



RECURSION

Lecture 8

CS2110 – Fall 2016

Overview references to sections in text

2

- Note: We've covered everything in JavaSummary.pptx!
- What is recursion? 7.1-7.39 slide 1-7
- Base case 7.1-7.10 slide 13
- How Java stack frames work 7.8-7.10 slide 28-32

Next week's recitation

3

Study material on loop invariants here:

[www.cs.cornell.edu/courses/CS2110/2016sp/online/
index.html](http://www.cs.cornell.edu/courses/CS2110/2016sp/online/index.html)

Link: on links and on lecture notes pages of course website.

Do that BEFORE the MANDATORY recitation.

Then, do some problem solving as in this week's recitation

More work for us, not less

Doing this for first time in 2110. We will make mistakes.

Appreciate your tolerance and patience as we try something that studies show works better than conventional lectures

Why flip the class this way?

4

Usual way. 50-minute lecture, then study on your own. One hour? Total of, say, 2 hours.

Disadvantages:

- Hard to listen attentively for 50 minutes. Many people tune out, look at internet, videos, whatever
- Much time wasted here and there
- You don't always know just how to study. No problem sets, and if there are, no easy way to check answers.
- Study may consist of reading, not doing. Doesn't help.

Why flip the class this way?

5

Flipped way. Watch short, usually 3-5 minute, videos on a topic. Then come to recitation and participate in solving problems.

Disadvantage: If you don't study the videos carefully, you are wasting your time.

Advantages

- Break up watching videos into shorter time periods.
- Watch parts of one several times.
- In recitation, you get to DO something, not just read, and you get to discuss with a partner and neighbors, ask TA questions, etc.

== versus equals

6

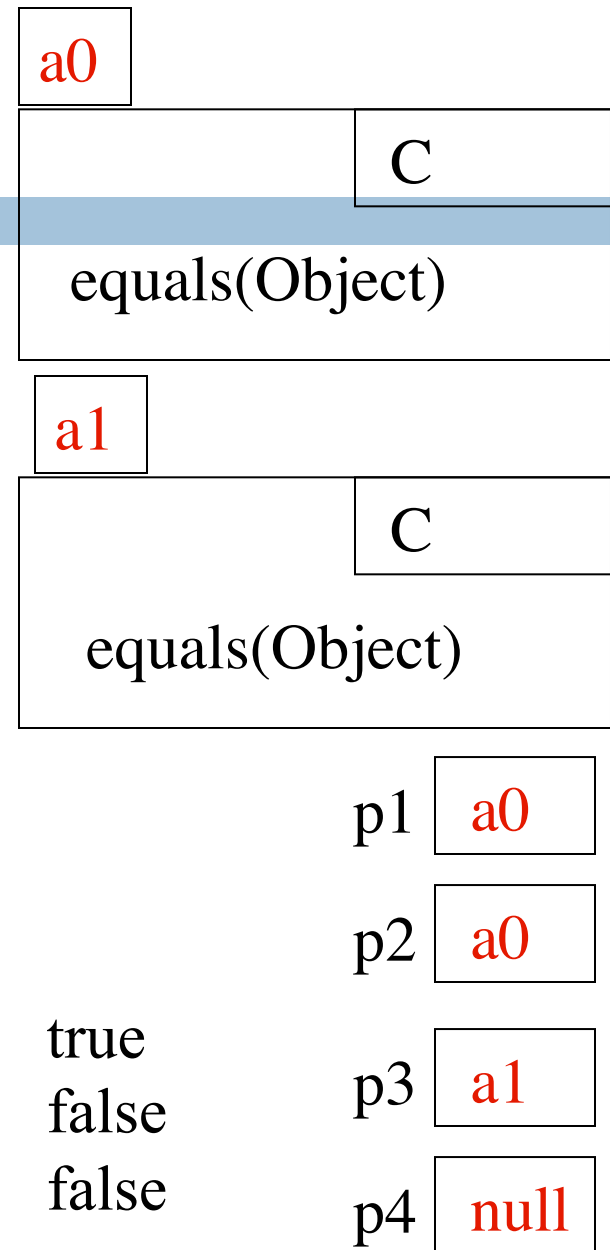
Use $p1 == p2$ or $p1 != p2$ to determine whether $p1$ and $p2$ point to the same object (or are both null).

