Static methods (contd.)

- Static methods represent procedural abstractions
- Why don't we use only static methods and no non-static methods? We could, but we want to use OOP, because we want to model the problem domain realistically. Objects and classes do that.
- Static methods: functional/procedural view of the problem and its solution.
- Non-static methods: object-oriented view of the problem and its solution.

Methods a *reusable* abstractions

- A method can be reused in different contexts
- Calling a method is "the same" as substituting its body in place of its call (replacing the parameters by the actual arguments,) but
- If we define a method, we can simply call it from more than one context without having to do copy and paste.

Example

```
public class Newtons {
  static double sqrt(double x, double tolerance)
  {
    double guess = 1.0;
    while (!close_enough(guess*guess,x,tolerance))
    {
      guess = improve(guess, x);
    }
    return guess;
  }
  static boolean close_enough(double a, double b,
                              double tolerance)
  {
    return (Math.abs(a-b) < tolerance);
  }
  static double improve(double g, double x)
  {
    return (g + x/g)/2;
  }
}
```

Example (contd.)

```
public class Newtons {
  static double sqrt(double x, double tolerance)
  {
    double guess = 1.0;
    while (!(Math.abs(guess*guess-x) < tolerance))
    {
      guess = (guess + x/guess)/2;
    }
    return guess;
  }
}
```

Searching for solutions

- Generic algorithm to search for solutions:

1. Start with some guess
2. While the guess is not good enough, repeat:
   (a) Improve the guess
3. The result is the final guess

Example: reusing methods

```
public class B {
  void q(int v)
  {
    int k = (v+1)*2+1;
    // ... do something with k
  }
}
public class C {
  void r(int w)
  {
    int u = (w-3)*2+1;
    // ... do something with u
  }
}
```

Example (contd.)

```
public class A {
  static int p(int n)
  {
    return n*2+1;
  }
}
public class B {
  void q(int w)
  {
    int k = A.p(w+1);
    // ... do something with k
  }
}
public class C {
  void r(int v)
  {
    int u = A.p(v-3);
    // ... do something with u
  }
}
```

Method overloading

- There can be several (static or not) methods with the same name...
- ...but the type or number of parameters must be different

Example

```
public class A {
  void f(int x)
  {
    System.out.println(“one: ”+x)
  }
  void f(boolean x)
  {
    System.out.println(“two: ”+x)
  }
}
public class B {
  void g()
  {
    A u = new A();
    u.f(5);
    u.f(false);
  }
}
```

Same for static methods

```
public class A {
  static void f(int x)
  {
    System.out.println("one: "+x)
  }
  static void f(boolean x)
  {
    System.out.println("two: "+x)
  }
}
public class B {
  void g()
  {
    A.f(5);
    A.f(false);
  }
}
```

Recursion

- A recursive method is a method that calls itself (directly or indirectly.)
- A recursive definition is a definition of something in terms of itself
- Some recursive definitions don't make sense, (e.g. from Webster's: growl: to utter a growl), but others do
- For example:
  - A *list of numbers* is either:
    * A single number, or
    * A number followed by a list of numbers.
  - For example:
    * 5 is a list of numbers
    * 7, 5 is a list of numbers (because 5 is a list)
    * 6, 7, 5 is a list of numbers (because 7, 5 is a list)
    * 8, 6, 7, 5 is a list of numbers (because 6, 7, 5 is a list)

Recursive functions

- Factorial: the factorial of a natural number n, written $n!$ is the multiplication of the first n positive integers, i.e.

$$n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot (n-2) \cdot (n-1) \cdot n \qquad (1)$$

But note that

$$1 \cdot 2 \cdot 3 \cdot ... \cdot (n-2) \cdot (n-1) = (n-1)! \qquad (2)$$

So by (1) and (2) we get

$$n! = (n-1)! \cdot n \qquad (3)$$

But we have to assume a "base case", by defining

$$0! = 1 \qquad (4)$$

Recursive functions (contd.)

