

Recursion

Recursion

- A recursive method has a *base case* (or several base cases) and a *recursive case*
 - In the base case, there are no recursive calls
 - In the recursive case, the method calls itself, but for a “smaller” task
- Recursive calls must eventually converge to a base case, when recursion stops
- .

Example 1

```
public String reverse (String s)
{
    if (s.length() < 2)
        return s;

    return reverse (s.substring (1)) +
        s.charAt(0);
}
```

Base case
(nothing to do)

Recursive case

ABCD

Take
substring

A

BCD

Reverse
substring

A

DCB

Append the
first char

DCBA

Example 2

```
public double pow (double x, int n)
{
    if (n == 0)
        return 1.0;

    double y = pow (x, n / 2);
    y *= y;
    if ( n % 2 != 0 )
        y *= x;
    return y;
}
```

Base case

Recursive case

Caution: NOT
`double y = pow(x, n / 2) * pow(x, n / 2);`

Need x^7

$7 / 2 == 3$

First get

$y = x^3$

Then square

$y * y = x^6$

7 is odd, so

$y * = x$

Example 3


```
ArrayList fruits = new ArrayList ( );  
fruits.add ("apples");  
fruits.add ("bananas");
```

```
ArrayList snacks = new ArrayList ( );  
snacks.add ("chips");  
snacks.add ("pretzels");
```

```
ArrayList food = new ArrayList ( );  
food.add ("Fruits");  
food.add (fruits);  
food.add ("Snacks");  
food.add (snacks);
```

```
System.out.println (food);
```

Recursive calls to
ArrayList's toString
method take place
here



Output:

[Fruits, [apples, bananas], Snacks, [chips, pretzels]]

Recursion vs. Iteration

```
public String reverse (String s)
{
    if (s.length () < 2)
        return s;

    return reverse (
        s.substring(1)) +
        s.charAt(0);
}
```

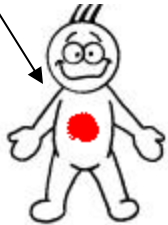
```
public String reverse (String s)
{
    String r = "";

    for (int i = s.length() - 1;
         i >= 0; i--)
        r += s.charAt (i);
    return r;
}
```

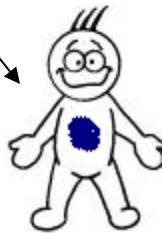
How old am I?



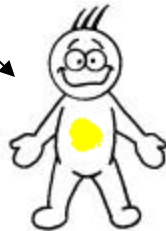
I don't know my age but I know that I am 6 years younger than red.



I don't know my age but I know that I am 3 years older than blue.



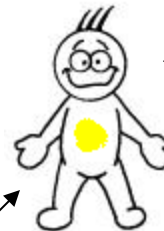
I don't know my age but I know that I am 4 years older than yellow.



I don't know my age but I know that I am 1 year younger than brown.



I am 25 years old



If brown is 25, I must be 24.



If yellow is 24, I must be 28.



If blue is 28, I must be 31.



If brown is 31, I must be 25.

A method is said to be recursive if it calls itself, either directly or indirectly.

That is, the method is used in its own definition.

Recursion can often be used to solve complex problems by reducing them to simpler problems of the same type. Put another way, **a recursive method solves a problem by returning increasingly simplified versions of the original problem until the simplest or base case is reached.**

When the base value is reached, it is plugged in the 2nd last method call. This allows another value to be calculated which in turn can be used to solve for the 3rd last method call and so on until the original method call's value is calculated.

A classic example of a recursive solution to a problem can be demonstrated using factorials.

The **factorial of 5** is calculated as follows:

$$5 * 4 * 3 * 2 * 1 = 120$$

The math. shorthand for writing the factorial of a number is to use the ! sign.

