## Imperative vs. Functional Programming

C++ and Java were designed for object-oriented ***imperative*** programming.

In ***imperative*** programming we write code that is thought of as specifying a ***sequence of actions*** that the machine is to perform when the code is executed.

The specified actions often ***update values*** stored in variables, data structures, and components of objects.

***Functional*** programming is different.
In pure functional programming:

- The code ***isn't*** thought of as specifying a sequence of actions: We think of it as specifying ***what*** is to be computed rather than ***how*** the computation is to be done.
- Execution of the code will ***never update*** values stored in variables, data structures, and components of objects.

***Functional*** programming is different.
In pure functional programming:

- The code ***isn't*** thought of as specifying a sequence of actions: We think of it as specifying ***what*** is to be computed rather than ***how*** the computation is to be done.
- Execution of the code will ***never update*** values stored in variables, data structures, and components of objects.

A common way to state the second property above is to say that, *in pure functional programming, variables, data structures, and objects are* ***immutable***:

- Once a variable has been given a value, its value *stays the same for as long as the variable exists.*
- Once a data structure or object has been created, values stored in its components *stay the same for as long as the data structure/object exists.*

Functional programming is a style of programming in which the code we write consists of:

- Definitions of *functions that have **<u>no side-effects</u>***.
- Expressions that call such functions.

We say a function f ***<u>has no side-effects</u>*** if a call of f does nothing except return a value, which implies:

- Values stored in variables, data structures, and object components before a call of f are ***not*** changed by execution of f.
- Execution of f does ***not*** create or initialize any variable that can be used after f returns (though it may create and return a new data object).
- Execution of f does ***not*** do any I/O.
- Execution of f does ***not*** throw an exception.

[While it ***isn't*** a side-effect to change values stored in variables, data structures, or objects that are ***<u>local</u>*** to a function, the functions we write in pure functional programming are not even allowed to do that!]

Since variables, data structures, and objects are immutable, and functions have no side-effects:

1. Code is easier to understand, reason about, and debug.
   - Inaccurate thinking about how and when stored values change can lead to bugs in imperative code.
   - Functional code can also be easier to understand and debug because the flow of information during code execution is simple, limited, and entirely explicit:

     In functional code a function passes information to the functions it calls (via arguments) and returns information (the result) to its caller. That's all!

     Flow of information during execution of imperative code is much more complex: ***Changing a stored value gives updated information to all parts of the code with access to the value***; to understand the code we would have to understand just when and how the updated information is used.

Since variables, data structures, and objects are immutable, and functions have no side-effects:

1. Code is easier to understand, reason about, and debug.
2. Code is easier to test.
   - We don't have to worry that a function will update variables and data inappropriately.
   - Functions in functional code are typically ***pure***. A ***<u>pure</u>*** function is one that not only ***has no side-effects*** but also has the property that ***the result it returns depends only on the argument value(s) it receives***: So its result does ***<u>not</u>*** depend on the values of variables and data that are defined outside the function.

     A pure function can be tested just by calling it with different argument values and checking the results.

     When we are programming in a bottom-up style (so that each function is written before any function that calls it), a pure function can be tested in this way ***as soon as it has been written***.

Since variables, data structures, and objects are immutable, and functions have no side-effects:

1. Code is easier to understand, reason about, and debug.
2. Code is easier to test.
   - We don't have to worry that a function will update variables and data inappropriately.
   - Functions in functional code are typically ***pure***.

     A pure function can be tested just by calling it with different argument values and checking the results.

     When we are programming in a bottom-up style (so that each function is written before any function that calls it), a pure function can be tested in this way ***as soon as it has been written***.
3. Code can be automatically parellelized.
   - Different arguments of a function call can be evaluated in parallel, as evaluation of one argument expression cannot interfere with or affect evaluation of another.

Since variables, data structures, and objects are immutable, and functions have no side-effects:

1. Code is easier to understand, reason about, and debug.
2. Code is easier to test.
3. Code can be automatically parellelized.

However, when considering benefits of functional programming we should also bear in mind that immutability does have a cost:

When executed on computer systems that are in common use today, programs written in a functional style are generally less efficient (are slower and use more memory) than equivalent imperative programs.

- For example, imperative code that frequently updates individual elements of a large array cannot, in general, be replaced with similarly efficient functional code.