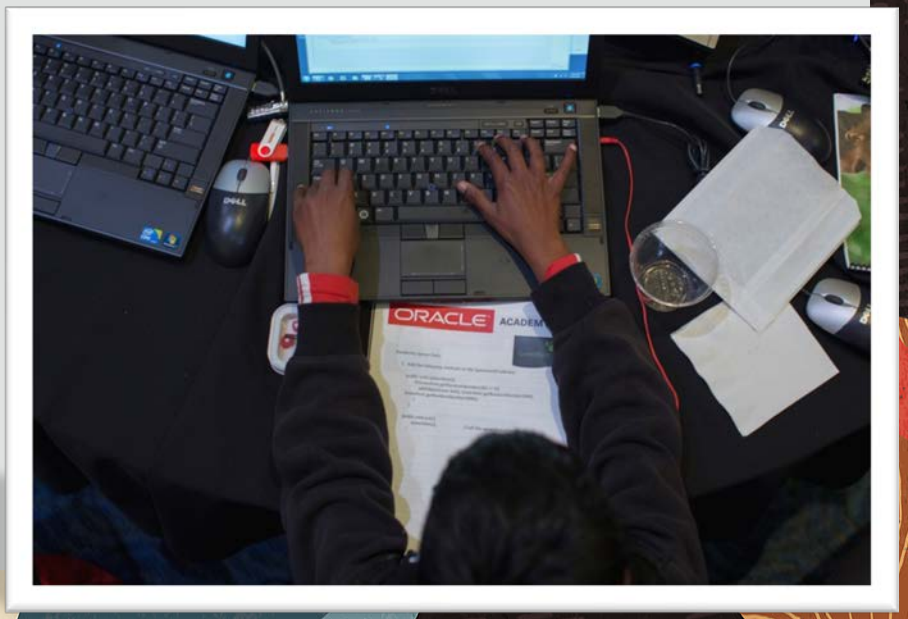

ORACLE
Academy

Oracle Academy
Java for AP Computer Science A

10-5

Recursion



ORACLE
Academy

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

Visual examples of recursive code will really help your understanding of recursion. The watch and step over/into tools in Eclipse help to get a better understanding of recursion

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 the system, specifically JVM memory

Recursion although powerful has to be used very carefully as it has serious memory implications if too many recursive calls are made.

Understanding Recursion

- The base case means the method 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 method cannot resolve the problem without a recursive call to itself
  - Convergence means that you recursively call the method until you reach the base case, then you return values or go up the chain to the original program unit

Once the base case has been reached and the problem has been solved, the program then goes back through the memory stack and executes the stored commands or unexecuted code.

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.

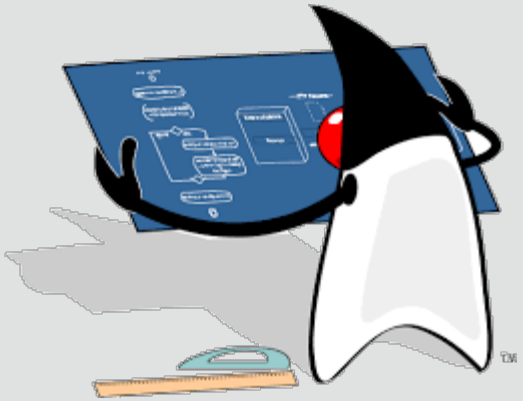
Given: $t_1 = 5$
Then: $t_{(n+1)} = t_n + 5$
Sequence: 5, 10, 15, ...

Backward Thinking:
Process of subtracting a number from itself.

Given: $t_1 = 5$
Then: $t_n = t_{(n-1)} + 5$
Sequence: … 15, 10, 5

Understanding Recursion

- If the base case is poorly chosen or the algorithm does not converge on the base case the Java process running on the computer will quickly run out of memory (stack overflow)



Forward and Recursive Sequences

- The forward thinking (loop) method is the more traditional method of solving problems
- Creating a method that does all of the processing and then returns the value back to the calling method
- This method is called once and completes all of the processing within that one method call

```
public static double forward(double limit) {
   //Declare local variables.
   double num1 = 5, result = 0;

   //Add n to r, d times.
   for (double i = 0; i < limit; i++) {
      result += num1;
   }//endfor
   return result;
}//end method forward
```

Forward Sequence Explained

- This explains the code on the previous slide
- The variable “result" 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

To get a full understanding add the code to a program and execute it within Eclipse using the debugger tools (watch, step etc..) to get a better understanding of how the execution is carried out.

