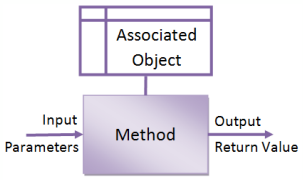


Figure : Representation of Method

- When we use a method, the method has associated object, from which it can use object data
- Method requires less parameters

Outline of Presentation

How to Create a Method?

- The following is the structure of method

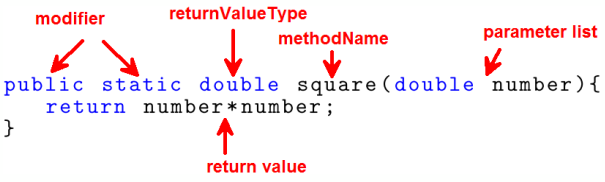
```
modifier returnValueType methodName(list of
    parameters){

}
```

- The following shows an example of how to define a method called square().

```
public static double square(double number){
    return number*number;
}
```

How to Create a Method?

- The following figure shows various parts of a Java method.



- The first line, excluding { is called method header or method specification.
- The collection of statements enclosed between { and } is called method body or method implementation.
- It is possible to have more than one statement (i.e., a collection of ordered statements) as the method body.

How to Create a Method?

- Since the square() is a method and not a function, it cannot be defined standalone. (Though in C++ it is possible, not in Java being 100% Object Oriented)
- This method must be defined in some class, as shown below:

```
class MyMath{
   public static double square(double number){
      return number*number;
   }
}
```

- Forward Note: Since this method is static, it will be called in the following way:

```
MyMath.square(4.0);
```


Method Parameters and Method Arguments

- In the method specification, a method can have a list of parameters, called formal parameters or just parameters.
  - In square(double number) method, number is called parameter.
- When we want to use a method, we pass values to the method. These values are called actual parameters or just arguments.
  - We called the method using MyMath.square(4.0); The value 4.0 is called argument.
- These parameters are optional.
- Some important points
  - We need to declare data type for each parameter.
  - If the method is declared to return some value (non void datatype), the method body must return some value using return statement.

Outline of Presentation

Calling a Method

- After a method is defined, we need to test it to check whether it works as expected?
- But, how to test it?
  - We test the method by calling it.
- How to call a method?
- If the method returns nothing, we call it in the following way:
  - methodName(arguments);
- If the method returns a value, we can call it in two ways:
  1. result = methodName(arguments);
  2. anotherMethod(methodName(arguments));
- If we are not interested in return value, we can call it in the following way:
  - methodName(arguments);

Calling a Method

- Suppose we have the following method

```
public static int min(int no1, int no2){
   ...
}
```

- We can call it in two ways:

```
result = min(10, 20);
System.out.println("min = "+min(10, 20));
```

Program: Define and Call min Method

```java
class TestMin{
   public static void main(String[] args){
      int a = 10;
      int b = 20;
      int min = min(a,b);
      System.out.println("Minimum of "+a+" and "+b+" is "+min);
   }
   public static int min(int no1, int no2){
      int min;
      if(no1 < no2)
         min = no1;
      else
         min = no2;
      return min;
   }
}
```


Program: Define and Call min Method

```
D:\programs\CJT\programs\method>javac TestMin.java

D:\programs\CJT\programs\method>java TestMin
Minimum of 10 and 20 is 10
```


Passing Parameters

- A method is very powerful.
- Once we define a method, we can call it multiple times, on different data.
- For example, a method that sorts students based on CPI can be used for sorting students of different semesters.
- For each semester, we can pass different set of students as arguments and the method will work for even new data.
- If a method takes multiple parameters, we have to pass corresponding arguments in the specified order with matching data type.