For example the factorial of 5 could be written **5!**

5! Is equal to $5 * 4 * 3 * 2 * 1$

5! Is also equal to $5 * 4!$ Extending this pattern we can create the following chart of factorials.

$$5! = 5 * 4!$$

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

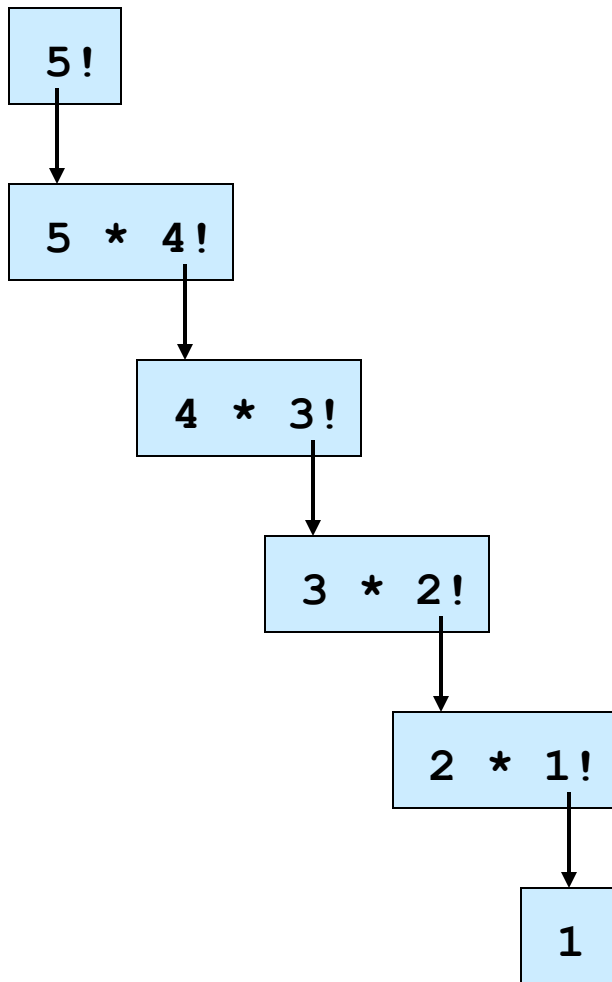
$$1! = 1 * 0!$$

Each of the preceding factorials can be rewritten in terms of another number multiplied by a factorial. This pattern works until we come to 0.

