C# Language

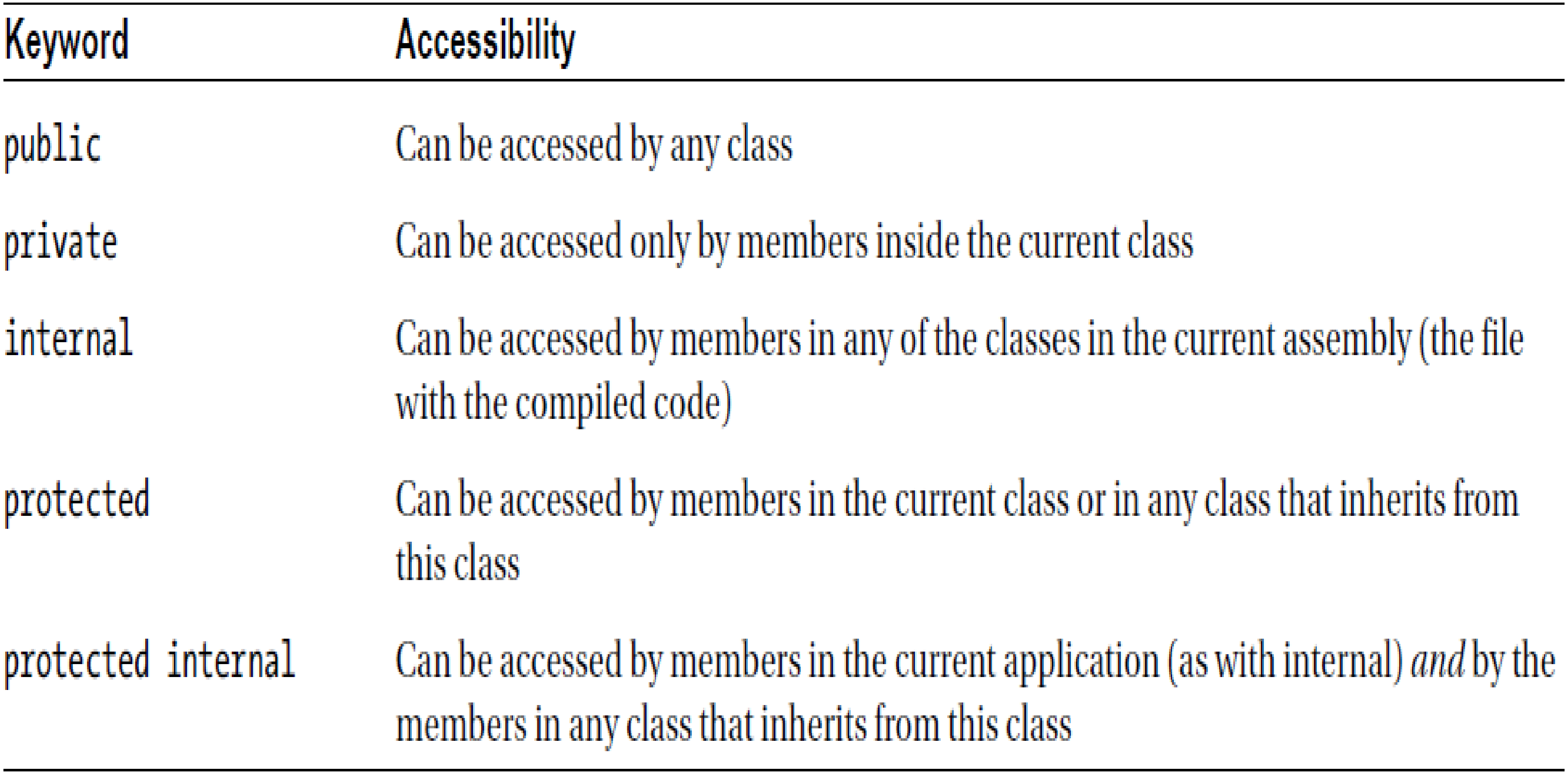
Today You will learn

- Building a basic class
- Value Types and Reference Types
- Understanding Namespaces and Assemblies
- Advanced Class Programming

Building a basic class

- Once you've defined the basic skeleton for your class, the next step is to add some basic data members.
- When you declare a member variable, you set its *accessibility*.
- The accessibility determines whether other parts of your code will be able to read and alter this variable.

