OOP

Procedural vs. OOP

Nouns and Verbs

Nouns -- data
Verbs -- operations

Procedural Structure

C/Pascal/etc. ...
Verb oriented
- decomposition around the verbs -- dividing the big operation into a series of smaller and smaller operations.

Nouns/Verb structure is not formal
- The programmer can group the verbs and nouns together (ADTs), but it's just a convention and the compiler does not especially help out.

OOP Structure

Objects

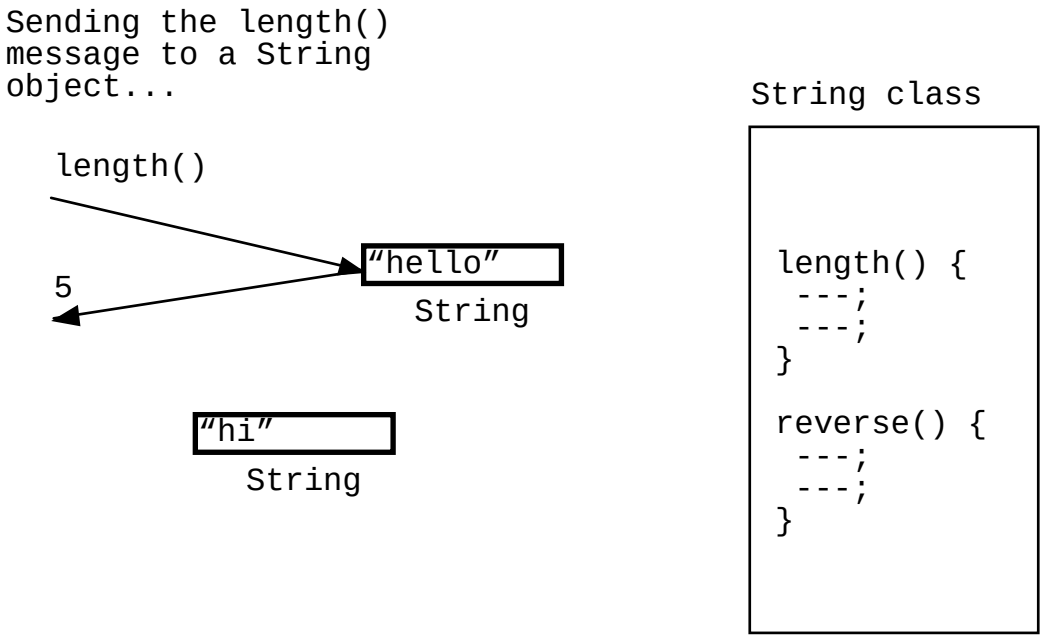
Storage
- Objects store state at runtime (ivars)

Behavior
- Objects will also in some sense take an active role. Each object has a set of operations that it can perform, usually on itself. (methods)

Class
- Every object belongs to a class that defines its storage and behavior.
- An object always remembers its class (in Java).
- "Instance" is another word for object -- an "instance" of a class.

Anthropomorphic -- self-contained
- Procedural variables are passive -- they just sit there. A procedure is called and it changes the variable.
- Objects are anthropomorphic-- the object has both storage and behavior to operate on that state.

String example
- Could have a couple String objects, each of which stores a sequence of characters. The objects belong to the String class. The String class defines the storage and operations for the String objects...



Class

Exists once -- there is one copy of the class in memory.
Defines the storage and behavior of its objects
Storage
Define the storage that objects of this class will have.
"instance variables" -- the variables that each object will use for its own storage.
Instance variables are usually just called "ivars".
Behavior
Define the behaviors that objects of this class can execute (methods).
String example
The String class defines the storage structure used by all String objects -- probably an array of chars of some sort
The String class also defines the operations that String objects can perform on themselves -- length(), reverse(), ...

Message / Receiver

Suppose we have Student objects, each of which has a current number of units. The message getUnits() requests the units from a student.
Java syntax:
a.getUnits()
send the "getUnits()" message to the receiver "a"
Receiver
The "receiver" is the object receiving the message. Typically, the operation uses the receiver's memory.

Method (code)

A "method" is executable code defined in a class.
The objects of a class can execute all the methods their class defines.

The String class defines the code for length() and reverse() methods. The methods are run by sending the "length()" or "reverse()" message to a String object.

Message -> Method resolution

Suppose a message is sent to an object --- x.reverse();
1. The receiver, x, is of some class -- suppose x is of the String class
2. Look in that class of the receiver for a matching reverse() method (code)
3. Execute that code "against" the receiver-- using its memory (instance variables)

In Java this is "dynamic" -- the message/method resolution uses the true, run-time class of the receiver.

OOP Design - Anthropomorphic, Modular

1. Objects responsible for their own state -- as much as possible, object's do not reach in to read or write the state of other objects.
2. Objects can send messages to each other -- requests
3. The object/message paradigm makes the program more modular internally. Each class deals with its own implementation details, but can be largely independent of the details of the other classes. They just exchange messages.

OOP Design Rule #1 -- Encapsulation

Objects "protect" their own state from direct access by other objects -- "encapsulation". Other objects can send requests, but only the receiver actually changes its own state. This allows more reliable software -- once a class is correct and debugged, putting it in a new context should not create new bugs.

Abstraction vs. Implementation

This is the old Abstract Data Type (ADT) style of separating the abstraction from the implementation, but structured as messages (abstraction) vs. methods (implementation)

OOP Design Process

Think about the objects that make up an application

Think about the behaviors or capabilities those objects should have

Endow the objects with those abilities as methods

If a capability does not occur to you in the initial design, that's ok. Add it to the appropriate class when needed -- the just needs to go in the right class

Co-operation

Objects send each other messages to co-operate

Tidy style

Experience shows that having each object operate on its own state is a pretty intuitive and modular way to organize things.