Hence, (3) and (4) together gives us an alternative, and recursive definition of (1):

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \cdot n & \text{otherwise} \end{cases}$$

This can be implemented as a static recursive method:

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

Execution of recursive methods

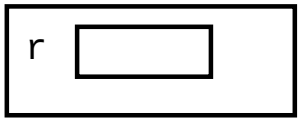
Consider the following client for this factorial function:

```
int r;
r = factorial(4);
```

Its execution proceeds as follows:

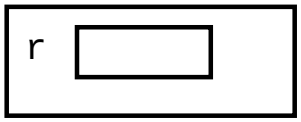
This is executed in some frame:

Some frame

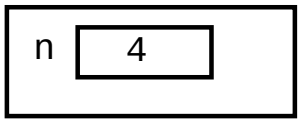


When we call `factorial(4);` a new frame for the method is created:

Some frame



factorial frame

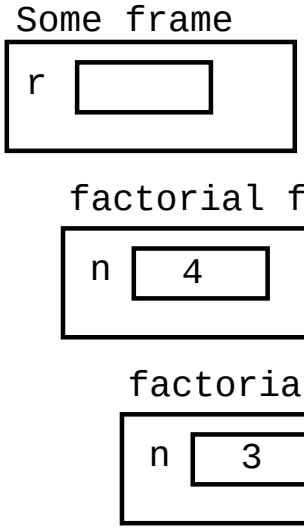


We execute the body of factorial; n is not 0 so we execute

```
return factorial(n-1)*n;
```

which in this frame is the same as

```
return factorial(4-1)*4;
```



pending computation:

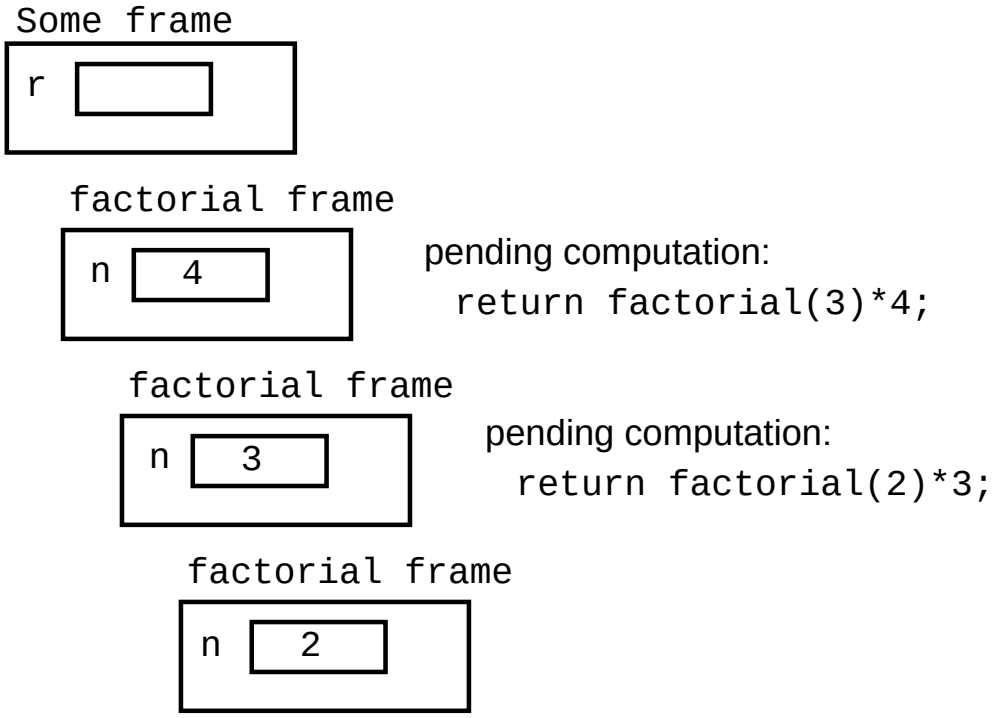
```
return factorial(3)*4;
```

Again, we execute the body of factorial;
again, n is not 0 so we execute

```
return factorial(n-1)*n;
```