Building a basic class



Keyword	Accessibility
`public`	Can be accessed by any class
`private`	Can be accessed only by members inside the current class
`internal`	Can be accessed by members in any of the classes in the current assembly (the file with the compiled code)
`protected`	Can be accessed by members in the current class or in any class that inherits from this class
`protected internal`	Can be accessed by members in the current application (as with internal) *and* by the members in any class that inherits from this class

Creating an Object

- When creating an object, you need to specify the **New** keyword.

```
 Product saleProduct = new Product();
// Optionally you could do this in two steps:
 Product saleProduct;
saleProduct = new Product();
```

- If you **omit the New** keyword, you'll declare the variable, but you won't create the object. Here's an example:

```
Product saleProduct;
```

- In this case, your saleProduct variable doesn't point to any object at all. If you try to use the saleProduct variable, you'll receive the common "null reference" error.

Creating an Object

- In some cases, you will want to declare an object variable without actually creating an object.

```
//Declare but don't create the product.
Product saleProduct;
```

- You Call a function that accepts a numeric product ID parameter and returns a Product object.

```
//Assign the Product object to the saleProduct variable.
saleProduct = FetchProduct(23);
```

- You can compress this code into one statement:

```
Product saleProduct = FetchProduct(23);
```

Adding Properties

- You can manipulate Product objects in a safe way. You can do this by adding ***Property Accessors***.
- **Accessors** usually have **two** parts.
  - **Get accessor** **-** Allows your code to retrieve data from the object.
  - **Set accessor** **-** Allows your code to set the object's data.
- In some cases, you might omit one of these parts, such as when you want to create a property that can be examined but not modified.
- **Accessors** are similar to any other type of method in that you can write as much code as you need.

Adding Properties

- Here’s a revised version of the Product class that renames its private member variables and adds public properties to provide access to them:

```
public class Product
    {
        private string name;
        private decimal price;
        private string imageUrl;

        public string Name
        {
            get
            {
                return name;
            }
            set
            {
                name = value;
            }
        }
    }
```

Adding Properties

- **Property accessors should start with an initial capital**.
- Usually, the private variable will have a similar name, but **begin with lowercase** or **prefixed with m_** (which means "member variable").
- The client can now create and configure an instance of the class by using its **properties and the familiar dot syntax**.

Adding Properties

- For example, if the object variable is named **saleProduct**, you can set the product name using the saleProduct.Name property.

```
Product saleProduct = new Product();
saleProduct.Name = "Kitchen Garbage";
```

Adding Properties

- You can create properties that can be read but not set (which are called **read-only** properties), and you can create properties that can be set but not retrieved (called **write-only**).
- All you need to do is leave out the accessor that you don’t need.

```
public decimal Price
{
    get
    {
        return price;
    }
}
```

Automatic Properties

- **Automatic properties** are properties without any code.
- When you use an automatic property, you declare it, but you don’t supply the code for the get and set accessors, and you don’t declare the matching private variable.
- Instead, the C# compiler adds these details for you.
- Because the properties in the Product class simply get and set member variables, you can replace any of them (or all of them) with automatic properties.

Automatic Properties

```
public decimal Price { get; set; }
```

- You don’t actually know what name the C# compiler will choose. However, it doesn’t matter, because you’ll never need to access the private member variable directly.

- Instead, you’ll always use the public Price property.

- The only disadvantage to automatic properties is that you’ll need to switch them back to normal properties if you want to add some more specialized code after the fact.

