Print

Classes and Encapsulation

Encapsulation and Data/Information Hiding

Back to Top

Let's review the Component Stereo example from the Week 1 lecture since a component stereo system illustrates the concepts of encapsulation and data/information hiding that is so natural to us. What we then want to do is approach our object-oriented programs using the same basic concepts of encapsulation and data/information hiding.

The Stereo Analogy (A Review)

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 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 programs, and it is called **encapsulation**.

When you determine what your requirements are for the system, you will partition the requirements into logical sets of features that the system must 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 the requirements into the components where they logically belong. If you need 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 tells us that you want to group logical functions and include all the data that you need to complete those functions into a single component. This is a very natural concept that we see and use every day.

NOTE: A quality object-oriented program is achieved only through careful abstraction and encapsulation of object details of a problem domain.

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, then 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.

We all know that 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 users 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 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 we need to understand. Think 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 designs. 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.

Programming Example on Encapsulation and Data/Information Hiding

Back to Top

After analyzing the stereo system example, we saw that a stereo system is simply a set of components, or objects, and each object is responsible for performing a very specific and unique task. Each piece of our home theater system can be viewed as an object that contains features and functions that are logically related to the services the component is providing to the system. For example, the speakers project audio and that is all they do. The TV is responsible for visually displaying a picture and again, that is all it does. Each object or system component possesses a very well-defined and specific purpose and when we combine all the home theater components together, we have a functioning system.

Each piece of our system also possesses very specific characteristics or attributes that set it apart from all the other components. For example, a speaker may have a tweeter and a TV contains a screen. These attributes, along with the well-established behaviors or functions of the components, together fully describe each component. When designing software, programmers combine a component's attributes with its functionality in the form of a class. This simple concept of combining the attributes and behaviors of a single entity into a single class is one of the fundamental characteristics of object-oriented programming and is called **encapsulation**.

Encapsulation can also be thought of as the process of packaging an object's attributes and methods into one standalone unit that can interact as a single entity much like a black box. The term black box is sometimes used to refer to a system that can be understood solely based on its inputs, a brief description of its processing and the resulting outputs.

You can use a black box without knowing any of the specific details of its internal functionality. With well-written code, all that a user needs to know is how to interface with the object. Take an iPod as an example of a black box. Most of us don't know (and probably don't care) how the internal mechanisms of the device work. What we do care about is being able to select songs, listen to music, view videos, and so forth. In addition to the fact that the user does not have to bother with the complexity of the internal workings of the device, encapsulation also provides a secure environment for the maker (or programmer) because of the data hiding that takes place. Using our iPod

example, a user is given a well-defined interface including buttons, an earphone jack, and maybe even a touchscreen with which he or she can access the functionality of the iPod without knowing anything about the internal workings of the device.

We use this concept of encapsulation in our object-oriented programs to not only create robust and logical code but also to protect the data held inside our objects. Remember, the principle of encapsulation implies that you want to group logical functions and include all the data you need to complete those functions into a single component. What would happen if our data became corrupt? So many things could go wrong if our home theater objects suddenly contained bad data. Just think about the ramifications of an employeePayroll object or a healthCareProcedure object operating with incorrect data! Someone might not receive a paycheck or might receive an incorrect medical procedure! As you can see, protecting our data, also known as data/information hiding, is extremely important. Thankfully, encapsulation assists us in protecting our data!

Recall our Week 1 lecture once again. We talked about how a disc player is a very complex component made up of hundreds of internal sub-circuits that perform the task of reading a disc. In order to limit access or protect these circuits, users of the disc player are very limited in how they can interact with it. There are, at most, three plugs we use to connect the player to the receiver. We also have access to a limited set of buttons which control the operation of the player. So, we see that the complex processing and data required to produce music is hidden, or kept private from outside components. This type of data/information hiding (literally hiding the internal operations and circuitry from the users) protects our disc player by preventing users from having direct access to its internal contents. This also applies to our object-oriented programs. Our classes provide external components with the absolute minimum amount of information and interaction necessary, hiding the internal processing. We do this by implementing access modifiers such as public and private.