Program: Print a Line using a Character

```java
class PrintLine{
   public static void main(String[] args){
      String lc = "-";
      int w = 15;
      printLine(lc, w);
   }
   public static void printLine(String lineChar, int width){
      for(int i=0;i<width;i++){
         System.out.print(lineChar);
      }
   }
}
```


Program: Print a Line using a Character

```
D:\programs\CJT\programs\method>javac PrintLine.java

D:\programs\CJT\programs\method>java PrintLine
---------------
```

- If we change the order of arguments, as shown below

```java
   public static void main(String[] args){
      String lc = "-";
      int w = 15;
      printLine(w, lc);
   }
```

- We get the following error

```
D:\programs\CJT\programs\method>javac PrintLine.java
PrintLine.java:5: error: incompatible types: int cannot be converted to String
                printLine(w, lc);
                          ^
```


Pass by Value

- When calling a method having a parameter of primitive data type, such as int, float, etc., the value of argument is passed to the method. It is called pass by value.
- The actual passed variable (argument variable) is not affected by any change done in the called method.
- Forward Note: Object is always passed by reference, not by value (i.e., copy).

Program: Increase Weight

```
class IncrementWeight{
   public static void main(String[] args){
      int weight=40;
      System.out.println("Weight before any
         increase is "+weight);
      incrementBy(weight, 15);
      System.out.println("Weight after
         increase of 15 is "+weight);
   }
   public static void incrementBy(int weight,
      int inc){
      for(int i=0;i<inc;i++){
         weight++;
      }
   }
}
```


Program: Increase Weight

```
D:\programs\CJT\programs\method>javac IncrementWeight.java

D:\programs\CJT\programs\method>java IncrementWeight
Weight before any increase is 40
Weight after increase of 15 is 40
```


Outline of Presentation

Method Abstraction

- General meaning of abstraction is *The act of withdrawing or removing something*.
- Method abstraction allows to separate the use of method from its implementation.
- It is also referred as Information (logic) hiding.
- The main advantage is, if the implementation of the method is changed, the user of the method will not get affected.

Program: Abstracting implementation of square() method

- To understand abstraction concept, we create two Java classes:
  - Math (which provides implementation of square() method)
  - MathUser (which uses square() method of Math class)
- First, we implement square() using logic of repeated addition in the file Math.java.

```
class Math{
   public static int square(int no){
      int result=0;
      for(int i=0;i<no;i++)
         result += no;
      return result;
   }
}
```


Program: Abstracting implementation of square() method

- We use square() method in the user class MathUser.java.

```
class MathUser{
   public static void main(String[] args){
      int result = Math.square(4);
      System.out.println("Square of 4 =
         "+result);
   }
}
```

- We compile both the classes together and run MathUser class.

```
D:\programs\CJT\programs\method>javac Math*.java

D:\programs\CJT\programs\method>java MathUser
Square of 4 = 16
```

Program: Abstracting implementation of square() method

- In real world scenarios, developers provide improved version of library and still the existing applications (called users) should be able to work without recompilation of user code.
- To simulate this scenario, we provide another implementation of square() method.

```
class Math{
   public static int square(int no){
      return no*no;
   }
}
```


Program: Abstracting implementation of square() method

- Now we recompile only Math class (in real world scenario, it would be to recompile or rebuild the library).
- We run our existing/old MathUser class, see timestamps.

```
D:\programs\CJT\programs\method>javac Math.java

D:\programs\CJT\programs\method>dir Math*.class
 Volume in drive D is D-LOGICAL
 Volume Serial Number is 48FE-8416

 Directory of D:\programs\CJT\programs\method

05-07-2020  10:13 AM               240 Math.class
05-07-2020  09:55 AM               672 MathUser.class
               2 File(s)            912 bytes
               0 Dir(s)  11,630,329,856 bytes free

D:\programs\CJT\programs\method>java MathUser
Square of 4 = 16
```

- The user (MathUser) will be happy as it gets the same output even after implementation is changed.

Outline of Presentation

Method Overloading

- Earlier we created a method to find maximum of two integer numbers.
- Suppose, we want to find out minimum of two floating-point numbers also?
- We might think to have two methods for two different types of input
  - minIntegers()
  - minDoubles()
- But Java supports to use same name for more than one method, but with different number and types of parameters (not return value).
- Being able to use same name for another method is called method overloading.
- When a method is called, the Java compiler can determine which method to invoke based on data type of the arguments.