Adding a Method

- **Methods** are simply **procedures or functions** that are built into your class.
- When you call a method on an object, the method does something useful, such as return some calculated data.

```
public class Product
 {
    //(Additional class code omitted for clarity.)
    public string GetHtml()
    {
        string htmlString = "<h1>" + Name + "</h1><br />"
            + "<h3>Costs: " + Price.ToString() + "</h3><br />"
            + "<img src='" + ImageUrl + "' />";
        return htmlString;
    }
 }
```

Adding a Constructor

- A **constructor** is a method that automatically runs when an instance is created.
- In C#, the constructor always has the same name as the name of the class.

```
public class Product
{
     //(Additional class code omitted for clarity.)
    public Product(string name, decimal price)
    {
         Name = name;
         Price = price;
    }
}
```

Adding a Constructor

- Now that you have a constructor, you can use it to create a new object instance.

```
Product saleProduct = new Product("Kitchen Garbage", 9.99m);
```

- If you don’t create a constructor, .NET supplies a **default public constructor** that does nothing.
- If you create at least one constructor, .NET will not supply a default constructor.

```
`This will not be allowed, because there is  no zero-argument constructor.
Product saleProduct = new Product();
```

Adding a Constructor

- As with ordinary methods, **constructors can be overloaded** with multiple versions, each providing a different set of parameters.
- When creating an object, you can choose the constructor that suits you best based on the information that you have available.

```
public class Product
    {
        //(Additional class code omitted for clarity.)
        public Product(string name, decimal price)
        {
            Name = name;
            Price = price;
        }

        public Product(string name, decimal price, string imageUrl)
        {
            Name = name;
            Price = price;
            ImageUrl = imageUrl;
        }
}
```

Adding an Event

- Classes can use the events to allow one object to notify another object that's an instance of a different class.
- As an illustration, the **Product class** example has been enhanced with a **PriceChanged event** that occurs whenever the price is modified through the property procedure.
- This event **won't fire** if code inside the class changes the underlying private price variable without going through the property.

Adding an Event

```
//Define a delegate that represents the event.
public delegate void PriceChangedEventHandler();
public class Product
{
    // Define the event using the delegate.
    public event PriceChangedEventHandler PriceChanged;
    public decimal Price
    {
        get { return  price; }
        set
        {  price = value;
            //Fire the event , provided there is at least one listener.
             if(PriceChanged != null)
              {      PriceChanged();      }
         }
     }
}
```

Handling an Event

- To **handle an event**, you first create a method called an ***event handler***. The **event handler** contains the code that should be executed when the event occurs.
- Then, you **connect the event handler to the event**.

The event handler would look like the simple method shown here:

```
public void ChangeDetected()
{
    //This code executes in response to the PriceChanged event
}
```

Handling an Event

- The next step is to **hook up the event handler to the event**.

Use simple assignment statement that sets the event PriceChanged to the event handling method changeDetected by using the += operator

```
Product saleProduct = new Product(“Kitchen”, “Garbage”, “49.99M”)
//This connects the saleProduct.PriceChanged event to an event handling
//procedure called ChangeDetected.
//procedure called ChangeDetected needs to match the
//PriceChangedEventHandler
saleProduct.PriceChanged += ChangeDetected;
//Now the event will occur in response to this code
salesProduct.Price = saleProduct.Price *2;
```

Value Types and Reference Types

- Simple data types are ***value types**,* while classes are ***reference types**.*
- This means a variable for a simple data type contains the **actual information** you put in it .
- **Object variables** actually store a **reference that points to a location in memory** where the full object is stored.