Forward and Recursive Sequences

- Backward Thinking (Recursion)
- The recursive method shown here demonstrates how recursion works with the backward method continually calling itself until the base case has been reached

```
public static double backward(double limit) {
   // Declare local variable.
   double num = 5;
   if (limit <= 1)//base case
      return num;
   else
      return backward(limit - 1) + num;
   //endif
}//end method backward
```

Recursive Sequence Explained

- There is no variable “result" 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 “result" (the result) value without explicit declaration or management

Linear Recursion Example

1. Create the following project (Do not run it)

```
public class RecursionExample {

   public static void main(String[] args) {
      recurseMethod(4);
   }//end method main

   static void recurseMethod(int num){
      if(num == 0)
         return;
      else{
         System.out.println("hello " + num);
         recurseMethod(--num);
         System.out.println(""+num);
         return;
      }//endif
   }//end method recurseMethod
}//end class RecursionExample
```

Linear Recursion Example

2. Trace through this example using the backwards thinking process on paper, what will be displayed?

```
static void recurseMethod(int num){
   if(num == 0)
      return;
   else{
      System.out.println("hello " + num);
      recurseMethod(--num);
      System.out.println(""+ num);
      return;
   }//endif
}//end method recurseMethod
```

To get a good understanding of this process it can be useful to write down on paper what you think the values will be at each stage of the process before executing the code. Then check if you were correct.

Linear Recursion Example

3. Run the program
4. Compare the output to the trace through that you created in step 2
5. Does it match what you thought was going to happen?

Output:

```
hello 4
hello 3
hello 2
hello 1
0
1
2
3
```

Tracing Through Linear Recursion

- Initially, num=4
- The base case is num = = 0
- Since num is not equal to 0, there is another call made to recurseMethod
- This means the return value of recurseMethod (4) will be recurseMethod(3). Following this through leads to recurseMethod(2), recurseMethod(1) and recurseMethod(0)
- This pattern is continued until the base case is reached (num==0)
- The remainder of the execution is now carried out

