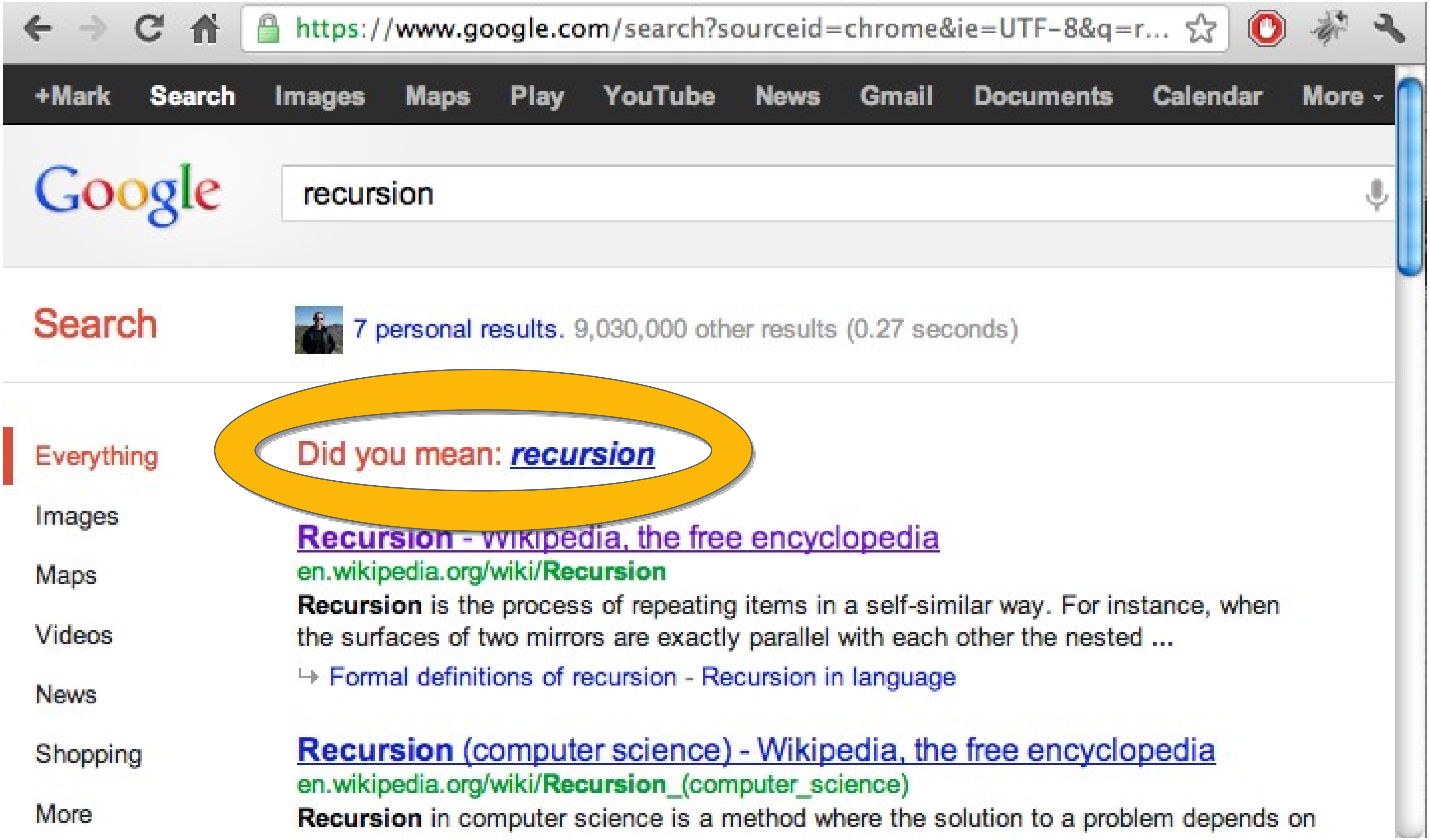
Object Oriented Programming with Java

Recursion

Recursion


https://www.google.com/search?sourceid=chrome&ie=UTF-8&q=r...
+Mark Search Images Maps Play YouTube News Gmail Documents Calendar More
Google
recursion
Search
7 personal results. 9,030,000 other results (0.27 seconds)
Everything
Images
Maps
Videos
News
Shopping
More
Did you mean: recursion
Recursion - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Recursion
Recursion is the process of repeating items in a self-similar way. For instance, when the surfaces of two mirrors are exactly parallel with each other the nested ...
Formal definitions of recursion - Recursion in language
Recursion (computer science) - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Recursion_(computer_science)