Do NOT use $p1.equals(p2)$ for this purpose, because it doesn't always tell whether they point to the same object! It depends on how `equals` is defined.

$p4.equals(p1)$

Null pointer exception!

$p2 == p1$ true
 $p3 == p1$ false
 $p4 == p1$ false



Sum the digits in a non-negative integer

7

```
/** = sum of digits in n.  
 * Precondition: n >= 0 */  
public static int sum(int n) {  
    if (n < 10) return n;  
  
    // { n has at least two digits }  
    // return first digit + sum of rest  
    return sum(n/10) + n%10 ;  
}
```

sum calls itself!



E.g. $\text{sum}(7) = 7$

E.g. $\text{sum}(8703) = \text{sum}(870) + 3;$

Two issues with recursion

8

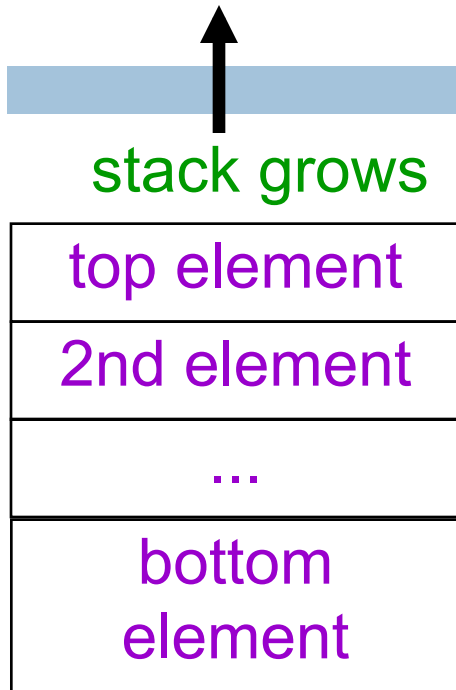
```
/** return sum of digits in n.  
 * Precondition: n >= 0 */  
public static int sum(int n) {  
    if (n < 10) return n;  
  
    // { n has at least two digits }  
    // return first digit + sum of rest  
    return sum(n/10) + n%10 + ;  
}
```

sum calls itself!

1. Why does it work? How does execution work?
2. How do we **understand** a given recursive method or how do we **write/develop** a recursive method?

Stacks and Queues

9



first	second	...	last
-------	--------	-----	------

Americans wait in a line. The Brits wait in a queue !

Stack: list with (at least) two basic ops:

- * Push an element onto its top
- * Pop (remove) top element

Last-In-First-Out (LIFO)

Like a stack of trays in a cafeteria

Queue: list with (at least) two basic ops:

- * Append an element
- * Remove first element

First-In-First-Out (FIFO)

Stack Frame

10

A “frame” contains information about a method call:

At runtime Java maintains a **a frame** **stack** that contains frames for all method calls that are being executed but have not completed.

local variables

parameters

return info

Method call: push a frame for call on **stack** assign argument values to parameters execute method body. Use the frame for the call to reference local variables parameters.

End of method call: pop its frame from the **stack**; if it is a function leave the return value on top of **stack**.

Frames for methods sum main method in the system

11

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return sum(n/10) + n%10;  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

Frame for method in the system
that calls method main

frame:

n _____
return info

frame:

r _____ args _____
return info

frame:

?
return info

Example: Sum the digits in a non-negative integer

12

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return sum(n/10) + n%10;  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

Frame for method in the system that calls method main: main is then called

main

r ____ args ____
return info

system

?
return info

Example: Sum the digits in a non-negative integer

13

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return sum(n/10) + n%10;  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