which in this frame is the same as

```
return factorial(3-1)*3;
```

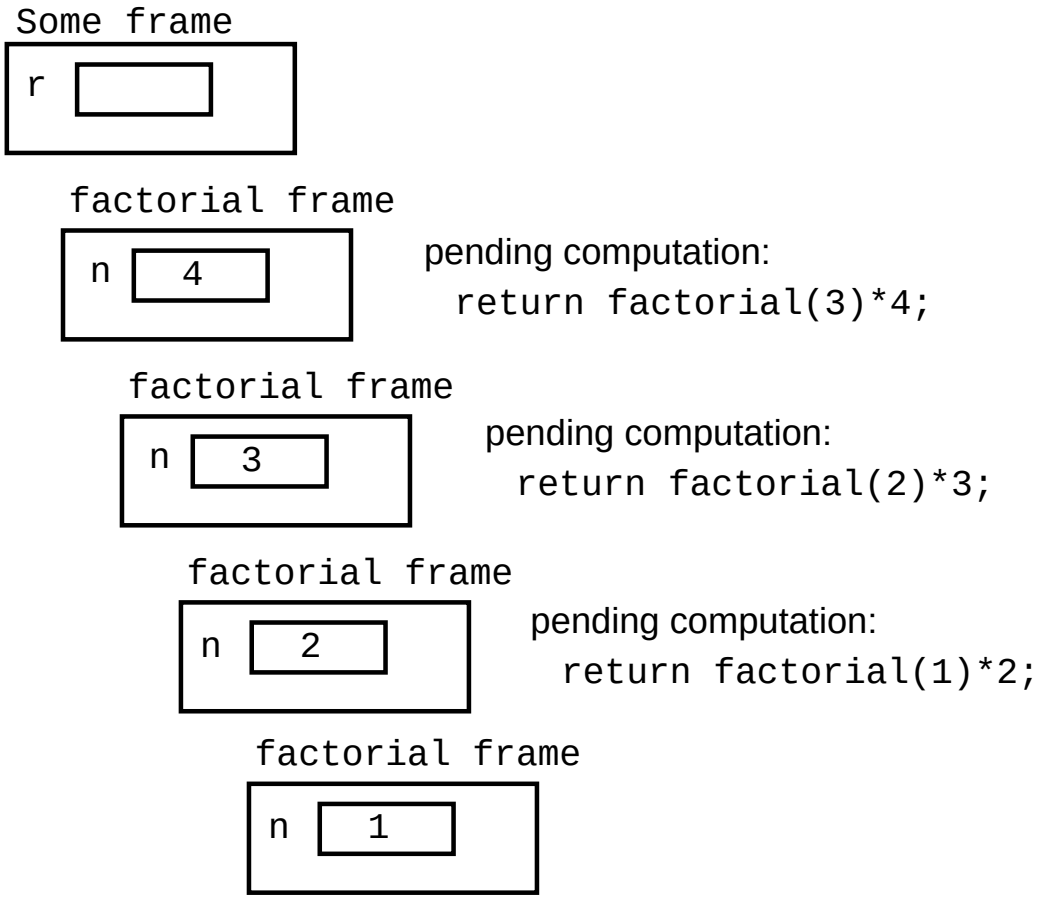


Again, we execute the body of factorial;
again, n is not 0 so we execute

```
return factorial(n-1)*n;
```