Linear Recursion Factorial Problem

6. Add the following factorial problem to the RecursionExample code:
   - Calls one copy of itself
   - Calls itself until the base case
   - Returns values from the lowest recursive call to original call

```
public static void main(String[] args) {
   recurseMethod(4);
   factorial(5);
}//end method main

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

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: d0 = 5, d1 = 4, d2 = 3, d3 = 2, d4 = 1
- With an initial value of 5 the given result will be
  - It returns: 5 * (4 * (3 * (2 * (1)))) or 120

Remember the program stores the values of each recursive call to the memory stack until the base case is reached when they are retrieved and executed.

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

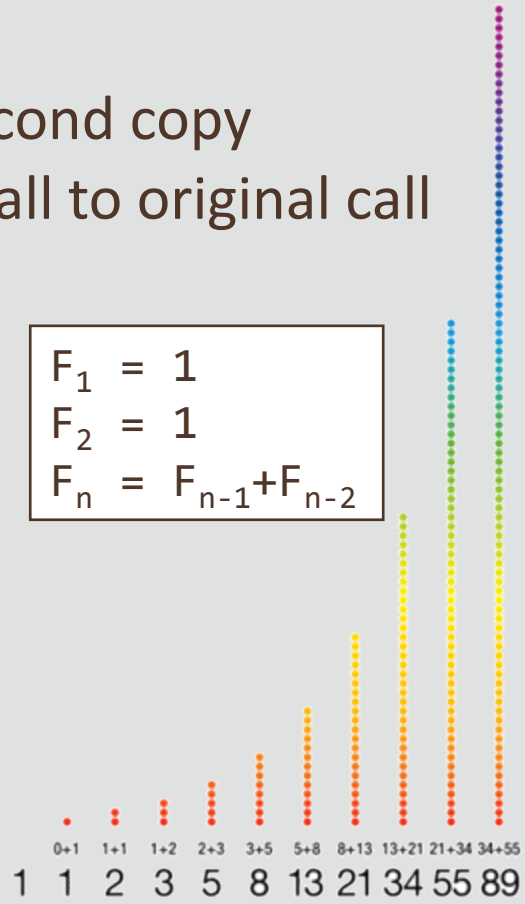
The Fibonacci sequence is commonly used to demonstrate recursion as it shows the benefits of recursion

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 copy