Recursion in computer science is a method where the solution to a problem depends on

What is Recursion?

Recursion generally means that something is defined in terms of itself

→ a function/method is recursive if it calls itself

→ data can be recursive if a class "has-a" field of its own type

Method Recursion

- We can call a method inside its own body

- The **recursive call** should logically solve a "smaller" problem

- We must have some way to stop, called a **base case**. It should be checked **before** the recursive call, otherwise, we have an infinite recursion!

Example: Factorial

- In mathematics, the factorial n! is defined as n!=n*(n-1)*…*2*1. It is defined for all non-negative numbers, and 0! = 1. Examples:

  5! = 1*2*3*4*5 → 5! = (1*2*3*4) * 5 → 5! = 4! * 5

- The **Base Case** is when n=0: we immediately know the answer. No recursion is necessary.
- The **Recursive Case** is when n>0: we know that whatever value n has, (n-1) will be one step closer to the base case of n=0.
  → assume the method is already correct; phrase n! = n*(n-1)!
  → call our method on (n-1), and multiply it by n.
  → let the recursive call do the rest!

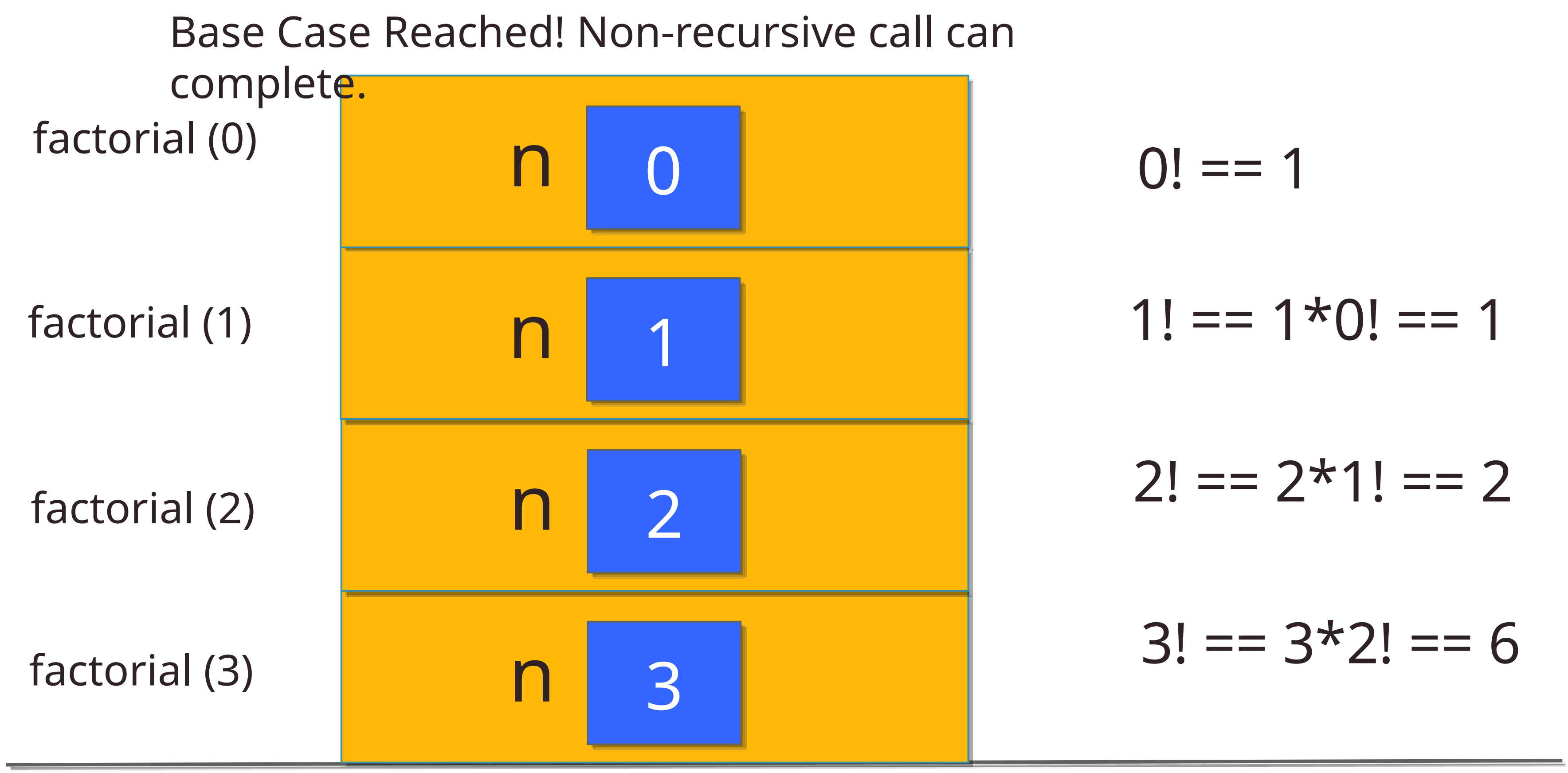
Example: Factorial

```java
public static int factorial (int n)
{
    //base case, no recursion
    if (n==0)
        return 1;

     else //recursive case: n! = n*(n-1)!
     {
         int nfact = n *  factorial(n-1);
         return nfact;
     }
}
```

