Reminder

- Deadline extended until Thursday, February 26th at 23:55

Kinds of methods

- Normal (non-static) methods...
  - define how objects in a class react to messages
  - therefore, they are applied to objects

    objectref.method(arg1,arg2,...,argn)

- Static methods...
  - define functions or procedures which do not affect objects in the their class
  - therefore, they are *not* applied to objects

    classname.method(arg1,arg2,...,argn)

Declaring methods

- Declaring normal methods

```
type method_name(type1 arg1, type2 arg2,
                 ..., typen argn)
{
    statements;
}
```

- Declaring static methods

```
static type method_name(type1 arg1, type2 arg2,
                        ..., typen argn)
{
    statements;
}
```

Example (contd.)

```
public class B
{
    public static void main(String[] args)
    {
        A.q();          // Prints Good bye
        A x = new A();  // Creates an A object
        x.p();          // Prints Hello
        A.p();          // Compile-time Error
        x.q();          // Prints Good bye
    }
}
```

Static variables

```
public class BankAccount
{
  float balance;

  BankAccount()
  {
    balance = 0.0f;
  }
  void deposit(float amount)
  {
    balance = balance + amount;
  }
  void withdraw(float amount)
  {
    if (amount < balance)
      balance = balance - amount;
  }
}
```

Static variables (contd.)

```
public class Bank {
  public static void main(String[] args)
  {
    BankAccount pete, amy;
    pete = new BankAccount();
    amy = new BankAccount();

    pete.deposit(700.0f);
    amy.deposit(800.0f);

    System.out.println(pete.balance);
    System.out.println(amy.balance);
  }
}
```

Static variables (contd.)

```
public class BankAccount
{
  static float balance;

  BankAccount()
  {
    balance = 0.0f;
  }
  void deposit(float amount)
  {
    balance = balance + amount;
  }
  void withdraw(float amount)
  {
    if (amount < balance)
      balance = balance - amount;
  }
}
```

Static variables (contd.)

```
public class Bank {
  public static void main(String[] args)
  {
    BankAccount pete, amy;
    pete = new BankAccount();
    amy = new BankAccount();

    pete.deposit(700.0f);
    amy.deposit(800.0f);

    System.out.println(pete.balance);
    System.out.println(amy.balance);
  }
}
```

Static methods access

- Since the frame of a static method does not have a reference to an object, static methods cannot access attributes of an object

```
public class A
{
  int n;
  void p()
  {
    System.out.println(n); //OK
  }
  static void q()
  {
    System.out.println(n); //WRONG
  }
}
```

Static methods access

- A static method can be called from a non-static context, but...
- A non-static method cannot be called from a static context, because in order to call a non-static method, you need to provide a reference to an object.

Static methods access

```
public class A
{
  void p()
  {
    System.out.println(‘‘bye’’);
  }
  static void q()
  {
    System.out.println(‘‘hello’’);
    p();
  }
}
```

Static methods access

```
public class A
{
  void p()
  {
    System.out.println(“bye”);
  }
  static void q()
  {
    System.out.println(“hello”);
    p();  // ERROR
  }
}
```

Static methods access

```
public class A
{
  void p()
  {
    System.out.println(‘‘bye’’);
  }
  static void q()
  {
    System.out.println(‘‘hello’’);
    this.p();  // ERROR
  }
}
```

Static methods access

```
public class A
{
  static void p()
  {
    System.out.println(‘‘bye’’);
  }
  static void q()
  {
    System.out.println(‘‘hello’’);
    p();
  }
}
```

Static methods access

```
public class A
{
  int n;
  void p()
  {
    System.out.println(n);
  }
  static void q()
  {
    System.out.println("hello");
    p();
  }
}
```

Static methods access

```
public class A
{
  int n;
  void p()
  {
    System.out.println(n);
  }
  static void q()
  {
    System.out.println(‘‘hello’’);
    this.p();
  }
}
```

Static methods access

```
public class A
{
  int n;
  static void p()
  {
    System.out.println(n);
  }
  static void q()
  {
    System.out.println("hello");
    p();
  }
}
```

Static methods access

```
public class A
{
  int n;
  static void p()
  {
    System.out.println(this.n); // ERROR
  }
  static void q()
  {
    System.out.println(“hello”);
    p();
  }
}
```

Static methods access

```
public class A
{
  static int n;
  static void p()
  {
    System.out.println(n);
  }
  static void q()
  {
    System.out.println(“hello”);
    p();
  }
}
```

Static methods access

```
public class A
{
  int n;
  void p()
  {
    System.out.println(this.n);
  }
  static void q()
  {
    System.out.println(“hello”);
    A some_object = new A();
    some_object.p();
  }
}
```

Methods: *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.

