C# PROGRAMMING FOUNDATIONS:

Composition and Inheritance

© August 2011 by
ERIC MATTHEWS

Copyright

C# Foundations by Eric Matthews is licensed under a Creative Commons Attribution 3.0 Unsupported License. This license allows you to

- Copy, distribute and transmit the work
- Adapt the work
- Make commercial use of the work

Under the following conditions:

You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work

With the understanding that:

Waiver — Any of the above conditions can be waived if you get permission from the copyright holder.

Public Domain — Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.

Other Rights — In no way are any of the following rights affected by the license:

- Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
- The author's moral rights;
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice — For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to one or more of the following web pages.

www.DeveloperGeekResources.com

www.EducationAnytime.com

Contents

Copyright **2**

Inheritance and composition ("kind of inheritance") *4*

A new frontier 4

BUILDING BUSINESS objects using composition 4

Organizing Multiple Classes in a Project for Reuse **8**

Is Composition Inheritance? **9**

Inheritance (the real meal deal) 10

Upcasting **11**

IS-A and HAS-A Relationships 12

HAS A **13**

IS A **14**

Put into practice – Final Class Exercise 15

Output Examples **15**

Code Examples **16**

INHERITANCE AND COMPOSITION ("KIND OF INHERITANCE")

A NEW FRONTIER

Hopefully you are at this stage and starting to get more comfortable with the subject matter. We are now going to explore the concept of inheritance. This section is going to introduce some new ideas and principles that may be very different than what you are accustomed to.

To compare this to the Martial Arts, you have just received your green belt and are starting to feel pretty good about coding in C#. Here we are going to show you a new set of moves for your next belt. Take this slowly as this subject is one of the cornerstones of oop.

BUILDING BUSINESS OBJECTS USING COMPOSITION

Let's discuss what appear to be four simple entities: Phone, Person, Address and Business. Anyone reading this will have a pretty reasonable understanding of each of these entities. You also are likely to have a reasonable understanding of how these entities relate to one another.

In a pure sense let's look at the minimum data requirements for each entity:

Phone
- Area code
- Exchange
- Number

Person
- Last name
- First name
- Middle name
- Birth date

Address
- Street
- City
- State
- Zip

Business
- Name

We can say 206-555-5555 is a phone number. Great, care to make the call? Whose phone number is it? A person? A business? A person can have many phone numbers, work, home, pager, cell phone. A business has many phones and phone numbers. Addresses belong to people and they also belong to businesses that contain people conducting business. People can have more than one job. Businesses can have more than one location. I could go on and on in quantifying these relationships as they reflect the "real world". In reality, these relationships get quite complex and difficult to quantify and capture in a technological form.

The term “object” is thrown around a lot, but in reality it is hard to achieve containment for an object (or entity as I have been referring to it) because seldom do objects stand on their own two feet. In practical reality most objects need to form a relationship with other objects to have meaning. Having a phone number without any other association is not all that practical.

What ends up happening is entities get created that are really combinations in part or in whole of many entities. This results in a fragmentation of both data and code. Even worse, the idea of discrete objects (phone, address, and invoice) gets lost.

With a little encapsulation and then a little inheritance I can contain an object at a more discrete and granular level, yet allow it to have association to other objects in both a practical and a flexible manner when developing business and other types of application. While on this subject I also wish to point out that you must have an understanding about the application you are developing. In other words, if you are developing and accounts payable application you have better know something about accounts payable or be willing to quickly learn. Some programmers and their management feel this is not important. NOTHING could be farther from the truth!

For these examples, I will not be coding all the methods that you would want to associate with a particular object. You of course could and would want to do this when implementing an object. And, in practical application of these principles you will still run up against the limitations of the DBMS that you are using to store your data. Actually limitations may have been a poor choice of words. Somewhere in your application, your OOP code will need to access data from a world that is somewhat relational in nature. In other words, the design and implementation of the database will be a determining factor on how your code gets written.

Again, I wish to emphasize that my goal is that you understand and can apply the concept of inheritance to the practical issues you have to code.

Let me demonstrate a small example where we use inheritance to create a business object that contains a person, their phone number and address. We will start with building the phone object.

