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:

```java
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

```java
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
```
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.

```java
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.

```java
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

```java
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

```java
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:
  
  ```java
  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:
  
  ```java
  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?
  
  ```java
  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

```java
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

```java
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;
}
```

Example: `sum(3)`

```
base case
n 0
rest 0
return 0
```

```
n 1
rest 1
return 1+0
```

```
n 2
rest 2
return 2+1
```

```
n 3
rest 3
return 3+3
```

**Final result:** 6
Another Option for Tracing a Recursive Method

```java
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:
  ```java
  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. Do one step!

4. Delegate the rest to recursion!

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
  - make a recursive call to process the rest of the string

- Example: print a string vertically, one character per line:
  ```java
  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:
    ```java
    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

- Let's design a recursive method called numOccur().
- numOccur(c, s) should return the number of times that the character c appears in the string s

- Method definition:
Tracing a Recursive Method on the Stack

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

```
umOccur('a', "aha")
```

Common Mistake

- This version of the method does not work:

```java
public static int numOccur(char c, String s) {
    if (s == null || s.equals("") || s.substring(1).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:

```java
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?

Testing for a Prefix

- Let's design a recursive method called isPrefix().

- isPrefix(str1, str2) should return:
  - true if str1 is a prefix of str2
  - false if str1 is not a prefix of str2

- Examples:
  - isPrefix("recur", "recurse") should return true
  - isPrefix("record", "recurse") should return false

- Special case: we will consider the empty string ("") to be the prefix of any string.
- Thinking recursively:
Testing for a Prefix (cont.)

- Put the method definition here:

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 Palindrome Checker

- A *palindrome* is a word or phrase that reads the same forward and backward (ignoring spaces, punctuation, and case).
  - examples: "radar", "mom", "live not on evil"

- `isPalindrome(str)` should return `true` if `str` is a palindrome, and `false` otherwise.

- Thinking recursively:

---

A Palindrome Checker (cont.)

- Put the method definition here: