Recursion

Definitions

- Recursion
  - Process of solving a problem by reducing it to smaller versions of itself
- Recursive algorithm
  - Algorithm that finds the solution to a given problem by reducing the problem to smaller versions of itself
  - Has one or more base cases
  - Implemented using recursive methods

Definitions

- Recursive method
  - Method that calls itself
- Base case
  - Case in recursive definition in which the solution is obtained directly
  - Stops the recursion

Definitions - Example

0! = 1 (By Definition!)
n! = n x (n – 1) ! If n > 0
3! = 3 x 2!
2! = 2 x 1!
1! = 1 x 0!

0! = 1 (Base Case!)

1! = 1 x 0! = 1 x 1 = 1
2! = 2 x 1! = 2 x 1 = 2
3! = 3 x 2! = 3 x 2 = 6

Definitions

- General solution
  - Breaks problem into smaller versions of itself
- General case
  - Case in recursive definition in which a smaller version of itself is called
  - Must eventually be reduced to a base case

Definitions

- Directly recursive: a method that calls itself
- Indirectly recursive: a method that calls another method and eventually results in the original method call. Method A calls method B, which in turn calls method A.

Definitions

- Infinite recursion
  - Recursive calls are continuously made until memory has been exhausted
  - Caused by either omitting base case or writing recursion step that does not converge on base case
  - This error is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

7

Tracing a Recursive Method

- Recursive method
  - Logically, you can think of a recursive method having unlimited copies of itself
  - Every recursive call has its own
    - Code
    - Set of parameters
    - Set of local variables

8

Tracing a Recursive Method

- After completing a recursive call
  - Control goes back to the calling environment
  - Recursive call must execute completely before control goes back to previous call
  - Execution in previous call begins from point immediately following recursive call

9

Designing Recursive Methods

- Identify general case(s)
- Identify base case(s)
- Provide direct solution to each base case
- Provide solutions to general cases in terms of smaller versions of general cases

10

Example Using Recursion: Factorials

- Factorial of n, or n! is the product
- n · (n – 1) · (n – 2) · … · 1
- With 1! equal to 1 and 0! Defined to be 1.
- Can be solved recursively or iteratively (nonrecursively)
- Recursive solution uses following relationship:
- n! = n · (n – 1)!

11

Using Iteration to find the factorial of a number

```
import java.util.Scanner;

public class Factorial
{

  public static void main(String[] args)
  {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter the number whose factorial is to be found: ");
    int n = scanner.nextInt();
    int result = factorial(n);
    System.out.println("The factorial of " + n + " is " + result);
  }

  public static int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
      result = result * i;
    }
    return result;
  }
}
```

12

```java
// Fig. 15.10: FactorialCalculator.java
// Iterative factorial method.

public class FactorialCalculator
{
   // recursive declaration of method factorial
   public long factorial( long number )
   {
      long result = 1;

      // iterative declaration of method factorial
      for ( long i = number; i >= 1; i-- )
         result *= i;

      return result;
   } // end method factorial

   // output factorials for values 0-10
   public void displayFactorials()
   {
      // calculate the factorials of 0 through 10
      for ( int counter = 0; counter <= 10; counter++ )
         System.out.printf( "%d! = %d\n", counter, factorial( counter ) );
   } // end method displayFactorials
} // end class FactorialCalculator
```

Iterative solution uses counter-controlled repetition

Outline

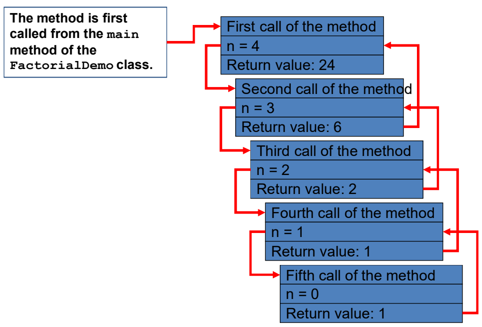
- FactorialTest.java

```java
// Fig. 15.11: FactorialTest.java
// Testing the iterative factorial method.

public class FactorialTest
{
   // calculate factorials of 0-10
   public static void main( String args[] )
   {
      FactorialCalculator factorialCalculator = new FactorialCalculator();
      factorialCalculator.displayFactorials();
   } // end main
} // end class FactorialTest
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Solving Problems With Recursion



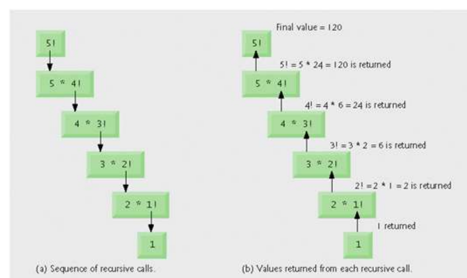


Fig. 15.2 | Recursive evaluation of 5!.