Recursive Calls: Details

- When a method calls itself, each call is distinct (separate)
  → each separate call has its ***own copy of local data in Stack***
  → for `factorial`, each call has its own value for parameter `n`.



Recursion Recipe

- To use recursion, you might want to follow this pattern:

1. Identify the base cases: times when you already know the answer
2. Identify the recursive cases: times when you can define one step of the solution in terms of others. Is the recursive step using the method on a "smaller" problem? (needs to be yes!)
3. Write code for the base case *first*
4. Write code for the recursive case *second*

→ handle any error conditions like base cases: e.g., factorial shouldn't be called on negative numbers, so choose how to exit meaningfully.

Recursion Example: Fibonacci

- The fibonacci sequence looks like:

  1, 1, 2, 3, 5, 8, 13, …

  → Its first two elements are each 1.

  → the nth element is the sum of the previous two elements.
- We can think of them as array slots:

  fib[0]=1, fib[6]=13, etc.
- Or calculate them with function calls:

  fib(0)=1, fib(6)=13, etc.

Fibonacci Code

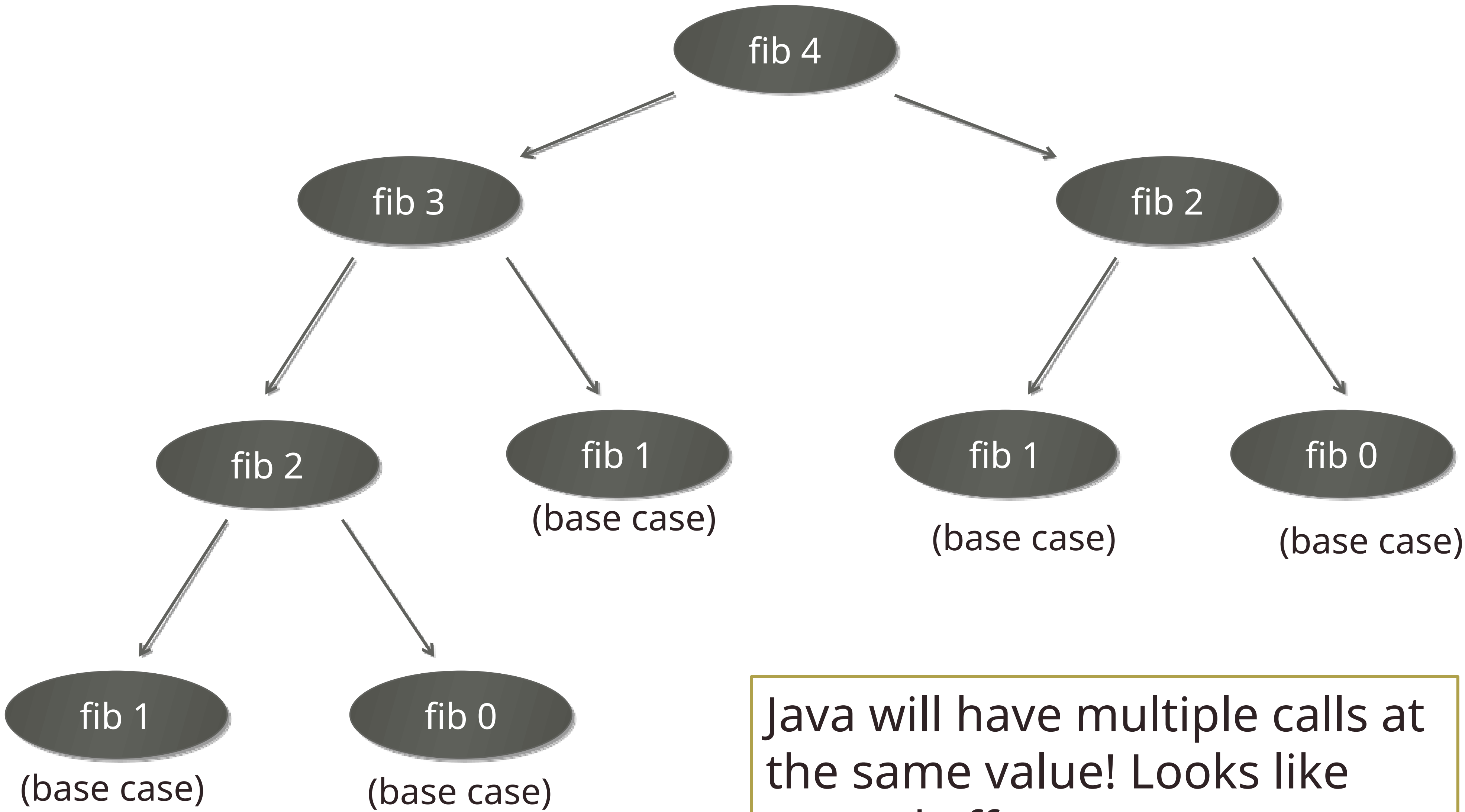
```
public static int fib (int n) {

   // base cases
   if (n==1 || n==0) { return 1; }

   //recursive case

   else {
      return fib(n-1) + fib(n-2) ;
   }
}
```

Visualizing Fibonacci Calls



Java will have multiple calls at the same value! Looks like wasted effort...

