Print

Object-Oriented Programming

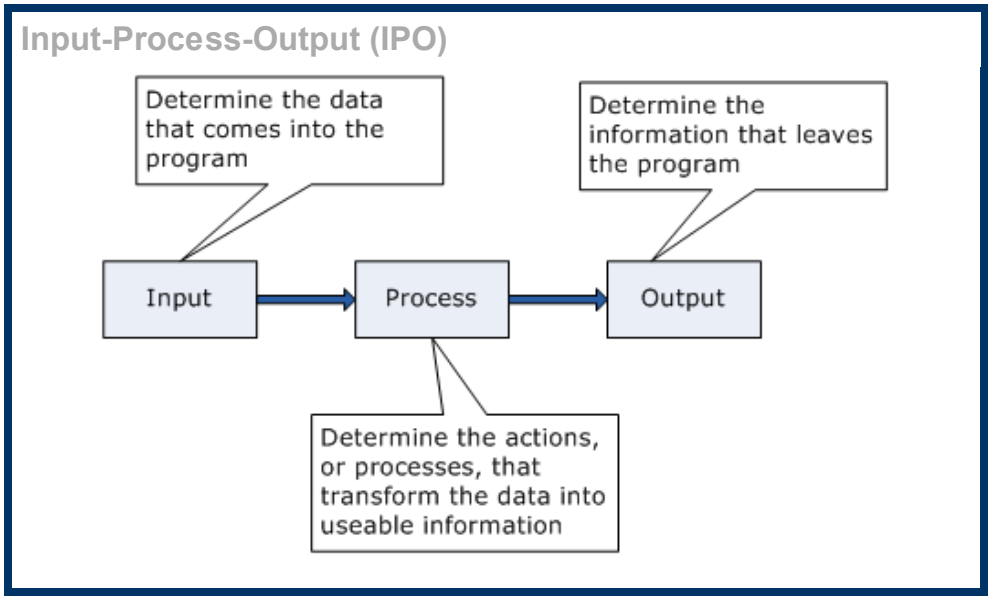
Software Development Process

Back to Top

In prior courses you used a structured, procedural programming strategy in your programming solutions. Procedural programming focuses mainly on logic associated with how data is processed, as represented in the **IPO Module** you learned about in your systems analysis and design course.

For example:

- **Input** the data;
- **Process** the data (convert it to useful information);
- **Output** the processed data (information).



Click here for transcript of this image.

One structured programming characteristic to note is that in structured programs the data and processes are separate entities and are only loosely related to one another. This is one of the distinguishing characteristics between structured and object oriented programs: in structured programs the data and processes are separated, but as we will see, in object oriented programs the data and processing are contained in a single entity called a **class**.

Introduction to Object-Oriented Terminology

Back to Top

Many students enter this course with a sense of fear because they may have heard how hard it is to understand object-oriented programming. They hear that object-oriented languages like Java, C#, or VB.NET are difficult to learn, so they start the course with a high level of anxiety. But the difficulty comes because we like to concentrate on the programming language, NOT the programming concepts. If you concentrate on concepts, then learning the language becomes much easier!

All of the object-oriented languages are based on the same principles of classes, polymorphism, inheritance, and operations, and if you strive to understand these concepts, learning the languages is fairly straightforward. Learning the first object-oriented language is always the hardest, but you will find that once you understand the principles of one language, learning the others will be a snap since you will only be learning how to use a new tool to implement the same concepts. Twenty years ago, carpenters used hammers and nails to build a house; these days they all use nail guns. The concepts and techniques used to build a house are the same. Only the tools have changed. This holds true for object-oriented programming languages as well; once you know the concepts and techniques, learning how to use a new tool is relatively easy. In this course, we are going to concentrate on the object-oriented concepts and practice them using C++. The students in the Java and C# sections are learning the same concepts; the only difference is that they are using a different tool to implement the concepts.

The question then becomes, why do we think object-oriented programming is so hard? Many believe it is hard because it is not a natural way for human beings to "think." However, I suggest that you already know and understand all of the underlying principles of object-oriented analysis and design, and that this is actually the way we human beings tend to think and conceptualize problems in the everyday world. To demonstrate, let us look at a real-world analogy.