Let us take a look at a simple example shown below: a simple Rectangle class with two data members and no member functions.

```
class Rectangle
{
public:
      int width;
      int height;
};
```

The program prompts the user to enter the dimensions of two rectangles. If the area of the first rectangle is larger than the area of the second one, the program scales the second rectangle up by increasing each of its dimensions twice and prints its final dimensions.

Below is the coding example of direct access to data members--no encapsulation used yet.

```
1  //Example of direct access to data members - no encapsulation used yet
2  #include <iostream>
3  using namespace std;
4  class Rectangle
5  {
6  public:
7
8        int width;
9        int height;
10 };
11 int main()
12 {
13 Rectangle r1, r2;
14 cout << "Enter both dimensions of the first rectangle: ";
15 cin >> r1.width >> r1.height;
16 cout << "Enter both dimensions of the second rectangle: ";
17 cin >> r2.width >> r2.height;
18 if (r1.width*r1.height > r2.width*r2.height )
19 {
20 r2.width = r2.width*2;
21 r2.height = r2.height*2;
22 cout << "The new dimension of the second rectangle is \n"
23 <<"width= "<<r2.width<<" height= "<<r2.height<<endl;
24 }
25 else
26 {    cout << "There is no change of the dimensions of the second rectangle\n";    }
27 return 0;
28 }
29
```

Click here for transcript of this image.

Program Sample Output:

```
Enter both dimensions of the first rectangle:  4 9
Enter both dimensions of the second rectangle: 2 4
The new dimension of the second rectangle is
width= 4 height= 8
Press any key to continue . . .
```

The following coding example now demonstrates the proper combination of information hiding with encapsulation.

Click on image to enlarge.

```
1  // This version of code combines both encapsulation and information hiding.
2  #include <iostream>
3  using namespace std;
4  class Rectangle
5  {
6  public:
7       int width, height;
8  };
9  void enterDimension(Rectangle &r)
10 {
11 cout << "Enter both dimensions for the rectangle ";
12 cin >> r.width >> r.height;
13 if (r.width < 0)
14     r.width = 0;
15 if (r.height < 0)
16     r.height = 0;
17 }
18 bool compareArea(const Rectangle& r1, const Rectangle& r2)
19 {
20 if (r1.width*r1.height > r2.width*r2.height)
21     return true;
22 else
23     return false;
24 }
25 void scaleUp(Rectangle &r, int percent)
```

```
30 void printRectangle(const Rectangle &r)
31 {
32 cout << "The new dimension of the second rectangle is\n";
33 cout << "width= " <<r.width << " height= " <<r.height <<endl;
34 }
35 int main()
36 {
37 Rectangle r1, r2;
38
39 enterDimension(r1);
40 enterDimension(r2);
41
42 if (compareArea(r1,r2))
43 {
44 scaleUp( r2, 200);
45 printRectangle(r2);
46 }
47 else
48 {
49 cout << " There is no change in the dimensions of the second rectangle\n ";
50 }
51 return 0;
```

```
26  {
27  r.width = r.width*percent/100;
28  r.height = r.height*percent/100; }
29  }
```