Program: Overloading min method(), Slide - I

```
class OverloadedMinMethods{
   public static void main(String[] args){
      System.out.println("Minimum of two
         integers, 10 and 50 is "+min(10,50));
      System.out.println("Minimum of two
         floating points, 10.0 and 50.0 is
         "+min(10.0,50.0));
   }
   public static int min(int no1, int no2){
      int min;
      if(no1 < no2)
         min = no1;
      else
         min = no2;
      return min;
   }
   public static double min(double no1, double
      no2){
```


Program: Overloading min method(), Slide - II

```
      double min;
      if(no1 < no2)
         min = no1;
      else
         min = no2;
      return min;
   }
}
```

Program: Overloading min method()

```
D:\programs\CJT\programs\method>javac OverloadedMinMethods.java

D:\programs\CJT\programs\method>java OverloadedMinMethods
Minimum of two integers, 10 and 50 is 10
Minimum of two floating points, 10.0 and 50.0 is 10.0
```


Program: Overloading min method(), Slide - I

- Suppose for the same, two method calls, we define overloaded min method with float and double data type.
- Which method will get called for int data type and which method will get called for floating-point (i.e., double) data type.
- We print version of method to know which method gets called.

```
class OverloadedMinMethods{
   public static void main(String[] args){
      System.out.println("Minimum of two
         integers, 10 and 50 is "+min(10,50));
      System.out.println("Minimum of two
         floating points, 10.0 and 50.0 is
         "+min(10.0,50.0));
   }
   public static float min(float no1, float
      no2){
```


Program: Overloading min method(), Slide - II

```
      System.out.println("Float");
      float min;
      if(no1 < no2)
         min = no1;
      else  min = no2;
      return min;
   }
   public static double min(double no1, double
      no2){
      System.out.println("Double");
      double min;
      if(no1 < no2)
         min = no1;
      else
         min = no2;
      return min;
   }
}
```


Program: Overloading min method()

```
D:\programs\CJT\programs\method>javac OverloadedMinMethods.java

D:\programs\CJT\programs\method>java OverloadedMinMethods
Float
Minimum of two integers, 10 and 50 is 10.0
Double
Minimum of two floating points, 10.0 and 50.0 is 10.0
```


Overloading Rules

- Two methods can be overloaded if either the number of parameters are different or the data type of parameters are different.
- If modifiers or return values are different, they do not help.
- Example: the following two methods have same data type of parameters. Though one is static and another is non-static, they can not be overloaded.

```
public static double min(double no1, double
   no2){
   ...
}

public double min(double no1, double no2){
   ...
}
```

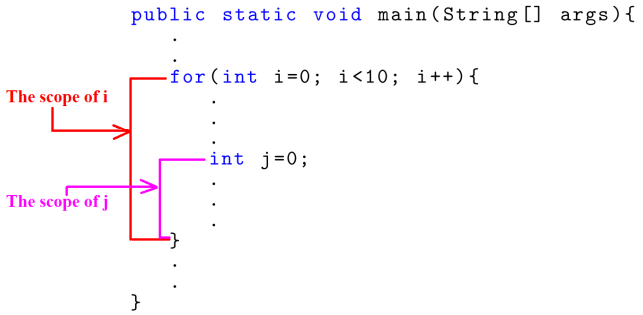

Outline of Presentation

Scope of Variables

- The scope of a variable is the portion of the program where the variable can be referenced.
- A variable defined inside a method is called a local variable.
- The scope of a local variable starts from its declaration and ends at the end of that block.
- A local variable must be declared and assigned a value before it is used.
- A parameter is actually a local variable and its scope is that entire method.

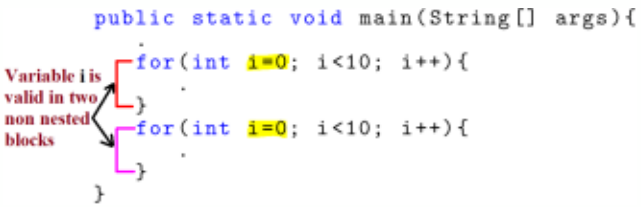
Examples of Scope of Variables

- A variable declared (e.g., i in the following example) in the header of for loop has its scope in the entire loop.
- A variable declared inside the loop body (e.g., j) has its scope from its declaration to the end of the block.