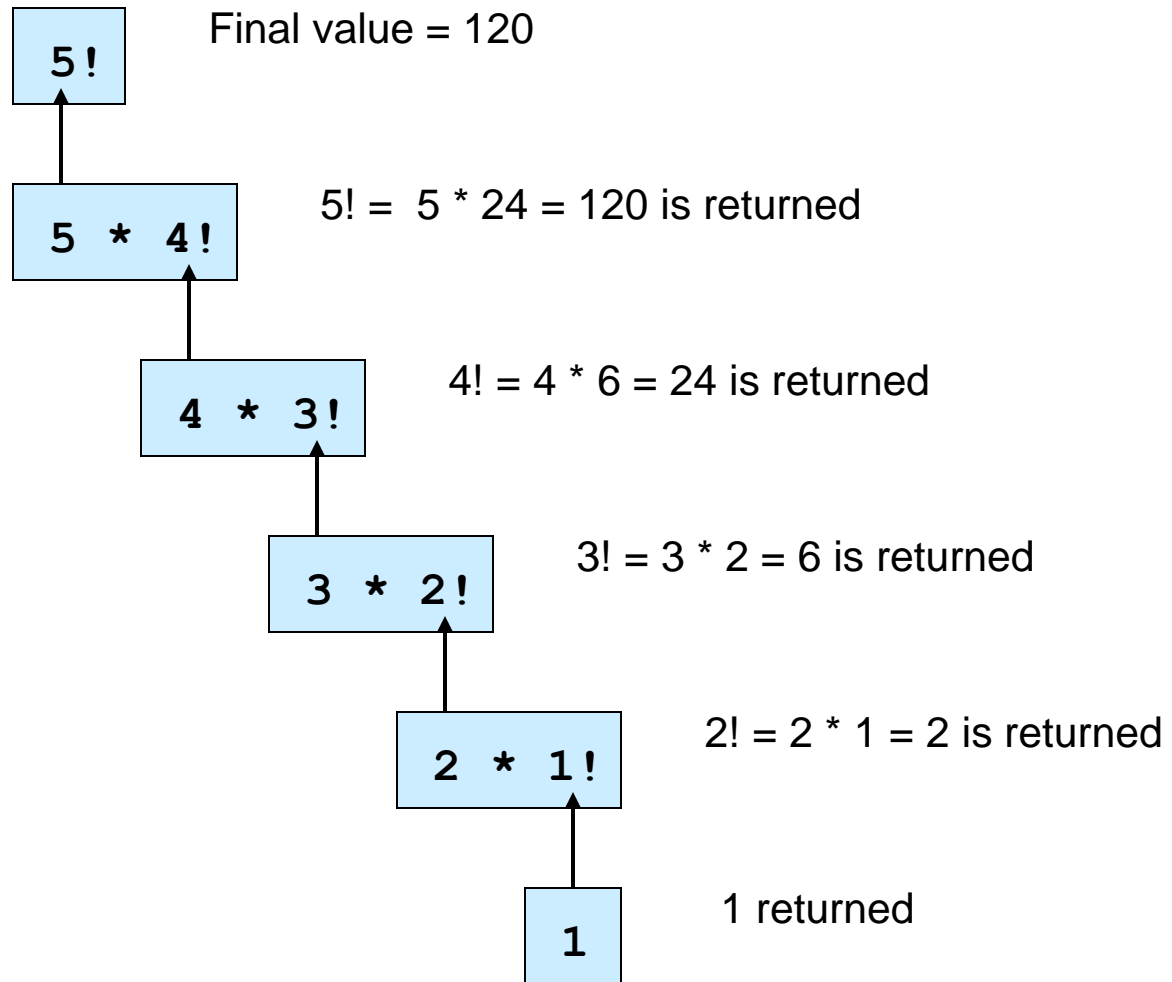
0! Has a value of 1. **We say 0! or 1 is the base case.**

This value can not be broken down any further or expressed as another factorial.

The following chart demonstrates the recursive flow of values from 5 down to the base case and back to the original method call.



(a) Procession of recursive calls.



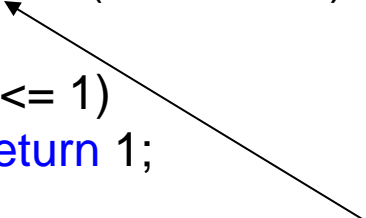
(b) Values returned from each recursive call.

Code demonstrating a recursive method 'factorial' which calculates 5!

RecursionFactorial.java

```
1 public class RecursionFactorial
2 {
3     public static void main(String args[])
4     {
5         System.out.println("The answer is " + factorial(5));
6     }
7     //The recursive method 'factorial' is defined and passed
8     //an argument of 5
9     static public int factorial(int number)
10    {
11        //determine if base case has been reached
12        if(number <= 1)
13            return 1;
14        //if number is greater than 1 it multiplies this
15        //number times the factorial of number - 1
16        else
17            return number * factorial(number - 1);
18    }
19 }
```

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

A black arrow originates from the 'factorial' parameter in the recursive call 'factorial(number - 1)' and points diagonally upwards and to the left, ending at the 'number' parameter in the method's signature 'factorial(int number)'. This illustrates how the method calls itself with a smaller value.

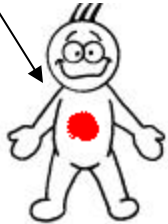
The method begins with a value of 5 being passed as the argument variable 5. Since $5 \leq 1$, the program flow skips to the next line and returns $5 * \text{factorial}(4)$. The value of $5 * \text{factorial}(4)$ is unknown, it is left in 'limbo' to be solved later. The flow of the program now concerns itself with $\text{factorial}(4)$. Since $4 \leq 1$, the program flow skips to the next line and returns $4 * \text{factorial}(3)$. This process repeats itself until the variable number is equal to 1.