Methods: reusable abstractions

Determining whether n is a prime number or not:

```
boolean result;
int i;

result = true;
i = 2;
while (i < n && result) {
  if (n % i == 0) {
    result = false;
  }
  i++;
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static void print_primes(int m)
  {
    boolean result;
    int n;

    n = 1;
    while (n <= m) {

      // Find out if n is prime...
      if (result)
        System.out.println(n);
      n++;
    }
  }
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static void print_primes(int m)
  {
    boolean result;
    int i, n;

    n = 1;
    while (n <= m) {
      result = true;
      i = 2;
      while (i < n && result) {
        if (n % i == 0) {
          result = false;
        }
        i++;
      }
      if (result)
        System.out.println(n);
      n++;
    }
  }
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static boolean is_prime(int n)
  {
    boolean result;
    int i;

    result = true;
    i = 2;
    while (i < n && result) {
      if (n % i == 0) {
        result = false;
      }
      i++;
    }
    return result;
  }
  //... rest of the class
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static void print_primes(int m)
  {
    boolean result;
    int i, n;

    n = 1;
    while (n <= m) {
      result = true;
      i = 2;
      while (i < n && result) {
        if (n % i == 0) {
          result = false;
        }
        i++;
      }
      if (result)
        System.out.println(n);
      n++;
    }
  }
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static boolean is_prime(int n) { ... }

  static void print_primes(int m)
  {
    boolean result;
    int n;

    n = 1;
    while (n <= m) {
      result = is_prime(n);
      if (result)
        System.out.println(n);
      n++;
    }
  }
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static boolean is_prime(int n) { ... }

  static void print_primes(int m)
  {
    int n;

    n = 1;
    while (n <= m) {
      if (is_prime(n))
        System.out.println(n);
      n++;
    }
  }
}
```

Methods: reusable abstractions

Problem: given three numbers, determine whether all of them are prime or their sum is prime

Methods: reusable abstractions

```
public class MyMathProcedures {
  static boolean is_prime(int n) { ... }

  static void threenumbers(int a, int b, int c)
  {
    if (is_prime(a) && is_prime(b) && is_prime(c)
        || is_prime(a+b+c)) {
      return true;
    }
    return false;
  }
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static boolean is_prime(int n) { ... }

  static void threenumbers(int a, int b, int c)
  {
    return (is_prime(a) && is_prime(b) && is_prime(
         || is_prime(a+b+c));
  }
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static boolean is_prime(int n) { ... }

  static void threenumbers(int a, int b, int c)
  {
    boolean result1, result2, result3, result4;
    int i;

    result1 = true;
    i = 2;
    while (i < a && result1) {
      if (a % i == 0) {
        result1 = false;
      }
      i++;
    }
    result2 = true;
    i = 2;
    while (i < b && result2) {
```

```
      if (b % i == 0) {
        result2 = false;
      }
      i++;
    }
    result3 = true;
    i = 2;
    while (i < c && result3) {
      if (c % i == 0) {
        result3 = false;
      }
      i++;
    }
    result4 = true;
    i = 2;
    while (i < a+b+c && result4) {
      if ((a+b+c) % i == 0) {
        result4 = false;
      }
      i++;
    }
    return result1 && result2 && result3 || result4
  }
```

```
}
```

Methods: reusable abstractions

```
public class MyMathProcedures {
  static boolean is_prime(int n)
  {
    boolean result;
    int i;

    result = true;
    i = 2;
    while (i < Math.sqrt(n) && result) {
      if (n % i == 0) {
        result = false;
      }
      i++;
    }
    return result;
  }
  //... rest of the class
}
```

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.

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

But note that

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

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

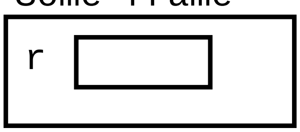
Consider the following client for this factorial function:

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

Its execution proceeds as follows:

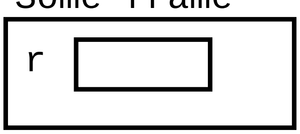
This is executed in some frame:

`Some frame`

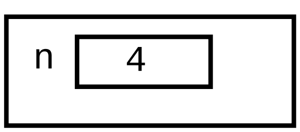


When we call `factorial(4);` a new frame for the method is created:

`Some frame`



`factorial frame`

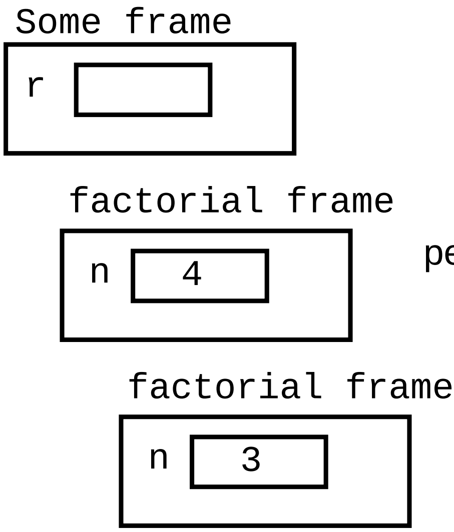


We execute the body of factorial; n is not 0 so we execute

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

which in this frame is the same as

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



pending computation:

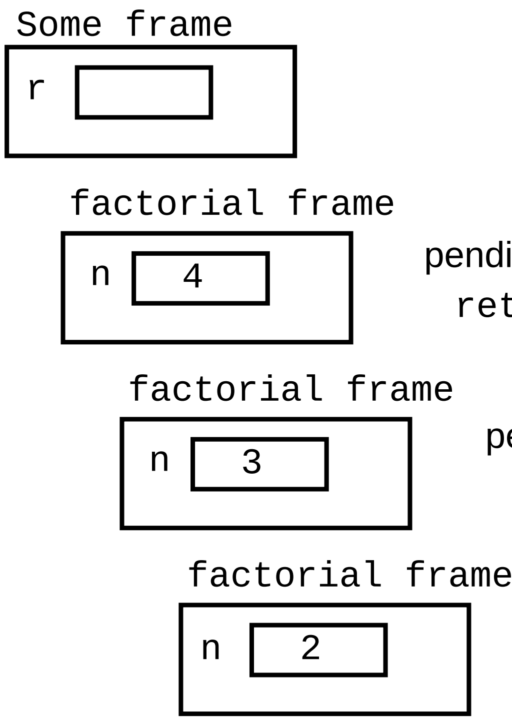
```
return factorial(3)*4;
```

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;
```



pending computation:

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

pending computation:

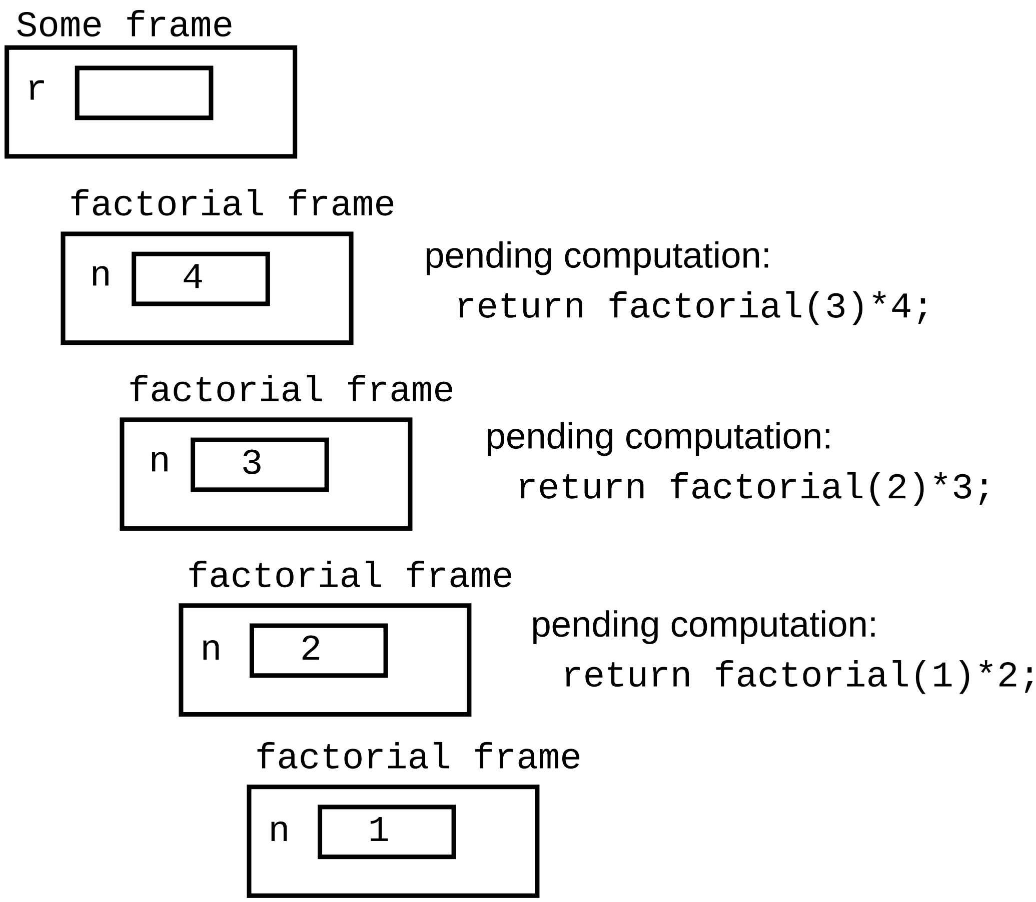
```
return factorial(2)*3;
```

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;
```

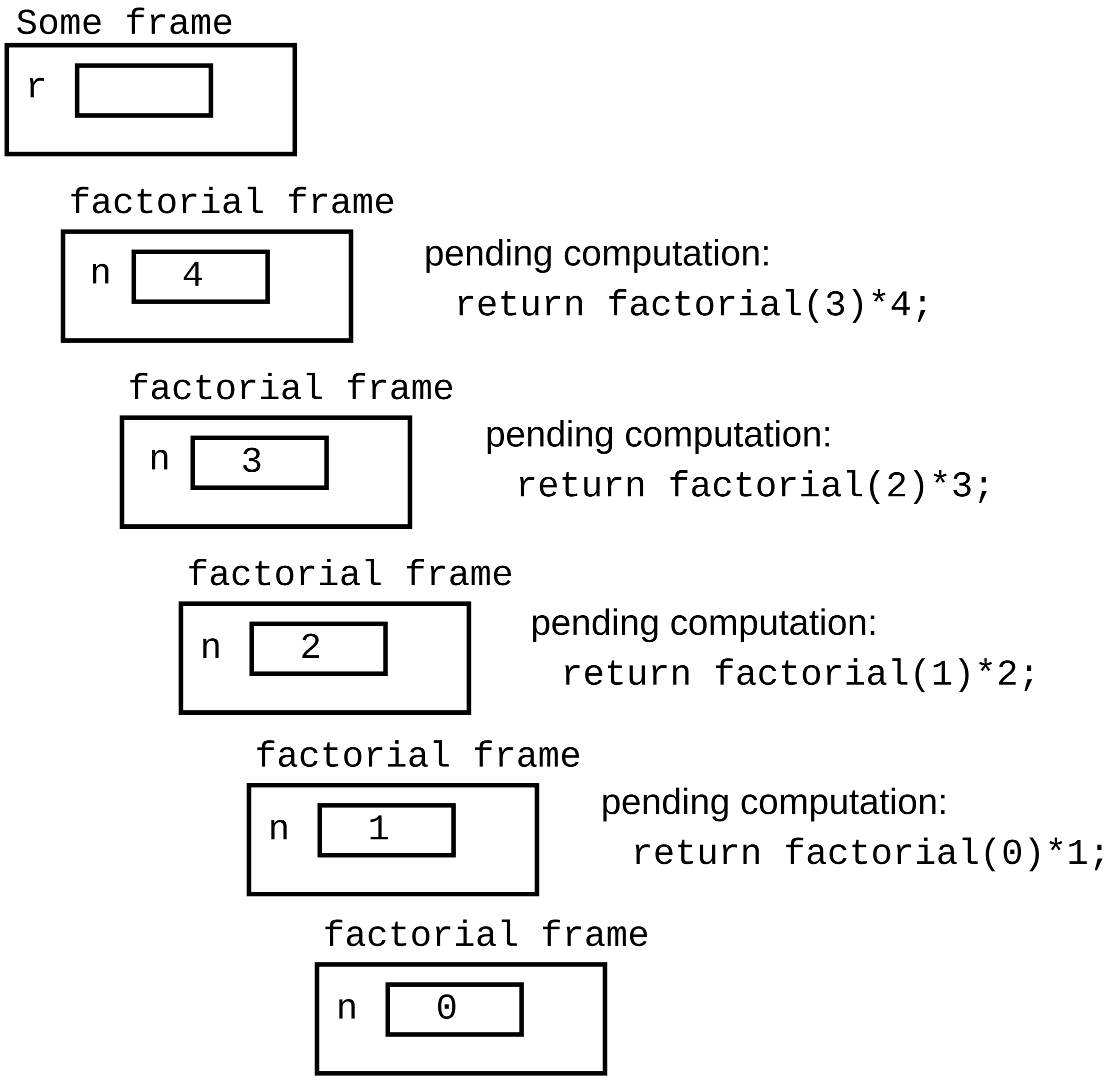


Again, we execute the body of factorial;
again, n is not 0 so we execute

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

which in this frame is the same as

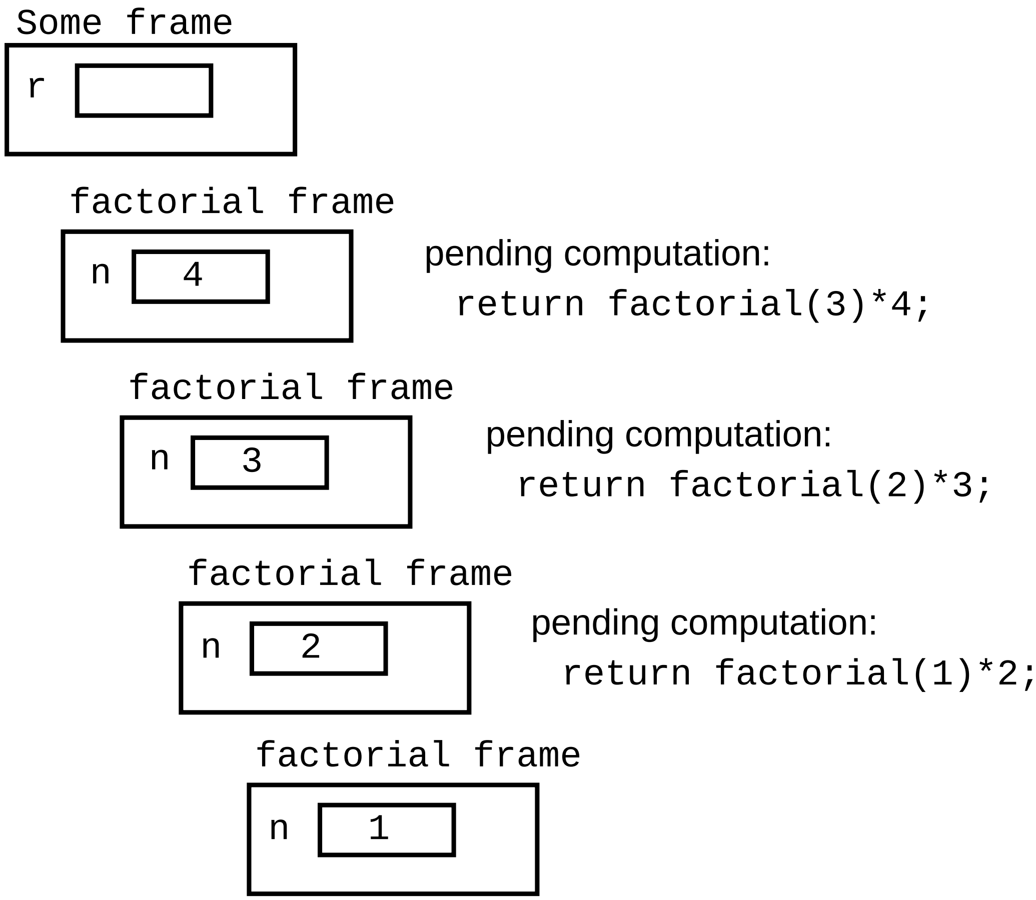
```
return factorial(1-1)*1;
```



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



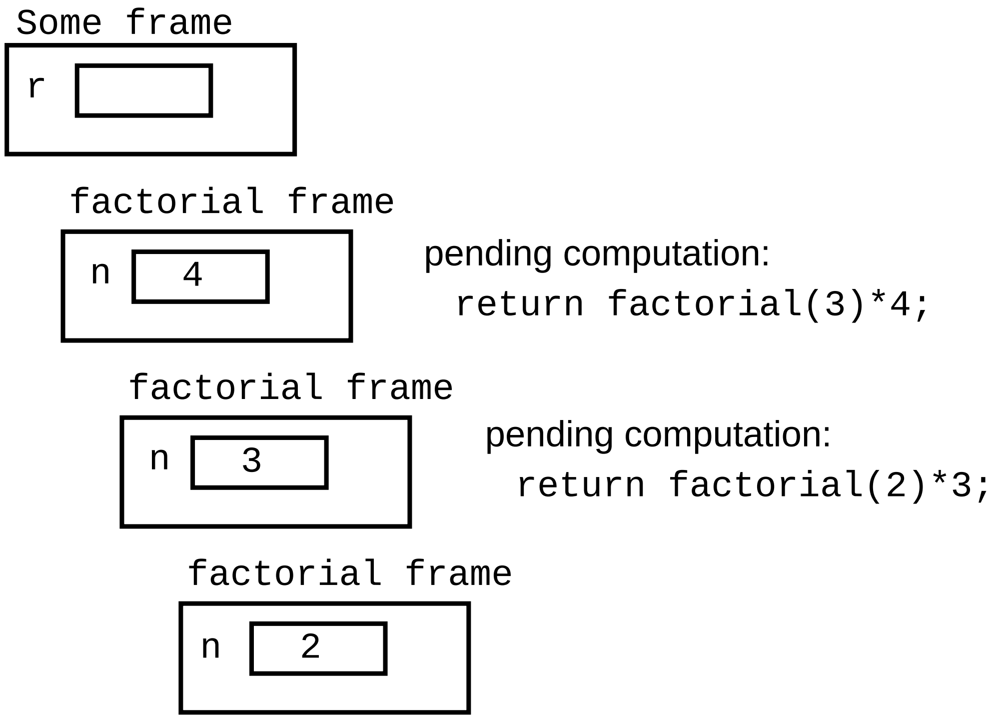
The pending computation here was:

```
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



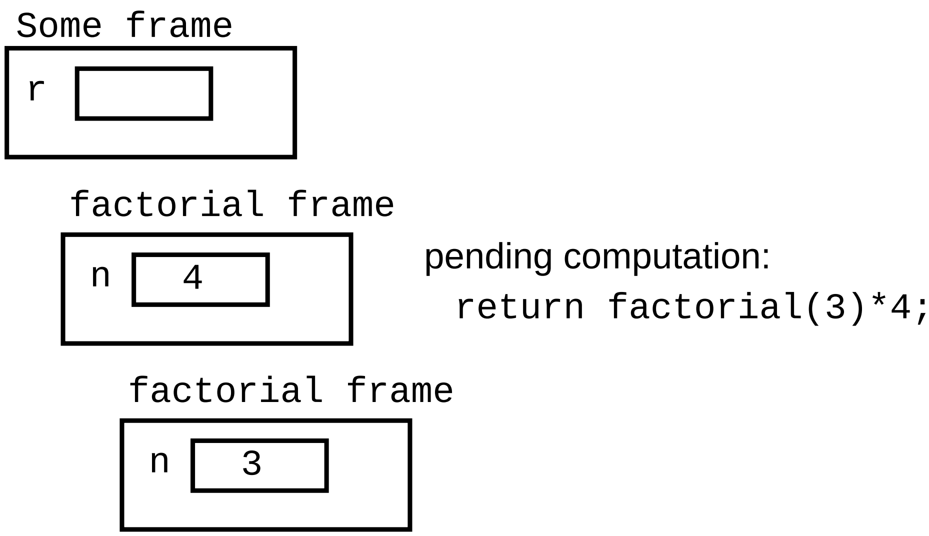
The pending computation here was:

```
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



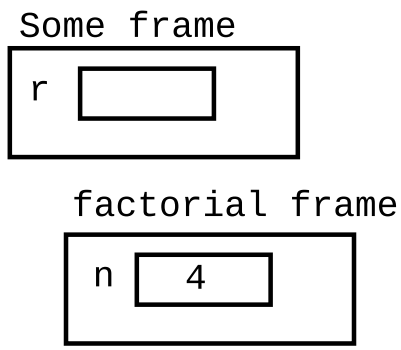
The pending computation here was:

```
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



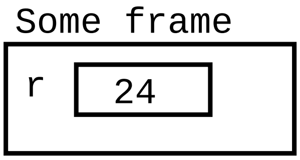
The pending computation here was:

```
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:

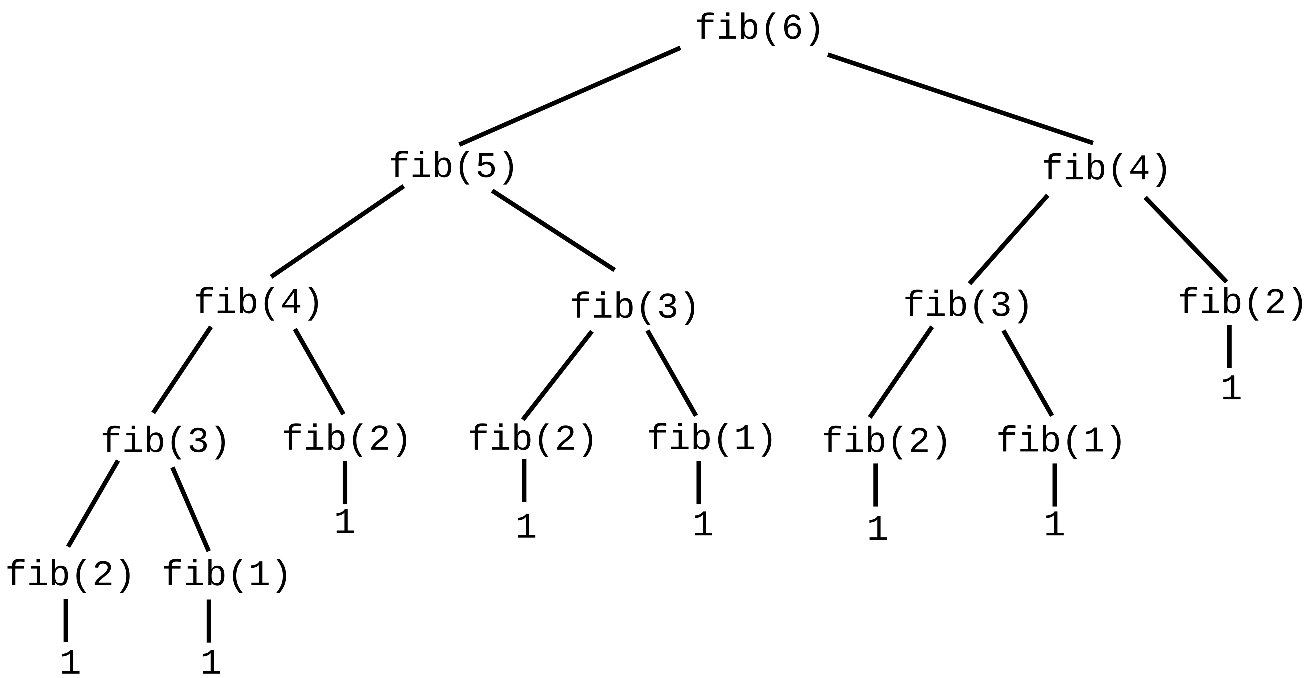
```
static int fib(int n)
{
    if (n <= 2) {
        return 1;
    }
    return fib(n-1)+fib(n-2);
}
```

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



The end