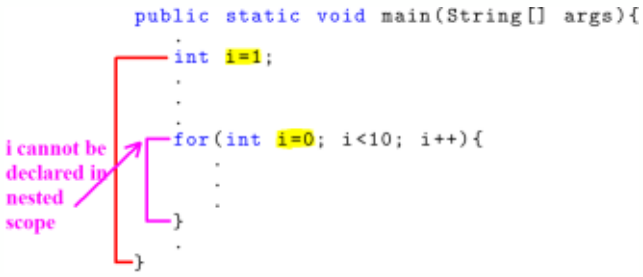
Examples of Scope of Variables

- We can declare a local variable with the same name in different blocks in a method.
- For example, i in the following example:



Examples of Scope of Variables

- We cannot declare a variable twice in the same block or in nested blocks.
- For example, we cannot declare i in the for loop, as i is already declared in the outer scope.



Outline of Presentation

What is Debugging?

- Debugging is finding errors (bugs) in a program and correcting them.
- Earlier,we learned that the errors can be categorized into two:
  1. Compilation Errors (are reported by compiler)
  2. Runtime Errors (occurs while program is running)
- Runtime errors can result in incorrect output or cause a program to terminate abnormally.

How to do Debugging?

- There are three approaches:
  - Hand trace the program (read the program and catch errors)
  - Insert print statements (might work for small programs)
  - Use Debugger utility (Effective approach)
- Java IDEs (popular: Netbeans, Eclipse, InteliJ) are equipped with debugger, which we can use in GUI mode.
- JDK installation provides command based debugger utility called jdb.

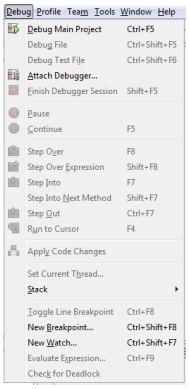


Figure : Debug Menu in Netbeans IDE

Features of Debugger

Features of debuggers differ from system to system; however most support the following common features:

- Executing a single statement at a time (single stepping)
- Tracing into or stepping over a method
- Setting breakpoints
- Displaying variables (Watch)
- Using call stacks (trace all method calls)
- Modifying variables

Outline of Presentation

What is Recursion?

- We learned how to call another method. For example, main() method called square() method.
- But, can a method call itself?
- Recursion is the process of a method calling itself, either directly or indirectly.
- Recursion allows to solve certain problems very easily, which are otherwise difficult to solve.Example: Fibonacci series. Suppose find 10th Fibonacci number.
- What is Fibonacci Series
  - Fibonacci series was introduced by Leonardo Fibonacci to model growth of rabbit population
  - The series starts with two 1s in succession
  - Each subsequent number is the sum of previous two numbers in the series
  - (1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...)

Modeling Solution using the Method Defined recursively

- We consider the example of Fibonacci Series: (1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...)
- We can see that $fib(1) = 1$ and $fib(2) = 1$
- $fib(3) = 1 + 1 = fib(1) + fib(2)$ same way $fib(4) = fib(2) + fib(3)$ and so on
- Thus, the formula of $fib()$ in terms of n becomes: $fib(n) = fib(n-2) + fib(n-1)$
- Thus the series can be defined recursively as
  1. $fib(1) = 1$;
  2. $fib(2) = 1$;
  3. $fib(n) = fib(n-2) + fib(n-1); n > 2$

Defining Java Method Recursively

- Two cases ($n = 1$ and $n = 2$) are not recursively defined.
- If we call the method for these two cases, the method returns the result immediately.
- These two cases are called base case or terminating or stopping condition.
- If we call the method with $n > 2$, the method divides the problem into two sub-problems of the same nature.
- For these two sub-problems, we can call the same method with different arguments, which is called a recursive call.