Iterative Version of Fibonacci

```
public static int fibIter (int n) {
  //base cases
  if (n==1 || n==0) { return 1; }

  //iterative cases
  int lower = 1;
  int higher = 1;
  for (int i = 2; i <=n; i++ ) {
     int temp = lower+higher;
     lower = higher;
     higher = temp;
  }
  return higher;
}
```

Adjusting memory size

If your program runs out of memory, you can have JVM allocate more memory to your process

- Use -Xss to set the thread stack size
- Use -Xmx to specify the maximum heap size
- Use -Xms to specify the initial Java heap size

Examples:

```
$ java -Xss1G myClass
```

```
$ java -Xss512M myClass
```

Considering Recursion

Recursion: **Pros**

- Sometimes much easier to reason about
- distinct method calls help separate concerns (separate our local data per call).
- Easy to maintain separate state (values) in each recursive call

Recursion: **Cons**

- Sometimes, lots of work is duplicated (leading to quite long running time)
- Overhead of a method call is more than overhead of another loop iteration

Considering Iteration

Iteration: **Pros**

- quick, barebones.
- Simpler control flow (we perhaps see how iterations will follow each other easier than with recursion)
- no stack overflow errors

Iteration: **Cons**

- some tasks do not lend well to iterative definitions (especially ones with lots of bookkeeping/state)
- We tend to be given mathematical, "recursive" definitions to problems, and then have to translate to an iterative version.

Recursion vs Iteration

So, which one is better? ...it depends on the situation

When might we prefer recursion?

When might we prefer iteration?

How, in general, might we try to convert a loop to a recursive method call?

Is there any problem that recursion or iteration can solve that we couldn't solve with the other?

Practice Problems

Create a recursive function that prints all the items of an int array in sequence. Example:

```
printArray(new int[]{6,-8,4,7,2})   →  6 -8 4 7 2
```

```
public static void printArray(int[] a, int index)
{
   if(index<0 || index>=a.length)
      return;
   System.out.println(a[index]);
   printArray(a,index+1);
}
```

- Modify the code to print in reverse order
- Can you think of an alternative solution for reverse order?
- What if the specs don't let you add the `index` parameter?

Practice Problems

Create a recursive function that prints all combinations of 0-9 digits in a variable-length string. Examples:

```
printDigit(2)         printDigit(3)                      printDigit(N)

    0 0                   0 0 0                              0 0 0 ... 0
    0 1                   0 0 1                              0 0 0 ... 1
    0 2                   0 0 2                              0 0 0 ... 2
    ...                   ...                                ...
    0 9                   0 0 9             ......           0 0 0 ... 9
    1 0                   0 1 0                              0 0 ... 1 0
    1 1                   0 1 1                              0 0 ... 1 1
    ...                   ...                                ...
    9 7                   9 9 7                              9 9 ... 9 7
    9 8                   9 9 8                              9 9 ... 9 8
    9 9                   9 9 9                              9 9 ... 9 9
```

Practice Problems

Create a method **`fill`** that fills in a **`grid`** (2D array of integers) with a value. The filling starts at **`seed`** (1D array with two coordinates) and spreads out to all neighbors having an **`oldValue`** which is replaced by the **`newValue`**.

```
fill(grid, oldValue, newValue, seed)
```

<u>Example</u>:

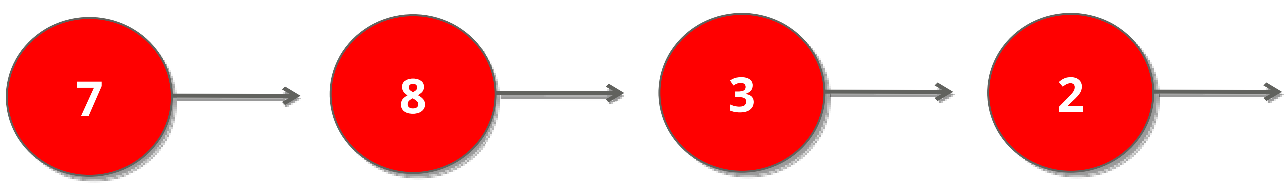
```
fill({{0,9,6,0,0,0},                          {{0,9,6,2,2,2},
      {0,7,4,0,0,0},                           {0,7,4,2,2,2},
      {1,3,0,7,8,0},                           {1,3,2,7,8,2},
      {0,9,5,1,0,1}}, 0, 2, {0,3})    →        {0,9,5,1,2,1}}
```

