

- intro to recursion
- recursion exercise
- recursive arithmetic example
- recursion exercises assigned

Next week and homework

- next week: recursion II
- homework by next week:
 - download, run and understand this week's example programs
 - complete the recursion exercises assigned here
 - read Horstmann, sections 13.4 – 13.6

Lab

- exercises assigned this week (not collected or graded) / reviewed next week
- first graded lab assigned in 'Stack applications'. See Canvas for due date

Review last week

- looked at some applications examples as we finished covering stacks
 - reviewed reverse digits homework
 - introduced stack lab, on adding large numbers

Introduction to this week

- introduce the next major topic of recursion this week
 - introduce recursion
 - many examples
 - will practice recursion in this week's programming exercises

Intro to recursion

Objective: recursion is a different way to do iteration. Sometimes better, sometimes not as good

Dictionary definition: “For recursion, see recursion”

- in CS "a recursive method call is where a method calls itself" e.g.

```
public void foo(~~~~~)
{
    ...;
    foo(~~~~); //recursive call to foo()
}
```

– this is direct recursion

- or recursion may be indirect e.g.

```
public void wah(~~~~~)
{
    ...;
    bar(~~~~);
}

public void bar(~~~~)
{
    ...;
    wah(~~~~);
}
```

- we've already seen how a return address stack is used to implement ‘regular’, non-recursive method call and return:
 - on call: push return address and local vars
 - on return: pop stack
- we'll see exactly the same for recursion – it's not a special case

First example of a recursive problem - factorial

- everyone remember factorial? For example:

4! is $4 * 3 * 2 * 1$

3! is $3 * 2 * 1$

– and so on, down to a terminating special case, by definition:

0! is 1

- generalize to N! ($N \geq 0$)

N! is 1 if $N == 0$ otherwise

N! is $N * (N - 1) * (N - 2) * \dots * 1$

- see that this gives a recursive definition of factorial. From the line above,

– $(N - 1) * (N - 2) * \dots * 1$ is actually $(N - 1)!$

– e.g. $4!$ is $4 * 3!$

– factorial is defined in terms of itself

- so for N:

N! is 1 if $N == 0$ otherwise

N! is $N * (N - 1)!$

Two Rules for writing recursion

- there are two Rules for writing recursive methods:

1. there must be a terminating case e.g.

0! is 1

– termination – a non-recursive statement

2. recursive call should always be simpler than itself e.g.

$4!$ is $4 * 3!$

– $3!$ is simpler than $4!$

- see how recursive factorial is evaluated e.g.

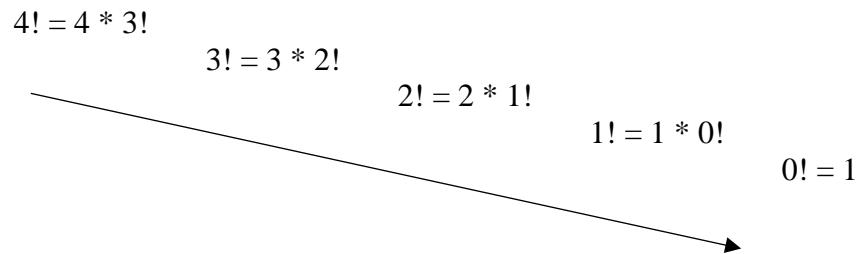


Figure 1 recursive descent

- see how Rule 2 is obeyed, with a simpler problem each time
- see how we reach a terminating condition to obey Rule 1
- recursive descent is where we apply Rule 2
 - (shown by the descending arrow in the drawing above)
 - we descend towards the simplest, terminating, non-recursive case
 - partial solutions are accumulated on the descent
- recursive ascent begins when we reach the Rule 1 terminating case
 - here partial results ascend the method calls:
 - (shown by the ascending arrow in the next drawing)

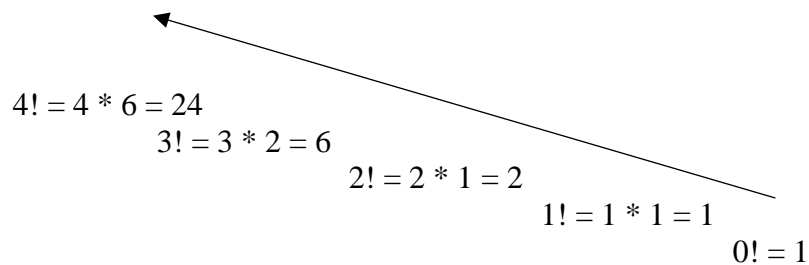


Figure 2 recursive ascent

- so $4!$ is 24

Recursive implementation of factorial

- here's a recursive implementation of factorial, will be used to show how the return address stack works with recursion

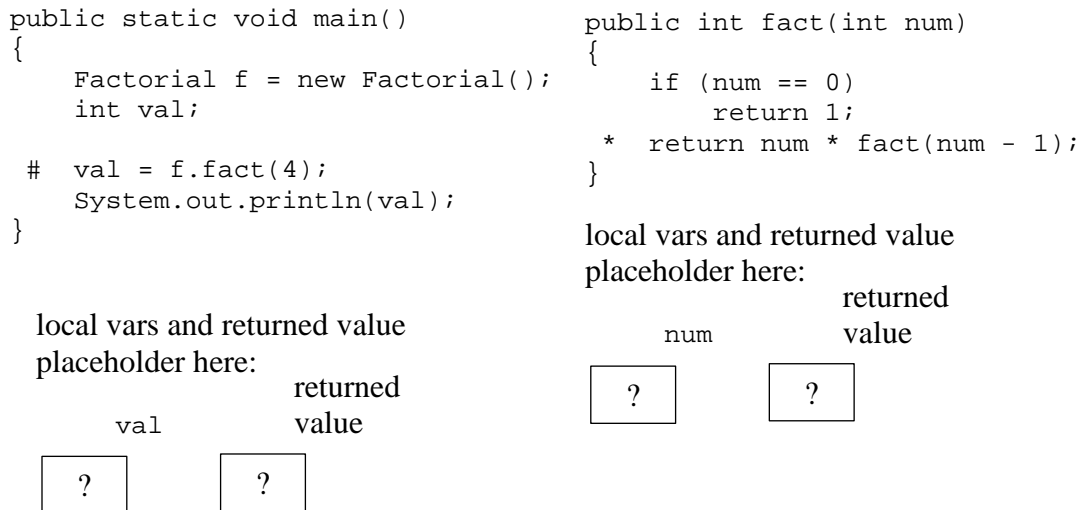


Figure 3 example program demonstrating recursion

- (BTW, notice how close this implementation is to the original recursive definition!)
- using # and * here to mark the return addresses
- since the method returns a value, then the returned value is effectively another local variable

Store return addresses and local vars on the return address stack

- a return address stack is used to implement method call and return
 - on a method call: push return address and any local vars, including a space for the method's returned value
 - on a return: pop return address and any local vars, updating any returned value first

Here's the recursive descent

- here's the return address stack at the end of recursive descent, where all the partial solutions have accumulated

5	*	1	?
4	*	2	?
3	*	3	?
2	*	4	?
1	#	val is ?	returned val is ?
call	return address	num	returned value

Figure 4 the return address stack at the end of recursive descent

- we push return address and local variables every method call
- until we reach the end of the recursive descent shown here, with no more method calls, so no more pushes
- so partial solutions accumulate downwards, as we push information to the stack

The recursive ascent

- reading the `fact()` method code, the recursive descent ends when `fact()` is called with `num` equal to 0, so that we execute a method return for the first time
 - here we return 1, so fill-in this returned value on the stack, then pop the stack to find the return address where we resume execution with these local variables
 - return address is the statement `return num * fact(num - 1);`
 - from the popped line, `num` is 1
 - the returned value of the method call `fact(num - 1)` is 1
 - so now we return `1 * 1` is 1
 - fill-in this returned value on the stack, then pop the stack to find where we resume execution
 - and so on
- so partial results ascend upwards in the method's returned value, as we pop the stack

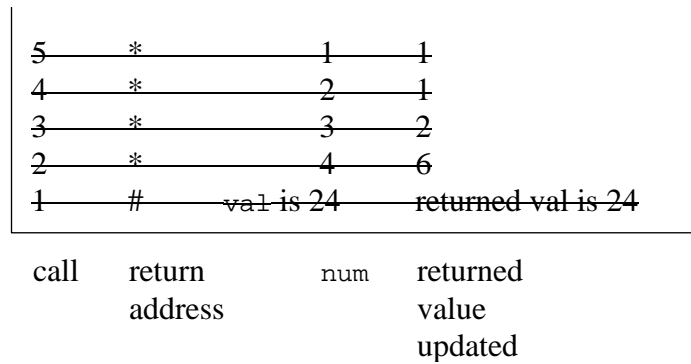


Figure 5 illustrating the recursive ascent

- (strikethrough like this is used here to indicate that a line is actually popped from the stack with each method return)
- notice that the method return value is updated each time, so that partial results ascend upwards as we pop the stack

Summary

- from Canvas, 'Recursion I' module, Example programs, download, read, run and understand my `Factorial` example program

Recursion exercise

Objective: do you understand recursion?

Here's a little recursion exercise

- the following method reads a character from the standard input stream each time it is called. Study the method then write the output on a piece of paper for yourself:

– input string is HELLO.

```
public void foo()  
{  
    char c;  
  
    c = (char) System.in.read();  
    if (c != '.')  
        foo();  
    System.out.print(c);  
}
```

– what's the output???

