
Module 8: Using Reference-Type Variables

Contents

Overview	1
Using Reference-Type Variables	2
Using Common Reference Types	15
The Object Hierarchy	23
Namespaces in the .NET Framework	29
Lab 8.1: Defining And Using Reference - Variables	35
Data Conversions	43
Multimedia: Type-Safe Casting	56
Lab 8.2 Converting Data	57
Review	63

This course is based on the prerelease Beta 1 version of Microsoft® Visual Studio .NET. Content in the final release of the course may be different from the content included in this prerelease version. All labs in the course are to be completed with the Beta 1 version of Visual Studio .NET.



Information in this document is subject to change without notice. The names of companies, products, people, characters, and/or data mentioned herein are fictitious and are in no way intended to represent any real individual, company, product, or event, unless otherwise noted. Complying with all applicable copyright laws is the responsibility of the user. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Microsoft Corporation. If, however, your only means of access is electronic, permission to print one copy is hereby granted.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2001 Microsoft Corporation. All rights reserved.

Microsoft, ActiveX, BizTalk, IntelliSense, JScript, Microsoft Press, MSDN, PowerPoint, Visual Basic, Visual C++, Visual C#, Visual Studio, Windows, and Windows Media are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names mentioned herein may be the trademarks of their respective owners.

Overview

- Using Reference-Type Variables
- Using Common Reference Types
- The Object Hierarchy
- Namespaces in the .NET Framework
- Data Conversions

In this module, you will learn how to use reference types in C#. You will learn about a number of reference types, such as string, that are built into the C# language and run-time environment. These are discussed as examples of reference types.

You will also learn about the C# object hierarchy and the **object** type in particular, so you can understand how the various reference types are related to each other and to the value types. You will learn how to convert data between reference types by using explicit and implicit conversions. You will also learn how boxing and unboxing conversions convert data between reference types and value types.

After completing this module, you will be able to:

- Describe the important differences between reference types and value types.
- Use common reference types, such as string.
- Explain how the **object** type works and become familiar with the methods it supplies.
- Describe common namespaces in the Microsoft® .NET Framework.
- Determine whether different types and objects are compatible.
- Explicitly and implicitly convert data types between reference types.
- Perform boxing and unboxing conversions between reference and value data.

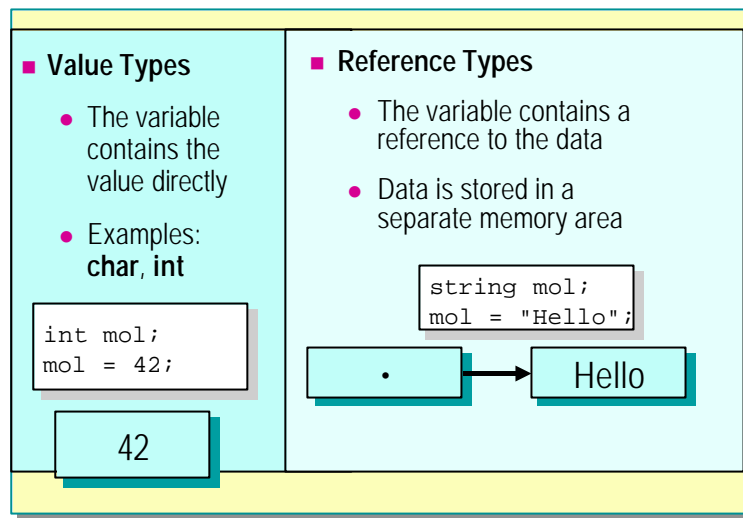
◆ Using Reference-Type Variables

- Comparing Value Types to Reference Types
- Declaring and Releasing Reference Variables
- Invalid References
- Comparing Values and Comparing References
- Multiple References to the Same Object
- Using References as Method Parameters

Reference types are important features of the C# language. They enable you to write complex and powerful applications and effectively use the run-time framework.

In this section, you will learn about reference-type variables and about how they are different from value-type variables. You will learn how to use and discard reference variables. You will also learn how to pass reference types as method parameters.

Comparing Value Types to Reference Types



C# supports basic data types such as `int`, `long` and `bool`. These types are also referred to as *value types*. C# also supports more complex and powerful data types known as *reference types*.

Value Types

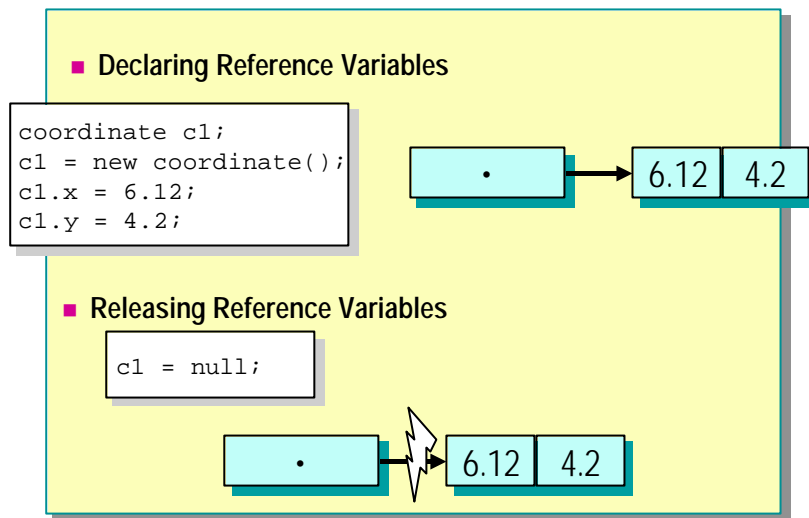
Value-type variables are the basic built-in data types such as `char` and `int`. Value types are the simplest types in C#. Variables of value type directly contain their data in the variable.

Reference Types

Reference-type variables contain a reference to the data, not the data itself. The data itself is stored in a separate memory area.

You have already used several reference types in this course so far, perhaps without realizing it. Arrays, strings, and exceptions are all reference types that are built into the C# compiler and the .NET Framework. Classes, both built-in and user-defined, are also a kind of reference type.

Declaring and Releasing Reference Variables



To use reference-type variables, you need to know how to declare and initialize them and how to release them.

Declaring Reference Variables

You declare reference-type variables by using the same syntax that you use when declaring value-type variables:

```
coordinate c1;
```

The preceding example declares a variable `c1` that can hold a reference to an object of type **coordinate**. However, this variable is not initialized to reference any **coordinate** objects.

To initialize a **coordinate** object, use the **new** operator. This creates a new object and returns an object reference that can be stored in the reference variable.

```
coordinate c1;  
c1 = new coordinate( );
```

If you prefer, you can combine the **new** operator with the variable declaration so that the variable is declared and initialized in one statement, as follows:

```
coordinate c1 = new coordinate( );
```

After you have created an object in memory to which `c1` refers, you can then reference member variables of that object by using the **dot** operator as shown in the following example:

```
c1.x = 6.12;  
c1.y = 4.2;
```

Example of Declaring Reference Variables

Classes are reference types. The following example shows how to declare a user-defined class called **coordinate**. For simplicity, this class has only two public member variables: *x* and *y*.

```
class coordinate
{
    public double x = 0.0;
    public double y = 0.0;
}
```

This simple class will be used in later examples to demonstrate how reference variables can be created, used, and destroyed.

Releasing Reference Variables

After you assign a reference to a new object, the reference variable will continue to reference the object until it is assigned to refer to a different object.

C# defines a special value called **null**. A reference variable contains **null** when it does not refer to any valid object. To release a reference, you can explicitly assign the value **null** to a reference variable (or simply allow the reference to go out of scope).

Invalid References

- **If You Have Invalid References**
 - You cannot access members or variables
- **Invalid References at Compile Time**
 - Compiler detects use of uninitialized references
- **Invalid References at Run Time**
 - System will generate an exception error

You can only access the members of an object through a reference variable if the reference variable has been initialized to point to a valid reference. If a reference is not valid, you cannot access member variables or methods.

The compiler can detect this problem in some cases. In other cases, the problem must be detected and handled at run time.