Data Recursion – Linked List

Data can also be recursive: when a class definition contains a field whose type is the same as the class being defined:

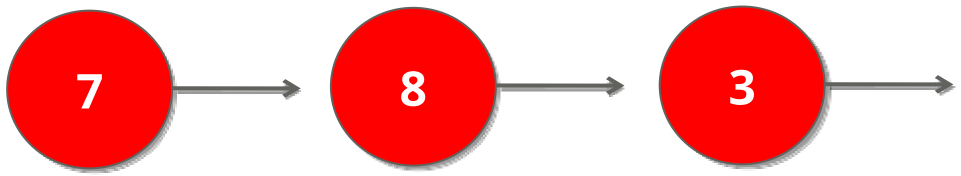
```
class Node {
  int value;
  Node next;
  …
}
```

recursive field



It looks a lot like the array **`int[] xs = {7,8,3,2};`**
→ could we implement the usual operations over our IntList that are usually available on arrays or ArrayList?

Data Recursion – Linked List Implementation




```java
public class Node {
 int value;
 Node next;
}

Node head = new Node();
head.value = 7;

head.next = new Node();
head.next.value = 8;

head.next.next = new Node();
head.next.next.value = 3;
```

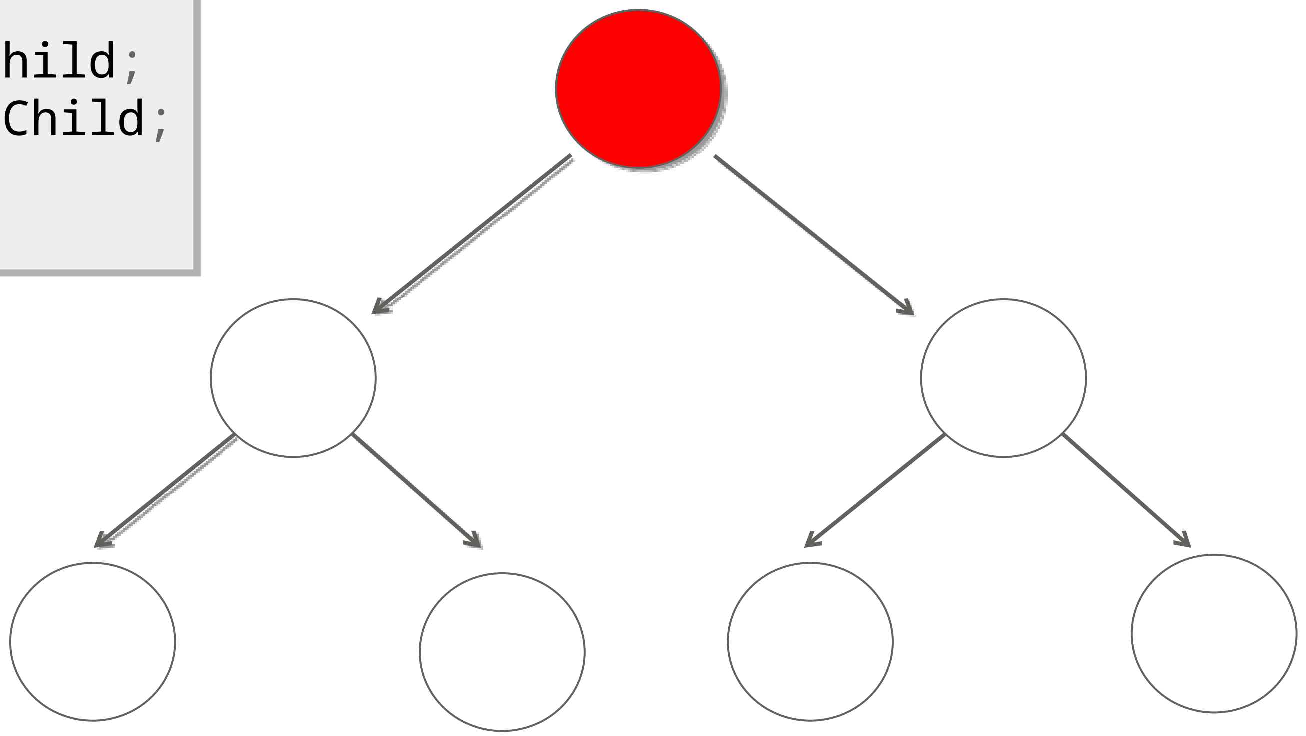
Lists can be very long though, so in practice, we don't do this manually but we use a loop

