The Master Parent Class: System.Object

To wrap up , I’d like to examine the details of the master parent class in the .NET platform:

Object. As you were reading the previous section, you might have noticed that the base classes in your hierarchies (Car, Shape, Employee) never explicitly specify their parent classes:

// Who is the parent of Car?

class Car

{...}

In the .NET universe, every type ultimately derives from a base class named System.Object (which can be represented by the C# object keyword [lowercase “o”]). The Object class defines a set of common members for every type in the framework. In fact, when you do build a class that does not explicitly define its parent, the compiler automatically derives your type from Object.

If you want to be very clear in your intentions, you are free to define classes that derive from Object as follows:

// Here we are explicitly deriving from System.Object.

class Car : object

{...}

Like any class, System.Object defines a set of members. In the following formal C# definition, note that some of these items are declared virtual, which specifies that a given member may be overridden by a subclass, while others are marked with static (and are therefore called at the class level):

public class Object

{

// Virtual members.

public virtual bool Equals(object obj);

protected virtual void Finalize();

public virtual int GetHashCode();

public virtual string ToString();

// Instance-level, nonvirtual members.

public Type GetType();

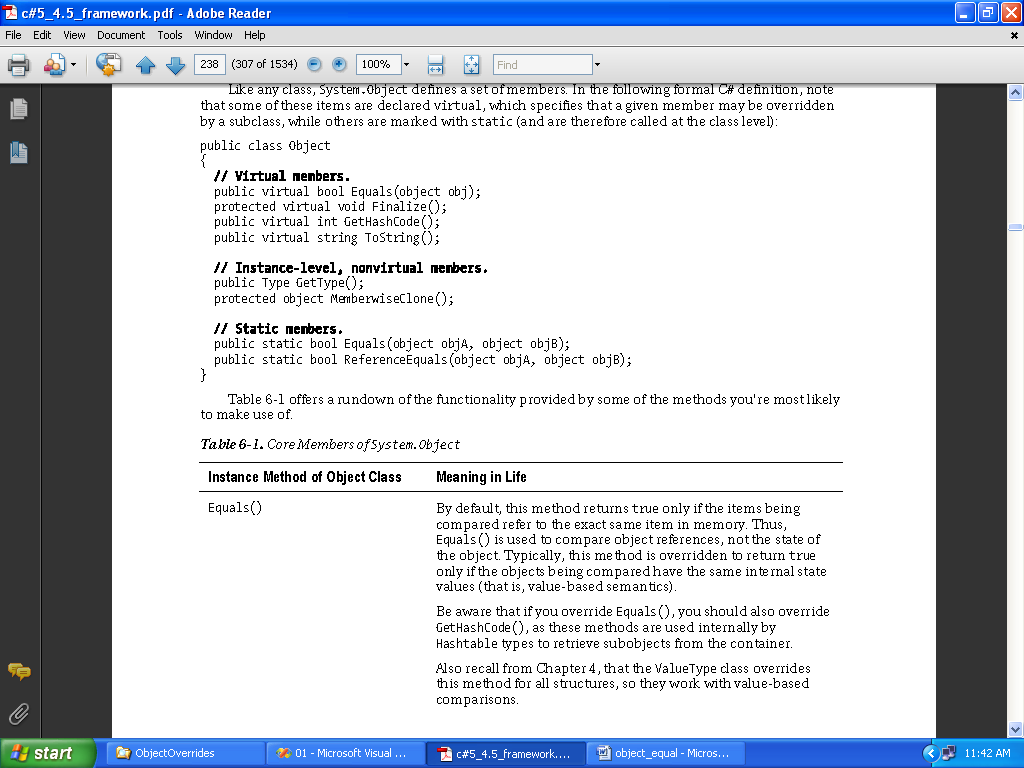
protected object MemberwiseClone();

