# The Imperative Paradigm

- Computation is accomplished by causing *values* (from expressions, input, etc.) to be *deposited* into (assigned to) *mutable cells* (variables, registers, memory cells, etc.) through precise *sequencing* of operations (via statements or control structures)
- Computation is performed by incremental changes to *state*
- Very close to the notion of a conventional vonNeumann architecture

# The Imperative Paradigm (cont.)

- Examples:
  - Assembly language
  - Java with a single class having only a `main` method and no expressions other than literals of the primitive types
  - A version of Python having no expressions other than literals of the primitive types (so no operators, no functions, etc.)

# The Procedural Paradigm

- A program is a (hierarchical) collection of *state transformers*
- State transformers may be constructed by combining other (simpler) state transformers
- Examples:
  - Java with a single class having only `void` methods and no expressions other than literals of the primitive types and calls to methods
  - Python with only `None`-returning functions and no expressions other than literals of the primitive types and calls to functions

# Shifts

- It might seem like the procedural paradigms is just the imperative paradigm with a couple extra “features” (primarily function/method calls/returns), but it’s really a major change in outlook
- It’s a so-called *paradigm shift*:
  - Horse to automobile
  - Calculator to computer
- Note also that you tend to worry less about low-level memory details than in the imperative paradigm

# The Object-Oriented Paradigm

- Computation is performed by a collection of *objects*
- Objects have *internal state* (which may be mutable), consisting of other objects
- Every object is an *instance* of a *class*
- The behavior of an object is determined by the object's class
- Typically, classes are hierarchically defined in terms of each other

# The Object-Oriented Paradigm (cont.)

- Computation proceeds when objects issue requests (in the form of *messages*) to other objects to perform actions
- When a message is received by an object, a *method* is *dispatched* by that object to process the message

# The Object-Oriented Paradigm (cont.)

- Notes
  - Java and Python are pretty unusual for object-oriented languages, for a number of reasons
  - The object-oriented paradigm isn't obligatorily related to the procedural paradigm, even though (many) people tend to think of it that way

# The Functional Paradigm

- Computation is performed by *applying functions* to *values*
- Here we mean functions in the mathematicological sense (aka the *referentially-transparent* sense)
- A program is a *composition of functions*
- There is no notion of mutable state at all
- So, there's no such thing as assignment in the functional paradigm

# The Functional Paradigm (cont.)

- So, there're no such things as variables (as the term is usually used in programming) in this paradigm
- This means that names (e.g., variable and function names) are completely unnecessary in this paradigm
  - But note that nearly all programming languages supporting this paradigm do permit the use of names, but primarily for named constants

# The Functional Paradigm (cont.)

- Examples:
  - A version of Java with only `final` variables, only non-`void` methods, expressions, and the `return` statement (but no other statements)
  - A version of Python with only value-returning functions, no assignment operations, no mutating-methods/operations, expressions, and the return statement (but no other statements)

# The Functional Paradigm (cont.)

- Functional programming languages tend to be smaller than other programming languages
- Functional programming languages tend to be trickier to implement “efficiently” than other programming languages. (Why?)
- Functional programming languages tend to have simpler semantics and are more suitable for formal reasoning, parallel programming, and other things

# Function Class

- Usually, functions are *first-class* entities in languages supporting the functional paradigm well
- Functions are *first-class* if
  - They can be created with complete generality
  - They can be passed as actual parameters to other functions with complete generality
  - They can be returned as the values of function applications with complete generality

# Function Class (cont.)

- Java methods are nowhere near being first-class
- Python functions are first-class, but they're not often exploited as such