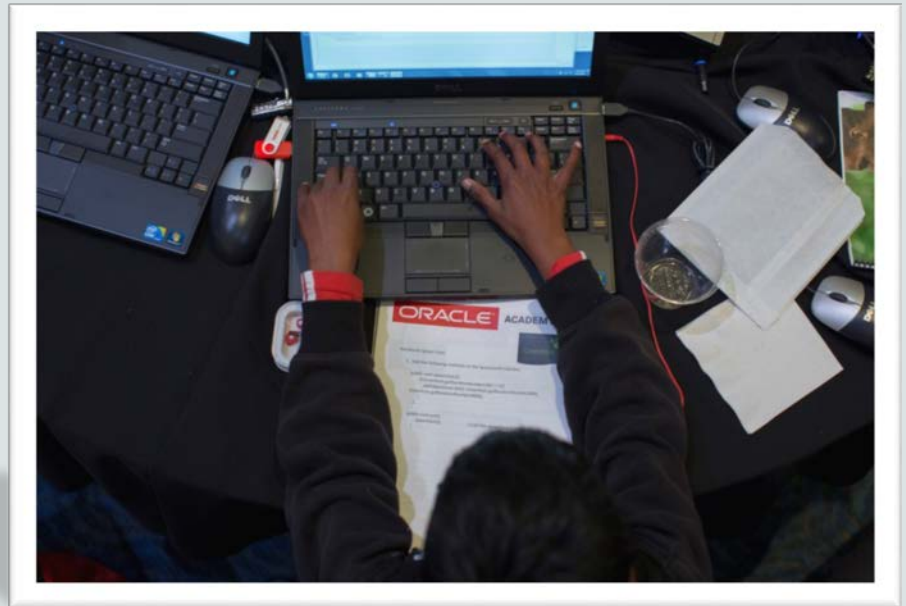




Java Programming

7-3

Recursion



Objectives

This lesson covers the following objectives:

- Create linear recursive methods
- Create non-linear recursive methods
- Compare the pros and cons of recursion

Understanding Recursion

- A recursive program is one that calls itself one or more times with each execution until it satisfies the base case arguments, then discontinues calling itself.
- It contains:
 - A base case: A segment of code that tells the program when to stop calling itself and return a value or void.
 - A recursive case: A call out to another copy of itself.
 - A pattern of convergence: The process of working backward through a problem's data set.

Understanding Recursion

The types of recursion are:

- Linear: Only one call to itself in the recursive case.
 - Useful and frequently used.
- Non-Linear: Two or more calls to itself in the recursive case.
 - Used less frequently because of its impact on system, specifically JVM memory.





Understanding Recursion

- The base case means the program does not need to recursively call itself any more. It returns the default value (or does nothing) at the bottom most activity.
- The recursive case occurs when the program cannot resolve the problem without a recursive call to itself.
- Convergence means that you recursively call the method until you reach the base case, and then you return values or go back up the chain to the original program unit.

Recursion Process

- Recursion looks backward through a chain of events, while traditional loops look forward through events.
- Recursion works backward through convergence on a base case, where the base case occurs when you are back to the beginning.

Forward Thinking:

Process of adding a number to itself.

Given: $t_1 = 5$

Then: $t_{(n+1)} = t_n + 5$

Sequence: 5, 10, 15, ...



Backward Thinking:

Process of subtracting a number from itself.

Given: $t_1 = 5$

Then: $t_n = t_{(n-1)} + 5$

Sequence: ... 15, 10, 5

Forward and Recursive Sequences

Forward Thinking (Loop)

```
public static double forward(double d) {  
    // Declare local variables.  
    double n = 5;  
    double r = 0;  
    // Add n to r, d times.  
    for (double i = 0; i < d; i++) {  
        r += n;  
    }  
    return r;  
}
```




Forward Sequence Explained

- The variable "r" in the forward thinking program is the result variable.
- It is defined initially and incremented with each iteration through the loop.
- This type of variable is essential when we navigate forward without recursion.