Source Phone.cs

```
using System;

namespace Entities
{
  public class Phone {
      public String area_cde;
      public String exchange;
      public String extension;

      public string phone_nbr_us() {
        String phn = "";
        phn = (area_cde + "-" + exchange + "-" + extension);
        return phn;
      }

      public string phone_nbr_europe() {
        String phn = "";
        phn = (area_cde + "." + exchange + "." + extension);
```

```
        return phn;
      }
 } //end phone class
} //end entity namespace
```

Next we need to build a person class

Source Person.cs

```
using System;

namespace Entities
{
  public class Person
  {
         public String last="";
         public String first="";
         public String middle="";

         public string fullname_fml()
         {
           String Person = "";
           if (middle == "") {
                Person = (first + " " + last);
                return Person;
           }
           else {
                Person = (first + " " + middle + " " + last);
                return Person;
           }
          }

         public string fullname_lmf()
         {
            String Person = "";
            if (middle == "") {
               Person = (last + " " + first);
               return Person;
          }
          else {
              Person = (last + " " + middle + " " + first);
              return Person;
          }
        }

  } //end phone class
} //end entity namespace
```

Finally we need to build an address class.

Source Address.cs

```
using System;

namespace Entities
{
```

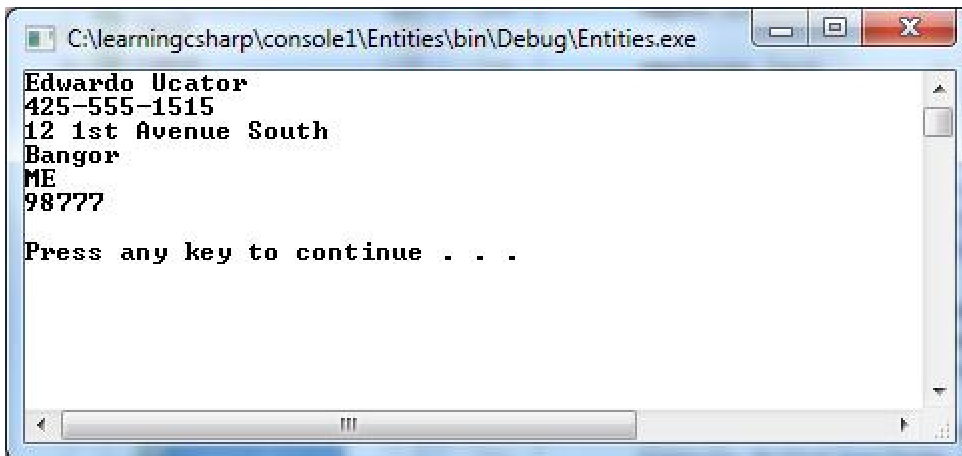
```
  public class Address
  {
      public String street="";
      public String city="";
      public String state="";
      public String zip="";

      public string return_formatted_us_address()
      {
        String address = "";
        address += street + "\n";
        address += city + "\n";
        address += state + "\n";
        address += zip + "\n";

        return address;
      }
  } //end phone class
} //end entity namespace
```

Now that we have all the classes we want to use we can create the business object.

Source Main_fcn.cs

```
using System;

namespace Entities
{

  public class Main_fcn {

    public static void Main(string[] args) {

      // to hold returned data
      String home_phone = "";
      String home_address = "";
      String employee_name = "";

      // instance our three objects
      Phone hmphn = new Phone();
      Address hmaddr = new Address();
      Person employee = new Person();

      //set data values, normally this would be happen
      //as result of database access, or entry from ui

      //phone data
      hmphn.area_cde = "425";
      hmphn.exchange = "555";
      hmphn.extension = "1515";

      //person data
      employee.last         = "Ucator";
      employee.first        = "Edwardo";

      //address data
```

```
      hmaddr.street       = "12 1st Avenue South";
      hmaddr.city         = "Bangor";
      hmaddr.state        = "ME";
      hmaddr.zip          = "98777";

      //use desired methods
      home_phone = hmphn.phone_nbr_us();
      employee_name = employee.fullname_fml();
      home_address = hmaddr.return_formatted_us_address();

      //display results
      Console.WriteLine(employee_name);
      Console.WriteLine(home_phone);
      Console.WriteLine(home_address);

      Console.Write("Press any key to continue . . . ");
      Console.ReadKey(true);
    }

  }
}
```



Organizing Multiple Classes in a Project for Reuse

There are a few items we need to cover regarding the code you are looking at. It is not so much related to the subject of composition, rather it is related to how to organize your classes into an effective means for reuse.