- did you see that output is actually the string reversed i.e.:

.OLLEH

- draw `c` on the stack to see why. Here it is:

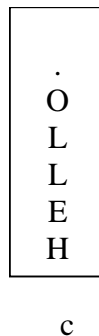


Figure 6 recursion exercise stack at the end of recursive descent

- recursive descent pushes the chars onto the stack in the order they are encountered, as shown above
- then recursive ascent pops each char off

Summary

- an excellent quick test and illustration. Cool!
- from Canvas, 'Recursion I' module, Example programs, download, read, run and understand my `Recursion exercise` example program. To run the program:
 - first create an `Exercise` object on the BlueJ workbench by running the default constructor
 - then run the `foo()` method
 - BlueJ opens the Terminal Window and allows you to enter the input string e.g. `HELLO`.
 - the method ends and output is shown in the Terminal Window as usual

Recursive arithmetic example

Objective: a final example of recursion

- early hardware provided only + and – arithmetic operators!
 - * was repeated addition
 - / was repeated subtraction

Multiplication * as recursive addition +

- * as recursive + is:
 - (assume non-negative)
$$a * b \quad \text{is } a \text{ if } b == 1 \text{ otherwise}$$
$$a + a * (b - 1)$$

e.g.

$$\begin{aligned} 3 * 4 &= 3 + 3 * 3 \\ 3 * 3 &= 3 + 3 * 2 \\ 3 * 2 &= 3 + 3 * 1 \\ 3 * 1 &= 3 \end{aligned}$$

- notice how the 2 Rules are followed:
 1. terminating case
 2. simpler call

Recursive implementation of *

- translate these 2 rules into Java, gives:

```
public int mult(int a, int b)
{
    if (b == 1)
        return a;
    return a + mult(a, b - 1);
}
```

- notice again how easily the code comes once we have the recursive definition!

Summary

- from Canvas, 'Recursion I' module, Example programs, download, read, run and understand my Multiplication example program
- be aware that recursion is expensive. Needs:
 - lots of memory – to stack everything
 - lots of time – method call and return overhead (stack and unstack) every call
- so an iterative solution may be more efficient
 - however, a recursive solution may be easier to understand and write for some problems
 - e.g. binary tree traversals, later

Recursion exercises assigned

Objective: your chance to write some recursive methods

- write and test these recursive methods for yourself
 - write recursive definition of problem first
 - then translate definition into a Java method
 - (use the same simple style as the previous example programs. Just a single method inside a class, run and test your method from BlueJ)

1. integer division by recursive subtraction

- e.g. $26 / 8$ is 3
 - terminology here : numerator / denominator
 - the algorithm is to recursively subtract denominator from the numerator until the denominator is bigger than the numerator e.g.

$$\begin{array}{rcl} 26 / 8 & = & 1 + 18 / 8 \\ & & 1 + 10 / 8 \\ & & 1 + 2 / 8 \\ & & 0 \quad \quad \quad \leftarrow \text{terminating case} \end{array}$$

- NOTE: integer division, so $2 / 8$ is 0, which is the terminating case. Recursive ascent begins from here, with partial results ascending upwards

2. exponentiation x^n by recursive multiplication

- e.g. 3^4 is $3 * 3 * 3 * 3 = 81$
 - assume integer only, valid ranges
 - recursive definition is given for this one:

$$\begin{array}{l} x^n = 1 \text{ if } n = 0 \\ x^n = x * x^{n-1} \text{ if } n > 0 \end{array}$$

3. palindromes by recursion

- e.g. “racecar”, “xxxx” are palindromes. Read the same forwards and backwards

- assume clean input
- return true if phrase is a palindrome

- algorithm:

```
  r      a      c      e      c      a      r
  ^                                  ^
  i                                  j
```

- start with index i, j at beginning and end of word, as above
- if characters are the same, call `palindrome ()` recursively with indexes moved
e.g.

```
  r      a      c      e      c      a      r
      ^              ^
      i              j
```

- and so on...

- hint: what is the terminating condition? When do we stop the recursion?

- when the indexes are equal, or cross over
- e.g. indexes meet, for an odd number of chars:

```
  r      a      c      e      c      a      r
                  ^
                  i
                  j
```

- e.g. indexes cross over, for an even number of chars:

```
  x      x      x      x
      ^      ^
      j      i
```

- hint: use a string for the characters e.g.

```
boolean palindrome(String phrase, int left, int right)
```

- remember, use `charAt()` to get a `char` from a `String`
- when running your method, BlueJ will want to see a `String` value as the first parameter. So double quotes `"` are required e.g. you would enter `"racecar"`, and so on

Summary

- reviewed next week
- (not collected or graded)

Next week and homework

- next week: recursion II
- homework by next week:
 - download, run and understand this week's example programs
 - complete the recursion exercises assigned here
 - read Horstmann, sections 13.4 – 13.6

Lab

- exercises assigned this week (not collected or graded) / reviewed next week
- first graded lab assigned in 'Stack applications'. See Canvas for due date