Forward and Recursive Sequences

Backward Thinking (Recursion)

```
public static double backward(double d) {  
    // Declare local variable.  
    double n = 5;  
    if (d <= 1) //base case  
        return n;  
    else  
        return backward(d - 1) + n;  
}
```



Recursive Sequence Explained

- There is no variable "r" in the recursive example because it is not required.
- The return result from each recursive method call passes back the solution, the version of the method that called it increments and returns the result to the prior level.
- This continues until you reach the top most method call.
- In a sense, the return value of the method call in conjunction with the recursive calling statement manages "r" (the result) value without explicit declaration or management.

Tracing Through Linear Recursion

- Trace through this recursive example using the backwards thinking process.
- Call `recur(4)` and keep track of `n` with each new call.

```
public static int recur(int n){  
    //base case  
    if (n <= 1) {  
        return 1;  
    }  
    else {  
        //recursive case  
        return 3 * recur(n - 1);  
    }  
}
```

Tracing Through Linear Recursion

- Initially, $n=4$. Since $n>1$, there is another call made to recur.
- This means the return value of $\text{recur}(4)$ will be equivalent to $3 * \text{recur}(3)$. When we follow through $\text{recur}(3)$, we see the return value will be equal to $3 * \text{recur}(2)$.
- This pattern is continued until the base case is reached.

```
public static int recur(int n){  
    //base case  
    if (n <= 1) {  
        return 1;  
    }  
    else {  
        //recursive case  
        return 3 * recur(n - 1);  
    }  
}
```

Keeping Track of Trace

- Keeping track of the trace through in a table can aid in retrieving the final result of the original call, recur(4).
- The table shows step by step what is being called and what the return value of each recursive call is.
- Recursion requires backwards thinking.
- Once a value given by the base case is reached, fill in the values for the previous calls.

n	Call	Results
4	recur(4)	3*recur(3)
3	recur(3)	3*recur(2)
2	recur(2)	3*recur(1)
1	recur(1)	1

Replace Calls with Actual Values

- The returned value for `recur(1)` is 1.
- With this information, replace the call of `recur(1)` with 1, which gives a value of 3 for a call of `recur(2)`.
- Replace the call of `recur(2)` with its actual value.
- Continue this pattern to find the final value of what is returned when `recur(4)` is called, which is 27.

n	Call	Results	Actual Value
4	<code>recur(4)</code>	$3 * \text{recur}(3)$	$3 * (3 * (3 * 1)) = 27$
3	<code>recur(3)</code>	$3 * \text{recur}(2)$	$3 * (3 * 1) = 9$
2	<code>recur(2)</code>	$3 * \text{recur}(1)$	$3 * 1 = 3$
1	<code>recur(1)</code>	1	1

Linear Recursion Factorial Problem

The following factorial problem:

- Calls one copy of itself.
- Calls itself until the base case.
- Returns values from the lowest recursive call to original call.

```
public static double factorial(double d) {  
  
    // Sort elements by title case.  
    if (d <= 1) {  
        return 1;  
    }  
    else {  
        return d * factorial(d - 1);  
    }  
}
```


Demonstration of Linear Recursion Factorial Problem

- Assuming the subscript (zero-based numbering) is the number of calls to the recursive function, where zero is the first call:
 - Calls to the method: $d_0 = 5$, $d_1 = 4$, $d_2 = 3$, $d_3 = 2$, $d_4 = 1$
- With an initial value of 5 the given result will be
 - It returns: $5 * (4 * (3 * (2 * (1))))$ or 120

The Fibonacci Sequence

- The Fibonacci Sequence is a series of numbers where the next number is found by adding up the two numbers before it: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- The 2 is found by adding the two numbers before it (1+1)
- Similarly, the 3 is found by adding the two numbers before it (1+2) and so on!
- The next number in the sequence above is $34+55 = 89$