```java
// Fig. 15.3: FactorialCalculator.java
// Recursive factorial method.

public class FactorialCalculator
{
   // recursive method factorial
   public long factorial( long number )
   {
      if ( number <= 1 ) // test for base case
         return 1; // base cases: 0! = 1 and 1! = 1
      else // recursion step
         return number * factorial( number - 1 );
   } // end method factorial

   // output factorials for values 0-10
   public void displayFactorials()
   {
      // calculate the factorials of 0 through 10
      for ( int counter = 0; counter <= 10; counter++ )
         System.out.printf( "%d! = %d\n", counter, factorial( counter ) );
   } // end method displayFactorials
} // end class FactorialCalculator
```

Base case returns 1

Recursion step breaks problem into two parts: one the method knows how to do, one the method does not

Portion method knows how to do

Recursive call: Portion method does not know how to do; smaller version of original problem

Original call to recursive method


```java
// Fig. 15.4: FactorialTest.java
// Testing the recursive factorial method.

public class FactorialTest
{
   // calculate factorials of 0-10
   public static void main( String args[] )
   {
      FactorialCalculator factorialCalculator = new FactorialCalculator();
      factorialCalculator.displayFactorials();
   } // end main
} // end class FactorialTest
```

Calculate and display factorials

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Example Using Recursion: Fibonacci Series

- Fibonacci series begins with 0 and 1 and has property that each subsequent Fibonacci number is the sum of previous two Fibonacci numbers.
- Fibonacci numbers or Fibonacci series or Fibonacci sequence are the numbers in the following integer sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144...
- The first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent number is the sum of the previous two.
- Fibonacci series defined recursively as:
  - fibonacci(0) = 0
  - fibonacci(1) = 1
  - fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)
- Recursive solution for calculating Fibonacci values results in explosion of recursive method calls

19

```
public class MyFibonacci {

  public static void main(String a[])
 {

     int febCount = 15;
     int[] feb = new int[febCount];
     feb[0] = 0;
     feb[1] = 1;

     for(int i=2; i < febCount; i++)
     {
        feb[i] = feb[i-1] + feb[i-2];
     }

     for(int i=0; i< febCount; i++){
          System.out.print(feb[i] + " ");
     }
  }
```

20

21

```
// Fig. 15.5: FibonacciCalculator.java
// Recursive fibonacci method.

public class FibonacciCalculator
{
   // recursive declaration of method fibonacci
   public long fibonacci( long number )
   {
      if ( ( number == 0 ) || ( number == 1 ) ) // base cases
         return number;
      else // recursion step
         return fibonacci( number - 1 ) + fibonacci( number - 2 );
   } // end method fibonacci

   public void displayFibonacci()
   {
      for ( int counter = 0; counter <= 10; counter++ )
         System.out.printf( "Fibonacci of %d is: %d\n", counter,
            fibonacci( counter ) );
   } // end method displayFibonacci
} // end class FibonacciCalculator
```

Two base cases

Two recursive calls

Original call to recursive method

22

```
// Fig. 15.6: FibonacciTest.java
// Testing the recursive fibonacci method.

public class FibonacciTest
{
   public static void main( String args[] )
   {
      FibonacciCalculator fibonacciCalculator = new FibonacciCalculator();
      fibonacciCalculator.displayFibonacci();
   } // end main
} // end class FibonacciTest
```

```
Fibonacci of 0 is: 0
Fibonacci of 1 is: 1
Fibonacci of 2 is: 1
Fibonacci of 3 is: 2
Fibonacci of 4 is: 3
Fibonacci of 5 is: 5
Fibonacci of 6 is: 8
Fibonacci of 7 is: 13
Fibonacci of 8 is: 21
Fibonacci of 9 is: 34
Fibonacci of 10 is: 55
```

Calculate and display Fibonacci values

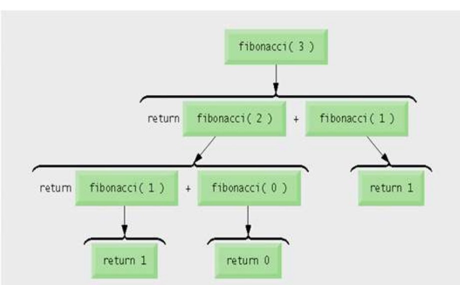


Fig. 15.7 | Set of recursive calls for `fibonacci( 3 )`.

23

Recursion vs. Iteration

- Any problem that can be solved recursively can be solved iteratively
- Both iteration and recursion use a control statement
  - Iteration uses a repetition statement
  - Recursion uses a selection statement
- Iteration and recursion both involve a termination test
  - Iteration terminates when the loop-continuation condition fails
  - Recursion terminates when a base case is reached
- Recursion can be expensive in terms of processor time and memory space, but usually provides a more intuitive solution

24

Recursion vs Iteration

- Any problem that can be solved recursively can also be solved iteratively (non-recursively).
- A recursive approach is normally preferred over an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug.
- A recursive approach can often be implemented with fewer lines of code.
- Another reason to choose a recursive approach is that an iterative one might not be apparent.

25

pow solution

```
// Returns base ^ exponent.
// Precondition: exponent >= 0
public static int pow(int base, int exponent) {
    if (exponent == 0) {
        // base case; any number to 0th power is
  1
        return 1;
    } else {
        // recursive case:  x^y = x * x^(y-1)
        return base * pow(base, exponent - 1);
    }
}
```

26

pow solution 2

```
// Returns base ^ exponent.
// Precondition: exponent >= 0
public static int pow(int base, int exponent) {
    if (exponent == 0) {
        // base case; any number to 0th power is
  1
        return 1;
    } else if (exponent % 2 == 0) {
        // recursive case 1:  x^y = (x^2)^(y/2)
        return pow(base * base, exponent / 2);
    } else {
        // recursive case 2:  x^y = x * x^(y-1)
        return base * pow(base, exponent - 1);
    }
}
```

27