Unit 4, Part 3

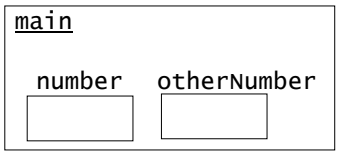
Recursion

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

Review: Method Frames

- When you make a method call, the Java runtime sets aside a block of memory known as the *frame* of that method call.

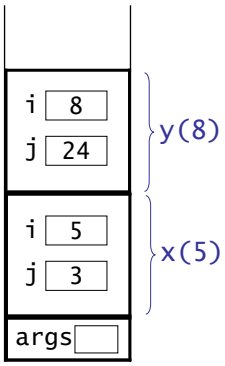


- The frame is used to store:
  - the formal parameters of the method
  - any *local variables* - variables declared within the method
- A given frame can only be accessed by statements that are part of the corresponding method call.

Frames and the Stack

- The frames we've been speaking about are stored in a region of memory known as *the stack*.
- For each method call, a new frame is added to the top of the stack.

```
public class Foo {
  public static int y(int i) {
      int j = i * 3;
      return j;
  }
  public static int x(int i) {
      int j = i - 2;
      return y(i + j);
  }
  public static void
    main(String[] args) {
      System.out.println(x(5));
  }
}
```



- When a method completes, its stack frame is removed.

Iteration

- Whenever we've encountered a problem that requires repetition, we've used *iteration* - i.e., some type of loop.
- Sample problem: printing the series of integers from n1 to n2, where n1 <= n2.
  - example: `printSeries(5, 10)` should print the following:

    `5, 6, 7, 8, 9, 10`
- Here's an iterative solution to this problem:

```
public static void printSeries(int n1, int n2) {
    for (int i = n1; i < n2; i++) {
        System.out.print(i + ", ");
    }
    System.out.println(n2);
}
```

Recursion

- An alternative approach to problems that require repetition is to solve them using *a method that calls itself*.
- Applying this approach to the print-series problem gives:

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- A method that calls itself is a *recursive* method.
- This approach to problem-solving is known as *recursion*.

Tracing a Recursive Method

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- What happens when we execute `printSeries(5, 7)`?

```
printSeries(5, 7):
    System.out.print(5 + ", ");
    printSeries(6, 7):
        System.out.print(6 + ", ");
        printSeries(7, 7):
            System.out.println(7);
            return
        return
    return
```

Recursive Problem-Solving

- When we use recursion, we solve a problem by reducing it to a simpler problem of the same kind.
- We keep doing this until we reach a problem that is simple enough to be solved directly.
- This simplest problem is known as the *base case*.

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {           // base case
        System.out.println(n2);
    } else {
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- The base case stops the recursion, because it doesn't make another call to the method.

Recursive Problem-Solving (cont.)

- If the base case hasn't been reached, we execute the *recursive case.*

```
public static void printSeries(int n1, int n2) {
    if (n1 == n2) {           // base case
        System.out.println(n2);
    } else {                  // recursive case
        System.out.print(n1 + ", ");
        printSeries(n1 + 1, n2);
    }
}
```

- The recursive case:
  - reduces the overall problem to one or more simpler problems of the same kind
  - makes recursive calls to solve the simpler problems

Structure of a Recursive Method

```
recursiveMethod(parameters) {
    if (stopping condition) {
        // handle the base case
    } else {
        // recursive case:
        // possibly do something here

        recursiveMethod(modified parameters);

        // possibly do something here
    }
}
```

- There can be multiple base cases and recursive cases.
- When we make the recursive call, we typically use parameters that bring us closer to a base case.

Tracing a Recursive Method: Second Example

```
public static void mystery(int i) {
    if (i <= 0) {     // base case
        return;
    }
    // recursive case
    System.out.println(i);
    mystery(i - 1);
    System.out.println(i);
}
```

- What happens when we execute `mystery(2)`?

Printing a File to the Console

- Here's a method that prints a file using iteration:

```
public static void print(Scanner input) {
    while (input.hasNextLine()) {
        System.out.println(input.nextLine());
    }
}
```

- Here's a method that uses recursion to do the same thing:

```
public static void printRecursive(Scanner input) {
    // base case
    if (!input.hasNextLine()) {
        return;
    }

    // recursive case
    System.out.println(input.nextLine());
    printRecursive(input);  // print the rest
}
```

Printing a File in Reverse Order

- What if we want to print the lines of a file in reverse order?
- It's not easy to do this using iteration. Why not?
- It's easy to do it using recursion!
- How could we modify our previous method to make it print the lines in reverse order?

```
public static void printRecursive(Scanner input) {
    if (!input.hasNextLine()) {   // base case
        return;
    }

    String line = input.nextLine();
    System.out.println(line);
    printRecursive(input);  // print the rest
}
```

Printing a File in Reverse Order (cont.)