Non-linear Recursion Fibonacci Problem

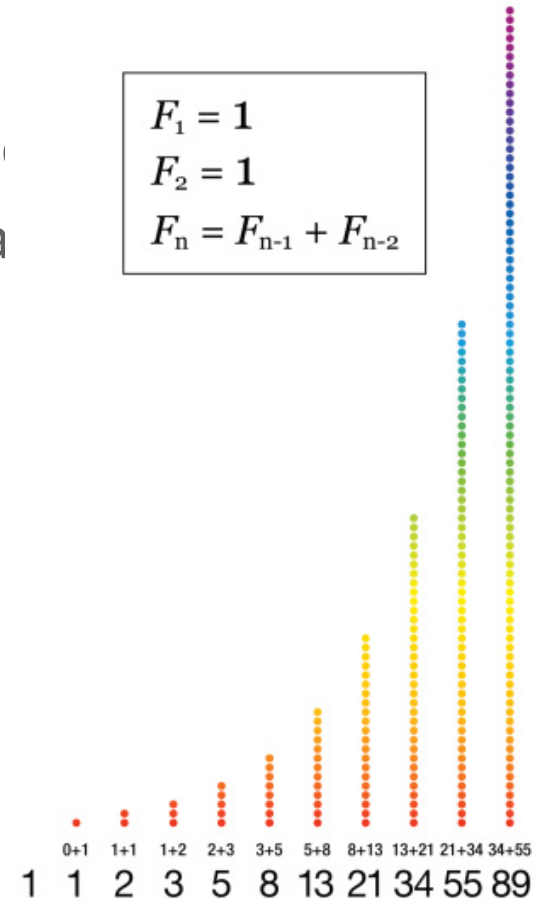
- The following Fibonacci problem:
 - Calls two or more copies of itself.
 - Calls first copy until the base case, then second.
 - Returns values from the lowest recursive call and adds them together for each series of calls.

$$F_1 = 1$$

$$F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

```
public static double fibonacci(double d) {  
    // Sort elements by title case.  
    if (d < 2) {  
        return d;  
    }  
    else {  
        return fibonacci(d - 1) + fibonacci(d - 2);  
    }  
}
```





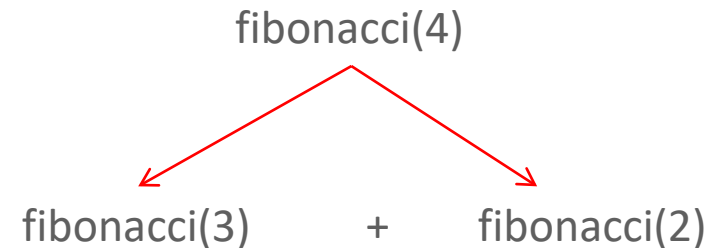
Tracing Through Non-Linear Recursion

- Tracing through a non-linear recursive method is slightly more complex than linear recursive methods.
- Although the concept of backward thinking is the same, it may be difficult to use a chart for keeping track of your calls.
- Using a tree diagram is more practical for tracing this type of recursion.

Trace a Call to fibonacci(4)