Only when the value of number is equal to one does the method start returning a 'solid' value which can then start solving for previous method calls.

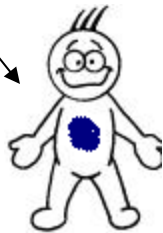
What is my Factorial



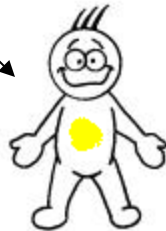
I don't know the value of 5! But I know it is equal to $5 * 4!$



I don't know the value of 4! But I know it is equal to $4 * 3!$



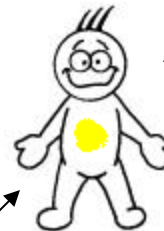
I don't know the value of 3! But I know it is equal to $3 * 2!$



I don't know the value of 2! But I know it is equal to $2 * 1!$



I am a base case with value 1
 $1! = 1 * 0!$ Which is equal to 1



If brown is 1, I must be $2 * 1 = 2$



If yellow is 2, I must be $3 * 2 = 6$



If blue is 6, I must be $4 * 6 = 24$



If red is 2, I must be $5 * 24 = 120$

Using Recursion to Calculate Exponents

Consider the exponent 2^4

It can be thought of as $2 * 2 * 2 * 2$

This could also be written as $2 * 2^3$

2^3 in turn could be written as $2 * 2^2$continuing the pattern to its logical conclusion

$$2^2 = 2 * 2^1$$

$$2^1 = 2^0$$

$$2^0 = 1$$

The base case in this example is point at which the exponent is 0.

Any number to the exponent 0 has a value of 1.

To solve an exponent recursively we take every exponent and rewrite it as a base number multiplied by an exponent one less than the previous.

Level

Initial

2^4

=

16

2

2

*

2^3

2

*

8

=

16

3

2

*

2^2

2

*

4

=

8

4

2

*

2^1

2

*

2

=

4

Base

2

*

2^0

= $2 * 1 = 2$

2 * 2

=

4

```
public class ExponentRecursion
{
    public static void main(String args[])
    {
        //The initial call to method power supplying two
        //arguments, 2 as the base and 4 as the exponent
        System.out.println( "The answer is " + power(2,4));
    }

    static int power(int base , int exponent)
    {
        //check to see if base case has been reached, exponent is 0
        if(exponent < 1)
            return 1;

        else
            //return the value of 2 multiplied by a recursive call to
            //power with a continually diminishing value of exponent
            return base * power(base, exponent - 1);
    }
}
```


String Reversal Using Recursion

In order to examine the next algorithm which rewrites a given word backwards, we need to know the function of three methods within the String class.

length() is method that simply returns the number of characters in a string as an integer.

```
int x ;
```

```
String word = "hello"
```

```
x = word.length()
```

In this case the value of x would be 5

```
public class StringLength
{
    public static void main(String args[])
    {
        String word = "hello";
        int x;
        x = word.length();
        System.out.println("The number of letters in the word are " + x);
    }
}
```

The String Method substring()

This String method takes an integer as an argument and returns the portion of the word beginning at a particular index in the String.

Remember that Strings are unique variables that are made up of an array of characters.

In the word 'hello', the individual letters correspond to the following indices(plural of index)

Index	Letter
0	'h'
1	'e'
2	'l'
3	'l'
4	'o'

Index	0	1	2	3	4	5
Word1	h	e	l	l	o	
Word2	t	o	d	a	y	
Word3	C	a	n	a	d	a

Using the above strings, some examples of the method substring would be as follows:

Word1.substring(1)	—————→	'ello'
Word1.substring(3)	—————→	'lo'
Word2.substring(2)	—————→	'day'
Word2.substring(3)	—————→	'ay'
Word3.substring(0)	—————→	'Canada'
Word3.substring(5)	—————→	'a'

The String Method `charAt()`

This method takes an integer as an argument and returns the letter at that particular index.