Up until now with a few exceptions we have had a single class for each project, and we have predominantly not worked with namespace. Here are some important points you need to now understand.

- A project can contain multiple .cs files.
- A .cs file can contain multiple classes.
- Multiple classes can live within a namespace
- A namespace is a logical grouping for classes

- A C# console project can only contain a single entry point (Main method)

If you tie these bullets back to the code you have been exposed to you it should begin to make some sense. If you are looking for the specifics on how to create multiple classes in a project refer to the document “C#-UsingSharpDevelop-QuickGuide.doc.” This subject can be found in the table of contents of that document.

Is Composition Inheritance?

I named the program “Composition” because this is the term used to denote a type of inheritance. Please note that technically “Composition” is a form of reuse but is not technically considered to be inheritance. I personally believe that “composition” is a restrictive form of inheritance. Now here is my rationale and feel free to disagree with me. Inheritance is a process through which an object can acquire the properties and methods of another object. Inheritance also allows for adding and changing the methods and properties related to the object acquired. An object can add methods and properties to a method it has acquired. These methods and properties would now be available only to that object and any other object that inherited it or called it using composition. Well, semantically speaking I can add functionality to an object whether or not I use true inheritance or not. Granted I may not be able to preserve true hierarchical classification using “composition”. But I still can none-the-less extend the functionality of an object using “composition”. I also believe that the definition of inheritance in oop is too narrow and should be more in line with the real dictionary definition of the word, which simply stated is “to acquire something”.

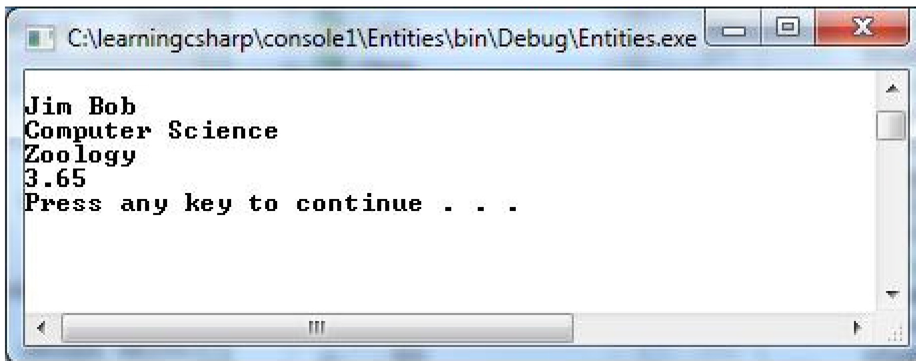
When you are done with this chapter I am more concerned that you have a practical understanding of “composition” and “inheritance” and how these techniques can be practically applied to solve real problems.

INHERITANCE (THE REAL MEAL DEAL)

This is a very important aspect of oop. The ability to not only reuse a class, but to extend it is a concept that requires a great deal of architectural consideration. We need to look at a trivial example in terms of code, but we want to look at an example that is practical and really demonstrates the power of this concept.

Think about the person class we just covered. It is a good start to begin to quantify a human being. But what if we were building an application for an academic institution and we needed to quantify students and teachers. Well, both are people (hey, no jokes). But both have attributes and needs that are different from one another. This is where inheritance can play a key role is letting us have our proverbial cake and eat it too.