Invalid References at Compile Time

The compiler is able to detect situations in which a reference variable is not initialized prior to use.

For example, if a **coordinate** variable is declared but not assigned, you will receive an error message similar to the following: “ Use of unassigned local variable c1.” The following code provides an example:

```
coordinate c1;  
c1.x = 6.12; // Will fail: variable not assigned
```


Invalid References at Run Time

In general, it is not possible to determine at compile time when a variable reference is not valid. Therefore, C# will check the value of a reference variable before it is used, to ensure that it is not **null**.

If you try to use a reference variable that has the value **null**, the run-time system will throw a **NullReferenceException** exception. If you want, you can check for this condition by using **try** and **catch**. The following is an example:

```
try {  
    c1.x = 45;  
}  
catch (NullReferenceException) {  
    Console.WriteLine("c1 has a null value");  
}
```

Alternatively, you can check for **null** explicitly, thereby avoiding exceptions. The following example shows how to check that a reference variable contains a non-null reference before trying to access its members:

```
if (c1 != null)  
    c1.x = 45;  
else  
    Console.WriteLine("c1 has a null value");
```

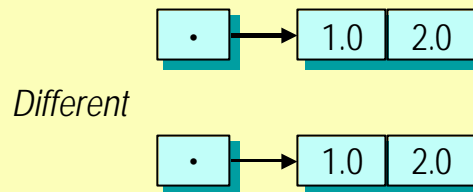
Comparing Values and Comparing References

■ Comparing Value Types

- `==` and `!=` compare values

■ Comparing Reference Types

- `==` and `!=` compare the references, not the values



The equality (`==`) and inequality (`!=`) operators might not work in the way you expect for reference variables.

Comparing Value Types

For value types, you can use the `==` and `!=` operators to compare values.

Comparing Reference Types

For reference types, you can use the `==` and `!=` operators to compare references. When comparing references with the `==` operator, you are determining whether the two reference variables are referring to the same object. You are *not* comparing the contents of the objects to which the variables refer.

Consider the following example, in which two coordinate variables are created and initialized to the same values:

```
coordinate c1= new coordinate( );
coordinate c2= new coordinate( );
c1.x = 1.0;
c1.y = 2.0;
c2.x = 1.0;
c2.y = 2.0;
if (c1 == c2)
    Console.WriteLine("Same");
else
    Console.WriteLine("Different");
```

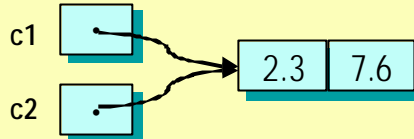
The output from this code is “ Different.” Even though the objects that *c1* and *c2* are referring to have the same values, they are references to different objects, so `==` returns **false**.

You cannot use the other relational operators (<, >, <=, and >=) for references because they are not defined in C#.

Multiple References to the Same Object

■ Two References Can Refer to the Same Object

- Two ways to access the same object for read/write



```
coordinate c1= new coordinate( );  
coordinate c2;  
c1.x = 2.3; c1.y = 7.6;  
c2 = c1;  
Console.WriteLine(c1.x + " , " + c1.y);  
Console.WriteLine(c2.x + " , " + c2.y);
```

Two reference variables can refer to the same object because reference variables hold a reference to the data. This means that you can write data through one reference and read the same data through another reference.

Multiple References to the Same Object

In the following example, the variable `c1` is initialized to point to a new instance of the class, and its member variables `x` and `y` are initialized. Then `c1` is copied to `c2`. Finally, the values in the objects that `c1` and `c2` reference are displayed.

```
coordinate c1 = new coordinate( );  
coordinate c2;  
c1.x = 2.3;  
c1.y = 7.6;  
c2 = c1;  
Console.WriteLine(c1.x + " , " + c1.y);  
Console.WriteLine(c2.x + " , " + c2.y);
```

The output of this program is as follows:

```
2.3 , 7.6  
2.3 , 7.6
```

Assigning `c2` to `c1` copies the reference so that both variables are referencing the same instance. Therefore, the values printed for the member variables of `c1` and `c2` are the same.

Writing and Reading the Same Data Through Different References

In the following example, an assignment has been added immediately before the calls to **Console.WriteLine**.

```
coordinate c1 = new coordinate( );  
coordinate c2;  
c1.x = 2.3;  
c1.y = 7.6;  
c2 = c1;  
c1.x = 99; // This is the extra statement  
Console.WriteLine(c1.x + " , " + c1.y);  
Console.WriteLine(c2.x + " , " + c2.y);
```

The output of this program is as follows:

```
99 , 7.6  
99 , 7.6
```

This shows that the assignment of 99 to *c1.x* has also changed *c2.x*. Because the reference in *c1* was previously assigned to *c2*, a program can write data through one reference and read the same data through another reference.

Using References as Method Parameters

■ References Can Be Used as Parameters

- When passed by reference, data being referenced may be changed

```
static void PassCoordinateByValue(coordinate c)
{
    c.x++; c.y++;
}
```

```
loc.x = 2; loc.y = 3;
PassCoordinateByValue(loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

You can pass reference variables in and out of a method.

References and Methods

You can pass reference variables into methods as parameters by using any of the three calling mechanisms:

- By value
- By reference
- Output parameters

The following example shows a method that passes three coordinate references. The first is passed by value, the second is passed by reference, and the third is an output parameter. The return value of the method is a coordinate reference.

```
static coordinate Example(
    coordinate ca,
    ref coordinate cb,
    out coordinate cc)
{
    // ...
}
```

Passing References by Value

When you use a reference variable as a value parameter, the method receives a copy of the reference. This means that for the duration of the call there are two references referencing the same object. It also means that any changes to the method parameter cannot affect the calling reference. For example, the following code displays the values 0 , 0:

```
static void PassCoordinateByValue(coordinate c)
{
    c = new coordinate( );
    c.x = c.y = 22.22;
}
coordinate loc = new coordinate( );
PassCoordinateByValue(loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

Passing References by Reference

When you use a reference variable as a **ref** parameter, the method receives the actual reference variable. In contrast to passing by value, in this case there is only one reference. The method does *not* make its own copy. This means that any changes to the method parameter will affect the calling reference. For example, the following code displays the values 33.33 , 33.33:

```
static void PassCoordinateByRef(ref coordinate c)
{
    c = new coordinate( );
    c.x = c.y = 33.33;
}
coordinate loc = new coordinate( );
PassCoordinateByRef(ref loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

Passing References by Output

When you use a reference variable as an **out** parameter, the method receives the actual reference variable. In contrast to passing by value, in this case there is only one reference. The method does *not* make its own copy. Passing by **out** is similar to passing by **ref** except that the method *must* assign to the **out** parameter. For example, the following code displays the values 44.44 , 44.44:

```
static void PassCoordinateByOut(out coordinate c)
{
    c = new coordinate( );
    c.x = c.y = 44.44;
}
coordinate loc = new coordinate( );
PassCoordinateByOut(out loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

Passing References into Methods

Variables of reference types do not hold the value directly, but hold a reference to the value instead. This also applies to method parameters, and this means that the pass-by-value mechanism can produce unexpected results.