For example using the string variable `word = "hello"`, the output of `word.charAt(0)` would be `'h'`

`word.charAt(4)` would be `'o'`

This method can take a string literal rather than a variable if desired.

`"hello".charAt(1)` would output `'e'`

`"Waterloo".charAt(4)` would output `'r'`

Back to Recursion...

In this example we will take a given word and write it backwards.

music **cisum**

We will do this by using the string methods discussed previously within a recursive method we will call reverse.

Our algorithm must address these issues:

- How long is the word?
- What is the base case of the recursion?
- How do we reduce the outputs of the method calls so that it eventually reaches the base case.

The answer to the first question can be answered by simply applying the `length()` method

```
String word = "music"
```

```
int x = word.length()    //the value of x would be 5
```

In order to write a word backwards, we must establish what the final letter of the word is so that we can put it at the beginning.

The final letter of the word can be discovered by trimming letters off the initial word until there is a substring of only one character.

Our base case will therefore be a substring with a length ≤ 1

The syntax would look something like this:

```
If(word.length() <= 1  
    return word;
```

By grabbing a substring at index **one** we create a new copy of word that begins on the 2nd letter of the initial word.

By repeating this process until word.length = 1 we finally come to the last letter of the word

Music		cisuM
	+ charAt(0) which is M	
usic		cisu
	+ charAt(0) which is u	
sic		cis
	+ charAt(0) which is s	
ic		ci
	+ charAt(0) which is i	
c		

Once the base case is reached, the `charAt(0)` syntax adds the first letter of each substring. As the effect cascades back up, the effect is the letters are added in reverse order.

```
public class StringReverse
{
    public static void main(String args[]) {
        String word = "music";

        System.out.println("The word reversed is " + reverse(word));
    }
    public static String reverse(String word)
    {
        if(word.length() <= 1)
            return word;
        else
            return reverse(word.substring(1)) + (word.charAt(0));
    }
}
```


Recursion-Summary Notes

- All recursive definitions have to have a terminating case
- If they didn't, there would be no way to terminate the recursive path
- Such a definition would cause *infinite recursion*
- This problem is similar to an infinite loop, but the non-terminating "loop" is part of the definition itself
- The non-recursive part is called the **base case**

- A method in Java can invoke itself; if set up that way, it is called a *recursive method*
- The code of a recursive method must be structured to handle both the base case and the recursive case
- Each call to the method sets up a new execution environment, with new parameters and local variables
- As always, when the method completes, control returns to the method that invoked it (which may be an earlier invocation of itself)

“The Da Vinci Code”

Have you ever seen the Fibonacci sequence? This is the sequence: 1, 1, 2, 3, 5, 8, 13, and 21. It is a sequence of numbers in which each number is equal to the sum of the previous two numbers.

In the novel, “The Da Vinci Code,” the author, Dan Brown, uses this sequence as part of the clues left behind by Jacques Saunière, the murdered curator.

Suppose that you are a friend of Jacques Saunière and you are with him on the day before he is murdered in the museum. You are told by him to write a Java program that will generate the first 8 numbers of the Fibonacci sequence by using recursion, and then give them to Mr. Ho. Mr. Ho will then make a long-distance phone call to Jacques’ granddaughter, Sophie Neveu, to tell her this secret account number: 1-1-2-3-5-8-13-21

Efficiency

```
public long fibonacci (int n)
{
    if (n < 2)
        return 1;
    else
        return fibonacci (n-2) +
            fibonacci (n-1);
}
```

fibonacci(100) takes 50
years to run

```
public long fibonacci (int n)
{
    long f1 = 1, f2 = 1, next;
    while (n > 2)
    {
        next = f1 + f2;
        f1 = f2;
        f2 = next;
        n--;
    }
    return f2;
}
```

Fibonacci (100) takes
50 milliseconds to run