- An iterative approach to reversing the file would need to:
  - read all of the lines in the file and store them in a temporary data structure (e.g., an array)
  - retrieve the lines from the data structure and print them in reverse order
- The recursive method doesn't need a separate data structure.
  - the lines are stored in the stack frames for the recursive method calls!

A Recursive Method That Returns a Value

- Simple example: summing the integers from 1 to n

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

- Example of this approach to computing the sum:

```
sum(6)  =  6 + sum(5)
        =  6 + 5 + sum(4)
        =  …
```

Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

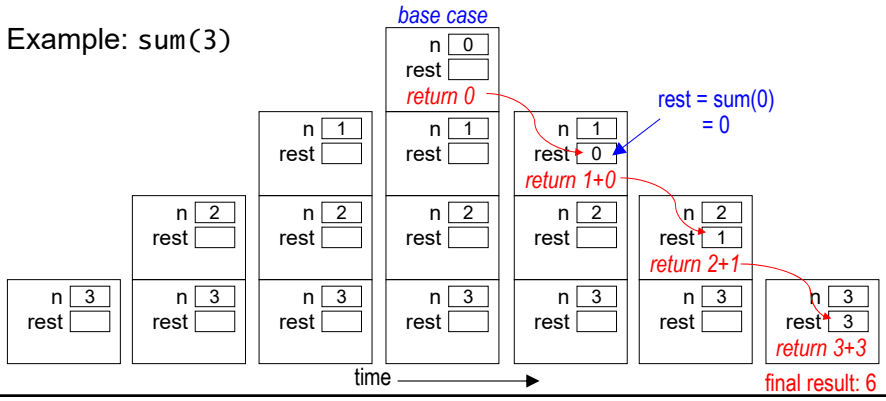
- What happens when we execute `int x = sum(3);` from inside the `main()` method?

Tracing a Recursive Method on the Stack

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

The final result gets built up ***on the way back*** from the base case!

Example: `sum(3)`



Another Option for Tracing a Recursive Method

```
public static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

Infinite Recursion

- We have to ensure that a recursive method will eventually reach a base case, regardless of the initial input.
- Otherwise, we can get *infinite recursion*.
  - produces *stack overflow* - there's no room for more frames on the stack!
- Example: here's a version of our `sum()` method that uses a different test for the base case:

```
public static int sum(int n) {
    if (n == 0) {
        return 0;
    }
    int rest = sum(n - 1);
    return n + rest;
}
```

  - what values of n would cause infinite recursion?

Designing a Recursive Method

1. Start by programming the base case(s).
   - *What instance(s) of this problem can I solve directly (without looking at anything smaller)?*
2. Find the recursive substructure.
   - *How could I use the solution to **any smaller version** of the problem to solve the overall problem?*
3. Solve the smaller problem using a recursive call!
   - store its result in a variable
4. Do your one step.
   - build your solution from the result of the recursive call
   - use *concrete cases* to figure out what you need to do

Processing a String Recursively

- A string is a recursive data structure. It is either:
  - empty ("")
  - a single character, followed by a string
- Thus, we can easily use recursion to process a string.
  - process one or two of the characters ourselves
  - make a recursive call to process the rest of the string
- Example: print a string vertically, one character per line:

```
public static void printVertical(String str) {
    if (str == null || str.equals("")) {
        return;
    }

    System.out.println(str.charAt(0)); // first char
    printVertical(str.substring(1));   // rest of string
}
```

Short-Circuited Evaluation

- The second operand of both the && and || operators will not be evaluated if the result can be determined on the basis of the first operand alone.

- `expr1 || expr2`
  if `expr1` evaluates to `true`, `expr2` is not evaluated, because we already know that `expr1 || expr2` is `true`
  - example from the last slide:

```
if (str == null || str.equals("")) {
    return;
}
// if str is null, we won't check for empty string.
// This prevents a null pointer exception!
```

- `expr1 && expr2`
  if `expr1` evaluates to _____, `expr2` is not evaluated, because we already know that `expr1 && expr2` is _____.

Counting Occurrences of a Character in a String

- `numOccur(c, s)` should return the number of times that the character `c` appears in the string `s`
  - `numOccur('n', "banana")` should return `2`
  - `numOccur('a', "banana")` should return `3`

- Take the approach outlined earlier:
  - base case: empty string (or null)
  - delegate `s.substring(1)` to the recursive call
  - we're responsible for handling `s.charAt(0)`

Applying the String-Processing Template

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {  // base case
        return __________;
    } else {                  // recursive case
        int rest = ____________________;
        // do our one step!
    }
}
```