Using the **coordinate** class as an example, consider the following method:

```
static void PassCoordinateByValue(coordinate c)
{
    c.x++;
    c.y++;
}
```

The **coordinate** parameter *c* is passed by value. In the method, both the *x* and *y* member variables are incremented. Now consider the following code that calls the **PassCoordinateByValue** method:

```
coordinate loc = new coordinate( );
loc.x = 2;
loc.y = 3;
PassCoordinateByValue(loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

The output of this code is the following:

```
3 , 4
```

This shows that the values referenced by **loc** have been changed by the method. This might seem to be in conflict with the explanation of pass by value given previously in the course, but in fact it is consistent. The reference variable *loc* is copied into the parameter *c* and cannot be changed by the method, but the memory to which it refers is not copied and is under no such restriction. The variable *loc* still refers to the same area of memory, but that area of memory now contains different data.

◆ Using Common Reference Types

- Exception Class
- String Class
- Common String Methods, Operators, and Properties
- String Comparisons
- String Comparison Operators

A number of reference-type classes are built in to the C# language. In this section, you will review some familiar built-in classes and learn more about how they work.

You can also use these built-in classes as models when creating your own classes.

Exception Class

- **Exception Is a Class**
- **Exception Objects Are Used to Raise Exceptions**
 - Create an **Exception** object by using **new**
 - Throw the object by using **throw**
- **Exception Types Are Subclasses of Exception**

You create and throw **Exception** objects to raise exceptions.

Exception Class.

Exception is the name of a class provided in the .NET Framework.

Exception Objects

Only objects of **Exception** type can be thrown with **throw** and caught with **catch**. In other respects, the **Exception** class is like other reference types.

Exception Types

Exception represents a generic fault in an application. There are also specific exception types (such as **InvalidCastException**). There are classes that inherit from **Exception** that represent each of these specific exceptions.

String Class

- Multiple Character Unicode Data
- Shorthand for System.String
- Immutable



```
string s = "Hello";  
  
s[0] = 'c'; // Compile-time error
```

In C#, the string type is used for processing multiple character Unicode character data. (The char type, by comparison, is a value type that handles single characters.)

The type name **string** is a shortened name for the **System.String** class. The compiler can process this shortened form; therefore **string** and **System.String** can be used interchangeably.

The **String** class represents an immutable string of characters. An instance of **String** is immutable: the text of a string cannot be modified once it has been created. Methods that might appear at first sight to modify a string value actually return a new instance of string that contains the modification.

Tip The **StringBuilder** class is often used in partnership with the **String** class. A **StringBuilder** builds an internally modifiable string that can be converted into an immutable **String** when complete. **StringBuilder** removes the need to repeatedly create temporary immutable **Strings** and can provide improved performance.

The **System.String** class has many methods. This course will not provide a full tutorial for string processing, but it will list some of the more useful methods. For further details, consult the .NET Framework SDK Help documents.

Common String Methods, Operators, and Properties

- Brackets
- Insert Method
- Length Property
- Copy Method
- Concat Method
- Trim Method
- ToUpper and ToLower Methods

Brackets []

You can extract a single character at a given position in a string by using the string name followed by the index in brackets ([and]). This process is similar to using an array. The first character in the string has an index of zero.

The following code provides an example:

```
string s = "Alphabet"  
char firstchar = s[2]; // 'p'
```

Strings are immutable, so assigning a character by using brackets is not permitted. Any attempt to assign a character to a string in this way will generate a compile-time error, as shown:

```
s[2] = '*'; // Not valid
```

Insert Method

If you want to insert characters into a string variable, use the **Insert** instance method to return a new string with a specified value inserted at a specified position in this string. This method takes two parameters: the position of the start of the insertion and the string to insert.

The following code provides an example:

```
string s = "C is great!";  
s = s.Insert(2, "Sharp ");  
Console.WriteLine(s); // C Sharp is great!
```

Length Property

The **Length** property returns the length of a string as an integer, as shown:

```
string msg = "Hello";  
int slen = msg.Length; // 5
```

Copy Method

The **Copy** class method creates a new string by copying another string. The **Copy** method makes a duplicate of a specified string.

The following code provides an example:

```
string s1 = "Hello";  
string s2 = String.Copy(s1);
```

Concat Method

The **Concat** class method creates a new string from one or more strings or objects represented as strings.

The following code provides an example:

```
string s3 = String.Concat("a", "b", "c", "d", "e", "f", "g");
```

The **+** operator is overloaded for strings, so the example above can be re-written as follows:

```
string s = "a" + "b" + "c" + "d" + "e" + "f" + "g";  
Console.WriteLine(s);
```

Trim Method

The **Trim** instance method removes all of the specified characters or white space from the beginning and end of a string.

The following code provides an example:

```
string s = "    Hello    ";  
s = s.Trim();  
Console.WriteLine(s); // "Hello"
```

ToUpper and ToLower Methods

The **ToUpper** and **ToLower** instance methods return a string with all characters converted to uppercase and lowercase, respectively, as shown:

```
string sText = "How to Succeed ";  
Console.WriteLine(sText.ToUpper()); // HOW TO SUCCEED  
Console.WriteLine(sText.ToLower()); // how to succeed
```

String Comparisons

- **Equals Method**
 - Value comparison
- **Compare Method**
 - More comparisons
 - Case-insensitive option
 - Dictionary ordering
- **Locale-Specific Compare Options**

You can use the `==` and `!=` operators on string variables to compare string contents.

Equals Method

The **System.String** class contains an instance method called **Equals**, which can be used to compare two strings for equality. The method returns a bool value that is **true** if the strings are the same and **false** otherwise. This method is overloaded and can be used as an instance method or a static method. The following example shows both forms.

```
string s1 = "Welcome";  
string s2 = "Welcome";  
  
if (s1.Equals(s2))  
    Console.WriteLine("The strings are the same");  
  
if (String.Equals(s1, s2))  
    Console.WriteLine("The strings are the same");
```

Compare Method

The **Compare** method compares two strings lexically; that is, it compares the strings according to their sort order. The return value from **Compare** is as follows:

- A negative integer if the first string comes before the second
- 0 if the strings are the same
- A positive integer if the first string comes after the second

```
string s1 = "Tintinnabulation";  
string s2 = "Velocipele";  
int comp = String.Compare(s1, s2); // Negative return
```

By definition, any string, including an empty string, compares greater than a **null** reference, and two **null** references compare equal to each other.

Compare is overloaded. There is a version with three parameters, the third of which is a bool value that specifies whether the case should be ignored in the comparison. The following example shows a case-insensitive comparison:

```
s1 = "cabbage";  
s2 = "Cabbage";  
comp = String.Compare(s1, s2, true); // Ignore case
```

Locale-Specific Compare Options

The **Compare** method has overloaded versions that allow string comparisons based on language-specific sort orders. This can be useful when writing applications for an international market. Further discussion of this feature is beyond the scope of the course. For more information, search for “System.Globalization namespace” and “CultureInfo class” in the .NET Framework SDK Help documents.

String Comparison Operators

- The == and != Operators Are Overloaded for Strings
- They Are Equivalent to `String.Equals` and `!String.Equals`

```
string a = "Test";  
string b = "Test";  
if (a == b) ... // Returns true
```

The == and != operators are overloaded for the **String** class. You can use these operators to examine the contents of strings.

```
string a = "Test";  
string b = "Test";  
if (a == b) ... // Returns true
```

The following operators and methods are equivalent:

- The == operator is equivalent to the **String.Equals** method.
- The != operator is equivalent to the **!String.Equals** method.

The other relational operators (<, >, <=, and >=) are not overloaded for the **String** class.

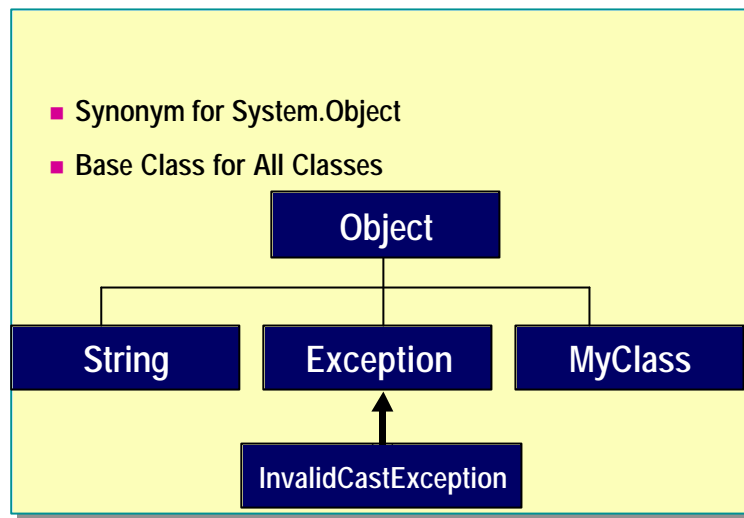
◆ The Object Hierarchy

- The object Type
- Common Methods
- Reflection

The C# classes are arranged in a hierarchy with the **Object** class at the top. The **object** type therefore describes the common behavior for all reference types in the C# language.

In this section, you will learn about the **object** type and how the object hierarchy works.

The object Type



The **object** type is the base class for all types in C#.

System.Object

The **object** keyword is a synonym for the **System.Object** class in the .NET Framework. Anywhere the keyword **object** appears, the class name **System.Object** can be substituted. Because of its convenience, the shorter form is more common.

Base Class

All classes inherit from **object** either directly or indirectly. This includes the classes you write in your application and those classes that are part of the system framework. When you declare a class with no explicit parent, you are actually inheriting from **object**.

Common Methods

■ Common Methods for All Reference Types

- **ToString** method
- **Equals** method
- **GetType** method
- **Finalize** method

The **object** type has a number of common methods that are inherited by all other reference types.

Common Methods for All Reference Types

The **object** type provides a number of common methods. Because every reference type inherits from **object**, every other reference type in C# has these methods too. These common methods include the following:

- **ToString**
- **Equals**
- **GetType**
- **Finalize**

ToString Method

The **ToString** method returns a string that represents the current object.

The default implementation, as found in the **Object** class, returns the type name of the class. The following example uses the **coordinate** example class defined earlier:

```
coordinate c = new coordinate( );  
Console.WriteLine(c.ToString( ));
```

This example will display “coordinate” on the console.

However, you can override the **ToString** method in class **coordinate** to render objects of that type into something more meaningful, such as a string containing the values held in the object.

Equals Method

The **Equals** method determines whether the specified object is the same instance as the current object. The default implementation of **Equals** supports reference equality only, as you have already seen.

Subclasses can override this method to support value equality instead.

GetType Method

This method allows extraction of run-time type information from an object. It is discussed in more detail in the Data Conversions section later in this module.

Finalize Method

This method is called by the run-time system when memory allocated to a reference is released.

Reflection

- You Can Query the Type of an Object
- System.Reflection Namespace
- The typeof Operator Returns a Type Object
 - Compile-time classes only
- GetType Method in System.reflection
 - Run-time class information

You can obtain information about the type of an object by using a mechanism called reflection.

The reflection mechanism in C# is handled by the **System.Reflection** namespace in the .NET Framework. This namespace contains classes and interfaces that provide a view of types, methods, and fields.

The **System.Type** class provides methods for obtaining information about a type declaration, such as the constructors, methods, fields, properties, and events of a class. A **Type** object that represents a type is unique; that is, two **Type** object references refer to the same object only if they represent the same type. This allows comparison of **Type** objects through reference comparisons (the == and != operators).

The typeof Operator

At compile time, you can use the **typeof** operator to return the type information from a given type name.

The following example retrieves run-time type information for the type `byte`, and displays the type name to the console.

```
using System;
using System.Reflection;
Type t = typeof(byte);
Console.WriteLine("Type: {0}", t);
```

The following example displays more detailed information about a class. Specifically, it lists the methods for that class.

```
using System;
using System.Reflection;
Type t = typeof(string); // Get type information
MethodInfo[] mi = t.GetMethods();
foreach (MethodInfo m in mi) {
    Console.WriteLine("Method: {0}", m);
}
```

GetType Method

The **typeof** operator only works on classes that exist at compile time. If you need type information at run time, you can use the **GetType** method of the **Object** class.

For more information about reflection, search for “System.Reflection” in the .NET Framework SDK Help documents

◆ Namespaces in the .NET Framework

- System.IO Namespace
- System.XML Namespace
- System.Data Namespace
- Other Useful Namespaces

The .NET Framework provides common language services to a variety of application development tools. The classes in the framework provide an interface to the Common Language Runtime, the operating system, and the network.

In this section, you will learn how to use some of the common namespaces within the framework. You are likely to use these namespaces on a regular basis, so it is important to be familiar with them.

The .NET Framework is large and powerful, and full coverage of every feature is beyond the scope of this course. For more detailed information, please consult the Visual Studio.NET and .NET Framework SDK Help documents.

System.IO Namespace

■ Access to File System Input/Output

- File, Directory
- StreamReader, StreamWriter
- FileStream
- BinaryReader, BinaryWriter

The **System.IO** namespace is important because it contains many classes that allow an application to perform input and output (I/O) operations in various ways through the file system.

The **System.IO** namespace also provides classes that allow an application to perform input and output operations on files and directories.

The **System.IO** namespace is large and cannot be explained in detail here. However, the following list gives an indication of the facilities available:

- The **File** and **Directory** classes allow an application to create, delete, and manipulate directories and files.
- The **StreamReader** and **StreamWriter** classes enable a program to access file contents as a stream of bytes or characters.
- The **FileStream** class can be used to provide random access to files.
- The **BinaryReader** and **BinaryWriter** classes provide a way to save and load objects to and from streams.

System.IO Example

A brief example follows, to show how a file can be opened and read as a stream. The example is not meant to illustrate all of the possible ways in which the **System.IO** namespace can be used, but does show how you can perform a simple file copy operation.

```
using System;
using System.IO; // Use IO namespace
// ...
StreamReader reader = new StreamReader("infile.txt");
// Text in from file
StreamWriter writer = new StreamWriter("outfile.txt");
// Text out to file
string line;
while ((line = reader.ReadLine()) != null)
{
    writer.WriteLine(line);
}

