Exception handling

- An exception is generated (raised) with the *throw* statement:

  ```
  throw object;
  ```

  where *object* is an instance of a subclass of `Exception` or `Throwable`

- The *try-catch* statement:

  ```
  try {
    statements;
  }
  catch (ExceptionSubclass1 e) {
    statements1;
  }
  catch (ExceptionSubclass2 e) {
    statements2;
  }
  .
  .
  .
  ```

Exception handling

- A try-catch statement executes its default statements in sequence, and
  - If no exception is raised, then computation continues after the catch clauses
  - Otherwise, if an exception is raised, the sequence of statements is interrupted, and execution continues in the catch clause that matches the type of the exception

- After a catch clause finishes, computation continues after the try-catch. This is, the flow of control does not return to the point where the exception occurred. *Note: It never returns to the method that raised the exception, in contrast with a method call.*

- An exception which is not caught by a try-catch, is "propagated", i.e. it is raised again

Exception handling

```
int some_method(int b)
{
  int a, c, d;
  a = 5;
  if (b != 0) c = a / b;
  else c = 0;
  d = c + 2;
  return d;
}
```

Exception handling

```
int some_method(int b)
{
  int a, c, d;
  try {
    a = 5;
    c = a / b;
    d = c + 2;
  }
  catch (ArithmeticException e) {
    d = 2;
  }
  return d;
}
```

Exception handling

```
class SomeClass {
  static int some_method(int b)
  {
    int a, c, d;
    try {
      a = 5;
      c = a / b;
      d = c + 2;
    }
    catch (ArithmeticException e) {
      d = 2;
    }
    return d;
  }
  static void yet_another_method()
  {
    int x = 5, y;
    y = some_method(x);
    System.out.println(y);
  }
}
```

Exception handling

```
class SomeClass {
  static int some_method(int b) throws ArithmeticE
  {
    int a, c, d;
    a = 5;
    c = a / b;
    d = c + 2;
    return d;
  }
  static void yet_another_method()
  {
    int x = 5, y;
    try {
      y = some_method(x);
    }
    catch (ArithmeticException e) {
      y = 2;
    }
    System.out.println(y);
  }
}
```

Exception handling

```
class Food {
  boolean fresh, smelly;
}
class FoulSmell extends Exception {
  public String toString() {
    return “Yuck”;
  }
}
class FoodPoison extends Exception {
  public String toString() {
    return “Ouch”;
  }
}
```

Exception handling

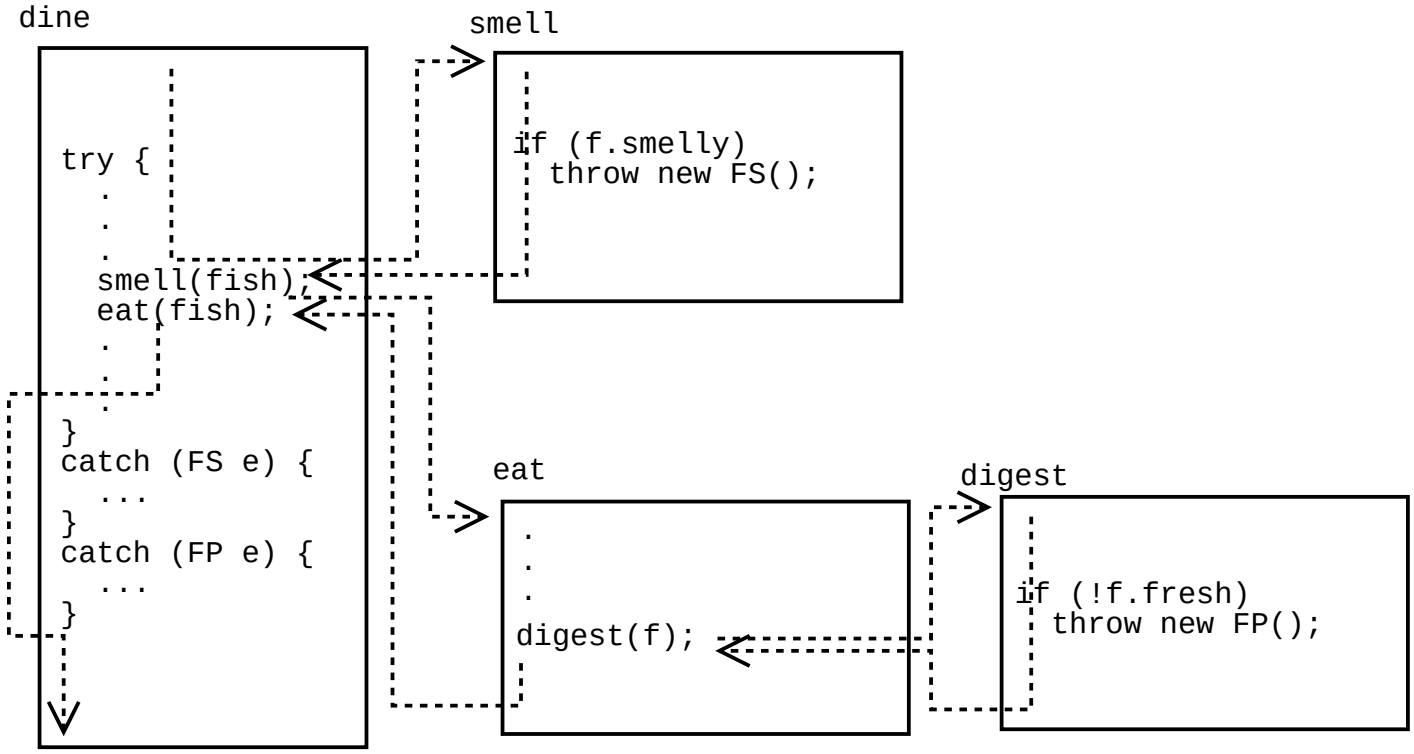
```
static void smell(Food f) throws FoulSmell
{
  if (f.smelly)
    throw new FoulSmell();
  System.out.println("Smells OK");;
}
static void eat(Food f) throws FoodPoison
{
  System.out.println("Hmmm...");
  digest(f);
}
static void digest(Food f) throws FoodPoison
{
  if (!f.fresh)
    throw new FoodPoison();
}
```

Exception handling

```
static void dine()
{
  try {
    Food fish = new Food();
    fish.smelly = false;
    fish.fresh = true;
    smell(fish);
    eat(fish);
  }
  catch (FoulSmell e) {
    System.out.println(e);
  }
  catch (FoodPoison e) {
    System.out.println(e);
  }
}
```

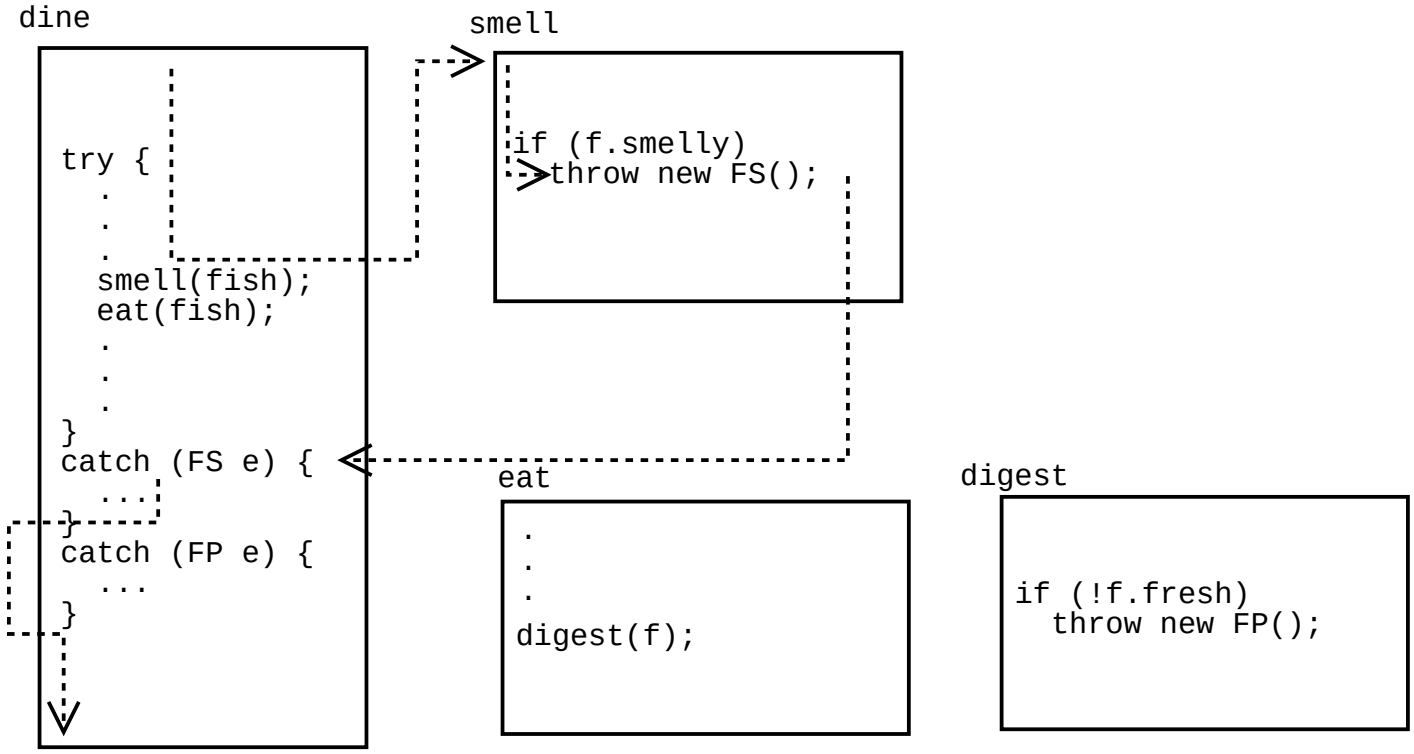
Exception handling

```
// fish.smelly = false; fish.fresh = true;
```



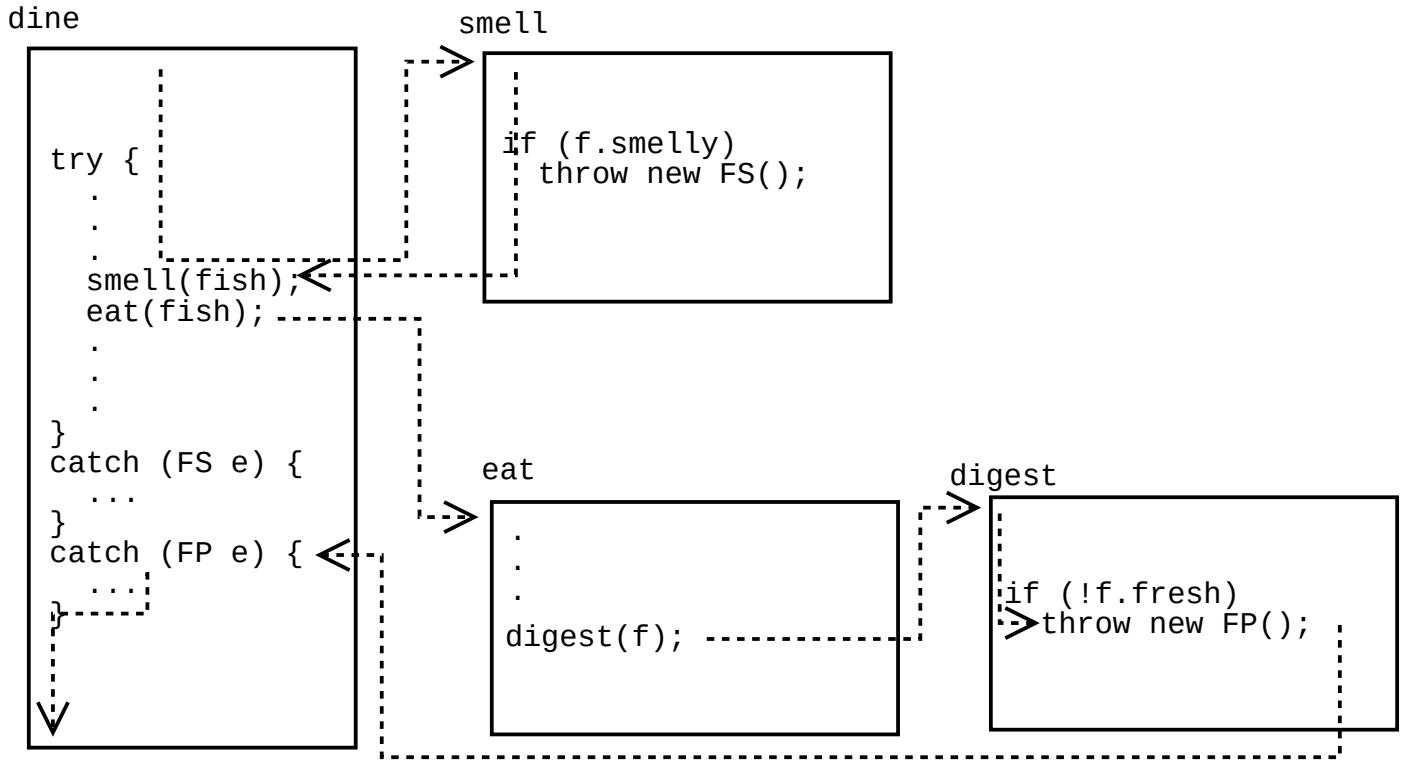
Exception handling

```
// fish.smelly = true;
```



Exception handling

```
// fish.smelly = false; fish.fresh = false;
```



Exception handling

- A method can throw more than one class of exceptions:

```
void m() throws A, B, ...
{
  ... throw new A() ...
  ... throw new B(...) ...
}
```

- ... but the exception needs not be raised explicitly in the method itself: it can be raised by another method called by `m`.

Exception handling

- Exceptions can be used not only for errors, but for control-flow too:

```
class Sheep {
  private int id;
  public Sheep(int i) { n = i; }
  public void jump()
  {
    System.out.println(“Sheep #”+id+” jumped”);
    if (id == 6)
      throw new LoudSound(i);
  }
}
```

Exception handling

```
class LoudSound extends Throwable {
  private int n;
  public LoudSound(int i) { n = i; }
  public toString()
  {
    return “I was in sheep #”+n;
  }
}
```

Exception handling

```
class GoToSleep {
  public static void main(String[] args)
  {
    try {
      for (int i = 1; i < 100; i++) {
        Sheep s = new Sheep(i);
        s.jump();
      }
      System.out.println(“zzzz...”);
    }
    catch (LoudSound s) {
      System.out.println(s);
    }
  }
}
```

Exception handling

- Some exceptions arise without an explicit throw.
- Some standard exceptions

```
Exception
  RunTimeException
    IndexOutOfBounds
    StringIndexOutOfBounds
    ArithmeticException (e.g. division by 0)
    NullPointerException
  NoSuchMethodException
  ClassNotFoundException
```

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:

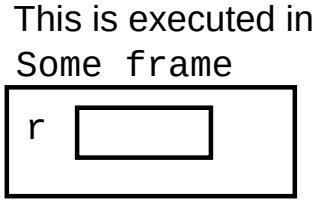
```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n-1)*n;
}
```

Execution of recursive methods

Consider the following client for this factorial function:

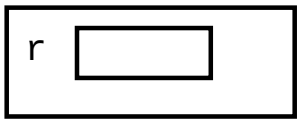
```
int r;
r = factorial(4);
```

Its execution proceeds as follows:

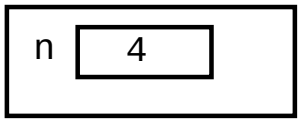


When we call `factorial(4);` a new frame for the method is created:

`Some frame`



`factorial frame`

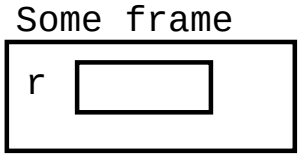


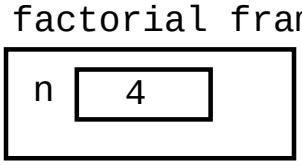
We execute the body of factorial; n is not 0 so we execute

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

which in this frame is the same as

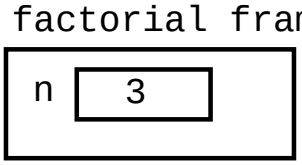
```
return factorial(4-1)*4;
```





pending computation:

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

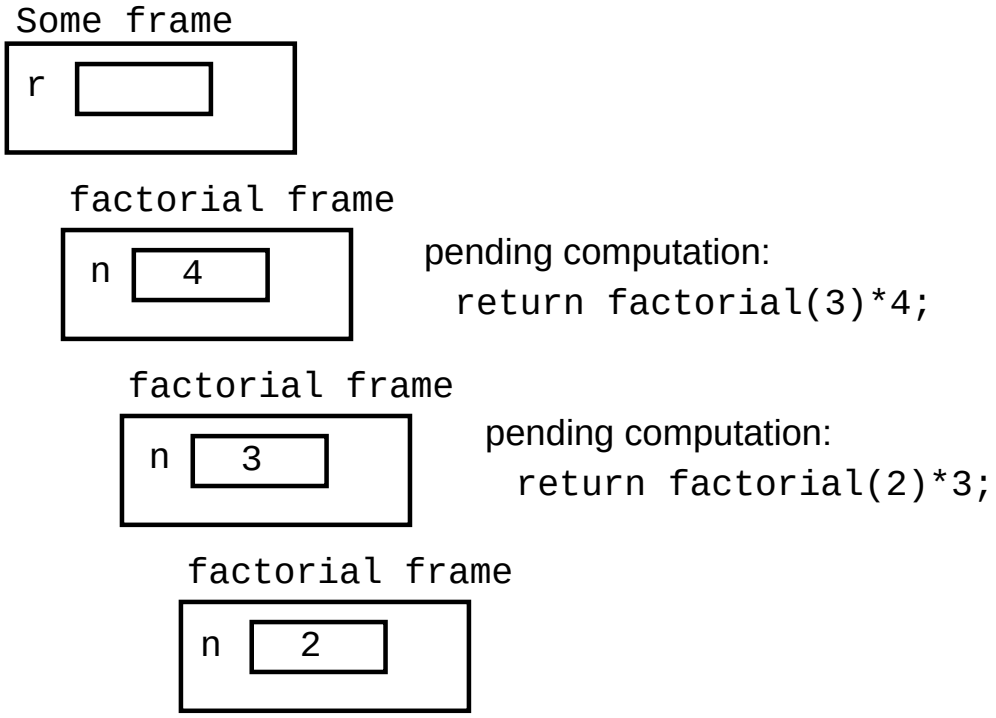


Again, we execute the body of factorial;
again, n is not 0 so we execute

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

which in this frame is the same as

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

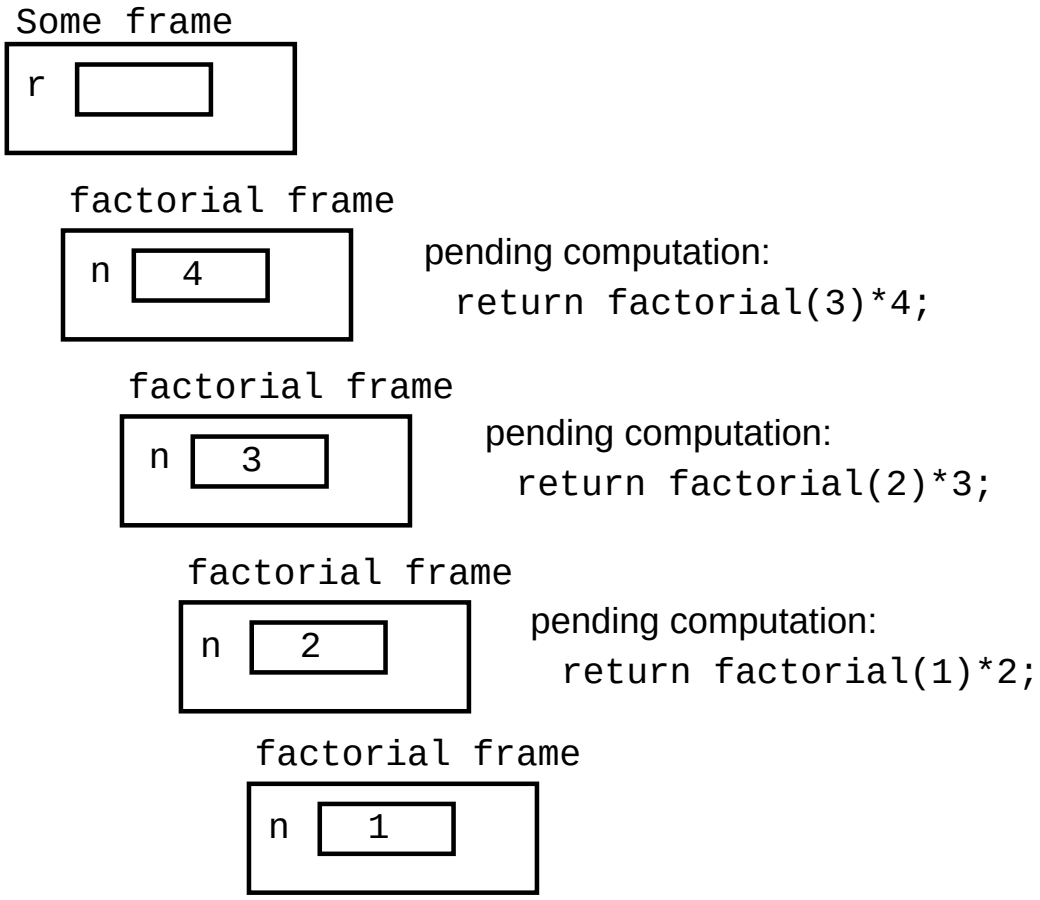


Again, we execute the body of factorial;
again, n is not 0 so we execute

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

which in this frame is the same as

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

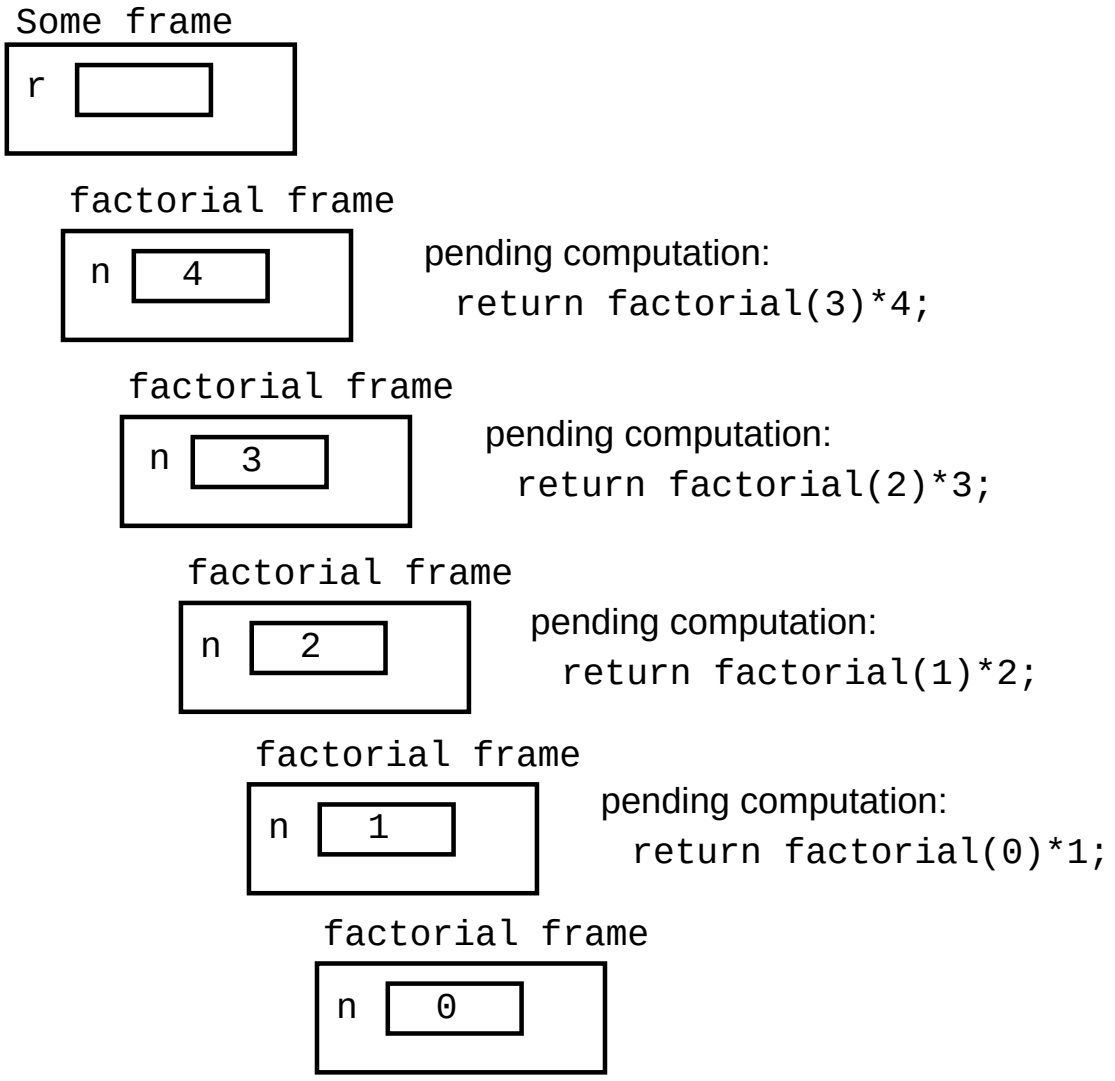


Again, we execute the body of factorial;
again, n is not 0 so we execute

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

which in this frame is the same as

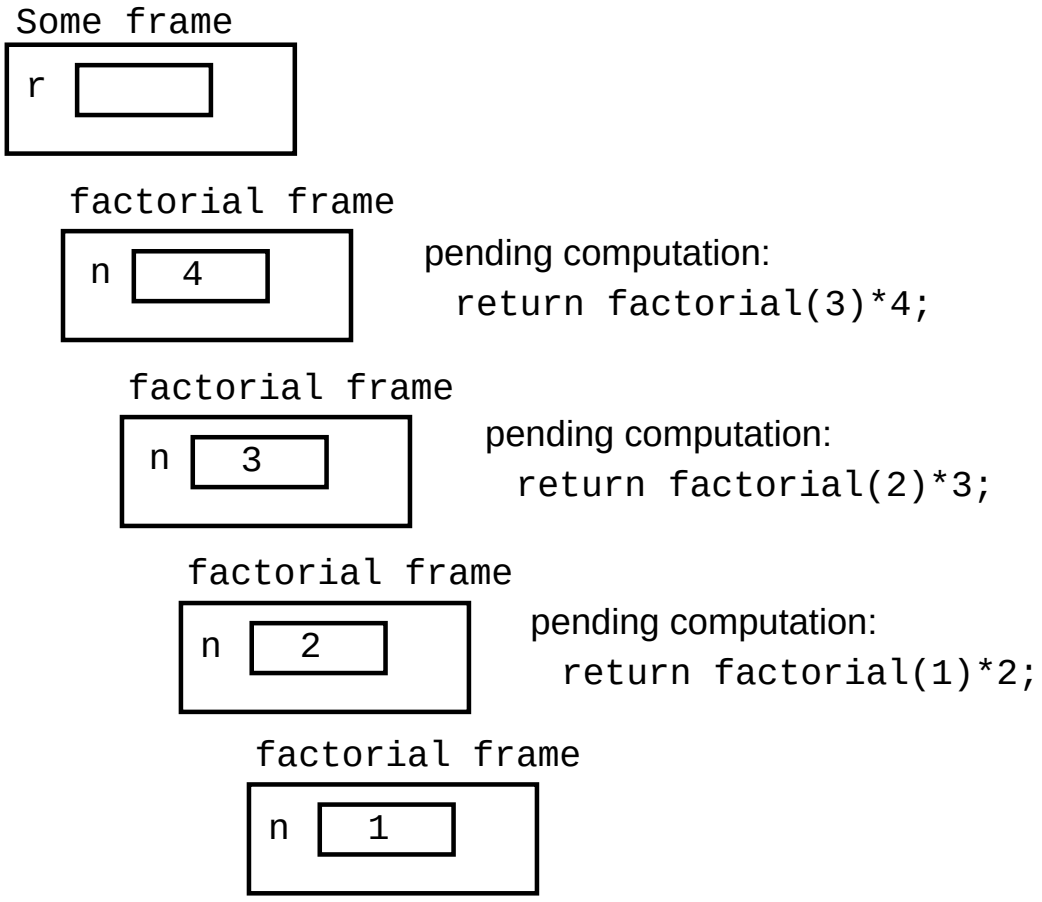
```
return factorial(1-1)*1;
```



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



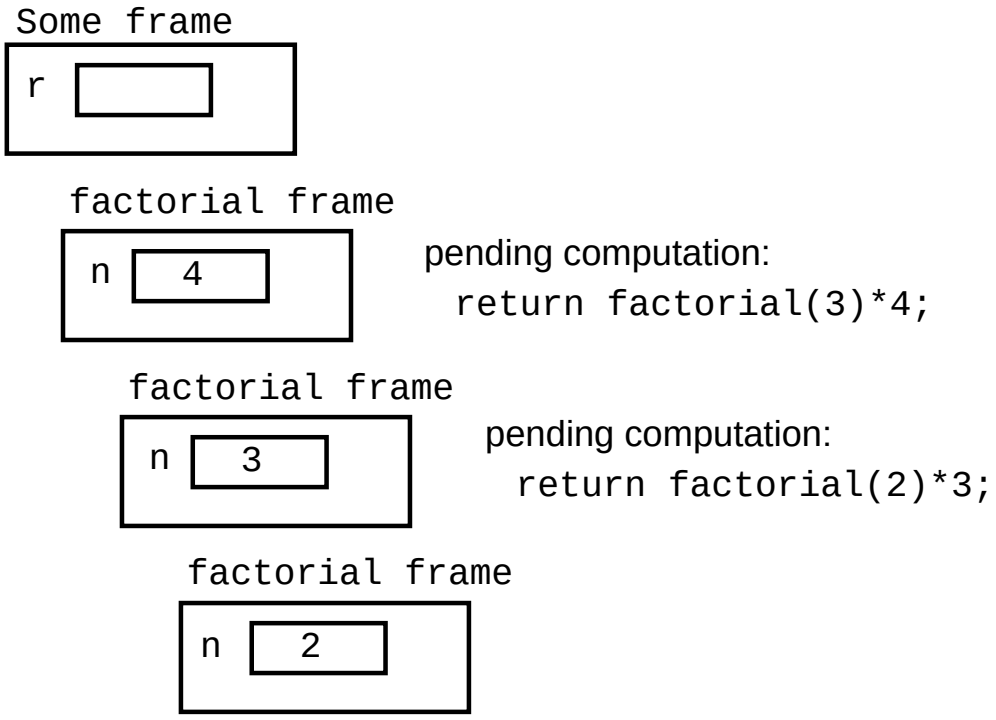
The pending computation here was:

```
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



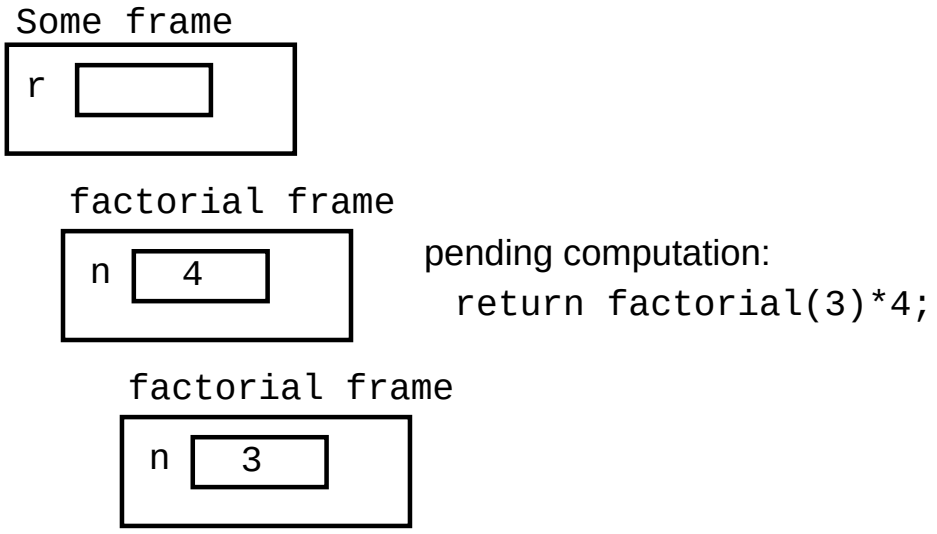
The pending computation here was:

```
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



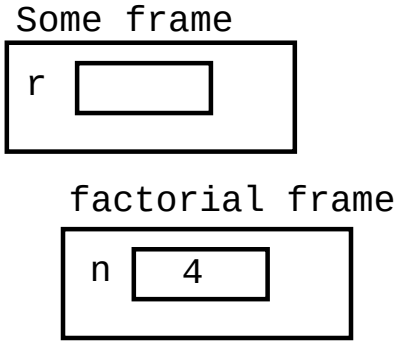
The pending computation here was:

```
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



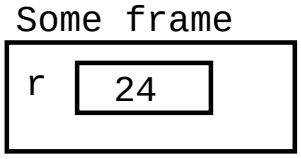
The pending computation here was:

```
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;
```

The end