- In **three** cases you will notice that object variables act a little **differently** than ordinary data types:
  - o Assignment operations
  - o Equality testing
  - o Passing parameters

Assignment Operations

- When you assign a simple data variable to another simple data variable, the **contents of the variable are copied.**

  ```
  integerA = integerB // integerA now has a copy of the contents of integerB.
  //There are two duplicate integers in memory.
  ```

- When you assign a **reference type** you **copy the reference that *points* to the object**, not the full object content.

```
//Create a new Product object.
Product productVariable1 = New Product("Kitchen Garbage", 49.99D);
//Declare a second variable.
Product productVariable1;

productVariable2 = productVariable1
' productVariable1 and productVariable2 now both point to the same thing.
' There is one object and two ways to access it.
```

Equality Testing

- When you compare **value types you're comparing the contents.**

```
If integerA == integerB Then
        //This is true as long as the integers have the same content.
End If
```

- When you compare **reference type variables**, you're actually testing whether **they're the same instance**. In other words, you're testing whether the references are pointing to the same object in memory, not if their contents match.

```
If productVariable1 == productVariable2 Then
        //This is True if both productVariable1 and productVariable2
        // point to the same thing.
        //This is False if they are separate objects, even if they have
        //identical content.
End If
```

Passing Parameters by Reference and by Value

You can use **two types of method parameters**.

- The standard type is ***pass-by-value***. When you use pass-by-value parameters, the method receives a copy of the parameter data.
  - That means if the method modifies the parameter, this change won’t affect the calling code.
  - By default, all parameters are pass-by-value.

- The second type of parameter is ***pass-by-reference***. With pass-by-reference, the method accesses the parameter value directly. If a method changes the value of a pass-by-reference parameter, the original object is also modified.

Passing Parameters by Reference and by Value

```
private void ProcessNumber(int number)
{
    number *= 2;
}
```

- Here’s how you can call ProcessNumber():

```
int num = 10;
ProcessNumber(num) //When this call completes, num will still be 10.
```

- When this code calls ProcessNumber() it passes a copy of the num variable. This copy is multiplied by two. However, the variable in the calling code isn’t affected at all.

Passing Parameters by Reference and by Value

- This behavior changes when you use the **ref** keyword, as shown here:

```
private void ProcessNumber(ref int number)
{
    number *= 2
}
```

- Now when the method modifies this parameter (multiplying it by 2), the calling code is also affected:

```
int num = 10;
ProcessNumber(num) //Once this call completes, Num will be 20.
```

Passing Parameters by Reference and by Value

- However, if you use **reference types**, just the *reference* that's transmitted.
- To understand the difference, consider this method:

```
private void ProcessProduct(Product prod)
{
    prod.Price *= 2;
}
```

- This code accepts a Product object and increases the price by a factor of 2. Because the Product object is passed by value, you might reasonably expect that the ProcessProduct() method receives a copy of the Product object.
- Instead, the ProcessProduct() method gets a copy of the *reference*. However, this new reference still points to the same in-memory Product object. That means that the change shown in this example will affect the calling code.

Understanding Namespaces and Assemblies

- Whether you realize it at first, **every piece of code in .NET exists inside a .NET type** (typically a class).
- In turn, **every type exists inside a namespace**.
- System namespace alone is stocked with several hundred classes.
- **Namespaces** can organize all the different types in the class library.
- Without namespaces, these types would all be grouped into a single long and messy list.