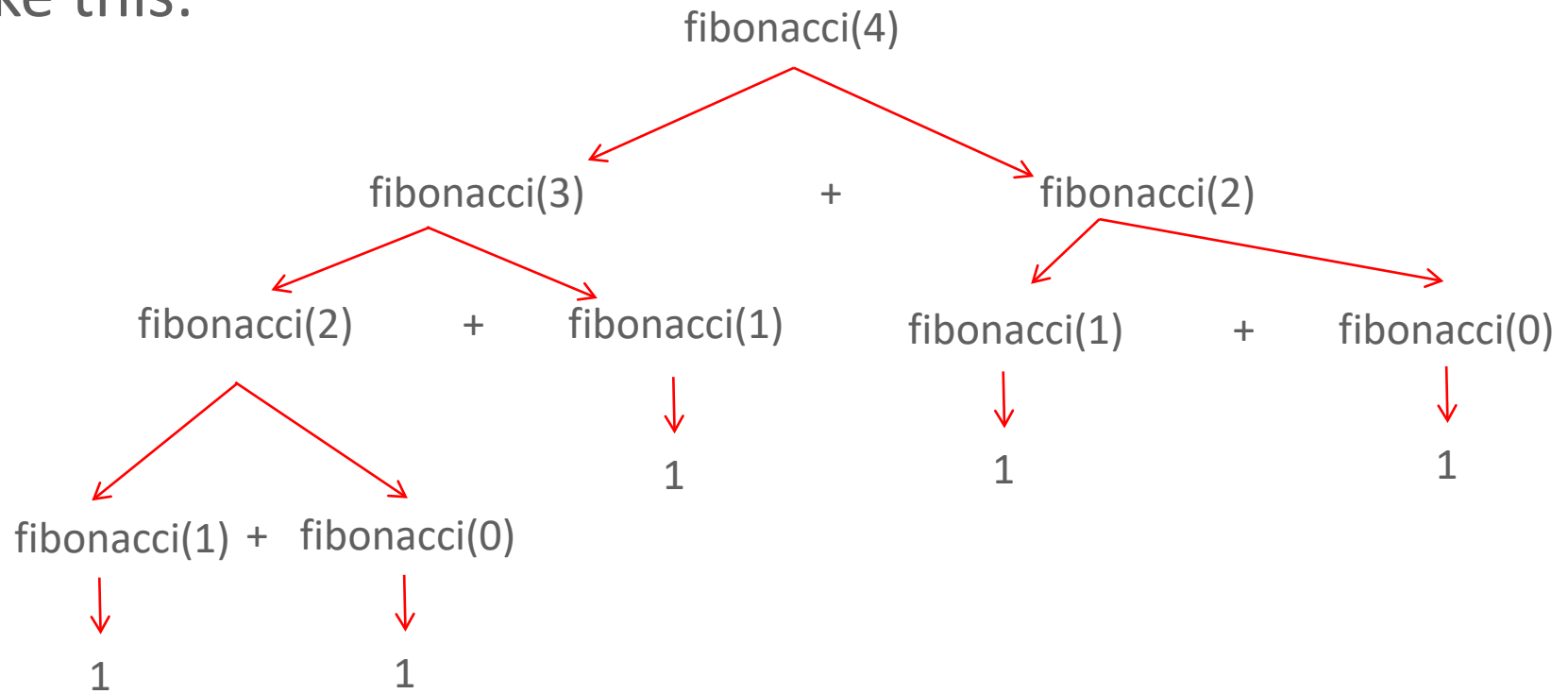
- Trace a call to fibonacci(4) to find out what the returned value is.
- When you make a call where $d = 4$, you get this result: $\text{fibonacci}(4) = \text{fibonacci}(3) + \text{fibonacci}(2)$.

```
public static double fibonacci(double d) {  
    // Sort elements by title case.  
    if (d < 2) {  
        return d;  
    }  
    else {  
        return fibonacci(d - 1) + fibonacci(d - 2);  
    }  
}
```