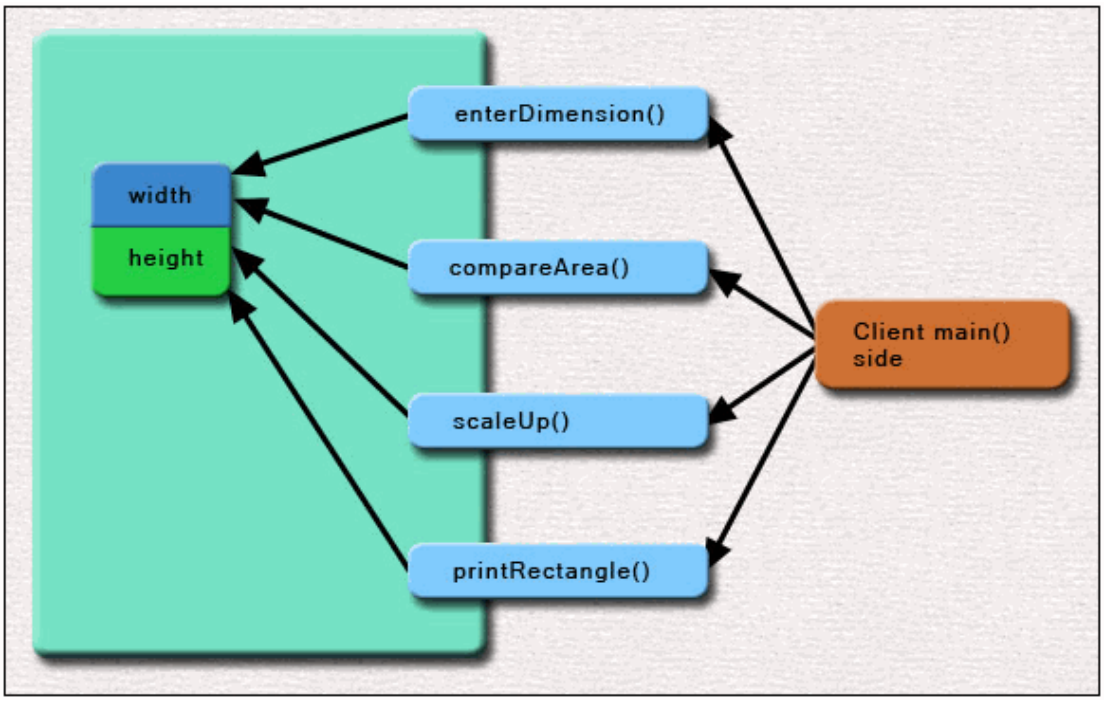
Lines 1-29

```
53  }
```

Lines 30-53

Click here for transcript of this program.

Object Diagram:



Program Sample Output:

```
Enter both dimensions for the rectangle 2 6
Enter both dimensions for the rectangle 5 8
 There is no change of the dimensions of the second rectangle
 Press any key to continue . . .
```

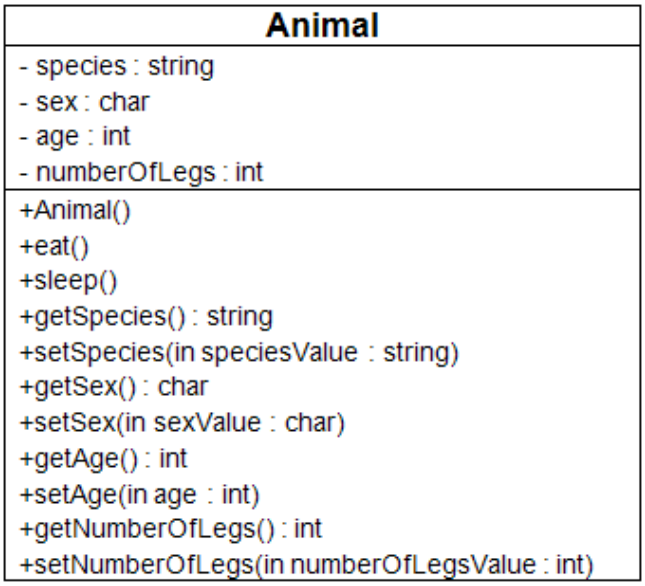
Class Anatomy

Back to Top

The Unified Modeling Language (UML) provides a way to communicate the structure of a class in a consistent way. There are three components in a UML diagram to describe a class:

1. Class name: Unique identifier;
2. Attributes: Characteristics that further describe a class; and
3. Operations: Actions that a class may engage in (also called behaviors or methods).

UML diagrams can be created using Word, Visio, or by using a comprehensive development solution such as IBM's Rational Rose. Let us look at how each of the three components above is included in the following UML class diagram.

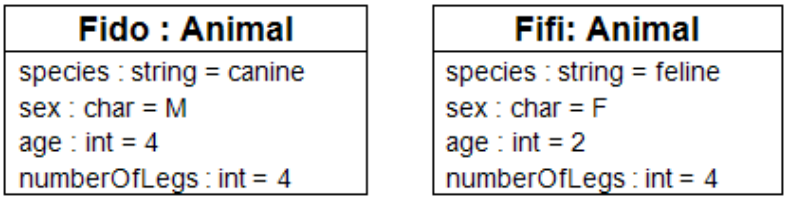


Top Segment - Class Name

First, examine the top segment of the Animal class diagram. It contains only a single word, "Animal". This is the name of our class which is also the first component required in a UML class diagram! Every class name should be unique and it should appropriately describe the objects it will create. In this case, any programmer can quickly deduce that the class diagram above would result in objects that represent animals. This could include dogs, cats, kangaroos, ferrets, cows, and perhaps even a human!