Suppose you want a new home theater system that has a TV, receiver, speakers, disc player, and radio tuner. It is probably clear that you think of a stereo system as components, or objects, where each object performs a specific and unique function. Speakers will contain tweeters and woofers, and it would not make much sense to put the tweeters in the DVD player. The TV contains a tube and a frequency tuner that would not go into the speakers. That is, each object contains features and functions that are logically related to the services the component is providing to the system. This simple concept is one of the fundamental characteristics of object-oriented programming and is called **encapsulation**.

We will use the same concept of encapsulation in our object-oriented programs. When you determine what your requirements are for the system, you will partition the requirements into logical sets of features that the system will perform. For example, you will normally come up with a set of printing requirements, a set of user interface requirements, a set of database access requirements, etc. Then, as you elaborate on these requirements, you will put them into the components where they logically belong. If you have a requirement to print a quarterly report, this would go into the printing component. If you have a requirement to query the database, it would go into the data component. The principle of encapsulation only implies that you want to group logical functions and include all the data you need to complete those functions into a single component. This is a very natural concept that we see and use every day.

A disc player is a very complex component made up of hundreds of internal sub-circuits that perform the task of reading a disc. However, as a user of the disc player, there are only a few things we can do to interact with it. There are, at most, three plugs we use to connect the player to the receiver, and we also have a few buttons we use to control the operation. So we see that the complex processing and data required to produce music is hidden, or kept private from outside components. This concept is called **data/information hiding** and is another key concept of object-oriented systems that we need to understand. Think of what would happen if we could get into the internal circuits of the disc player--yikes. The player would not work very long, and we might get quite a shock! The same is true of our object-oriented programs. We provide external components with the absolute minimum amount of information and interaction necessary to get the job done--no more, no less--and the internal details of the processing are kept hidden.

We know that the components of a stereo do not operate in isolation and must work together to produce music. The disc player reads a disc and passes the signals to the receiver, and the receiver then passes the signals to the speakers so you can hear the music. Thus, the components of the stereo must pass **messages** between themselves in a well-defined sequence to produce music. The same is true of the object-oriented designs that we will generate. Each object provides

some specific service and then passes messages to the other objects in the system to produce the final result.

If you look at the CD player, DVD player, or tape player, you will see that each of them has play, stop, and pause buttons. However, since these devices are for uniquely different functions, each of these operations is implemented differently inside the device. But from a human perspective, they all perform the same logical operation of controlling playback. This is called **polymorphism**, which means the same operation may behave differently on different classes. For example, when someone says to you, "Let's go driving," a particular image comes to mind. However, if you say, "Let's go drive the car," a very specific implementation of driving is invoked. Now, if you say, "Let's go drive the boat," or "Let's go drive the motorcycle," in each case you are logically still driving, but the implementation of driving is different.

In the example above, you may have noticed that the term used was disc player, which could mean a CD player, DVD player, Video Disc player, or so on. If you consider these different types of disc players, you will notice they all have some common characteristics. For example, they all have play, pause, stop, and disc eject buttons. However, each of them has a few characteristics that make them unique. For instance, a CD player only has two audio plugs that are used to connect to the receiver, the DVD player will have one optical plug, and a video disc player will have three plugs. So, in the description above, the *term disc* player was used **generalize** the three basic types of disc players into one **superclass**, which describes and contains all of the attributes and operations common to all types of disk players. Each of the specific types of disc players inherits all the attributes and operations of the superclass, and then each subclass of player includes specific operations and attributes to make a unique type of disc player. By defining these types of **generalizations**, or **inheritance**, we are able to simplify the design and reduce the total number of classes.

