In functional programming, each function we write ***just returns the value of a single expression.***

In Java, the body of such a function can be written as follows:

```
{
  return  a single expression  ;
}
```

Here is a very simple Java function of this kind:

```
static float f (int n, float x)
{
  return n+x ;
}
```

In functional programming, each function we write ***just returns the value of a single expression.***

In Java, the body of such a function can be written as follows:

```
{
   return  a single expression  ;
}
```

Here is a second Java function of this kind (that computes the **factorial** of a positive int argument):

```
// factorial(n) ⇒ n! = 1 * 2 * ... * (n-1) * n if 1 ≤ n ≤ 20
static long factorial (int n)
{
  return (n == 1)
           ? 1
           : factorial(n-1) * n ;
}
```

In functional programming, each function we write ***just returns the value of a single expression.***

Here is a second Java function of this kind:

```
// factorial(n) ⇒ n! = 1 * 2 * ... * (n-1) * n if 1 ≤ n ≤ 20
static long factorial (int n)
{
  return (n == 1) ? 1 : factorial(n-1) * n ;
}
```

Reminder re the Java (and C++) ? : Ternary Operator:

- If the value of *boolean-expr* is **true**, the value of *boolean-expr ? expr*$_1$ *: expr*$_2$ is the value of *expr*$_1$. (In this case *expr*$_2$ is *not* evaluated.)
- If the value of *boolean-expr* is **false**, the value of *boolean-expr ? expr*$_1$ *: expr*$_2$ is the value of *expr*$_2$. (In this case *expr*$_1$ is *not* evaluated.)

In functional programming, each function we write ***just returns the value of a single expression.***

Here is a second Java function of this kind:

```
// factorial(n) ⇒ n! = 1 * 2 * ... * (n-1) * n if 1 ≤ n ≤ 20
static long factorial (int n)
{
  return (n == 1) ? 1 : factorial(n-1) * n ;
}
```

Reminder re the Java (and C++) ? : Ternary Operator:

- If the value of *boolean-expr* is **true**, the value of *boolean-expr* ? $expr_1$: $expr_2$ is the value of $expr_1$. (In this case $expr_2$ is *not* evaluated.)
- If the value of *boolean-expr* is **false**, the value of *boolean-expr* ? $expr_1$: $expr_2$ is the value of $expr_2$. (In this case $expr_1$ is *not* evaluated.)

Example 1 The value of (3 < 4) ? 5+1 : 7+2 is: **6**
Example 2 The value of (3 > 4) ? 5+1 : 7+2 is: **9**

In functional programming, each function we write ***just returns the value of a single expression.***

Here is a second Java function of this kind:

```
// factorial(n) ⇒ n! = 1 * 2 * ... * (n-1) * n if 1 ≤ n ≤ 20
static long factorial (int n)
{
  return (n == 1) ? 1 : factorial(n-1) * n ;
}
```

Reminder re the Java (and C++) ? : Ternary Operator:

- If the value of *boolean-expr* is **true**, the value of *boolean-expr* ? $expr_1$: $expr_2$ is the value of $expr_1$. (In this case $expr_2$ is *not* evaluated.)
- If the value of *boolean-expr* is **false**, the value of *boolean-expr* ? $expr_1$: $expr_2$ is the value of $expr_2$. (In this case $expr_1$ is *not* evaluated.)

Example 3 The value of (3 < 4) ? 5+1 : 7/0 is: **6**

Example 4 The value of (3 > 4) ? 5/0 : 7+2 is: **9**

In functional programming, each function we write ***just returns the value of a single expression.***

Here is a second Java function of this kind:

```
// factorial(n) ⇒ n! = 1 * 2 * ... * (n-1) * n if 1 ≤ n ≤ 20
static long factorial (int n)
{
  return (n == 1) ? 1 : factorial(n-1) * n ;
}
```

Why Factorial Works

- When 1 < n ≤ 20, **factorial(n) returns the right result if factorial(n-1) returns the right result:**
  If factorial(n-1) returns 1 * 2 * ... * (n-1),
  then factorial(n) returns 1 * 2 * ... * (n-1) * n.
- factorial(1) returns the right result, 1, because evaluating (n==1) ? 1 : factorial(n-1) * n when n==1 does **not** cause factorial(n-1) * n to be evaluated.

