RECURSION

Lecture 7
CS2110 – Fall 2014

Overview references to sections in text

- 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

A little about generics –used in A3

public class DLinkedList<E> { …} // E is a type parameter

```
/** Values in d1 can be ANY objects —String, JFrame, etc. */
DLinkedList d1= new DLinkedList();
…
String x= ((String) d1.getHead()).getValueOf(); // cast is needed
```

```
/** The values in d2 are only objects of class String */
DLinkedList<String> d2= new DLinkedList<String>();
…
String s= d2.getHead().getValueOf(); // no cast is needed
```

What does generic mean?

From Merriam-Webster online:

ge·ner·ic *adjective*

a : relating or applied to or descriptive of all members of a genus, species, class, or group **:** common to or characteristic of a whole group or class **:** typifying or subsuming **:** not specific or individual

generic applies to that which characterizes every individual in a category or group and may suggest further that what is designated may be thought of as a clear and certain classificatory criterion

Sum the digits in a non-negative integer

```
/** 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!

E.g. sum(7) = 7

E.g. sum(8703) = sum(870) + 3;

Two issues with recursion

```
/** 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 it execute?

2. How do we understand a given recursive method, or how do we write/develop a recursive method?

Stacks and Queues

7

stack grows

top element
2nd element
...
bottom element

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

first	second	…	last

Queue: list with (at least) two basic ops:
* Append an element
* Remove first element

First-In-First-Out (FIFO)

Americans wait in a line, the Brits wait in a queue !

Stack Frame

8

A "frame" contains information about a method call:

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

a frame

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.

Example: Sum the digits in a non-negative integer

9

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

public static void main(…) {
  int r= sum(824);
  System.out.println(r);
}
```

frame: n ___ / return info

frame: r ___ args ___ / return info

frame: ? / return info

Frame for method in the system that calls method main

Example: Sum the digits in a non-negative integer

10

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

public static void main(…) {
  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

11

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

public static void main(…) {
  int r= sum(824);
  System.out.println(r);
}
```

Method main calls sum:

n 824 / return info

main: r ___ args ___ / return info

system: ? / 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(…) {
  int r= sum(824);
  System.out.println(r);
}
```

n >= 10, sum calls sum:

n 82 / return info

n 824 / return info

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(…) {
  int r= sum(824);
  System.out.println(r);
}
```

n >= 10. sum calls sum:

	n 8 / return info
	n 82 / return info
	n 824 / return info
main	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(…) {
  int r= sum(824);
  System.out.println(r);
}
```

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

	n 8 / return info 8
	n 82 / return info
	n 824 / return info
main	r ___ args ___ / return info
system	? / return info

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(…) {
  int r= sum(824);
  System.out.println(r);
}
```

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

	8
	n 82 / return info 10
	n 824 / return info
main	r ___ args ___ / return info
	? / return info

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(…) {
  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

	10
	n 824 / return info 14
main	r ___ args ___ / return info
	? / return info

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(…) {
  int r= sum(824);
  System.out.println(r);
}
```

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

	14
main	r 14 args ___ / return info
	? / return info

Summary of method call execution

18

Memorize this!

- 1. A frame for a call contains parameters, local variables, and other information needed to properly execute a method call.
- 2. To execute a method call: push a frame for the call on the stack, assign arg values to pars, and execute method body.

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

  When method body finishes, pop frame from stack and (for a function) push the return value on the stack.
- For function call: When control given back to call, it pops the return value and uses it as the value of the function call.

Questions about local variables

19

```
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

20

Math definition of n factorial

0! = 1

n! = n * (n-1)! for n > 0

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

Math definition of b^c for c >= 0

$b^0 = 1$

$b^c = b * b^{c-1}$ 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

21

- 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.