Understanding Namespaces and Assemblies

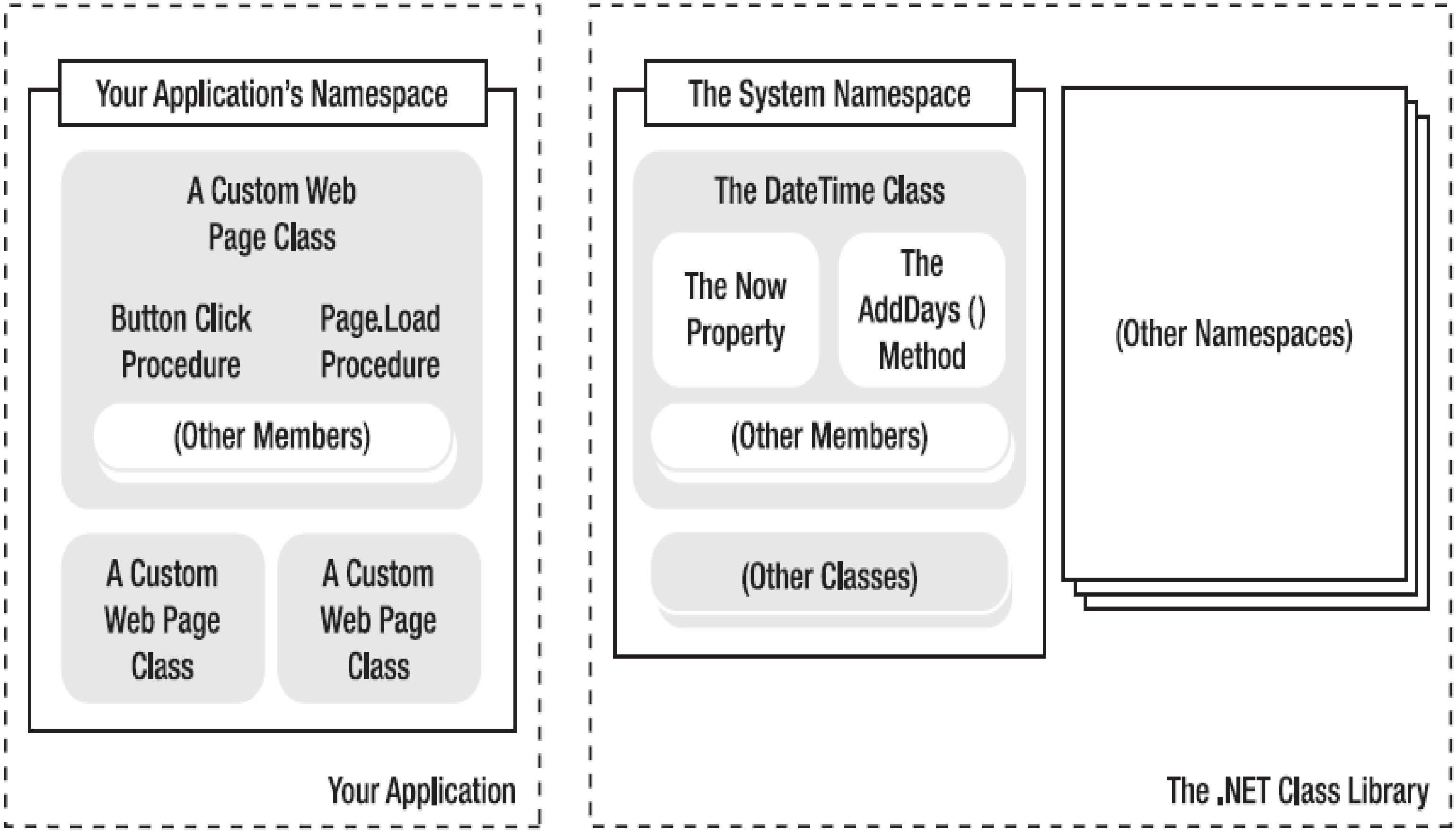


Figure 3.1 *A look at two namespaces*

Using Namespaces

- If you want to organize your code into multiple namespaces, you can define the namespace using a simple block structure, as shown here:

```
namespace MyCompany
{
    namespace MyApp
    {
        public class Product
        {
         //Code goes here
        }
    }
}
```

Using Namespaces

- In the preceding example, the Product class is in the namespace **MyCompany.MyApp**.
- Code **inside** this namespace can access the **Product class by name**.
- Code **outside** it needs to use the **fully qualified name**, as in **MyCompany.MyApp.Product**.
- This ensures that you can use the components from various third-party developers without worrying about a name collision.
- If those developers follow the recommended naming standards, their classes will always be in a namespace that uses the name of their company and software product.
- The fully qualified name of a class will then almost certainly be unique.

Using Namespaces

- **Namespaces** don’t take an **accessibility** keyword and can be **nested** as many layers deep as you need.
- You can declare the **same namespace** in various code files.
- In fact, more than one project can even use the same namespace.
- **Namespaces** are really nothing more than Convenient, **logical containers** that help you organize your classes.