To summarize, these features are predominant in object-oriented programming languages:

- Classes
- Objects
- Encapsulation
- Interfaces
- Polymorphism
- Inheritance

- A **class** is a template from which **objects** are instantiated (created). For example, an Automobile class could have a Dodge as an object.
- **Objects** have attributes (data) and behaviors (methods). For the Dodge object given in the example above, the color blue could be one of its attributes and MoveForward could be one of its methods.
- **Encapsulation** refers to the techniques of packaging these attributes and properties into one standalone unit. These units can be looked at as a black box.
- To be able to use (or reuse) this black box in a program, all we really need to know is how to **interface** with it; we do not necessarily need to know what it contains within.
- **Polymorphism** (which may be one of the more complex features of object-oriented programming) is the ability to create methods that act differently depending on the context.
- **Inheritance** is the ability to extend a class to a "derived class." As an example, an AntiqueCar class may be derived from the Car class.

While there are other more subtle characteristics of object-oriented systems that we will discuss in the upcoming weeks, an understanding of encapsulation, data/information hiding, message passing, and inheritance forms the basis for understanding the other concepts that we will address in the upcoming weeks. So, if you understand these basic underlying concepts, which are very natural to us, then you will be well on your way to understanding object-oriented programming. As we begin our journey into learning C++, remember to concentrate on object-oriented concepts. If you do, it will be much easier for you to understand why we do things the way we do.

Class and Objects

Back to Top

Object-oriented programming uses a different approach than structured programming. Instead of placing the focus on the data as done in structured programming, object-oriented programs look at the real-world objects involved in the scenario, observe their characteristics, their behaviors, and interactions, and eventually encapsulate all the data and operations necessary to process the data into a **class**.

So, what exactly is a class?

A **class** is a blueprint of a set of real-world objects that share the same attributes and behaviors. For example, we can define a *Car* **class** that specifies the blueprint for an actual car. Our *Car* **class** may specify that a car needs to have a color, a number of wheels, a manufacturer, a serial number, and a model. It can also specify how a car moves, stops, accelerates, and responds to questions such as "Who is your manufacturer?", "How many miles have you driven so far?", etc. Note that a blueprint is not an actual car; we create specific car **objects** based on the *Car* **class** definition and, as you probably have realized, we can have many specific instances (**objects**) of a car based on the same *Car* **class** blueprint.

So, what exactly is an object?

Objects are people, places, or things (nouns) that are included in your problem scenario, which can be further generalized into a program structure called a **class**. For example, John's car, which is a *Chevrolet Camaro, built in 2010, blue, has 4 wheels, serial number L12341Z, with a current mileage of 20,125 miles*, is an **object** of our Car **class**. The specific values of the attributes in object John's car represent the **state** of that object.

So, through the analysis process of real-world objects, we determine their important **attributes** (characteristics) and **behaviors** (operations). The behavior is identified using the verbs in the system description and the characteristics are identified using the adjectives or some nouns in the problem statement or user requirements. By encapsulating the attributes and behaviors within a **class**, a real-world object is generalized as a specific data type in the code of an object-oriented program.

Every class consists of **class members**. For example, attributes are the variables that we will create in the classes and behaviors are also known as operations or methods. It should be noted that the variables in our classes can also be instances of other objects; that is, classes can be composed of other objects (i.e., a car engine is composed of several pistons).

When a class is designed, we strive for the following goals:

- Bind together attributes and behaviors into one unit so both elements will belong together in a group to form a composite data object.
- Support encapsulation and information hiding so programmers can separate what is to be done from how it is to be done.
- Decide what behaviors and attributes are allowed to be accessed from outside the class.

Each time we add classes to a program, we need to make sure that we pay attention to these goals to ensure that we have quality object-oriented programs.

Now that we understand the basic concept of a class, it is important to understand that the class is a data type that is the centerpiece of any object-oriented program. All object-oriented programming languages and their respective software development environments give us the tools to create user-defined data types that more closely match the problem domain we are trying to model.

