CS193j, Stanford
Winter, 2001-02

Handout #3
Nick Parlante

OOP Concepts

Object Oriented Programming, OOP, is the must influential paradigm of our time. This handout summarizes the most basic style, elements, and vocabulary of OOP that are common to all OOP languages. OOP languages can have weird features, but the basic ideas of OOP are pretty straightforward.

Pre-OOP

In a classical compiled language like Pascal or C, data-structures it is the programmer's duty to devise and enforce logical groupings of the data types and the functions that operate on them. The programmer can put related functions together in one file, but the grouping is just a convention and the compiler does not enforce it in a significant way. In C, a disciplined programmer can build code which is well structured. In OOP, much of the structure is a formal part of the language which makes it easier to get right.

OOP

In OOP, the logical arrangement of the code is changed. Instead of an informal arrangement of functions into different files, functionality is officially and tightly grouped with the type that it operates on. The OOP style groups all the operations together according to what they operate on — all the hash table operations are part of the hash table class, all the string operations are part of the string class. Put another way: if variables are nouns and functions are verbs, then OOP divides up everything around the nouns.

Historically, there are many ways to structure code, but the OOP style has proved to work quite well. OOP yields a pretty obvious decomposition and facilitates code re-use. Most importantly, OOP keeps the role of client and implementor separate. Keeping the client side separate from the implementation side has many advantages. Don't just think "oh yeah, client separate from implementor — too basic for me." It's not sexy, but it's useful. OOP formalizes and enforces the separation. It no longer requires any special skill to keep them separate; the language makes it the most convenient way to proceed.

The strengths of OOP help the most when writing large programs, programming in teams, and perhaps most importantly, packaging up code into libraries for use by others.

Libraries -- Code Re-Use

One of the obvious goals of language design has been to make it possible, finally, to stop re-creating common code for every project. Lists, hash-tables, windows, buttons ... these are things that should not have to be re-created every time. We would like a library once and for all to implement the hash-table in a modular way, and then all programs can just use it.

For code re-use to really work, it must be easy to be a client of that code -- in other words, the language needs good modularity features. OOP, and Java are good at modularity, so they are able to do a good job with libraries and so appear to be our best hope of finally obtaining broad code re-use.

The Clueless Client Test

The "clueless client test: a language has good support for code re-use if it is possible to build a library of code, give it to a client who wants that behavior, and that client can get the behavior from the library even if the client is clueless or a little clumsy (or, say, is in a big hurry). The language should encourage the right client behavior, discourage common client errors, and politely alert the client if there is an error. The DArray in C (an old CS107 C program), for example, fails the clueless client test: the client needs to use it exactly right, or it blows up in their face.

Ultimately, this is C's limitation in the modern world: modularity in C is not that easy to use. Therefore libraries are not that easy to use. For most programming problems, not having access to libraries for common code problems looms as a huge disadvantage. Likewise, this is one of Java's great advantages -- a thousand off-the-shelf library classes available for the programmer to use.

Class

The most basic concept in OOP is the Class. A class is like a type in classical language. Instead of just storing size and structural information for its data, a class also stores the operations which will apply to the data. Class = Storage + Behavior. A class is like an Abstract Data Type in Pascal or C— it creates a logical coupling between data and the operations on that data. Bundle the verbs with their nouns.

Object

An object is a run-time value that stores state and belongs to some class. Objects know what class they belong to, and so they automatically know what operations they are capable of. The word "instance" is another word for "object".

Message and Method

OOP uses "messages" instead of function calls. Sending a message to an object causes that object to perform an operation on itself. In that case, the object receiving the message and performing the operation is known as the "receiver". The receiver looks at the message, figures out the appropriate operation to perform, and executes that operation on itself. The receiver knows what operations it can perform, because it knows its class, and the class defines all the operations for its instances. The code corresponding to a particular message is known as the "method" for that message. A message is just a string like "pop()". The method for pop() is the code in the Stack class which is triggered by the "pop()" message. The C++ specific term for method is "member function".

```
// Traditional programming
// Call the foo() operation, and pass it the data to operate on
foo(x);


// OOP programming
// Send a "foo()" message to the object -- the "receiver" of the message.
// The receiver gets the message, finds the matching method code in its class,
// and the method code executes against the receiver.
x ← "foo()"
```

The method that executes depends on the class of the receiver. If you send the print() message to a Stack object, it executes the print() method in the Stack class.