Middle Segment - Class Attributes

Next, take a look at the middle segment of the Animal class diagram. This segment contains four attributes or characteristics that are common to all animals and assist in further describing each individual object. According to the class diagram, we want to track species, sex, age, and the number of legs associated with each animal in our system. Keep in mind that every Animal object we create will contain all four of these data attributes. As an example, let's consider two Animal objects: a dog named Fido and a cat named Fifi.



As you can see, both Fido and Fifi have a value associated with their species, sex, age, and numberOfLegs attributes. However, the values stored in each object describe only that object. Fido is a 4-year-old male dog with 4 legs. Fifi, on the other hand, is a 2-year-old female cat with 4 legs. We can create thousands of Animal objects, each of which will have their own unique set of Animal attributes.

Bottom Segment - Class Methods

Finally, we need to discuss the last segment of our class diagram which contains class methods. Naturally, we want

to be able to both fill the attributes of our class objects with data as well as extract data from those same attributes. In order to do this, we use "set" and "get" methods, often referred to as setters/mutators and getters/accessors. Setters are methods designed to fill (or set) the attributes with data values. Conversely, getters are methods used to extract (or get) data from the attributes. Notice that the names of the attributes follow the "get" and "set" prefixes. For example, the setter for our species attribute is called setSpecies and the getter is called getSpecies. We use getters and setters to control access to the attributes. For example, setAge() could make sure that the program never tries to assign a negative number to age. After all, an animal can never be less than zero years old!

Further review of the UML diagram shows that we also want a method to describe how an animal eats and another to allow the animal to sleep. As you might imagine, every animal is going to eat and sleep at some point so we need to implement these behaviors/methods appropriately.

Another important class method that we cannot forget is the constructor. A constructor is called when an object is created and is typically used to initialize all of the object's attributes. The name of the constructor is always the same as the name of the class, and it can never return a value.

Public and Private Access - "+" and "-"

Notice that the attributes listed in the UML diagram begin with a "-" or a "+." A "-" means that the item is private and no external program, object, or other entity can access it directly. In other words, the following line of code located in any entity outside our object would be invalid:

```
age = 14;
```

We usually make attributes private in order to protect them from being modified without our express permission. So, if external entities cannot modify the value of our attributes, how do the values of our attributes ever change? You may have already guessed that a setter, defined inside the same class as its associated attribute, will do the trick. For example, setAge is preceded with a "+" in our Animal class diagram which means that it is public. Public attributes and methods can be directly accessed by external programs, objects, and other entities. Therefore, external entities can suggest a new value for our age attribute by passing it to setAge. setAge is then responsible for determining if the suggested value is valid and if it is, accessing the age attribute directly. As you can see, getters and setters are always public. In fact, many but not all methods are declared public.

C++ Syntax on Creating Classes and Objects

Back to Top

Class is the template from which objects are built. Here is an example of class Rectangle that declares data members' width, height, and four member-function prototypes.

```
class Rectangle
{
private:

int width, height;

public:

void enterDimension();
bool compareArea(Rectangle& r);
void scaleUp(int percent);
void printRectangle();
};
```

Function prototypes are defined within the class scope. When the member functions are implemented outside of the class, we need to indicate the name of the class to which these member functions belong. For example:

```
void Rectangle::enterDimension()
{
  cout << "Enter both dimensions for the rectangle ";
  cin >> width >> height;

  if (width < 0)
      width = 0;
  if (height < 0)
      height = 0;

}
bool Rectangle::compareArea(Rectangle& r)
{
  if (width*height > r.width*r.height)
    return true;
  else
    return false;
}
```

Click here for transcript of this image.

Below is the complete code.