Recursive fib() Method

- Thus, the fib() can be defined recursively as follows:

```
int fib(int n){
      if(n==1 || n==2)
         return 1;
      else
         return fib(n-1) + fib(n-2);
}
```


Program: Find Fibonacci Number

```
class Fibonacci{
   public static void main(String[] args){
      System.out.println("10th Fibonacci
         number is "+fib(10));
   }
   public static int fib(int n){
      if(n==1 || n==2)
         return 1;
      else
         return fib(n-2) + fib(n-1);
   }
}
```


Program: Find Fibonacci Number

```
D:\programs\CJT\programs\method>javac Fibonacci.java

D:\programs\CJT\programs\method>java Fibonacci
10th Fibonacci number is 55
```


Finding nth Factorial

- Factorial of a number is defined as
  - $0! = 1$
  - $1! = 1 \text{ X } 1 = 1$
  - $2! = 2 \text{ X } 1 \text{ X } 1 = 2$
  - $3! = 3 \text{ X } 2 \text{ X } 1 \text{ X } 1 = 6$
  - $4! = 4 \text{ X } 3 \text{ X } 2 \text{ X } 1 \text{ X } 1 = 24$
- Factorial of any number is the number multiplied by the factorial of its previous number, i.e., $n! = n * factorial(n-1)$
- Base case or terminating condition is $0! = 1$
- Thus, the factorial can be defined recursively as
  1. $factorial(0) = 1;$
  2. $factorial(n) = n * factorial(n-1); n > 0$

Program: Find nth Factorial

```
class Factorial{
   public static void main(String[] args){
      System.out.printf("0! =
         %5d\n",factorial(0));
      System.out.printf("1! =
         %5d\n",factorial(1));
      System.out.printf("2! =
         %5d\n",factorial(2));
      System.out.printf("3! =
         %5d\n",factorial(3));
      System.out.printf("4! =
         %5d\n",factorial(4));
   }
   public static int factorial(int n){
      if(n==0) return 1;
      else     return n*factorial(n-1);
   }
}
```


Program: Find nth Factorial

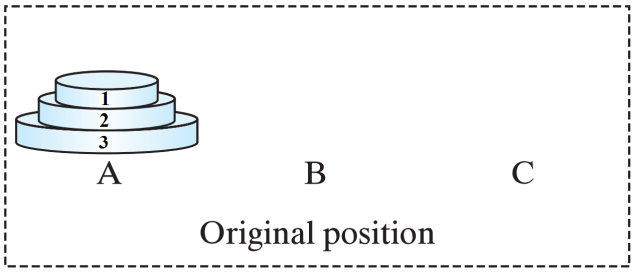
```
D:\programs\CJT\programs\method>javac Factorial.java

D:\programs\CJT\programs\method>java Factorial
0! =     1
1! =     1
2! =     2
3! =     6
4! =    24
```

Tower of Hanoi Problem

- The Towers of Hanoi problem is a classic problem that can be solved easily using recursion but is difficult to solve otherwise.
- The problem is about moving a specified number of disks of distinct sizes from one tower to another while respecting the following rules:
  - There are n disks labeled 1, 2, 3, ...n and there are three towers labeled A, B, and C.
  - No disk can be placed on top of a smaller disk at any time
  - All the disks are initially placed on tower A
  - Only one disk and only from top can be moved at a time
- The objective is to move all the disks from tower A to B with the help of tower C.

Solution of Tower of Hanoi, Slide - I

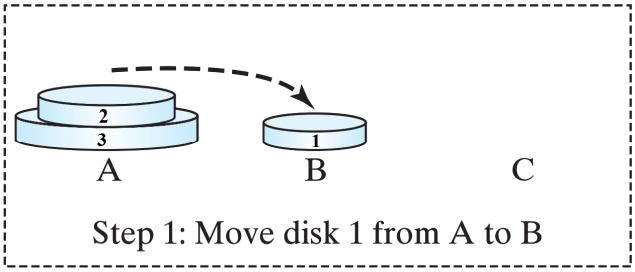
Steps of solution for n = 3



Adapted from An Introduction to Java Programming, Y. Daniel Liang, Eigth Edition, Prentice Hall