Source Student.cs

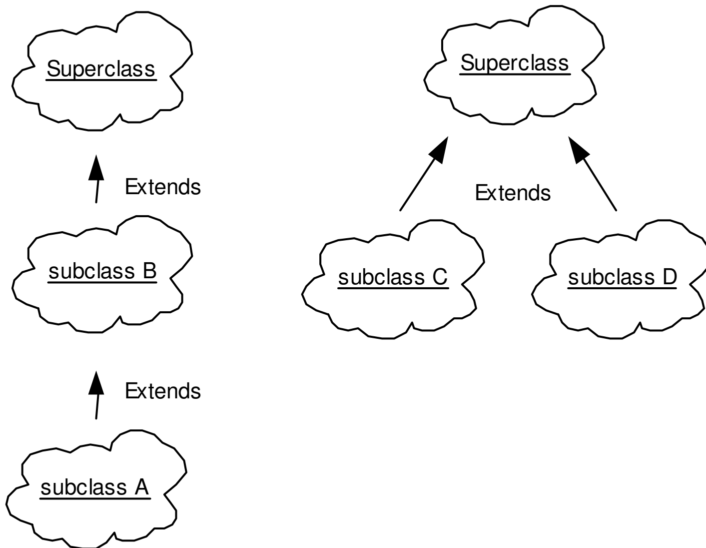



```
using System;

namespace Entities
{
  public class Student : Person
  {
    public String major="";
    public String minor="";
    public double gpa=0;

    public double GPA()
    {
      return gpa;
    }

    public string getMajor()
    {
      return major;
    }

    public string getMinor()
    {
      return minor;
    }
```

```
 } // end student class
} //end entity workspace
```

As you can see we can create a new student class that adds the methods and attributes that are unique to a student, while inheriting the general attributes and methods of the person class. Inheritance is achieved by using the **":"** operator. In our example the person class is referred to as the **superclass** and the student class is referred to as the **subclass**.

Scenarios for Superclass/Subclass

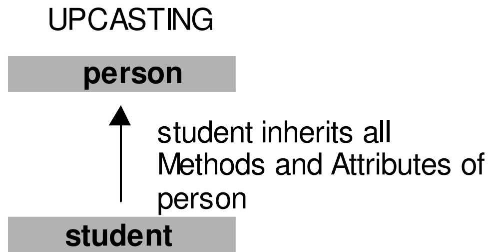


As you can see we were able to extend the Student class by inheriting the Person class. In doing so we get to use all of the methods in the Person class, but also get to add methods that are specific to students.

I hope your light bulb has come on in analyzing these examples and their application to the ideas I have presented. Inheritance and composition are not for every programming task. However, where hierarchical classification and re-classification is obvious then it should not only be given consideration, but should be your first choice.

Upcasting

We have actually been covering this concept since beginning the subject of inheritance. I made this a separate subject, as I wanted to point out that inheritance is really about ancestry, or derivation. One can easily get hung-up and think that the "person" class spawns the student class (reverse the arrow in the picture below). This however is not the case. The fact is that "person" exists as an independent piece of code that may or may not (in this case it does) stand on its own. Other classes we create can derive or inherit this code.



For me it is an incredibly subtle but major point that inheritance be used in situations where the base class (or superclass as it is termed) can meet some of our needs but not all of them. I think the person, and student classes though trivial in robustness illustrates this point very well. If we ponder the semantics of the relationship between a student entity and a person entity we can conclude the following:

- A student can be a person
- A person is not necessarily a student
- A student entity will have attributes that are different from a person entity
- A student entity will have attributes that are the same as the person entity

Given this assessment an upcasted solution of student to person makes the most sense as the student entity has a dependency on the person entity, though the person entity exists without dependency to student.

C#'s solution for inheritance is one that is hierarchical in nature. The tree in singular, and the implementation is such that object in the upper part of the hierarchy do not have a dependency on the objects below them.

Another incredibly subtle point of inheritance is that when you inherit something you are really doing more than just instantiating the object. You are actually creating a **type** of that object that you can extend.

This can be a pretty abstract and mind-blowing concept. It is also a concept that can incur the wrath of opponents to oop that considers such an idea something that can create convoluted code. I have heard the arguments from both sides on this subject and both sides offer compelling points. While I agree this is a very difficult concept to both understand and follow in code, I do not believe it is convoluted if properly thought out and implemented. In fact I believe that used properly and with a great deal of thought that this is as great an invention as pizza!