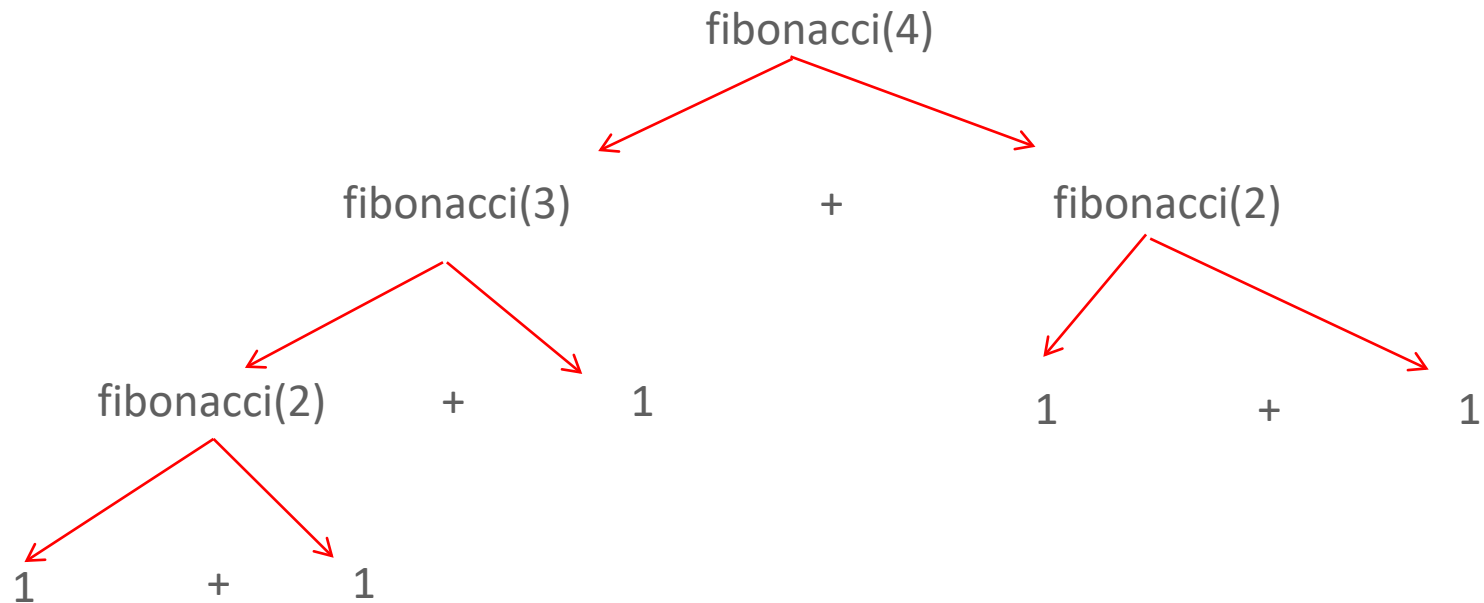
Continue to Trace Through to Base Case

Continuing the trace through until you reach the base case for all branches, which will give you a tree that looks like this:



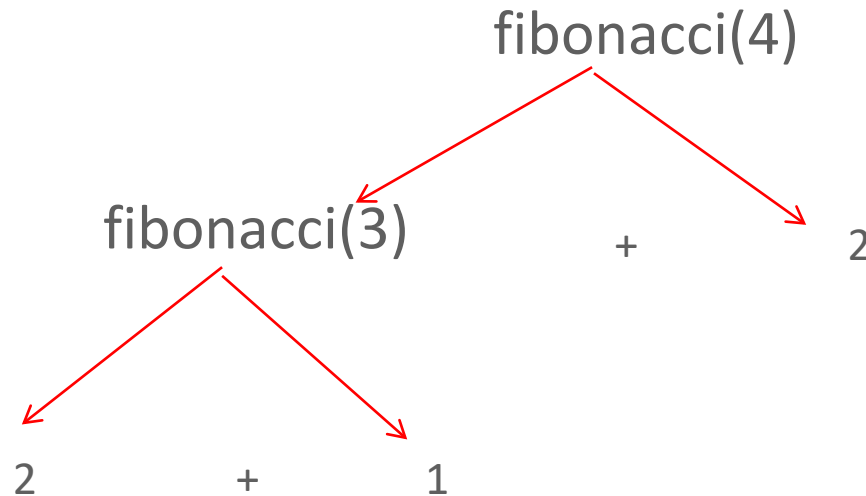
Continue to Trace Through to Base Case

The new tree looks like this:



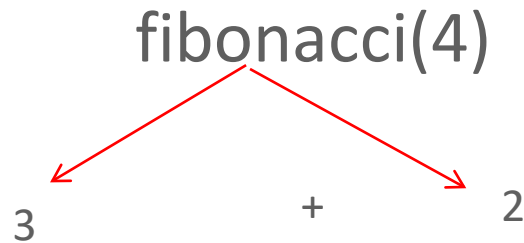
Continue to Trace Through to Base Case

Since $\text{fibonacci}(2) = 1 + 1 = 2$, change it in the tree:



Fibonacci Problem Conclusion

- Next, replace fibonacci(3) with it's value ($2+1 = 3$).



- Finally, replace fibonacci(4) with its return value ($3+2=5$).
- The conclusion is that $\text{fibonacci}(4) = 5$.

Recursion Pros and Cons

- Pros:
 - Once understood, often more intuitive.
 - Simpler, more elegant code.
- Cons:
 - Uses more function calls.
 - Can cause performance issues.
 - Uses more memory.

Recursion Pros and Cons

- When programming with either trees or sorted lists then recursion is generally the best method of interacting with these data structures.
- The UNIX operating system uses recursion when executing directory structure commands.
- Even if you do not use recursion in your programs, you still need to understand it since other programmers' code may use recursion.



Terminology

Key terms used in this lesson included:

- Base case
- Class method
- Class variable
- Convergence
- Inner class
- Linear recursion
- Nested class

Terminology

Key terms used in this lesson included:

- Non-linear recursion
- Recursion
- Recursive case

Summary

In this lesson, you should have learned how to:

- Create linear recursive methods
- Create non-linear recursive methods
- Compare the pros and cons of recursion