Here is a third Java function of this kind:

```
// returns n^k if k > 0 and |n|^k < 2^63
static long pwr(long n, int k)
{
  return k == 1
           ? n
           : (k & 1) == 0            // true if k is even
               ? pwr(n*n, k/2)       // returned if k is even
               : pwr(n*n, k/2) * n;  // returned if k is odd
}                 // Note that / performs integer division!
```

Why pwr Works (bearing in mind that if $k > 1$ then $1 \leq k/2 < k$)

When $k > 1$ and $|n|^k < 2^{63}$, pwr(n,k) $\Rightarrow$ the right value, n^k, if the recursive call pwr(n*n, k/2) $\Rightarrow$ the right value, $(n*n)^{k/2}$, because:

- If k is even, $(n*n)^{k/2} = n^k$.
- If k is odd, $(n*n)^{k/2}*n = (n*n)^{(k-1)/2}*n = n^{k-1}*n = n^k$.

When $k = 1$, pwr(n,k) returns the right value, n.

Like many functions in functional programming, factorial and pwr use:

- **conditional expressions** (c ? e_1 : e_2 expressions)
- **recursion**

Functional programming also makes use of ***functions that take functions as arguments***:

As an illustration of this, consider a function with header

```
static long sigma(Function<Integer,Long> g, int m, int n)
```

that returns the *sum* of the results of applying the function given by its parameter g to each integer i, m ≤ i ≤ n.

Examples Suppose MyClass is the class that contains the above functions factorial and pwr. Then:

`sigma(MyClass::factorial, 3, 7)`

returns 3! + 4! + 5! + 6! + 7! = 5910.

`sigma(i->MyClass.pwr(i,5), 3, 7)`

returns $3^5 + 4^5 + 5^5 + 6^5 + 7^5 = 28975$.

Here `i->MyClass.pwr(i,5)` is a "lambda expression": It denotes an unnamed function that maps an integer i to i^5.

As another example, we now use the above function **sigma** to write a function

```
static long sum_powers(int m, int n, int k)
```

that returns $m^k + (m+1)^k + ... + n^k$.

Thus when m = 2, n = 5, and k = 4 we have that:

sum_powers(2, 5, 4) ⇒ $2^4 + 3^4 + 4^4 + 5^4$ = 16+81+256+625 = 978

This function can be written as follows:

```
static long sum_powers(int m, int n, int k)
{  return sigma(i -> MyClass.pwr(i,k), m, n);  }
```

The function sigma we have been using can be written in a functional style, as follows:

```
static long sigma (Function<Integer,Long> g, int m, int n)
{ return  (m > n)  ?  0  :  g.apply(m) + sigma(g, m+1, n); }
```

Here g.apply(m) *calls the Function given by the value of parameter* g, *passing* m's *value as its argument.* (Java does ***<u>not</u>*** allow this call to be written as g(m)!)

Functional programming can also use ***functions that return functions as their results***.

Any function that takes a function as argument or returns a function as its result is called a ***higher order function***.

Example of a Function That Returns a Function as Its Result

In math, we can *compose* functions $f : A \to B$ and $g : B \to C$ to give a function $g \circ f : A \to C$ such that $(g \circ f)(a) = g(f(a))$.

Thus if $f : \mathbb{Z}^+ \to \mathbb{Z}^+$ and $g : \mathbb{Z}^+ \to \mathbb{Z}^+$ are defined by $f(n) = n!$ and $g(n) = n + 5$, then $(g \circ f)(n) = n! + 5$.

Here is an analogous Java function:

```
static <A,B,C> Function<A,C> compose(Function<B,C> g,
                                     Function<A,B> f)
{ return n -> g.apply(f.apply(n)); }
```

`compose(n -> n+5, MyClass::factorial)` returns a function that maps `n` to `n! + 5`.

∴ `sigma(compose(n -> n+5, MyClass::factorial), 3, 6)` returns (3!+5) + (4!+5) + (5!+5) + (6!+5) = 890.

