CS107
Autumn 2000

Handout 36
November 8, 2000

OOP Vocabulary

Handout written by Julie Zelenski.

My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is.
—T. Rensch *Object Oriented Programming*

Object-oriented programming is a paradigm present in a variety of programming languages. This handout summarizes the basic style, elements, and vocabulary that are common to object-oriented languages. OOP is a hot topic because it is proving itself to be a better way to engineer software, yielding better decomposition and more opportunities for code reuse. These strengths are especially important for large programs and for programs written by teams of programmers.

What is object-oriented programming?

The benefits of abstract data types (ADTs) have been proven over time: it makes good sense to logically group data-structures and the operations that manipulate them. However in a classical compiled language like Pascal or C, it is up to you, as the programmer, to devise and enforce these groupings. In contrast, an object-oriented language provides strong support for the association of procedures with each data type.

You can think of object-oriented programming as a way of breaking up program functionality into *objects*. An object is a data structure and a set of operations that can manipulate and access that data. An object is like a variable / data-type that has "come to life" and knows how to respond to requests. The programmer no longer matches up the right procedure with the right data-type, instead, variables themselves know which operations they implement.

An object-oriented program is designed as a collection of interacting objects, each of which provides a specific functionality. These objects communicate with one another by sending messages. These messages can request that an object perform a certain action, or ask the object to reveal or change some of its data. The conventional function call is replaced by message-passing.
This requires a different approach to programming. Traditional imperative programming breaks the problem into steps and directs the flow of control through those steps. The object-oriented approach to problem solving relies on decomposing the program into objects which have well-defined areas of responsibility and putting these objects together like building blocks to handle the task as hand.

Object-oriented languages start by providing good support for software engineering practices such as abstraction and modularity and adding inheritance to facilitate code-sharing. The characteristics of any object-oriented language will include:

Data abstraction:	Objects provide the ultimate “black box” by grouping functionality with data. The abstract data type is only accessed and modified through a well-defined set of methods, known as its public interface. The implementor can change the implementation as long as it remains true to the interface specification. This means you can modify data structures, use faster algorithms, fix bugs, or extend functionality without disrupting how a data type is used.
Encapsulation:	Each object has its own private data, these internals can be protected or hidden by the compiler, and the public interface provides the only means that can be used to ask the object to modify that data. The user of an object has full visibility to those methods offered by an object, and no visibility to its data. This ensures data integrity.
Inheritance:	A new data-type can be defined by modifying the definition of an existing data-type. The new type inherits all of the data and behavior of its parent type. You can add new data and functionality as well as change the behavior of existing operations. This allows you to re-use code and construct a hierarchical partitioning of functionality.
Polymorphism:	"Many forms." The same message can be sent to more than one kind of object. In conventional programming, the programmer is responsible for choosing functions that operate properly on a given data type, in object-oriented programming, the data type (the object) is responsible for providing the appropriate implementation.

Vocabulary

Because the concepts here are fairly different than in conventional programming, many new terms have been introduced to talk about these ideas.

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, a class also includes the operations relevant to that data type. A class is a super-ADT (abstract data type)— it bundles traditional data typing information with the actions that will operate on that type.

Object	An object is a run-time value which belongs to some class. If a class is a type, an object is a variable. A variable of a given class is called an *instance* of that class. The interface for a class defines the properties that all of its instances have.
Messaging	Instead of calling a function to perform some operation on an object, you send the object a *message* asking it to perform that operation on itself. It is the object's responsibility to consider the message and do what action is appropriate. Depending on the class of the object, different code may be executed. This is useful since the same message may by meaningful to different classes, although the actual code may be different. The implementation for a message is called a *method.* A message is a string like "Pop", the method for "Pop" is the code in the Stack class which is triggered by the "Pop" Message. There is only one "Pop" message, but many classes may have Pop methods.
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 it are *subclasses*. By convention, the root of the tree is called the "Object" class. The semantics of the hierarchy are that any class includes all the properties of its superclasses. In this way the hierarchy is general towards the root and specific towards its leaves. The hierarchy helps add logic to a collection of classes. It also enables similar classes to share properties through inheritance.
Inheritance	A subclass *inherits* all of the data and functionality of its parent classes. In particular, a class inherits all of the methods. When an object receives a message, it checks for a corresponding method. 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 and so on recursively. This means that a class automatically responds to all the messages of its superclasses. Most OOP languages include controls to limit how the data and methods are inherited. A subclass can also extend beyond the inherited functionality by adding data and defining new methods.
Overriding	A class inherits all the methods of its superclasses, but a class can choose to respond to a message in a different way by re-defining a method. When an object receives a message, it checks its own methods before consulting its superclass. If the object's class and its superclass both contain a method for a message, the object's

method is used. In other words, the first method found in the hierarchy takes precedence. When a subclass responds to a message in a different way than its superclass does, the subclass is said to have *overridden* its superclass's method— the class overrides and intercepts the message before it gets to the superclass.

Polymorphism A big word for a simple concept. Often, many classes in a program will respond to some common message, but in different ways. In a graphics program, many classes are likely to implement the method "DrawSelf", and each version is different. In the program, any graphic object can safely be sent the DrawSelf message without knowing its exact class since all the classes implement 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 where the class of an object cannot be determined at compile-time.

Programming Philosophy

The thought processes of the programmer are as much a part of object-oriented programming as any of the terms above. Programs must be thought of as a collection of cooperating pieces of data (objects) rather than a thread of control. Design becomes data-oriented rather than process-oriented. Certain tasks like user-interface programming are fundamentally object-oriented.

Object-oriented programmers use phrases like: “when the user presses this button, it sends a message to this object which calculates the answer and then sends a message to this text field to update the display and...” instead of “we wait for the user to click the mouse, and then we figure which button they had the mouse over and, based on that, we decide what action to do, and call a function to do that, then we go back to waiting for the user to do something else...“