Object-Oriented Design Example

Back to Top

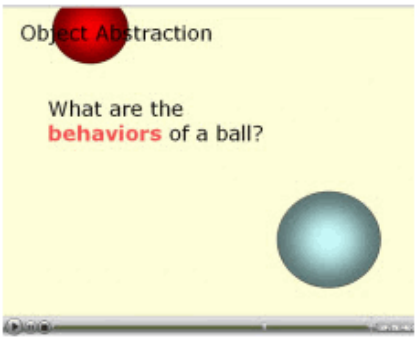
Now, let's drill down and look at object orientation as it relates to a simple real-world object, a ball. What are the attributes of a ball (characteristics that describe it)? Every ball has color and a size described by its radius. Every ball is also composed of a specific material, but to keep this example simple, we will not include this particular characteristic of a ball in our demonstration. This demonstrates that when we create our class designs, we should only include the minimum amount of data required depending on the problem scenario. In this example, the important details are color and size, but the material is not important so we leave it out.

Now, what are the behaviors of a ball (what is a ball responsible for doing)? Well, if you drop a ball from a specific height, it will rebound to a predictable height. Also, if you push a ball along a flat surface, the ball rolls. As we do our analysis, we then determine the behaviors we want to include in our ball class. In this example, we want to include the roll and bounce behavior.

Ball Abstraction

Click on the link above to view the tutorial.

This is a short tutorial that defines what objects are and how to develop an abstraction of an object based on its attributes and behaviors.



These object-oriented attributes and behaviors are then modeled in our object-oriented programs, but before we code the class, let's look at a "picture" of the Ball class using a conceptual class diagram that is the most common way of documenting the structure of a class. The picture is called a **class/object diagram** and is part of the **Unified Modeling Language** (UML), which is the industry standard for documenting the structure and behavior of an object-oriented program. The UML symbol used to represent a class is a square, which is divided into three sections (cells). The name of the class is always given in the first cell. Use Pascal casing naming convention for class name.

Ball	The name of the class, Ball, is always given in the first cell. Use Pascal casing for class names.
color **radius**	The second cell contains a list of the state variables of the Ball class. These are the attributes used to describe a specific ball. Use camel casing for state variable names.

Roll( ) **Bounce**	The third cell contains a list of the methods of the Ball class. These are the behaviors of a ball - what a ball does. Use Pascal casing for method names.

Note: the symbol + means "public" and - means "private members."

Using this strategy, **classes** are used to convert the program from one large process to many simpler processes. Each of these smaller processes is coded and debugged separately, then tested as an interactive, cohesive unit of code.

C++ Syntax of Class Definition - Data Member and Member Functions

Back to Top

A class is a collection of a fixed number of components that are called members of the class. These members can be either member functions or data members. Data members within a C++ class are declared just as any other variable would be declared. If the member of the class is a function, the function prototype is used to declare it. The best way to demonstrate these concepts is with an example. The following diagram shows a base class that defines time. This is a well illustrated example in many textbooks that cover C++ and is also the one used in the textbook for this class.

```
class Time
{
public:
    void setTime(int, int);
    void getTime(int&, int&);
    void printTime() const;
    void incrementMin();
    void incrementHrs();

private:
    int hour;
    int minute;
};
```

Now, let's look at some of the key points.

- The class is called Time. The word class is a reserved one in C++. The members of the class are enclosed within curly braces. The closing curly brace needs to be terminated with a semicolon or else a syntax error will occur.
- The above class has seven members, five of which are member functions and two of which are data members. The functions are declared by using the function prototype.
- Although the members of a function can appear in any order, all members are private by default. It is common practice to start off with the public members (by using the access modifier public), and then follow them with the private members of the class.
- The private data members cannot be accessed outside of the class. This is a necessary factor in information hiding. Normally, you do not want to make the data members of an object accessible to the user or to other members outside of the class for security purposes.

Because the class has been defined, we can now instantiate objects of the class. This is shown in the code below:

```
class Time
{public:
    void setTime(int, int);
    void getTime(int&, int&);
    void printTime() const
    void incrementMin();
    void incrementHrs();
private:
    int hour;
    int minute;
};
int main()
{
Time firstTime;
Time secondTime;

firstTime.setTime(12,10);//these two functions are legal since the functions are public
secondTime.setTime(5,36);

firstTime.hour = 12;//these two will produce error since the data members are private
firstTime.minute = 10;

return 0;
}
```

- Two objects were created in the code above. Both of the objects are time objects and will have the same data members as the class. When an object is created, it will be able to access all public functions of the class. Any private members, however, will not be accessible, and thus, the last two lines of code above will cause an error.
- It should be noted, however, that if an object is created (instantiated) within the definition of a member function, then it will have access to both the private and public members of the class.