Solution of Tower of Hanoi, Slide - II

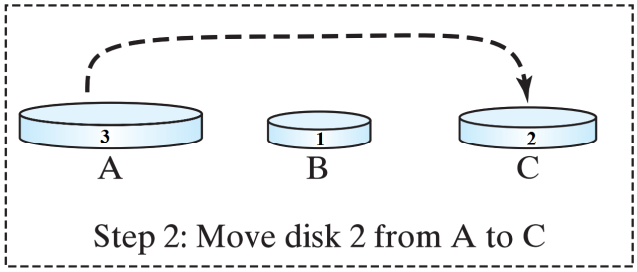
Steps of solution for n = 3



Adapted from An Introduction to Java Programming, Y. Daniel Liang, Eigth Edition, Prentice Hall

Solution of Tower of Hanoi, Slide - III

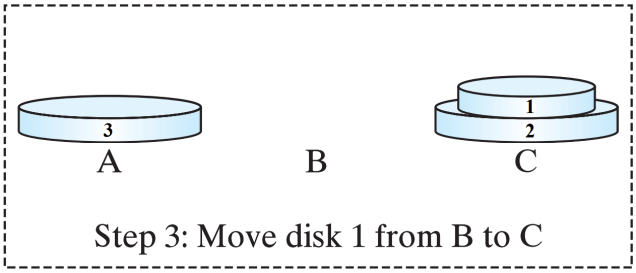
Steps of solution for n = 3



Adapted from An Introduction to Java Programming, Y. Daniel Liang, Eigth Edition, Prentice Hall

Solution of Tower of Hanoi, Slide - IV

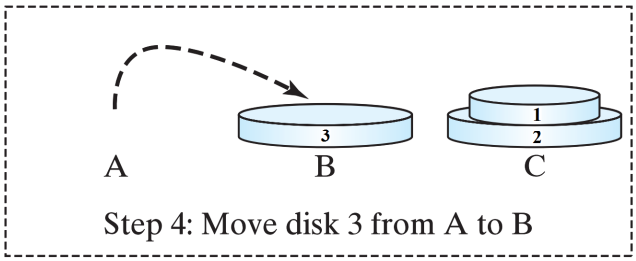
Steps of solution for n = 3



Adapted from An Introduction to Java Programming, Y. Daniel Liang, Eigth Edition, Prentice Hall

Solution of Tower of Hanoi, Slide - V

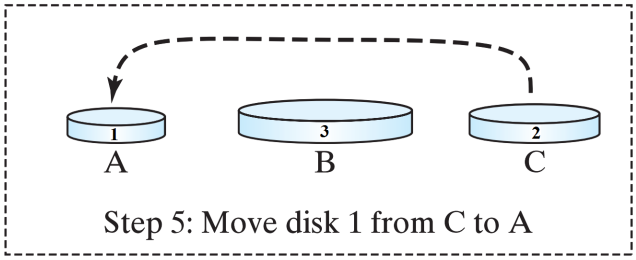
Steps of solution for n = 3



Adapted from An Introduction to Java Programming, Y. Daniel Liang, Eigth Edition, Prentice Hall

Solution of Tower of Hanoi, Slide - VI

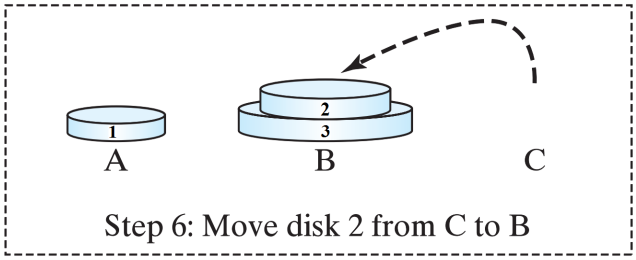
Steps of solution for n = 3



Adapted from An Introduction to Java Programming, Y. Daniel Liang, Eigth Edition, Prentice Hall