  - Returns values from the lowest recursive call to original call for each series of calls

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

$$F_1 = 1$$
$$F_2 = 1$$
$$F_n = F_{n-1}+F_{n-2}$$



Tracing Through Non-Linear Recursion

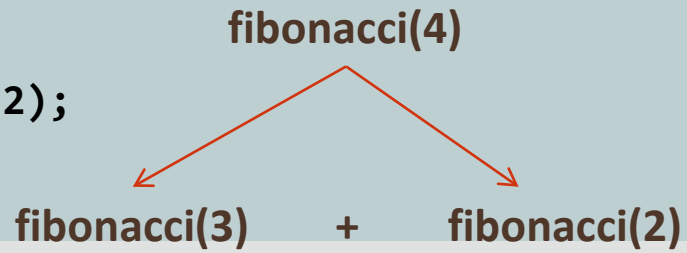
- Tracing through a non-linear recursive method is slightly more complex than linear recursive methods
- Although the concept of backward thinking is the same, it may be difficult to use a chart for keeping track of your calls
- Using a tree diagram is more practical for tracing this type of recursion

A tree is useful as you can keep track of all of the branches of calls and the values created and stored there.

Trace a Call to fibonacci(4)

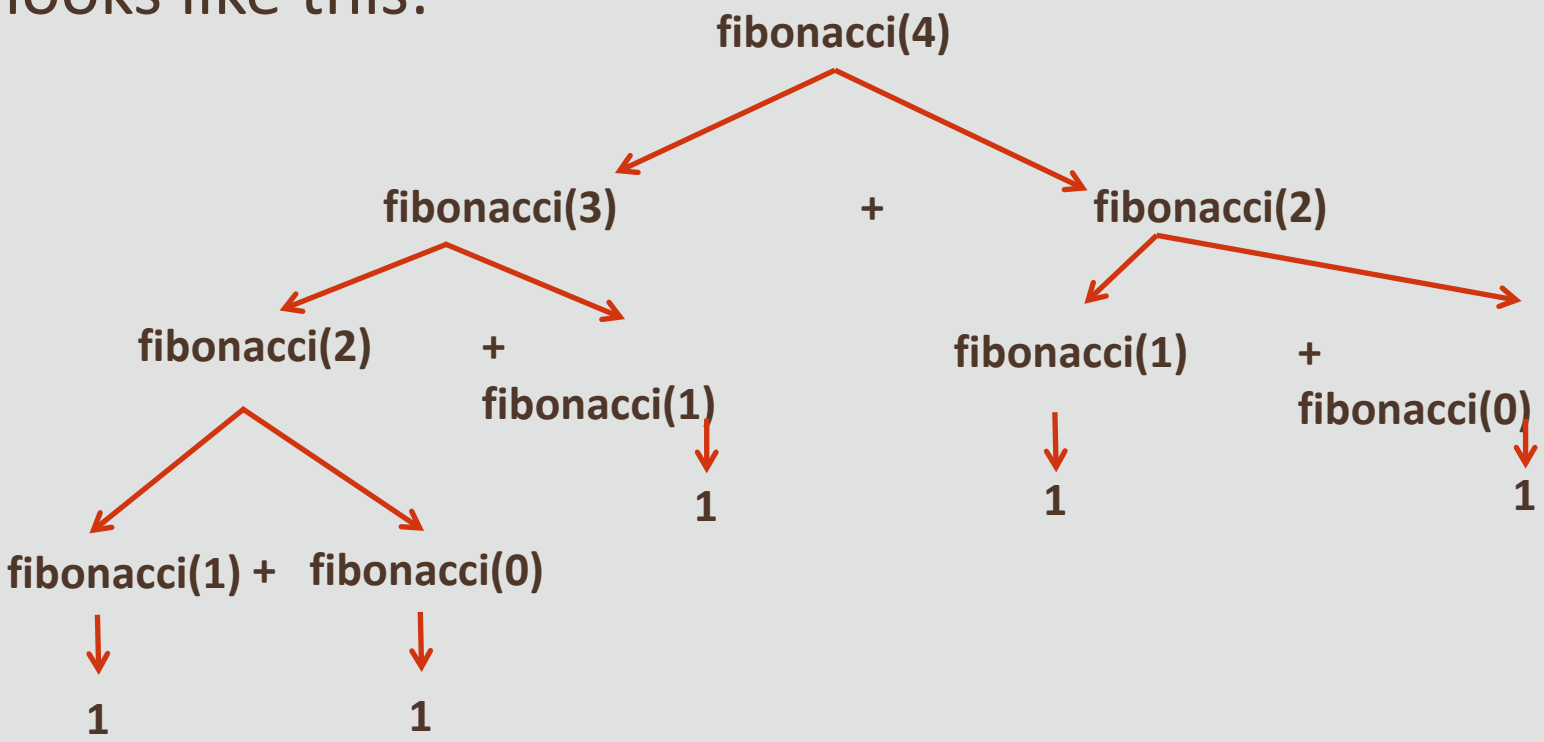
- Trace a call to fibonacci(4) to find out what the returned value is
- When you make a call where d = 4, you get this result: fibonacci(4) = fibonacci(3) + fibonacci(2)

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



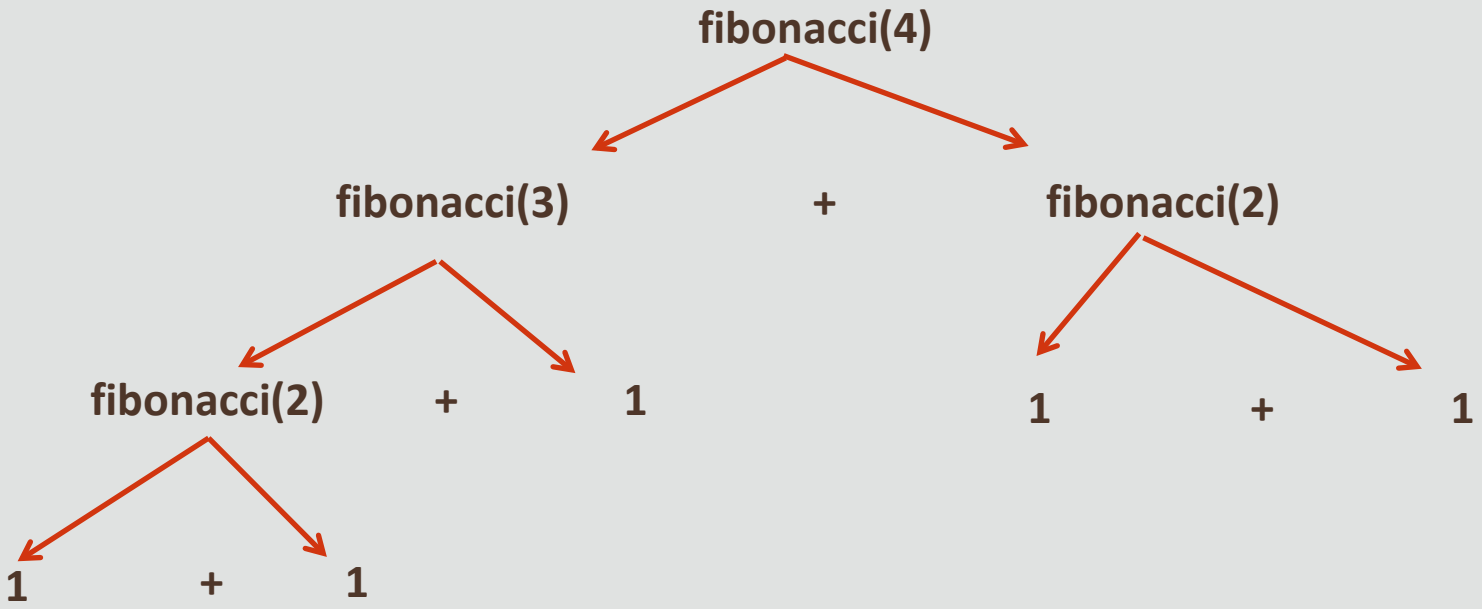
Continue to Trace Through to Base Case

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



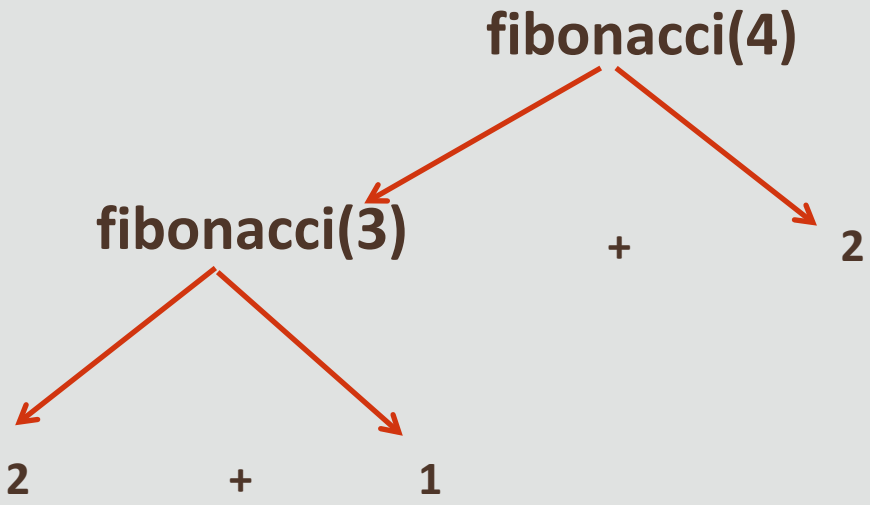
Continue to Trace Through to Base Case

- The new tree looks like this:



Continue to Trace Through to Base Case

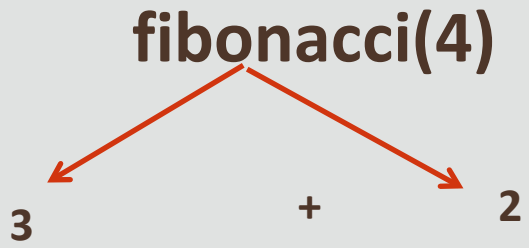
- Since fibonacci(2) = 1+1 = 2, change it in the tree:



There are many useful algorithms that use recursion such as Towers of Hanoi and Quick Sort.

Fibonacci Problem Conclusion

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



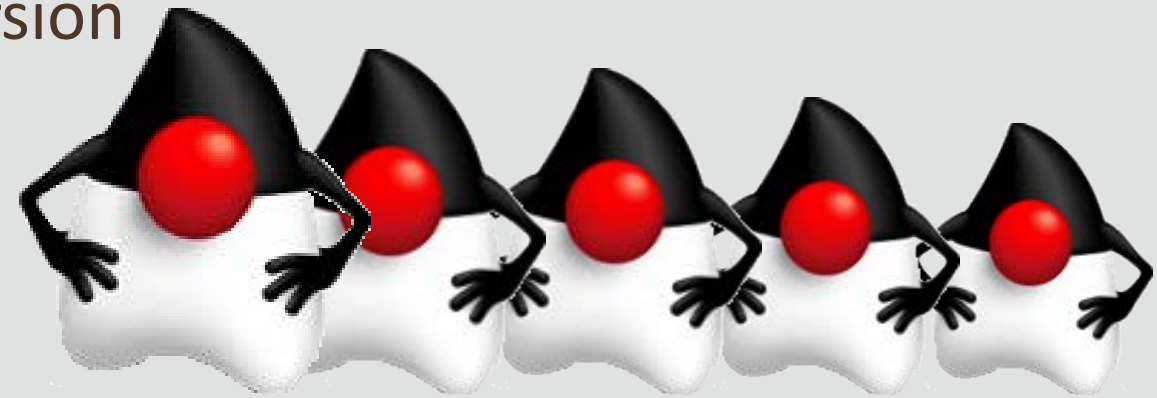
- Finally, replace fibonacci(4) with its return value (3+2=5)
- The conclusion is that fibonacci(4) = 5

Recursion Pros and Cons

- Pros:
  - Once understood, often more intuitive
  - Simpler, more elegant code
- Cons:
  - Uses more method 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
  - 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


ORACLE
Academy