reader.Close();
writer.Close();
```

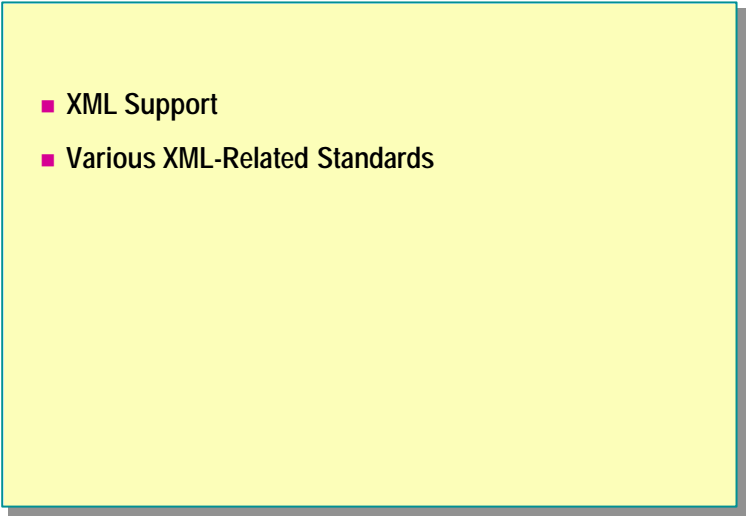
To open a file for reading, the code in the example creates a new **StreamReader** object and passes the name of the file that needs to be opened in the constructor. Similarly, to open a file for writing, the example creates a new **StreamWriter** object and passes the output file name in its constructor. In the example, the file names are hard-coded, but they could also be string variables.

The example program copies a file by reading one line at a time from the input stream and writing that line to the output stream.

ReadLine and **WriteLine** might look familiar. The **Console** class has two static methods of that name. In the example, the methods are instance methods of the **StreamReader** and **StreamWriter** classes, respectively.

For more information about the **System.IO** namespace, search for “System.IO namespace” in the .NET Framework SDK Help documents.

System.XML Namespace

- 
- XML Support
 - Various XML-Related Standards

Applications that need to interact with Extensible Markup Language (XML) can use the **System.XML** namespace, which provides standards-based support for processing XML.

The **System.XML** namespace supports a number of XML-related standards, including the following:

- XML 1.0 with document type definition (DTD) support
- XML namespaces
- XML schemas
- XPath expressions
- XSL/T transformations
- DOM Level 2 core
- Simple Object Access Protocol (SOAP) 1.1

The **XMLDocument** class is used to represent an entire XML document. Elements and attributes in an XML document are represented in the **XMLElement** and **XMLAttribute** classes.

A detailed discussion of XML namespaces is beyond the scope of this course. For further information, search for “System.XML namespace” in the .NET Framework SDK Help documents.

System.Data Namespace

- **System.Data.SQL**
 - SQL Server specific
- **System.Data.ADO**
 - Interact with OLEDB and ODBC
 - Generic database drivers

The **System.Data** namespace contains classes that constitute the ADO.NET architecture. The ADO.NET architecture enables you to build components that efficiently manage data from multiple data sources. ADO.NET provides the tools to request, update, and reconcile data in *n*-tier systems.

Within ADO.NET, you can use the **DataSet** class. In each **DataSet**, there are **DataTable** objects, and each **DataTable** contains data from a single data source, such as Microsoft SQL Server™.

The **System.Data.SQL** namespace provides direct access to SQL Server. Note that this namespace is specific to SQL Server.

For access to other relational databases and sources of structured data, there is the **System.Data.ADO** namespace, which provides high-level access to the OLEDB and Open Database Connectivity (ODBC) database drivers.

A detailed discussion of the **System** namespaces is not within the scope of this course. For further information, search for “System.Data namespace” in the .NET Framework SDK Help documents.

Other Useful Namespaces

- **System Namespace**
- **System.Net Namespace**
- **System.Net.Sockets Namespace**
- **System.Windows.Forms Namespace**

There are many other useful namespaces and classes in the .NET Framework. This course does not discuss them all at length, but the following information might be helpful when you search the reference files and documentation:

- The **System** namespace contains classes that define commonly used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions. Other classes provide services that support data type conversion, method parameter manipulation, mathematics, remote and local program invocation, and application management.
- The **System.Net** namespace provides a simple programming interface to many of the protocols found on the network today. The **System.Net.Sockets** namespace provides an implementation of the Microsoft Windows® Sockets interface for developers who need to low-level access to Transmission Control Protocol/Internet Protocol (TCP/IP) network facilities.
- **System.WinForms** is the graphical user interface (GUI) framework for Windows applications, and provides support for forms, controls, and event handlers.

For more information about **System** namespaces, search for “System namespace” in the .NET Framework SDK Help documents.

Lab 8.1: Defining And Using Reference-Type Variables



Objectives

After completing this lab, you will be able to:

- Create reference variables and pass them as method parameters.
- Use the system frameworks.

Prerequisites

Before working on this lab, you should be familiar with the following:

- Creating and using classes
- Calling methods and passing parameters
- Using arrays

Estimated time to complete this lab: 45 minutes

Exercise 1

Adding an Instance Method with Two Parameters

In Lab 7, you developed a **BankAccount** class.

In this exercise, you will re-use this class and add a new instance method, called **TransferFrom**, which transfers money from a specified account into this one. If you did not complete Lab 7, you can obtain a copy of the **BankAccount** class in the *install_folder*\Labs\Lab08\Starter folder.

✍ To create the TransferFrom method

1. Open the Bank.sln project in the *install_folder*\Labs\Lab08\Starter\Bank folder.
2. Edit the **BankAccount** class as follows:
 - a. Create a public instance method called **TransferFrom** in the **BankAccount** class.
 - b. The first parameter is a reference to another **BankAccount** object, called **accFrom**, from which the money is to be transferred.
 - c. The second parameter is a **decimal** value, called *amount*, passed by value and indicating the amount to transfer.
 - d. The method has no return value.
3. In the body of **TransferFrom**, add two statements that perform the following tasks:
 - a. Debit *amount* from the balance of **accFrom** (by using **Withdraw**).
 - b. Test to ensure that the withdrawal was successful. If it was, credit *amount* to the balance of the current account (by using **Deposit**).

The **BankAccount** class should be as follows:

```
class BankAccount
{
    ... additional code omitted for clarity ...

    public void TransferFrom(BankAccount accFrom, decimal
↪ amount)
    {
        if (accFrom. Withdraw(amount))
            this. Deposit(amount);
    }
}
```

4. Save and compile your code. Correct any errors.

✍ To test the **TransferFrom** method

1. Open the **Test** class. This is the test harness.
2. In the **Main** method, add code to create two **BankAccount** objects, each having an initial balance of \$100. (Use the **Populate** method.)
3. Add code to display the type, account number, and current balance of each account.
4. Add code to call **TransferFrom** and move \$10 from one account to the other.
5. Add code to display the current balances after the transfer.

The **Test** class could be as follows:

```
static void Main( )
{
    BankAccount b1 = new BankAccount( );
    b1.Populate(100);

    BankAccount b2 = new BankAccount( );
    b2.Populate(100);

    Console.WriteLine("Before transfer");
    Console.WriteLine("{0} {1} {2}",
        b1.Type( ), b1.Number( ), b1.Balance( ));
    Console.WriteLine("{0} {1} {2}",
        b2.Type( ), b2.Number( ), b2.Balance( ));

    b1.TransferFrom(b2, 10);

    Console.WriteLine("After transfer");
    Console.WriteLine("{0} {1} {2}",
        b1.Type( ), b1.Number( ), b1.Balance( ));
    Console.WriteLine("{0} {1} {2}",
        b2.Type( ), b2.Number( ), b2.Balance( ));
}
```

6. Save your work.
7. Compile the project and correct any errors. Run and test the program.

Exercise 2

Reversing a String

In Module 5, you developed a **Utils** class that contained a variety of utility methods.

In this exercise, you will add a new static method called **Reverse** to the **Utils** class. This method takes a string and returns a new string with the characters in reverse order.

✍ To create the Reverse method

1. Open the `Utils.sln` project in the *install folder*\Labs\Lab08\Starter\Utils folder.
2. Add a public static method called **Reverse** to the **Utils** class, as follows:
 - a. It has a single parameter called *s* that is a reference to a **string**.
 - b. The method has a **void** return type.
3. In the **Reverse** method, create a **string** variable called *sRev* to hold the returned string result. Initialize this string to `""`.
4. To create a reversed string:
 - a. Write a loop extracting one character at a time from *s*. Start at the end (use the **Length** property), and work backwards to the start of the string. You can use array notation (`[]`) to examine an individual character in a string.

Tip The last character in a string is at position **Length** – 1. The first character is at position 0.

- b. Append this character to the end of *sRev*.

The **Utils** class might contain the following:

```
class Utils
{
    ... additional methods omitted for clarity ...

    //
    // Reverse a string
    //

    public static void Reverse(ref string s)
    {
        int k;
        string sRev = "";

        for (k = s.Length - 1; k >= 0 ; k--)
            sRev = sRev + s[k];

        // Return result to caller
        s = sRev;
    }
}
```

✍ To test the Reverse method

1. Edit the **Test** class. This class contains the test harness.
2. In the **Main** method, create a **string** variable.
3. Read a value into the **string** variable by using **Console.ReadLine**.
4. Pass the string into **Reverse**. Do not forget the **ref** keyword.
5. Display the value returned by **Reverse**.

The **Test** class might contain the following:

```
static void Main( )
{
    string message;

    // Get an input string
    Console.WriteLine("Enter string to reverse:");
    message = Console.ReadLine( );

    // Reverse the string
    Utils.Reverse(ref message);