Solution of Tower of Hanoi, Slide - VII

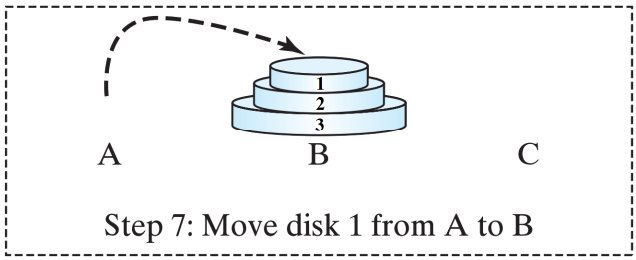
Steps of solution for n = 3



Adapted from An Introduction to Java Programming, Y. Daniel Liang, Eigth Edition, Prentice Hall

Solution of Tower of Hanoi, Slide - VIII

Steps of solution for n = 3

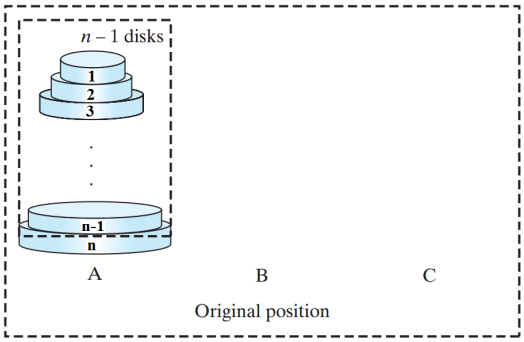


Adapted from An Introduction to Java Programming, Y. Daniel Liang, Eigth Edition, Prentice Hall

Modeling Solution using Recursive Definition, Slide - I

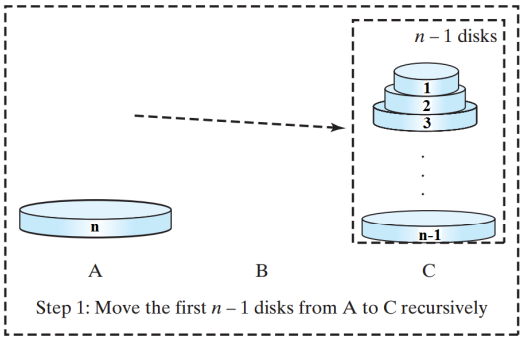
- For three disks, i.e., n = 3, we can find solution manually.
- However, for even n = 4, the problem becomes complex.
- The solution is inherently recursive in nature.
- The base case is n=1. If n == 1, we can simply move disk from A to B.
- When $n > 1$, we can split the original problem into three subproblems and solve these subproblems recursively.
  1. Move the first n-1 disks from A to C with the help of tower B.
  2. Move disk n from A to B
  3. Move the n-1 disks from C to B with the help of tower A.

Modeling Solution using Recursive Definition, Slide - II



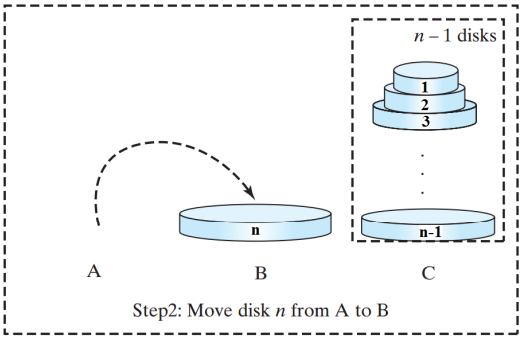
Adapted from An Introduction to Java Programming, Y. Daniel Liang, Eigth Edition, Prentice Hall

Modeling Solution using Recursive Definition, Slide - III



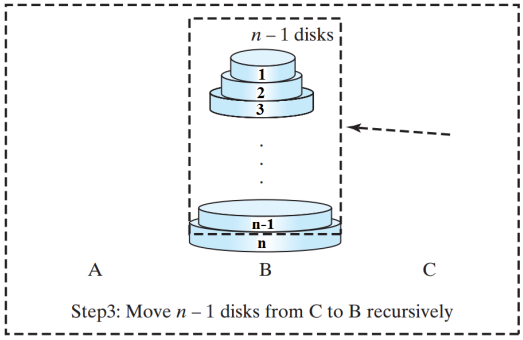
Adapted from An Introduction to Java Programming, Y. Daniel Liang, Eigth Edition, Prentice Hall