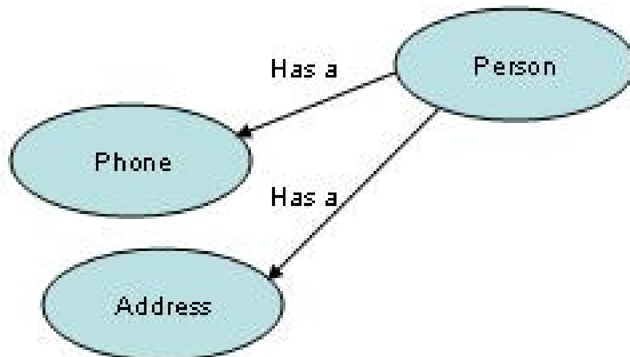
IS-A AND HAS-A RELATIONSHIPS

As we conclude this introduction to composition and inheritance I want to tie the two techniques together for you. Developing world class applications in C# involve understanding how to build your objects. I have seen far too many oop zealots go absolutely off the rails on this. I have also seen many structured programmer learn oop and continue to write structured programs, not taking advantage of all oop has to offer. Both sides are not hitting the sweet spot which is somewhere in the middle.

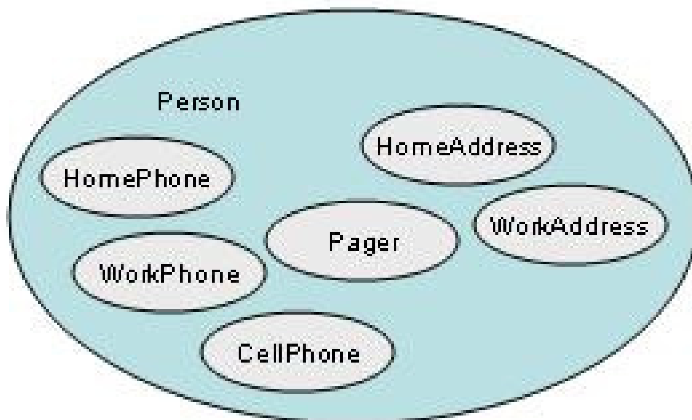
Good world class oop development is understanding how and when to use what. In the absence of good architectural leadership (and there does exist a big void here) you are left to your own devices. My point is that planning and design is just as important as coding when it comes to oop.

HAS A

I think at the arial view we can agree that a Person can have a phone, and that a Person can have a Phone. The phone and address objects are compositional entities to the person entity.



We can also further decide that in defining the notion of a person entity that there is a need to define these entity relationships to a more atomic level as described below. There are many sticky issues we can discuss that center around how atomic we get in defining these relationships, but for now let's just agree that the following picture is what we want to roll with.



Even if we cannot agree, since person, phone, and address are discrete entities it is not a big deal to implement another object or either type phone or address in person. In fact, it is a whole line of code.

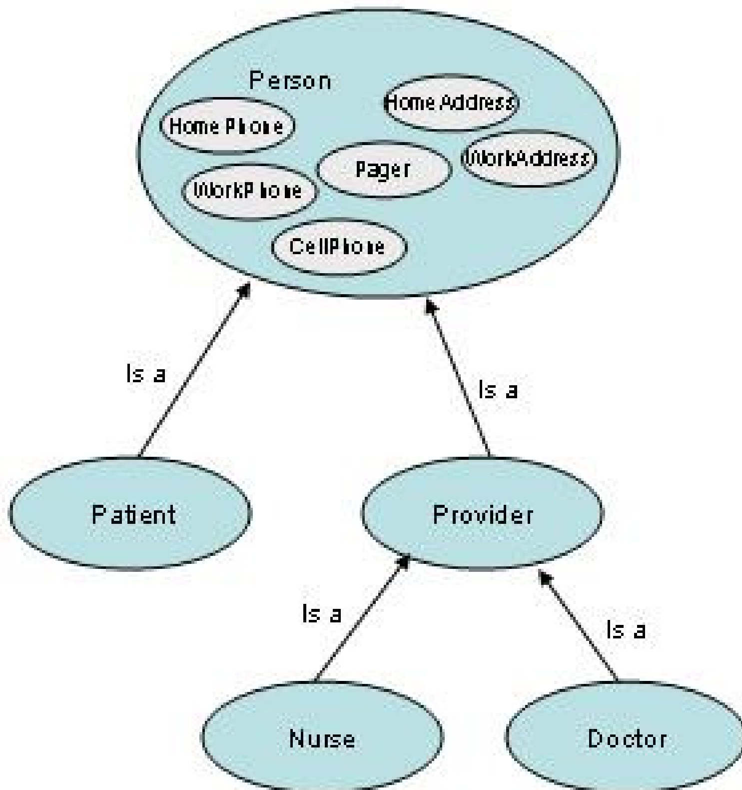
Address SummerHome = new Address()