which in this frame is the same as

```
return factorial(2-1)*2;
```

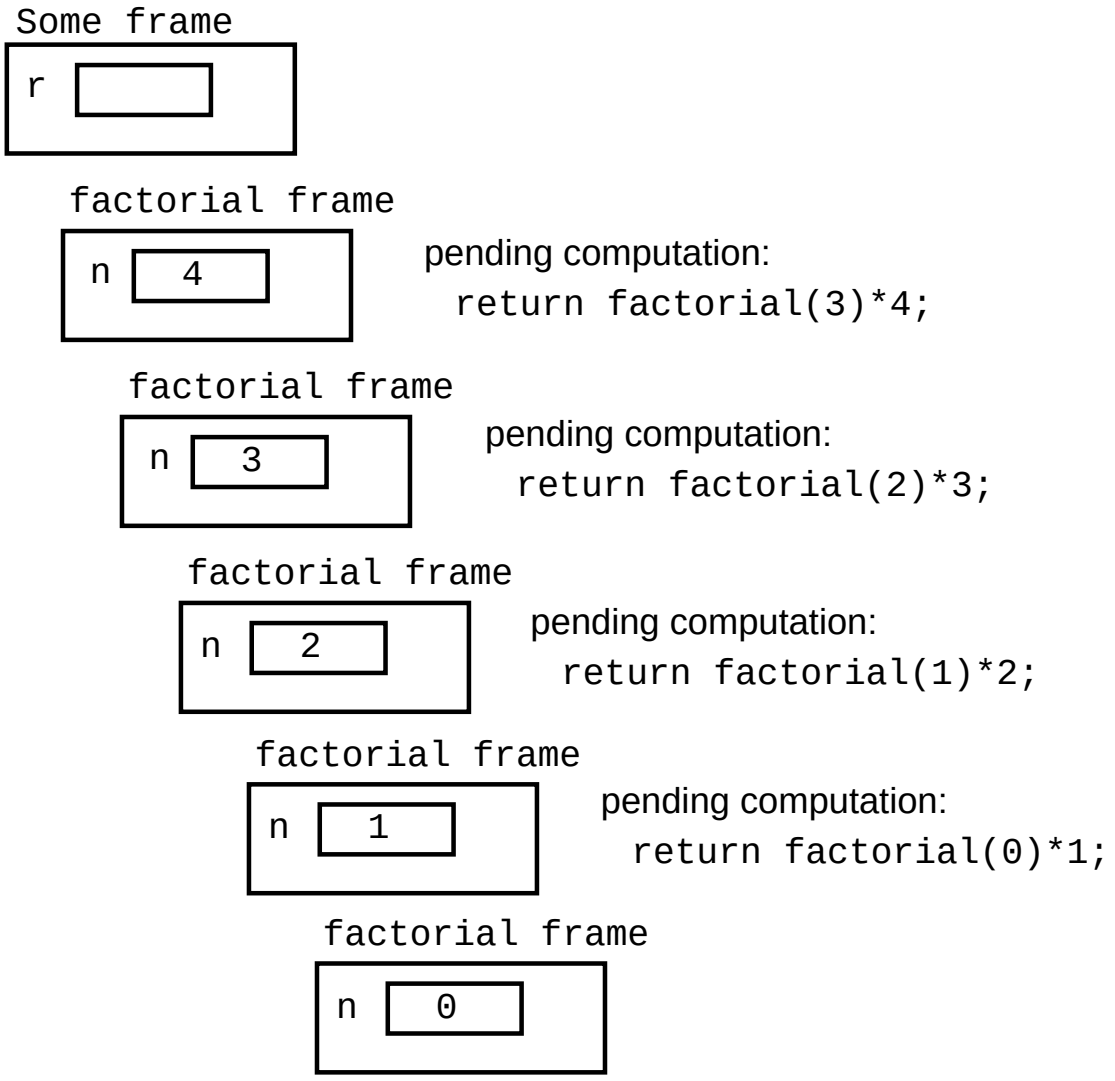


Again, we execute the body of factorial;
again, n is not 0 so we execute

```
return factorial(n-1)*n;
```