Method main calls sum:

main

n 824
return info

r ____ args ____
return info

system

?
return info

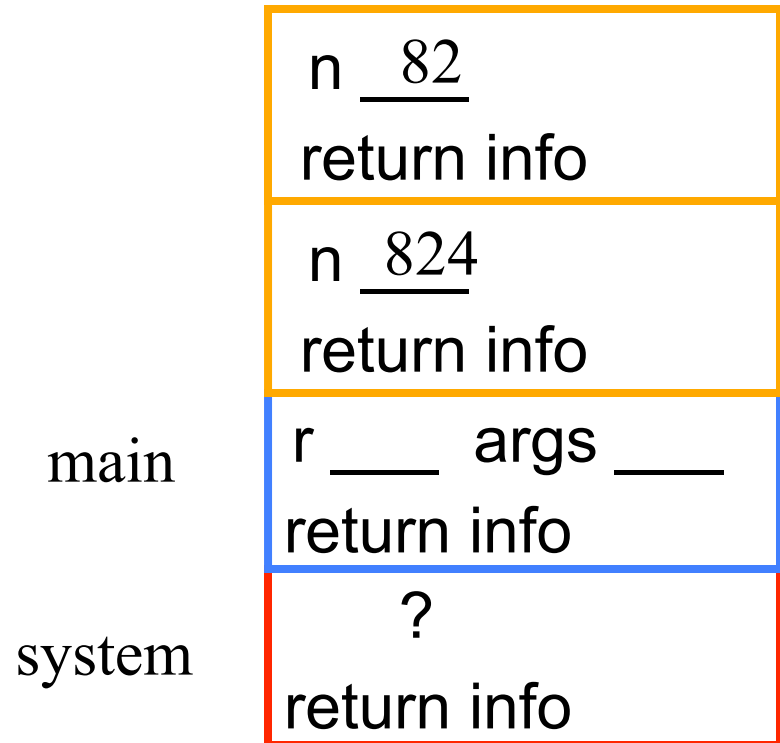
Example: Sum the digits in a non-negative integer

14

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return sum(n/10) + n%10;  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

$n \geq 10$ sum calls sum:



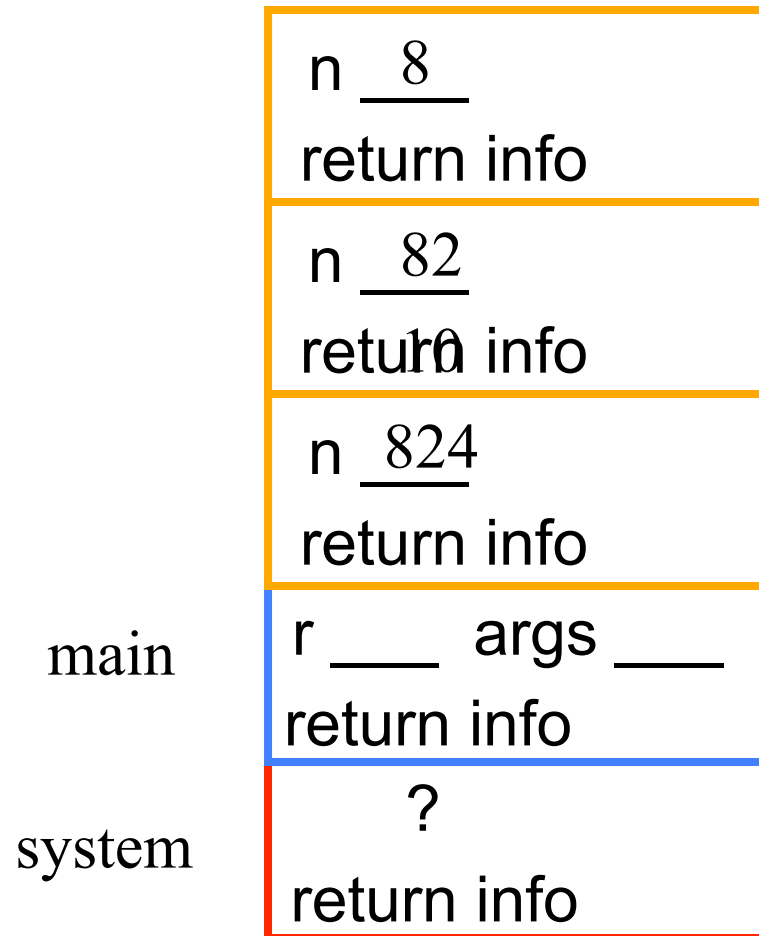
Example: Sum the digits in a non-negative integer

15

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return sum(n/10) + n%10;  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

$n \geq 10$. sum calls sum:



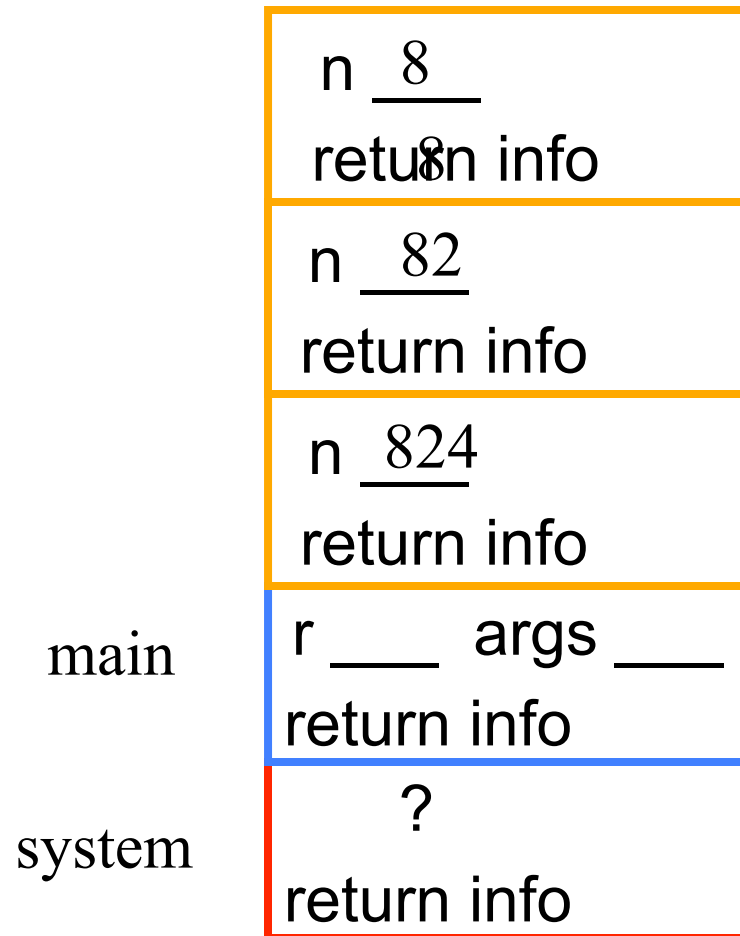
Example: Sum the digits in a non-negative integer

16

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return sum(n/10) + n%10;  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

n < 10 sum stops: frame is popped
and n is put on stack:



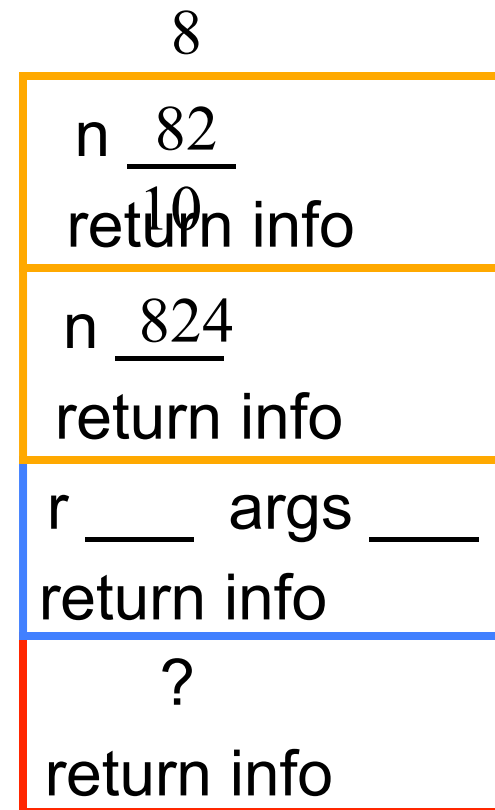
Example: Sum the digits in a non-negative integer

17

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return sum(n/10) + n%10;  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

main



Using return value 8 stack computes
 $8 + 2 = 10$ pops frame from stack puts
return value 10 on stack