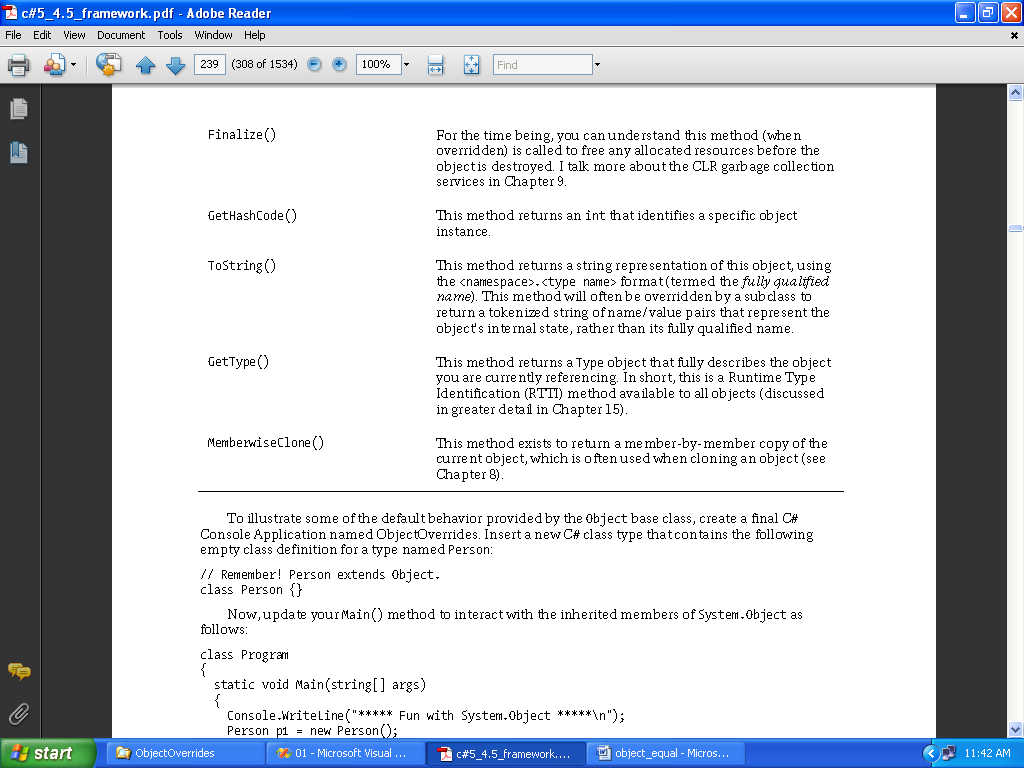
// Static members.

public static bool Equals(object objA, object objB);

public static bool ReferenceEquals(object objA, object objB);

}





Overriding System.Object.Equals()

Let’s also override the behavior of Object.Equals() to work with value-based semantics. Recall that by default, Equals() returns true only if the two objects being compared reference the same object instance in memory. For the Person class, it may be helpful to implement Equals() to return true if the two variables being compared contain the same state values (e.g., first name, last name, and age).

First of all, notice that the incoming argument of the Equals() method is a general System. Object. Given this, your first order of business is to ensure the caller has indeed passed in a Person object, and a s an extra safeguard, to make sure the incoming parameter is not a null reference.

After you have established the caller has passed you an allocated Person, one approach to

implement Equals() is to perform a field-by-field comparison against the data of the incoming object to the data of the current object:

public override bool Equals(object obj)

{

if (obj is Person && obj != null)

{

Person temp;

temp = (Person)obj;

if (temp.FirstName == this.FirstName

&& temp.LastName == this.LastName

&& temp.Age == this.Age)

{

return true;

}

else

{

return false;

}

}

return false;

}

Here, you are examining the values of the incoming object against the values of your internal values (note the use of the this keyword). If the name and age of each are identical, you have two objects with the exact same state data and, therefore, return true. Any other possibility results in returning false. While this approach does indeed work, you can certainly imagine how labor intensive it would be to implement a custom Equals() method for nontrivial types that may contain dozens of data fields. One common shortcut is to leverage your own implementation of ToString().

If a class has a prim-andproper implementation of ToString() that accounts for all field data up the chain of inheritance, you can simply perform a comparison of the object’s string data:

public override bool Equals(object obj)

{

// No need to cast "obj" to a Person anymore,

// as everything has a ToString() method.

return obj.ToString() == this.ToString();

}

Notice in this case that we no longer need to check whether the incoming argument is of the correct type (a Person, in our example), as everything in .NET supports a ToString() method. Even better, we no longer need to perform a property-by-property equality check, as we are not simply testing the value returned from ToString().

Overriding System.Object.GetHashCode()

When a class overrides the Equals() method, you should also override the default implementation of GetHashCode(). Simply put, a *hash code* is a numerical value that represents an object as a particular state. For example, if you create two string variables that hold the value Hello, you would obtain the same hash code. However, if one of the string objects were in all lowercase (hello), you would obtain

different hash codes.

By default, System.Object.GetHashCode() uses your object’s current location in memory to yield the

hash value.

However, if you are building a custom type that you intend to store in a Hashtable type

(within the System.Collections namespace), you should always override this member, as the Hashtable will be internally invoking Equals() and GetHashCode() to retrieve the correct object.