Importing Namespaces

- Having to type long, fully qualified names is certain to tire your fingers and create overly verbose code.
- To simplify matters, it’s standard practice to **import the namespaces** you want to use.
- When you import a namespace, you don’t need to **type the fully qualified type names.**
- Instead, you can use the types in that namespace as though they were defined locally. To import a namespace, you use the **using** statement.
- These statements must appear as the **first lines in your code file**, outside of any namespaces or block structure

Importing Namespaces

```
using MyCompany.MyApp;
```

- Consider the situation without importing a namespace:

```
MyCompany.MyApp.Product  salesProduct = new
    MyCompany.MyApp.Product (…);
```

- It’s much more manageable when you import the MyCompany.MyApp namespace. Once you do, you can use this shortened syntax instead:

```
Product salesProduct = new Product(…);
```

Importing Namespaces

- Importing namespaces is really just a convenience. It has **no effect** on the performance of your application.

- In fact, whether you use namespace imports, the compiled IL code will look the same.

- That's because the language compiler will **translate your relative class references into fully qualified class names when it generates an EXE or a DLL file**.

Assemblies

- All .NET classes are contained in ***assemblies*.**

- **Assemblies** are the **physical files that contain compiled code**.

- Typically, assembly files have the extension **.exe** if they are **stand-alone** applications, or **.dll** if they’re **reusable components**.

Assemblies

- An **assembly** can contain **multiple namespaces**.
- Conversely, more than one assembly file can contain classes in the same namespace.
- Technically, **namespaces** are a ***logical*** way to group classes.
- **Assemblies**, however, are a ***physical*** package for distributing code.

Advanced Class Programming

- **Containment or Aggregation :**
- For example, the following code shows a ProductCatalog class, which holds an array of Product objects:

```
public class ProductCatalog
{
    private Product[] products;
    //other class code goes here
}
```

Inheritance

- **Inheritance** is a form of code **reuse**.
- It allows one class to acquire and extend the functionality of another class.
- With inheritance, **constructors are never inherited**.
- The only way to handle this problem is to add a constructor in your derived class (TaxableProduct) that calls the right constructor in the base class (Product) using the **base** keyword.

Inheritance

```
public class TaxableProduct : Product
{
    private decimal taxRate = 1.15M;
    public decimal TotalPrice
    {
        get
        {
            //The code can access the Price property because it’s
            //a public part of the base class Product
            // The code cannot access the private price variable, however.
            return (Price * taxRate);
        }
    }
    public TaxableProduct(string name, decimal price, string imageurl) :
         base(name, price, imageurl)
    {
    }
}
```

Static Members

- **Static** properties and methods which can be used without a live object. Static members are often used to provide useful functionality related to an object.
- Static members have a wide variety of possible uses
- To create a static property or method we have to just use ***static*** keyword after the accessibility keyword

Static Members

```
public class TaxableProduct : Product
{
    //(Additional class code omitted for clarity )
    private static decimal taxRate = 1.15M;

    // Now we can call TaxableProduct.TaxRate, even without an object
    public static decimal TaxRate
     {
          get
            { return taxRate;}
          set
             { taxRate = value;}
     }
}
```

Static Members

- You can now get or set the tax rate information **directly from the class**, without needing to create an object first:

```
//Change the TaxRate. This will affect all TotalPrice calculations for any
//TaxableProduct object.
 TaxableProduct.TaxRate = 1.24M;
```

- Static data **isn't tied to the lifetime of an object**. In fact, it's available throughout the life of the entire application.
- This means static members are the closest thing .NET programmers have to global data.
- A **static member can't access an instance member**. To access a nonstatic member, it needs an actual instance of your object.

Casting Objects

- Object variables can be converted with the same syntax that’s used for simple data types. This process is called ***casting***.
- When you perform casting, you don’t actually change anything about an object. What you **change is the variable that points to the object**.
- An object variable can be cast into one of three things:
  - **Itself**
  - **An interface that it supports**
  - **Base class from which it inherits**
- You can’t cast an object variable into a string or an integer. Instead, you need to call a conversion method, if it’s available, such as ToString() or Parse().