Controlling Access to Class Members - Private, Public, and Protected

Back to Top

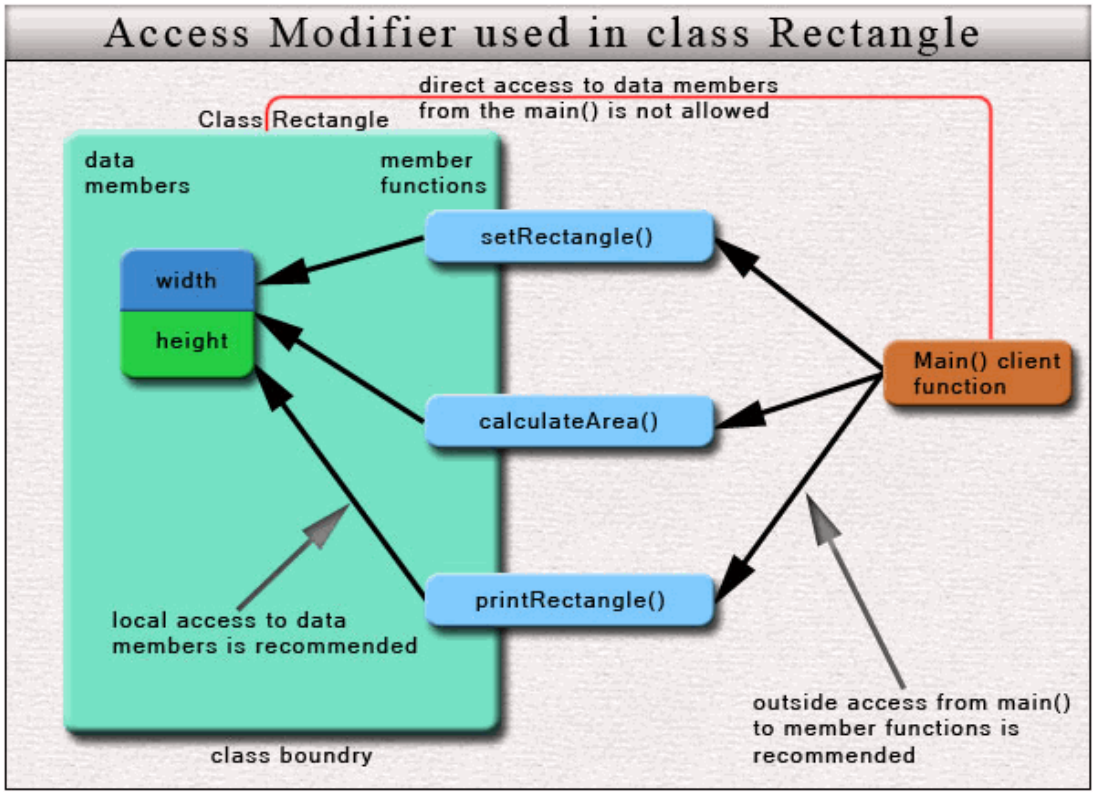
Early in the lecture, I mentioned one goal of C++ class construction: to allow the server programmer to gain access to all class components from outside of the class. This can be accomplished by making the right choice of member functions and the right choice of data members. In this section, we will discuss the techniques that we can use that will allow the class designer to gain and control access to class member functions and data members. Take a look at the following program -- a Rectangle class.

```
#include <iostream>
using namespace std;
class Rectangle
{ // start of the class scope

private: //private access modifier for data members
int width, height; // data members to access
public: //public access modifier for members functions
void setRectangle(double aWidth, double aHeight) // set both data members
{
      width = aWidth;
      height = aHeight;
}
double calculateArea() // calculate the area of a rectangle
{
      return width*height;
}
void printRectangle() // print object state
{
       cout << "Width = " <<width << " Height = " <<height <<endl;
}
} ; // end of the class scope
int main()// the server functions that server programmer maintains
{
      Rectangle r1, r2; // define two data
      r1.setRectangle(4,10); // set width and height
      r2.setRectangle(12,5);
      r1.printRectangle ();
      r2.printRectangle ();
      return 0;
}
```

Click here for transcript of this image.