    // Display the result
    Console.WriteLine(message);
}
```

6. Save your work.
7. Compile the project and correct any errors. Run and test the program.

Exercise 3

Making an Uppercase Copy of a File

In this exercise, you will write a program that prompts the user for the name of a text file. The program will check that the file exists, displaying a message and quitting if it does not. The file will be opened and copied to another file (prompt the user for the file name), but with every character converted to uppercase.

Before you start, you might want to look briefly at the documentation for **System.IO** in the .NET Framework SDK Help documents. In particular, look at the documentation for the **StreamReader** and **StreamWriter** classes.

🔗 To create the file-copying application

1. Open the CopyFileUpper.sln project in the *install.folder*Labs\Lab08\Starter\CopyFileUpper folder.
2. Edit the **CopyFileUpper** class and add a **using** statement for the **System.IO** namespace.
3. In the **Main** method, declare two **string** variables called *sFrom* and *sTo* to hold the input and output file names.
4. Declare a variable of type **StreamReader** called *srFrom*. This variable will hold the reference to the input file.
5. Declare a variable of type **StreamWriter** called *swTo*. This variable will hold the reference to the output stream.
6. Prompt for the name of the input file, read the name, and store it in the **string** variable *sFrom*.
7. Prompt for the name of the output file, read the name, and store it in the **string** variable *sTo*.
8. The I/O operations that you will use can raise exceptions, so begin a **try-catch** block that can catch **FileNotFoundException** (for non-existent files) and **Exception** (for any other exceptions). Print out a meaningful message for each exception.
9. In the **try-catch** block, create a new **StreamReader** object using the input file name in *sFrom*, and store it in the **StreamReader** reference variable *srFrom*.
10. Similarly, create a new **StreamWriter** object using the input file name in *sTo*, and store it in the **StreamWriter** reference variable *swTo*.
11. Add a **while** loop that loops if the **Peek** method of the input stream does not return -1. Within the loop:
 - a. Use the **ReadLine** method on the input stream to read the next line of input into a **string** variable called *sBuffer*.
 - b. Perform the **ToUpper** method on *sBuffer*.
 - c. Use the **WriteLine** method to send *sBuffer* to the output stream.
12. After the loop has finished, close the input and output streams.

13. The CopyFileUpper.cs file should be as follows:

```
using System;
using System.IO;

class CopyFileUpper
{
    static void Main( )
    {
        string      sFrom, sTo;
        StreamReader srFrom;
        StreamWriter swTo;

        // Prompt for input file name
        Console.Write("Copy from ");
        sFrom = Console.ReadLine( );

        // Prompt for output file name
        Console.Write("Copy to: ");
        sTo = Console.ReadLine( );

        Console.WriteLine("Copy from {0} to {1}", sFrom,
            ↪sTo);

        try
        {
            srFrom = new StreamReader(sFrom);
            swTo    = new StreamWriter(sTo);

            while (srFrom.Peek( ) != -1)
            {
                string sBuffer = srFrom.ReadLine( );
                sBuffer = sBuffer.ToUpper( );
                swTo.WriteLine(sBuffer);
            }
            swTo.Close( );
            srFrom.Close( );

        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("Input file not found");
        }
        catch (Exception e)
        {
            Console.WriteLine("Unexpected exception");
            Console.WriteLine(e.ToString( ));
        }
    }
}
```

14. Save your work. Compile the project and correct any errors.

✍ To test the program

1. Open a Command window and go to the *install folder*\Labs\Lab08\Starter\CopyFileUpper\bin\debug folder.
2. Execute CopyFileUpper.
3. When prompted, specify a source file name of ***drive:\path\CopyFileUpper.cs***
(This is the source file you have just created.)
4. Specify a destination file of **Test.cs**
5. When the program is finished, use a text editor to examine the Test.cs file. It should contain a copy of your source code in all uppercase letters.

◆ Data Conversions

- Converting Value Types
- Parent/Child Conversions
- The is Operator
- The as Operator
- Conversions and the object Type
- Conversions and Interfaces
- Boxing and Unboxing

This section explains how to perform data conversions between reference types in C#. You can convert references from one type to another, but the reference types must be related.

In this section, you will learn about:

- Permitted and prohibited conversions between reference types.
- Conversion mechanisms (casts, **is**, and **as**).
- Special considerations for conversion to and from the **object** type.
- The reflection mechanism, which allows examination of run-time type information.
- Automatic conversions (boxing and unboxing) between value types and reference types.

Converting Value Types

- **Implicit Conversions**
- **Explicit Conversions**
 - Cast operator
- **Exceptions**
- **System.Convert Class**
 - Handles the conversions internally

C# supports implicit and explicit data conversions.

Implicit Conversions

For value types, you have learned about two ways to convert data: implicit conversion and explicit conversion using the cast operator.

Implicit conversion occurs when a value of one type is assigned to another type. C# only allows implicit conversion for certain combinations of types, typically when the first value can be converted to the second without any data loss. The following example shows how data is converted implicitly from **int** to **long**:

```
int a = 4;  
long b;  
b = a; // Implicit conversion of int to long
```

Explicit Conversions

You can explicitly convert value types by using the cast operator, as shown:

```
int a;  
long b = 7;  
a = (int) b;
```

Exceptions

When you use the cast operator, you should be aware that problems might occur if the value cannot be held in the target variable. If a problem is detected during an explicit conversion (such as trying to fit the value 9,999,999,999 into an **int** variable), C# might raise an exception (in this case, the **OverflowException**). If you want, you can catch this exception by using **try** and **catch**, as shown:

```
try {  
    a = checked((int) b);  
}  
catch (Exception) {  
    Console.WriteLine("Problem in cast");  
}
```

For operations that involve integers, use the **checked** keyword or compile with the appropriate compiler settings, otherwise checking will not be performed.

System.Convert Class

Conversions between the different base types (such as **int**, **long**, and **bool**) are handled within the .NET Framework by the **System.Convert** class.

You do not usually need to make calls to methods of **System.Convert**. The compiler handles these calls automatically.

Parent/Child Conversions

- **Conversion to Parent Class Reference**
 - Implicit or explicit
 - Always succeeds
 - Can always assign to object
- **Conversion to Child Class Reference**
 - Explicit casting required
 - Will check that the reference is of the correct type
 - Will raise **InvalidCastException** if not

You can convert a reference to an object of a child class to an object of its parent class, and vice versa, under certain conditions.

Conversion to Parent Class Reference

References to objects of one class type can be converted into references to another type if one class inherits from the other, either directly or indirectly.

A reference to an object can always be converted to a reference to a parent class object. This conversion can be performed implicitly (by assignment or as part of an expression) or explicitly (by using the cast operator).

The following examples will use two classes: **Animal** and **Bird**. **Animal** is the parent class of **Bird**, or, to put it another way, **Bird** inherits from **Animal**.

The following example declares a variable of type **Animal** and a variable of type **Bird**:

```
Animal a;  
Bird b;
```

Now consider the following assignment, in which the reference in *b* is copied to *a*:

```
a = b;
```

The **Bird** class inherits from the **Animal** class. Therefore, a method that is found in **Animal** is also found in **Bird**. (The **Bird** class might have overridden some of the methods of **Animal** to create its own version of them, but an implementation of the method will exist nonetheless.) Therefore, it is possible for references to **Bird** objects to be assigned to variables containing references to values of type **Animal**.

In this case, C# performs a type conversion from **Bird** to **Animal**. You can explicitly convert **Bird** to **Animal** by using a cast operator, as shown:

```
a = (Animal) b;
```

The preceding code will produce exactly the same result.

Conversion to Child Class Reference

You can convert a reference to a child type, but you must explicitly specify the conversion by using a cast. An explicit conversion is subject to run-time checking to ensure that the types are compatible, as shown in the following example:

```
Bird b = (Bird) a; // Okay
```

This code will compile successfully. At run time, the cast operator performs a check to determine whether the value in the variable really is of type **Bird**. If it is not, the run-time **InvalidCastException** is raised.

If you attempt to assign to a child type without a conversion operator, as in the following code, the compiler will display an error message stating, “Cannot convert implicitly type ‘Animal’ to type ‘Bird.’”

```
b = a; // Will not compile
```

You can trap a type conversion error by using **try** and **catch**, just like any other exception, as shown in the following code:

```
try {  
    b = (Bird) a;  
}  
catch (InvalidCastException) {  
    Console.WriteLine("Not a bird");  
}
```

The is Operator

- Returns true If a Conversion Can Be Made



```
Bird b;  
if (a is Bird)  
    b = (Bird) a; // Safe  
else  
    Console.WriteLine("Not a Bird");
```

You can handle incompatible types by catching **InvalidCastException**, but there are other ways of handling this problem, such as the **is** operator.

You can use the **is** operator to test the type of the object without performing a conversion. The **is** operator returns **true** if the value on the left is not **null** and a cast to the class on the right, if performed, would complete without throwing an exception. Otherwise, **is** returns **false**.