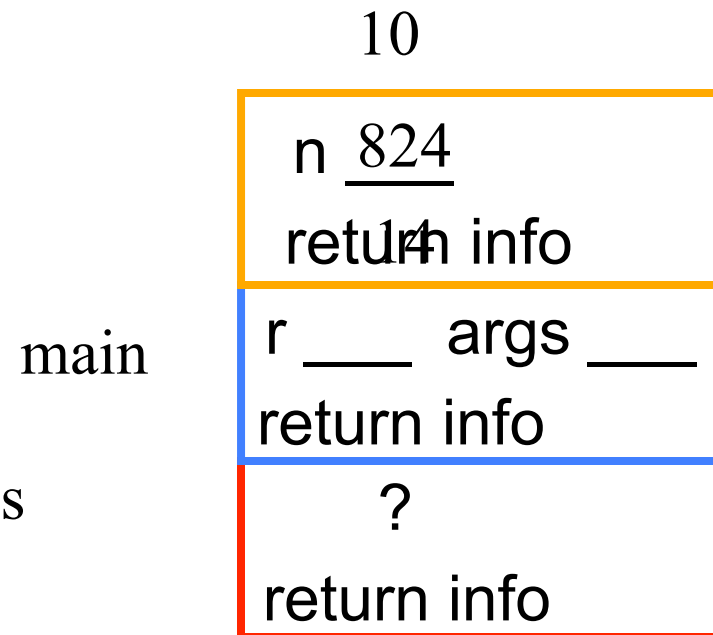
Example: Sum the digits in a non-negative integer

18

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return sum(n/10) + n%10;  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

Using return value 10 stack computes
 $10 + 4 = 14$ pops frame from stack
puts return value 14 on stack



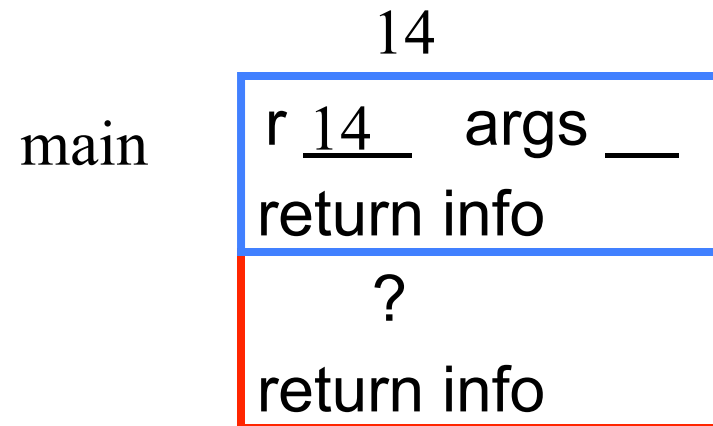
Example: Sum the digits in a non-negative integer

19

```
public static int sum(int n) {  
    if (n < 10) return n;  
    return sum(n/10) + n%10;  
}
```

```
public static void main(  
    String[] args) {  
    int r= sum(824);  
    System.out.println(r);  
}
```

Using return value 14 main stores
14 in r and removes 14 from stack



Memorize method call execution!

20

A frame for a call contains parameters, local variables, and other information needed to properly execute a method call.

To execute a method call:

1. push a frame for the call on the stack,
2. assign argument values to parameters,
3. execute method body,
4. pop frame for call from stack, and (for a function) push returned value on stack

When executing method body look in frame for call for parameters and local variables.

Questions about local variables

21

```
public static void m(...) {  
    ...  
    while (...) {  
        int d= 5;  
        ...  
    }  
}
```

```
public static void m(...) {  
    int d;  
    ...  
    while (...) {  
        d= 5;  
        ...  
    }  
}
```

In a call `m(...)`
when is local variable `d` created and when is it destroyed?
Which version of procedure `m` do you like better? Why?

Recursion is used extensively in math

22

Math definition of n factorial

$$0! = 1$$

$$n! = n * (n-1)! \quad \text{for } n > 0$$

E.g. $3! = 3*2*1 = 6$

Math definition of b^c for $c \geq 0$

$$b^0 = 1$$

$$b^c = b * b^{c-1} \quad \text{for } c > 0$$

Easy to make math definition into a Java function!

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

Lots of things defined recursively:
expression grammars trees

We will see such things later

Two views of recursive methods

23

- How are calls on recursive methods executed?

We saw that. Use this only to gain understanding / assurance that recursion works

- How do we understand a recursive method — know that it satisfies its specification? How do we write a recursive method?