The following diagram describes the class Rectangle and its relationship of main() function. There are three components in this class: the border that separates everything inside the class from everything outside the class, the data members, and the member functions. As you can see, the two data members are declared as private components inside of the class. The three member functions implementations are declared as public components inside of the class. However, the interfaces that are known to the client are used outside of the class. When the main() function needs the values of the Rectangle fields (e.g., computing area, setting the field values, or printing the values), main() calls the member functions getArea(), setRectangle(), and printRectangle() instead of directly accessing the two private data fields: width and height. The dashed line below means that the direct access to data members is not allowed.



Click here for transcript of this image.

Program Documentation Back to Top

There are two reasons for limiting access to the data members.

- The first reason is to limit the changes to the program when data designs are to be changed. We only need to change the member function implementation (function body) if the interfaces of member functions stay the same; the client main() functions remain untouched. This is very important for maintenance purposes. The set of functions that have to change is well defined - they are all listed in the class definition, and there is no need to inspect the rest of the program for possible implications.
- The second reason for not allowing direct access to data members is that the client main() function expressed in terms of function calls to member functions is a lot easier to understand than the code expressed in terms of detailed computations over the value of data members. The member functions now have the responsibility of doing the work for the client main() function.

In order to achieve these two advantages, data members inside of the class should be private to the class, making them inaccessible from outside of it. Member functions instead should be declared public, making them accessible from outside of the class. This way, we could also prevent the client main() function from creating dependencies on the data members. We don't want to create dependencies between different parts of a program. In general, dependency in the programming world means

- difficulties in code reuse in the same project;
- more changes of code during project maintenance than necessary;
- more cooperation among programmers during the project development period.

In summary:

- When data members are tagged as private, they are only accessible to the member functions of the same class and to the friend functions (friend functions will be covered later on).

- When member functions are tagged as public, they are available to the rest of the program.
- The two rules listed above do not actually prevent you from making your data member private and making your member functions public. In traditional design, however, data members are kept private while member functions are kept public.
- Although public member functions are used commonly, private member functions are rarely used in real practice. The use of the private member function outside of the containing class will result in a syntax error.
- In addition to the public and private access modifiers, C++ supports a third access modifier - protected. When your class members are tagged as protected, they are only available to the member functions of the base class and to the member functions of the derived classes (derived class is the one that inherits from the base class either directly or indirectly. The details of inheritance will be covered in detail during a later week).
- When no access modifier is present, C++ class members will default to private. Do not rely on the defaults; instead, please specify your access modifiers explicitly.
- All rules listed above will enhance reusability of your overall class design and modifiability of your project. Applying these rules can enhance your program quality while reducing the likelihood of human miscommunication and decreasing the amount of coordination among your team members.

Summary

Back to Top

While there are other subtle characteristics of object-oriented systems, an understanding of encapsulation, data/information hiding, message passing, and inheritance form the basis for understanding other concepts that we will address in the upcoming weeks. To reiterate,

- encapsulation - grouping together common operations and attributes into a single component;
- data abstraction/information hiding - exposing only those attributes and operations that are necessary for the system to work; keep everything else hidden;
- message passing - identifying the communications paths and content that allow the different components to interact;
- generalization/inheritance - defining a superclass that contains common attributes and operations and then defining specific types of subclasses based on the superclass.

As stated in the introduction, once you get over the initial hump, you will find that object-oriented analysis and design is naturally intuitive and fun. If you have any questions concerning the concepts or would like any further explanation, please post a question to the TDAs.

Back to Top