which in this frame is the same as

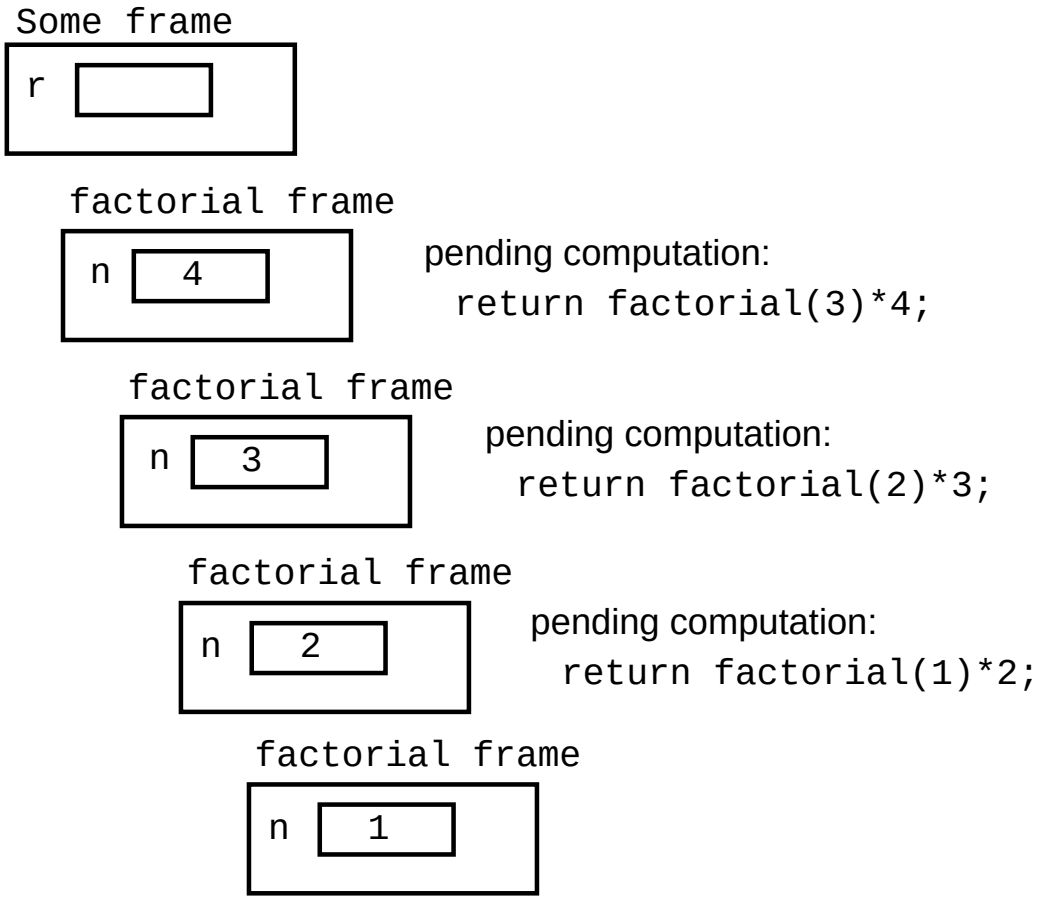
```
return factorial(1-1)*1;
```



Now, we have reached the base case, and n is 0, so we execute:

```
return 1;
```