If you send that same print() message to a HashTable object, you get the (different) print() method in the HashTable class. **It's what you say, and the class of the receiver that you say it to.**

Message Send Syntax

In Java and C++, the syntax for sending a message looks like appending the message to the desired receiver with a dot:

```
x.foo(); // Send the "foo()" message to the receiver "x"
```

Receiver Relative Code

We say that "the method executes against the receiver." This just means that the method code operates on the data of the receiver. So in code like the following...

```
x ← "removeAllElements()"

y ← "addElement(12)"
```

The receiver x is, apparently, going to remove all the elements it is storing, while the receiver y is adding an element. The method operates on the receiver it is sent to — x or y in this case. This makes methods different from plain C functions. C functions operate on the parameters they are passed. Methods, on the other hand, always have a receiver object to operate on. Parameters are only necessary for extra data -- notice how removeAllElements() above doesn't need any parameters. This "receiver relative" style is a nice feature of OOP -- it reduces the need for parameters somewhat.

Encapsulation

"Encapsulation" refers to protecting the internals of an object from direct manipulation by the client. The client can send messages, but the client cannot change the bits in the object directly. In C , it's just a convention that the client should not mess with or depend on the implementation of an Abstract Data Type (ADT). With encapsulation, the compiler enforces the separation. The class implementation can indicate which parts of the implementation are protected so that client code accessing those parts will not compile.

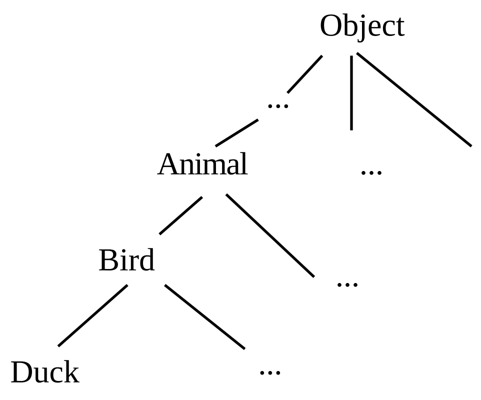
Or put another way: the client cannot mess with the object's state — the client can only send messages. The object's state is only touched by its own methods. Once those methods are correct and debugged, the object can be given to any client, and that client should not be able to perform an operation on the object which puts it in an incorrect state. This depends on the methods being correct which is still difficult. But when they are correct at last, the client should not be able to mess things up.

Part 2 — Inheritance

Modularity and encapsulation are perhaps the most important parts of OOP. OOP also includes a significant hierarchical component...

Hierarchy

Classes in OOP are arranged in a tree-like hierarchy. A class' "superclass" is the class above it in the tree. The classes below a class are its "subclasses." The semantics of the hierarchy are that classes have all the properties of their superclasses. In this way the hierarchy is general up towards the root and specific down towards its leaves. The hierarchy helps add logic to a collection of classes. It also enables similar classes to share properties through "inheritance" below. A hierarchy is useful if there are several classes which are fundamentally similar to each other. In C++, a "base class" is a synonym for superclass and "derived class" is a synonym for subclass.



Inheritance

"Inheritance" is the process by which a class inherits the properties of its superclasses. Methods in particular are inherited. When an object receives a message, it checks for a corresponding method in its class. If one is found, it is executed. Otherwise the search for a matching method travels up the tree to the superclass of the object's class. This means that a class automatically responds to all the messages of its superclasses.

Overriding

When an object receives a message, it checks its own methods first before consulting its superclass. This means that if the object's class and its superclass both contain a method for a message, the object's method takes precedence. In other words, the first method found in the hierarchy takes precedence. This is known as "overriding," because it gives a class an easy way to intercept messages before they get to its superclass. Most OOP languages implement overriding based on the run-time class of objects. In C++, run-time overriding is an option invoked with the "virtual" keyword.

Polymorphism

A big word for a simple concept. Often, many classes in a program will respond to some common message. In a graphics program, many of the classes are likely to implement the method "drawSelf()." In the program, such an object can safely be sent the drawSelf() message without knowing its exact class since all the classes implement or inherit drawSelf(). In other words, you can send the object a message without worrying about its class and be confident that it will just do the right thing. Polymorphism is important when the code is complex enough that you are no longer sure of the exact class of an object — you can just send it a message and rely on it doing the right thing at run time based on its class.