Recursion
What is Recursion?

- Recursion is a method where the solution to a problem depends on solutions to smaller instances of the same problem.

- The classic form of recursion is a method that calls itself.

- We can sometimes use recursion to break down seemingly complex problems into straightforward, simple solutions.
Introduction to Recursion

- So far in this course we have had methods that call other methods.
  - For example, the main() method calls the square() method.
  - However in a recursive method is a method that calls itself.
A First Example

- This program simply counts from 1 to 3.
- Note that `upToThree` calls itself on line 10, this is a recursive call.

```java
public static void main(String[] args) {
    upToThree(1);
}

private static void upToThree(int i) {
    if(i >= 3) {
        System.out.println("Base case, i=" + i);
    } else {
        System.out.println("Recursive case, i=" + i);
        upToThree(i + 1);
    }
}
```
First Example

- The method is called first with the argument 1.
- At line 6 the ‘base case’ is checked.
  - The base case is the condition, once met, we stop making recursive calls.
- Since the base case is not satisfied, we jump to the recursive case.

```java
class Example {
    public static void main(String[] args) {
        upToThree(1);
    }

    private static void upToThree(int i) {
        if (i >= 3) {
            System.out.println("Base case, i=" + i);
        } else {
            System.out.println("Recursive case, i=" + i);
            upToThree(i + 1);
        }
    }
}
```
First Example

- Once in the branch for the recursive case, the message is printed indicating that we are going to make recursive call.

- Then we call `upToThree` with a value of 2.

```java
public static void main(String[] args) {
    upToThree(1);
}

private static void upToThree(int i) {
    if(i >= 3) {
        System.out.println("Base case, i=" + i);
    } else {
        System.out.println("Recursive case, i=" + i);
        upToThree(i + 1);
    }
}

Recursive case, i=1
First Example

- This call to `upToThree` with a value of 2 is checked against the base case condition.

- It is not satisfied, so we again enter the recursive case.

- The message is printed, i is incremented passed to yet another call of `upToThree`

```java
public static void main(String[] args) {
    upToThree(1);
}

private static void upToThree(int i) {
    if (i >= 3) {
        System.out.println("Base case, i=\" + i + ");
    } else {
        System.out.println("Recursive case, i=\" + i + ");
        upToThree(i + 1);
    }
}
```

Recursive case, i=1
Recursive case, i=2
First Example

- This call to `upToThree` with a value of 3 is checked against the base case condition.
- The base case is satisfied!
- The message is printed, and we return. No more recursive calls.

```java
public static void main(String[] args) {
    upToThree(1);
}

private static void upToThree(int i) {
    if (i >= 3) {
        System.out.println("Base case, i=" + i);
    } else {
        System.out.println("Recursive case, i=" + i);
        upToThree(i + 1);
    }
}
```

Recursive case, i=1
Recursive case, i=2
Base case, i=3
Visualizing Recursion

- To understand how recursion works, it helps to visualize what’s going on.

- To visualize, we can utilize what we know about the call stack.
  
  - Remember, the call stack is a stack data structure we use to manage memory for our method calls.
  
  - We push a ‘stack frame’ onto the call stack for every method call.
  
  - We pop the ‘stack frames’ off the call stack once that method completes execution.
Visualizing Recursion

- If a method calls itself recursively, you just push another copy of the method onto the stack.

- Let's look at our simple recursion example again and see what that call stack looks like...

http://goo.gl/q4yBte
Thinking Recursively

- A common computer programming tactic is to divide a problem into sub-problems of the same type as the original, solve those sub-problems, and combine the results.

- This is often referred to as the divide-and-conquer method.

- The job of the recursive cases can be seen as breaking down complex inputs into simpler ones.

- In a properly designed recursive function, with each recursive call, the input problem must be simplified in such a way that eventually the base case must be reached.
Rules for Writing Recursive Methods

- Rule 1: **Have a Base Case**
  - You must always have some base case which can be solved without recursion

- Rule 2: **Make Progress**
  - For cases that are to be solved recursively, the recursive call must always be a case that makes progress toward the base case.
Importance of a Base Case

- Consider this code...

```java
public static void main(String[] args) {
    noBaseCase(4);
}

private static void noBaseCase(int i) {
    if(i == 3) {
        System.out.println("Base case, i=" + i);
    } else {
        System.out.println("Recursive case, i=" + i);
        noBaseCase(i + 1);
    }
}
```
Infinite Recursion

- Neglecting to write a base case, or testing for it incorrectly, can cause an infinite loop.
- StackOverflowError!
Why use Recursive Methods?

- In computer science, some problems are more easily solved by using recursive methods.
- For example:
  - Traversing through directories of a file system.
  - Traversing through a tree of search results.
  - Some sorting algorithms recursively sort data.
- Note that any problem solved with recursion can be solved with iteration, but sometimes its much more difficult!
Case Study: Factorials

- A factorial in mathematics is defined as the product of all positive integers less than or equal to a given number.

- Factorials can be denoted using the “!” character

- If n! is defined as the product of all positive integers from 1 to n, then:

  \[
  \begin{align*}
  0! &= 1 \\
  1! &= 1 \times 1 = 1 \\
  2! &= 2 \times 1 \times 1 = 2 \\
  3! &= 3 \times 2 \times 1 \times 1 = 6 \\
  4! &= 4 \times 3 \times 2 \times 1 \times 1 = 24 \\
  5! &= 5 \times 4 \times 3 \times 2 \times 1 \times 1 = 120
  \end{align*}
  \]
Case Study: Factorials

- We can therefore describe in pseudocode one solution to the problem:

  if \( n == 0 \), the factorial is 1
  if \( n >= 1 \), the factorial is \( n \times \text{factorial}(n - 1) \)

- So for \( 4! \) we could trace one possible solution:

  \[
  4! = 4 \times 3!
  \]
  \[
  3! = 3 \times 2!
  \]
  \[
  2! = 2 \times 1!
  \]
  \[
  1! = 1 \times 0!
  \]
  \[
  0! = 1
  \]
Case Study: Factorials

- So we could write this in Java so follows – note how our method refers back to itself in the return statement:

```java
public static int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```
Case Study: Factorials

```java
public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```
Case Study: Factorials

```java
public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```
Case Study: Factorials

```java
public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

```
main
int ans = factorial(2)
```
Case Study: Factorials

```java
public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

factorial(2)

main

int ans = factorial(2)
Case Study: Factorials

```java
public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

```
factorial(2)
```

```
main
int ans = factorial(2)
```
Case Study: Factorials

```java
public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

```java
factorial(2)

main

int ans = factorial(2)
```
Case Study: Factorials

```java
public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

```
factorial(2)
return 2 * factorial(1)
main
int ans = factorial(2)
```
Case Study: Factorials

public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}

factorial(1)

factorial(2)

return 2 * factorial(1)

main

int ans = factorial(2)
Case Study: Factorials

```java
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

```
main
int ans = factorial(2)
return 2 * factorial(1)
factorial(2)
factorial(1)
```
Case Study: Factorials

```java
public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

```
factorial(1)

factorial(2)
return 2 * factorial(1)

main
int ans = factorial(2)
```
Case Study: Factorials

```java
public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}

factorial(1)
return 1 * factorial(0)
factorial(2)
return 2 * factorial(1)
main
int ans = factorial(2)
```
Case Study: Factorials

```java
public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

factorial(0)
factorial(1)
return 1 * factorial(0)
factorial(2)
return 2 * factorial(1)
main
int ans = factorial(2)
Case Study: Factorials

```java
public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

```
factorial(0)
factorial(1)
return 1 * factorial(0)
factorial(2)
return 2 * factorial(1)
main
int ans = factorial(2)
```
Case Study: Factorials

```java
public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

```
factorial(0)
return 1

factorial(1)
return 1 * factorial(0)

factorial(2)
return 2 * factorial(1)

main
int ans = factorial(2)
```
Case Study: Factorials

public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}

factorial(1)
return 1 * 1

factorial(2)
return 2 * factorial(1)

main
int ans = factorial(2)
Case Study: Factorials

```java
public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

```
factorial(2)
return 2 * 1
main
int ans = factorial(2)
```
Case Study: Factorials

```java
public static void main()
{
    int ans = factorial(2);
}

public static int factorial(int n)
{
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return n * factorial(n-1);
    }
}

main
int ans = 2
```
Recursion Review

- Recursion is a technique where you design a method that refers back to itself.
- There are also recursive data structures (i.e. an object that is composed of instances of itself).
- A correct recursive function requires that you articulate a “base case” that specifies the end of the recursive process.
- If a base case is omitted then infinite recursion occurs, resulting in a condition known as “Stack Overflow” (the stack literally overflows with method calls and you run out of memory).
Recursion Review

- Recursion can sometimes make problem solving simpler given that recursive algorithms tend to be shorter and more concise.

- However, recursion can be incredibly inefficient, especially when computing very complex results (as we will see in the next section).
Case Study: Fibonacci Numbers

- The Fibonacci sequence is a series of numbers observed by the medieval Italian mathematician, Leonardo Fibonacci.

- The sequence starts with the numbers 0 and 1. Numbers further along in the sequence are equal to the sum of the previous two numbers. For example:

0
1
1 ( = 0 + 1)
2 ( = 1 + 1)
3 ( = 2 + 1)
5 ( = 3 + 2)
8 ( = 5 + 3)
Case Study: Fibonacci Numbers

- We can obtain the n-th Fibonacci number by applying the following algorithm:

  if n == 0, fibonacci_number = 0
  if n == 1, fibonacci_number = 1
  if n > 1, fibonacci_number = fibonacci(n-1) + fibonacci(n-2)
Case Study: Fibonacci Numbers

```java
public static long fibonacci( long n )
{
    if ( n == 0 || n == 1 )
        return n;
    else
        return fibonacci( n - 1 ) + fibonacci( n - 2 );
}
```
Case Study: Fibonacci Numbers
Programming Example

- Implement the Fibonacci numbers algorithm in a recursive manner.
- Count the number of method calls to compute the n-th fibonacci number.
- Print the count.
- See recursion/Fibonacci.java
Memoization is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

- We often do this in recursion to save work.

- A very common data structure to use for this purpose? A HashMap!
Programming Example

- Use memorization to optimize our fibonacci calculation.
- See recursion/
  FibonacciWithMemoization.java
In-class Exercise

- Write a palindrome tester function.