```
if (a is Bird)  
    b = (Bird) a; // Safe, because "a is Bird" returns true  
else  
    Console.WriteLine("Not a Bird");
```

You can think of the relationship between inherited classes as an “is a kind of” relationship, as in “A bird is a kind of animal.” References in the variable *a* must be references to **Animal** objects, and *b* is a kind of animal. Of course, *b* is a bird as well, but a bird is just a special case of an animal. The converse is not true. An animal is not a type of bird. Some animals are birds, but it is not true that all animals are birds.

So the following expression can be read as “If *a* is a kind of bird,” or “If *a* is a bird or a type derived from bird.”

```
if (a is bird)
```

The as Operator

- Converts Between Reference Types, Like Cast
- On Error
 - Returns null
 - Does not raise an exception

```
Bird b = a as Bird; // Convert  
  
if (b == null)  
    Console.WriteLine("Not a bird");
```

You can use the **as** operator to perform conversions between types.

Example

The following statement performs a conversion of the reference in *a* to a value that references a class of type **Bird**, and the runtime automatically checks to ensure that the conversion is acceptable.

```
b = a as Bird;
```

Error Handling

The **as** operator differs from the cast operator in the way it handles errors. If, in the preceding example, the reference in variable *a* cannot be converted in a reference to an object of class **Bird**, the value **null** is stored in *b*, and the program continues. The **as** operator never raises an exception.

You can rewrite the previous code as follows to display an error message if the conversion cannot be performed:

```
Bird b = a as Bird;  
if (b == null)  
    Console.WriteLine("Not a bird");
```

Although **as** never raises an exception, any attempt to access through the converted value will raise a **NullReferenceException** if it is **null**. Therefore, you should always check the return value from **as**.

Conversions and the object Type

- The **object** Type Is the Base for All Classes
- Any Reference Can Be Assigned to **object**
- Any **object** Variable Can Be Assigned to Any Reference
 - With appropriate type conversion and checks
- The **object** Type and is

```
object ox;  
ox = a;  
ox = (object) a;  
ox = a as object;
```

```
b = (Bird) ox;  
b = ox as Bird;
```

All reference types are based on the **object** type. This means that any reference can be stored in a variable of type **object**.

The **object** Type Is the Base for All Classes

The **object** type is the base for all reference types.

Any Reference Can Be Assigned to **object**

Because all classes are based directly or indirectly on the **object** type, you can assign any reference to a variable of type **object**, either with an implicit conversion or with a cast. The following code provides an example:

```
object ox;  
ox = a;  
ox = (object) a;  
ox = a as object;
```

Any **object** Variable Can Be Assigned to Any Reference

You can assign a value of type **object** to any other object reference, if you cast it correctly. Remember that the run-time system will perform a check to ensure that the value being assigned is of the correct type. The following code provides an example:

```
b = (Bird) ox;  
b = ox as Bird;
```

The preceding examples can be written with full error checking as follows:

```
try {  
    b = (Bird) ox;  
}  
catch (InvalidCastException) {  
    Console.WriteLine("Cannot convert to Bird");  
}  
b = ox as Bird;  
if (b == null)  
    Console.WriteLine("Cannot convert to Bird");
```

The object Type and is

Because every value is derived ultimately from **object**, checking a value with the **is** operator to see if it is an **object** will always return **true**.

```
if (a is object) // Always returns true
```

Conversion and Interfaces

- **An Interface Can Only Be Used to Access Its Own Members**
- **Other Methods and Variables of the Class Are Not Accessible Through the Interface**

You can perform conversions by using the casting operators, **as** and **is**, when working with interfaces.

For example, you can declare a variable of an interface type, as shown:

```
IHashCodeProvider hcp;
```

Converting a Reference to an Interface

You can use the cast operator to convert the object reference into a reference to a given interface, as shown:

```
IHashCodeProvider hcp;  
hcp = (IHashCodeProvider) x;
```

As with conversion between class references, the cast operator will raise an **InvalidCastException** if the object provided does not implement the interface. You should determine whether an object supports an interface before casting the object, or use **try** and **catch** to trap the exception.

Determining Whether an Interface Is Implemented

You can use the **is** operator to determine whether an object supports an interface. The syntax is the same as the syntax used for classes:

```
if (x is IHashCodeProvider) ...
```

Using the as Operator

You can also use the **as** operator as an alternative to casting, as shown:

```
HashCodeProvider hcp;  
hcp = x as IHashCodeProvider;
```

As with conversion between classes, if the reference that is being converted does not support the interface, the **as** operator returns **null**.

After you have converted a reference to a class into a reference to an interface, the new reference can only access members of that interface, and cannot access the other public members of the class.

Example

Consider the following example to learn how converting references to interfaces works. Suppose you have created an interface called **IVisual** that specifies a method called **Paint**, as follows:

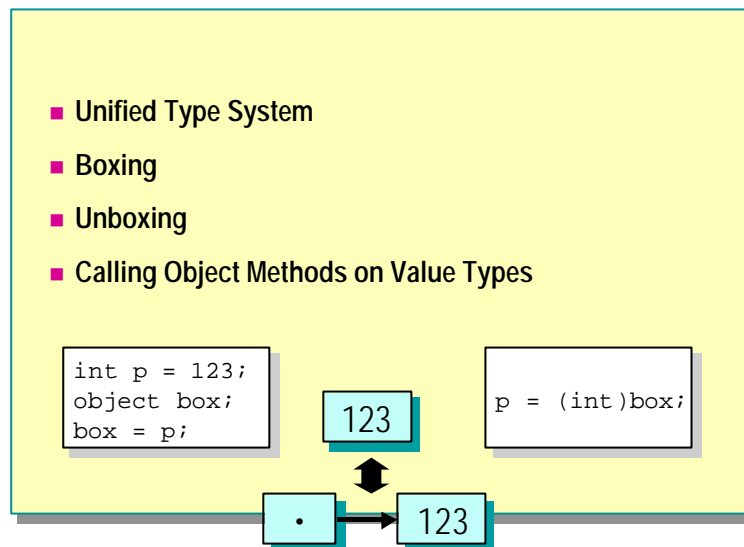
```
interface IVisual  
{  
    void Paint( );  
}
```

Suppose that you also have a **Rectangle** class that implements the **IVisual** interface. It implements the **Paint** method, but it can also define its own methods. In this example, **Rectangle** has defined an additional method called **Move** that is not part of **IVisual**.

You can create a **Rectangle**, *r*, and use its **Move** and **Paint** methods, as you would expect. You can even reference it through an **IVisual** variable, *v*. However, despite the fact that *v* and *r* both refer to the same object in memory, you cannot call the **Move** method by using *v* because it is not part of the **IVisual** interface. The following code provides examples:

```
Rectangle r = new Rectangle( );  
r.Move( );           // Okay  
r.Paint( );          // Okay  
IVisual v = (IVisual) r;  
v.Move( );           // Not valid  
v.Paint( );          // Okay
```

Boxing and Unboxing



C# can convert value types into object references and object references into value types.

Unified Type System

C# has a unified type system that allows value types to be converted to references of type **object** and object references to be converted into value types. Value types can be converted into references of type **object**, and vice versa.

Values of types like `int` and `bool` can therefore be handled as simple values most of the time. This is normally the most efficient technique because there is none of the overhead that is associated with references. However, when you want to use these values as if they were references, they can be temporarily *boxed* for you to do so.

Boxing

Expressions of value types can also be converted to values of type **object**, and back again. When a variable of value type needs to be converted to **object** type, an object *box* is allocated to hold the value and the value is copied into the box. This process is known as *boxing*.

```
int p = 123;
object box;
box = p;           // Boxing (implicit)
box = (object) p;  // Boxing (explicit)
```

The boxing operation can be done implicitly, or explicitly with a cast to an object. Boxing occurs most typically when a value type is passed to a parameter of type **object**.

Unboxing

When a value in an object is converted back into a value type, the value is copied out of the box and into the appropriate storage location. This process is known as *unboxing*.