This requires a totally different approach.

Thinking about how the method gets executed will confuse you completely! We now introduce this approach.

How to understand what a call does

24

Make a copy of the method spec,
replacing the parameters of the
method by the arguments

spec says that the
value of a call
equals the sum of
the digits of n

sum(654)

sum of digits of **n**

sum of digits of **654**

```
/** = sum of the digits of n.  
 * Precondition: n >= 0 */  
public static int sum(int n) {  
    if (n < 10) return n;  
    // n has at least two digits  
    return sum(n/10) + n%10 ;  
}
```


Understanding a recursive method

25

Step 1. Have a precise spec!

Step 2. Check that the method works in **the base case(s)**: Cases where the parameter is small enough that the result can be computed simply and without recursive calls.

If $n < 10$ then n consists of a single digit. Looking at the spec we see that that digit is the required sum.

```
/** = sum of digits of n.  
 * Precondition:  $n \geq 0$  */  
public static int sum(int n) {  
    if ( $n < 10$ ) return n;  
  
    // n has at least two digits  
    return sum(n/10) + n%10 ;  
}
```

Understanding a recursive method

26

Step 1. Have a precise spec!

Step 2. Check that the method works in **the base case(s)**.

Step 3. Look at the **recursive case(s)**. In your mind replace each recursive call by what it

does according to the method spec and verify that the correct result is then obtained.

```
return sum(n/10) + n%10;
```

```
return (sum of digits of n/10) + n%10;    // e.g. n = 843
```

```
/** = sum of digits of n.  
 * Precondition: n >= 0 */  
public static int sum(int n) {  
    if (n < 10) return n;  
  
    // n has at least two digits  
    return sum(n/10) + n%10 ;  
}
```

Understanding a recursive method

27

Step 1. Have a precise spec!

Step 2. Check that the method works in **the base case(s)**.

Step 3. Look at the **recursive case(s)**. In your mind replace each recursive call by what it does acc. to the spec and verify correctness.

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the pars of the method.

$$n/10 < n$$

```
/** = sum of digits of n.  
 * Precondition: n >= 0 */  
public static int sum(int n) {  
    if (n < 10) return n;  
  
    // n has at least two digits  
    return sum(n/10) + n%10 ;  
}
```

Understanding a recursive method

28

Step 1. Have a precise spec!

Important! Can't do step 3 without it

Step 2. Check that the method works in **the base case(s)**.

Step 3. Look at the **recursive case(s)**. In your mind replace each recursive call by what it does according to the spec and verify correctness.

Once you get the hang of it this is what makes recursion easy! This way of thinking is based on math induction which we don't cover in this course.

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the pars of the method

Writing a recursive method

29

Step 1. Have a precise spec!

Step 2. Write the **base case(s)**: Cases in which no recursive calls are needed Generally for “small” values of the parameters.

Step 3. Look at all other cases. See how to define these cases in terms **of smaller problems of the same kind**. Then implement those definitions using recursive calls for those **smaller problems of the same kind**. Done suitably point 4 is automatically satisfied.

Step 4. (No infinite recursion) Make sure that the args of recursive calls are in some sense smaller than the pars of the method

Examples of writing recursive functions

30

For the rest of the class we demo writing recursive functions using the approach outlined below. The java file we develop will be placed on the course webpage some time after the lecture.

Step 1. Have a precise spec!

Step 2. Write the **base case(s)**.

Step 3. Look at all other cases. See how to define these cases in terms **of smaller problems of the same kind**. Then implement those definitions using recursive calls for those **smaller problems of the same kind**.

The Fibonacci Function

31

Mathematical definition:

$$\begin{aligned} \text{fib}(0) &= 0 \\ \text{fib}(1) &= 1 \\ \text{fib}(n) &= \text{fib}(n - 1) + \text{fib}(n - 2) \quad n \geq 2 \end{aligned}$$

two base cases!

Fibonacci sequence: 0 1 1 2 3 5 8 13 ...

```
/** = fibonacci(n). Pre: n >= 0 */
static int fib(int n) {
    if (n <= 1) return n;
    // { 1 < n }
    return fib(n-2) + fib(n-1);
}
```



Fibonacci (Leonardo
Pisano) 1170-1240?