If we ponder this idea in terms of reuse it does seem to be pretty cosmic.

IS A

I work in the healthcare field so I am going to implement my person class for that domain. You should have no problem applying this "object construction logic" to your business domain.

Consider the following drawing.



Our goal is to define, create, and reuse a person entity. Again the central high level idea is to maximize reuse while segregating functionality that is entity unique. For example…"prescribe medication", which is a function that in our picture only a Doctor is allowed to do, whereas "Height" is a function/attribute that is generic to any entity that is a person. You certainly get my drift.

We can simplify an entities relationship to another entity in terms of "is a" and "has a" relationships. Being a simpleton myself, I love simplified systems. In terms of OOP terminology "is a" relationships are ones of inheritance…"has a" relationships are ones of composition.

PUT INTO PRACTICE – FINAL CLASS EXERCISE

In this exercise you are to create a Teacher class. The teacher class should inherit the Person class. In the Teacher class add the following methods:

```
public string getyearsOfService()

public string setYearsOfService(String service_years)

public string setTenure(String tenure_flg)

public string getTenure()
```

The setTenure() method needs to be coded to only accept a “Y” or a “N.” You are to make this case insensitive.

The setYearsOfService() method needs to be coded to only accept numbers, with the maximum number being 60. Note: the consumer of this method will be entering data as a String (even though a number is required)

Both set methods should be written to return the status (see example output below).

Additionally the set methods should populate variables that are marked as private (see code example below).

The two get methods will return the data that was collected by the set methods.

Add to the Main_fcn class to use and test your Teacher class.

Output Examples

```
setTenure-ok
y
setYearsOfService-ok
12
```

Or, if incorrect data is entered, output looks like…

Incorrect value set for setTenure() method

```
setTenure-rejected-invalid value supplied
```

Non numeric value set for setYearsOfService() method

```
setYearsOfService-rejected-invalid value supplied
```

Out of range number set for setYearsOfService() method

```
setYearsOfService-rejected-number of years exceeds allowable limit
```

Code Examples

The following code snippet shows how to create a test to see if a value is numeric.

```
using System;
class testNumeric
{

    public int mynum = 10;
    public String input_num = "";
    public bool canConvert;

    public static void Main(string[] args)
      {

      testNumeric t1 = new testNumeric();
      t1.input_num = "35";
      t1.canConvert = int.TryParse(t1.input_num, out t1.mynum);
      if (t1.canConvert == true) {
                Console.WriteLine("Number entered was: " + t1.input_num);

      } else {
            Console.WriteLine("A non-numeric value was entered");
      }

      testNumeric t2 = new testNumeric();
      t2.input_num = "A12";
      t2.canConvert = int.TryParse(t2.input_num, out t2.mynum);
      if (t2.canConvert == true) {
                Console.WriteLine("Number entered was: " + t2.input_num);

      } else {
            Console.WriteLine("A non-numeric value was entered");
      }

            Console.Write("Press any key to continue . . . ");
            Console.ReadKey(true);
    }

}
```

The following code example demonstrates how to use entities marked as private.

```
using System;
class privateExample
{
```

```
    private String myPrivateString = "";

    public void setMyPrivateString(string inp) {
        this.myPrivateString = inp;
    }

    public void getMyPrivateString() {
        Console.WriteLine(myPrivateString);
    }


        public static void Main(string[] args)
        {
              privateExample pE = new privateExample();
              pE.setMyPrivateString("hello");
              pE.getMyPrivateString();


              Console.Write("Press any key to continue . . . ");
              Console.ReadKey(true);
        }
}
```