Data Recursion – Tree

What if our Node had two branches?

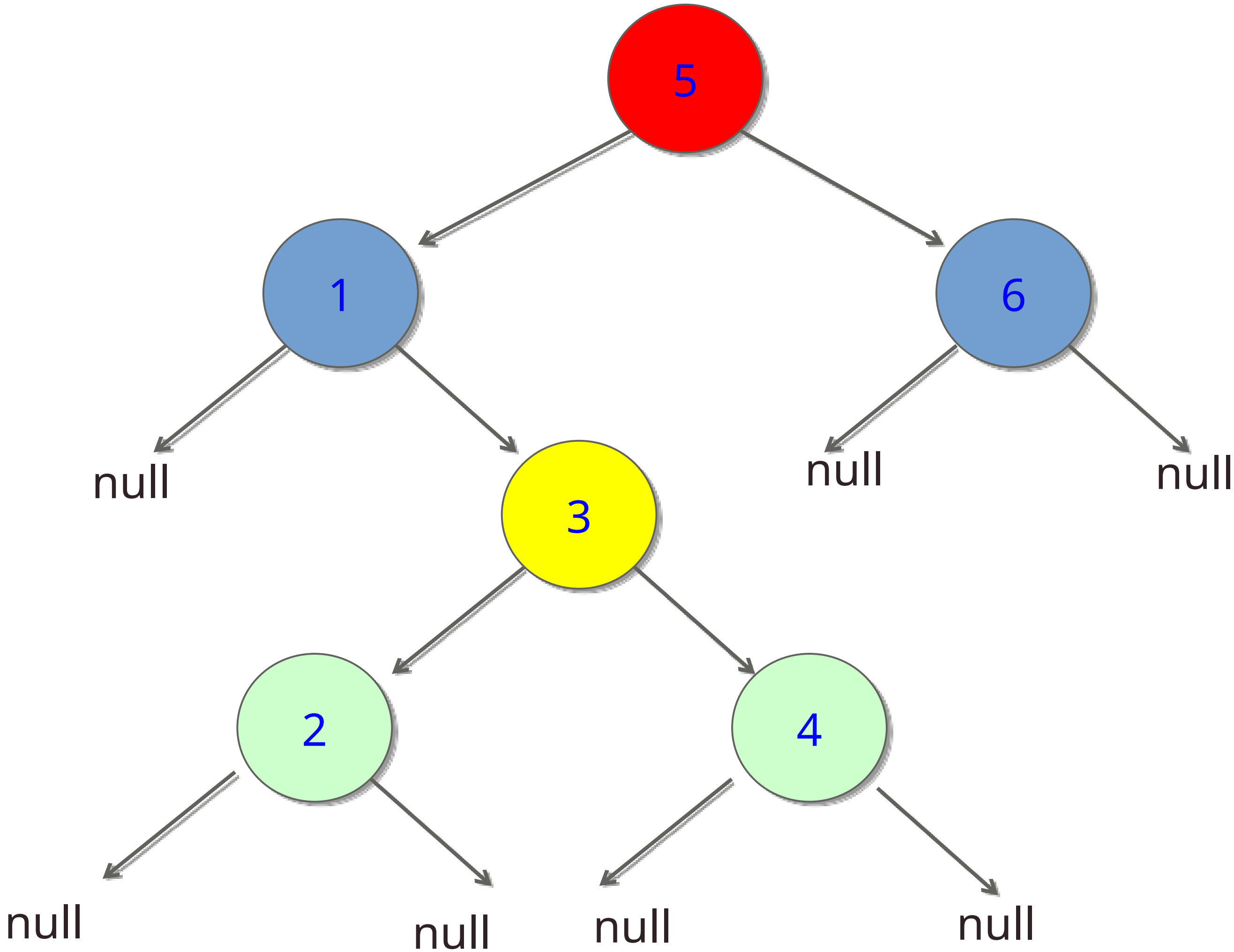
```
public class Node {
    public int value;
    public Node leftChild;
    public Node rightChild;
    …
}
```