```
p = (int) box;    // Unboxing
```

You must perform unboxing with an explicit cast operator.

If the value in the reference is not the exact type of the cast, the cast will raise an **InvalidCastException**.

Calling Object Methods on Value Types

Because boxing can take place implicitly, you can call methods of the object type on any variable or expression, even those having value types. The following code provides an example:

```
static void Show(object o)
{
    Console.WriteLine(o.ToString());
}
Show(42);
```

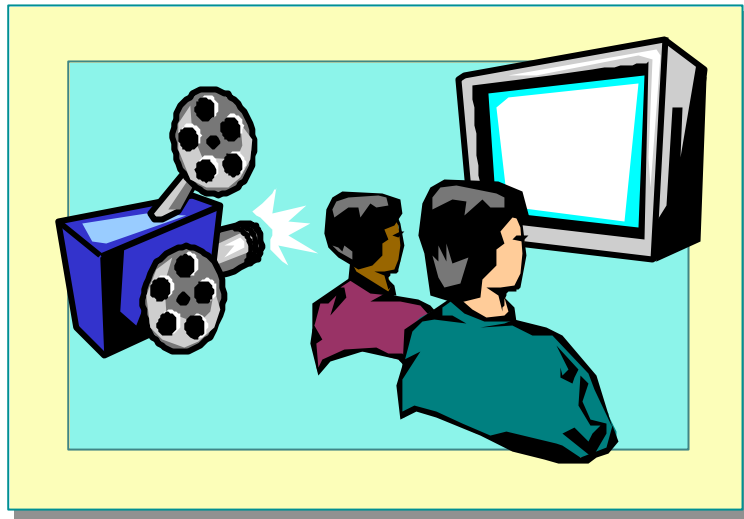
This works because the value 42 is implicitly boxed into an **object** parameter, and the **ToString** method of this parameter is then called.

It produces the same result as the following code:

```
object o = (object) 42; // Box
Console.WriteLine(o.ToString());
```

Note Boxing does *not* occur when you call **Object** methods *directly* on a value. For example, the expression `42.ToString()` does not box 42 into an **object**. This is because the compiler can statically determine the type and discerns which method to call.

Multimedia: Type-Safe Casting



Lab 8.2 Converting Data



Objectives

After completing this lab, you will be able to:

- Convert values of one reference type to another.
- Test whether a reference variable supports a given interface.

Prerequisites

Before working on this lab, you should be familiar with the following:

- Concepts of object-oriented programming
- Creating classes
- Defining methods

Estimated time to complete this lab: 30 minutes

Exercise 1

Testing for the Implementation of an Interface

In this exercise, you will add a static method called **IsItFormattable** to the **Utils** class that you created in Lab 5. If you did not complete that lab, you can obtain a copy of the class in the *install folder*\Labs\Lab08\Starter folder.

The **IsItFormattable** method takes one parameter of type **object** and tests whether that parameter implements the **System.IFormattable** interface. If the object does have this interface, the method will return **true**. Otherwise, it will return **false**.

A class implements the **System.IFormattable** interface to return a string representation of an instance of that class. Base types such as **int** and **ulong** implement this interface (after the value has been boxed). Many reference types, for example **string**, do not. User-defined types can implement the interface if the developer requires it. For more information about this interface, consult the .NET Framework SDK Help documentation.

You will write test code that will call the **Utils.IsItFormattable** method with arguments of different types and display the results on the screen.

✎ To create the **IsItFormattable** method

1. Open the InterfaceTest.sln project in the *install folder*\Labs\Lab08\Starter\InterfaceTest folder.
2. Edit the **Utils** class as follows:
 - a. Create a public static method called **IsItFormattable** in the **Utils** class.
 - b. This method takes one parameter called *x* of type **object** that is passed by value. The method returns a **bool**.
 - c. Use the **is** operator to determine whether the passed object supports the **System.IFormattable** interface. If it does, return **true**; otherwise return **false**.

The completed method should be as follows:

```
using System;

...

class Utils
{
    public static bool IsItFormattable(object x)
    {
        // Use the is operator to test whether the
        // object has the IFormattable interface

        if (x is IFormattable)
            return true;
        else
            return false;
    }
}
```

✍ To test the `IsItFormattable` method

1. Edit the file `Test` class.
2. In the `Main` method, declare and initialize variables of types `int`, `ulong`, and `string`.
3. Pass each variable to `Utils.IsItFormattable()`, and print the result from each call.
4. The class `Test` might be as follows:

```
using System;
class Test
{
    static void Main( )
    {
        int i=0;
        ulong ul=0;
        string s = "Test";

        Console.WriteLine("int: {0}",
        ↪Utils.IsItFormattable(i));
        Console.WriteLine("ulong: {0}",
        ↪Utils.IsItFormattable(ul));
        Console.WriteLine("String: {0}",
        ↪Utils.IsItFormattable(s));
    }
}
```

5. Compile and test the code. You should see `true` for the `int` and `ulong` values, and `false` for the `string` value.

Exercise 2

Working with Interfaces

In this exercise, you will write a **Display** method that will use the **as** operator to determine whether the object passed as a parameter supports a user-defined interface called **IPrintable** and call a method of that interface if it is supported.

✎ To create the Display method

1. Open the TestDisplay.sln project in the *install folder* \ Labs\Lab08\Starter\TestDisplay folder.

The starter code includes the definition for an interface called **IPrintable**, which contains a method called **Print**. A class that implements this interface should use the **Print** method to display to the console the values held inside the object. Also defined in the starter code files is a class called **Coordinate** that implements the **IPrintable** interface.

A **Coordinate** object holds a pair of numbers that can define a position in two-dimensional space. You do not need to understand how the **Coordinate** class works (although you might want to look at it). All you need to know is that it implements the **IPrintable** interface and that you can use the **Print** method to display its contents.

2. Edit the **Utils** class as follows:
 - a. Add a public static **void** method called **Display** in the **Utils** class. This method should take one parameter, an **object** passed by value, called *item*.
 - b. In **Display**, declare an interface variable called *ip* of type **IPrintable**.
 - c. Convert the reference in the parameter *item* into a reference to the **IPrintable** interface that uses the **as** operator. Store the result in *ip*.
 - d. If the value of *ip* is not **null**, use the **IPrintable** interface to call **Print**. If it is **null**, the object does not support the interface. In this case, use **Console.WriteLine** to display to results of the **ToString** method on the parameter instead.

The completed method should be as follows:

```
public static void Display(object item)
{
    IPrintable ip;

    ip = (item as IPrintable);

    if (ip != null)
        ip.Print( );
    else
        Console.WriteLine(item.ToString( ));
}
```

✍ To test the Display method

1. Within the **Main** method in the **Test** class, create a variable of type **int**, a variable of type **string**, and a variable of type **Coordinate**. To initialize the **Coordinate** variable, you can use the two-parameter constructor:

```
Coordinate c = new Coordinate(21.0, 68.0);
```

2. Pass these three variables, in turn, to **Utils.Display** to print them out.
3. The code should be as follows:

```
class Test  
{  
    static void Main( )  
    {  
        int num = 65;  
        string msg = "A String";  
        Coordinate c = new Coordinate(21.0, 68.0);  
  
        Utils.Display(num);  
        Utils.Display(msg);  
        Utils.Display(c);  
    }  
}
```

4. Compile and test your application.

If Time Permits

Testing the Method

If you want to try the **IsItFormattable** method that you created in Exercise 1 with a user-defined class, use the **BankAccount** class that you developed in a previous lab.

Re-write the **Display** method from Exercise 2 by using the cast operator. Remember to catch any **InvalidCastException** that C# might throw in response to errors.

Review

- Using Reference-Type Variables
- Using Common Reference Types
- The Object Hierarchy
- Namespaces in the .NET Framework
- Data Conversions

-
1. Explain how a memory is allocated and de-allocated for a variable of reference type.
 2. What special value indicates that a reference variable does not contain a reference to an object? What happens if you try to access a reference variable with this value?
 3. List the key features of the **String** class.
 4. What type is the base type for all classes?

5. Explain the difference between the cast operator and the **as** operator when used to convert between class references.

6. List ways in which you can determine the type of an **object**.