Understanding a recursive method

22

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 >= 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

23

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.

```
/** = 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 ;
}
```

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

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

Understanding a recursive method

24

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.

```
/** = 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 ;
}
```

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

Understanding a recursive method

25

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 will see later in the 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

26

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

27

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

28

Mathematical definition:

fib(0) = 0
fib(1) = 1
two base cases!
fib(n) = fib(n − 1) + fib(n − 2), n ≥ 2

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

Example: Is a string a palindrome?

29

```
/** = "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));
}
```

Substring from s[1] to s[n-1]

isPal("racecar") returns true
isPal("pumpkin") returns false

Example: Count the e's in a string

30

```
/** = 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

Computing a^n for $n >= 0$

31

Power computation:

- $a^0 = 1$
- If n != 0, $a^n = a * a^{n-1}$
- If n != 0 and even, $a^n = (a*a)^{n/2}$

Java note: For ints x and y, x/y is the integer part of the quotient

Judicious use of the third property gives a logarithmic algorithm, as we will see

Example: $3^8 = (3*3) * (3*3) * (3*3) * (3*3) = (3*3)^4$

Computing a^n for $n >= 0$

32

Power computation:

- $a^0 = 1$
- If n != 0, $a^n = a * a^{n-1}$
- If n != 0 and even, $a^n = (a*a)^{n/2}$

```
/** = a**n. Precondition: n >= 0 */
static int power(int a, int n) {
  if (n == 0) return 1;
  if (n%2 == 0) return power(a*a, n/2);
  return a * power(a, n-1);
}
```

Conclusion

33

Recursion is a convenient and powerful way to define functions

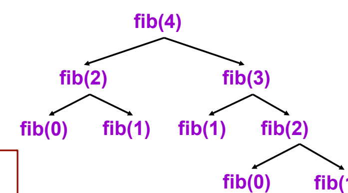
Problems that seem insurmountable can often be solved in a “divide-and-conquer” fashion:

- Reduce a big problem to smaller problems of the same kind, solve the smaller problems
- Recombine the solutions to smaller problems to form solution for big problem

Extra material: memoization

34

Execution of fib(4) is inefficient. E.g. in the tree to right, you see 3 calls of fib(1).



To speed it up, save values of fib(i) in a table as they are calculated. For each i, fib(i) called only once. The table is called a cache

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

Memoization (fancy term for “caching”)

35

Memoization: an optimization technique used to speed up execution by having function calls avoid repeating the calculation of results for previously processed inputs.

- The first time the function is called, save result
- The next time, look the result up
  - Assumes a “side effect free” function: The function just computes the result, it doesn’t change things
  - If the function depends on anything that changes, must “empty” the saved results list

Adding memoization to our solution

36

Before memoization:

```
static int fib(int n) {
   int v= n <= 1 ? n : fib(n-1) + fib(n-2);
   return v;
}
```

The list used to memoize

```
/** For  0 <= k < cached.size(),  cached[k] = fib(k) */
static ArrayList<Integer> cached=  new ArrayList<Integer>();
```

After memoization

37

```
/** For  0 <= k < cached.size(),  cached[k] = fib(k) */
static ArrayList<Integer> cached=  new ArrayList<Integer>();

static int fib(int n) {
  if (n < cached.size()) return cached.get(n);

  int v=  n <= 1 ? n :  fib(n-1) + fib(n-2);

  if (n == cached.size())
        cached.add(v);
  return v;
}
```

This works because of definition of cached

Appends v to cached, keeping cached's definition true

Memoization uses a static field

38

Recursive functions should NOT use static fields. It just doesn't work. Don't try to write recursive functions that way.
The one case that it works is suitably written memoization.

```
/** For  0 <= k < cached.size(),  cached[k] = fib(k) */
static ArrayList<Integer> cached=  new ArrayList<Integer>();

static int fib(int n) {
  if (n < cached.size()) return cached.get(n);
  int v=  n <= 1 ? n :  fib(n-1) + fib(n-2);
  if (n == cached.size()) cached.add(v);
  return v;
}
```