recursive fields



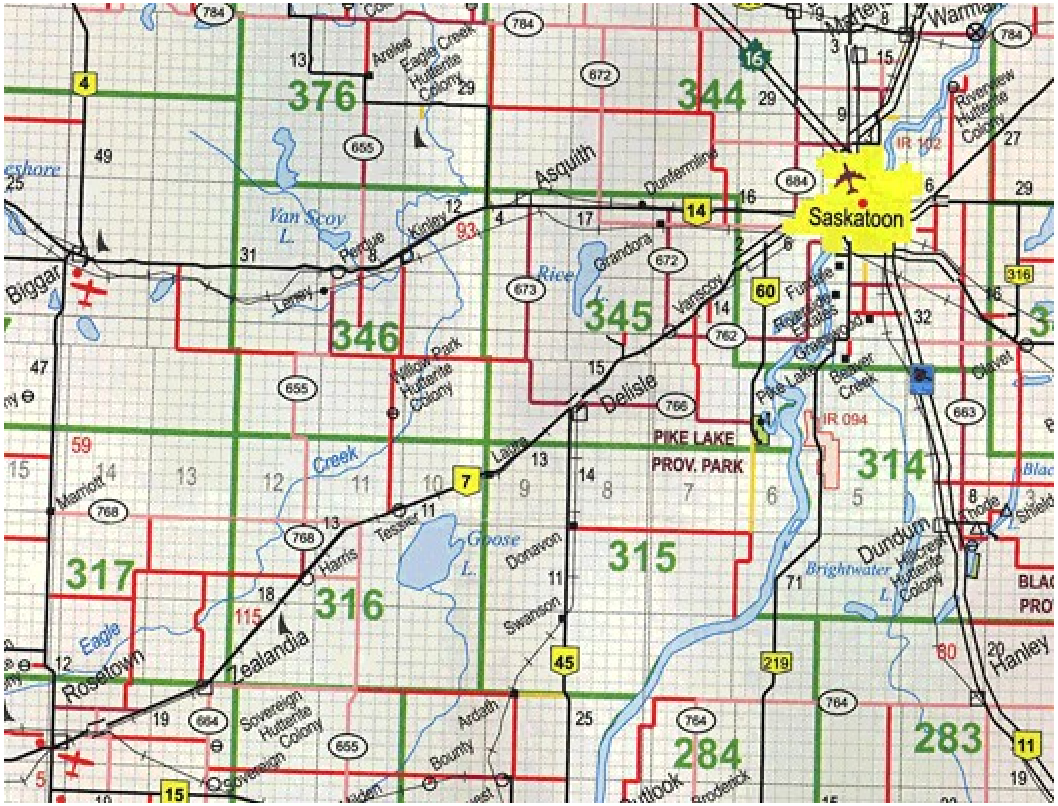
Base Cases in Data Recursion

We will again end the recursion with a base case: the null value



Recursion in action!

Recursion has many applications in search, sorting, tree traversal and graph problems.



Practice Problems

1. Create the following tree
2. Use recursion to print the tree in the following order: 1, 2, 3, 4, 5, 6

```java
public class Node {
    int value;
    Node leftChild;
    Node rightChild;
}
```