Modeling Solution using Recursive Definition, Slide - IV



Adapted from An Introduction to Java Programming, Y. Daniel Liang, Eigth Edition, Prentice Hall

Modeling Solution using Recursive Definition, Slide - V



Adapted from An Introduction to Java Programming, Y. Daniel Liang, Eigth Edition, Prentice Hall

Recursive Definition of Tower of Hanoi Solution

- We want to define a method that moves n disks from fromTower to the toTower with the help of auxTower.

```
void moveDisks(int n, char fromTower, char
    toTower, char auxTower);
```


Recursive Definition of Tower of Hanoi Solution

- The method is defied recursively as:

```
void moveDisks(int n, char fromTower,
    char toTower, char auxTower){
    if(n==1)
        System.out.println("Move disk "+n+" from
            "+fromTower+
            " to "+toTower);
    else{
        moveDisks(n-1,fromTower, auxTower,
            toTower);
        System.out.println("Move disk "+n+" from
            "+fromTower+
            " to "+toTower);
        moveDisks(n-1,auxTower, toTower,
            fromTower);
    }
}
```


Program: Tower of Hanoi, Slide - I

```
import java.util.*;
class TowerOfHanoi{
    public static void main(String[] args){
        Scanner input=new Scanner(System.in);
        System.out.print("Enter number of disks
            to move = ");
        int diskCount = input.nextInt();
        System.out.println("The moves are ");
        moveDisks(diskCount,'A','B','C');
    }
    public static void moveDisks(int n, char
        fromTower,
        char toTower, char auxTower){
        if(n==1)
            System.out.println("Move disk "+n+"
                from "+fromTower+
                " to "+toTower);
        else{
```


Program: Tower of Hanoi, Slide - II

```
            moveDisks(n-1,fromTower, auxTower,
                toTower);
            System.out.println("Move disk "+n+"
                from "+fromTower+
                " to "+toTower);
            moveDisks(n-1,auxTower, toTower,
                fromTower);
        }

    }
}
```

Program: Tower of Hanoi, Slide - III

```
D:\programs\CJT\programs\method>javac TowerOfHanoi.java

D:\programs\CJT\programs\method>java TowerOfHanoi
Enter number of disks to move = 3
The moves are
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
Move disk 1 from C to A
Move disk 2 from C to B
Move disk 1 from A to B
```


Recursion versus Iteration

Iteration

1. It is explicit repetition controlled by loop construct.
2. Terminated using loop variable.
3. Loop body is executed repeatedly.
4. Loop has no additional overhead
5. Generally, loop does not require separate variables for each iteration.

Recursion

1. It is implicit repetition without loop control.
2. Terminated by base or terminating condition.
3. Method is called recursively.
4. Recursion has overhead of memory and time.
5. Each time the recursive method is called, memory space is required for all local variables and parameters.

Recursion versus Iteration

Iteration

1. Any problem that can be solved recursively can be solved iteratively.
2. Iterative solution should be used if it is obvious and the nature of the problem is iterative.

Recursion

1. Recursive solution comes with many negative aspects–needs too much space and too much time for method calls and returns.
2. Recursive solution should be used over iterative, if the nature of the problem is recursive or writing an iterative solution is very complicated.

Summary of key terms

- Procedure, function, method
- Creating a method, method parameters, method arguments
- Calling/using a method
- Passing parameters, pass by value
- Method abstraction, multiple implementations
- Method overloading, overloading rules
- Scope of variables
- Debugging, debugger
- Recursion, Fibonacci, Factorial, Tower of Hanoi, Recursion versus Iteration

References

- An Introduction to Java Programming, Y. Daniel Liang, PHI
- An Introduction to Java Programming, Y. Daniel Liang, Eigth Edition, Prentice Hall