We get rid of the frame, and pass the returned value to the caller



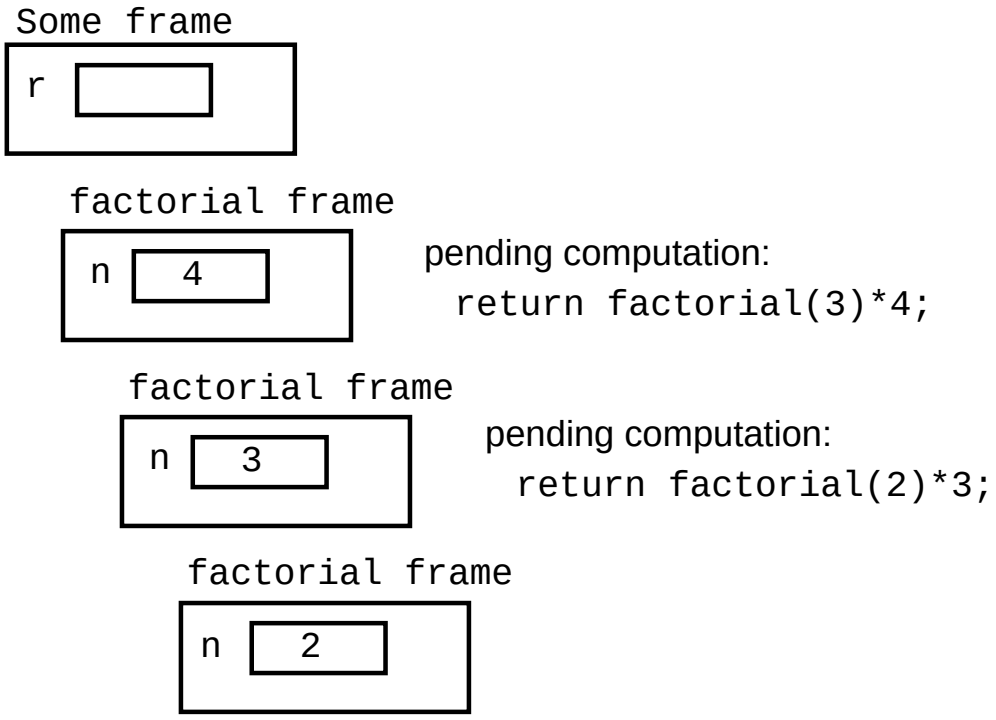
The pending computation here was:

```
return factorial(0)*1;
```

and the method called `factorial(0)`
returned 1, so this pending computation is now:

```
return 1*1;
```

We get rid of the frame, and pass the returned value to the caller



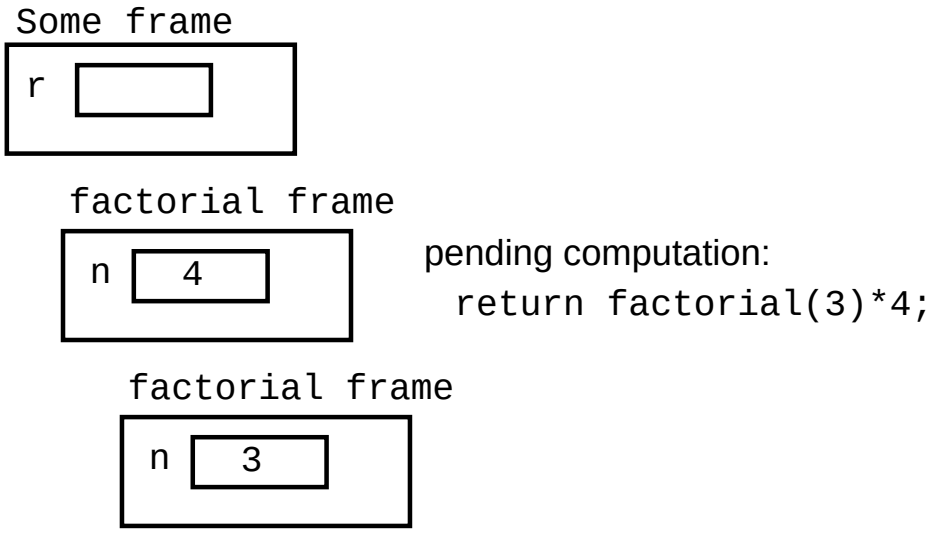
The pending computation here was:

```
return factorial(1)*2;
```

and the method called `factorial(1)`
returned 1, so this pending computation is now:

```
return 1*2;
```