Common Lisp S-expressions

The textual expressions used in Lisp code are called ***S-expressions***; S stands for "symbolic".

- There are two kinds of S-expression: ***atoms*** and ***lists***
  The *empty list* is ***both*** a list *and* an atom; it is the ***only*** S-expression that's both a list and an atom.
  The empty list can be written as **()** or as **NIL**.
- The kinds of atom we will use in this course are:
  - **Numbers** [e.g., 129, -45.33, 72.1e-4, 67/4]
  - **Symbols** [e.g., X, DOG, APPLE23, NIL, FACTORIAL]
  - Strings [e.g., "asn.txt"]
    The only strings we will use will be filenames.
- There are two kinds of list:
  - **Proper Lists** [e.g., (GO (X (AT) 17) (HA Y) B)]
  - Dotted Lists [e.g., (GO (X (AT) 17) (HA Y) . B)]

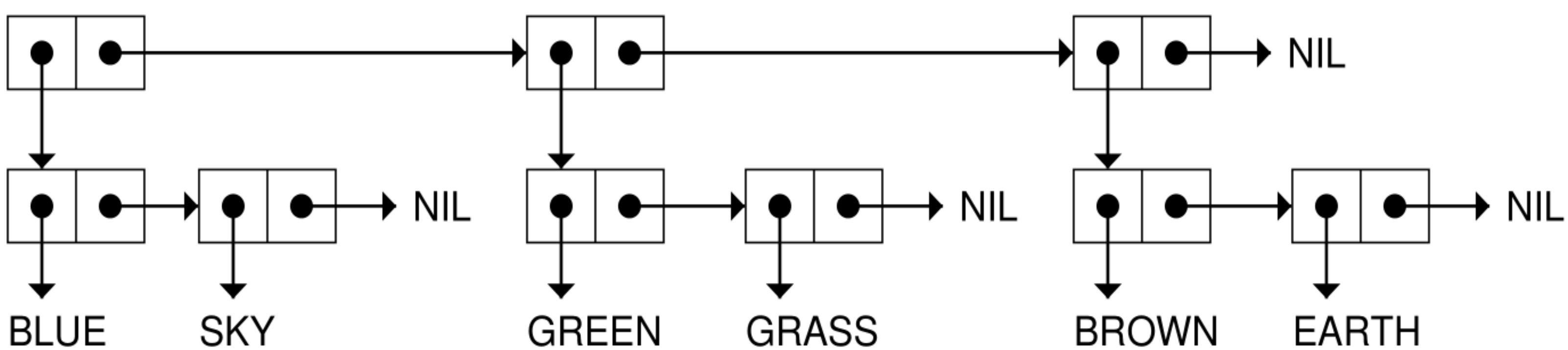
  If a function you write for this course returns a dotted list, then either your code has a bug or an inappropriate argument value was passed to the function.

- Each S-expression is a textual representation of a Lisp data object that's *also* called an S-expression.
- Similarly, the Lisp data objects that are represented by atoms, numbers, symbols, strings, and proper/dotted lists are *also* called atoms, numbers, symbols, strings and proper/dotted lists.

For example, the (proper) list

    ((BLUE SKY) (GREEN GRASS) (BROWN EARTH))

is a textual representation of a Lisp data object, *also* called a (proper) list, that is depicted as follows on p. 34 of Touretzky:



S-expressions are used:

1. As Lisp **code**--i.e., expressions that can be evaluated. For example,

   `(sqrt (+ (* 3 2) (- 4 1)))`

   is an S-expression that can be evaluated.

   - All Lisp code is in the form of S-expressions.
   - But most S-expressions *cannot* be regarded as Lisp code. For example, the S-expressions `((+ 2) y 5)` and `(3 x z)` *cannot* be evaluated.

2. As Lisp **data**. Example: `((john smith) (2001 06 13))`
   In this course:

   - If a Lisp variable has a value, then the value will usually be an S-expression.
   - More generally, if a Lisp expression can be evaluated, then its value (i.e., the result of its evaluation) will usually be an S-expression.