Click on image to enlarge.

```
1  #include <iostream>
2  using namespace std;
3  class Rectangle
4  {
5  private:
6       int width, height;
7  public:
8       void enterDimension();
9       bool compareArea(Rectangle& r);
10      void scaleUp(int percent);
11      void printRectangle();
12 };
13 void Rectangle::enterDimension()
14 {
15 cout << "Enter both dimensions for the rectangle ";
16 cin >> width >> height;
17 if (width < 0)
18     width = 0;
19 if (height < 0)
20     height = 0;
21 }
22 bool Rectangle::compareArea(Rectangle& r)
23 {
24 if (width*height > r.width*r.height)
25     return true;
26 else
27     return false;
28 }
29
```

Lines 1-29

```
30 void Rectangle::scaleUp(int percent)
31 {
32 width = width*percent/100;
33 height = height*percent/100;
34 }
35 void Rectangle::printRectangle()
36 {
37 cout << "The new dimension of the second rectangle is\n";
38 cout << "width= " <<width << " height= " <<height <<endl;
39 }
40 int main()
41 {
42 Rectangle r1, r2;
43 r1.enterDimension();
44 r2.enterDimension();
45 if (r1.compareArea(r2))
46 {
47 r2.scaleUp(200);
48 r2.printRectangle();
49 }
50 else
51 {
52 cout << " There is no change in the dimensions of the second rectangle\n ";
53 }
54 return 0;
55 }
56
```

Lines 30-56

Click here for transcript of this program.

Program Sample Output:

```
Enter both dimensions for the rectangle 5 10
Enter both dimensions for the rectangle 2 4
The new dimension of the second rectangle is
width= 4 height= 8
Press any key to continue . . .
```

Implementing Encapsulation and Information/Data Hiding

Back to Top

Let's take a quick step back, now that we understand how to create and manipulate an object from our C++ class. Do you recall the concept of encapsulation that we described at the beginning of the lecture? We said that encapsulation is a concept in which each object we create contains features and functions that are logically related to the services that class is providing to the program. In this case, our dog object contains attributes (species, sex, age, and numberOfLegs) and methods (getters, setters, sleep, and eat) that are logically related and well-defined in our Animal class. Our Animal class allows other programs to use the concept of an animal in their own logic without implementing specific animal traits and behaviors. We have essentially created a concise, standalone unit that takes input, manipulates that input, and returns a result much like the black box below. External entities do not need to know the logic designed into our Animal class (black box) in order to create and use an Animal object. Instead, they only need to be aware of the Animal class' interface or list of publically available methods. We have successfully encapsulated the characteristics and behaviors of animals into an easy-to-use class!

It is important to note that encapsulation protects our data fields, but it is not all about data protection; encapsulation is also about providing independence for program components. When encapsulation is used, if the names or the data types of attributes are to be changed, only the getter and setter functions need to be changed. All other parts of the program remain unaffected. This is a real advantage of using encapsulation. External entities that utilize encapsulated code avoid data dependencies by not using the names of data fields directly.

Earlier in the lecture we also briefly discussed the concept of information/data hiding. Manufacturers of disc players don't want their customers to have direct access to the internal circuitry of their products just as we don't want external entities to have direct access to the private attributes and methods in our Animal class. In order to implement information/data hiding, we provide external components with the absolute minimum amount of

information and interaction necessary to get the job done. The internal details of our class' processing are kept hidden. This makes our programs more reliable, robust, and easier to maintain.

We implemented information/data hiding in our Animal class by declaring the attributes as private and implementing public getters and setters. In this way, we restrict access to or hide our attributes from external entities. Also, by declaring methods as public only when it is necessary for external objects to invoke a class's operations, we restrict access to internal methods that may contain important operations that could leave our data in an erroneous state if invoked incorrectly. For example, we might create a private method called findFood that would be **hidden** from external entities and would only be used by the public eat method as necessary.

Summary

Back to Top

In summary, encapsulation and information/data hiding work hand-in-hand to not only increase the quality of our code but also to provide external entities with easy and secure access to potentially complicated code. Encapsulation allows the implementation of a class to change without affecting external entities using that class. In addition, critical class attributes and methods are hidden or protected from accidental and/or malicious changes, thus promoting data integrity. By making our attributes and select methods private and by providing external entities with a clearly defined interface, we can use encapsulation and information/data hiding to our advantage and create reliable, high-quality and reusable code!

Back to Top