Statue in Pisa Italy
Giovanni Paganucci
1863

Check palindrome-hood

32

A String palindrome is a String that reads the same backward and forward.

A String with at least two characters is a palindrome if

- (0) its first and last characters are equal and
- (1) chars between first & last form a palindrome:

have to be the same

e.g. AMANAPLANACANALPANAMA

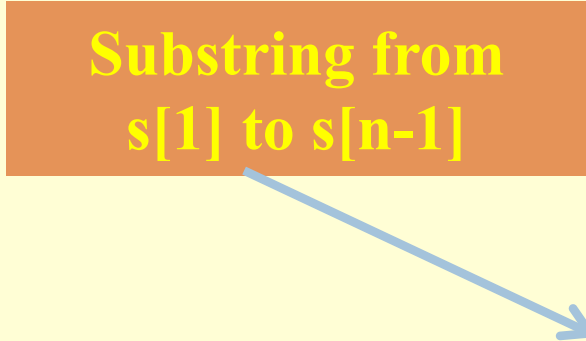
have to be a palindrome

A recursive definition!

Example: Is a string a palindrome?

33

```
/** = "s is a palindrome" */  
public static boolean isPal(String s) {  
    if (s.length() <= 1)  
        return true;  
  
    // { s has at least 2 chars }  
    int n= s.length()-1;  
    return s.charAt(0) == s.charAt(n) && isPal(s.substring(1,n));  
}
```



isPal("racecar") returns true

isPal("pumpkin") returns false

- A man a plan a caret a ban a myriad a sum a lac a liar a hoop a pint a catalpa a gas an oil a bird a yell a vat a caw a pax a wag a tax a nay a ram a cap a yam a gay a tsar a wall a car a luger a ward a bin a woman a vassal a wolf a tuna a nit a pall a fret a watt a bay a daub a tan a cab a datum a gall a hat a fag a zap a say a jaw a lay a wet a gallop a tug a trot a trap a tram a torr a caper a top a tonk a toll a ball a fair a sax a minim a tenor a bass a passer a capital a rut an amen a ted a cabal a tang a sun an ass a maw a sag a jam a dam a sub a salt an axon a sail an ad a wadi a radian a room a rood a rip a tad a pariah a revel a reel a reed a pool a plug a pin a peek a parabola a dog a pat a cud a nu a fan a pal a rum a nod an eta a lag an eel a batik a mug a mot a nap a maxim a mood a leek a grub a gob a gel a drab a citadel a total a cedar a tap a gag a rat a manor a bar a gal a cola a pap a yaw a tab a raj a gab a nag a pagan a bag a jar a bat a way a papa a local a gar a baron a mat a rag a gap a tar a decal a tot a led a tic a bard a leg a bog a burg a keel a doom a mix a map an atom a gum a kit a baleen a gala a ten a don a mural a pan a faun a ducat a pagoda a lob a rap a keep a nip a gulp a loop a deer a leer a lever a hair a pad a tapir a door a moor an aid a raid a wad an alias an ox an atlas a bus a madam a jag a saw a mass an anus a gnat a lab a cadet an em a natural a tip a caress a pass a baronet a minimax a sari a fall a ballot a knot a pot a rep a carrot a mart a part a tort a gut a poll a gateway a law a jay a sap a zag a fat a hall a gamut a dab a can a tabu a day a batt a waterfall a patina a nut a flow a lass a van a mow a nib a draw a regular a call a war a stay a gam a yap a cam a ray an ax a tag a wax a paw a cat a valley a drib a lion a saga a plat a catnip a pooh a rail a calamus a dairyman a bater a canal Panama

Example: Count the e's in a string

35

```
/** = number of times c occurs in s */  
public static int countEm(char c String s) {  
    if (s.length() == 0) return 0;  
    // { s has at least 1 character }  
    if (s.charAt(0) != c)  
        return countEm(c s.substring(1));  
    // { first character of s is c }  
    return 1 + countEm (c s.substring(1));  
}
```

substring s[1..]
i.e. s[1] ...
s(s.length()-1)

- `countEm('e' "it is easy to see that this has many e's") = 4`
- `countEm('e' "Mississippi") = 0`