Determining Our One Step

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        // do our one step!
```

- In our one step, we take care of `s.charAt(0)`.
  - we build the solution to the larger problem on the solution to the smaller problem (in this case, `rest`)
  - does what we do depend on the value of `s.charAt(0)`?

- ***Use concrete cases to figure out the logic!***

Consider this concrete case…

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        // do our one step!
        ...

numOccur('r', "recurse")
```

---

numOccur('r', "recurse")
c = 'r', s = "recurse"

Consider Concrete Cases

```
numOccur('r', "recurse")     # first char is a match
```

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
  What is our one step?

```
numOccur('a', "banana")     # first char is not a match
```

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem?
  What is our one step?

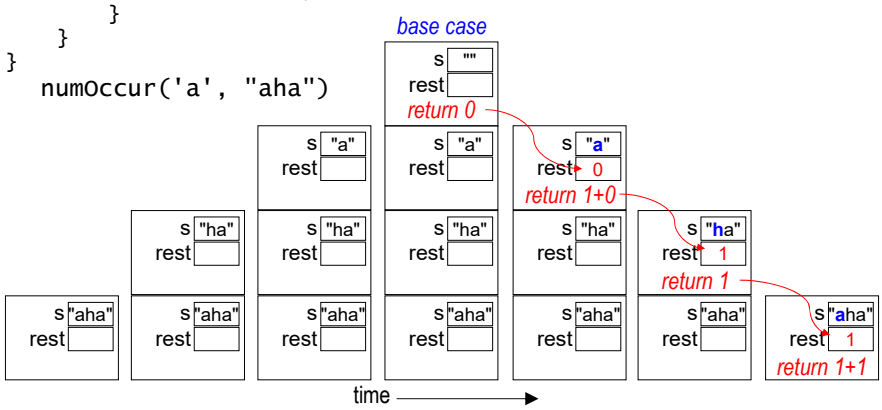
Now complete the method!

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        if (s.charAt(0) == c) {

            return ____________________;
        } else {

            return ____________________;
        }
    }
}
```

Tracing a Recursive Method on the Stack

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    } else {
        int rest = numOccur(c, s.substring(1));
        if (s.charAt(0) == c) {
            return 1 + rest;
        } else {
            return rest;
        }
    }
}
   numOccur('a', "aha")
```

The final result gets built up ***on the way back*** from the base case!



Common Mistake

- This version of the method does *not* work:

```
public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    }

    int count = 0;
    if (s.charAt(0) == c) {
        count++;
    }

    numOccur(c, s.substring(1));
    return count;
}
```

Another Faulty Approach

- Some people make `count` "global" to fix the prior version:

```
public static int count = 0;

public static int numOccur(char c, String s) {
    if (s == null || s.equals("")) {
        return 0;
    }

    if (s.charAt(0) == c) {
        count++;
    }

    numOccur(c, s.substring(1));
    return count;
}
```

- Not recommended, and not allowed on the problem sets!
- Problems with this approach?

Recursion vs. Iteration

- Some problems are much easier to solve using recursion.
- Other problems are just as easy to solve using iteration.
- Recursion is a bit more costly because of the overhead involved in invoking a method.
  - also: in some cases, there may not be room on the stack
- Rule of thumb:
  - if it's easier to formulate a solution recursively, use recursion, unless the cost of doing so is too high
  - otherwise, use iteration

Extra Practice: A Recursive Palindrome Checker

- A *palindrome* is a string that reads the same forward and backward.
  - examples: "radar", "mom", "abcddcba"
- `isPal(s)` should return `true` if `s` is a palindrome, and `false` otherwise.
- We need more than one base case. What are they?
- How should we reduce the problem in the recursive call?

Consider Concrete Cases!

`isPal("radar")`

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem...?
  What is our one step?

`isPal("modem")`

- what is its solution?
- what is the next smaller subproblem?
- what is the solution to that subproblem?
- how can we use the solution to the subproblem...?
  What is our one step?

A Recursive Palindrome Checker (cont.)

- Method definition (assuming no nulls):

```
public static boolean isPal(String s) {
    int len = s.length();
    if (len <= 1) {

        return __________;

    } else if (_____________________________________) {

        return __________;
    } else {

        boolean isPalRest = _____________________________;

        // do our one step!


    }
}
```