Casting Objects

```
//Create a TaxableProduct.
TaxableProduct  theTaxableProduct  =
    new TaxableProduct(“Kitchen Garbage”, 49.99M, “garbage.jpg”);

//Cast the TaxableProduct reference to a Product reference
  Product  theProduct  =  theTaxableProduct;
```

- You **don’t lose any information** when you perform this casting.
- There is still just one object in memory (with two variables pointing to it), and this object really *is* a **TaxableProduct.**

Casting Objects

- However, when you use the variable **theProduct** to access your **TaxableProduct** object, you'll be limited to the properties and methods that are defined in the Product class. That means code like this won't work:

```
//This code generates a compile-time error.
decimal TotalPrice = theProduct.TotalPrice;
```

- Even though **theProduct** actually holds a reference that points to a **TaxableProduct** and even though the **TaxableProduct** has a **TotalPrice property**, you can't access it through **theProduct**.
- That's because **theProduct** treats the object it refers to as an ordinary Product.

Partial Classes

- **Partial classes** give you the **ability to split a single class into more than one source code (.cs) file**.
- A partial class behaves the same as a **normal class**.
- This means every method, property, and variable you've defined in the class is available everywhere, **no matter which source file contains it**.
- When you compile the application, the compiler tracks down each piece of the Product class and assembles it into a complete unit.
- It doesn't matter what you name the source code files, so long as you keep the class name consistent.

Partial Classes

```
//This part is stored in file Product1.cs
public partial class Product
{
    public string Name { get;   set;}
    public event PriceChangedEventHandler PriceChanged;
    private decimal price;
    public decimal Price
    {
          get
          {    return price;    }
           set
           {
                 price = value;
                 //Fire the event, provided there is at least one listener
                 if(PriceChanged != null)
                  {  PriceChanged();   }
           }
    }
    public string ImageUrl   {   get;  set;    }
    public Product (String name, decimal price, string imageUrl)
    {
      Name = name;     Price = price;      ImageUrl = imageUrl;
    }
}
```

Partial Classes

```
// This part is stored in file Product2.cs
Public partial class Product
{
    public string GetHtml()
     {
         string htmlString;
         htmlString = “<h1>” + Name + “</h1><br  />”
         htmlString = “<h3>Costs : ” + Price.ToString() + “</h3><br  />”
         htmlString = “<img src=‘ “ +ImageUrl + “ ‘ />”;
          return htmlString;
     }
}
```

Generics

- **Generics** allow you to create **classes that are parameterized by type**.

- In other words, you create a **class template** that supports any type.

- When you **instantiate that class**, you specify the type you want to use, and from that point on, your object is “locked in” to the type you chose.

Generics

- Iimagine you use an **ArrayList** to track a catalog of products. You intend to use the ArrayList to store Product objects, but there’s nothing to stop a piece of misbehaving code from inserting strings, integers, or any arbitrary object in the ArrayList. Here’s an example:

```
 //Create the ArrayList.
    ArrayList products = new ArrayList();
// Add several Product objects.
    products.Add(product1)
    products.Add(product2)
    products.Add(product3)
//Notice how you can still add other types to the ArrayList.
    products.Add("This string doesn't belong here.")
```

Generics

- The **solution** is a new **generic List collection class**. Because it uses generics, you must **lock it into a specific type** whenever you instantiate a List object.
- To do this, you specify the class you want to use in angle brackets after the class name

```
//Create the List for storing Product objects.
List<Product>  products  =  new  List<Product>();
```

- Now you can add only Product objects to the collection:

```
//Add several Product objects.
products.Add(product1)
products.Add(product2)
products.Add(product3)
//This line fails. In fact, it won't even compile.
products.Add("This string can't be inserted.")
```

Note: You can find the List class, and many more collections that use generics, in the **System.Collections.Generic** namespace.