We get rid of the frame, and pass the returned value to the caller



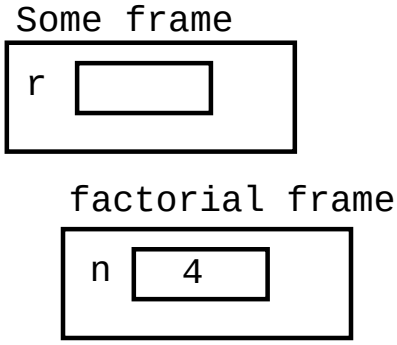
The pending computation here was:

```
return factorial(2)*3;
```

and the method called `factorial(2)`
returned 2, so this pending computation is now:

```
return 2*3;
```

We get rid of the frame, and pass the returned value to the caller



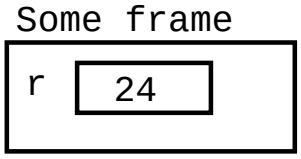
The pending computation here was:

```
return factorial(3)*4;
```

and the method called `factorial(3)`
returned 6, so this pending computation is now:

```
return 6*4;
```

We get rid of the frame, and pass the returned value to the caller



The pending computation here was:

```
r = factorial(4);
```

which returned 24, so this pending computation is now:

```
r = 24;
```

Recursion on other types

- Problem: given a string s, return the reverse of the string

- Analysis:
  - Notation:
    * $\mathsf{rev}(s)$ is the reverse of s
    * s_i is the i-th character of s
    * $\mathsf{len}(s)$ is the length of s
    * $\mathsf{rest}(s)$ is the string s without its first character s_0 (i.e. $\mathsf{rest}(s) = s_1 s_2 ... s_n$ where $n = \mathsf{len}(s) - 1$)
  - Formal definition of reverse:

$$\mathsf{rev}(s) = \begin{cases} \text{“”} & \text{if } s = \text{“”} \\ \mathsf{rev}(\mathsf{rest}(s)) + s_0 & \text{otherwise} \end{cases}$$

Reverse (contd.)

- For example:

$$
\begin{aligned}
\mathsf{rev}(\text{“}abcd\text{”}) &= \mathsf{rev}(\text{“}bcd\text{”}) +' a' \\
&= (\mathsf{rev}(\text{“}cd\text{”}) +' b') +' a' \\
&= ((\mathsf{rev}(\text{“}d\text{”}) +' c') +' b') +' a' \\
&= (((\mathsf{rev}(\text{“”}) +' d') +' c') +' b') +' a' \\
&= (((\text{“”} +' d') +' c') +' b') +' a' \\
&= ((\text{“}d\text{”} +' c') +' b') +' a' \\
&= (\text{“}dc\text{”} +' b') +' a' \\
&= \text{“}dcb\text{”} +' a' \\
&= \text{“}dcba\text{”}
\end{aligned}
$$

Reverse (contd.)

```
public class MoreStringOperations {
  static String reverse(String s)
  {
    if (s.equals(“”)) {
      return “”;
    }
    return reverse(rest(s))+s.charAt(0);
  }
  static String rest(String s)
  {
    String result =“”;
    int i = 1;
    while (i < s.length()) {
      result = result + s.charAt(i);
      i++;
    }
    return result;
  }
}
```

Double recursion

- Problem: Compute the n-th Fibonacci number
- Analysis: The Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, ...is defined by:

$$fib(n) = \begin{cases} 1 & \text{if } n \leqslant 2 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

- Implementation:

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

Iteration vs recursion

- Iterative solution to the Fibonacci problem:

```
static int fib(int n)
{
    int a, b, c, i;
    a = 1;
    b = 1;
    c = 1;
    i = 3;
    while (i <= n) {
        c = a + b;
        a = b;
        b = c;
        i++;
    }
    return c;
}
```

Execution trees