Programming in C#

Trainer: Venkat Shiva Reddy

01 - Basics

Goals:

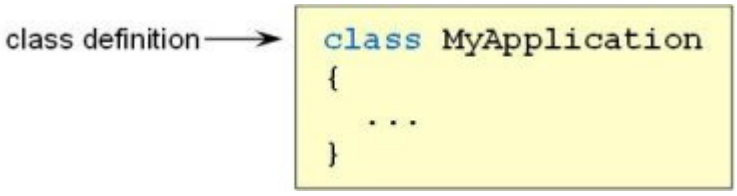
- Introduce the basics of a class.
- Describe the `Main` method.
- Survey types, variables, arrays, operators, control constructs, and comments.
- Show how to perform text based input and output.

Overview

We will now introduce the core of the C# language; that is, we will cover the features that every program will need to use. Most of this section will be a relatively straightforward examination of syntax so you may need to practice a bit of delayed gratification - we'll get to cooler stuff very soon.

Class

A class is the primary way for the C# developer to create a new data type. The definition syntax is the keyword `class`, the class name, and then curly braces enclosing the class body.

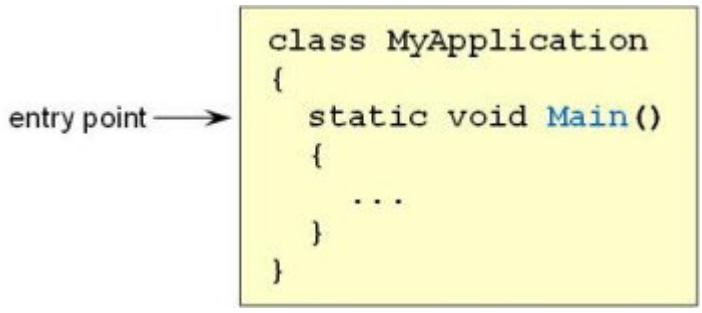


A class typically contains two types of things: variables (i.e. data) and methods (i.e. functions). All variables and methods must be members of some type because the language does not allow global variables or global methods. For now, we will keep things simple and create classes that contain just the `Main` method. We will expand on the capabilities of a class in later chapters.

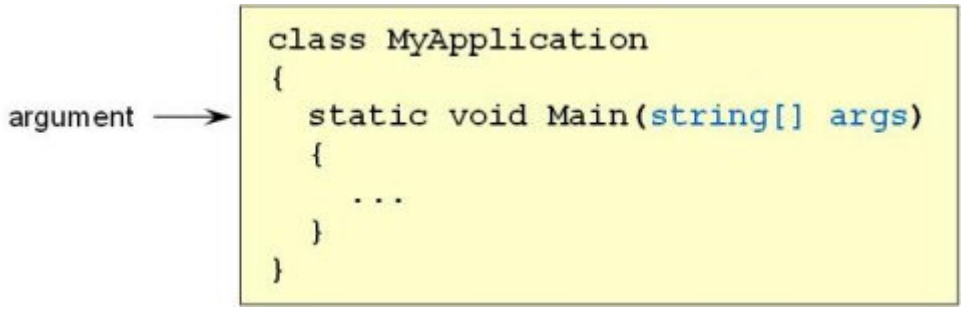
Note that there are several other ways to create new types: `struct`, `enum`, `delegate`, and `interface`; however, each of these are more specialized than `class`.

Main method

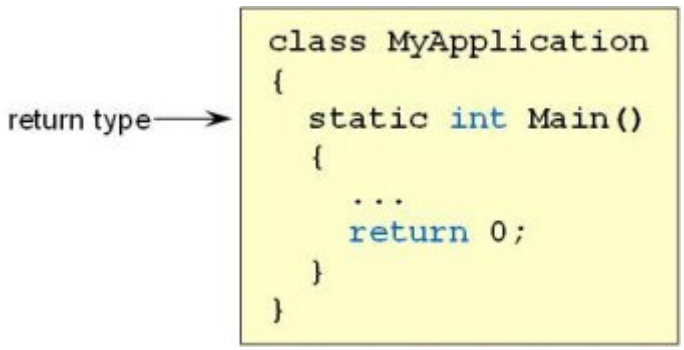
The `Main` method defines the application entry point. The name is case sensitive so the capital `M` is required. `Main` must be a `static` method of some class. We will discuss the meaning of `static` in a later chapter.



In addition to serving as the application entry point, `Main` can also interact with the environment. `Main` can take an array of strings as its argument. The strings will be obtained from the command line when the program is invoked.



`Main` can also return an `int` code back to the invoking environment. The return code is most often used to indicate the success or failure of the program execution. Traditionally, a return value of zero indicates success while non-zero values are used as error codes to indicate failure.



Environment interaction is not extremely common in these days of graphical user

interfaces. However, some application categories such as development tools (compilers and linkers) still use command line interfaces.

Simple types

The set of core types offered by C# are called the simple types. These are analogous to what other languages call built-in types or primitive types.

	Type	Description	Special format for literals
Boolean	bool	Boolean	true false
character	char	16 bit Unicode character	'A' '\x0041' '\u0041'
integer	sbyte	8 bit signed integer	none
	byte	8 bit unsigned integer	none
	short	16 bit signed integer	none
	ushort	16 bit unsigned integer	none
	int	32 bit signed integer	none
	uint	32 bit unsigned integer	U suffix
	long	64 bit signed integer	L or l suffix
	ulong	64 bit unsigned integer	U/u and L/l suffix
floating point	float	32 bit floating point	F or f suffix
	double	64 bit floating point	no suffix
	decimal	128 bit high precision	M or m suffix
string	string	character sequence	"hello"

There is a Boolean type named `bool` which can take on only two values: `true` and `false`.

The character type is 16 bit Unicode. Character literals are enclosed in single quotes; for example, a capital A would be written as `'A'`. A less common but interesting special case allows a character to be specified by giving the hexadecimal or Unicode value. The capital A character has a Unicode value of 65 (hex `0041`) so it could be written as either `'\u0041'` or `'\x0041'`.

There are several different flavors of integer that vary by size and by whether they are signed or unsigned. Signed types can store both positive and negative values while unsigned types store only positive numbers. Common practice is to use `int` as the primary integer type since 32 bits is usually large enough for most needs and a signed quantity is typically preferred.

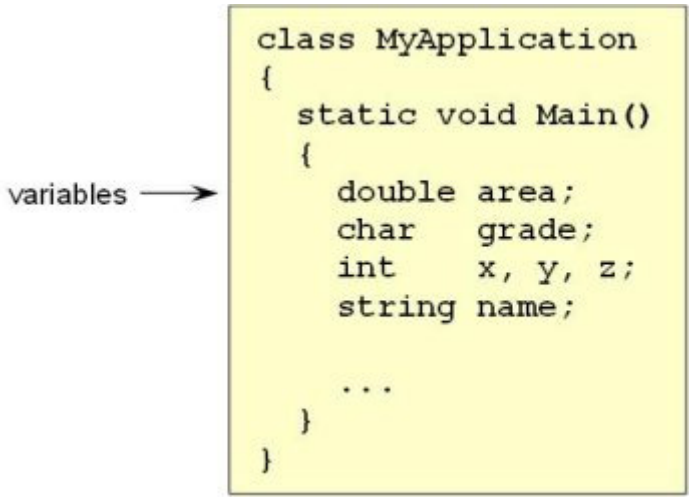
There are three different sizes of floating point types: `float`, `double`, and `decimal`. The 64-bit type `double` is the most commonly used of the three. The `float` type is only 32

bits and so is often considered too small for any application that does serious floating point arithmetic. The `decimal` type provides extremely high precision at the expense of range and so is intended for specialized financial and scientific calculations.

The `string` type represents a sequence of characters. String literals are enclosed in double quotes.

Local variables

Local variables are declared inside a method and are only accessible inside the method body. The declaration syntax is type name followed by comma separated list of variables with a semicolon ending the statement.



Local variables can optionally be given an initial value when they are declared. The initialization syntax is the assignment operator `=` followed by the initial value. The initial value can be a literal or an expression.

literals → int width = 2; int height = 4;

expression → int area = width * height;

```
class MyApplication
{
  static void Main()
  {
    int width  = 2;
    int height = 4;

    int area = width * height;

    ...
  }
}
```

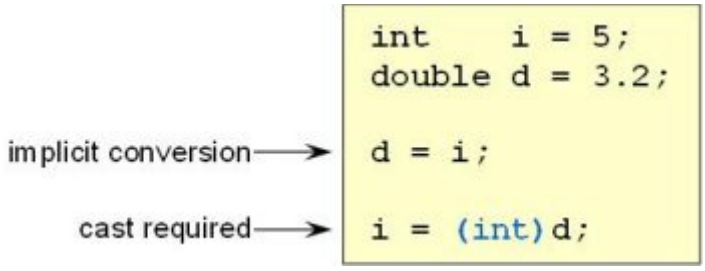
Local variables must be assigned to before they can be used. The compiler tracks the state of all local variables and rejects any attempt to use them before they have been given a value. This is an extremely nice service provided by the compiler and should help to eliminate an entire category of program bugs.

error, x not set → int y = x * 2;

```
class MyApplication
{
  static void Main()
  {
    int x;
    int y = x * 2;

    ...
  }
}
```

Type conversion

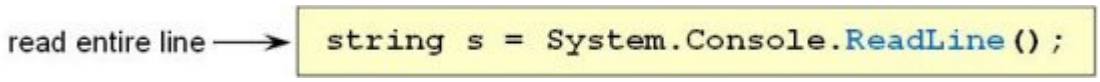
It is occasionally necessary to convert a value from one simple type to another; for example, converting from `int` to `long` or from `double` to `int`. These conversions fall into two categories: implicit and explicit. An implicit conversion happens automatically and there is no need for any special action on the part of the programmer. On the other hand, an explicit conversion requires that the programmer explicitly request the change by adding a cast to their code. The cast syntax is the destination type name enclosed in parentheses. In general, conversions that could lose information require an explicit cast. For example, a cast is needed to convert `double` to `float`, `float` to `int`, or `long` to `int`.



Console input and output

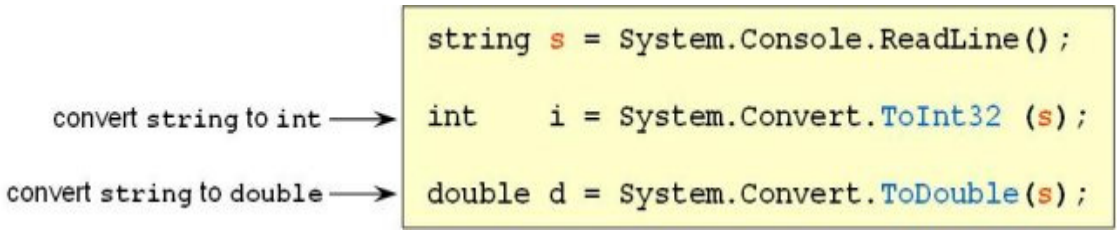
C# console applications interact with the user through a text based console window. The .NET Framework Class Library provides a class named `Console` to facilitate console IO. The `Console` class is part of the `System` namespace.

The simplest way to read input is using the `ReadLine` method which reads an entire line of input at once. The result is a string containing the entire input line.

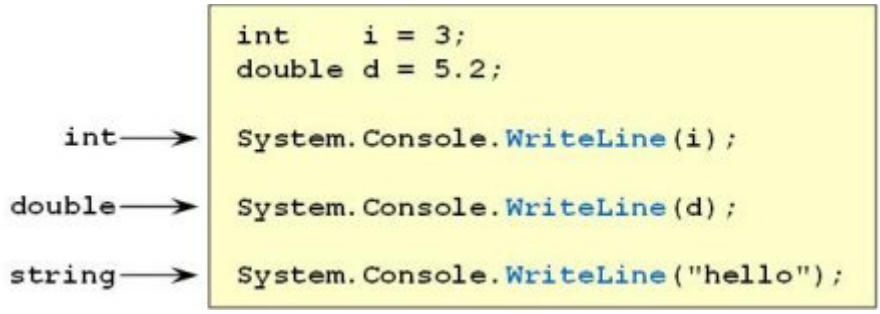


If more control over input is needed, the `Read` method can be used to read each input character one at a time.

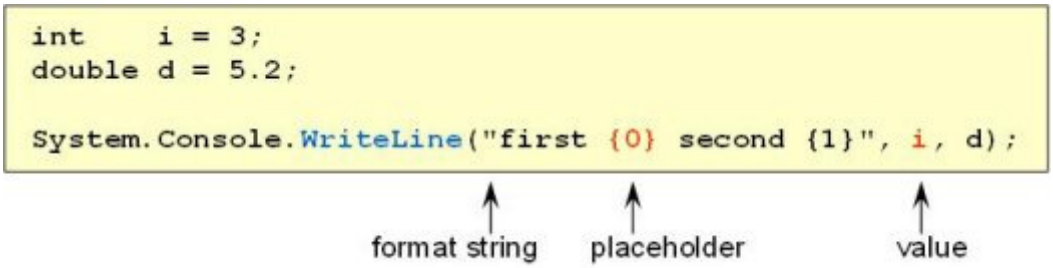
After reading the user input, the next step is to get it into the desired form. If the user is entering a string value such as their name, then you are done since the input data is already a string. However, if the user is entering a numeric value such as their age or salary, then the input string must be converted to a numeric form. The .NET Framework Class Library supplies a class named `Convert` that can perform most commonly needed conversions. The `Convert` class is in the `System` namespace.



The `Console` class also supplies output methods. There are several methods named `WriteLine` that write various data types to the console. In addition to writing the data, `WriteLine` adds a line terminator to the output so any subsequent output will appear on a new line.



There is a fancier version of `WriteLine` that allows multiple values to be printed in a single call. The first argument is a format string which contains some literal text and some special symbols called format specifiers. The format specifiers act as placeholders and will be replaced with actual values during printing. A format specifier is written as an integer inside curly braces (e.g. `{0}` or `{1}`). After the format string come additional arguments, typically one for each format specifier in the format string. The additional arguments are numbered starting at zero. During printing, the format specifiers are replaced with the value of the corresponding argument.



The `Console` class also offers a set of methods name `Write` that work just like `WriteLine` except that they do not add a line terminator to the output.

Namespace

A namespace represents a logical group of types. The .NET Framework Class Library makes heavy use of namespaces to partition it into smaller pieces that are easier for users to handle. The core of the library is gathered into a namespace called `System`. To access the members of a namespace, you use the namespace name and then the dot operator as a separator.

```
string s = System.Console.ReadLine();

int    i = System.Convert.ToInt32(s);
```

access `System` namespace

Prefixing each use of a library component with the namespace name quickly becomes tedious and makes the code unnecessarily verbose. A `using` directive can be used to obtain shorthand access to the components of a namespace. The `using` directive is typically placed at the top of source file. The syntax consists of the keyword `using`, the namespace name, and a semicolon.

using directive →

short names ↔

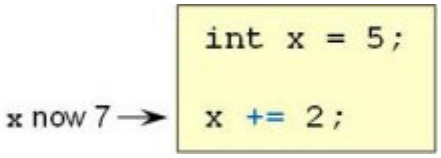
```
using System;

class MyApplication
{
  static void Main()
  {
    string s = Console.ReadLine();

    int i = Convert.ToInt32(s);

    ...
  }
}
```

Arithmetic operators

A comprehensive set of arithmetic operators are available. The standard `+`, `-`, `*`, and `/` operators are present. A slightly more interesting offering is the remainder operator (`%`) which computes the remainder of integer division. In addition, there are some specialized operators which manipulate the binary representation of a variable

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
%	remainder
<<	left shift
>>	right shift
&	bitwise and
|	bitwise or
^	bitwise xor
~	bitwise complement

Each arithmetic operator has a corresponding combination assignment version that does two things at once: it performs the arithmetic and assigns the result. For example, the combination operator for addition is +=.

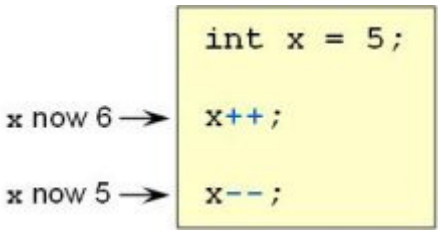


The designers of the C language family (C, C++, Java, C#) seem to have agreed that providing such combination operators is desirable since there are quite a few of them available.

Operator	Description
+=	add / assign
-=	subtract / assign
*=	multiply / assign
/=	divide / assign
%=	remainder / assign
<<=	left shift / assign
>>=	right shift / assign
&=	bitwise and / assign
|=	bitwise or / assign
^=	bitwise xor / assign

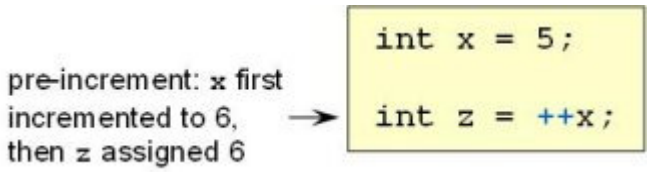
There are two specialized operators to perform the increment (++) and decrement (--) operations. The basic use of these operators is extremely simple: they add or subtract 1

from the variable to which they are applied.

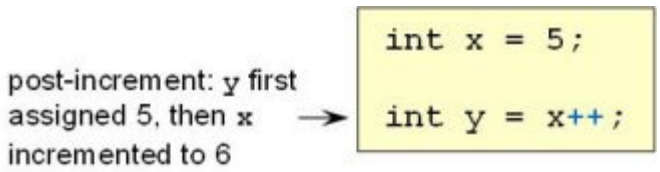


It turns out that the increment and decrement operators are slightly more powerful (and subtle) then they at first appear. The operators can actually be placed on either side of a variable and they have a slightly different effect in the two different positions. This added ability only comes into play when the increment/decrement is used as part of a larger expression. In that case, the position of the operator determines when the increment/decrement is performed.

When the operator is placed in front of the variable the operation is performed first, before the variable is used in the larger expression. This is called pre-increment or pre-decrement.



When the operator is placed after the variable the operation is performed second; that is, after the variable is used in the larger expression. This is called post-increment or post-decrement.



Finally, there is the somewhat cryptic conditional operator `?:` that provides a shorthand way of writing an if-then-else statement. The `?` and the `:` serve as delimiters between the three operands. The first operand is a Boolean expression. The Boolean expression is evaluated, if true, the result is the expression after the `?` otherwise the result is the expression after the `:`.

```
int x = 5;
int y = 3;
int max;

max = x > y ? x : y;
```

if x > y then the result is x else the result is y

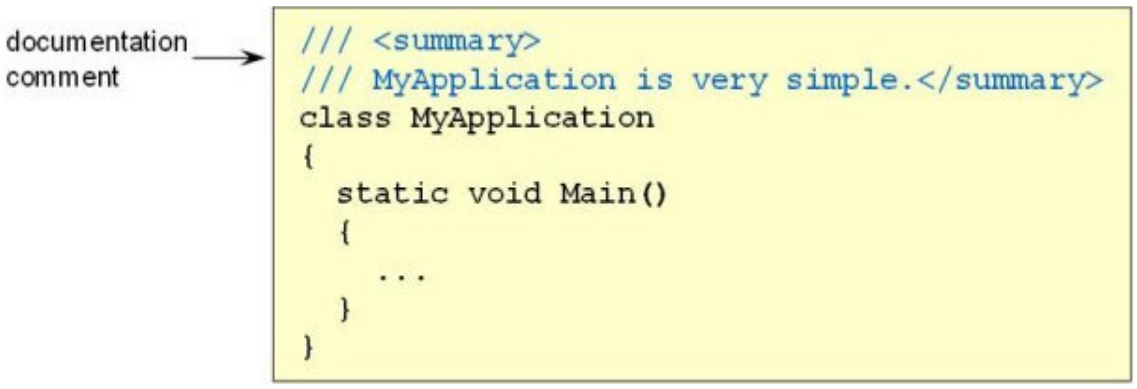
Comments

There are two primary comment styles: multi line and single line. A multi line comment is delimited beginning with `/*` and ending with `*/`. A single line comment begins with `//` and extends to the end of the line.

delimited →
single line →

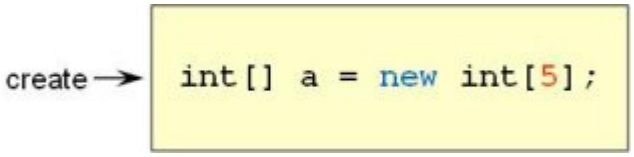
```
class MyApplication
{
  static void Main()
  {
    /* delimited comment can extend
       across multiple lines */

    int x; // comment goes to end of line
    ...
  }
}
```

There is one additional type of comment that you might see called a documentation comment. The documentation comment contains XML that serves to document the code for users. The C# compiler provides a `/doc` switch to build an XML file from the documentation comments.

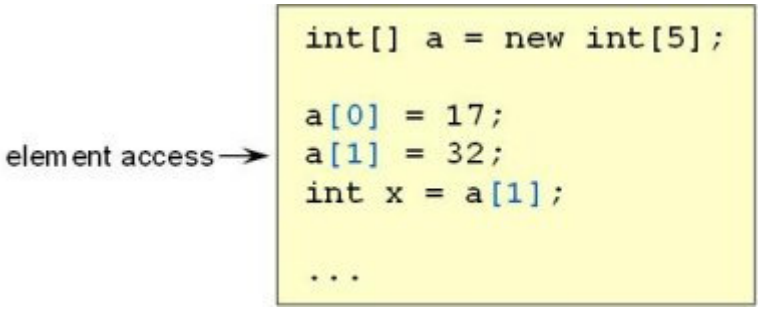


Array

Arrays provide efficient storage for multiple data elements. An array is specified using square brackets `[ ]` and must be created using the `new` operator. To create an array, both the element type and the length must be specified.



Array elements are accessed using `[ ]` and the desired index. Indices start at 0, so the valid indices for a length 5 array are 0, 1, 2, 3, 4. Bounds checking is performed to prevent access to memory outside the array. If an invalid index is used, the CLR traps the error and generates an exception of type `IndexOutOfRangeException`. The bounds checking is performed at runtime and does add a small amount of overhead; however, the designers of the CLR thought that the added program safety and robustness were well worth the time.



The length of an array is fixed at the point it is created and it cannot be resized. For convenience, the length of an array is recorded in its `Length` property which can be accessed using the dot operator.

number of elements →

```
int[] a = new int[5];

...

int l = a.Length;
```

When an array is first created, each element is set to a default value based on its type. For all the numeric types such as `int`, `double`, etc. the default value is 0. For other types the story is a bit more interesting: an array of `bool` has each element set to `false` and an array of characters starts with each element set to the null character (`0x0000`).

---

if else statement

The primary way to perform conditional execution is with an `if` statement. An `if` statement has an associated Boolean expression and a statement: if the Boolean expression is true then the statement is executed; otherwise, the statement is skipped. C# is case sensitive so `if` must appear in lower case. Also note that the parentheses around the Boolean expression are required.

if statement →

```
int  temperature = 127;
bool boiling     = false;

if (temperature >= 100)
  boiling = true;
```

An `if` statement can optionally contain an `else` part. The statement associated with the `else` part gets executed only if the expression is false.

else part →

```
int x = 3;
int y = 5;
int min;

if (x < y)
  min = x;
else
  min = y;
```

Boolean operators

Boolean expressions are built using the comparison and logical operators. The comparison operators provide ways to test common properties such as equal, greater than, less than, etc. The logical operators perform the Boolean logic functions and, or, not.

==	equal
!=	not equal
<, <=	less
>, >=	greater
&&	and
||	or
!	not

both must be true →

either can be true →

c can't be a letter →

```
int  x;
char c;

if (x > 0 && x < 10)
  ...

if (c == 'y' || c == 'Y')
  ...

if (!Char.IsLetter(c))
  ...
```

Code block

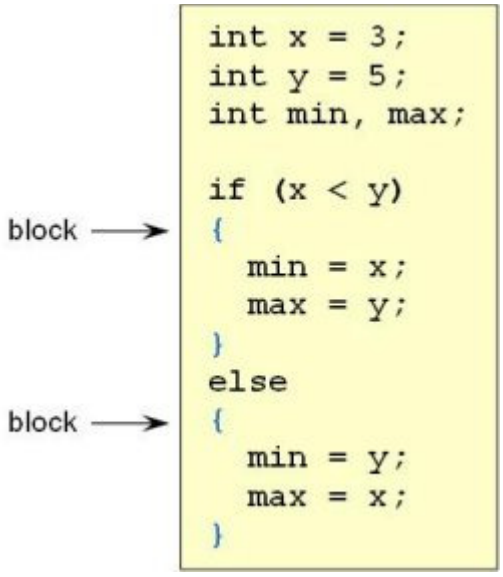
Most of the control constructs such as the `if` statement and the loops permit only a single statement as their body. Placing multiple statements is an error.

error →

error →

```
int x = 3;
int y = 5;
int min, max;

if (x < y)
  min = x;
  max = y;
else
  min = y;
  max = x;
```

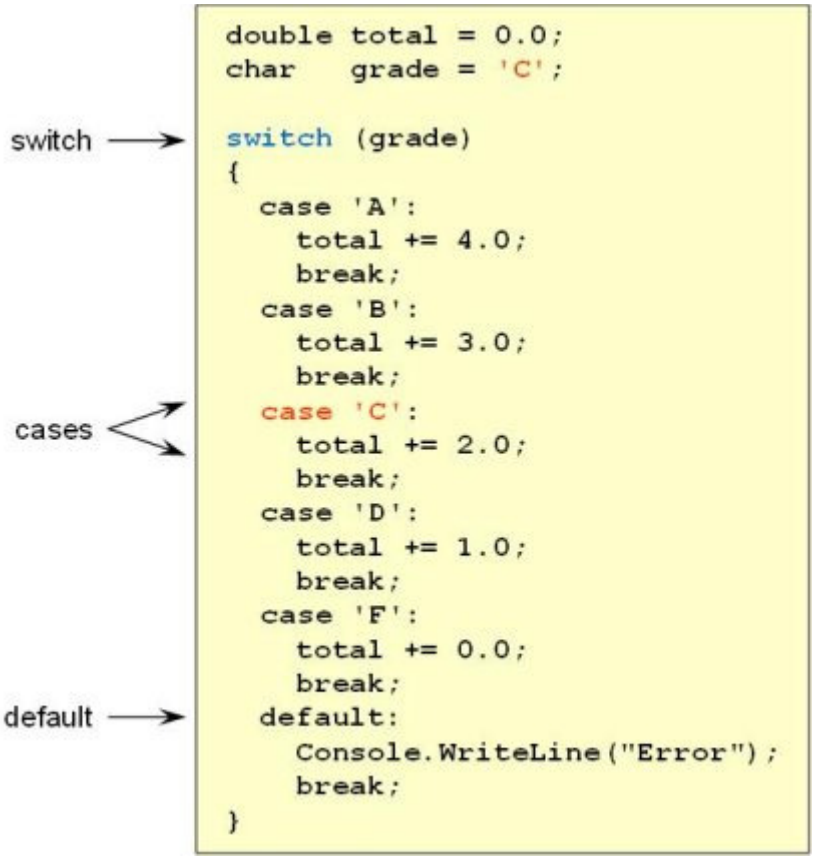
In order to associate multiple actions with a control construct, the statements must be placed inside curly braces. The curly braces form what is sometimes called a compound statement or a code block.



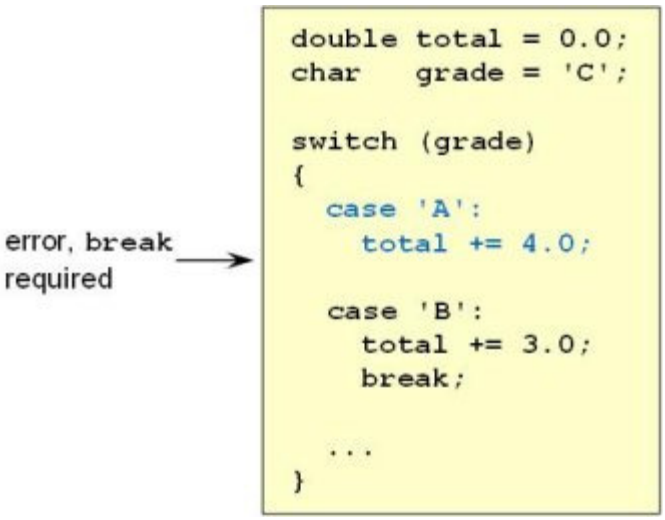

```
int x = 3;
int y = 5;
int min, max;

if (x < y)
{
  min = x;
  max = y;
}
else
{
  min = y;
  max = x;
}
```

switch statement

A `switch` statement performs selection from a fixed set of options. It consists of an expression followed by a set of cases. The expression is evaluated and the result compared against each case. If a match is found, the code associated with that case is executed. If no match is found, the `default` case is executed. The `default` case is completely optional.

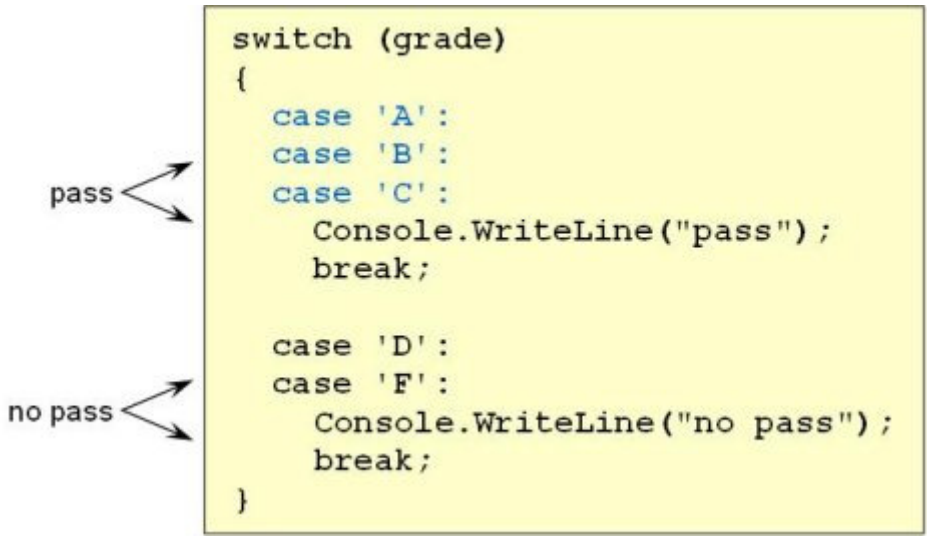


A `break` is required to mark the end of each `case` that contains associated code. It is a compile time error if the `break` is omitted.



It is possible to associate multiple options with a single action by placing the cases one after the other with no intervening code. Note that it is legal to omit the `break` from a case if it does not have any associated code; this fact is crucial in understanding why multiple labels are allowed and only a single `break` is required at the end of the entire

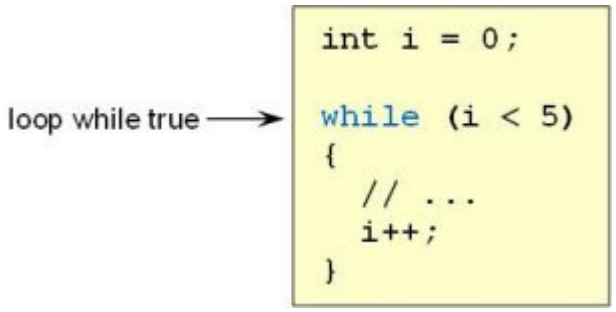
construct.



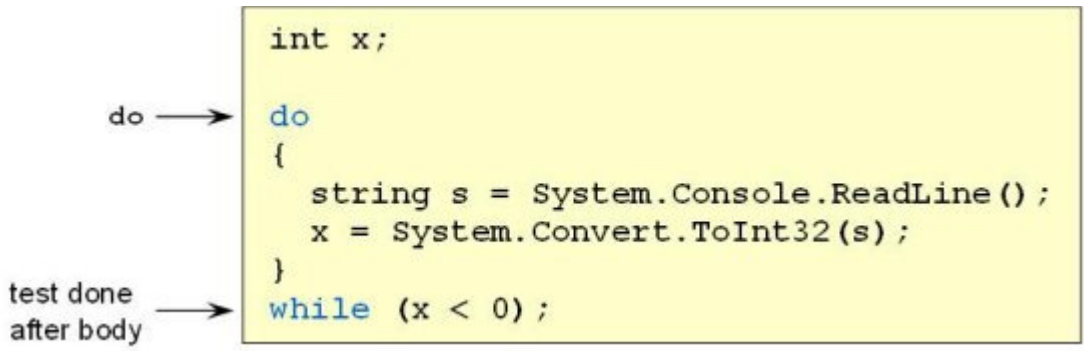
Loops

C# offers 4 loops: `while`, `do`, `for`, and `foreach`.

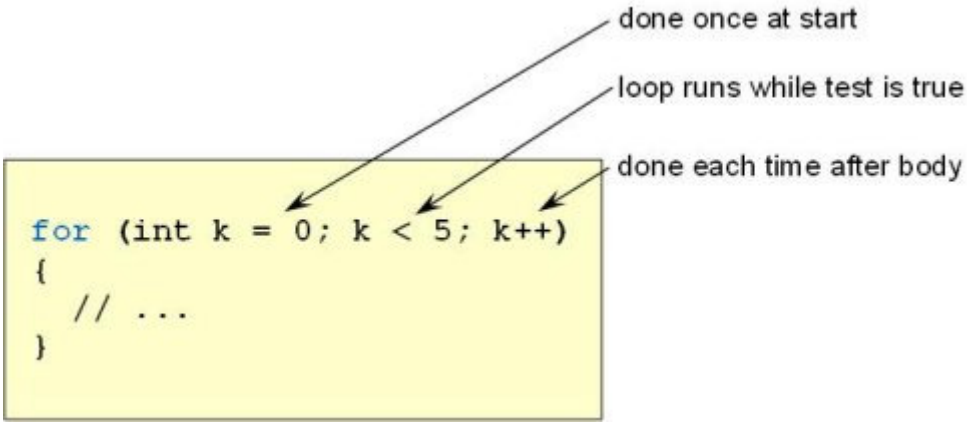
A `while` loop is the most basic looping construct. It consists of the keyword `while` followed by a Boolean expression and then a statement or code block. As long as the Boolean expression continues to evaluate to true, the associated code will get executed.



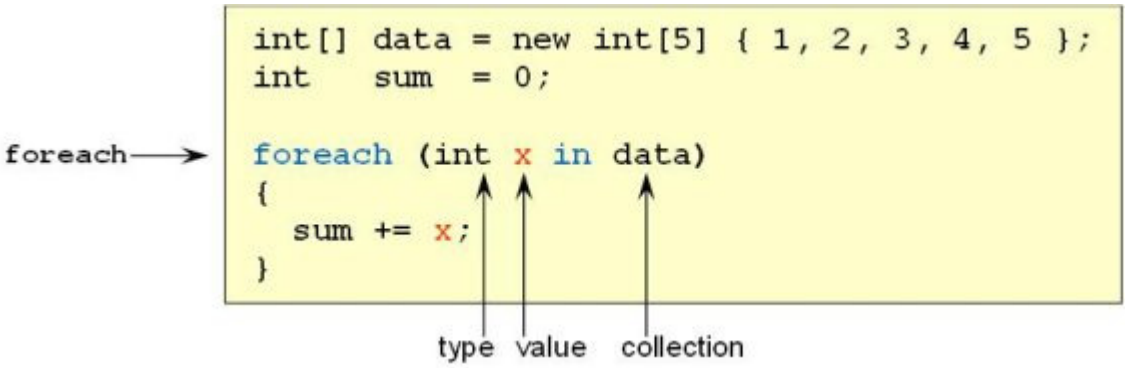
A `do` loop is both more complicated and less useful than a `while` loop. It consists of the keyword `do`, a statement or code block that forms the body of the loop, the keyword `while`, a Boolean expression, and a semicolon to end the whole construct. The odd thing about the `do` loop is that the test condition is at the bottom of the loop. When the loop begins to run, the test is not evaluated so the body is executed immediately. Only after the body is executed is the test evaluated. If the test is evaluates to true, then the process repeats; otherwise the loop ends. This behavior means that the body of a `do` loop is always executed at least once.



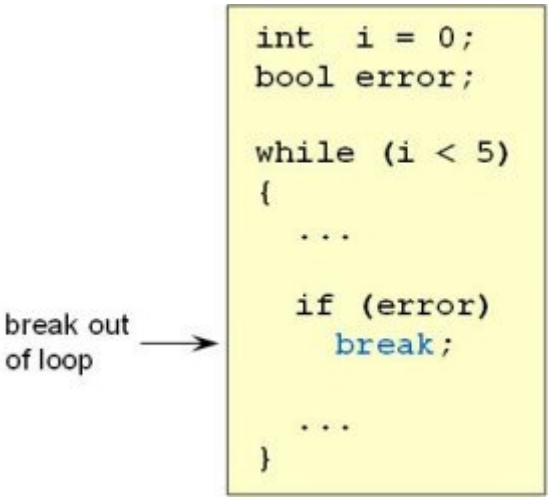
A `for` loop has three parts separated by semicolons. The first part is used for initialization and is performed only once when the loop first begins execution. The second part is the test condition used to determine if the loop should continue to run. The third part is executed after each iteration and so is typically used to increment a loop counter.



A `foreach` loop is a specialized construct for use with collections of data such as arrays. The syntax and use is simple and very readable: the keyword `foreach`, an open parenthesis, the type of data stored in the collection, a variable name used to hold each successive value from the collection, the keyword `in`, the collection to be traversed, a close parenthesis, and finally the loop body. The loop body is executed once for each element in the collection.



The keyword `break` can be used to exit any of the loops before the loop would normally terminate. `break` is most often used to abort processing if an error occurs.



01- Basics Exercise

Goals:

- Experiment with console input and output.
- Declare and use variables.
- Practice with language control constructs.
- Create and manipulate arrays.
- Perform environment interaction.

Overview

Write some simple programs to practice with language basics such as variables, control constructs, arrays, input and output facilities, etc.

Part 1- Console Input and Output

Here we will practice reading and writing to the console using the Console class in the System namespace. In addition, we will examine how to use the methods in the Convert class to convert from the characters the input methods deliver to numeric types such as int and double. Note that the documentation for Console and Convert can be easily accessed from within Visual Studio .NET by placing the cursor on the name in the text editor and hitting the F1 key. Feel free to browse the documentation and experiment with any methods you find.

Steps:

1. **Basic output.**
   The Console class provides two output methods: Write and WriteLine. There are many overloaded versions of each method allowing them to handle all the required types. For example, there are versions of both methods that take an int argument,

other versions that take a double, versions that takes a string, etc. The difference between Write and WriteLine is that WriteLine automatically adds a line terminator to the output. Experiment with both Write and WriteLine. Print literal values and variables of several different types. A good set of types might be string, int, double, and char. It might also be interesting to see how a bool value is printed. Note that there is a version of WriteLine that takes no arguments and simply prints a line terminator. This might be useful to visually separate the output of your program.

2. **Formatted output**.
   There are versions of Write and WriteLine that take a format string followed by a variable number of other arguments. The format string can contain placeholders which consist of an integer inside curly braces. The integer corresponds to the position of one of the additional parameters, so {0} represents the first parameter after the format string, {1} is the second, {2} the third, and so on. WriteLine replaces the placeholder with the value of the specified parameter and prints the resulting string. For example, the following can be used to embed the value of the variable a in the output string:
   ```
   Console.WriteLine("The value of a is {0}", a);
   ```
   Experiment with this version of WriteLine. It might be interesting to print several variables at once.
3. **Input.**
   The Console class offers two input methods: Read and ReadLine. Read returns a single character and ReadLine an entire line of input. For our purposes the easiest approach is to ask our users to enter one input value per line so we can simply assume each call to ReadLine will get one input value. The ReadLine method returns the next input line as a string. For example:
   ```
   string s = Console.ReadLine();
   ```
   We can then keep the value as a string or convert it to another type, as in:
   ```
   int i = Convert.ToInt32(s);
   double d = Convert.ToDouble(s);
   ```
   Experiment with the Convert methods by reading a few strings and converting to appropriate types.

Part 2- Loops and conditionals

Here we will practice with loops and conditionals by writing a program to calculate a value for Pi using the following formula:
Pi/4 = 1 â€" 1/3 + 1/5 â€" 1/7 + 1/9 ...
In other words, we are summing:
1/(2k+1) for k = 0, 1, 2, ...
The more terms in the sum, the more precise the calculated value.

Steps:

1. Write a loop to calculate the individual terms in the series and sum them. To keep things simple, you can choose to simply sum the first 1000 elements of the series. Print out the resulting sum. Note that when calculating terms in the series, you have to be careful to force floating point arithmetic. If you write the following integer expression you will get integer arithmetic in which every term will be zero since integer division truncates the result:
   1 / (2 * k + 1)
   An easy way to force floating point arithmetic is to use 1.0 in the numerator instead of 1, as in:
   1.0 / (2 * k + 1)
   Another consideration is that the sign of the terms in the series alternates: positive, negative, positive, and so on. The kth term is positive if k is even, and negative if k is odd. You can test k using the remainder operator (%) to see if the remainder on division by two is 0 (even) or 1 (odd):
   `if (k % 2 == 0) // add term, else subtract`

Part 3- Switch

Write a program to implement a very simple calculator. The calculator will read the operation (+, -, *, /) and the two operands, perform the operation, and output the result.

Steps:

1. Read the operation from the user using ReadLine. The operation can remain as a string or you can convert it into a single character using Convert.ToChar.

2. Read the two operands using ReadLine. The user will need to enter each on a separate line. Convert the strings into doubles.
3. Use a switch statement to determine the operation. Perform the operation and output the result.
4. Use the default case of the switch to notify the user if they enter an invalid operation.

Part 4- Array

Write a program to find the minimum value contained in an array.

Steps:

1. Create an array of 10 integers.
2. Read in values from the user and load them into the array. Use a for loop to loop through the array and assign the values. Use the Length property of the array in the stop condition of the loop. Note that it is not possible to use a foreach loop here since foreach gives read only access to the array elements.
3. Search through the array and locate the minimum value. Use a foreach loop to loop through the array.
4. Output the array contents and the minimum value. Use a foreach loop to loop through the array.

Part 5- Multidimensional Array (optional)

Multiple array dimensions are supported. The dimensions are comma separated within the square brackets. For example, the following code creates a 3x4 array of int.

```
int[,] a = new int[3,4];
```

Element access is performed using a single set of square brackets with the indices for each dimension comma separated. For example:

```
a[0,0] = 17;    // write a value into row 0, element 0
a[0,1] = 32;    // write a value into row 0, element 1
int x = a[0,1]; // read row 0, element 1
```

The GetLength method is used to obtain the number of elements in a particular dimension. Note that Length gives the total number of elements in the array. The following code illustrates.

```
int[,] a = new int[3,4];

int l1 = a.Length;       // yields 12 the total number of elements in all dimensions
int l2 = a.GetLength(0); // yields 3, the number of rows
int l3 = a.GetLength(1); // yields 4, the number of elements in each row
```

Steps:

1. Create a 2x5 array of double.
2. Print out Length (the total number of elements).
3. Use the GetLength method to determine the number of elements in each dimension. Print out the values.
4. To practice with the element access syntax, write a value into any element of your choice.
5. Use a foreach loop to iterate through the array. Does foreach go through a single row or the entire array?

Part 6- Environment interaction (optional)

Tools such as compilers, linkers, DOS commands, etc. often use command line interfaces. Arguments are passed on the command line and are processed by the program to control its behavior. A return value is sent back at the end of execution to indicate success or failure. Convention dictates that a return value of zero is used to indicate success and non-zero values to indicate failure. Different non-zero return codes can be used for different error conditions. Batch files often test the return value to determine how to proceed.

The Main method can optionally perform environment interaction: it can receive its command line arguments as an array of string and it can return an int to indicate success or failure. To support the various methods of environment interaction, there are four allowable signatures for Main.

```
static void Main() { ... }
static int  Main() { ... }
static void Main(string[] args) { ... }
```

```
static int  Main(string[] args) { ... }
```

In this lab we will practice accessing the program's environment from with the Main method.

Steps:

1. Declare a class called EnvironmentInteraction and add a Main method. Declare Main so that it takes an array of string argument and returns an int.

```
2.    static int Main(string[] args)
3.    {
4.      ...
}
```

5. Use a foreach loop to loop through the command line parameters and print out each one.
6. Use a return statement to return an exit code of 0 from Main. An exit code of 0 indicates successful completion. Non-zero values can be used to indicate error conditions.
7. From within Visual Studio .NET you can set the command line parameters by right-clicking the project, selecting Properties, opening the Configuration Properties folder, clicking on Debugging and entering a value in the Command Line Arguments property. Alternatively, you can run the program from a command window and type the parameters on the command line.
8. Run the program. The exit code should be displayed in the debug window if you are running from within Visual Studio .NET.
9. The System.Environment class offers an alternative way to access a program's environment. Environment provides the ability to get the command line and to exit the program.

```
10.   string[] cmdLine = Environment.GetCommandLineArgs();
11.
Environment.Exit(0);
```

   Experiment with the Environment class. Notice that the command line argument array includes the program name. You may want to explore the documentation to see what else the class offers. For example, one thing that may be of interest is the ability to access environment variables.

02-Class

Goals:

- Introduce the concept of `class`.
- Describe how to define fields and methods.
- Show how to create and manipulate objects.
- Examine the most common parameter passing options.

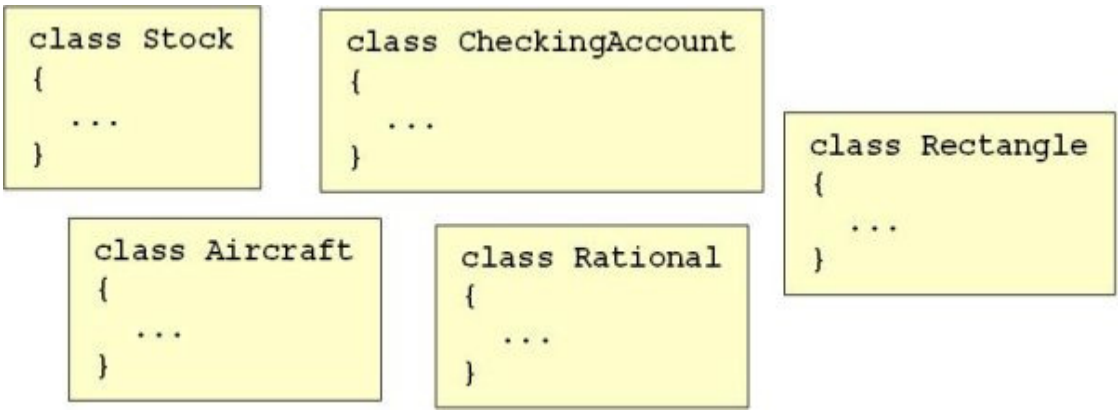
Overview

C# code is primarily organized into classes. A class contains both the data and operations needed to implement a type. For example, if a drawing program needed the concept of 'rectangle' then all the code needed to implement the rectangle type would be collected into a `Rectangle` class. Grouping code in this way is not only clean and organized, it also makes it easy to locate the appropriate code for maintenance, extension, or debugging.

Class

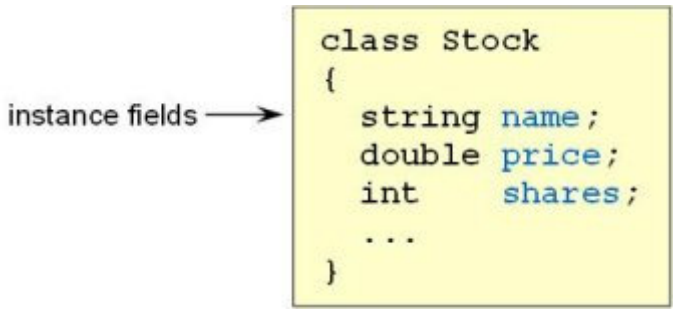
A class represents a concept in the application domain. Programmers working on a graphics program might have classes such as `Circle`, `Rectangle`, and `Line`. A team working on a financial application would have classes such as `Stock`, `Bond`, and `Property`. In C#, a class is the primary way to create a user defined data type.

A class is defined using the keyword `class`, the name of the class, and the body of the class enclosed in curly braces.



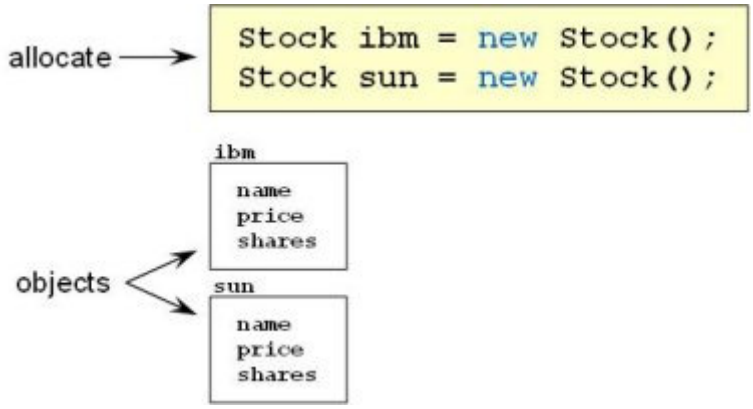
Fields

A class can contain fields to store data. Fields are defined using the type name and the field name. Fields defined in this way are called "instance fields" or "instance variables" since each instance of the class gets it own copy. Note that fields are defined at class scope; that is, they are not defined inside a method.



Object creation

Objects of class type are created using the `new` operator. `new` allocates memory for all the instance fields of the class.

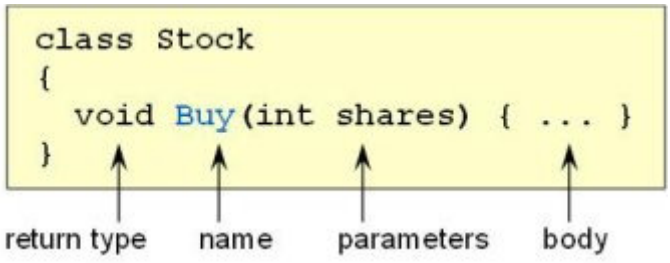


Member access

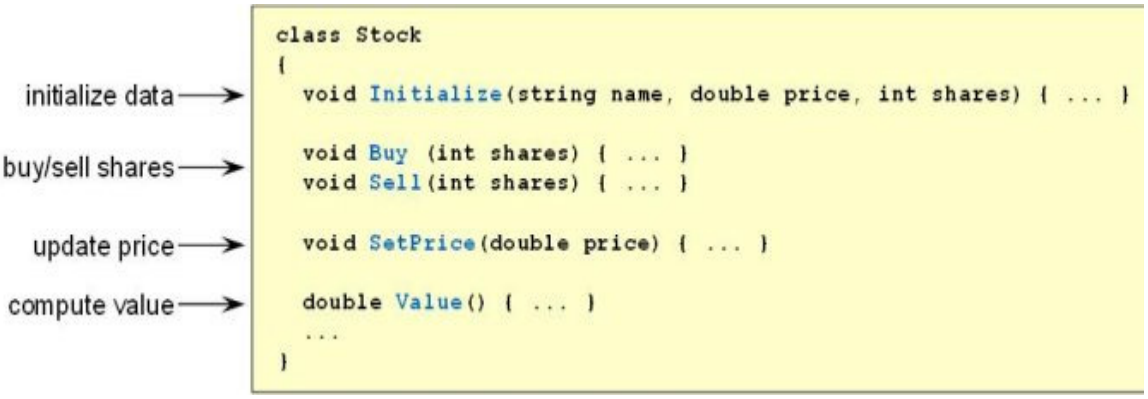
The dot operator is used to access class members. Instance members must be accessed using an object so the syntax is typically `object.member`.

Methods

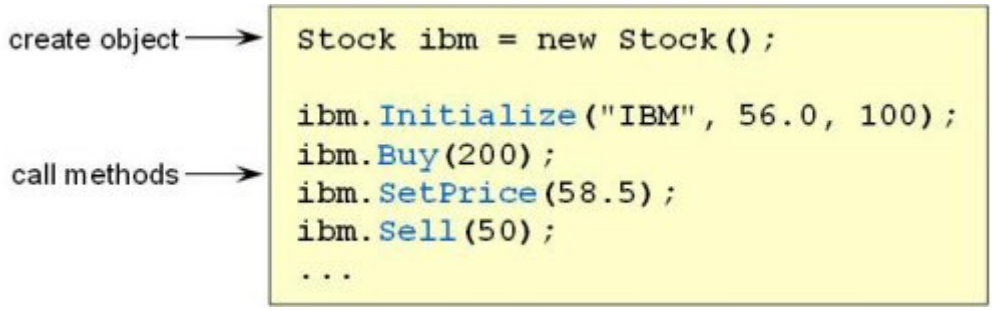
The operations supported by a class are defined using methods. For example, a rectangle type might support operations such as `Draw`, `Move`, `Area`, etc. while a stock type would offer `Buy`, `Sell`, `SetPrice`, etc. In C#, methods must be placed inside a class. A method is defined by specifying the return type, name, parameter list, and body. A return type of `void` is used for methods that do not return a value. Methods defined in this way are called "instance methods" since they must be invoked on an object of the class.



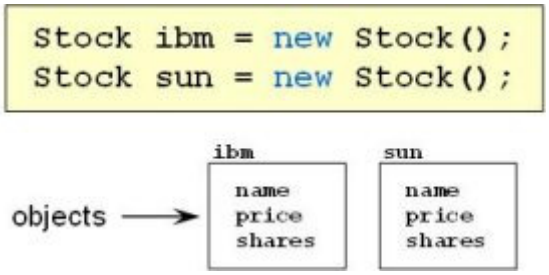
The set of methods offered by a stock class might look something like the following.



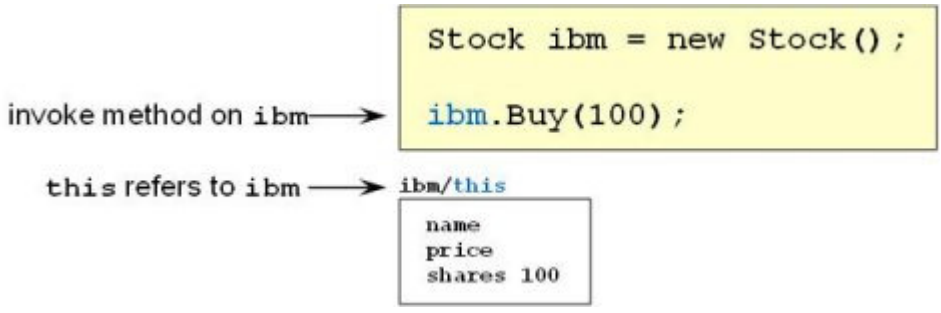
Methods are invoked by applying the dot operator to an object, selecting the desired method by name, and enclosing the arguments in parentheses.



Methods typically need access to the fields of the class. For example, the rectangle `Area` method must access the width and height of the rectangle, and the stock `Buy` method needs to modify the number of shares owned. To understand how methods are able to access needed fields it helps to examine the concept of the "current object". Suppose there two stock objects in existence: `ibm` and `sun` as shown below.

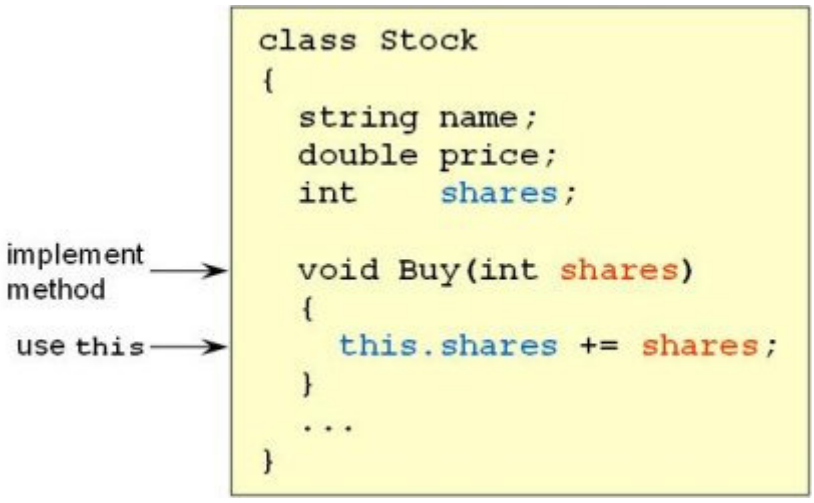


Each time an instance method is invoked, a particular object must be specified. For example, a method call such as `ibm.Buy(...)` would manipulate the `ibm` object whils `sun.Sell(...)` would access `sun`. Thus in each method call there is the concept of the "current object"; that is, the object on which the method was invoked. The notion of the current object is formalized by the keyword `this`. When a method is invoked, the runtime arranges for `this` to represent the object on which the method was called. `this` exists only for the duration of the method call and is only accessible from inside the method.

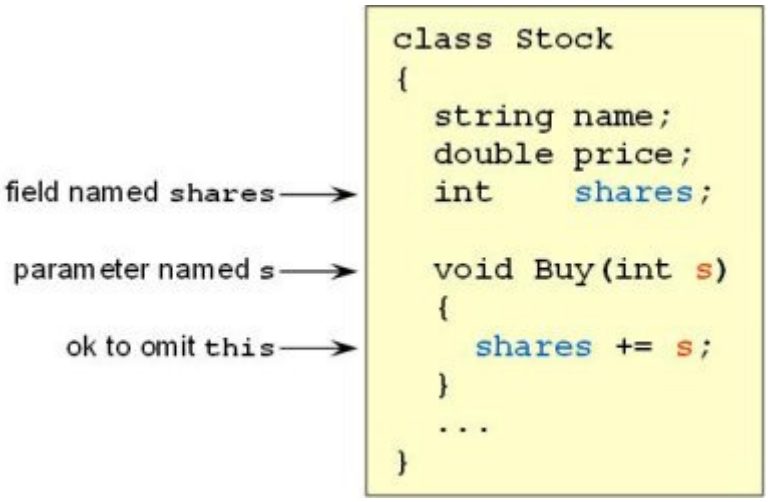


Method implementations use `this` as a handle to get access to the members of the current

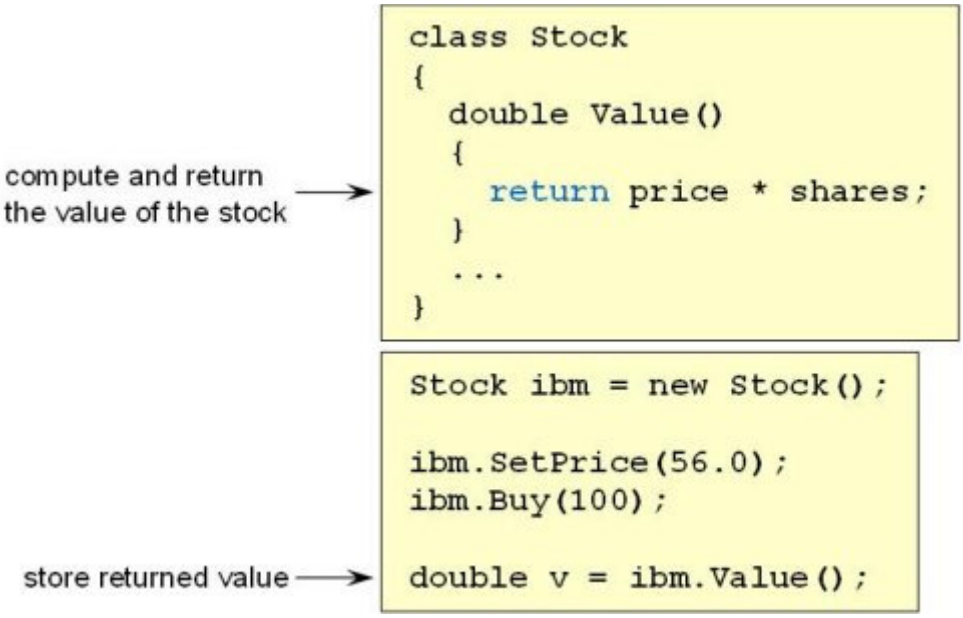
object.



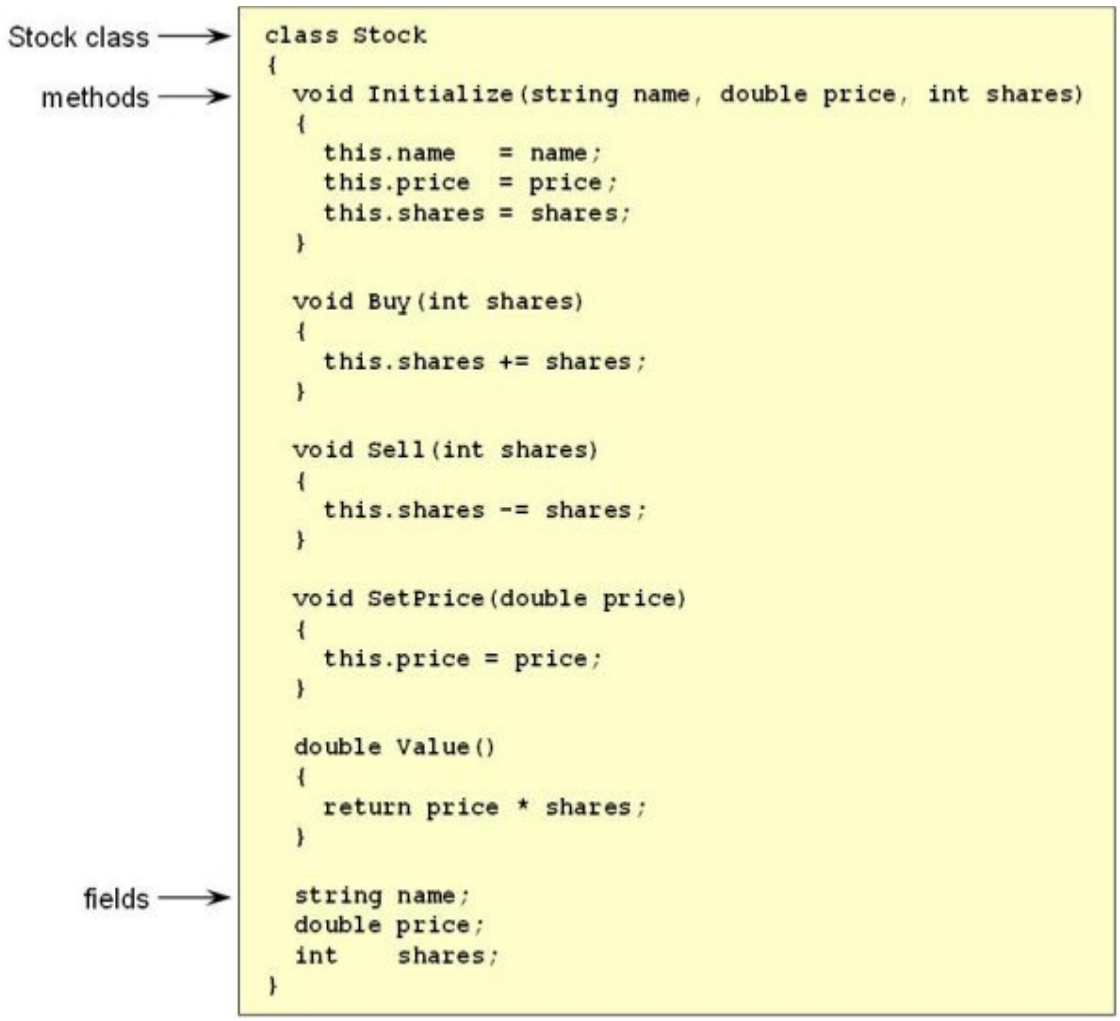
In the previous example, the method parameter and a field were both named `shares`. Because of the ambiguity, use of `this` was required in order to access the field. If `this` were omitted, the code would have compiled; however, both uses of the name `shares` would have referred to the parameter and the field would not have been updated. In situations where there is no ambiguity, `this` can be safely omitted and the correct class member will still be accessed.



Methods may return a value. To return a value, the method declaration must specify the type of data returned and the method implementation must use the keyword `return` to send the data out of the method. For example, the stock class might supply a method such as `Value` which computes and returns the current value of the stock.

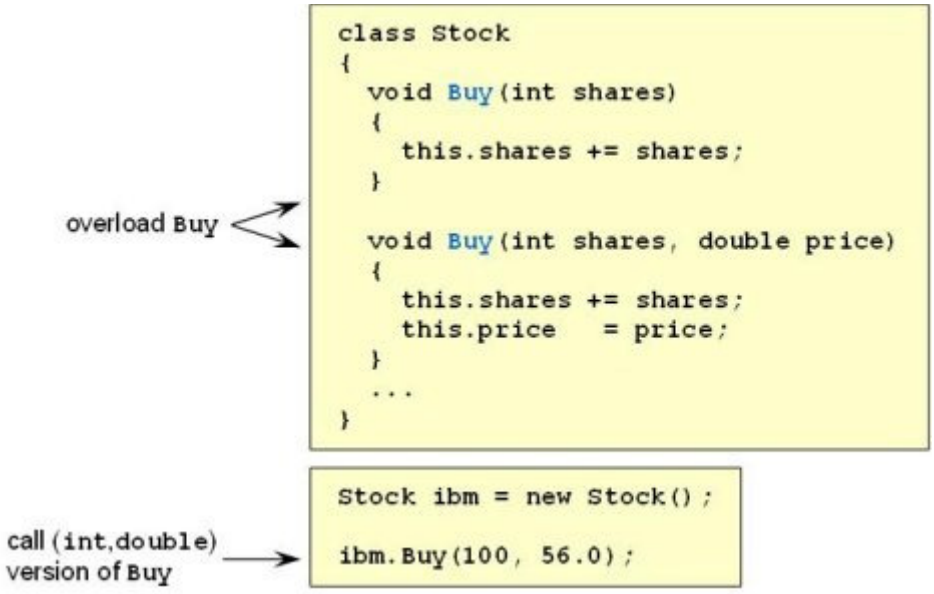


The entire implementation of the stock class is shown below for reference.



Method overloading

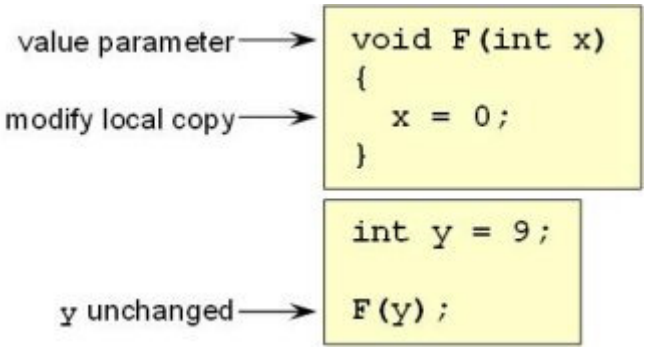
It is legal for a class to offer more than one method with the same name as long as the parameter lists are different. For example, the stock class could offer multiple versions of the `Buy` method that perform slightly different operations. The compiler examines the client code and calls the correct version based on the type and amount of data passed.



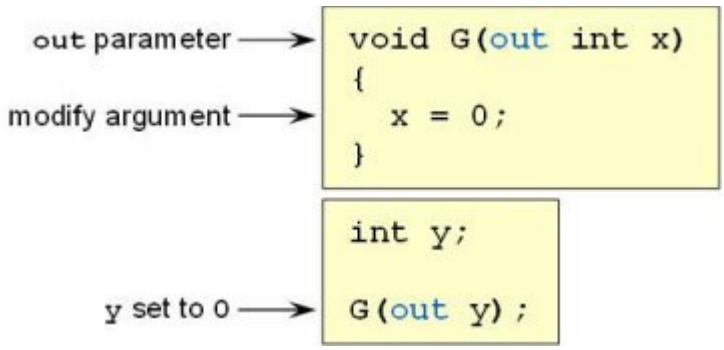
Parameter passing

There are three primary ways to pass parameters to methods: value, `ref`, or `out`.

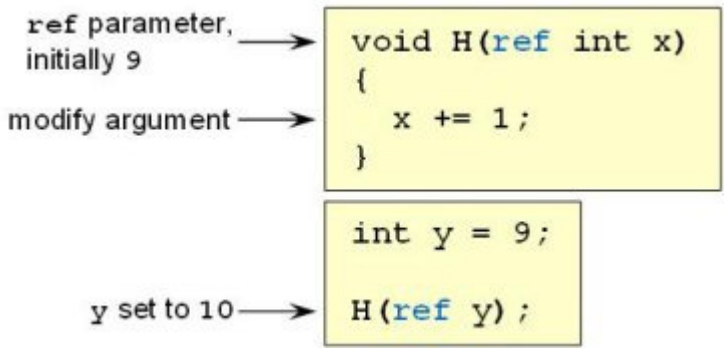
A value parameter can be thought of as "in only"; that is, the data passed is copied into the method and the method then operates on the local copy. Any changes made by the method modify only the local copy and are lost when the method ends. Pass by value is the default parameter passing mechanism so there is no keyword needed to specify its use.



An out parameter is "out only"; that is, no data is passed into the method but an assignment to the parameter is visible in the client code. The keyword `out` must be placed on both the method definition and the method call to create an out parameter.



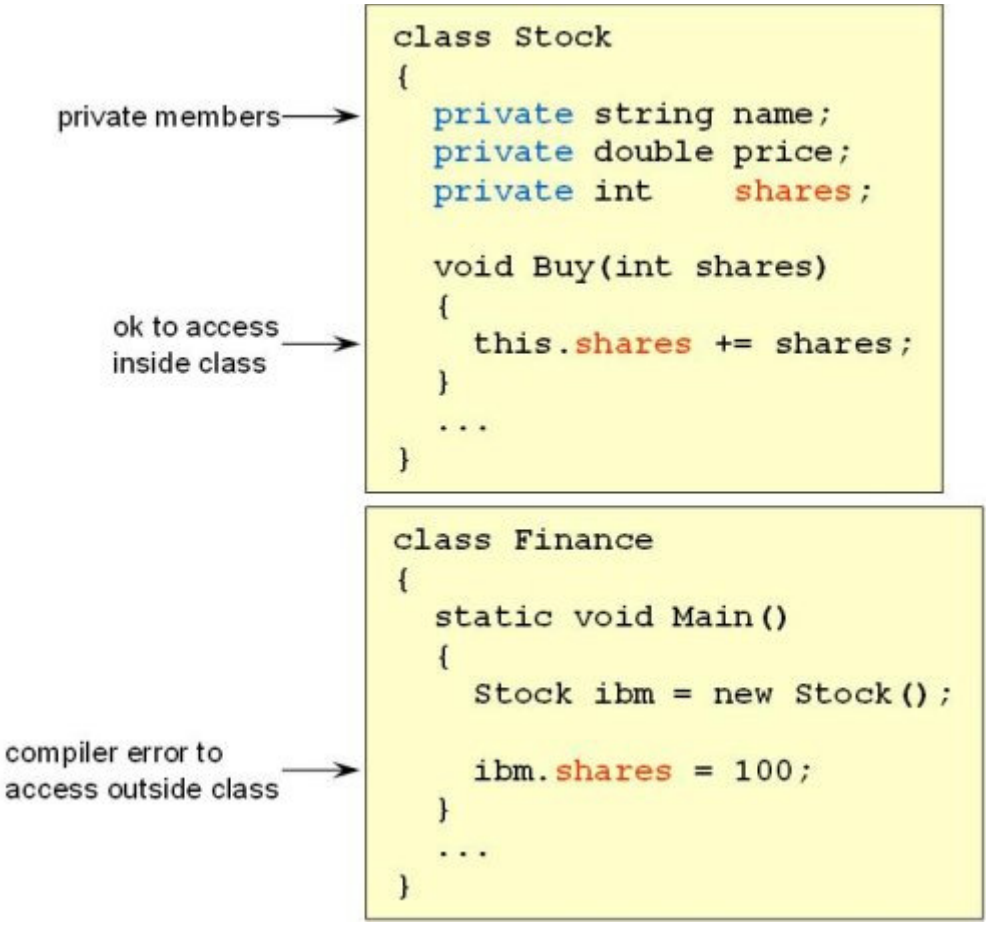
A `ref` parameter is "in and out"; that is, the parameter is just a "handle" or "reference" to the actual argument. The value is available for reading and/or writing and any changes made inside the method are visible in the outside world. The keyword `ref` must be placed on both the method definition and the method call to create a ref parameter.



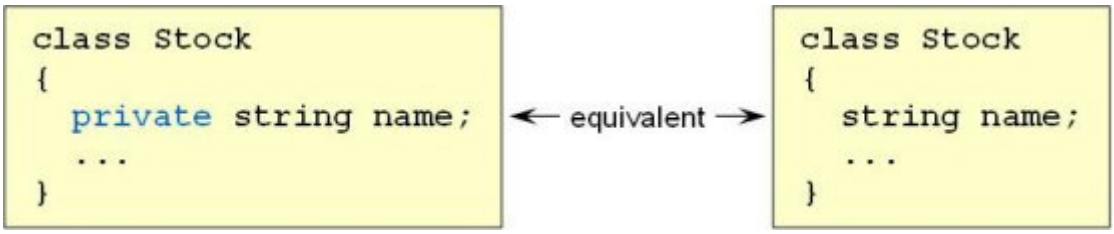
Member access

C# provides the ability to control access to class members. The three most common access levels are `public`, `private`, and `protected`. We will cover `public` and `private` here and discuss `protected` later during the module on inheritance.

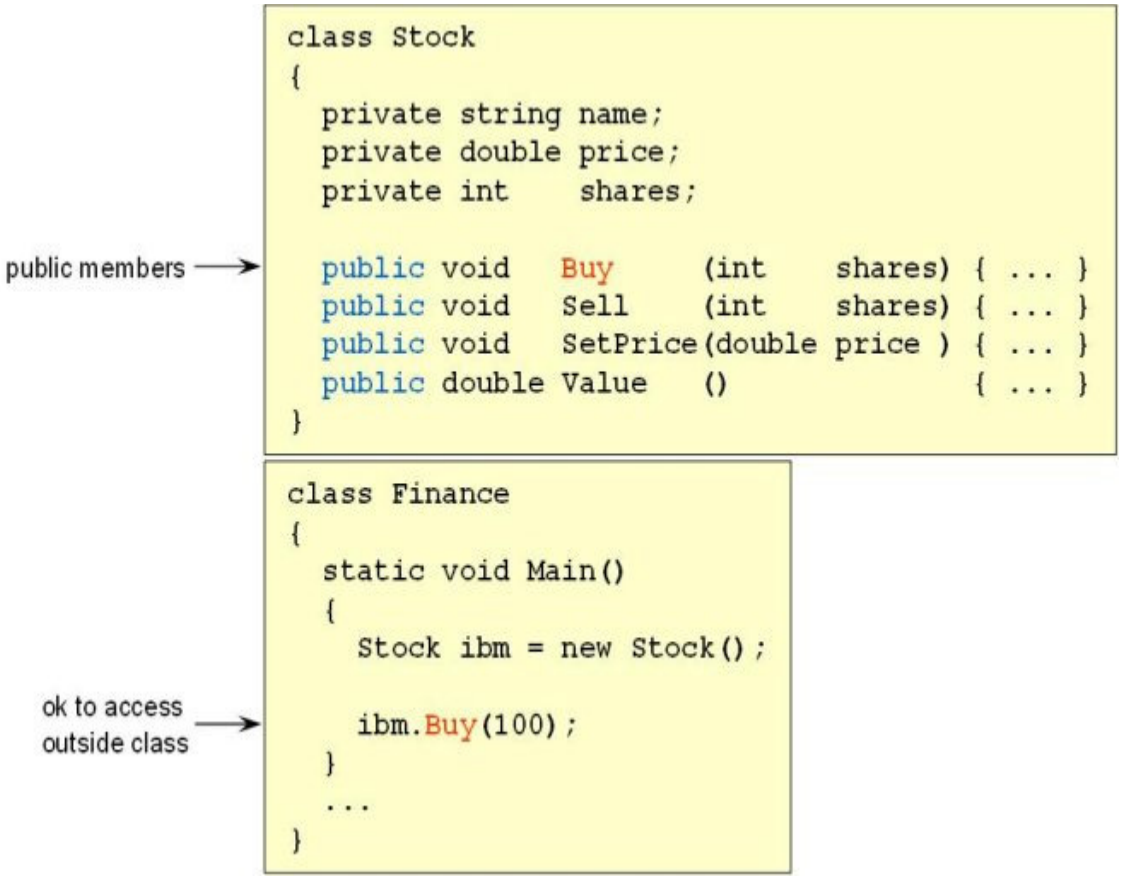
The class designer can choose to limit the usage of a field or method by adding the keyword `private` to its declaration. `private` members can only be accessed from inside that class. Consider the following implementation of a stock class where the fields have been made private. Note how the methods inside the stock class are able to access the private fields while the client code outside the stock class cannot.



Private is the default access level, so the following two declarations are equivalent.



To allow client code to access a field or method, the member can be made public by adding the keyword `public` to its declaration.




```
class Stock
{
  private string name;
  private double price;
  private int    shares;

  public void   Buy     (int    shares) { ... }
  public void   Sell    (int    shares) { ... }
  public void   SetPrice(double price ) { ... }
  public double Value   ()              { ... }
}
```

```
class Finance
{
  static void Main()
  {
    Stock ibm = new Stock();

    ibm.Buy(100);
  }
  ...
}
```

The public/private split allows a class to be divided into interface and implementation. The implementation is kept private while the interface is made public. Client code can only access the public interface and is prevented from directly accessing the implementation. This concept is often described using the terms 'encapsulation' and 'information hiding'. The most commonly cited advantage of this coding style is that the class designer is free to modify the implementation at any time without breaking any client code. This is possible since the client code cannot access the implementation directly anyway so changes have no impact.

Naming conventions

Some naming conventions have been adopted by C# developers in order to increase the uniformity of code written by different programmers in different organizations. Most names use the "intercaps" convention where the first letter of each word is capitalized.

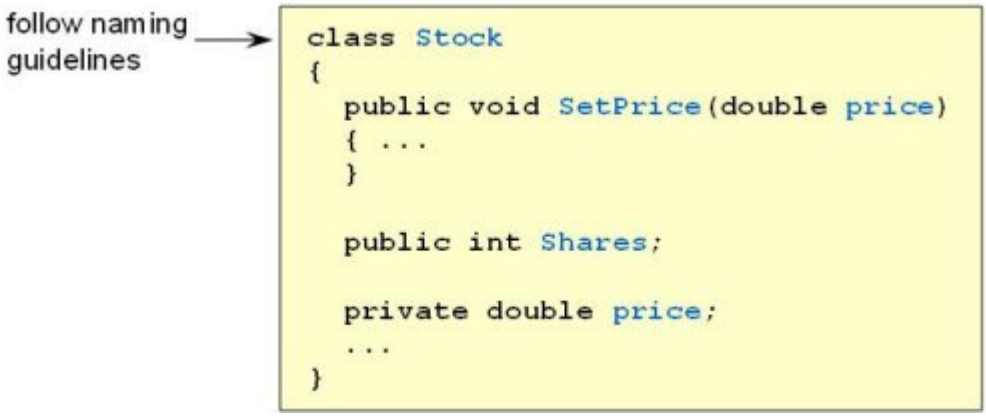
Class names are intercaps with an initial capital letter.

Method names are intercaps with an initial capital letter.

Method parameters are intercaps with an initial lower case letter.

Public fields are intercaps with an initial capital letter.

Private fields are intercaps with an initial lower case letter.



02- Class Exercise

Goals:

- Write a class that demonstrates various common coding patterns.
- Use instance fields to store state.
- Use instance methods to manipulate and report state.
- Apply access modifiers as appropriate.
- Practice with the various parameter passing mechanisms.

Overview

A class is the primary unit of code in C#. A class defines a new type that models some aspect of the 'real' world. In this activity, we will write code that demonstrates a few of the most common coding patterns for classes. We will also explore the various ways parameters can be passed to methods.

Part 1- Point

Here we will implement a class to model a two dimensional point. The point class will provide a good example of the most common type of class: instance fields store the state of an object and methods implement the behavior of the class by manipulating and reporting the state.

Steps:

1. Create a new project in Visual Studio .NET for the point exercise.
2. Create a class called Point to represent a two dimensional point.

```
class Point
{
  ...
}
```

6. Add two instance fields to store the state of the point. The fields are private since we only want code in the Point class to access them. Users will be forced to go through the public interface that

we will add shortly.

```
7.    private int x; // x coordinate
      private int y; // y coordinate
```

8. Implement the following Initialize method to load the passed values into the private fields.

```
public void Initialize(int x, int y)...
```

9. Implement public get and set "access methods" for the x and y coordinates. The practice of making the data private and providing public access methods is relatively common in object oriented programming.

```
10.   public void SetX(int x) ...
11.   public void SetY(int y) ...
12.
13.   public int GetX() ...
      public int GetY() ...
```

14. Implement a public Translate method to move the point by the given x and y offsets. The implementation is simple: each offset only needs to be added to the corresponding coordinate.

```
public void Translate(int xOffset, int yOffset) ...
```

15. Implement a public Scale method to move the point to a new location. The new coordinates are obtained by multiplying the old coordinates by the specified scale value. This operation may seem odd, but it actually has a nice geometric interpretation: it corresponds to moving the point to a different place on the line connecting the point and the origin.

```
public void Scale(int a) ...
```

16. Implement a public Distance method to compute the distance between the current point and the parameter.

```
public double Distance(Point p) ...
```

Recall that the formula for the distance between points is:

```
square root((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))
```

You can use the Math.Sqrt method to compute the square root.

17. Code a Driver class with a Main method to test your work.

Part 2- Parameter passing

C# offers three ways to pass a parameter to a method: value, ref, out.

Steps:

1. Create a new project in Visual Studio .NET for the parameter passing exercise. Create a simple class called ParameterTest to hold the parameter test code.
2. Write a method called Value that takes a single integer by value. Because the parameter is passed by value, any changes made inside the method affect only a local copy and will be lost when the method ends.

```
3.   public void Value(int x)
4.   {
5.           ...
}
```

   Create a class called Driver and add a Main method to it. Create a ParameterTest object and call the Value method and verify that it behaves as expected.

6. Write a method called Swap that takes a takes two integers by reference and swaps their values. Call the method and verify that the changes made inside the method are visible after the call. Note that the keyword ref is required in both the parameter definition and in the method call. Reference parameters are sometimes called "in out" parameters because a value goes in and comes out of the method. Since a value is being passed in, the compiler requires that a variable passed as a reference parameter already be initialized. Verify this behavior by attempting to pass an uninitialized variable as a reference parameter.

```
7.   public void Swap(ref int a, ref int b)
8.   {
9.           ...
}
```

10. Write a method called MinMax that takes an array of integers and two out parameters. Locate both the minimum and maximum values in the array and assign them to the out

parameters in order to return both values to the calling code. Call the method and verify that data is being returned through the parameters.

```
11.  public void MinMax(int[] a, out int min, out int max)
12.  {
13.          ...
     }
```

Part 3- Variable length parameter list

C# supports variable length parameter lists. A variable length parameter is declared by putting the keyword params on a one dimensional array. The caller can then pass any number of arguments and the compiler creates an array and loads it with the passed values. There are a few restrictions: the variable parameter must be the last argument and it can only be applied to a one dimensional array. An example is shown below with one regular parameter and a variable length parameter. Users calling the method pass a string argument followed by zero or more integers.

```
void Process(string prefix, params int[] values)
{
        //...
}

void Test()
{
        Process("data");                    // zero ints
        Process("data", 1);                  // one int
        Process("data", 1, 2, 3);             // three ints
        Process("data", new int[] {1, 2, 3, 4}); // explicit array ok too
}
```

Steps:

1. Write a method called Sum that uses the params technique to take a variable number of integers, compute their sum, and return the result.

Part 4- Robot (optional)

Here we will create a class to model a simple robot. The robot will have a name, a location in a two dimensional world, and will record the compass direction it is facing. The Robot will begin with only simple behavior: initialization and printing of its state. More complex abilities such as moving and turning will be added as the exercise progresses.

Steps:

1. Create a new project in Visual Studio .NET for the robot exercise.
2. Create a class called Robot to represent a robot.

```
class Robot
{
  ...
}
```

6. Add four instance fields to store the state of the robot. The fields are private since we only want code in the Robot class to access them. Users will be forced to go through the public interface that we will add shortly.

```
private string name;
private int    x;
private int    y;
private int    direction;
```

10. Next we need to model the compass directions North, East, South, and West. There are many ways this can be done in C#. We will choose the simplest option and define a public integer field for each direction. These values can be assigned to the direction field to indicate the direction the robot is facing. We will make this a bit more realistic by introducing the C# keyword const to mark the directions as constants. Adding const makes our code more logically correct since we are creating symbolic constants. Later we will see other options for creating symbolic constants.

```
public const int North = 0;
public const int East  = 1;
public const int South = 2;
public const int West  = 3;
```

14. Add an implementation of the following Initialize method.

The method should set each field to the corresponding value passed as a parameter.

```
public void Initialize(string name, int x, int y, int direction)
{
  ...
}
```

18. Add an implementation of the following Print method. The method should print the current state of the robot (name, x, y, and direction) to the console.

```
public void Print()
{
  ...
}
```

When printing the direction, a function such as the following might be convenient to map from the internal integer representation to a human readable string. Note that the position in the array of each string corresponds to the value for the constants, so the string "North" is in the first position since the value of the constant North is 0. The method is private since it is for the exclusive use of the Robot class.

```
private string ToString(int direction)

{
  string[] names = { "North", "East", "South", "West" };

  return names[direction];
}
```

22. Create a class called World containing a Main method. Inside Main, create a Robot and call the Initialize and Print methods to test your work.
23. We will now add some more interesting behavior to the Robot class beginning with a method called Move. The Move method will move the robot forward one unit in the direction it is currently facing. For example, if the robot is currently at position (0,0) and is facing north, calling Move will move the robot one unit in the positive y direction (i.e. North) so the robot will then be at position (0,1). Implement the following public Move method for the Robot class. Call Move from Main to test your work.

```
public void Move()
{
  ...
}
```

27. Next we add the ability for the robot to turn either left or right. Turning the robot modifies the direction the robot is facing. For example, if the robot is currently facing north and turns left, the robot would then be facing west.

    Add two symbolic constants to represent Left and Right.

```
public const int Left  = 4;
public const int Right = 5;
```

    Modify the ToString method so it works with the new constants.

```
private string ToString(int direction)
{
  string[] names = { "North", "East", "South", "West", "Left", "Right" };

  return names[direction];
}
```

    Implement the following public Turn method. The parameter to is used to indicate whether the turn is to the Left or to the Right. Call Turn from Main to test your work.

```
public void Turn(int to)
{
  ...
}
```

28. We will now add a random number generator to the Robot. The .NET Framework class library provides a class named System.Random that will do the work for us.

    Add the following field to the robot simulation. Notice how the generator uses the current time as the seed to ensure a different sequence of random numbers each time the program is run.

```
private Random random = new Random((int)DateTime.Now.Ticks);
```

    Add a method called Step to the Robot class which makes use of the random number generator. The generator offers a method named Next which takes an int parameter (called the bound) and returns a random number between 0 and bound-1. The step method should choose randomly between moving and turning. A reasonable probability distribution might be a 50% chance for Move(), a 25% chance for Turn(Left), and a 25% chance for

Turn(Right). Notice that calling Next(2) on the random number generator has an even probability of returning 0 or 1 thus providing a convenient way to create a 50/50 chance. Call Step from Main to test your work.

```
public void Step()
{
  ...
}
```

29. Here we have the robot search for a goal. The goal is simply an (x,y) coordinate pair. To keep the simulation from running indefinitely we will define a "world size" and restrict the robot's movement to coordinates in the valid range. Add the following constant to the Robot class.

```
public const int Size = 10;
```

Modify the Move method to keep the coordinates in the range 0 through Size-1. The following code accomplishes this very concisely.

```
public void Move()
{
  ...
  //
  // force coordinates back into valid range
  //
  x = (x + Size) % Size;
  y = (y + Size) % Size;
}
```

Add a method called At to the Robot class that determines whether the robot is at the given coordinates. This method will be useful to test whether the robot has reached the goal.

```
public bool At(int x, int y)
{
  ...
}
```

Hardcode an (x,y) coordinate pair in the Main method to serve as the goal.

```
int goalx = 5;
int goaly = 5;
```

Repeatedly call Step on the robot until it reaches the goal. Use the At method to determine if the robot is at the goal. You may also want to create several robots and have them compete to find the goal.

03-Initialization

Goals:

- Present the default initial values for instance fields.
- Describe how to use variable initializers.
- Show how to write and invoke constructors.

Overview

Here we discuss the various options available to initialize the instance fields of a class.

Default values

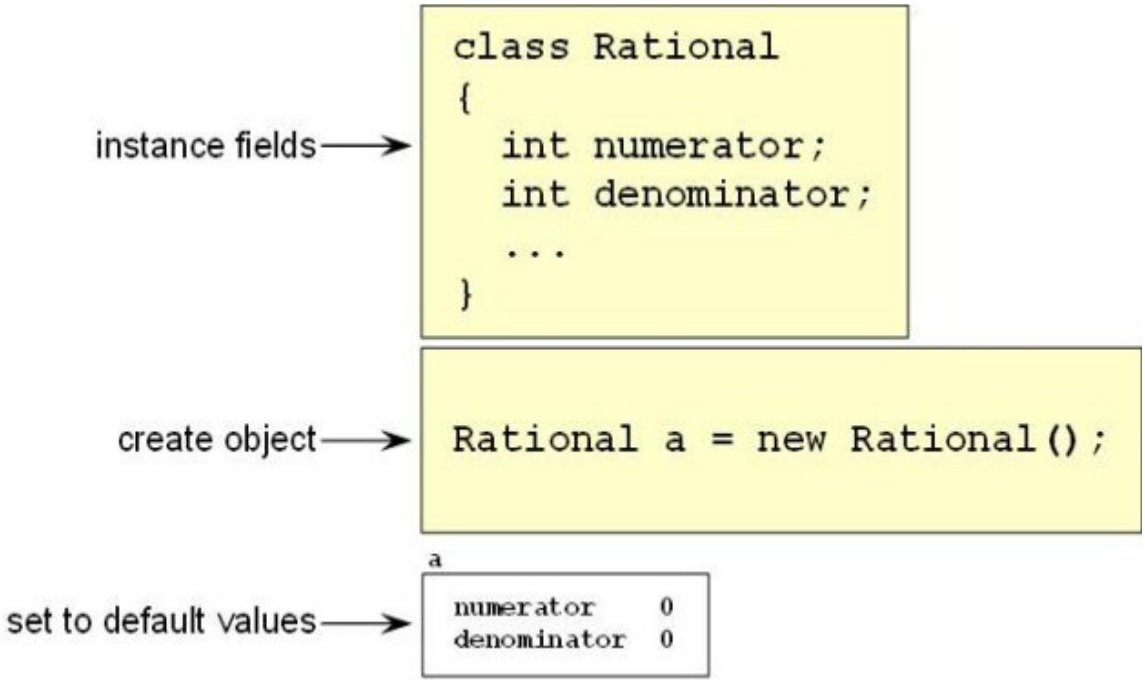
Instance fields of a class are set to default values when an object is created. The default value depends on the type of the field.

Numeric fields such as `int`, `float`, `double`, etc. are set to zero.

Fields of type `bool` are set to `false`.

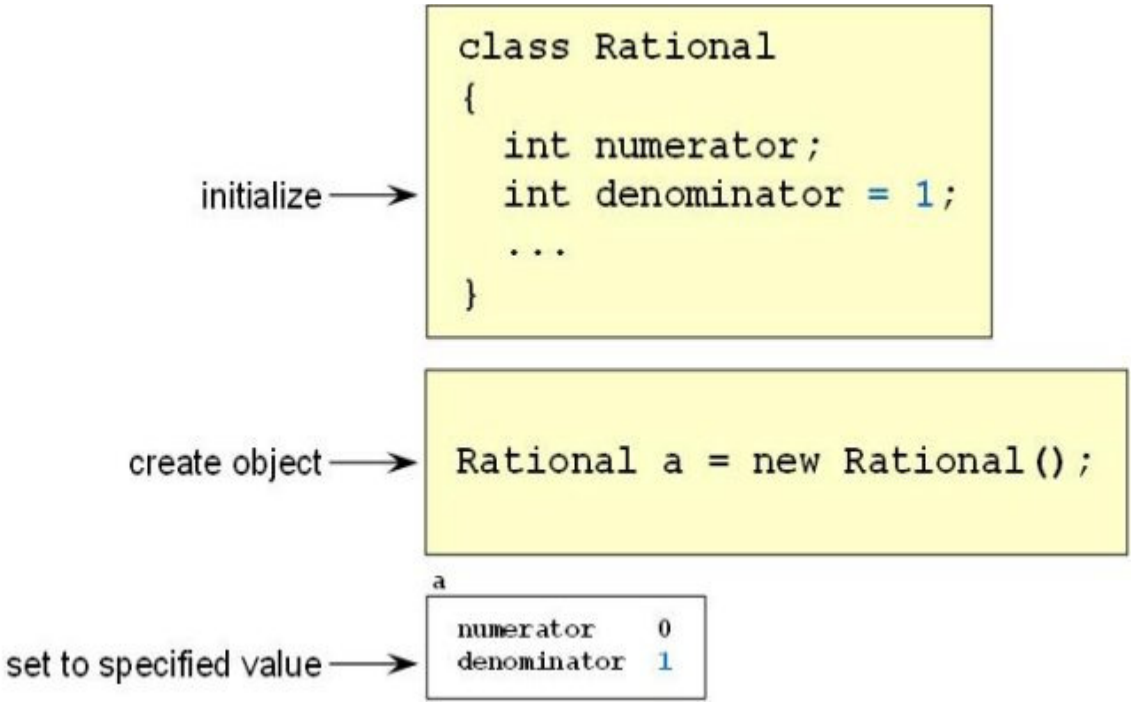
Characters fields are set to the null character (often written `'\x0000'`).

References are set to `null`. We will discuss references in the module on reference types.



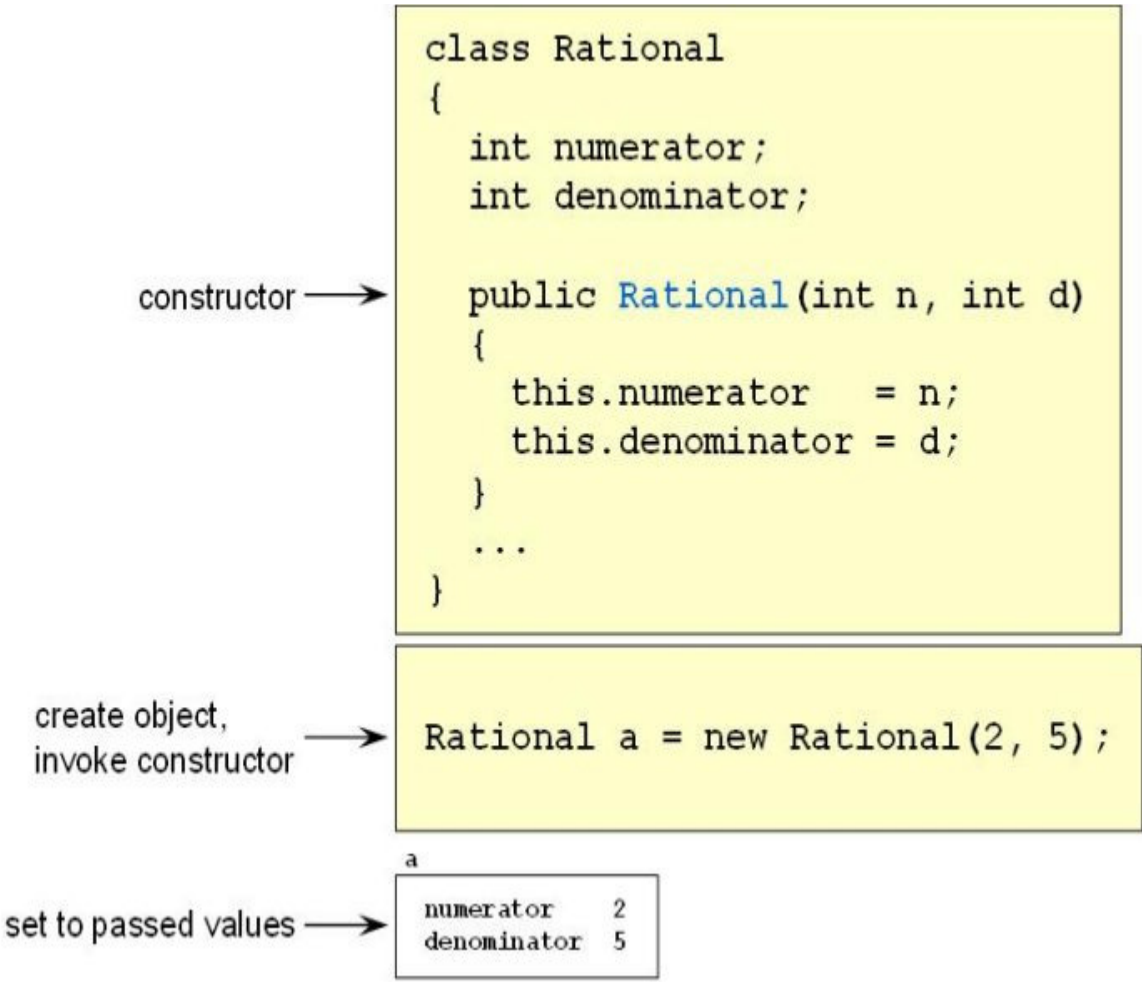
Variable initializer

Instance fields can be initialized inline, that is, at the point of declaration. The official name for this technique is 'variable initializer'.



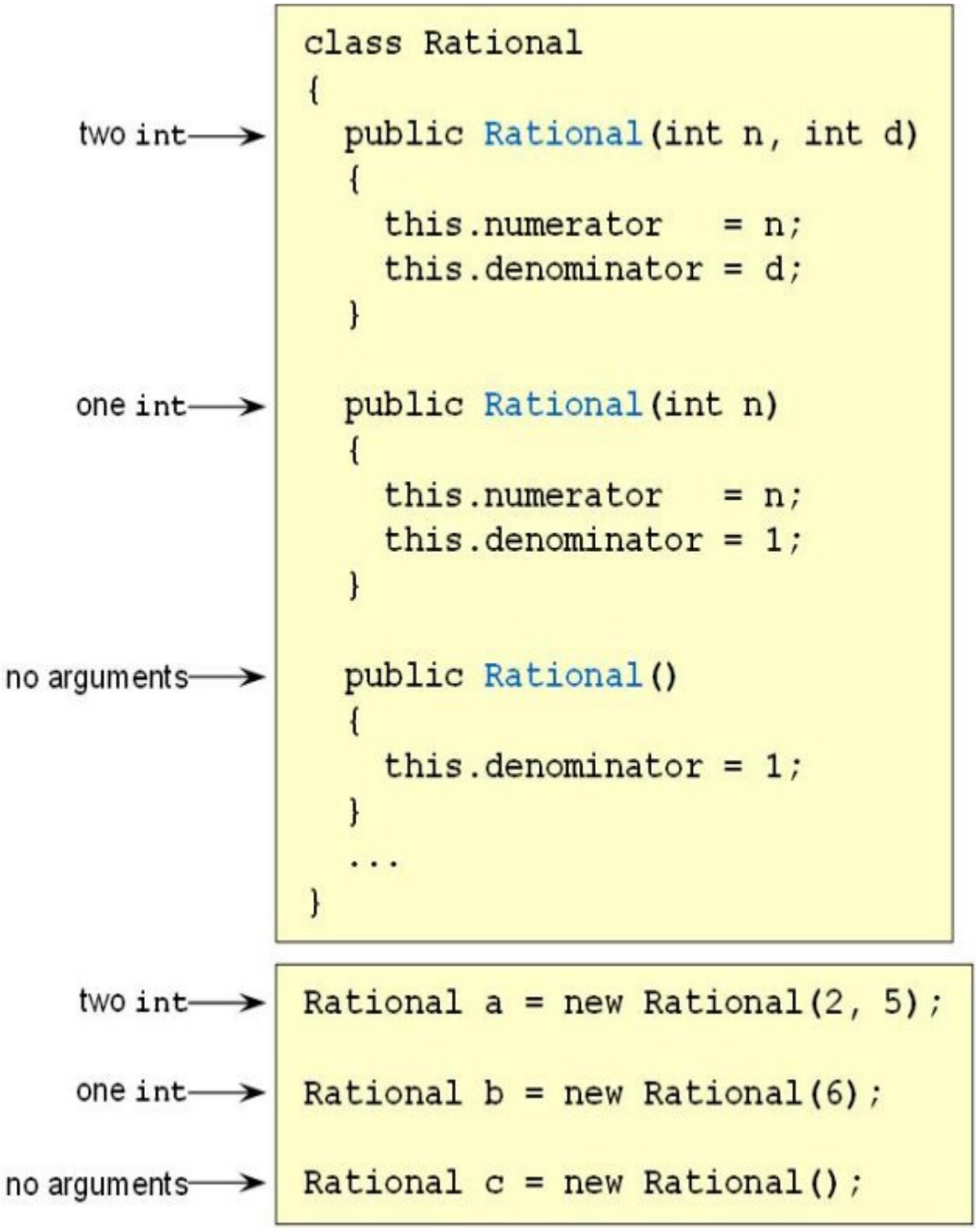
Constructor

The most powerful initialization technique is to provide a constructor. A constructor is a special method that is called automatically each time an object is created. The definition of a constructor is similar to a regular method except for two small differences: the name of the constructor must be the same as the name of the enclosing class and no return type is allowed. The client code does not invoke the constructor directly, instead the client creates objects using the `new` operator and passes any needed arguments inside the parentheses after the type name. The constructor is called automatically as part of the object creation and initialization process.



Constructor overloading

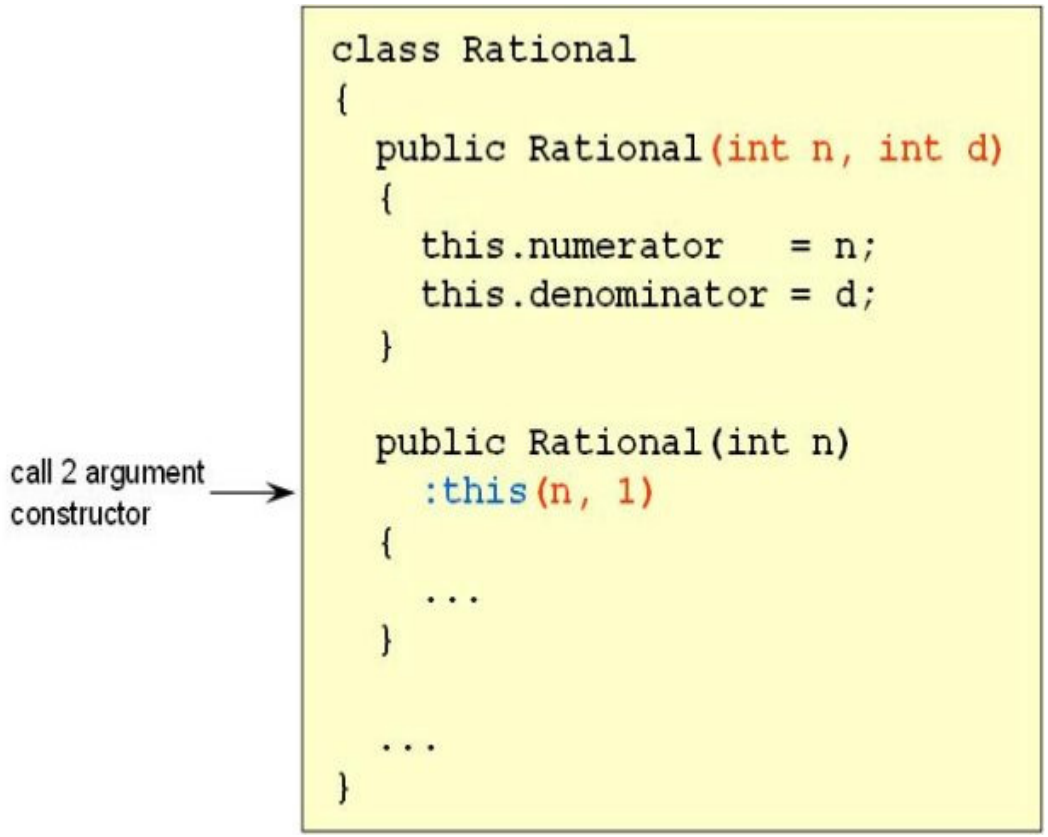
A class can supply multiple constructors with different parameter lists. It is even possible to write a constructor that takes no arguments. The no argument constructor is sometimes called the 'default constructor'. The client code chooses the constructor to call by passing the appropriate parameters when an object is created.



Constructor initializer

One constructor can call another constructor in the same class. The syntax is `:this(...)` placed after the constructor signature but before the constructor body. The number and type of arguments passed determine which constructor version gets called. This technique is called a "constructor initializer". The main benefit is that it allows common

initialization code to be placed in one constructor and any other constructors can simply make a call rather than repeating the code.



Compiler generated constructor

If a class does not define any constructors, then the compiler automatically generates a constructor that takes no arguments and has an empty body. Conversely, if a class defines even one constructor, then the compiler assumes the class designer is taking control of construction and does not create a constructor.

Initialization order

The various initialization options are executed in a well defined order. First, all fields are set to their default values. Second, the variable initializers are executed in order of their declaration. Finally, the constructor is run.

03- Initialization Exercise

Goals:

- Practice with variable initializers for instance fields.
- Write instance constructors.
- Have one instance constructor call another.

Overview

Initialization of data before use is relatively important for program correctness. C# offers three options for initializing fields and provides well defined rules for the execution order of the various options. The first initialization option for fields is to simply accept their default values. The numeric types get set to zero, Booleans to false, characters to the null character, and references to null. This case is relatively simple so we will not bother to test it in this exercise. The second initialization option is to assign an initial value at the point of definition. This case is also simple enough that we will not spend time testing it. The third initialization option is the use of constructors. This is the most complicated and powerful option and so will be the main topic covered in this activity.

Part 1- Instance fields

Here we explore the two most interesting options for initializing an instance field: variable initializers and constructors. Initializing an instance field at the point of definition is called a "variable initializer". Constructors are initialization functions that get called automatically when an object is created.

Steps:

1. Here we will work with a Book class which represents a library book. We will store the title and author along with a flag to indicate whether the book is checked out. Implement a

constructor for the Book class that takes three arguments: the title, author, and flag. To verify that your code is working correctly add a WriteLine statement to the constructor that prints the values being used for the initialization. To test your work, create a class called BookTest containing a Main method. In the Main method create a book object and pass arguments to the constructor.

```
class Book
{
        private string title;
        private string author;
        private bool   available;

        public Book(string title, string author, bool available)
        {
                ...
        }
        ...
}
```

13. Add a second constructor to Book that takes only the title and author. Set the flag to true. Modify the Main method to create a Book using this new constructor.

```
class Book
{
        public Book(string title, string author)
        {
                ...
        }
        ...
}
```

21. Add a third constructor to Book that takes only the title. Set the author to "anonymous" and the flag to true. Modify the Main method to create a Book using this new constructor.

```
class Book
{
        public Book(string title)
        {
                ...
        }
        ...
}
```

29. Add a default (no argument) constructor to Book. Set the title to "untitled", the author to "anonymous", and the flag to true. Modify the Main method to create a Book using this new constructor.

```
class Book
```

```
{
        public Book()
        {
                ...
        }
        ...
}
```

37. C# allows one constructor to call another constructor in the same class using a "constructor initializer". A constructor initializer can be used to eliminate repeated code. The most general constructor is fully implemented while the more specialized cases call the general one passing in whatever user specified data they have available and filling in the rest with hardcoded values. Rewrite the Book class constructors to take advantage of constructor initializers. The 3 argument constructor should be fully implemented and all other constructors should call it using the constructor initializer syntax.

```
class Book
{
        public Book(string title, string author)
                :this(title, author, true) // constructor initializer
        {
                ...
        }
        ...
}
```

04-Static

Goals:

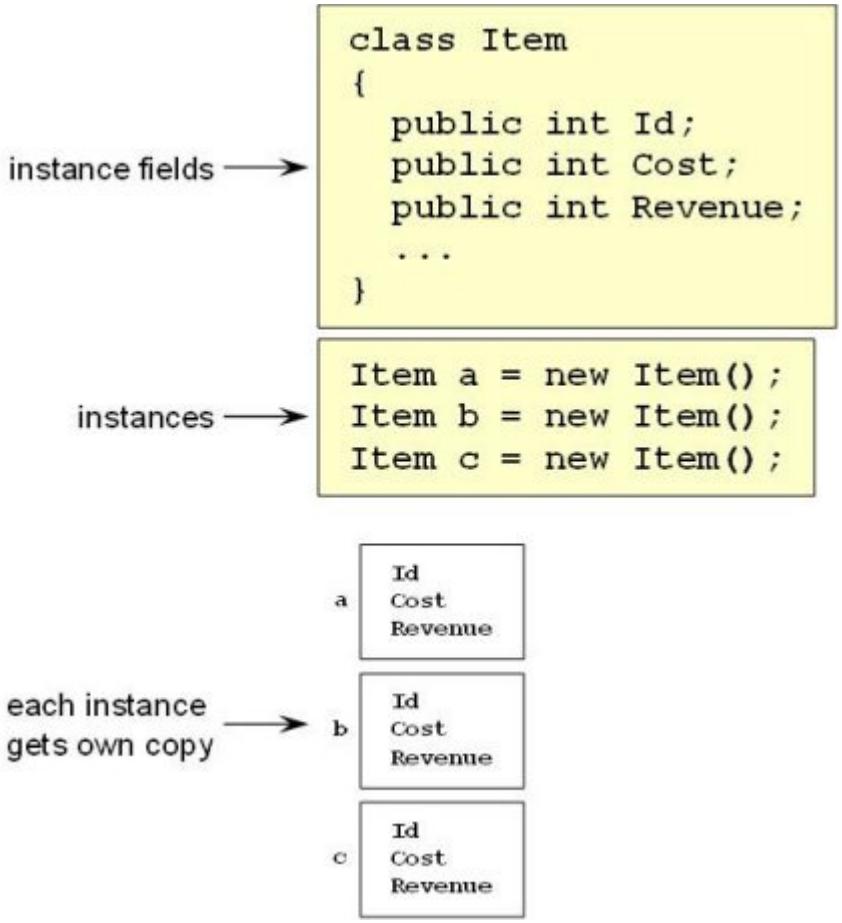
- Introduce the concept of static fields and methods.
- Show how to declare static fields.
- Show how to implement static methods.
- Show how to access static members.
- Discuss the three initialization options for static fields: default value, inline initialization, and static constructor.

Overview

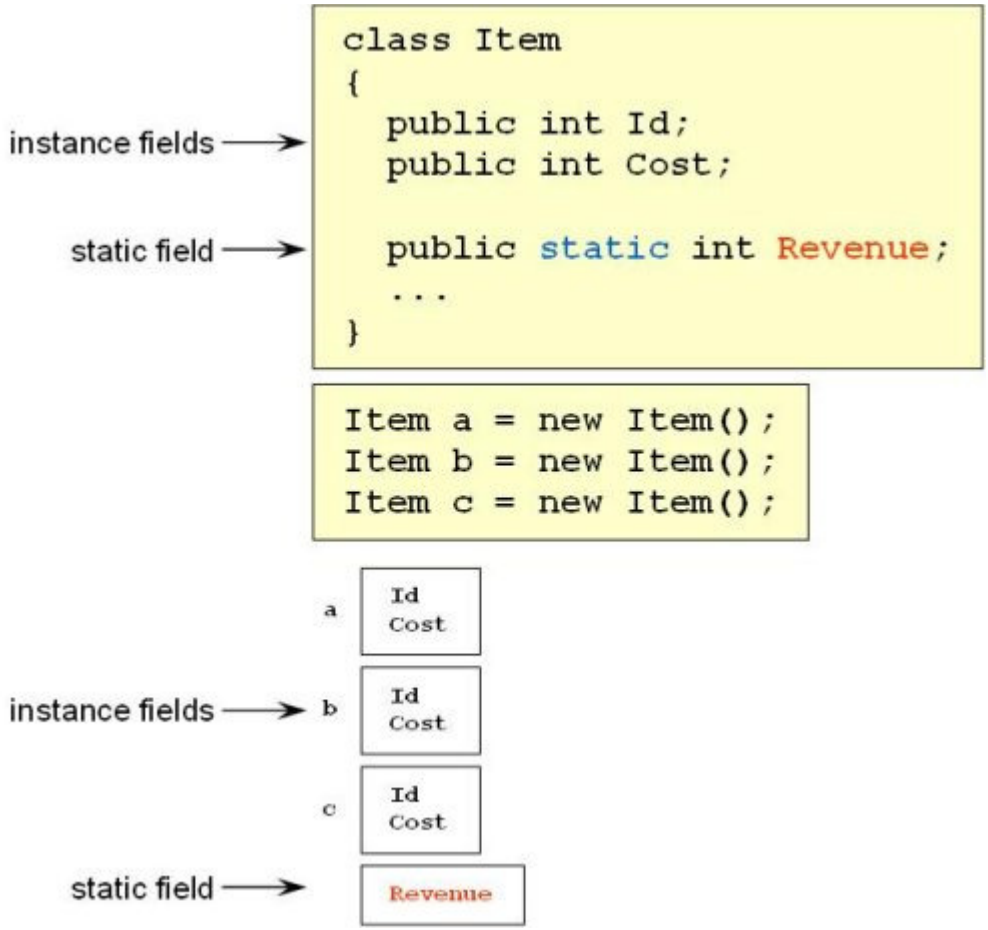
The keyword `static` can be applied to fields and methods. Static fields implement the idea of data that is shared by multiple clients. Static methods are typically either access methods to static data or utility methods that do not need to maintain state across calls.

Static field

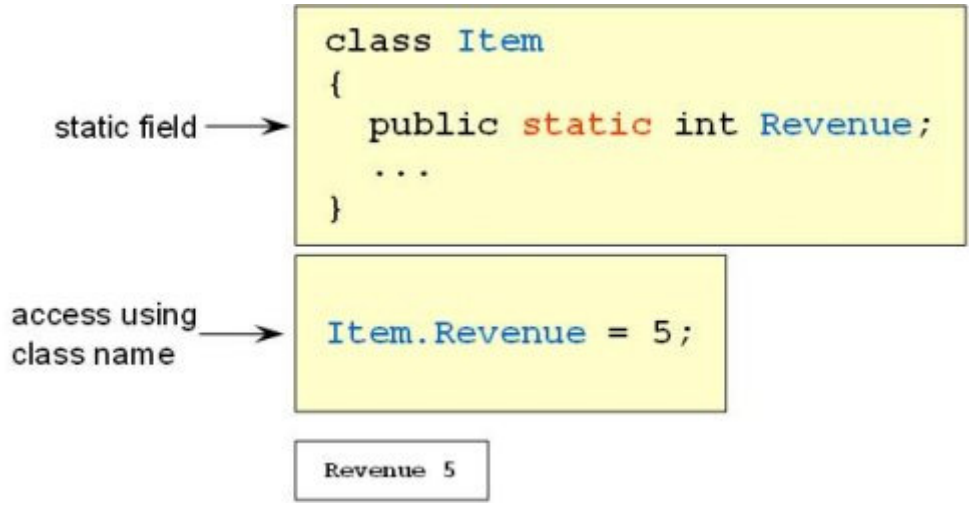
Regular fields are often called "instance fields" because each instance of the class gets their own copy. Consider the following `Item` class containing three instance fields and notice how the objects look in memory.



By contrast, there is only one copy of each `static` field and the single copy is shared by all objects. Consider the new version of the `Item` class shown below. The `Revenue` field has been made `static`; therefore, there is a single copy generated that is separate from all `Item` objects.



Client code that needs to use a static field must access it using the class name.



Static fields are used to implement the concept of "shared resource" since the single copy can be shared by multiple clients. In other languages, this role is often filled by global variables. The type designer can choose the access level for a static field: private static

fields are shared only by instances of that type while public static fields are shared by all clients.

The memory for static fields is allocated by the CLR when the class is initialized. Conceptually, you can think of this as occurring "at program startup"; however, the exact timing is up to the CLR and in many cases this setup will occur later. The key thing to take away is that the memory for a static field will be available for use even if no objects of that type have been created.

Static field default values

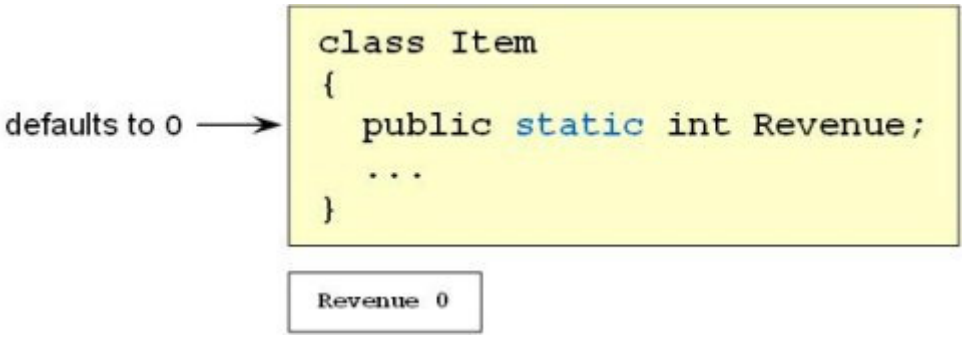
Static fields of a class are set to default values when the class is initialized by the CLR. The default values depend on the type of the field.

Numeric fields such as `int`, `float`, `double`, etc. are set to zero.

Fields of type `bool` are set to `false`.

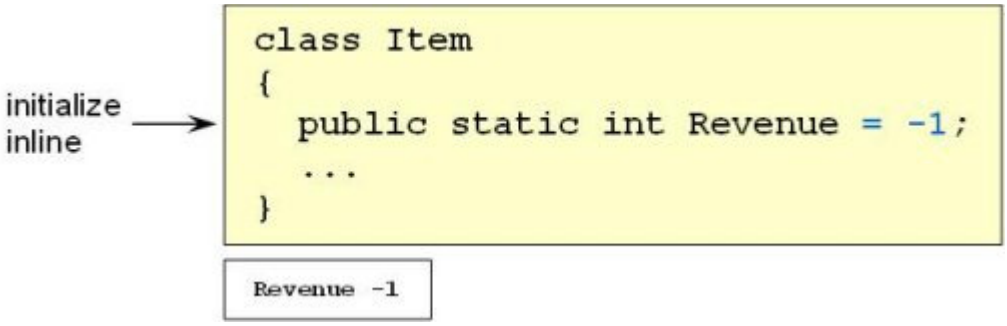
Characters fields are set to the null character (often written `'\x0000'`).

References are set to `null`. We will discuss references in the module on reference types.



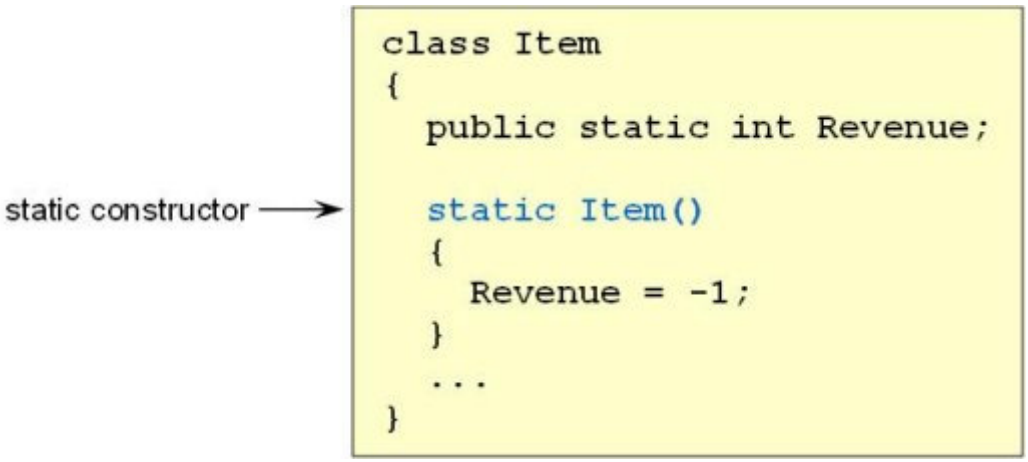
Static variable initializer

Static fields can be initialized inline, that is, at the point of declaration. The official name for this technique is 'static variable initializer'.



Static constructor

The most powerful initialization technique is to provide a static constructor. The static constructor is defined using the keyword `static`, the class name, and an empty argument list. No return type, access modifier, or parameters are allowed. The CLR invokes the static constructor automatically when the class is initialized. The exact timing of the invocation is not guaranteed, but it will be run after the static variable initializers have been executed.



Static method

A method can be made static by adding the keyword `static` to its declaration.

static method →

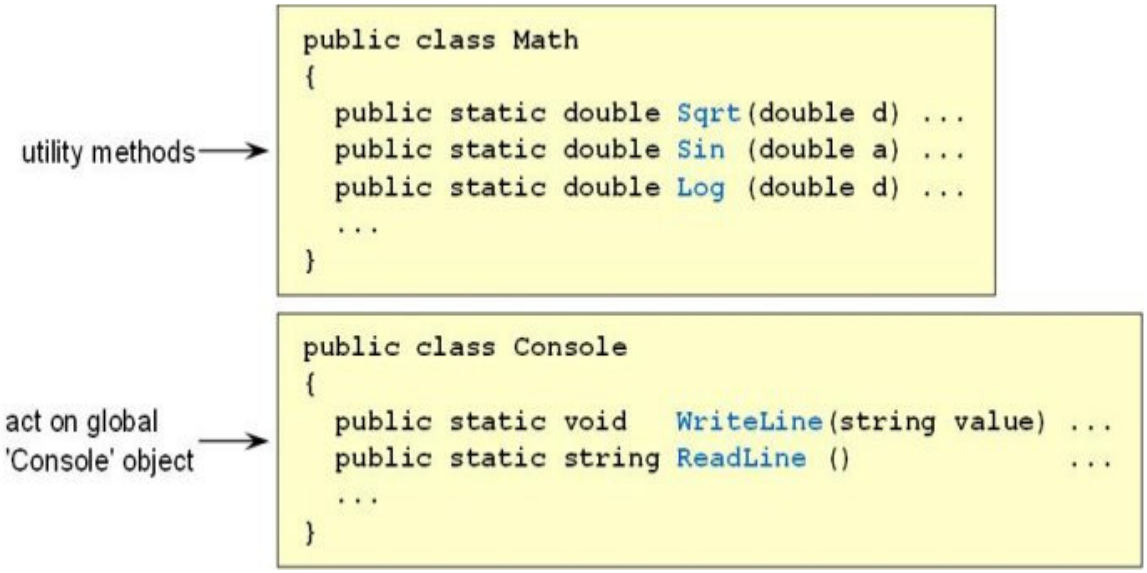
```
class Item
{
  private static int revenue;

  public static void ResetRevenue()
  {
    revenue = 0;
  }
  ...
}
```

A static method must be invoked using the type name.

call static method →

```
Item.ResetRevenue();
```

Static methods are not as powerful as instance methods. In particular, static methods can only access the static parts of their type and are not allowed to access any instance fields or methods. This limitation means static methods are not quite as useful as instance methods; however, static methods do still have their place. Static methods are typically used for 'utility' methods that do not need to maintain state across calls. Several good examples of this application can be seen in the `Math` and `Convert` classes from the .NET Framework Class Library. A less common application of static methods is as accessors or manipulators of some 'global' object. The classic example of this is the library `Console` class where the read and write methods manipulate the console attached to the executing process.


utility methods
public class Math
{
public static double Sqrt(double d) ...
public static double Sin (double a) ...
public static double Log (double d) ...
...
}
act on global
'Console' object
public class Console
{
public static void WriteLine(string value) ...
public static string ReadLine () ...
...
}

04- Static Exercise

Goals:

- Code a class with static fields and methods.
- Practice with variable initializers for static fields.
- Write a static constructor.

Overview

Intuitively, static fields have much in common with the "global variables" used by other languages: the key idea is that there is one copy of each static field that is shared by all clients.

C# offers three options for initializing static fields and provides well defined rules for the execution order of the various options. The first initialization option is to simply accept their default values. The default value depends on the type: numeric types get set to zero, Booleans to false, characters to the null character, and references to null. The second initialization option is to assign an initial value at the point of definition. The third initialization option is the use of a static constructor. All the static initialization code is run only once since their is only one copy of the static data.

Methods can be static. Code inside a static method can only access the static parts of the class, instance fields and methods are not available.

Both static fields and methods must be accessed only through the classname, never through an instance. This rule is sensible because the statics are associated with the class as a whole and not with a particular instance.

Part 1- Static field

Here we will use a static field to generate id numbers for a student class. The field will be static since we want all students to draw from the same source of id numbers; that is, we want a shared resource.

Steps:

1. Consider the following Student class. There are currently two fields: name and age. The constructor takes the user data and assigns it to the fields. There is no additional code required for this part of the exercise.

```
class Student
{
        string name;
        int    age;

        public Student(string name, int age)
        {
                this.name = name;
                this.age  = age;
        }
}
```

12. Each student needs an id number to uniquely identify them at their university. Add an integer field to the student class to represent the id number.
13. The student id number will be generated inside the student class using a static field to store the next value to be assigned. A static field is appropriate since we want all students to get their id number from the same source in order to guarantee a unique id number for each student. Add a static nextId field to the Student class. Use a static constructor or inline initialization to set the initial value to 1. Modify the constructor to assign from nextId to the id field. Be sure to increment the nextId field after using its current value.
14. Add a driver class with a Main method. Create a few Student objects to test your implementation.

Part 2- Static methods

Static methods are often used for small "utility" operations that take in some data through parameters, perform a calculation, and return the result. These methods do not need a this object associated with the call since they get all the information they need through their parameters. Making the methods static also has the advantage that they are easier for clients to use since they are invoked simply using the class name

rather than through an instance of the class.

Steps:

1. Create a class called MathUtils. Add two public static methods Min and Max. The methods should take two int parameters, perform their operation, and return the result.
2. Create a driver class with a Main method. Call the Min and Max methods to test your work.

Part 3- Random number generator (optional)

Here we create a class that supplies a convenient random number generator. The class will make use of a static field and a static method. It will also give practice with the options for initializing a static field: static variable initializer and static constructor.

The .NET Framework class library provides a class named System.Random that we will use to implement our random number generator. The Random class will do most of the work for us; however, Random requires users to instantiate an object of the Random class and call instance methods to generate random numbers. This type of interface makes it easy for the Framework library designer since the Random object needs to store state information. Forcing users to instantiate the class ensures that memory is allocated for the use of the generator.

This interface is not ideal for users who would like one generator that can easily be shared by their entire application. To support this model of programming we will create a class that wraps a Random object with a static method interface.

Steps:

1. Complete the implementation of the following RandomNumberGenerator class. The generator exposes just one method Next which returns a random number between 0 and bound-1. You will need to add a static field of type System.Random. Use a variable initializer to create a Random object and assign it to your field. The default constructor for the Random class uses

the current time to create a seed. This will give a different sequence of values each time the program is run so it is a good choice for our use.

```
2.    class RandomNumberGenerator
3.    {
4.              public static int Next(int bound)
5.              {
6.                        ...
7.              }
8.              ...
      }
```

9. Using a static variable initializer to allocate the Random object is less than ideal because the entire initializer must fit in one line of code and the required expression is relatively complicated. Modify the implementation of the random number generator to use a static constructor to perform the initialization. Use of a static constructor will allow the initialization code to be spread across several lines which should make it more readable.

```
10.   public class RandomNumberGenerator
11.   {
12.             static RandomNumberGenerator()
13.             {
14.                       ...
15.             }
16.             ...
      }
```

17. Add a Driver class with a Main method to test your work.

05-Reference Types

Goals:

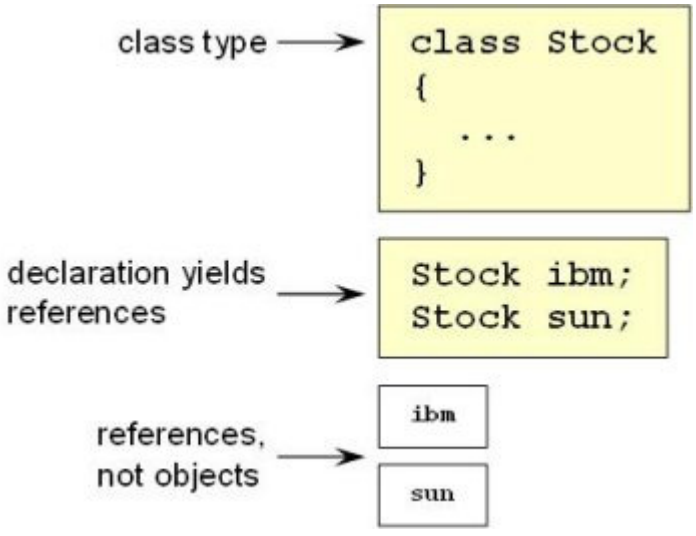
- Introduce the concept of references.
- Describe the behavior of references in declaration, assignment, and parameter passing.
- Discuss issues surrounding null references.
- Briefly talk about memory management and garbage collection.

Overview

Arrays and classes are reference types. This means that two things are involved when dealing with instances: a reference and the object itself. A reference is used as a handle to manipulate the object. This organization has a number of implications for how reference types behave during operations such as assignment and parameter passing.

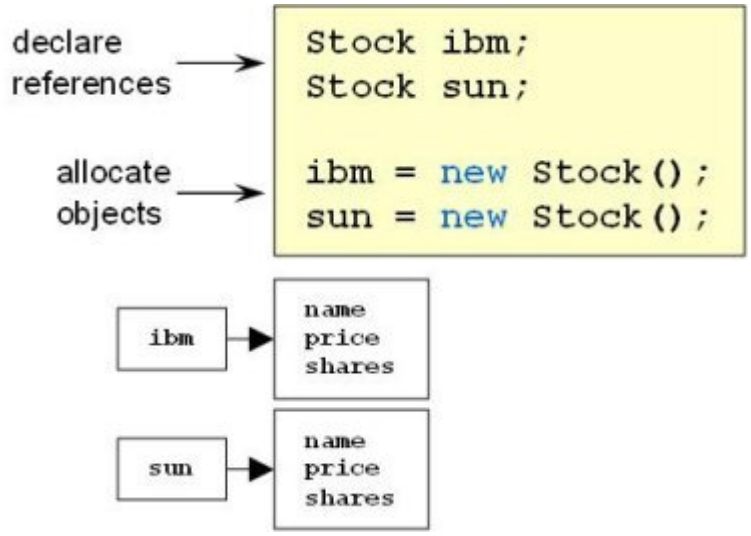
Reference declaration

A variable declaration of a class type yields only a reference, it does not create an object of that class. References are often called "handles" or "pointers" since they are used to manipulate or point at objects.

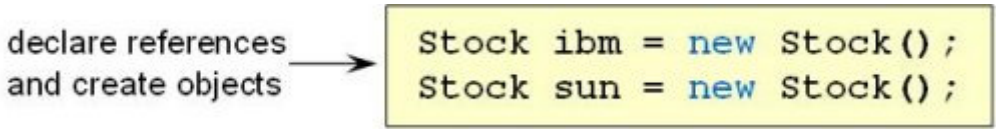


Object creation

Objects are created using the `new` operator and reference variables are used to access them. To see this, it is instructive to look at some code where the two steps are done separately. In the example shown below, first a few `Stock` references are created and then objects are allocated using `new`. Notice how the assignment operator is used to make the reference variables refer to the newly created objects.

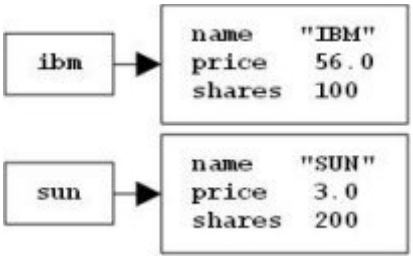


It is also possible to do both the reference declaration and object creation in a single line of code as shown below.

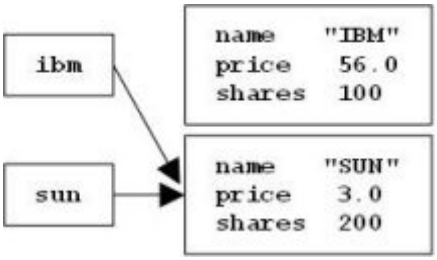


Reference assignment

Assigning references copies only the reference, it does not copy the data inside the object. To see this, consider the following code which shows two references each referring to a separate object.



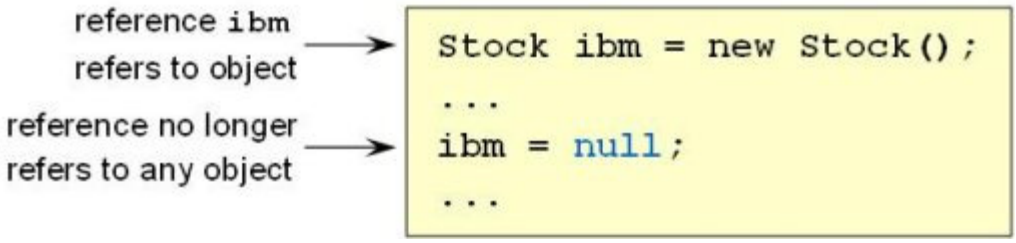
Now consider performing an assignment such as "`ibm = sun;`". Because `ibm` and `sun` are both references, the assignment copies only the reference. In other words, `ibm` now refers to the same object as `sun`. The result of such an assignment is shown below. Note that the object `ibm` used to refer to is no longer accessible since there are no references to it. We will discuss what happens to such objects later in this module when we talk about memory management.



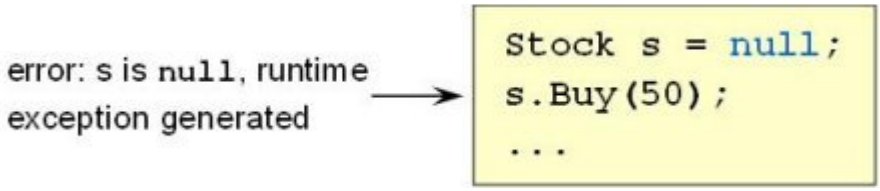
---

null

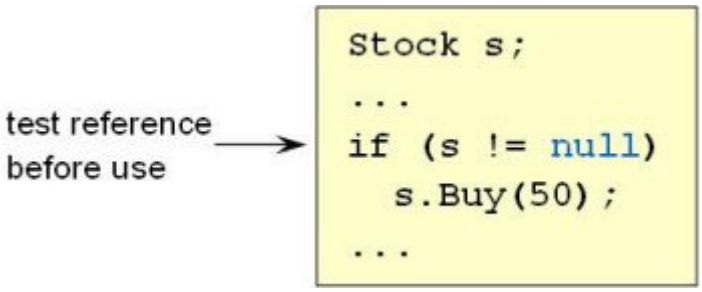
The link between a reference and an object can be broken by setting the reference to `null`. For example, in the code below, the reference `ibm` is initially referring to an object. After the assignment, the link has been severed and the reference is no longer referring to any object.



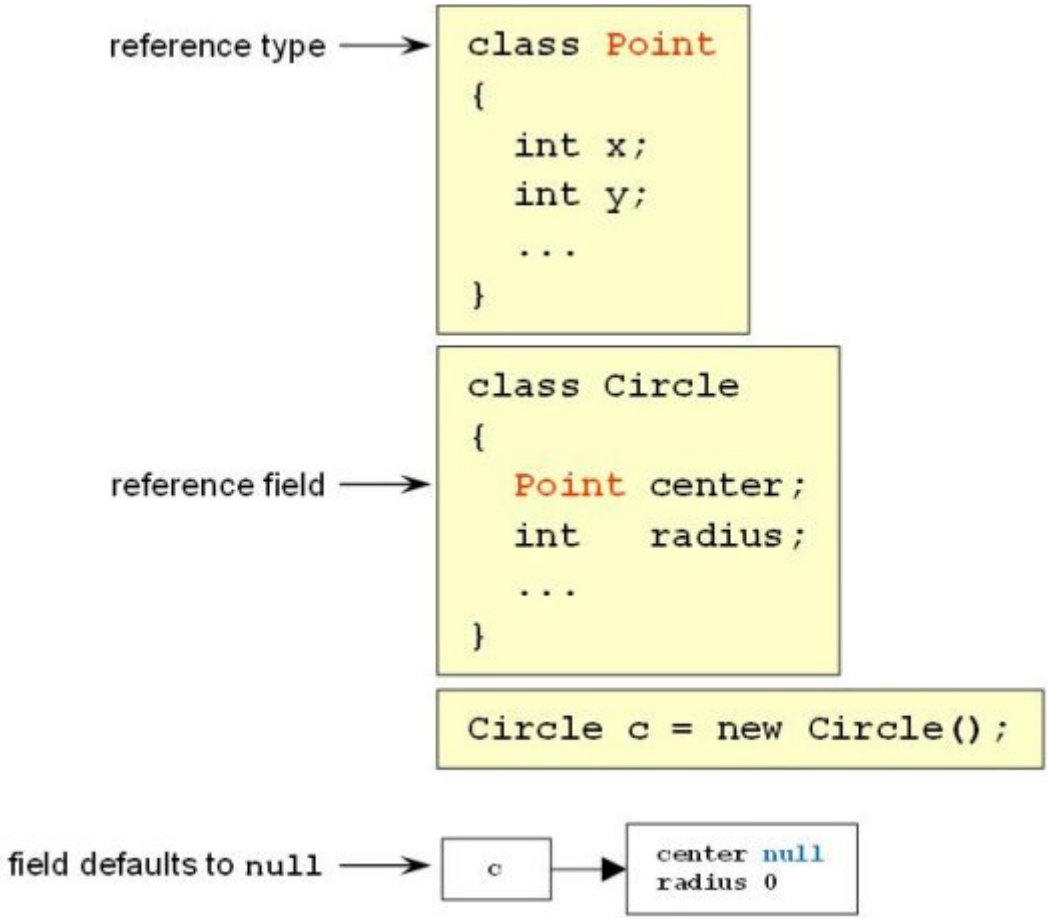
Attempting to use a null reference is an error. The error is detected at runtime by the CLR. The CLR stops the operation and throws a `NullReferenceException` to notify the program of what went wrong. The `try`/`catch` exception handling syntax can be used to trap and handle the exception. We will discuss exceptions in greater detail in a later module.



In order to avoid a `NullReferenceException`, it is common to test a reference before use. Only if the reference is not null should the operation proceed.

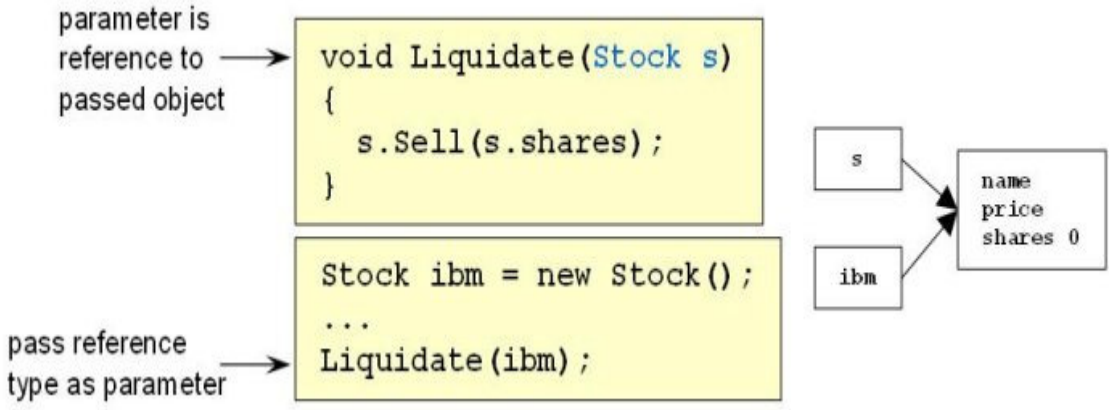


It is common to use a reference type as a field. For example, a parent might hold a reference to each child, a highway would need to keep references to each city it connects, and a circle must have a reference to its center point. Recall that each field gets assigned a default value based on its type: numeric fields get set to zero, Boolean fields are set to false, etc. In a similar fashion, reference fields are set to `null` when the object containing them is allocated. The code below shows the case for "circle with center point".



Reference parameters

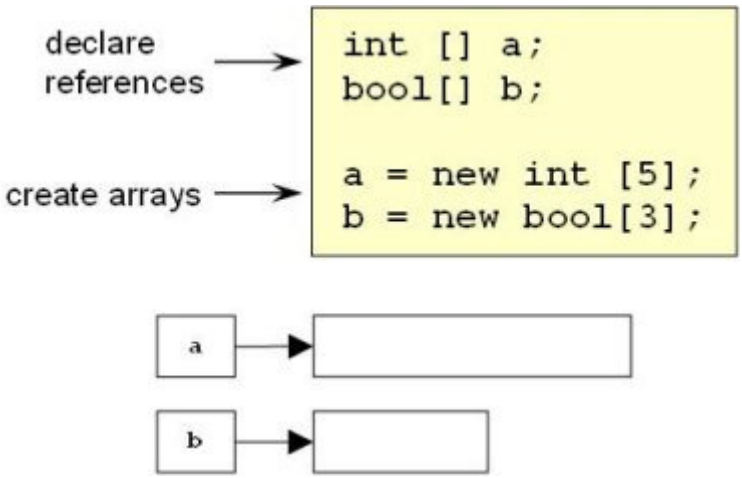
When reference types are passed as parameters, only the reference is passed and the object is not copied. The figure below illustrates the situation.



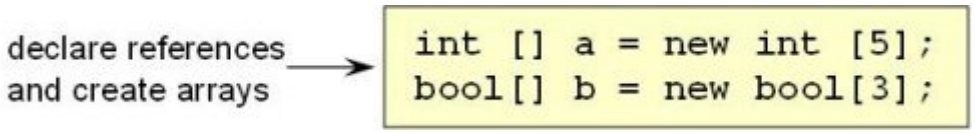
Passing a reference has two important implications. First, reference parameters can be quite efficient since only a reference is passed and there is no need to copy all the data inside the object. Second, because the method holds a reference to the original object, any changes it makes modify that object. The effects of these changes will be visible to the client code after the method returns.

Array

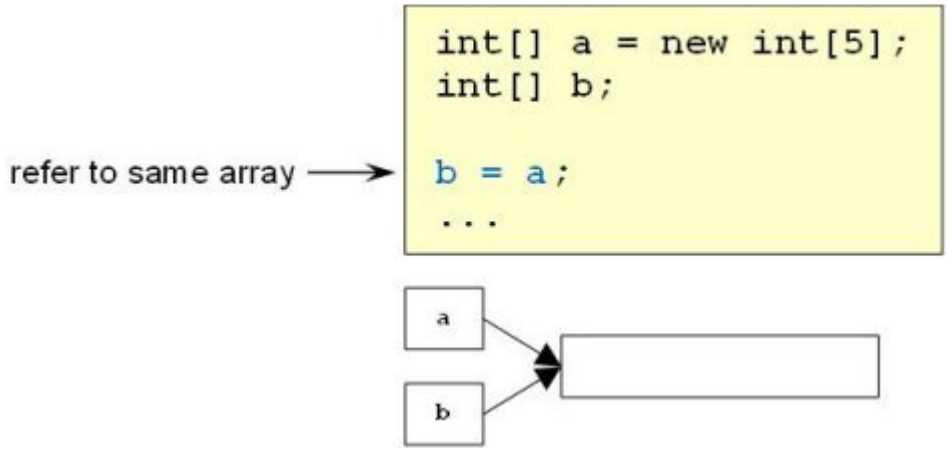
Arrays are reference types just like class types so they are implemented as reference/object pairs. An array declaration creates only a reference. The associated array object must be created using the `new` operator. The figure below illustrates the two steps needed to create an array.



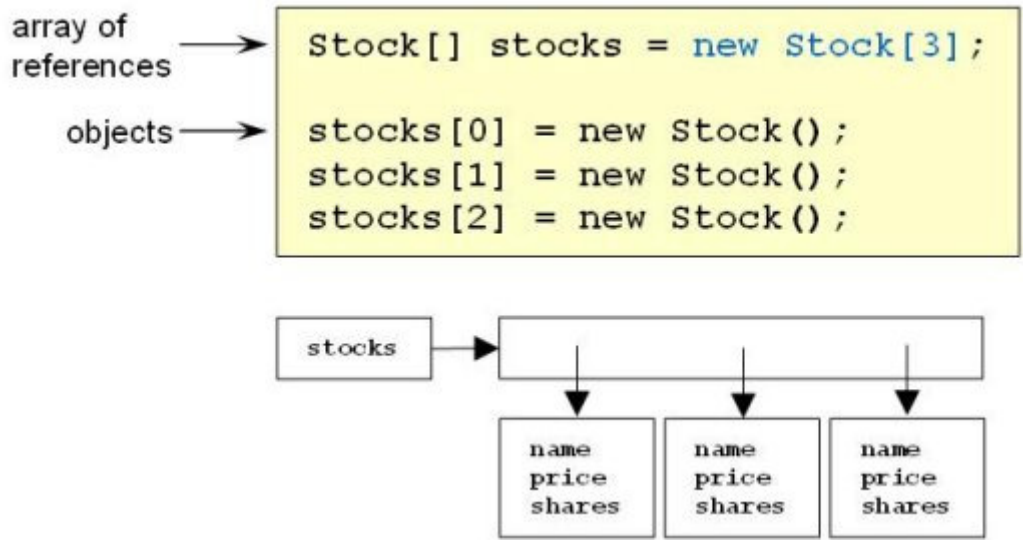
It is common to declare the reference and create the array in a single line of code. This style is a bit cleaner since the reference is created and then immediately bound to an array object. The figure below shows the two steps being performed together.



Because arrays are references type, they behave just like class types in operations such as assignment and parameter passing. For example, assigning one array to another copies the reference and not the data inside the array. This situation is shown below.



As a final variation on arrays, we will look at a slightly more complicated case: an array of references. Creating an array of a class type with an expression such as "`Stock[] s = new Stock[3];`" creates an array of `Stock` references with each reference initially set to `null`. It is important to emphasize that at this point we have only an array of references, there are no `Stock` objects in existence yet. As a second step, we need to create the `Stock` objects and use the array elements to refer to them. The situation is illustrated below.

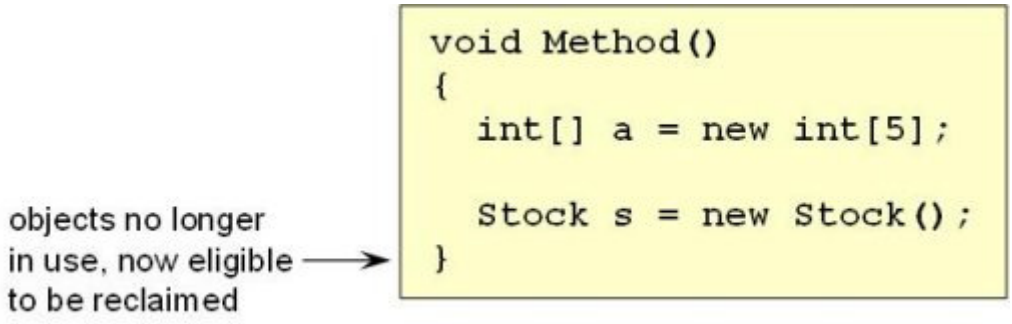


Memory management

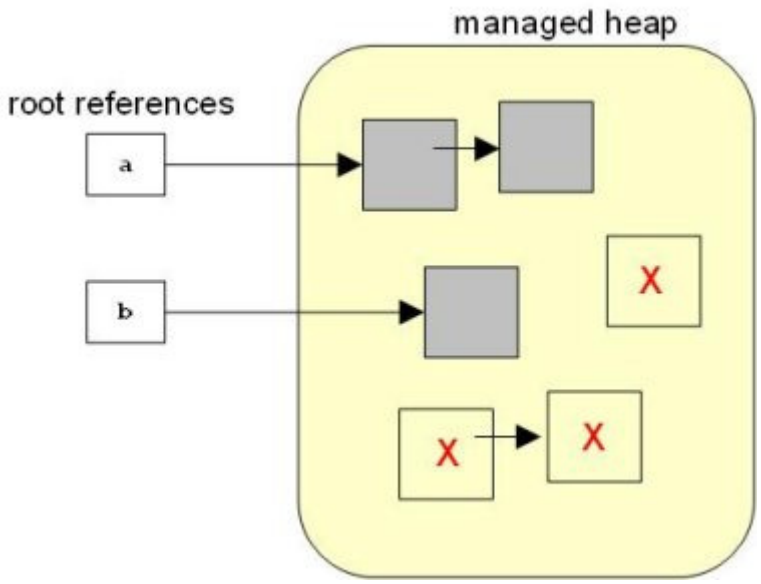
In the last few modules we have allocated quite a few objects using `new` but have not worried about where the memory comes from nor how it gets reclaimed when we have finished with the objects. Happily, there is not much for the programmer to do since C# and .NET automatically recycle the memory of unused objects. The situation is similar to what many children experience during their earliest years. During play time, the child removes a toy from the toy box and plays with it for a while. When they grow bored they

simply set the toy down where they are and return to the toy box for another one. After a while, there are discarded toys strewn all around the house. At this point, a responsible adult comes along and collects all the toys and returns them to the toy box where they are available for reuse.

More technically, memory for objects of reference types comes from an area of memory called the "managed heap". The .NET heap is called "managed" since the CLR takes responsibility for managing the allocation and reclamation of the memory. Operator `new` allocates memory for reference types from the managed heap and the "garbage collector" reclaims the memory occupied by unused objects. An object becomes unused when it is no longer reachable from the program. For example, if the only reference to an object is a local variable, then at the end of the method the local variable goes out of scope and the reference disappears. The object is now unused and is eligible to be reclaimed by the garbage collector.



The garbage collector has quite a difficult job since it must determine which objects are still active and which objects are no longer in use by the program. To do this, the garbage collector examines all the "root" references such as live local variables and static fields. It follows these references and marks any objects it finds as being still active. After all live objects have been marked, the garbage collector assumes all other objects are no longer in use and recycles the memory they occupy. The basic idea is illustrated in the figure below.



The CLR controls the timing of garbage collection. The common advice is that the CLR and the garbage collector are intelligent enough to do the job without any interference from the programmer. However, the `GC.Collect` method can be used to force the garbage collector to run if it becomes necessary.

Despite the sophisticated support for memory management offered by the CLR, it is still possible for a program to allocate so many objects that it runs out of available memory. When this happens, the CLR will notify the program by throwing an `OutOfMemoryException`. The program can use the exception handling `try`/`catch` construct to trap and handle the exception.

05- Reference Types Exercise

Goals:

- Create a reference/array pair.
- Make a class with a reference field.
- Pass a reference as a parameter.
- Practice with an array of references.

Overview

In this lab we create a beehive simulation that demonstrates the key features of reference types. There will be three main classes involved: Hive, Bee, and Keeper. In addition there will be a simple test class called Driver with a Main method to get the simulation started.

Part 1- Hive

We begin the simulation with a class to represent the beehive. The main job of the hive is to contain the cells in which the nectar ferments and becomes honey.

Steps:

1. Create a class Hive. Add a Cells field to represent the cells: the field should be a reference to an array of int and should be public to allow the access by the bees. Note that the field is just an array reference, there is not array at this point.

```
class Hive
{
        ...
}
```

5. Define two public integer constants Empty and Full to indicate the state of each cell. Note the use of the keyword const to create symbolic constants whose value cannot be changed.

```
class Hive
{
        public const int Empty = 0;
```

```
9.          public const int Full  = 1;
10.         ...
    }
```

11. Write a constructor for Hive. The constructor should take an integer argument specifying the number of cells the hive should contain. Allocate an array of the requested size using new and store the array in the Cells field. Initialize each cell to Empty.

```
12.  class Hive
13.  {
14.         public Hive(int size)
15.         {
16.                 ...
17.         }
18.         ...
    }
```

Part 2- Bee

Next we need a class to represent the bee. The bee's job is to go out into the countryside, gather nectar, return to the hive, and deposit the nectar into an empty cell.

steps:

1. Create the Bee class.

```
2.   class Bee
3.   {
4.          ...
    }
```

5. We will need some randomization to make the simulation interesting. Add the following simple random number generator to the Bee class. The Next method of the Random class takes an int argument called the bound and returns a random number between 0 and bound-1. The Random class is part of the standard library.

```
6.   class Bee
7.   {
8.          private Random random = new Random();
9.          ...
    }
```

10. The bee's job is to search for nectar, gather the nectar, return to the hive, and deposit the nectar in one of the cells. Thus, the bee is always in one of four states: searching,

gathering, returning, or depositing. Add symbolic integer constants for each of the four states and an integer field that records the current state. The constants and the state field will only be needed inside the bee class so they can be private.

11. A bee lives in a hive. To model this, add a Hive reference as a field of the bee class. Note that this field is not a hive embedded inside the bee, it is simply a reference to a hive that exists outside of the bee.
12. Add a constructor to the bee class. The construction should take a Hive reference as a parameter. Inside the constructor set the bee's hive field to the parameter hive and set the bee's state to searching.

```
class Bee
{
        public Bee(Hive hive)
        {
                ...
        }
        ...
}
```

20. The bee work algorithm is concisely captured using a state machine. A bee is in one of four states: searching, gathering, returning and depositing. During each time period, each bee performs work depending on its current state. When searching, a bee has a one in five chance of finding a flower with nectar. If nectar is found, the bee starts gathering. If nectar is not found, the bee continues searching. It takes one time unit for a bee to gather the nectar, then it returns to the hive. It takes one time unit for a bee to return to the hive, then it deposits the nectar in a cell. In order to deposit the nectar the bee must find an empty cell. The bee chooses a cell at random and checks if the cell is empty. If the cell is empty, the bee fills it and then begins searching again. If the cell is full, the bee keeps looking for an empty cell. The bee uses its hive reference field to access the hive of which it is a member. This is captured by the following Work method. Feel free to use this method in your program or to code it yourself.

```
class Bee
{
        public void Work()
        {
                switch (state)
                {
                        case Searching:
                        {
                                Console.Write("S");
```

```
30.
31.                                     if (random.Next(5) == 0)
32.                                             state = Gathering;
33.
34.                                     break;
35.                             }
36.                             case Gathering:
37.                             {
38.                                     Console.Write("G");
39.
40.                                     state = Returning;
41.
42.                                     break;
43.                             }
44.                             case Returning:
45.                             {
46.                                     Console.Write("R");
47.
48.                                     state = Depositing;
49.
50.                                     break;
51.                             }
52.                             case Depositing:
53.                             {
54.                                     Console.Write("D");
55.
56.                                     //
57.                                     // choose random cell to deposit
58.                                     //
59.                                     int i = random.Next(hive.Cells.Length);
60.
61.                                     if (hive.Cells[i] == Hive.Empty)
62.                                     {
63.                                             hive.Cells[i] = Hive.Full;
64.                                             state = Searching;
65.                                     }
66.
67.                                     break;
68.                             }
69.                     }
70.             }
71.             ...
    }
```

Part 3- Keeper

The bee keeper will run the simulation. The keeper will create the hive, create the bees, call the bee's Work method, and periodically collect the honey from the hive.

Steps:

1. Create the Keeper class.

```
class Keeper
{
        ...
}
```

5. Add a method to Keeper to collect all the nectar from a hive. The method takes a Hive parameter, counts the number of filled cells inside the hive, sets all the cells to empty, and returns the count of the number of filled cells. Notice that the parameter is a reference so any changes made to the parameter will be visible back in the calling code.

```
class Keeper
{
        public int Collect(Hive hive)
        {
                ...
        }
        ...
}
```

13. Add a Run method to the keeper class.

```
class Keeper
{
        public void Run()
        {
                ...
        }
}
```

20. Inside Run, create a hive with 20 cells.
21. Inside Run, create an array of Bee references. Note that creating an array of a reference type creates an array of references and not an array of objects. When the Bee array is created we get an array of Bee references all initialized to null - at this point we have no Bee objects. Next, loop through the array and create a Bee object for each slot in the array. Pass the Hive into the Bee constructor as you create each bee.

```
class Keeper
{
        public void Run()
        {
                ...
                //
                // array of 5 Bee references - no Bee objects here
                //
                Bee[] bees = new Bee[5];
```

```
31.                     ...
32.             }
    }
```

33. Add the following control code to the Run method. Notice how the bees work for a while and then the keeper collects the honey and the cycle begins again.

```
34.   class Keeper
35.   {
36.             public void Run()
37.             {
38.                     ...
39.                     while (true)
40.                     {
41.                             for (int i = 0; i < 20; i++)
42.                             {
43.                                     for (int j = 0; j < bees.Length; j++)
44.                                             bees[j].Work();
45.                             }
46.
47.                             Console.WriteLine("collect {0}", Collect(hive));
48.                     }
49.             }
    }
```

Part 4- Driver

Here we add a simple driver class to start the simulation.

Steps:

1. Create a class Driver with a Main method. Inside Main, create a Keeper and call its Run method.

```
2.              class Driver
3.              {
4.                      static void Main()
5.                      {
6.                              Keeper keeper = new Keeper();
7.
8.                              keeper.Run();
9.                      }
    }
```

06-Properties

Goals:

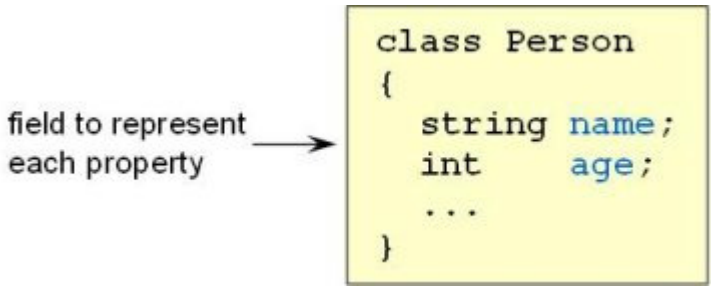
- Show how to define and use properties.
- Discuss static properties.
- Briefly describe the advantages of using the C# property syntax.

Overview

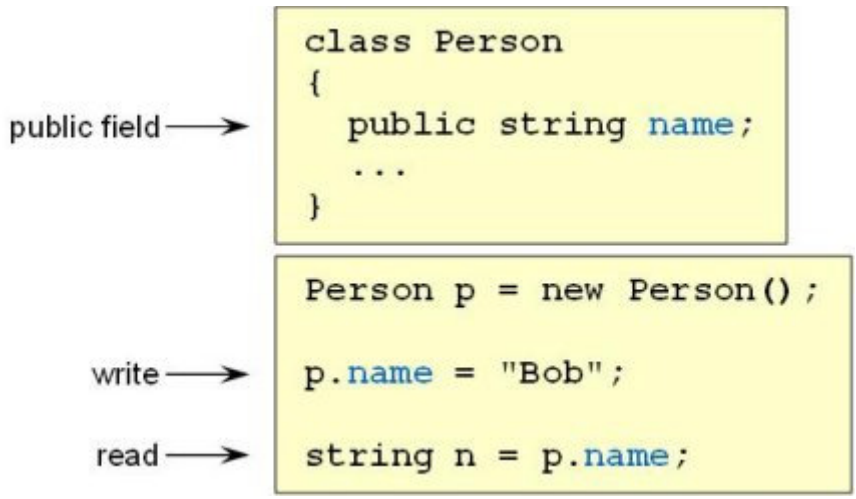
Properties model the traits of an object or a class. C# provides special syntax to define and access properties. Programmers are encouraged to use the special syntax to implement properties because it results in clean client code and makes the intent of the class designer explicit to both human readers and automated tools.

Motivation

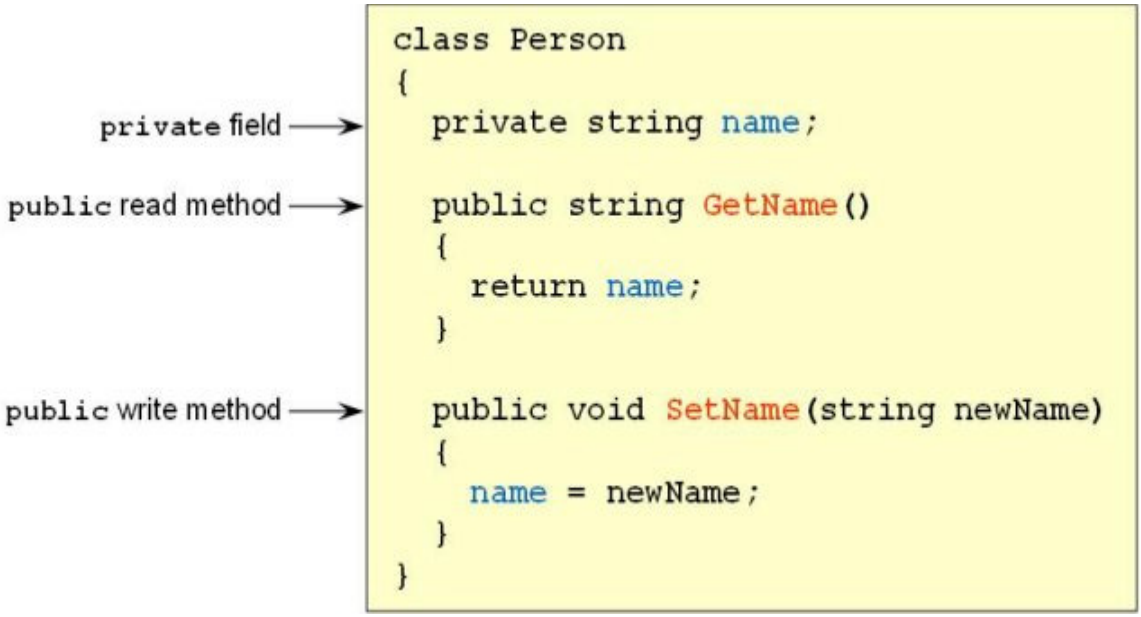
A class is typically a software model of a real world thing. The thing being modeled has certain properties; for example, a person has a name and an age, a stock has a symbol and a price, a circle has a radius, and so on. Other common terms for the same concept are trait, characteristic, or quality. A property can be implemented simply by defining a field to store the current value as shown below.



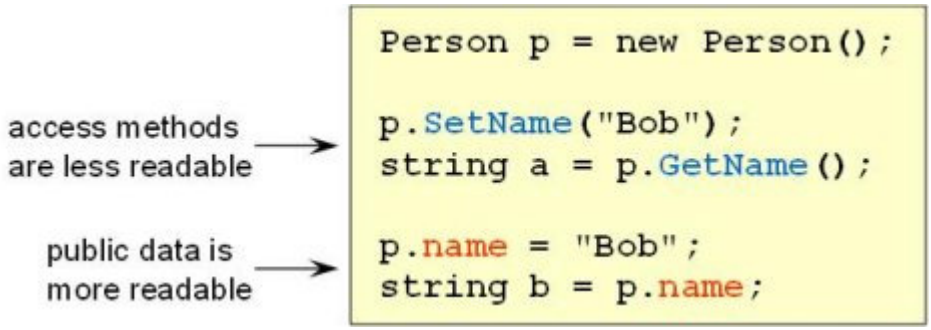
Client code often needs access to the property in order to read and/or write the value. The simplest way to achieve this is to make the fields public. Public data makes the resulting client code clean and simple. A write operation is accomplished by using the assignment operator to load a new value into the field and a read operation is a normal field access.



Despite its simplicity, public data is not used very often in practice since it has some disadvantages from a design point of view. One flaw is that there is no way for the class to perform error checking on the data being loaded into the fields. For example, a client could give a person a negative age or an empty name and the class would not be able to detect the error. Another problem is that there is no way to implement a read-only property since public data is implicitly readable and writeable by all external code. The common way to avoid the problems associated with public data is to make the data private and provide public read and write access methods. By convention, the access methods are typically named `GetXXX` and `SetXXX` where `XXX` is the name of the property in question. The private data / public access method pattern gives the class designer much more control than public data since the client no longer has direct access to the fields and must go through the access methods. This allows the access methods to perform error checking when new values are loaded. Any attempt to load invalid data can be rejected. A simple implementation without any error checking is shown below. It would be easy to extend this solution to include any needed `if` statements to perform any desired validation. It would also be trivial to make the property read-only simply by omitting the `Set` method.



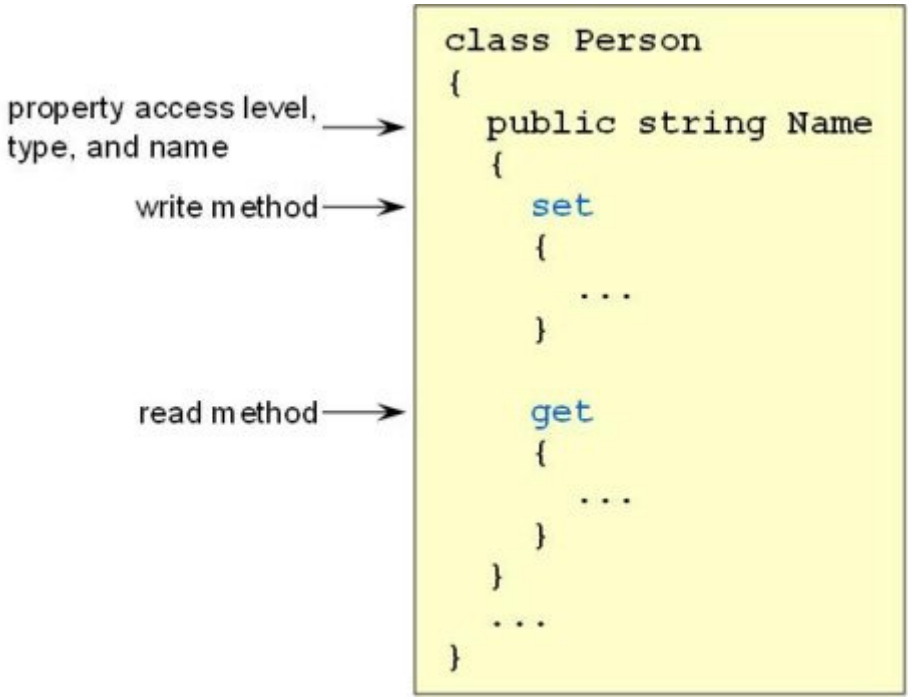
The client code for properties implemented with access methods is not quite as clean as the public data case. The write operation for a public field uses an expression such as `object.property = value;` where the left hand side of the assignment operator is the property being modified and the right hand side is the new value. This syntax is intuitive since human readers generally find it easy to identify a line of code containing an assignment operator as a write operation. However, with an access method, the write operation is performed by calling the set method and passing the new value in as an argument. This syntax is arguably more difficult to visually identify as a write operation since the assignment operator is not used. A similar case can be made for a read operation. The read operation for a public field is typically a noun phrase such as `object.property`. This is intuitive since it agrees with the idea that a property should be a noun because it represents some trait of the type being modeled. In contrast, the read operation using a typical access method is a verb phrase such as `object.GetXXX`. This conflict can lead to client code that is less intuitive than when public data is used.



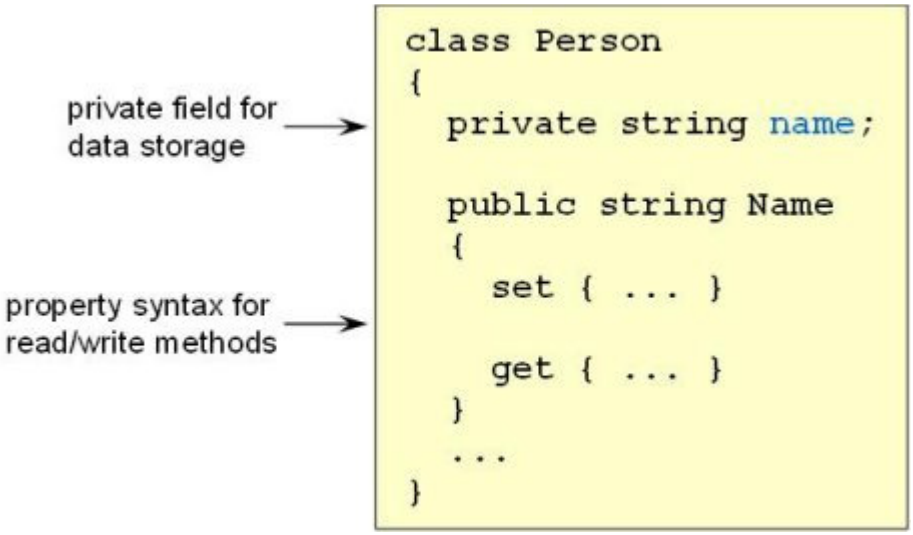
C# properties

The two goals of access control and clean client code are somewhat in conflict. Public data yields simple and intuitive client code but does not give the class designer enough control. Private data / public access methods provide the ability to do error checking and implement read-only properties but leads to less readable client code. To solve this dilemma, C# offers special syntax to implement properties that combines the advantages of both conventional techniques.

C# provides special language support for the implementation of properties which specifies a structured way to define the read and write accessors. There are five components of the property declaration: access level, type, name, write accessor, and read accessor. The basic structure is shown in the figure below.

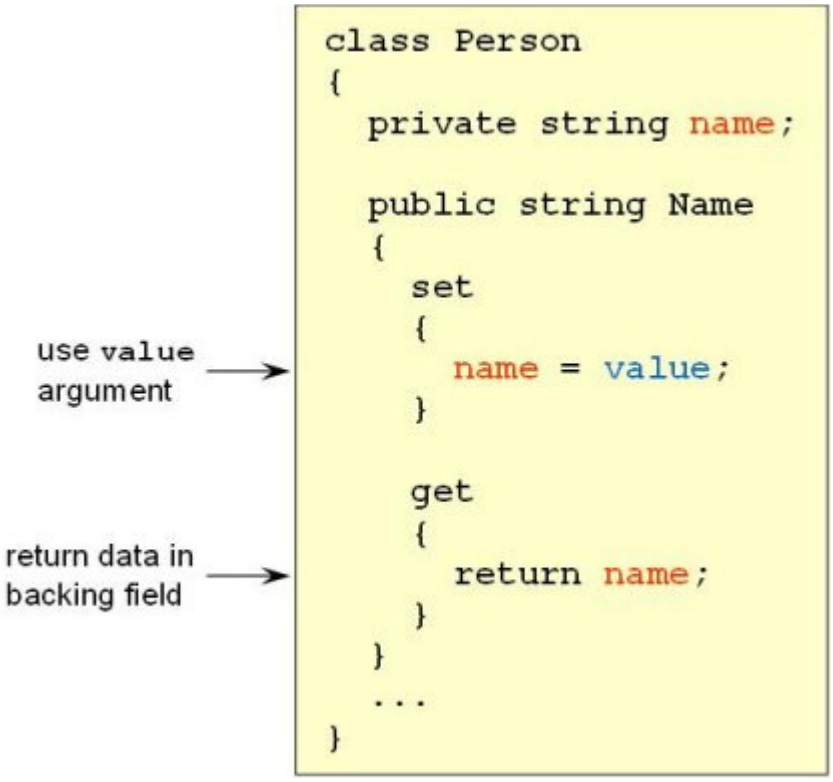


Recall that a property definition typically has three components: an instance variable for data storage, a write access method, and a read access method. A C# property definition provides only the accessors and does not include any needed data storage. The most common case then, is to define a field to store the property data. The common wisdom suggests choosing a name for the field that is similar to the name of the property; for example, a variable called `name` for a property called `Name` (the difference in case is sufficient to make them distinct in C#).

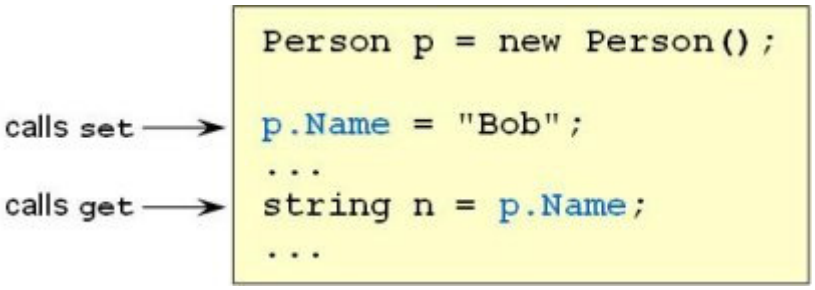


The implementation of the accessors is perhaps the most interesting part of the syntax. The write accessor for a property is named `set`. The definition syntax is reminiscent of a traditional write access method but makes a few assumptions that reduce the amount of code required. Basically, the entire first line of a traditional write method is missing; that is, the return type, method name, and parameter list are not specified. This simplification makes sense because traditional write access methods all have the same form: a single argument whose type matches the type of the property and `void` return type. The main consequence of this minimal syntax is that the programmer does not have control over the name of the parameter; instead, the C# design team chose the name `value` for the implicit parameter. Other than the lack of the method header, the set accessor is much like a method. The body can contain an arbitrary set of statements to be run whenever the accessor is invoked.

The read access method for a property is named `get`. A traditional read access method takes no arguments and has a return type that matches the type of the property. Again, since this is so common, the property syntax does not contain these details explicitly. The body of the accessor specifies the statements to be run when the get accessor is used. The get accessor has an implied return type that matches the type of the property so the body is required to return an appropriate value or it is a compile time error. A complete property implementation is shown below.



The client code for C# properties is identical to accessing a public field. The compiler analyzes how the property is being used and invokes the appropriate accessor automatically. Specifically, the `set` accessor is invoked when the property is being used on the left hand side of the assignment operator while the `get` accessor is used in all other cases. Some sample client code is shown below.



The C# property syntax gives much of the power of the private data / public access methods pattern while retaining all the benefits of a public data implementation. In particular, a property can be read-only, write-only, or read-write. This is accomplished simply by providing or omitting the `set` or `get` accessors as desired. C# properties also give the class designer the ability to add any needed error checking code to the `set` and `get` implementations.

Static property

A property that applies to an entire class rather than to an instance can be created by applying the keyword static to the property definition as shown below. As with a static method, inside the accessors of a static property the instance methods and instance fields of the class are not accessible. That is, only the static parts of the class are usable and any attempt to access non-static members is a compile time error.

define static property →

```
class Item
{
  private static int revenue;

  public static int Revenue
  {
    get
    {
      return revenue;
    }

    set
    {
      revenue = value;
    }
  }
  ...
}
```

A static property must be accessed through the type name. It is a compile time error to attempt access using an instance of the class.

access using type name →

```
Item.Revenue = 5;
```

Benefits

The decision to support properties as a first class language feature is an interesting one. Adding the feature made the language larger and thus more difficult to learn; however,

there are several good arguments as to why the special property syntax is worthwhile. First, the client code using the property syntax is much cleaner than using traditional access methods. Second, it makes the code more self documenting since otherwise, maintenance programmers would need to deduce the existence of a property from the presence of a pair of set/get methods. Finally, the property syntax allows property information to be carried through compilation and into the resulting IL code. The IL can be interrogated programmatically using the `System.Reflection` API and the properties of a class determined. Visual Studio .NET makes heavy use of this feature to display Intellisense information and to populate properties windows for GUI components in the design tools.

06- Properties Exercise

Goals:

- Code read-only and read-write properties for a class.
- Use both instance and static properties.

Overview

Properties provide an elegant solution to a common design dilemma: public vs. private data. Public data provides clean access syntax but can break encapsulation. Private data and public access methods provide maximum encapsulation but can require awkward syntax in use. Properties give the best of both solutions.

Part 1- Set-up

This part sets up the example we will use to experiment with properties.

Steps:

1. Create a class to represent a savings account. Add private fields for the account number and the account balance. Code a constructor to initialize the account number and the balance.

```
class SavingsAccount
{
        private int    accountNumber;
        private double balance;

        public SavingsAccount(int accountNumber, double balance)
        {
                ...
        }
        ...
}
```

12. Create a class called SavingsAccountTest and add a Main method. Create a Account and test your work.

```
class SavingsAccountTest
{
```

```
15.             static void Main()
16.             {
17.                     SavingsAccount account = new SavingsAccount(12345,
    2000);
18.                     ...
19.             }
20.             ...
    }
```

Part 2- Read-only instance properties

Here we add read-only instance properties.

Steps:

1. Add read-only instance properties AccountNumber and Balance to the savings account class. The implementation should simply return the current value of the accountNumber and balance fields. Test your work.

```
2.    class SavingsAccount
3.    {
4.              public int    AccountNumber { ... }
5.              public double Balance       { ... }
6.              ...
    }
```

Part 3- Calculated property

Here we add a property whose value is calculated as needed, i.e. it is not backed by a field.

Steps:

1. Add the read-only instance property IsOverdrawn to the savings account class. The property should be true if the balance of the account is negative and false otherwise. Test your work.

```
2.    class SavingsAccount
3.    {
4.              public bool IsOverdrawn { ... }
5.              ...
    }
```

Part 4- Read-Write instance property

Here we add a property that supports both read and write access.

Steps:

1. Add the read-write instance property InterestEarned to the savings account class. The property should be backed by a field that stores the amount of interest the account has earned. Include validation code in the set accessor that ensures the value being assigned is positive. If an attempt is made to assign invalid data, print an error message and return without performing the assignment. Test your work.

```
2.    class SavingsAccount
3.    {
4.              private double interestEarned; // field
5.
6.              public double InterestEarned { ... } // property
7.
8.              ...
      }
```

Part 5- Read-Write static property

Here we add a static property that supports both read and write access.

Steps:

1. Add the read-write static property InterestRate to the savings account class. The property is static because it applies to all accounts, i.e. all accounts earn the same rate of interest. The property should be backed by a static field that stores the value. Include validation code in the set accessor that ensures the value being assigned is positive. If an attempt is made to assign invalid data, print an error message and return without performing the assignment. Test your work.

```
2.    class SavingsAccount
3.    {
4.              private static double interestRate; // field
```

```
5.
6.              public static double InterestRate { ... } // property
7.
8.              ...
    }
```

07-Indexers

Goals:

- Describe the concept of indexers.
- Show the definition and invocation syntax for indexers.

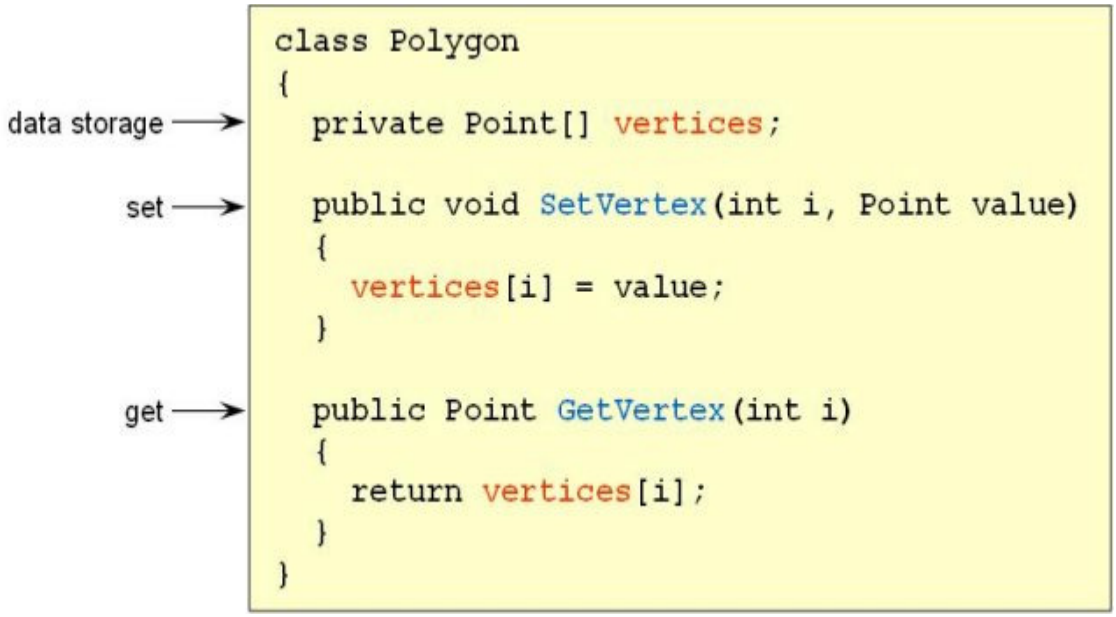
Overview

An indexer provides a way for a class to support array-like indexing using the square brackets operator `[ ]`. This works well for any class that represents a collection of data keyed by index.

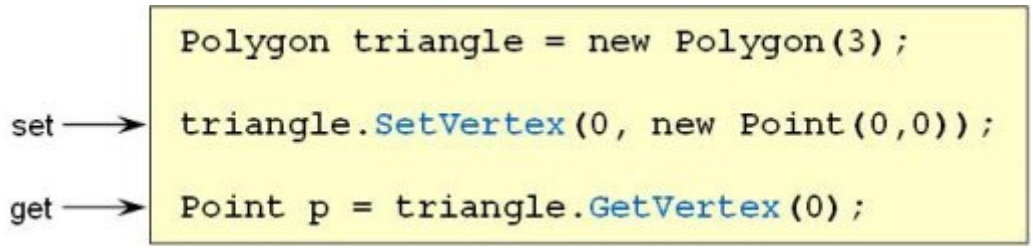
Motivation

Some classes can be thought of as a collection of other objects. A department at a company might contain all of the employees that work in that department. An investment portfolio would contain the current set of stocks owned. A polygon could store all the points that make up the shape. In each case, it might make sense to allow client code to access elements of the collection by giving a unique index. For example, an employee record might be requested from the department by specifying the employee's id number or name. A particular stock might be extracted from an investment portfolio by giving its ticker symbol. The points of a polygon might be indexed by an integer representing the order in which the points are connected.

Each collection class would need some private data structure behind the scenes in which to store the collection of data. The most common case would be a simple array but more sophisticated implementations are possible as well. To allow client code to access the elements of the collection, public access methods would likely be provided. The set access method would require two parameters: the index and the data. The get access method would take the index as an argument and return the corresponding value. An implementation for a simple polygon class is shown below.



Clients of the polygon class use the get and set methods for access to the collection of points.

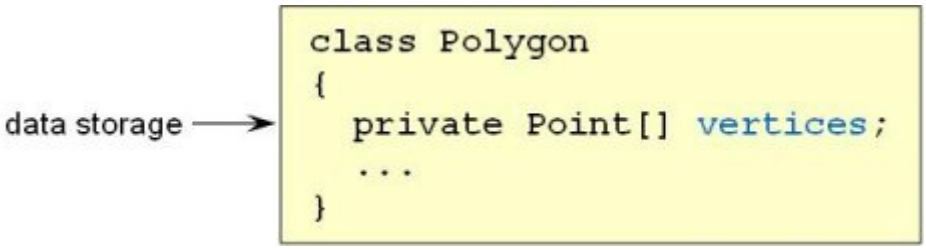


The private data / public access methods pattern described above works reasonably well. The clients interact with the access methods while the data store is hidden behind the scenes. As an alternative, the class designer can supply an indexer instead of traditional get and set methods. The special indexer syntax allows the square bracket operator to be used for access instead of regular get and set methods calls. The primary advantage is that the indexer client code is more concise than analogous code using traditional access methods.

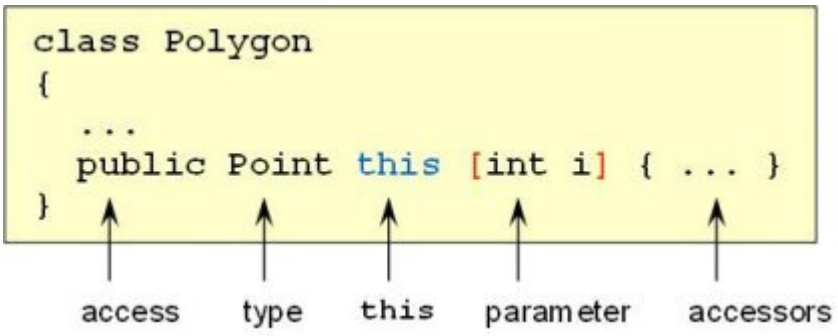
Basic indexer

Recall that there are typically three components to the implementation of any data access pattern: the get method, the set method, and the field used to store the information. Indexers provide a fancy way of writing the get and set access methods but do not give any help with data storage. Therefore, lurking behind the scenes of any indexer implementation will be some sort of data store, typically just a private field. The polygon

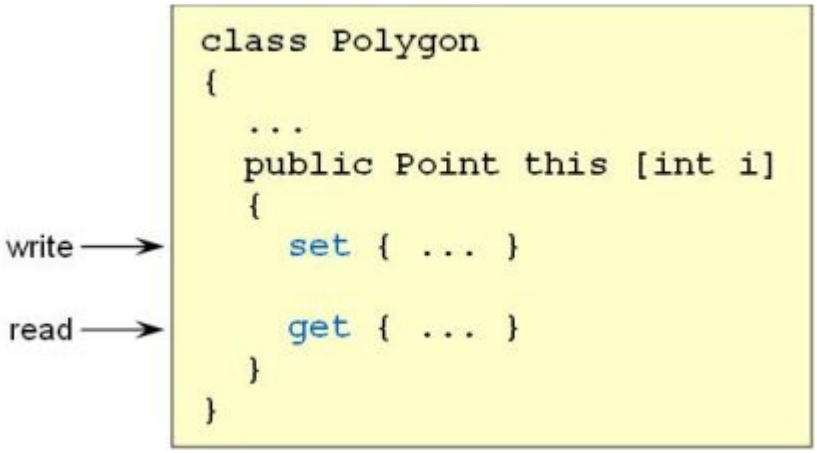
example would likely use an array of points to store the data as shown below.



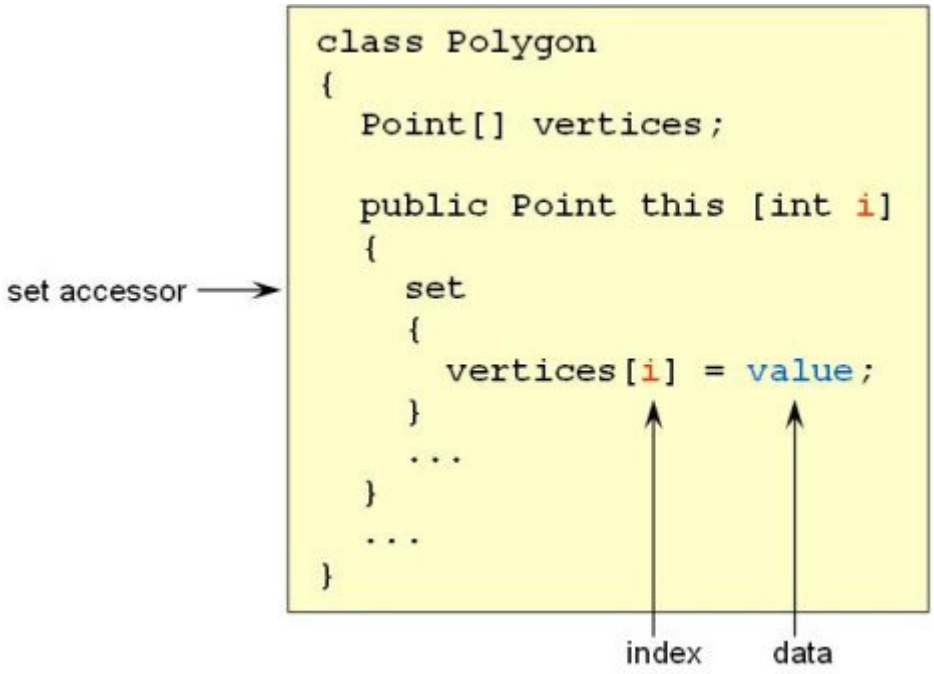
Indexers provide a special way to write get and set access methods that are invoked using the square brackets operator rather than traditional method call syntax. The indexer definition syntax is a bit obscure and fairly lengthy. First, the access level such as `public` or `private` is specified. Next comes the type of data being indexed; for example, our polygon sample would specify `Point` while the investment portfolio example would use `Stock`. After the type comes the keyword `this`; there is no option here, the keyword must be included in the definition of every indexer. Next, placed inside square brackets, comes the type and name of the index. Lastly comes the body of the indexer with the get and set access methods defined inside. The syntax is summarized in the figure below.



The get and set accessors are defined inside the body of the indexer. The definition of the accessors is simplified by the fact that much of the interface information is already captured in the indexer declaration. In particular, the indexer declaration contains the type of data being indexed and the type and name of the index itself.

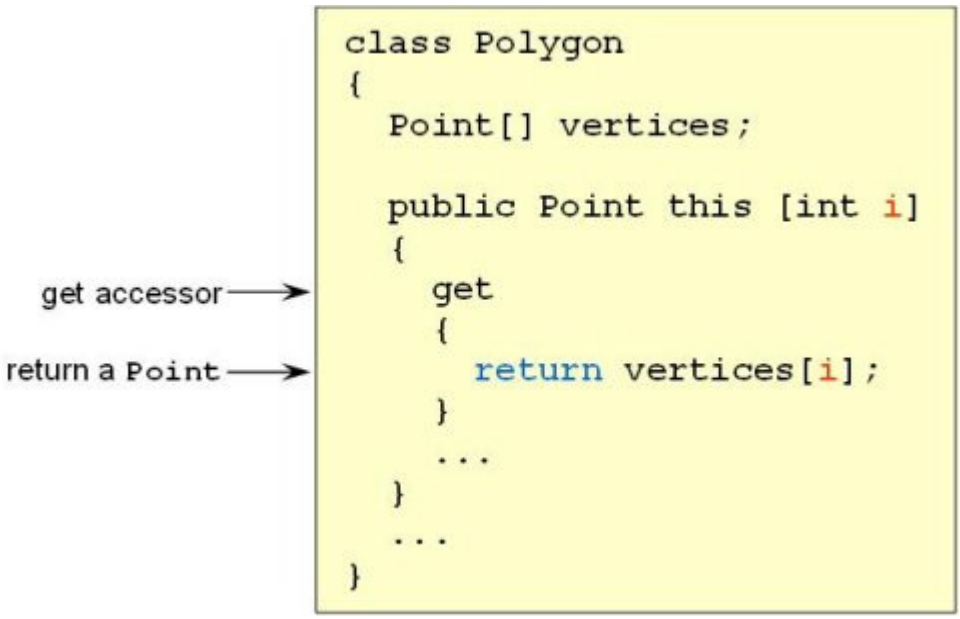


Recall that a traditional set method for indexed data would take two arguments: the index and the data. The implementation of a set accessor does not specify these parameters explicitly; in fact, the method header is completely omitted and much of the parameter information is taken from the indexer declaration. The name of the index is determined by the name in the indexer declaration. The name of the data parameter is the special symbol `value`. The term `value` is imposed on the programmer by the C# language designers and cannot be changed. The example below shows the definition of the set accessor for the polygon example.

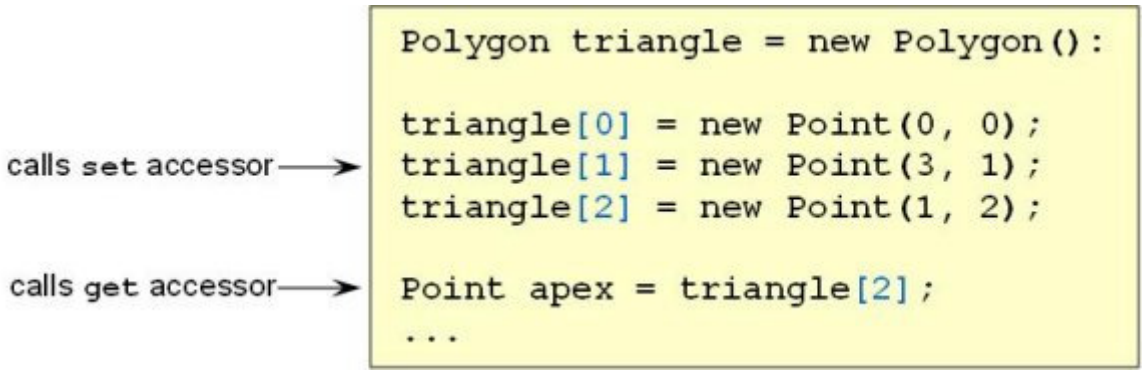


A traditional get method would take only the index as an argument and return the corresponding data. The implementation of the get accessor dispenses with this detail and specifies only the method body preceded by the keyword `get`. The get accessor must

return an object of the type being indexed as expected for any get access method.

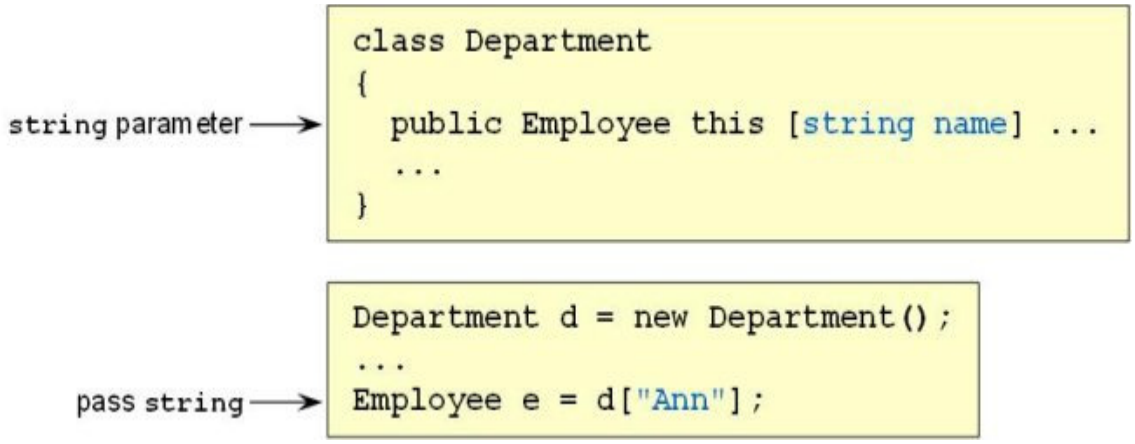


The client code that makes use of an indexer is clean and simple. The client applies the square brackets operator to an object of the class and the compiler invokes the appropriate accessor automatically. When the indexer is used on the left hand side of the assignment operator the `set` accessor is used. In all other cases the `get` accessor is invoked.



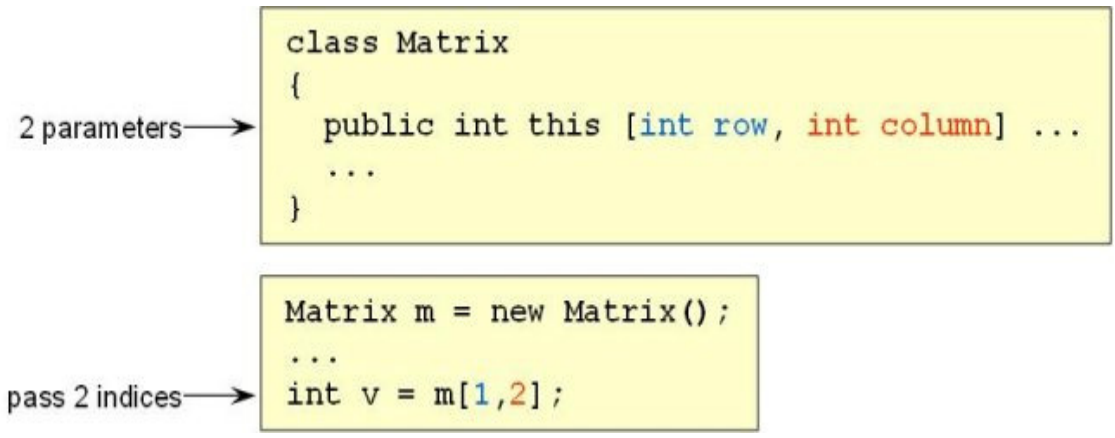
Index type

Most indexers have a simple integer index; however, the index can actually be of any type. The code below illustrates this by showing a department class that uses an indexer to allow employees to be accessed by name. The type of the index in this case is a string.



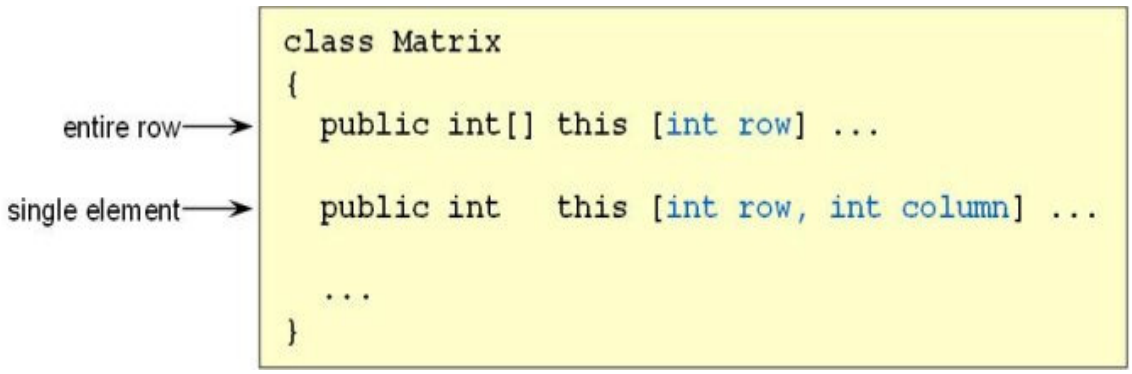
Index number

Indexers support multiple parameters. The indices are placed inside the square brackets and separated by commas. The code below shows a matrix class with an indexer that requires two parameters: the row and column.



Indexer overloading

A class may offer multiple indexers as long as each version has a unique set of indices. The example below shows a matrix class that offers two indexers. The first indexer allows the user to pass a single index, the row number, to access an entire row of data. The second indexer requires both a row and a column index and allows access to a single element of the matrix


entire row
single element
class Matrix
{
public int[] this [int row] ...
public int this [int row, int column] ...
...
}

07- Indexers Exercise

Goals:

- Use an indexer provided by a library class.
- Implement several indexers of varying complexity.

Overview

Indexers provide a convenient syntax to get or set the elements of a collection. The most common indexers provide access by integer index; for example, the elements of an array, the characters in a string, or the cells of a matrix. More sophisticated indexers allow access based on other types. For example, a string index could be used by an employee directory to lookup an employee by last name.

Part 1- Using an indexer - string

Indexers are relatively common in the .NET Framework library. The most visible example is probably string which offers read-only access to the characters it is storing.

Steps:

1. Write a simple program that creates a string and uses the indexer to read individual characters. The first character is at index 0, the second at index 1, etc. String provides a read-only Length property which can be used to determine the maximum allowable index. Attempt to change a character in the string using the indexer. The compiler should issue an error saying that the indexer is read-only.

Part 2- Simple indexer - Student grades

In most schools each letter grade has a numeric equivalent, an 'A' is worth 4.0, a 'B' worth 3.0, and so on. Adding up these values and

dividing by the number of courses produces the grade point average or gpa. The gpa is thus a single number that gives a measure of academic performance.

Suppose a student attending a four year university would like to track their scholastic performance each year. The student would simply need to keep four gpa values, one for each year. In this exercise we will code a student class that stores the four gpa numbers and provides a read-write indexer to access the values.

Steps:

1. Create a Student class. Add a private array of double to store the four gpas.
2. Add a read-write indexer for the gpa values. Use the year as the index: 1 for the first year, 2 for the second year, and so on. Be careful to map the year to the array index: the year will be 1-4 while the array storing the gpa values is zero based. Simply subtract 1 from the year to get an appropriate index. The trickiest part to the implementation is how to react to an invalid index. The real solution would probably be to throw an exception, but since we have not yet covered exceptions we have to be content with a less sophisticated approach: in the get accessor, print an error message and return -1, in the set accessor, print and error message and return without performing the requested assignment.

```
3.    class Student
4.    {
5.              public double this[int year]
6.              {
7.                        get
8.                        {
9.                                  ...
10.                       }
11.
12.                       set
13.                       {
14.                                 ...
15.                       }
16.             }
      }
```

17. Create a Main program and test your work.

Part 3- Sophisticated indexer - photo album (optional)

An album is a group of photos. Since an album can be viewed as a collection it is a reasonable candidate for an indexer to provide access to its underlying parts. In this case we will provide two different indexers for the photos contained in the album: a read-write indexer to access photos by integer index and a read-only indexer to access photos by title.

The trickiest part of this implementation is how to react to failure, i.e. when the user provides an invalid index or a title that is not found among the photos. The set accessor is easy to deal with since we can simply refuse to perform the requested assignment. The get accessor should just return null if the requested photo is not found. In production code we might want to use an exception to notify the user if either access method fails.

Steps:

1. Create a class Photo to represent a photograph. A realistic photograph class might store a title, description, creation date, link to a disk file containing the image, etc. For our purposes we will keep things simple and store only a title. When implementing the Photo class use standard practices: store the data in a private field, provide a public property to access the data, and a constructor to do initialization.

```
class Photo
{
        ...
}
```

5. Create a class Album to hold the photographs. Store the photographs in an array of Photo. Write a constructor that takes an integer capacity and allocates an array of the requested size.

```
class Album
{
        Photo[] photos;

        public Album(int capacity)
        {
                ...
        }
        ...
}
```

15. Add a read-write indexer to the Album class that gives access to the underlying array of photos. The return type of the indexer will be Photo and the parameter an int. Perform validation on the index to ensure it is within range.

```
class Album
{
        public Photo this[int index]
        {
                get
                {
                        ...
                }

                set
                {
                        ...
                }
        }
        ...
}
```

31. Add a read-only indexer to the Album class that gives access to the underlying array of photos by title. The return type of the indexer will be Photo and the parameter a string. Use the == operator to compare the title the user is looking for with the title of each photo in the array.

```
class Album
{
        public Photo this[string title]
        {
                get
                {
                        ...
                }
        }
        ...
}
```

42. Create a Main program and test your work.

08-Inheritance

Goals:

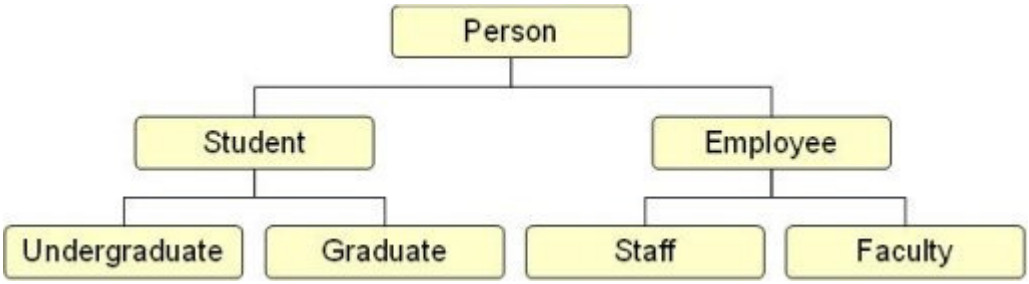
- Introduce the syntax and semantics of inheritance.
- Discuss the `protected` access level.
- Show how to call base class methods and constructors.

Overview

Inheritance allows a 'derived' class to be created from an existing 'base' class. The new class inherits the fields and methods of the base class. In addition, the new class can add new methods, add new fields, and override existing methods.

Motivation

Inheritance provides a way to model the real world concept of generalization/specialization. Consider the different types of people at a university described in the following diagram.



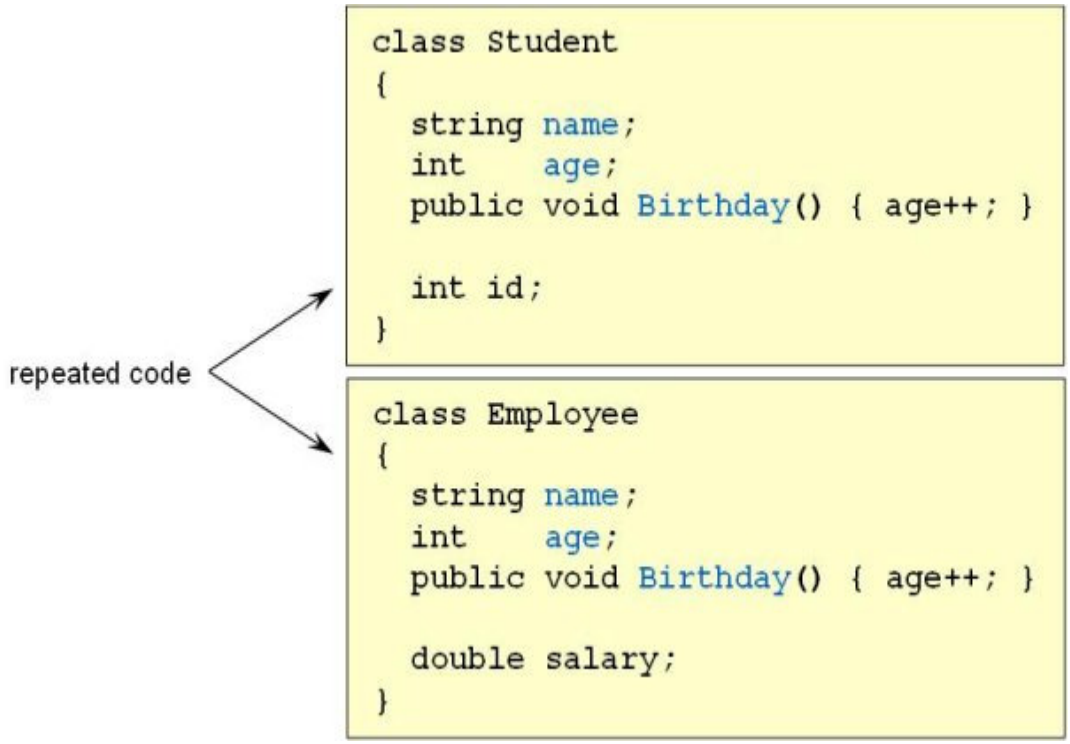
The types at the top of the hierarchy are very general since everyone at the university is a person. Proceeding down the hierarchy things become more specialized because some of the people at the university are students and some are employees. Students and employees have some behavior in common but differ in other ways. The parts they share are captured in the base person class while their differences appear in the derived student and employee classes. Similar comments apply lower in the hierarchy since some of the students at the university are undergraduates and some are graduates. The common term for this type of generalization/specialization is the "is-a" relationship. One might say "a student is a person" or "a faculty member is a employee".

The example above intentionally uses types of people in order to demonstrate that the

concept of the is-a relationship is not exclusive to programming. Most humans naturally create mental categories such as person, fruit, employee, shape, mammal, etc. to help them deal with the complexity of the real world. The categories capture the elements that all members of the group share. For example, all employees might have an id number and make a certain salary, all shapes might have a color and take up a certain area when drawn on a piece of paper, and so on. This organization makes the world a much simpler place because an understanding of a category gives a lot of information about the behavior of each member. In other words, if you understand a basic concept such as apple, it will help you when you encounter a new member of the group such as Fuji or Braeburn for the first time.

Inheritance gives developers a way to express the is-a relationship in their code. This is especially important in object-oriented programming where the goal is to model the real world accurately. If in the real world, the people at a university are categorized as shown in the diagram above, then a software model of the university should be sure to capture those relationships. The common wisdom then, says that whenever the is-a relationship exists in the application domain, it should be modeled in software using inheritance.

At this point, the discussion of the meaning and benefits of inheritance has been mostly theoretical. It might help to look at a more concrete advantage as well: elimination of repeated code. Consider the two classes below and note how several lines of identical code appears in both.

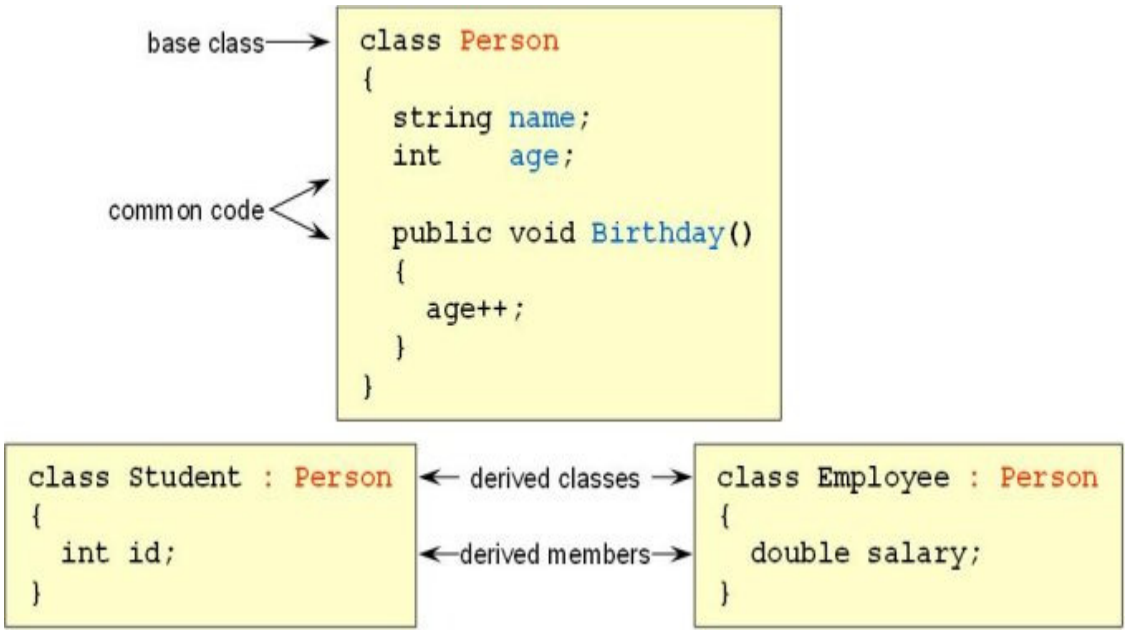


The disadvantages of repeated code are well known: extra work to program and extra work to debug and maintain. In the following discussion, we will see how to rewrite the

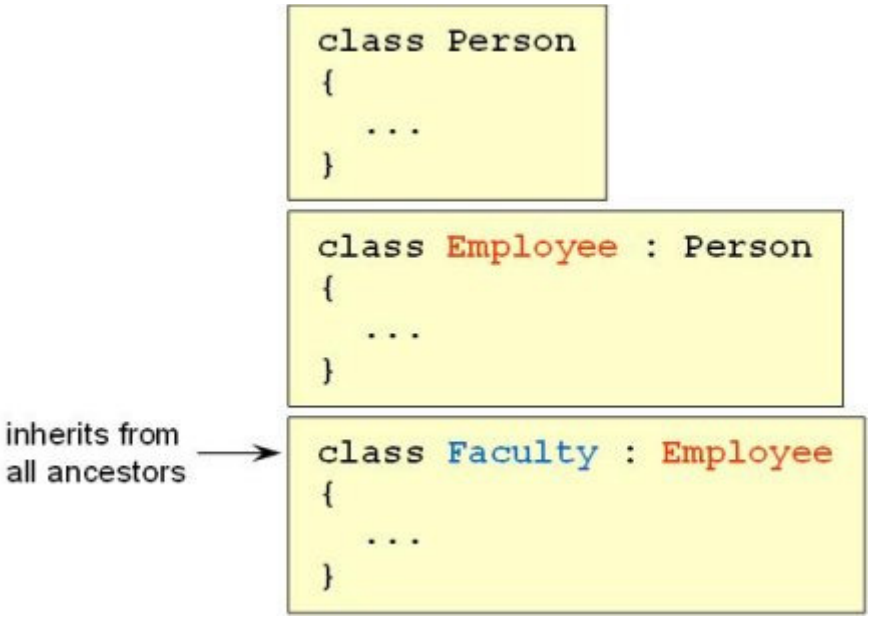
above example so the repeated code is factored out into a base class called `Person`. The `Student` and `Employee` classes will be derived from `Person` to obtain the functionality they need without having to repeat the implementation of the shared code.

Basic syntax

A class specifies its base class by placing a colon and the base class name as part of its declaration. The example code below shows a base class `Person` and two derived classed `Student` and `Employee`. Note how the fields and methods that apply to both students and employees appear in the base class while the specific members are placed in the derived classes.

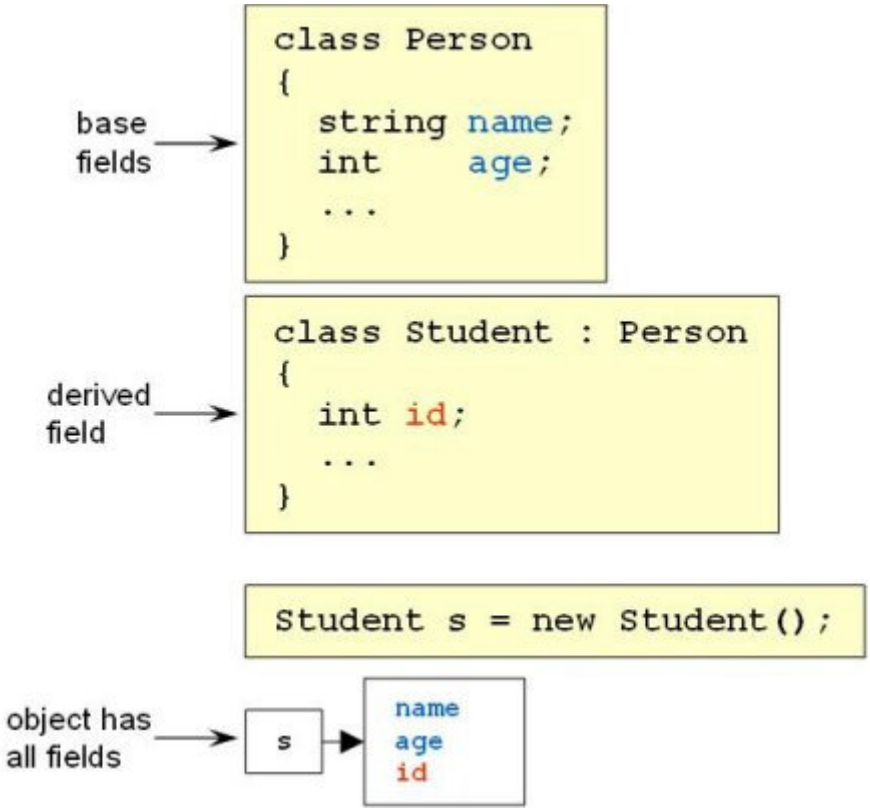


An inheritance hierarchy can be as deep as required. For example, a `Faculty` class could be derived from `Employee` which, in turn, is derived from `Person`. The `Faculty` class inherits the members of all ancestor classes; that is, it inherits from `Employee` and `Person`.



Memory allocation

Creating a derived class object allocates memory for the fields declared in both the base class and the derived class.



Base services

Public members of the base class are available to clients of the derived class. That is, client code that is using an instance of the derived class can access any public fields or call any public methods that are declared in the base class.

base method →

```
class Person
{
  public void Birthday()
  {
    age++;
  }
  ...
}
```

derived object →

call base class method →

```
Student s = new Student();

s.Birthday();

...
```

Single inheritance

C# offers only single inheritance which means that a class can have only one direct base class. This restriction is actually imposed by the CLR so single inheritance is the rule for all .NET languages. The designers of .NET acknowledge that multiple inheritance is more powerful than single inheritance; however, they decided that the extra complications and ambiguous situations that arise in the presence of multiple base classes was not worth this extra power.

```
class Student : Person
{
  ...
}
```

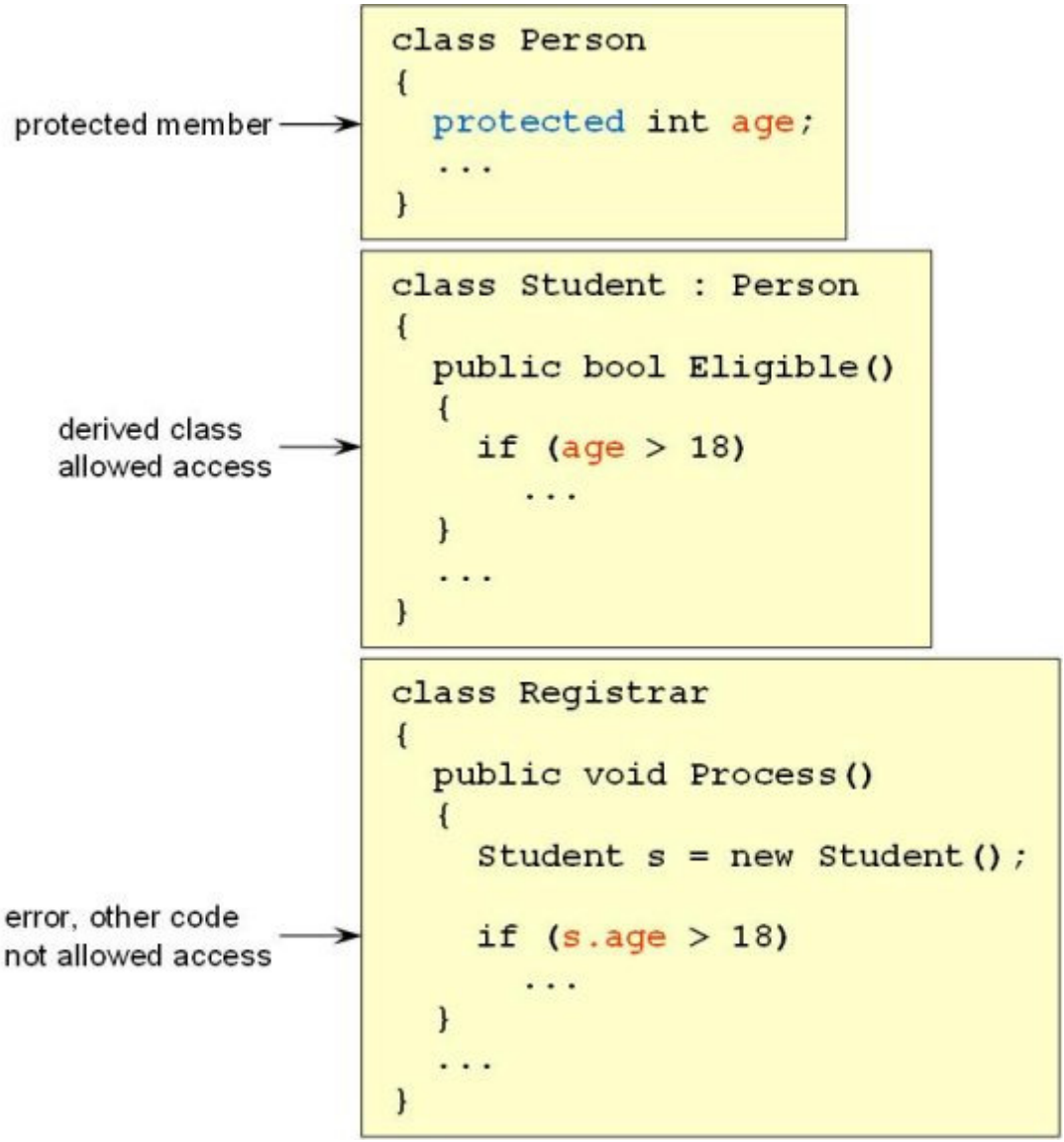
```
class Employee : Person
{
  ...
}
```

error, only one base allowed →

```
class Graduate : Student, Employee
{
  ...
}
```

Protected access

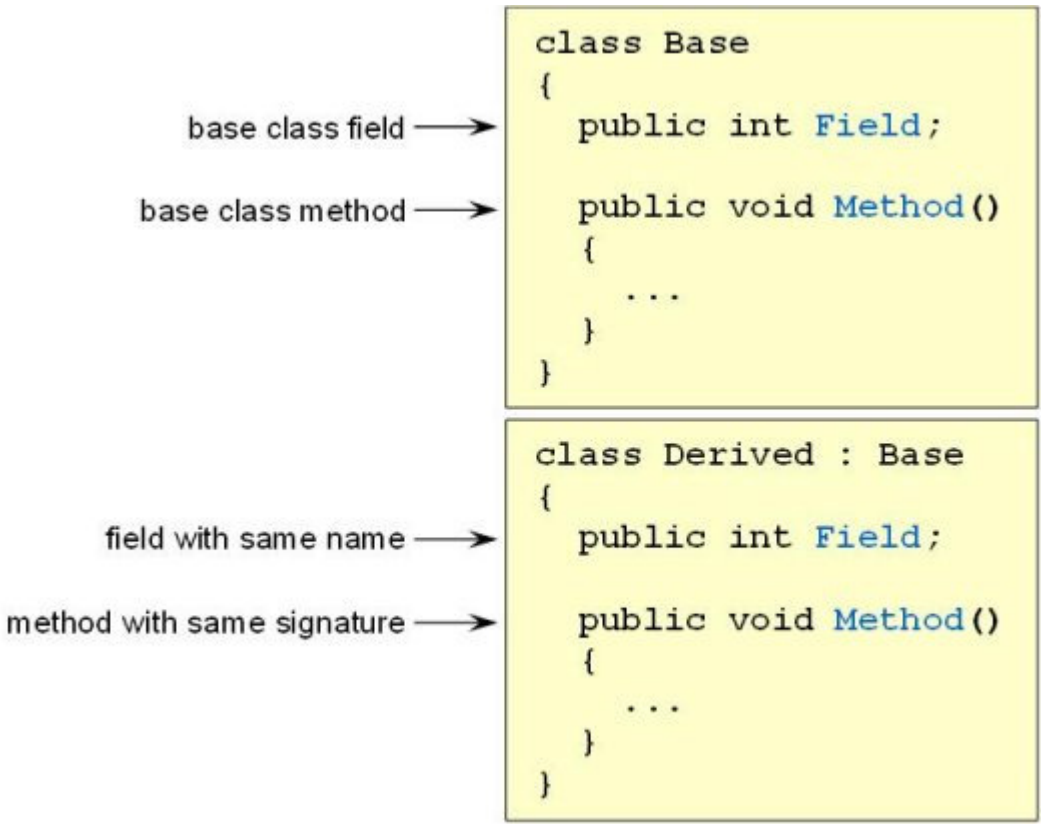
There is a third access level for class members in addition to the more common `public` and `private`. The `protected` access level grants access to derived classes but prevents access from all other client code.



Hiding inherited member

It is possible that a base class and a derived class might contain an identical member. The

example below shows the issue.



This preceding code has two ambiguities. The base class and derived class have an accessible field with the same name and they contain a method with the same signature (name and parameter list). To see how such situations might arise in practice, let's revisit the `Person` and `Student` classes and consider how the development process might have taken place over time.

Suppose the person class was created first. In most countries, each person is given an id number by the government so it would make sense for person to have an id field and one or more access methods as shown below.

government assigns id to each person →
set method for person id →

```
class Person
{
  int id;

  public void SetId(int id)
  {
    this.id = id;
  }
  ...
}
```

Once the basic person class is complete, another team member might begin writing a derived class such as `Student`.

`Student` derived from `Person` →

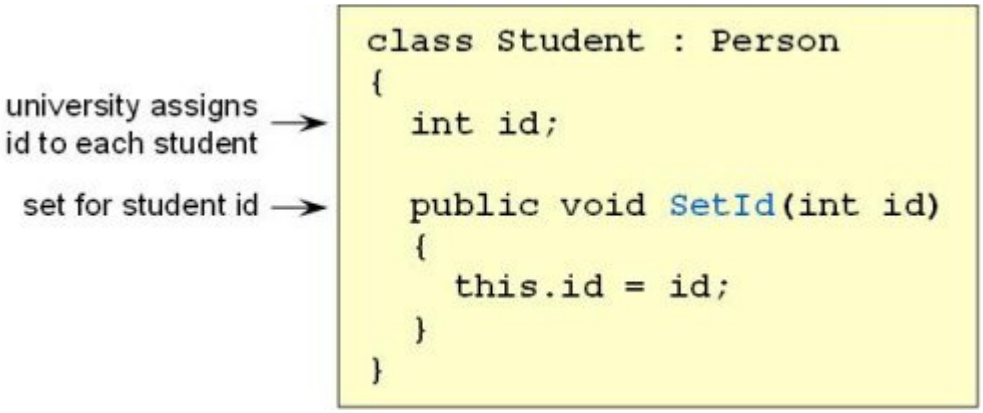
```
class Student : Person
{
}
```

At this point in the development process there are no ambiguities. Because everything is working correctly, the designers of `Person` and `Student` decide to release their classes to the rest of the development team. The other team members are happy because they have been waiting for these classes in order to move their own parts of the project forward. Client code such as the following now begins to be written.

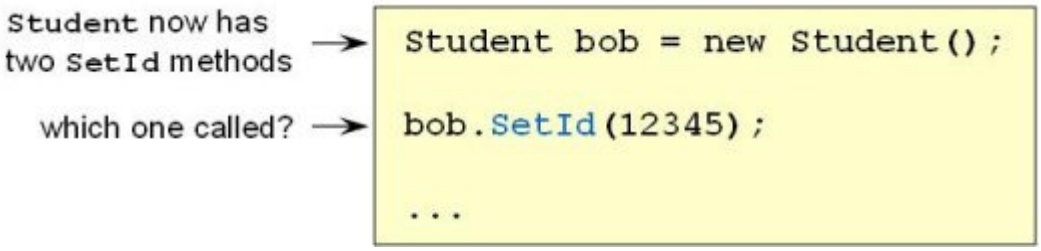
`Student` object →
invokes `Person` class `SetId` method →

```
Student bob = new Student();

bob.SetId(12345);

...
```

Now suppose the `Student` class designer realizes that most universities assign each student a unique id number. They return to the class and add an id field and one or more access methods as shown below.



The addition of the `SetId` method to the `Student` class has resulted in an ambiguity since `Student` now has two methods with the same signature. Consider the effect on the client code: which method should be called now, the `Person` version or the `Student` version?



It turns out that the derived class method takes precedence over the base class method. Therefore, the behavior of the client code will change, where previously the `Person` version of `SetId` was invoked now the `Student` version will be called.

The C# design team considered this type of behavior change too important to ignore; therefore, ambiguities between base and derived classes generate a warning from the compiler. When the designer of the `Student` class rebuilds the code, they will see a warning and be aware that the addition of the `SetId` method will change the behavior of client code. At this point, the `Student` class designer has a couple of choices. Arguably, the best decision would be to change the name of the `Student` version of the `SetId` method to something that doesn't conflict with the inherited method. If this is not an option, they must add the keyword `new` to the definition of the student `SetId` method to acknowledge that the new method will hide the inherited one. Adding `new` to the method definition will suppress the compiler warning. Note that the behavior of the client code did still change, and the `Student` class designer should probably make an effort to notify all the clients of the change. The point is that the change did not occur silently.

label with new to acknowledge hiding of inherited version →

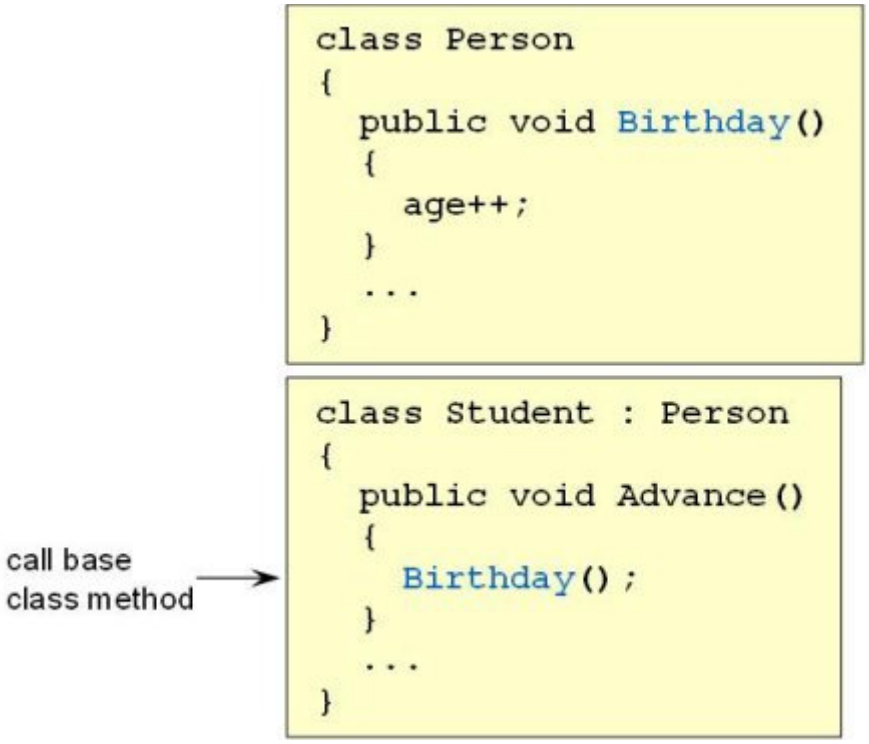
```
class Student : Person
{
  int id;

  public new void SetId(int id)
  {
    this.id = id;
  }
}
```

The keyword `new` may also need to be applied to fields whenever a derived class field conflicts with a field inherited from its base class. In the `Person` and `Student` example, `new` is not needed on the `id` field of `Student` even though `Person` defines a field with the same name. This is because the person `id` field is private and so is not accessible to `Student` and therefore there is no ambiguity.

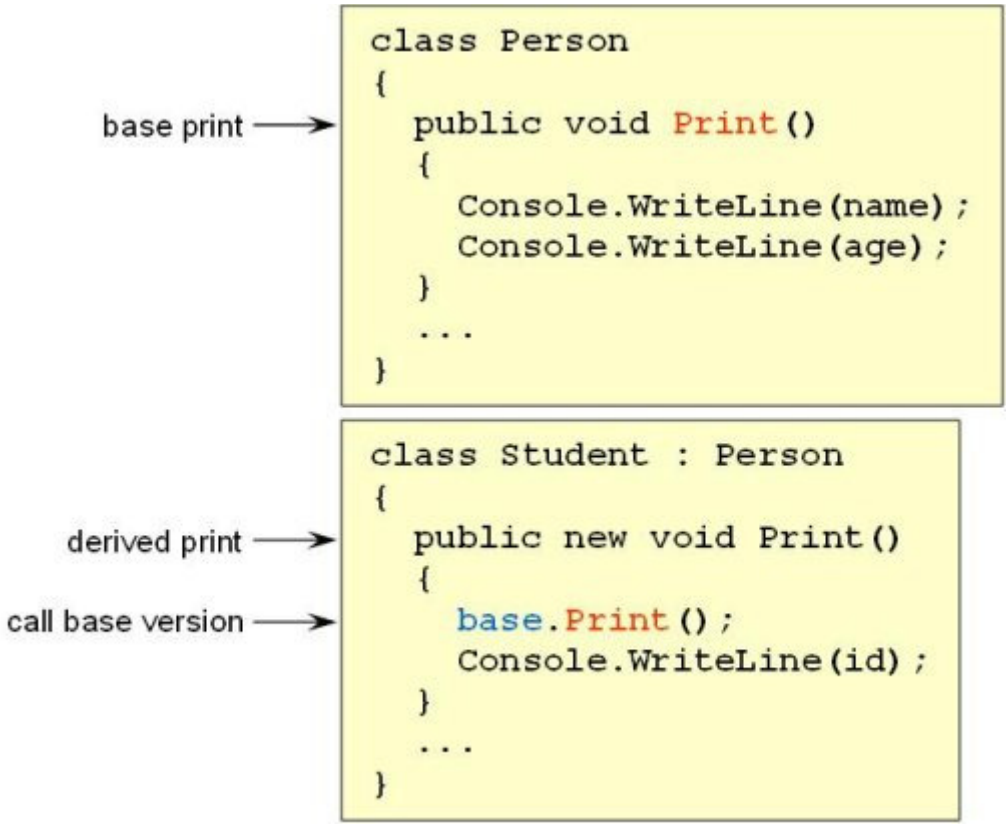
In practice, inheritance hierarchies created by a single team will likely be carefully designed to minimize these types of ambiguities so the use of `new` should be rare.

Invoking base class methods

A derived class often needs to invoke a method defined in its base class. Generally, this is extremely easy and the inherited method can be called just as if it were a method of the derived class. The code below shows this simple case.

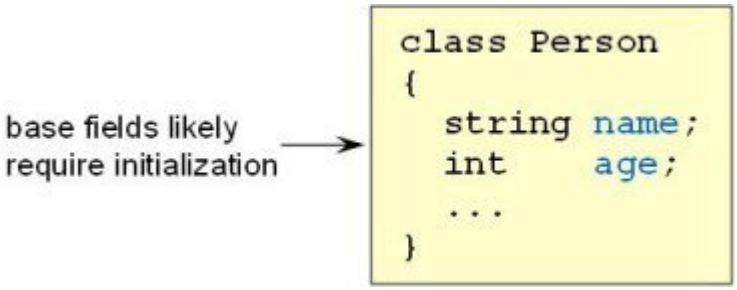


If the base and derived class both have a method with the same signature, the derived class must use the keyword `base` to get access to the base class version. If the keyword `base` were omitted from the call, the derived class version would be invoked instead of the base class version.

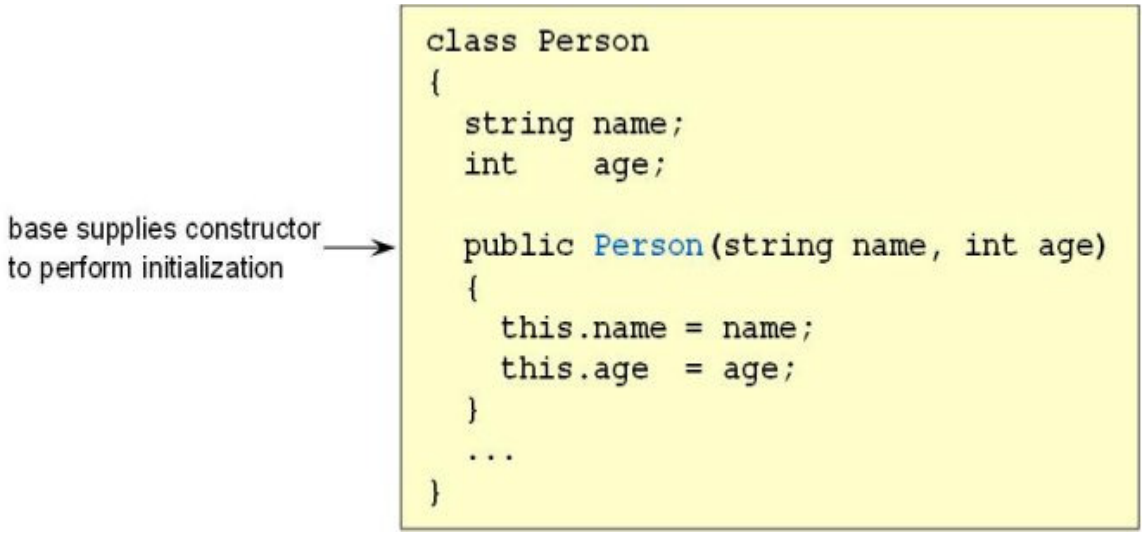


Construction and inheritance

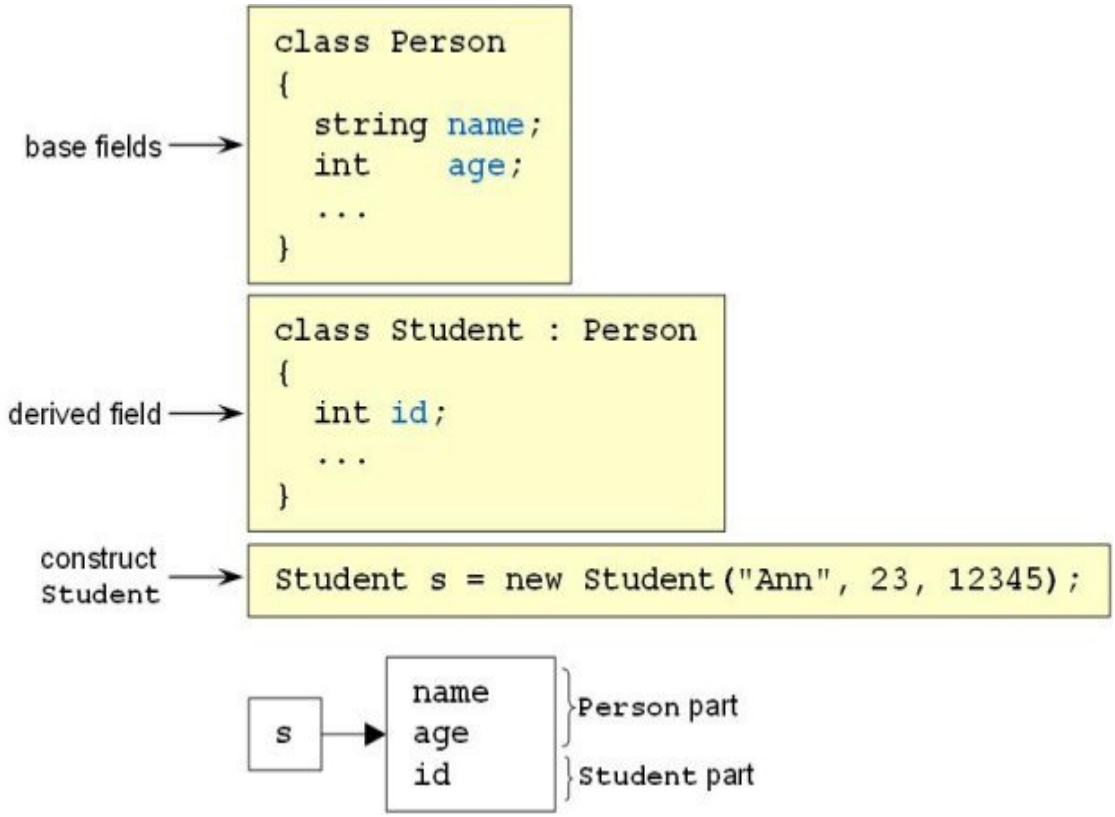
It is common for a base class to require initialization. For example, the following class would likely need to initialize the person's name and age.



The base class would probably offer one or more constructors to do the required initialization.

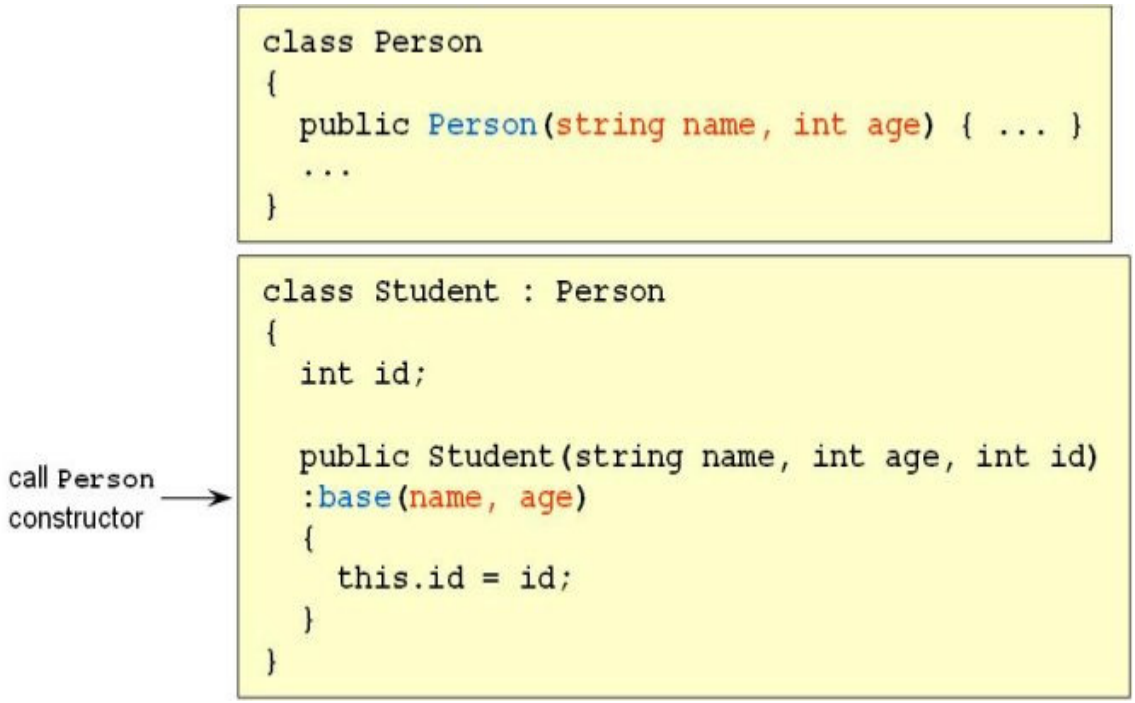


Derived classes inherit all the fields of the base class and add their own. When a derived class object is created, both the derived class part of the object and the base class part must be initialized.



The derived class constructor can invoke a base class constructor to initialize the base class part of the object using the `:base(...)` syntax. The constructor arguments are placed inside the parentheses as in a regular method call and the entire construct goes after the method signature but before the open brace for the constructor body. At runtime,

the base class constructor will be executed first followed by the body of the derived class constructor. The official name for this technique is a "constructor initializer".



08- Inheritance Exercise

Goals:

- Practice creating an inheritance hierarchy.
- Explore how common code can be placed in a base class.
- Call a base class constructor.
- Use new in a derived class to replace an inherited method.
- Use chaining to call a base class method from a derived class method.

Overview

In this lab we will use inheritance to model different assets in an investment application. The main base class will be Asset and we will have two derived classes: Stock and Property. We will also explore a multilevel inheritance hierarchy by deriving Building and Art from the Property class.

Part 1- Classes and fields

Here we will setup the inheritance hierarchy for the investment application. We will start off by defining the classes and their fields. Most of the code has been included to reduce the time needed for this basic part of the lab.

Steps:

1. Asset class. Create a class named Asset and give it three private fields: a string for the name, a double for the cost, and an int for the year the asset was purchased.

```
class Asset
{
        private string name;
        private double cost;
        private int    yearPurchased;
        ...
}
```

8. Stock class. Create a class named Stock that is derived from

Asset. Give it three private fields: a string for the ticker symbol, an int for the number of shares owned, and a double for the price.

```
class Stock : Asset
{
        private string symbol;
        private int    shares;
        private double price;
        ...
}
```

15. Property class. Create a class named Property and derive it from Asset. Give it one private field: a double for the assessed value.

```
class Property : Asset
{
        private double assessment;
        ...
}
```

20. Building class. Create a class named Building and derive it from Property. Give it one private field, a string for the address.

```
class Building : Property
{
        string address;
        ...
}
```

25. Art class. Create a class named Art and derive it from Property. Give it two private fields: a string for the name of the artist and an int for the year in which it was created.

```
class Art : Property
{
        private string artist;
        private int    yearCreated;
        ...
}
```

Part 2- Construction

Here we will practice with constructors in inheritance hierarchies. Remember that derived classes are responsible for initializing their own fields and those of their base class. Because of this, constructors tends to get more parameters farther down the hierarchy.

Steps:

1. Add a public constructor to each class in the inheritance hierarchy. Each constructor should take one argument for each field that it needs to initialize. For example, the Asset constructor will have three arguments since Asset has three fields. The Property constructor will have four arguments since it needs values for the three fields from Asset and an additional value for its own field. In your implementation of the derived class constructors, call the base class constructor using the :base syntax.
2. Create a driver class named Invest and add a Main method. Use code such as the following to test your work. You may need to adjust the parameter order to match your implementation.

```
  class Invest
  {
        static void Main()
        {
              Stock    microsoft  = new Stock   ("Microsoft", 8000,
1994, "MSFT", 200, 56);
              Building beachHouse = new Building("Beach House",
9000, 1964, 35000, "123 Seashore Ave, Malibu, CA");
              Art      nighthawks = new Art     ("Nighthawks", 850,
1955, 7500, "Edward Hopper", 1942);
              ...
        }
  }
```

Part 3- Code in base class

Here we will examine how common code can be placed in the base class and then be available to all derived classes. Without inheritance the common code would have to be repeated in every class.

Steps:

1. Add the following AmortizedCost method to the Asset class. There is no code to write here, simply copy the sample into your application.

```
  class Asset
  {
        public double AmortizedCost(int currentYear)
        {
              return cost / (currentYear - yearPurchased);
        }
```

```
8.              ...
    }
```

9. Apply the AmortizedCost method to each of your sample assets. Notice the benefit here: we wrote one method that now works for all types in the inheritance hierarchy.

```
10.  class Invest
11.  {
12.             static void Main()
13.             {
14.                         ...
15.                         Console.WriteLine("Amortized costs per year:");
16.                         Console.WriteLine("Microsoft stock ${0}", microsoft
    .AmortizedCost(2002));
17.                         Console.WriteLine("Beach house     ${0}",
    beachHouse.AmortizedCost(2002));
18.                         Console.WriteLine("Nighthawks      ${0}",
    nighthawks.AmortizedCost(2002));
19.                         ...
20.             }
    }
```

Part 4- Hiding inherited member

It can occasionally happen that a method with the same signature appears in both a base class and a derived class. When this happens, the derived class version hides the base class version. Some times this is intentional: the base class does something one way and the derived class wants to do it a bit differently. The designers of C# want to make sure that such hiding does not happen accidentally so they ask that the derived class method be labeled with the keyword new. If the method is not labeled new then a warning is generated by the compiler. This technique forces class designers to acknowledge when they are hiding an inherited method. The theory is that this will lead to more robust code since hiding will not occur accidentally.

Steps:

1. Add a public ComputeValue method to the Asset, Stock, and Property classes. The versions in the derived classes must be labeled with new to avoid a compiler warning. The implementation of the Asset version should return the cost, the Stock version should calculate the value using price and shares, and the Property version should return the assessed value.

```
2.   class Asset
```

```
3.    {
4.            public double ComputeValue()
5.            {
6.                    ...
7.            }
8.            ...
      }
```

9. Apply the ComputeValue method to each of your sample assets.

```
10.   class Invest
11.   {
12.           static void Main()
13.           {
14.                   ...
15.                   Console.WriteLine("Valuations:");
16.                   Console.WriteLine("Microsoft stock ${0}", microsoft
      .ComputeValue());
17.                   Console.WriteLine("Beach House     ${0}",
      beachHouse.ComputeValue());
18.                   Console.WriteLine("Nighthawks      ${0}",
      nighthawks.ComputeValue());
19.                   ...
20.           }
      }
```

Part 5- Chaining to base class

A derived class method may want to take advantage of some of the services offered by its base class by calling one of the base class methods. This often occurs between base and derived versions of the same method. The keyword base is used to indicate the base class version should be called in preference to the derived class method of the same name.

Steps:

1. Add a public Print method to each class in the hierarchy. Use base to allow the derived class versions to chain to the base class versions. After the call to the base version finishes, the derived versions can finish the job by printing the values of the fields in the derived class. Remember to use new in the signature of the derived class versions.

```
2.    class Asset
3.    {
4.            public void Print()
5.            {
6.                    ...
```

```
7.          }
8.          ...
    }
```

9. Apply the Print method to each of your sample assets.

```
10.  class Invest
11.  {
12.          static void Main()
13.          {
14.                  ...
15.                  microsoft .Print();
16.                  beachHouse.Print();
17.                  nighthawks.Print();
18.                  ...
19.          }
    }
```

09-Binding

Goals:

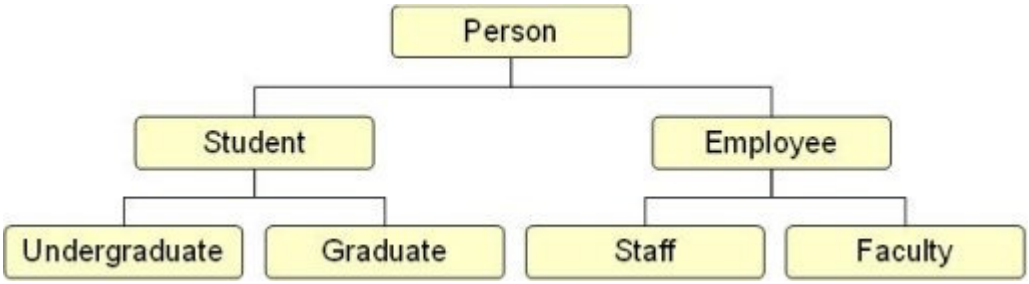
- Explain type compatibility under inheritance.
- Describe static and dynamic method binding.
- Discuss the details of dynamic binding: `virtual`, `abstract`, and `override`.
- Show how to use dynamic binding and polymorphism to write generic code.
- Cover downcasting and type testing.

Overview

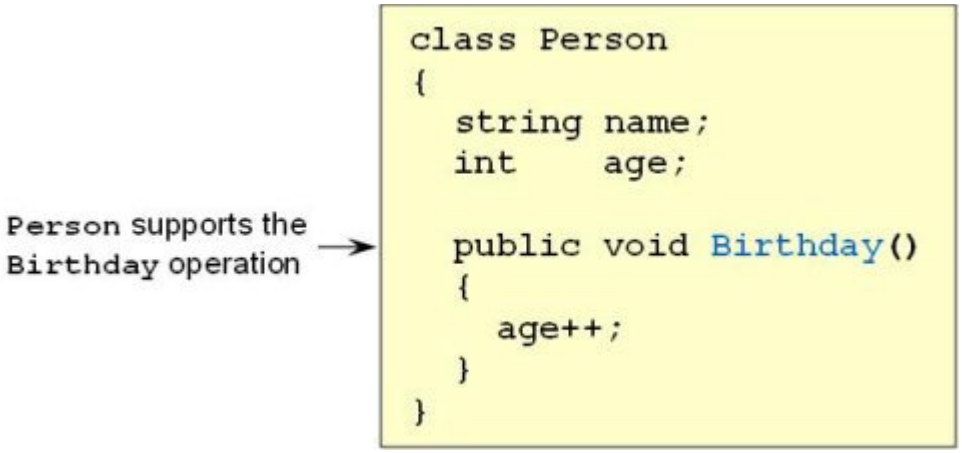
The most obvious advantages of inheritance are that it helps eliminate repeated code and allows program structure to mirror the organization of the real world. When inheritance is combined with reference compatibility and dynamic method binding things get even more exciting. Clients can then write extremely generic client code that works correctly for any type in the hierarchy. Dynamic binding ensures the correct code is executed for the type of object being processed.

Type compatibility

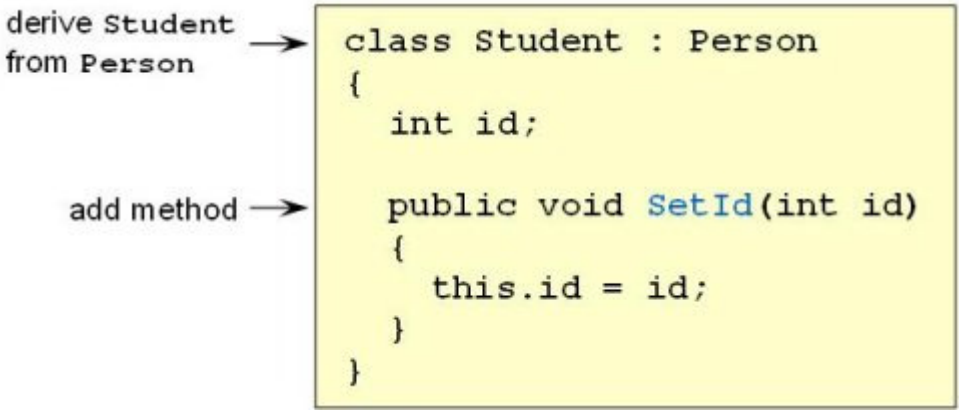
C# inheritance models the "is-a" relationship so a derived class inherits all the behavior of its base class. To explore the implications of this idea, we will return to the simple person/student/employee inheritance hierarchy which models people at a university.



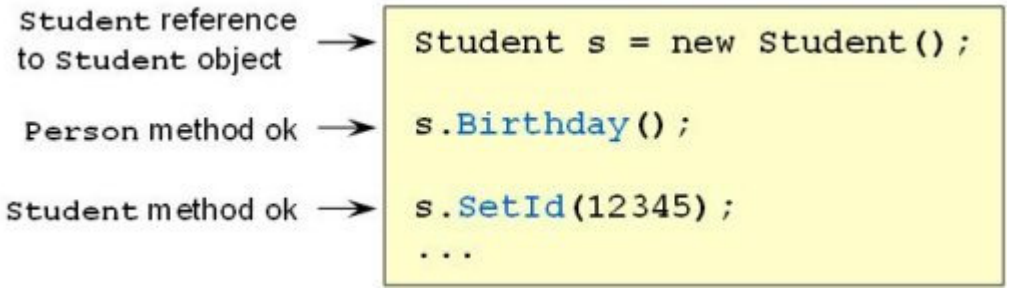
Suppose the base class person contains a public method named `Birthday` and some supporting fields.



Now we derive a student class from the person class. The student class inherits the `Birthday` operation and adds a public `SetId` method and a supporting field.



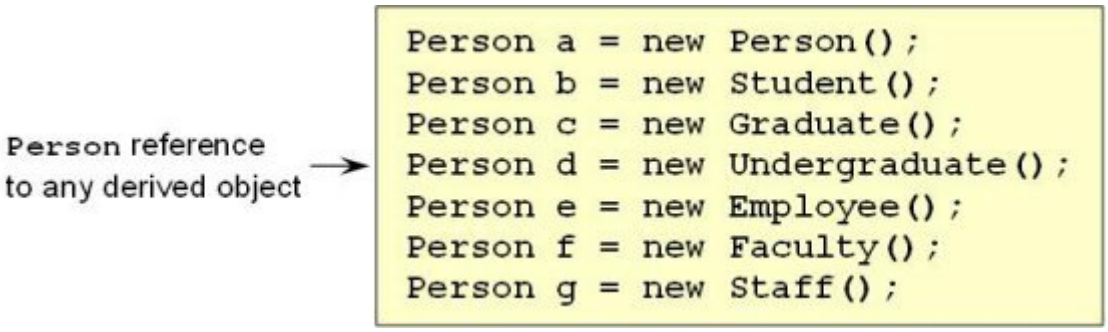
Typical client code using the student class might look like that shown below. A `Student` object is created and a `Student` reference is used to refer to the object. Using the `Student` reference, the client code can invoke the inherited `Birthday` method and the student `SetId` method.



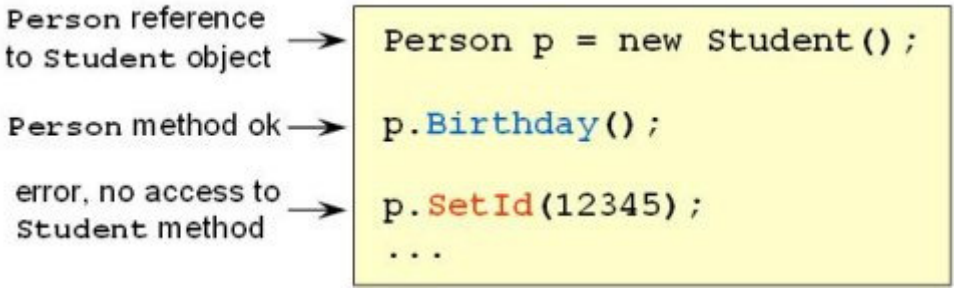
Things get more interesting when we consider the concept of type compatibility. Type compatibility is a natural consequence of the fact that inheritance models the is-a relationship. The formal way to express this idea is to say something like "a derived class

object has all the behavior of its base class and so can be used in any situation where a base class object is expected." In practice this means that we are able to use a base class reference to refer to any derived class object. In our person hierarchy, we could use a `Person` reference to refer to a `Student` object. The assignment is legal because a `Student` is-a `Person`.

In fact, a `Person` reference can refer to any object in the hierarchy.



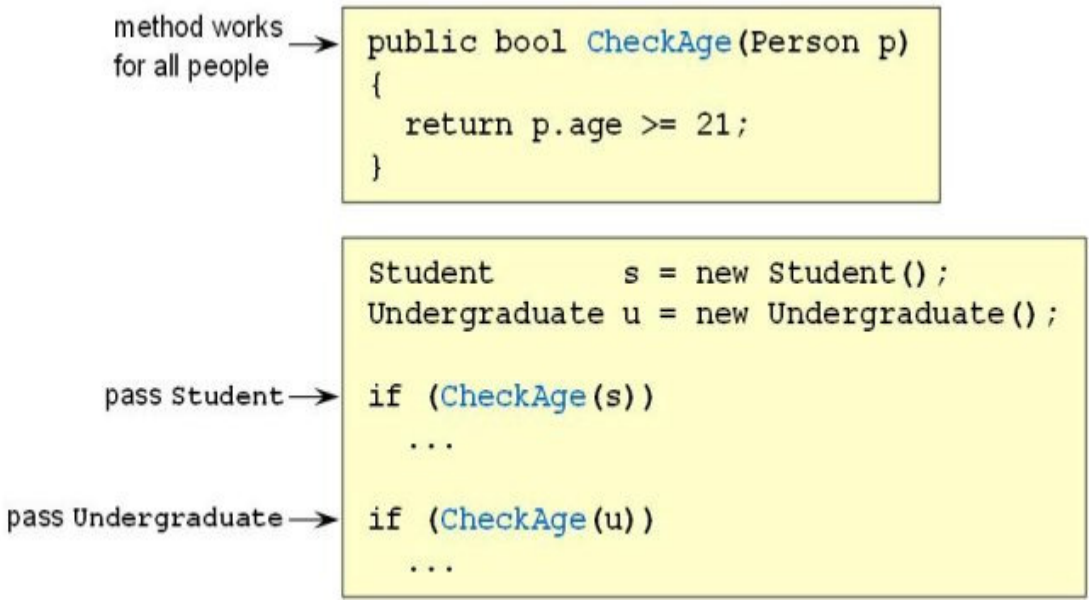
Let's put aside for the moment the major applications of this technique and simply examine the power and limitations of the code. The major advantage of using a base reference is that our code becomes a little bit more generic since only the type of the object is hardcoded. If we later decided to use a different type of object, say an `Undergraduate`, we would only need to modify the `new` statement where the object is created while the type of the reference used to manipulate the object would not have to change. The key limitation is that we are only allowed to access the `Person` part of any object when using a `Person` reference.



This is a key concept that shows up in many different situations so it's probably worth repeating. The type of the reference determines what access is allowed to the object. If the reference is of type `Person`, then only `Person` members can be accessed regardless of the actual type of the object. The object might be an `Undergraduate` that supports many

other useful operations; however, they will not be available through a `Person` reference.

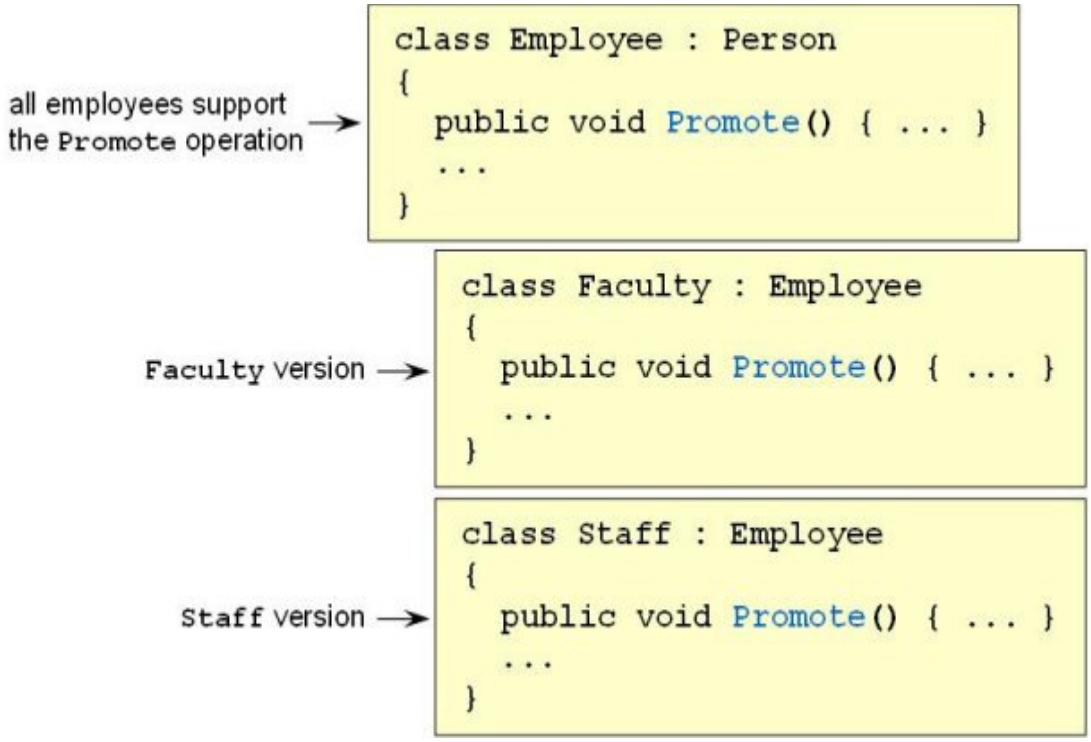
Despite this limitation, the use of a base class reference is an important and powerful technique. To begin to see the benefits, consider the following `CheckAge` method which has a parameter of type `Person`. Because the parameter is a base class reference, any derived class object can be passed as the argument. The `CheckAge` method is said to be generic because it works for all people.



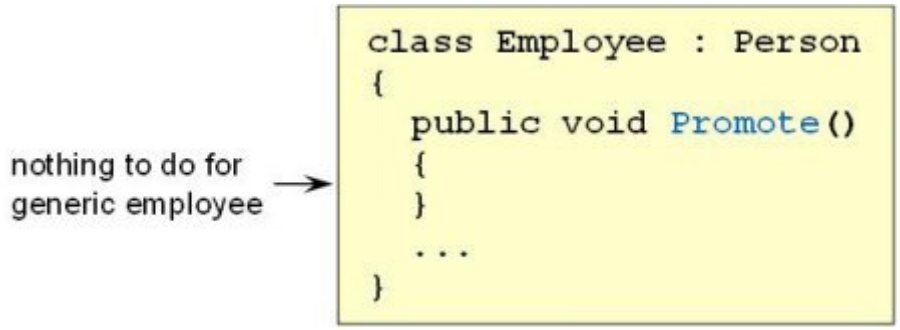
Without the type compatibility provided through inheritance it would be painful to support the `CheckAge` operation for each type in the hierarchy. Individual methods would need to be written for each type. The ability to write a single method that works for all types in the hierarchy provides a nice savings in code, coding time, and maintenance effort.

Method binding

Classes from an inheritance hierarchy often contain methods which logically perform the same operation. For example, every class in a hierarchy of different types of employees would naturally offer a `Promote` method.



Each class would implement their `Promote` method as appropriate to their type. Ironically, the `Employee` version would likely be the trickiest to code since it is difficult to know how to promote a generic employee. We'll solve this problem by simply putting an empty body for the implementation.



The `Faculty` promote method would of course involve a salary increase. In addition, faculty members might have their level increased each time they are promoted. When they reach a certain level they are rewarded with tenure so they can pursue long term research without the need to continuously publish.

Faculty implementation →

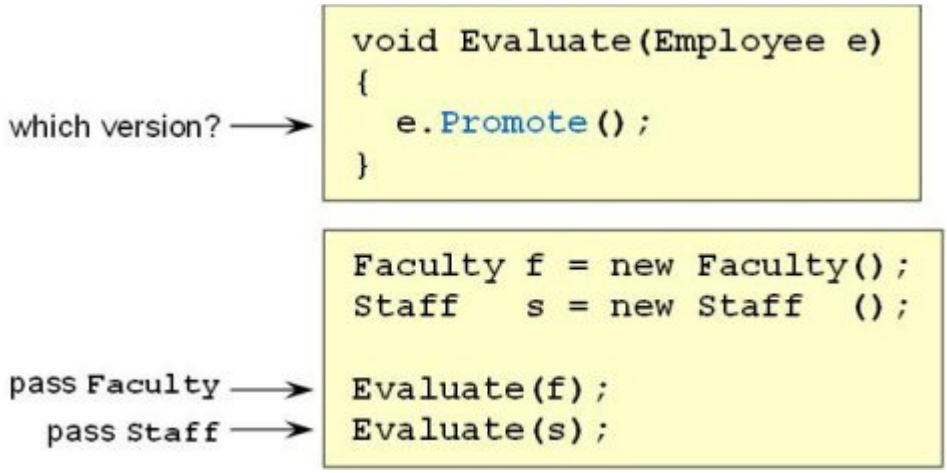
```
class Faculty : Employee
{
  public void Promote()
  {
    salary *= 1.2;

    level++;
    if (level > 4)
      tenure = true;
  }
  ...
}
```

The `Staff` version of promote would probably be quite simple with only a small increase in salary.

Staff implementation →

```
class Staff : Employee
{
  public void Promote()
  {
    salary *= 1.1;
  }
  ...
}
```

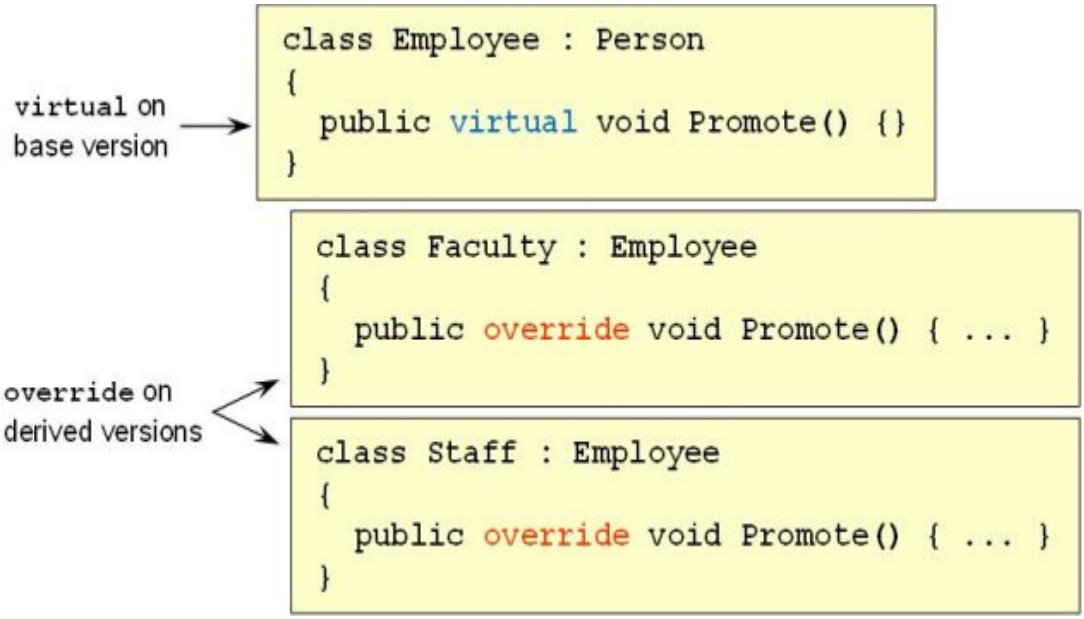
Now things get quite interesting since `Faculty` and `Staff` objects have two `Promote` methods, the one they inherit from `Employee` and their own version. To see the issue, consider the following client code. The `Evaluate` method has a parameter of type `Employee`. Type compatibility means we can pass an `Employee`, a `Faculty`, or a `Staff` member. Inside the method, the `Promote` method is called. Which version should be invoked?



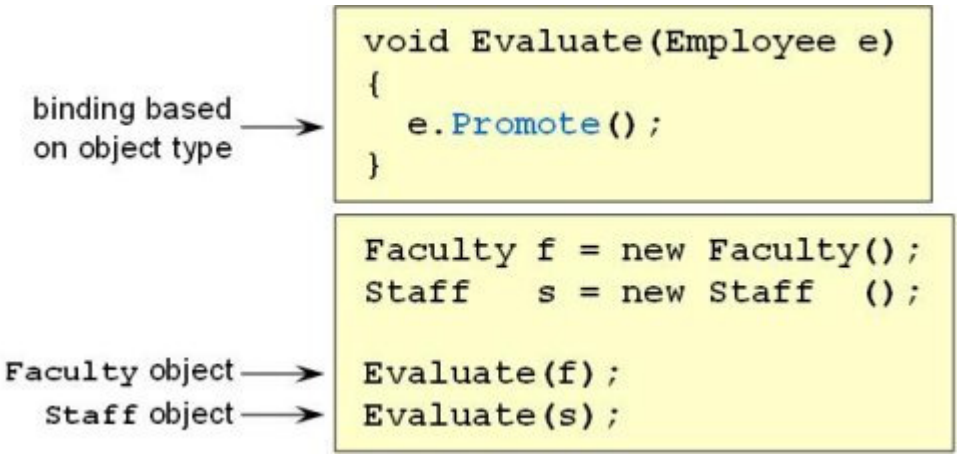
The process of deciding which method to invoke is called binding. There are two types of binding which are called different things in different programming languages. The first type is called static binding in the .NET world. Other names for this type of binding are early binding and compile time binding. Static binding decides which method to call based only on the type of the reference. In the `Evaluate` example static binding would always choose the `Employee` version of `Promote`. The other type of binding is called dynamic binding in .NET. Other names for this concept include runtime binding or late binding. Dynamic binding decides which method to call based on the actual type of the object involved. In essence, dynamic binding puts off the binding decision until runtime when the actual type of the object is known. At runtime, the dynamic binding mechanism determines the type of the object and invokes the appropriate version. In the `Evaluate` example, if the parameter happens to be a `Faculty` member then the `Faculty` version gets called, if it is a `Staff` member then the `Staff` version would be invoked.

The employee example has been carefully constructed so that dynamic binding is appropriate for the `Promote` operation. The key observation is that the `Promote` methods that appear throughout the hierarchy are all version of the same method. That is, the details of their implementations are of course different but conceptually they all perform the same operation.

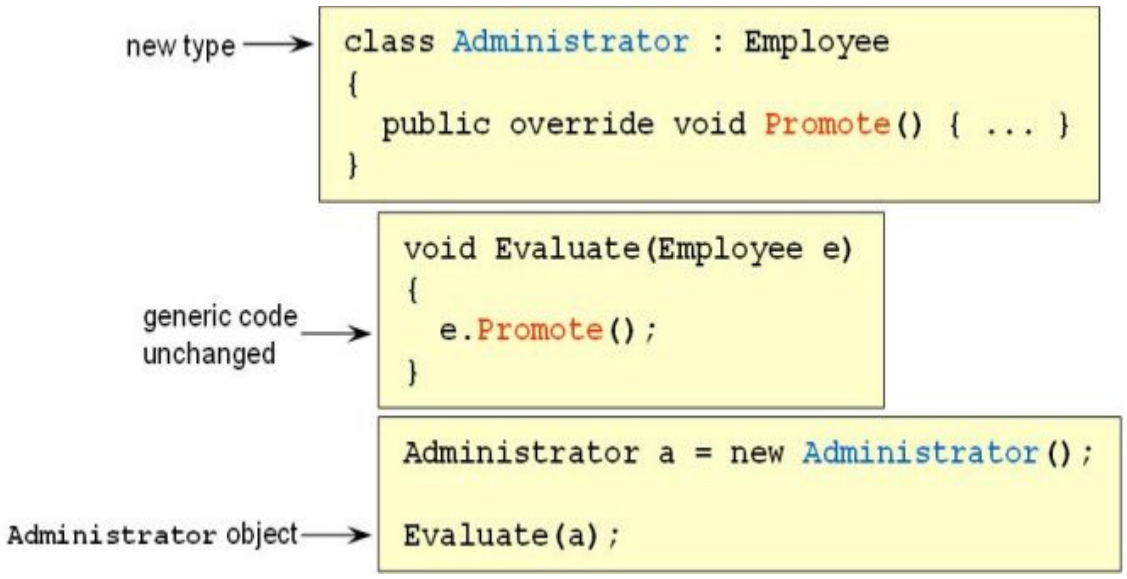
To get static binding we need do nothing since it is the default. For dynamic binding we need to apply two new keywords: `virtual` and `override`. The keyword `virtual` goes on the method at the top of the hierarchy where the operation first appears. The derived class versions are labeled with `override` to indicate they are the more specific version of the inherited virtual method. The implementation for the employee hierarchy is shown below.



Calls to the `Promote` method will now be dynamically bound. The behavior is illustrated by the code below.

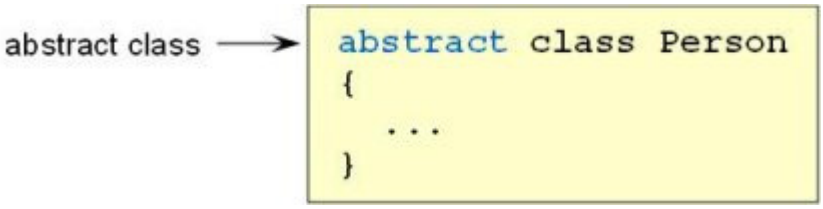


The thing that's so cool about all this is that the `Evaluate` method automatically adapts its behavior to whatever type of object is passed to it. We can even create new derived types of `Employee` and pass them to `Evaluate` and have their version of `Promote` called. This type of dynamic behavior is often called polymorphism (literally "many shapes" or "many forms"). Polymorphism lets us write generic code such as the `Evaluate` method that works correctly for all types in an inheritance hierarchy.

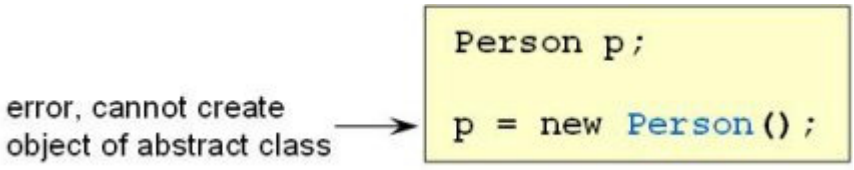


Abstract class

Some classes represent abstract concepts. Classic examples of this type of class include person, student, employee, shape, fruit, mammal, etc. These are abstract concepts because they represent only categories and not real concrete types. To model this in code, C# provides the `abstract` keyword. Any class that represents an abstract concept should be labeled `abstract` to make the intent of the class clear to the compiler and to other programmers. In the university people hierarchy, it is likely that the `Person`, `Student`, and `Employee` classes would all be abstract.

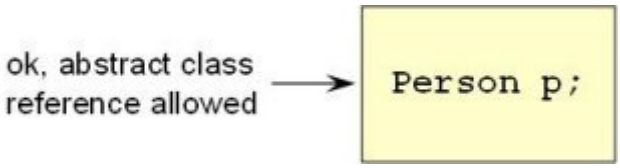


Marking a class `abstract` prevents objects of that class from being created. This makes sense because objects of the type do not exist in the real world anyway since the class is modeling an abstract concept.



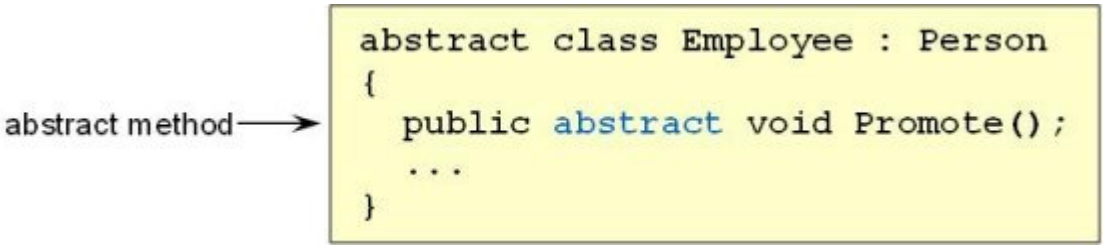
References are allowed even if the class is abstract. The references will be used to refer to

objects of concrete derived types.



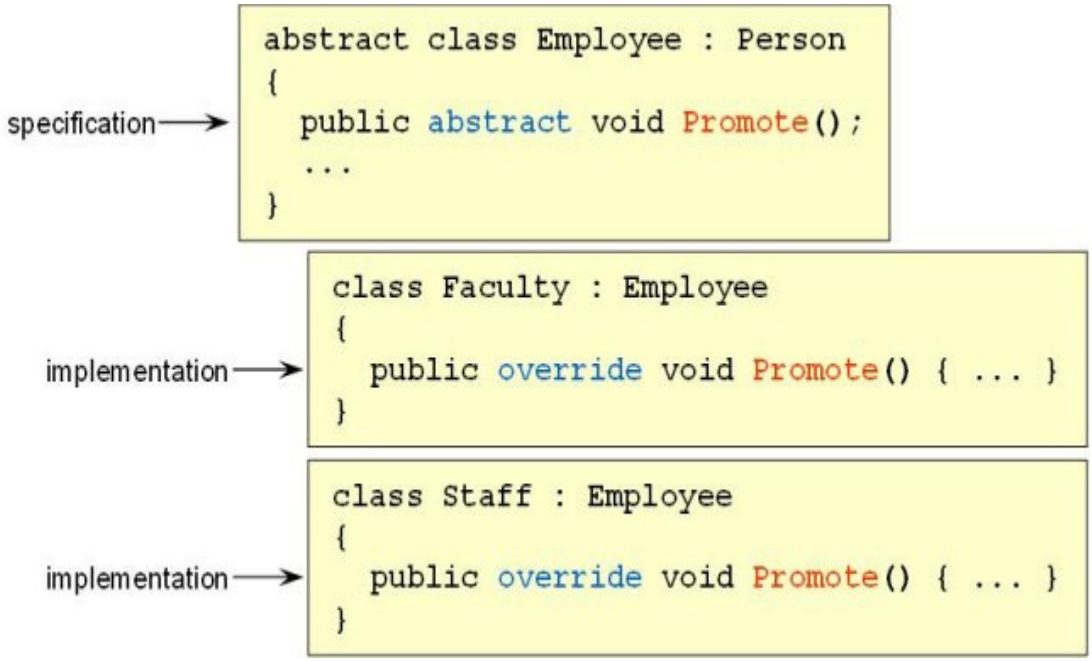
Abstract method

Sometimes there is no sensible implementation for a method in a base class. The `Promote` method in the `Employee` class provides a good example since there is no way to promote a generic employee. A meaningful implementation is only possible in a concrete derived class such as `Faculty` or `Staff`. The `abstract` keyword is applied to a method to indicate it is an abstract operation. Abstract methods are declaration only and cannot contain an implementation. The signature of an abstract method is followed only by a semicolon where the method body would normally appear.



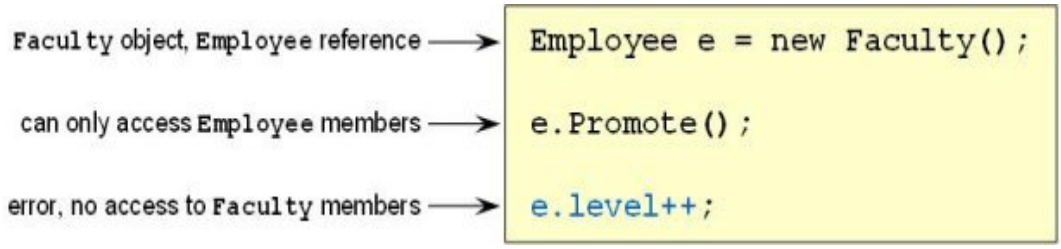
A class containing an abstract method is considered to be incompletely specified. For this reason, a class with an abstract method must be declared abstract or it is a compile time error.

An abstract method acts as a specification or contract that must be fulfilled by the derived classes. Derived classes need to provide an implementation for the inherited abstract method or they in turn become abstract classes. The derived class implementations are labeled with the keyword `override`.

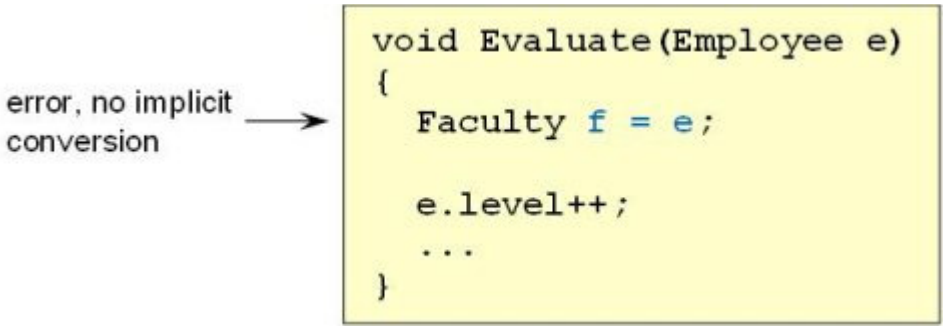


Downcasting

The type of reference determines what members of an object are available. This is generally a good thing since it limits generic code to only the operations all types in the hierarchy share. This is a key reason why writing generic code makes sense. This limitation can be frustrating when the derived class offers many services beyond those of the base class. The additional members are, of course, not available through the base reference.

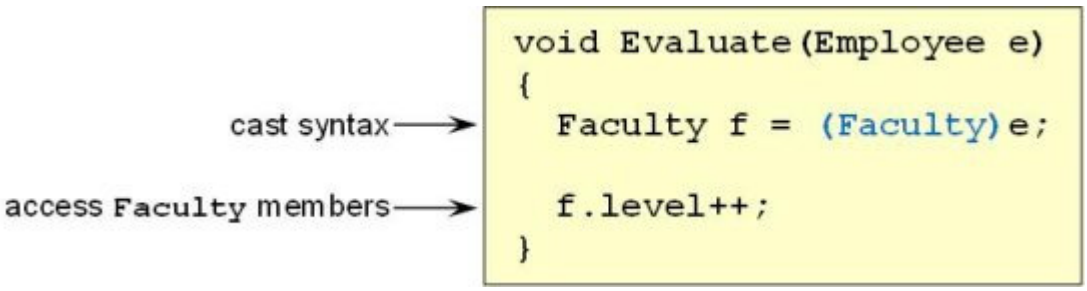


To access the additional members, we need to do a type conversion to obtain a more specific reference to the object. This type of conversion is often called a downcast since the conversion is moving down the hierarchy from base class to derived class. The compiler will not do these conversions automatically since there is the possibility that the conversion may fail. To see the issue consider the following code.

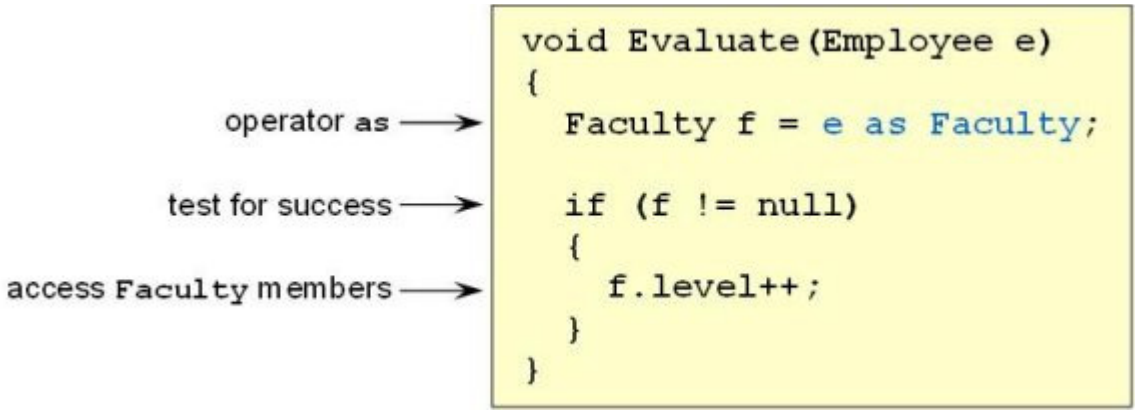


Sometimes the conversion will succeed (if the passed object is in fact a `Faculty`) and sometimes it will fail (if the passed object is not a `Faculty`). This possibility for failure is what motivates the compiler to refuse to do the conversion implicitly and to require the programmer to explicitly ask for it. An analogy might be the signing of a liability waiver. The compiler forces the programmer to acknowledge the possibility for failure and accept the risk and consequences of that failure.

The first way to do the conversion is with traditional cast syntax; that is, by placing the destination type name inside parentheses. If the object is of the specified type then the cast succeeds and a reference of the desired type is obtained. However, if the object is not of the right type the CLR will throw an `InvalidCastException`. The program can handle the exception using the `try`/`catch` syntax.



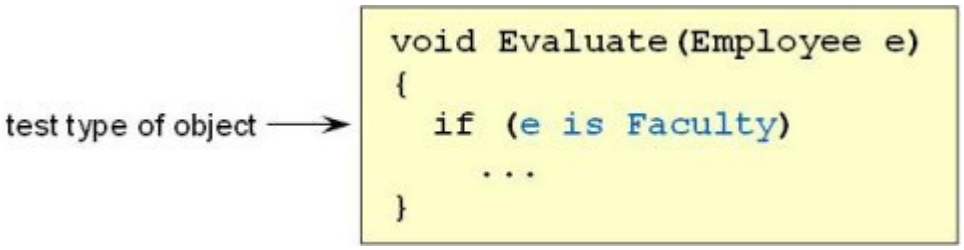
The other way to do the conversion is with the `as` operator. If the conversion succeeds then a valid reference results. If the conversion fails then the reference will be `null`.



There is one issue that is probably obvious by this time but which might be worth pointing out anyway. In this kind of type conversion, the thing that is being converted is the reference and not the object. We are simply obtaining a different reference to an existing object and the object itself is completely unchanged.

Type testing

It is possible to test the type of an object using operator `is`. The operator forms a Boolean expression that evaluates to true if the object is of the tested type and to false otherwise. Note that the type of the object is being tested and not the type of the reference. This should be obvious since the type of the reference is of course already known.



09- Binding Exercise

Goals:

- Write virtual and abstract methods.
- Override inherited methods in a derived class.
- Downcast from base to derived reference.
- Use dynamic binding to write generic code.

Overview

The process of deciding which method to call is often termed "binding". The topic seems rather simple at first: the method name and signature are used by the compiler to determine which method to call. However, the situation is more complicated since inheritance hierarchies can contain several methods with exactly the same name and same parameter list. In this case, C# allows the programmer to control how binding is performed.

The default behavior is to choose the method based on the type of the reference. This is called static binding because it can be performed at compile time (i.e. the type of the reference is known at compile time). Static binding has a couple of benefits: it is efficient and it is easy for programmers to determine by inspection which method will be called.

Programmers can make a method abstract or virtual to request that calls be bound based on the type of the object rather than the type of the reference. Since a base class reference can refer to any type of derived object, the compiler does not have enough information to perform this type of binding. The binding must then be delayed until runtime which is why it is often called "dynamic binding" or "late binding". Runtime binding does add a bit of extra overhead to program execution but is generally implemented efficiently. Dynamic binding is best known for supporting the creation of generic code, i.e. code that works for many different types.

Part 1- Setup

We will use a relatively simple inheritance hierarchy to practice with dynamic binding: a base class Pet and two derived classes Cat and Dog. In this first part we just setup the structure of the example. There are no methods and no dynamic binding yet. Feel free to code this on your own using the text descriptions or take advantage of the sample code provided.

Steps:

1. Code a class to represent a Pet. Make the pet class abstract since it models an abstract concept that should never be instantiated. Add a public string field to store the name of the pet. Add a protected constructor that sets the name. Making the constructor protected gives access to only the derived classes.
2. Derive a Cat class from Pet. Add a public bool field which stores whether or not the cat is picky. Provide a public constructor that sets the name and the pickiness of the cat. Call the Pet constructor to set the name rather than assigning to the protected field directly.
3. Derive a Dog class from Pet. Add a public bool field which stores whether or not the dog can fetch. Provide a public constructor that sets the name and the fetching ability of the dog. Call the Pet constructor to set the name rather than assigning to the protected field directly.
4. Create a class called Household. Add a Main method that creates a cat and a dog.

```
5.  using System;
6.
7.  namespace Binding
8.  {
9.          abstract class Pet
10.         {
11.                 public string Name;
12.
13.                 protected Pet(string name)
14.                 {
15.                         this.Name = name;
16.                 }
17.         }
18.
19.         class Cat : Pet
20.         {
21.                 public bool IsPicky;
22.
```

```
23.                     public Cat(string name, bool IsPicky)
24.                             : base(name)
25.                     {
26.                             this.IsPicky = IsPicky;
27.                     }
28.             }
29.
30.             class Dog : Pet
31.             {
32.                     public bool CanFetch;
33.
34.                     public Dog(string name, bool canFetch)
35.                             : base(name)
36.                     {
37.                             this.CanFetch = canFetch;
38.                     }
39.             }
40.
41.             class Household
42.             {
43.                     static void Main()
44.                     {
45.                             Cat c = new Cat("Ozzie", false);
46.                             Dog d = new Dog("Hank",  true);
47.                     }
48.             }
    }
```

Part 2- Reference compatibility

An inheritance hierarchy sets up an "is-a" relationship between a derived class and its base; for example, a cat is a pet. This leads to type compatibility: a base reference can refer to a derived object so a Pet reference can refer to either a Cat or a Dog. This generic behavior comes with a limitation in that only the base class members can be accessed through a base reference.

Steps:

1. Modify the Main method in the Household class to use Pet references to refer to the Dog and Cat. Attempt to access the members through the Pet references. The compiler will only allow access to the members of Pet and not to members of Dog or Cat.

Part 3- Virtual Method

The keyword virtual is used to turn on dynamic binding for a method. Virtual is used only on the method in the base class. Derived classes that provide their own versions of the method use the keyword override.

Steps:

1. Add a virtualMove method to the Pet class. Move should take a bool that indicates whether the pet should move fast or slow. Print a message that indicates the name and behavior: pets walk when moving slow and run when moving fast. Notice that we are able to fully implement the Move method here because there is reasonable generic behavior: walking and running make sense for all pets.

```
abstract class Pet
{
        public virtual void Move(bool fast)
        {
                ...
        }
        ...
}
```

9. Override the Move method in the Cat class. Change the behavior from the generic walking and running to something more catlike: cats pounce when moving fast and slink when moving slow.

```
class Cat : Pet
{
        public override void Move(bool fast)
        {
                ...
        }
        ...
}
```

17. Override the Move method in the Dog class. Change the behavior from the generic walking and running to something more doglike: dogs bound when moving fast and stride when moving slow.

```
class Dog : Pet
{
```

```
20.             public override void Move(bool fast)
21.             {
22.                             ...
23.             }
24.             ...
    }
```

Part 4- Abstract method

The keyword abstract is similar to virtual in that it also turns on dynamic binding for a method. As with virtual, abstract is used only on the method in the base class while derived classes use the keyword override. The key difference is that an abstract method is signature only - no implementation is allowed. Derived classes must implement all inherited abstract methods or they become abstract classes.

Steps:

1. Add an abstractSpeak method to the Pet class. There is no reasonable implementation for the Speak method that would apply to all pets; therefore we chose an abstract method rather than a virtual method.

```
2.    abstract class Pet
3.    {
4.              public abstract void Speak();
5.              ...
    }
```

6. Override the Speak method in the Cat class. Print the pets name and the string "meows".

```
7.    class Cat : Pet
8.    {
9.              public override void Speak()
10.             {
11.                             ...
12.             }
13.             ...
    }
```

14. Override the Speak method in the Dog class. Print the pets name and the string "barks".

```
15.   class Dog : Pet
16.   {
17.             public override void Speak()
18.             {
19.                             ...
```

```
20.             }
21.             ...
    }
```

Part 5- Generic code

The key benefit of type compatibility, virtual methods, abstract methods, and dynamic binding is their support for generic code. Code is "generic" if it can be run for multiple types. To demonstrate generic coding we will write two methods to exercise our pets: the first one will be simple to show the basics, the second one will be slightly more interesting.

Steps:

1. Add a method named Exercise to the Household class. The method should take the pet that needs the workout and an integer to indicate the duration. The pet should Speak before the workout and then Move the number of times indicated by the duration. Call Exercise from Main passing first a Cat and then a Dog. Verify that the calls to Speak and Move are being dynamically bound. The advantage of this coding pattern is that we only have to write the Exercise method once and it will work with all derived types, even new derived types that we create in the future.

```
2.    static void Exercise(Pet p, int duration)
3.    {
4.              ...
    }
```

5. Add another version of Exercise to the Household class that takes an array of Pet references. Loop through the array and have each pet Speak and Move once. In Main, create an array of Pet references an fill it with some cats and some dogs. Pass the array to Exercise and verify that the calls to Speak and Move are being dynamically bound.

```
6.    static void Exercise(Pet[] p)
7.    {
8.              ...
    }
```

Part 6- Specific code: type testing and

downcasting

Code that uses a base reference is restricted to accessing only the base class parts of the object. To get around this restriction, a downcast can be used to obtain a more specific reference. The resulting code has greater access to the object but is less generic.

Steps:

1. Add the following Play method to the Household class. Since the parameter type is Pet reference, either a Cat or a Dog can be passed. It is possible to determine the actual type of an object using operator is. The operator yields true if the object is of the tested type and false otherwise. Code the Play method to determine whether the parameter is a Cat or a Dog and print a message stating the type.

```
2.    static void Play(Pet p)
3.    {
4.              if (p is Cat)
5.                        ...
6.              ...
}
```

7. Cats and Dogs play very differently so we will need to execute specific code depending on the type of the object passed to Play. Picky cats will not play at all and even non-picky cats tire quickly so they should not move fast for too long. Dogs that can't fetch should not be played with either. Notice that to implement the required behavior we need to determine whether a cat is picky and whether a dog can fetch. We cannot make this determination using a Pet reference so we will need to create more specific references in order to get full access to the objects.

   A downcast is required to convert from base reference to derived reference. Downcasting is considered a bit dangerous since there is the possibility that the cast will fail. For example, if you have a Pet reference referring to a Dog then casting to Dog will succeed but casting to Cat will fail. There are two ways to code a downcast: traditional cast syntax or the as operator. The two options differ in how they react to failure: a traditional cast throws an InvalidCastException while the as operator yields a null reference. Modify the Play method to downcast the parameter to the appropriate type. To practice with both forms of casting, use

cast syntax for cats and the as operator for dogs. After the downcast, use the specific references to determine if the passed cat is picky or if the dog can fetch. Implement the Play algoritms appropriately.

Notice that introducing type testing and downcasting into the Play method makes it less generic. Play only works for Cats and Dogs and not for any other type of Pet. If we create a new derived class in the future we will have to modify Play to handle the new case. This is another reason that downcasting is to be avoided when possible.

```
Cat c = (Cat)p;
...
Dog d = p as Dog;
```

10-Interfaces

Goals:

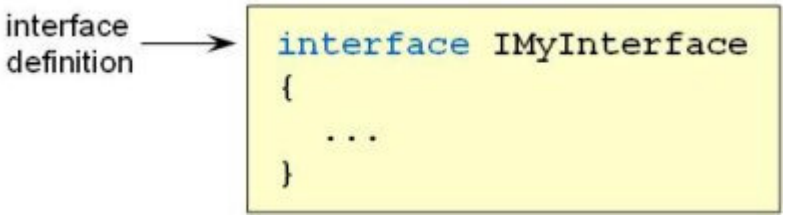
- Introduce the concept of interface.
- Show how to define an interface.
- Show how a class implements an interface.
- Show how to implement two interfaces that contain methods with the same signature.
- Discuss how to use interface references to write generic code.

Overview

An interface specifies a set of method signatures, indexer declarations, property declarations, and events. A class implements an interface by including the interface in its list of base classes and coding the interface contents. A type can implement as many interfaces as required. Clients can write generic code by using an interface reference to access objects of any type that implements the interface.

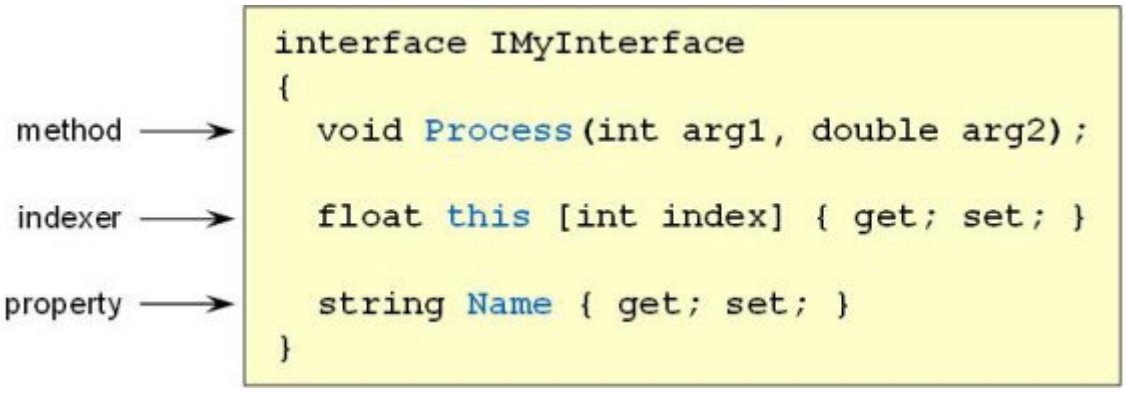
Interface definition

An interface is defined using the keyword `interface`, specifying a name, and providing a body enclosed in curly braces. It is common practice to prefix the name of an interface with an `I` but is not required.



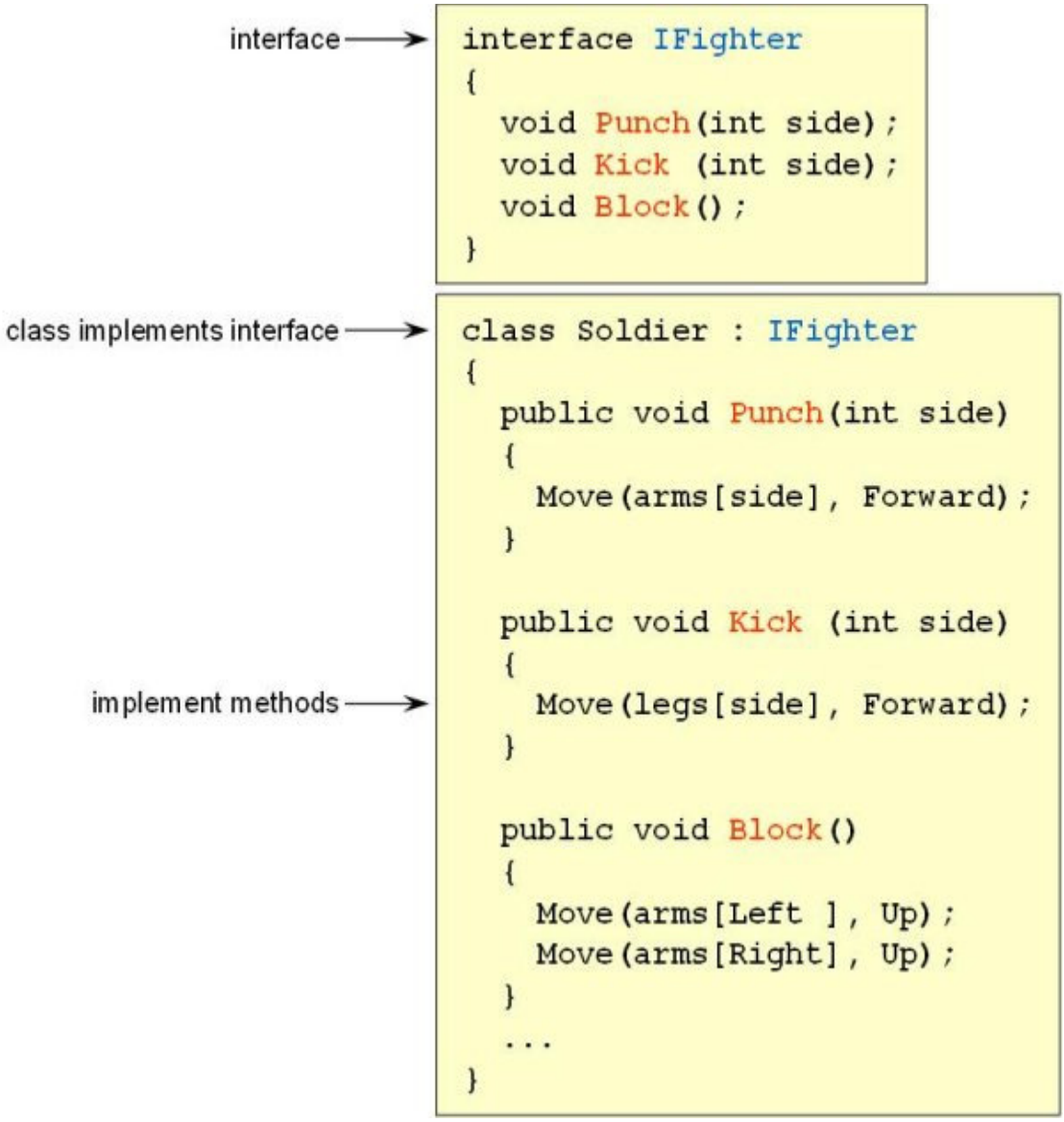
An interface body is limited to declarations for methods, indexers, properties, and events. Fields, constructors, constants, statics, etc. can not be included in an interface. Interface members are implicitly public and it is a compiler error to specify the access level explicitly. No implementations are allowed in an interface. Methods are specified by giving only the return type, name, and parameter list followed by a semicolon. Indexer declarations specify type, indices, and get/set accessor declarations. Property declarations list the type, name, and get/set accessor declarations. Indexers and properties can be read-

write, read-only, or write-only by specifying accessors as desired; however, at least one accessor is required. An interface can also contain events but we won't discuss it here since we have not yet covered events.

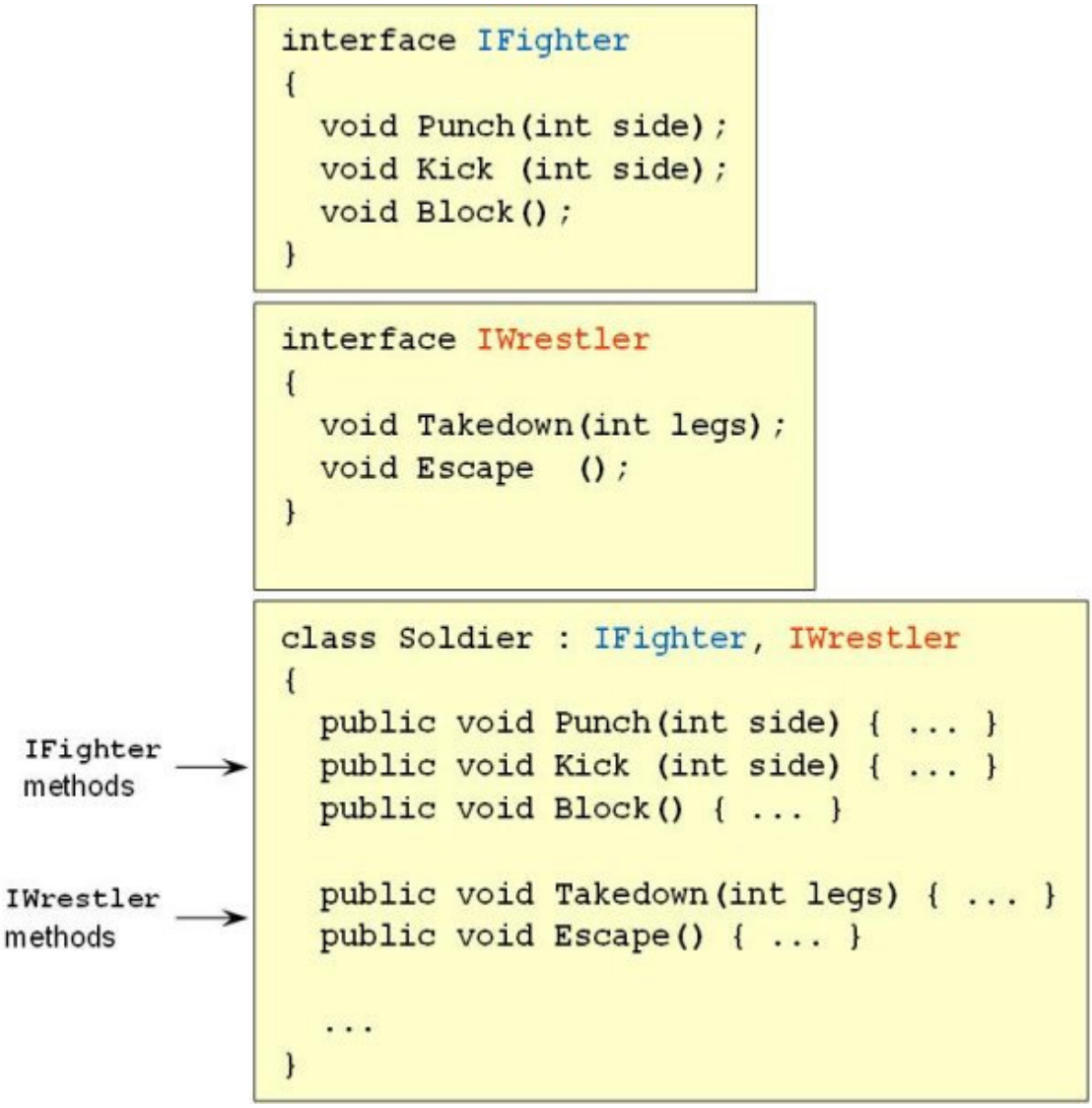


Interface implementation

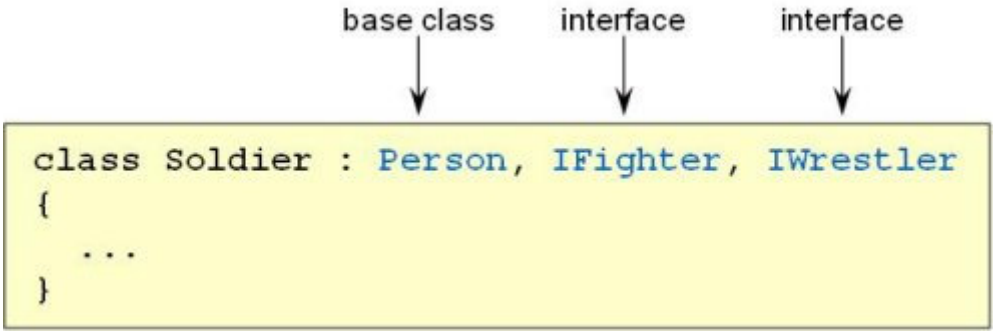
A class implements an interface by including the interface name in its list of base classes and coding the interface members. The `public` access level for interface members is implied in the interface but must be explicitly specified in the implementing class.



A class can implement multiple interfaces by listing each interface in its base class list and coding all the elements of each interface.

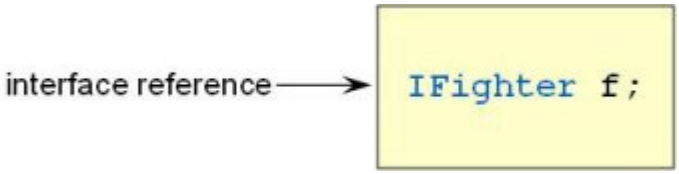


If a class has a base class and implements interfaces, the base class must appear first in the list of base classes or it is a compile time error.

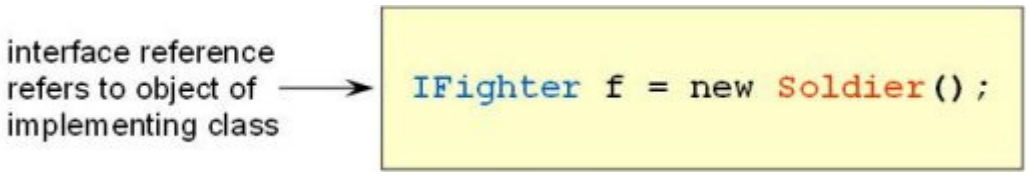


Interface reference

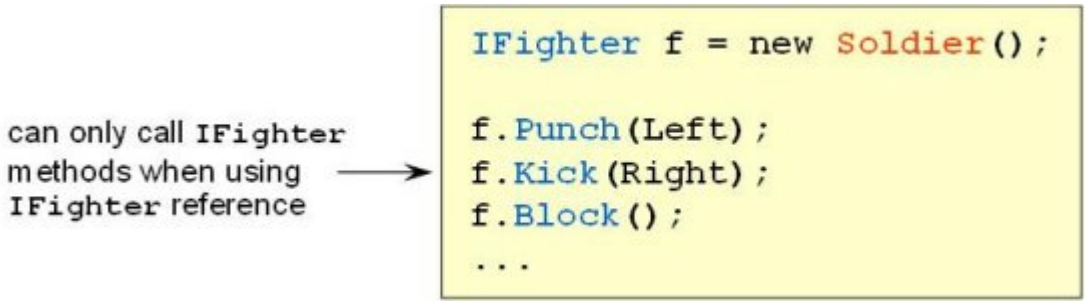
It is possible to declare a reference of an interface type.



Interface references are used to refer to objects of types that implement the interface. For example, if a `Soldier` class implements the `IFighter` interface, then an `IFighter` reference can be used to refer to a `Soldier` object.



As always in C#, the type of the reference determines what access is allowed. Using an interface reference restricts the client code to only the members that are part of the interface.



Generic code

Using an interface reference to access implementing class objects is an important technique because it allows the writing of generic code. Consider the `WarmUp` method shown below which uses an interface reference of type `IFighter` as its parameter.

generic code, works for all `IFighter` types →

```
void WarmUp(IFighter f)
{
  f.Punch(Left );
  f.Punch(Right);
  f.Kick (Left );
  f.Kick (Right);
}
```

An object of any class that implements `IFigther` can be passed to the `WarmUp` method.

ok since `Soldier` implements `IFighter` →

```
Soldier s = new Soldier();

WarmUp(s);
```

At this point, the example is not terribly compelling since we have only one class that implements `IFighter`. As the number of implementing classes grows, the benefits get magnified since the `WarmUp` method works unchanged with all the new types.

implement `IFighter` →

```
class Monkey : IFighter
{
  ...
}
```

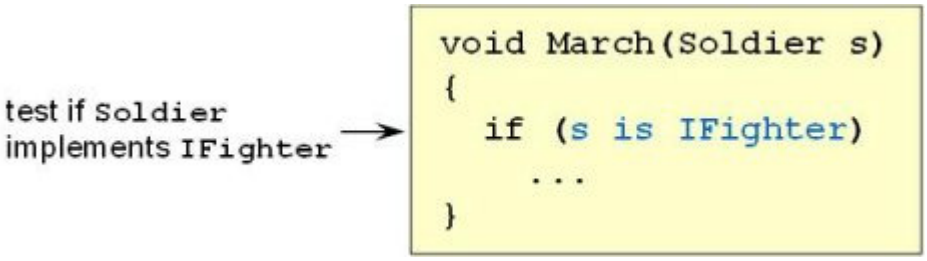
```
class Robot : IFighter
{
  ...
}
```

ok, both implement `IFighter` →

```
Monkey m = new Monkey();
Robot  r = new Robot ();

WarmUp(m);
WarmUp(r);
```

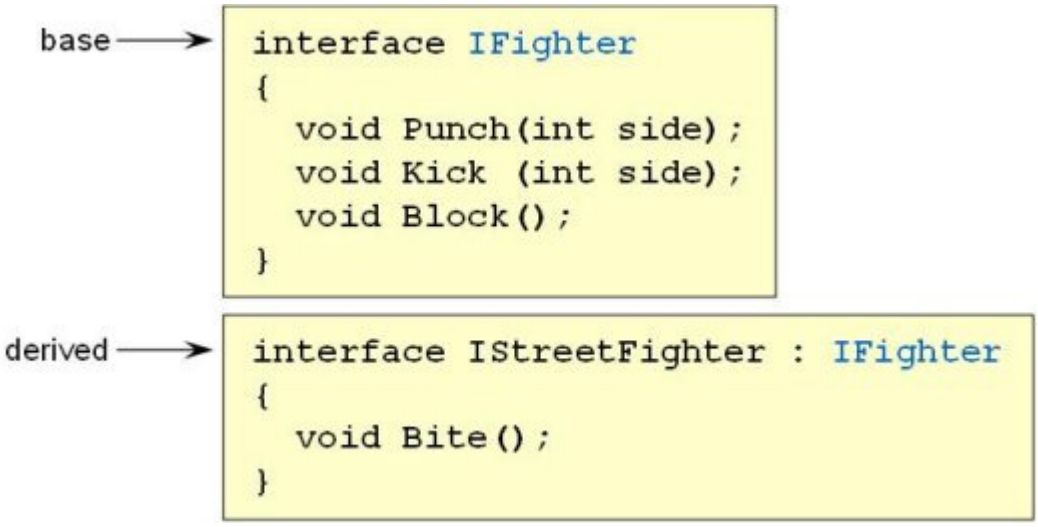
Type testing

The `is` operator can be used to determine if an object implements an interface. The operator creates a Boolean expression that evaluates to true if the object being tested implements the specified interface.

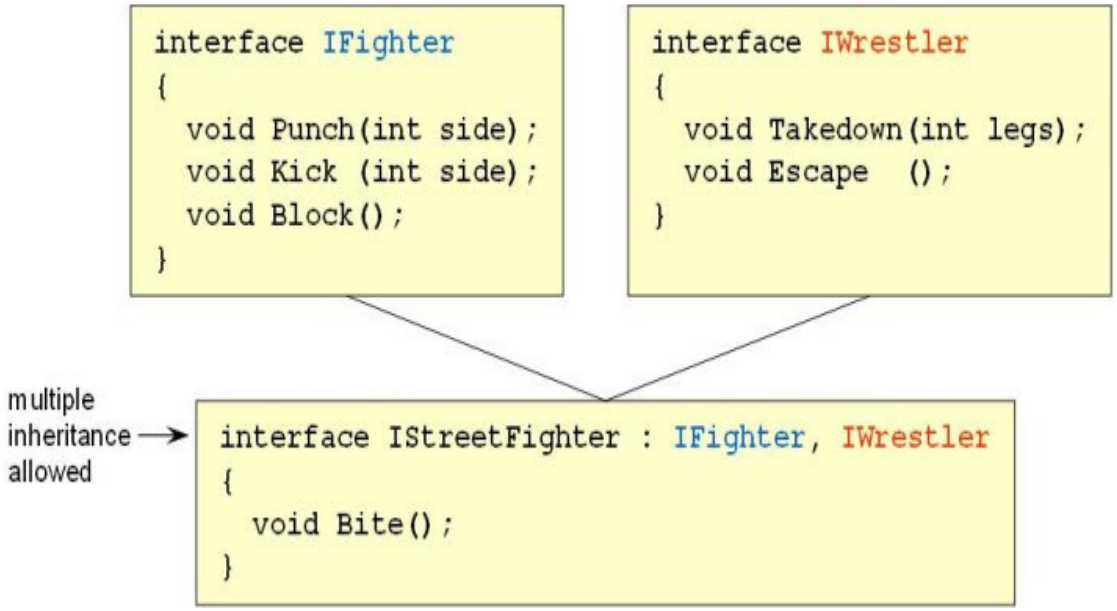


Interface inheritance

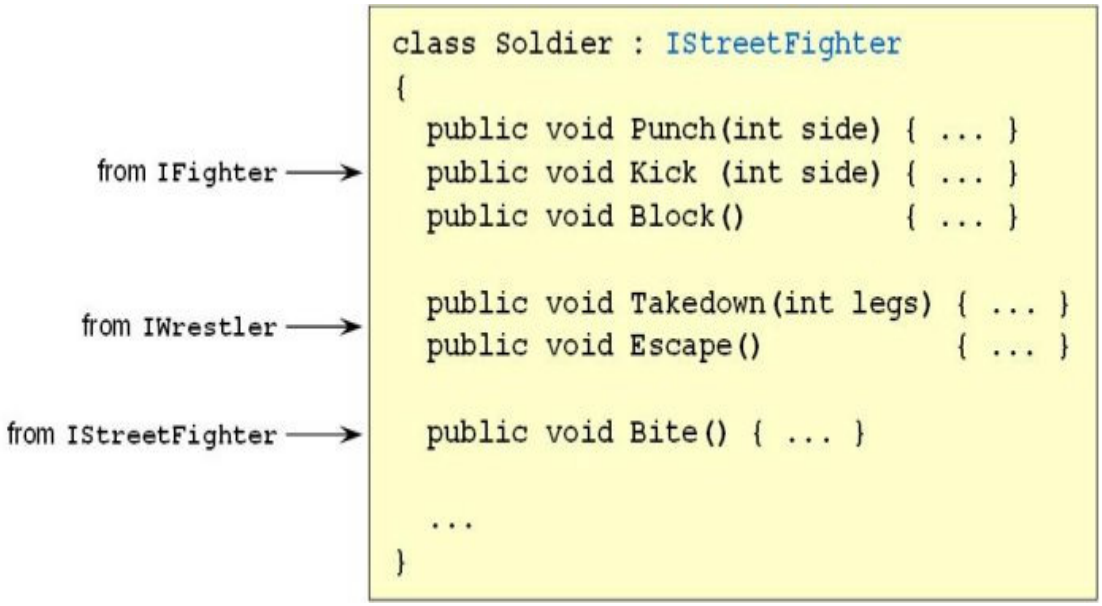
One interface can be derived from another in a process very similar to inheritance with classes. The derived interface inherits all the contents of it base interface plus it can add new members.



Interfaces are also allowed to have multiple base interfaces. The derived interface inherits all the members of every base interface and can add new members as well.



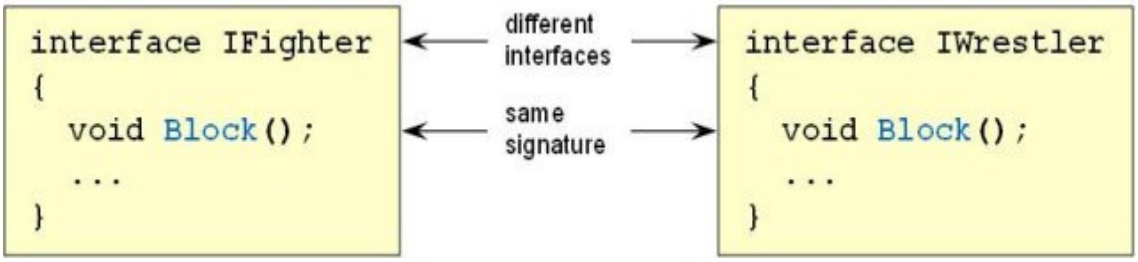
A class that implements a derived interface is required to code all the members of the base and derived interfaces.



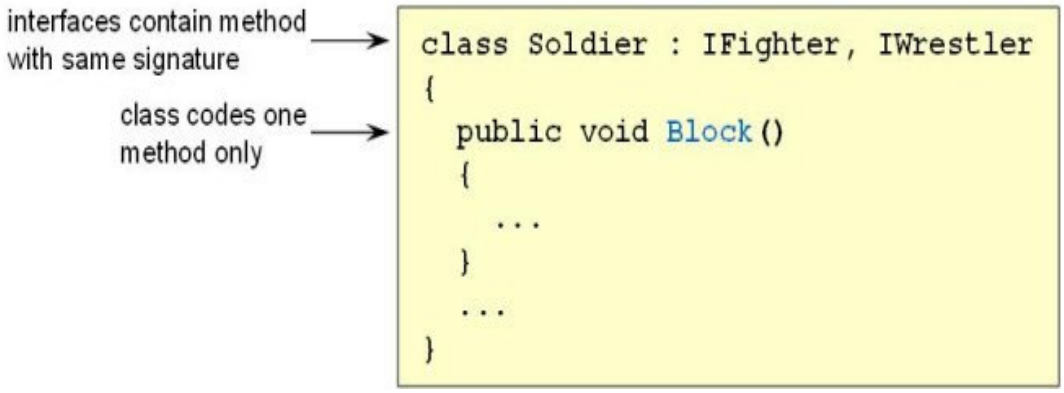
Ambiguity

It sometimes happens that two interfaces contain a method with the same signature (name and parameter list). For example, both an `IFighter` and an `IWrestler` interface might contain a method called `Block`. The duplicate method causes a problem for any class that

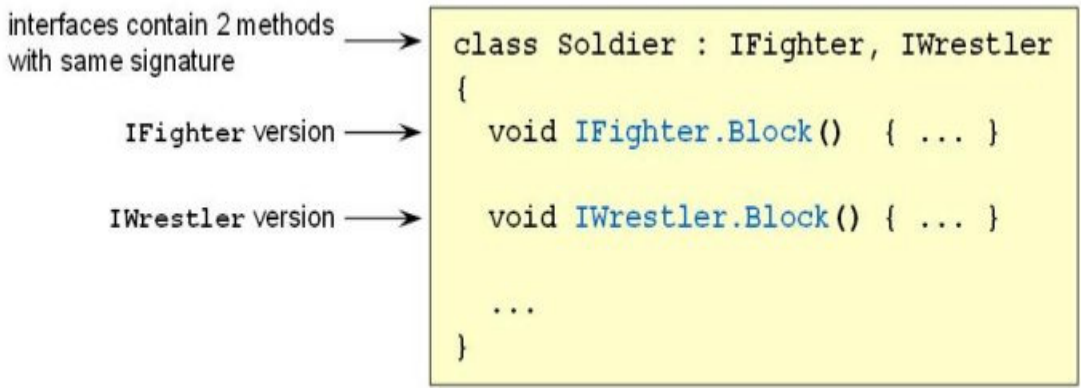
would like to implement both interfaces.



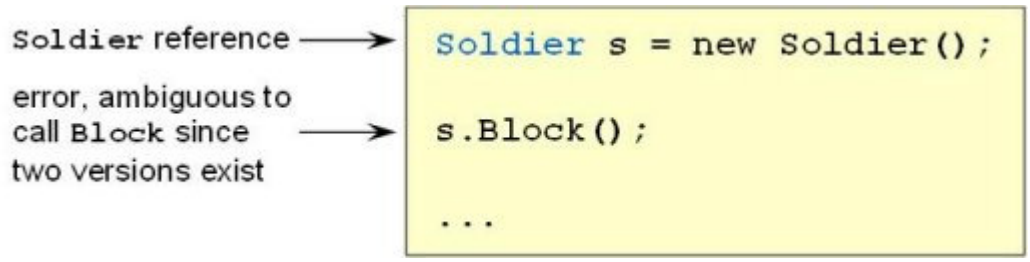
C# provides two ways to overcome the ambiguity. In the first technique, the class implements a single method to fill both roles. The main problem with this technique is the difficulty of deciding what code to put in the single method. Do you combine the code that would normally go in the separate methods together in the single implementation? Do you choose just one behavior and ignore the other one? Because of the difficulty in choosing an implementation, this solution is not used too often in practice.



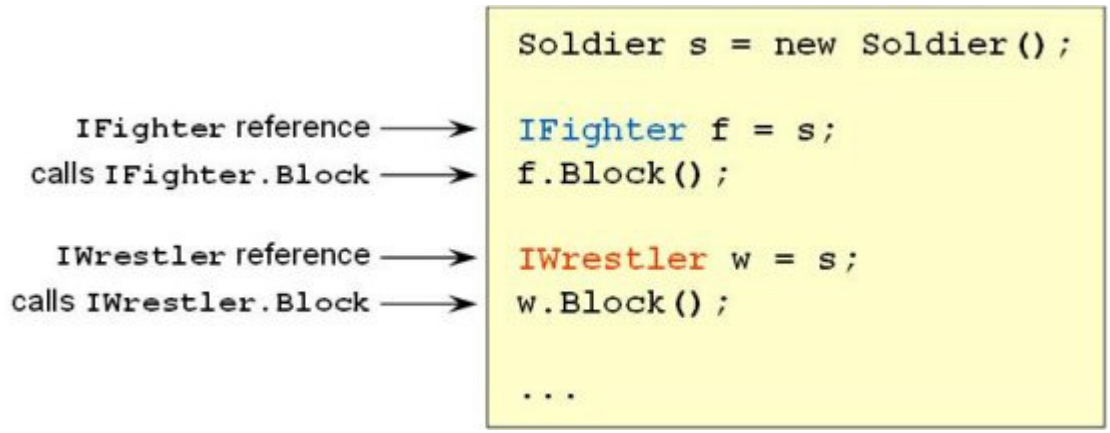
The second technique allows the programmer to code the two methods separately. Since the two methods have the same signature there is a bit of special syntax that must be used to specify which method belongs to which interface. Each method implementation is qualified with the interface name in order to provide the needed clarification. The official name for this technique is "explicit method implementation."



The basic idea behind calling an explicit method implementation is quite simple. There are two methods with the same signature so we somehow have to indicate which of the two we would like. Simply using a reference of the class type and attempting the call will not work since it is ambiguous.



The right way to call an explicit method implementation is through an interface reference. The type of the reference is used to determine which version gets called. For example, using an `IFighter` reference would invoke the `IFighter` version while going through an `IWrestler` reference would call the `IWrestler` implementation.



Before we finish, let's examine one minor syntactic detail with explicit method implementation. Look back at the two `Block` methods inside the `Soldier` class and notice that there is no `public` access modifier anywhere on the implementations. This is a bit of an odd situation since normally omitting the access modifier would mean the

methods default to `private`. If the methods were truly private we would be violating one of the basic rules of interfaces (that the contents are public). So we have what is affectionately known in the industry as a "special case". The compiler considers the methods to be private when a `Soldier` reference is used but `public` when an interface reference is used.

10- Interfaces Exercise

Goals:

- Write an interface.
- Code classes that implement an interface.
- Show how inheritance and interfaces are used together.
- Resolve ambiguity using explicit method implementation.

Overview

An interface is a group of declarations: methods, indexers, properties, etc. The interface provides the declarations only - a class supports the interface by supplying the necessary implementations.

In this lab we will attempt to illustrate not only the basics of interfaces, but also the common situation where interfaces and inheritance are used in the same design. Here is the idea: C# supports only single inheritance so many times, inheritance is used to model the primary characteristic while secondary traits are spun off into interfaces. This often yields a very flexible design in which the classes in the inheritance hierarchy implement only those interfaces that are applicable to them.

In our example, we will first create an inheritance hierarchy for different types of employees. There will be a base class Employee and three derived classes Programmer, Manager, and Intern. We use inheritance here for two reasons. The first reason is simply a judgment call: it seems logically correct from a design point of view that the trait of being an Employee is fundamental and deserves to be represented using inheritance. The second reason is more concrete: we would like to put code in the Employee base class that all derived classes can use. In the first part of the lab we use only inheritance, no interfaces.

Next we would like to promote our employees by coding a Promote method. However, not all employees are eligible for promotion: Programmers and Managers get promoted, but Interns are only temporary employees and are not eligible. Because not all our classes should support the Promote operation, it would be incorrect to add Promote to the Employee class. Instead, we define a separate interface IPromoteable for the Promote method. Programmer and

Manager implement the interface while Intern does not.

Part 1- Inheritance

Implement an inheritance hierarchy for the different types of employees at a company. The abstract base class will represent a generic employee. The three derived classes represent the various types of concrete employees: programmers, managers, and interns. Each class has some fields, a constructor, and a print method. Feel free to code this on your own using the text descriptions or take advantage of the sample code provided.

Steps:

1. Code a class to represent an Employee. Make the employee class abstract since it models an abstract concept that should never be instantiated. Add two protected fields: a string for the name and a double for the salary. Add a protected constructor that sets the name and salary. Making the constructor protected gives access to only the derived classes. Add a public virtual Print method that prints the name and salary of the employee to the console.
2. Derive a Programmer class from Employee. Add a protected double field to store the average overtime worked by the programmer. Provide a public constructor that sets the name, salary, and overtime of the programmer. Call the Employee constructor to set the name and salary rather than assigning to the protected fields directly. Override the virtual Print method to chain to the base class version first and then print out the derived class fields.
3. Derive a Manager class from Employee. Add a protected string field to store the name of the manager's secretary. Provide a public constructor that sets the name, salary, and secretary of the manager. Call the Employee constructor to set the name and salary rather than assigning to the protected fields directly. Override the virtual Print method to chain to the base class version first and then print out the derived class fields.
4. Derive an Intern class from Employee. Add a protected int field to store the number of months of the internship. Provide a public constructor that sets the name, salary, and internship length. Call the Employee constructor to set the name and salary rather than assigning to the protected fields directly. Override the

virtual Print method to chain to the base class version first and then print out the derived class fields.

```
5.    using System;
6.
7.    namespace Interfaces
8.    {
9.            abstract class Employee
10.           {
11.                   protected string name;
12.                   protected double salary;
13.
14.                   protected Employee(string name, double salary)
15.                   {
16.                           this.name   = name;
17.                           this.salary = salary;
18.                   }
19.
20.                   public virtual void Print()
21.                   {
22.                           Console.WriteLine("Name  : {0}", name);
23.                           Console.WriteLine("Salary: {0}", salary);
24.                   }
25.           }
26.
27.           class Programmer : Employee
28.           {
29.                   protected double averageOT;
30.
31.                   public Programmer(string name, double salary, double
    averageOT)
32.                           :base(name, salary)
33.                   {
34.                           this.averageOT = averageOT;
35.                   }
36.
37.                   public override void Print()
38.                   {
39.                           Console.WriteLine("Programmer");
40.                           base.Print();
41.                           Console.WriteLine("Average OT: {0}",
    averageOT);
42.                   }
43.           }
44.
45.           class Manager : Employee
46.           {
47.                   protected string secretaryName;
48.
49.                   public Manager(string name, double salary, string
    secretaryName)
50.                           : base(name, salary)
51.                   {
```

```
52.                                 this.secretaryName = secretaryName;
53.                         }
54.
55.                         public override void Print()
56.                         {
57.                                 Console.WriteLine("Manager");
58.                                 base.Print();
59.                                 Console.WriteLine("Secretary: {0}",
    secretaryName);
60.                         }
61.             }
62.
63.             class Intern : Employee
64.             {
65.                         protected int lengthOfInternship;
66.
67.                         public Intern(string name, double salary, int
    lengthOfInternship)
68.                                 :base(name, salary)
69.                         {
70.                                 this.lengthOfInternship = lengthOfInternship;
71.                         }
72.
73.                         public override void Print()
74.                         {
75.                                 Console.WriteLine("Intern");
76.                                 base.Print();
77.                                 Console.WriteLine("Length of
    internship(months): {0}", lengthOfInternship);
78.                         }
79.             }
    }
```

Part 2- Interface

To promote our employees we will write an interface containing a Promote method. Programmers and Managers will implement the interface. We will then write some generic code to work with a group of employees and promote only those that are eligible.

Steps:

1. Code an interface named IPromoteable containing a Promote method. Promote should take no arguments and return void.
2. Implement the IPromoteable interface in both the Programmer and Manager classes. Programmers get a 10% raise when promoted and Managers a 50% raise.

3. Create a driver class InterfaceTest containing a Main method. Make an array of Employee references and fill the array with a mix of Programmers, Managers, and Interns. Loop through the array and call Promote on those objects that implement the IPromoteable interface. Recall that you can use the is operator to test if an object implements a particular interface. Note that you will not be able to call Promote through an Employee reference so casting will be required. You may want to use the Print method to display the contents of the array before and after the promotions.

Part 3- Ambiguity (optional)

Ambiguity will result when a class attempts to implement two interfaces containing methods with exactly the same signature. The class has two options: implement one method to fill both roles or use explicit implementation to provide both versions. Here we will examine a case which requires explicit implementation.

Suppose our company embarks on an initiative to improve the skills of the managers by requiring them all to take continuing education courses. As the managers move through the courses they get promoted from one level to the next. Notice that the term Promote is now used to mean advancement of position within the company and advancement through the course levels at school.

Steps:

1. Write an interface that defines a promote operation for students.

```
interface IGoodStudent
{
        void Promote();
}
```

5. Use explicit method implementation to allow the Manager class to supply both IPromoteable.Promote and IGoodStudent.Promote. In the IGoodStudent version simply print a message commending the manager on being such a good student. Remember that no access modifier is allowed on explicit method implementations.
6. Modify your test code to examine each employee in the array and call the IGoodStudent version of Promote as appropriate. Remember that you will need a IGoodStudent reference in order to

call the method. To obtain an IGoodStudent reference, either a separate variable or a cast can be used as shown in the code below.

```
Manager m = new Manager();
...
IGoodStudent gs = m; // separate variable
gs.Promote();
...
((IGoodStudent)m).Promote(); // cast
```

11-Exceptions

Goals:

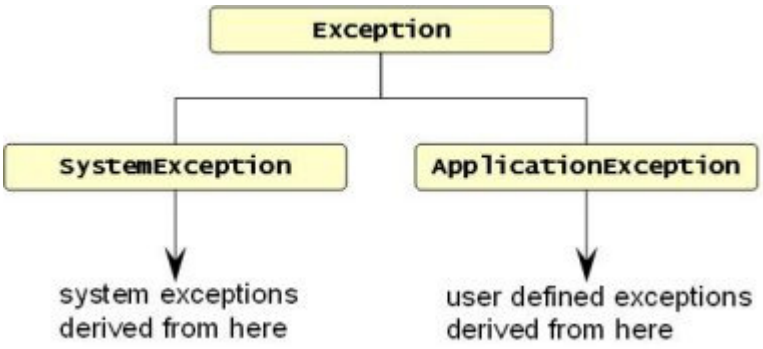
- Discuss C# exception handling
- Cover keywords `try`, `catch`, `throw`, and `finally`.
- Introduce a few of the .NET Framework Library exception types
- Show how to create a custom exception.

Overview

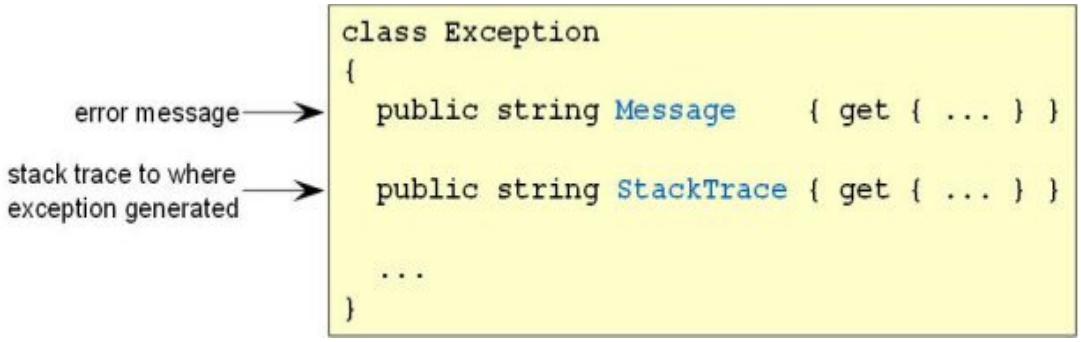
Traditional error notification involves methods reporting success or failure through a return value, a global variable, or by invoking an error handler. Exceptions are an alternative to these traditional techniques. The main benefit of using exceptions is that it separates the point of error detection from the handler code in a very clean way.

Exception hierarchy

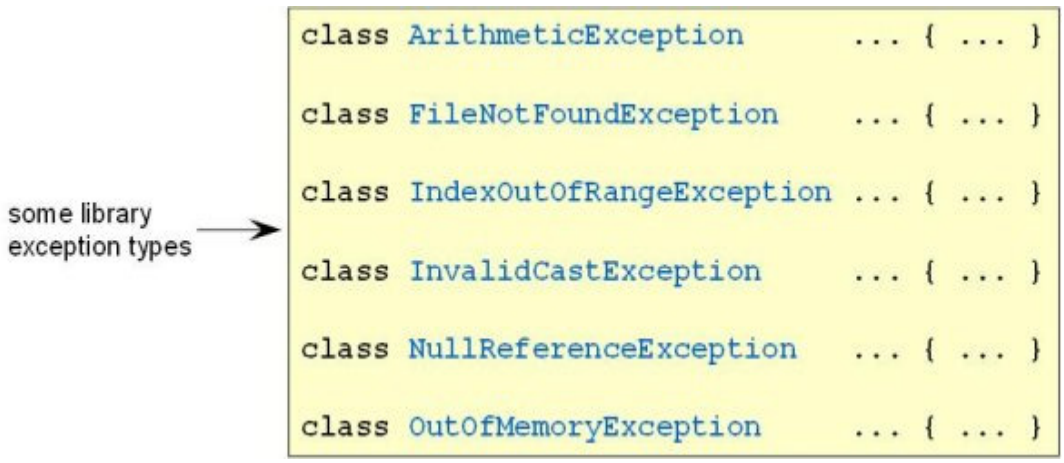
Exceptions must be classes derived from the library class `Exception`. There are two broad subcategories, one for exceptions generated by the CLR and the .NET Framework (the system exceptions) and one for user defined exceptions (the application exceptions).



The `Exception` class offers some basic services. The two most generally useful features are the read-only properties `Message` and `StackTrace`. The error message is set by the code that creates the exception while the stack trace is filled in automatically. All exception types inherit the services of the `Exception` class, plus they are free to add more information specific to the type of condition they represent.

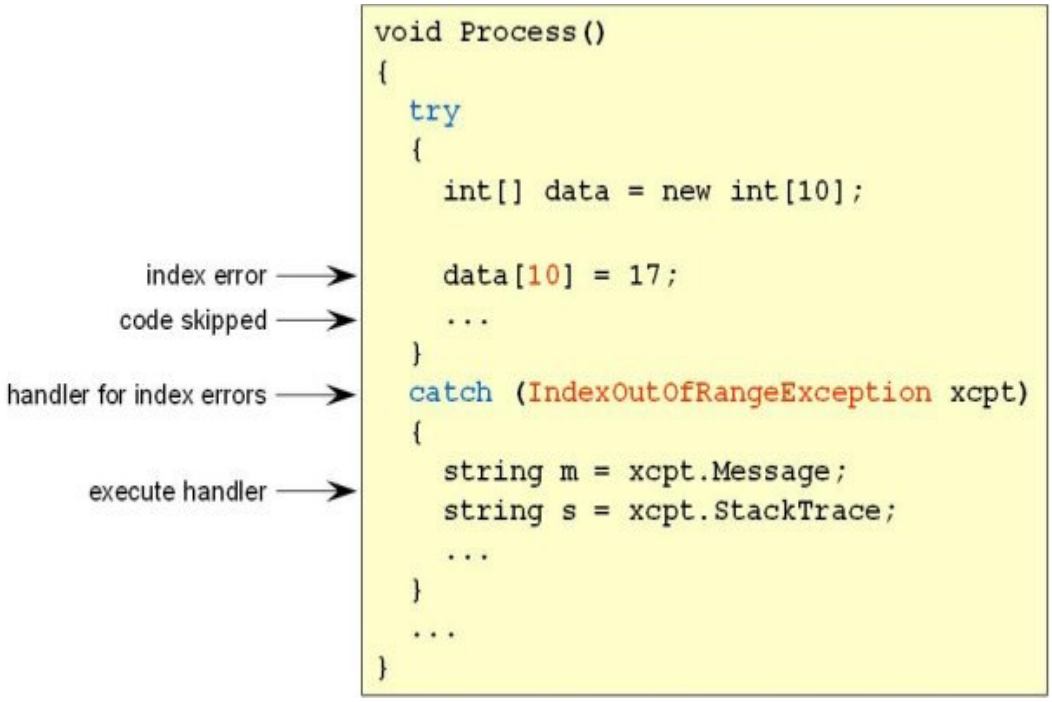


The .NET Framework class library defines many exception types for common error conditions. These exceptions are typically used by the CLR or library classes to report errors back to the application code. By convention, the class names all end in `Exception`.



Handling an exception

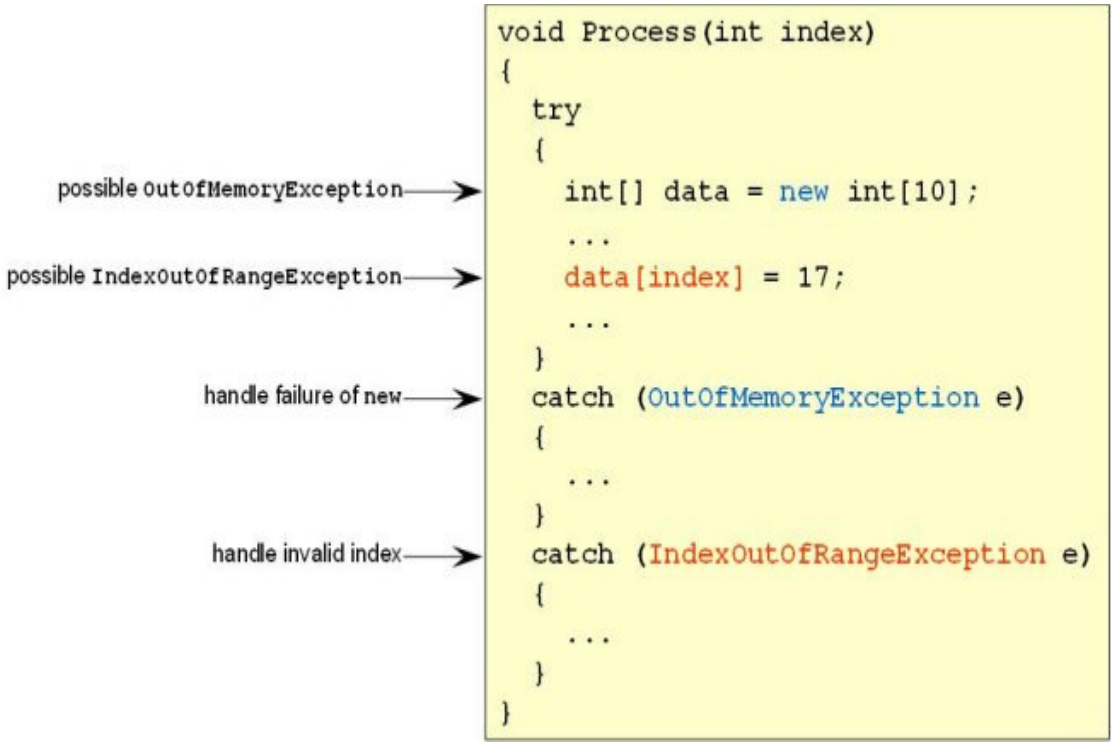
The CLR and .NET Framework Class Library throw exceptions to indicate errors. For example, the CLR will throw an `IndexOutOfRangeException` when an invalid array index is used. To handle an exception, the client must place their code inside a `try/catch` construct. The real work goes inside the `try` block while the error handling code is placed in the associated `catch` block. The control flow through a `try/catch` is fairly straightforward. When an exception is thrown, the normal control flow is stopped and the remainder of the `try` block will be skipped. Control jumps to the `catch` block and the statements in the block are executed. When the execution of the `catch` block has finished, linear control flow resumes from the line of code after the `catch` block. There is no way to automatically return to where the exception occurred and continue from that point.



A few more details about what goes on might be of interest. When the CLR or library code throws an exception, they actually create an object of the appropriate type and stuff it full of interesting information about what went wrong. In the example above, the CLR would instantiate an `IndexOutOfRangeException` object. A reference to this object is passed to the handler and appears in what looks like a method parameter. Glance back at the `catch` block shown above and notice the thing named `xcpt`. That is a reference to the exception object that was passed to the handler and is used to access the information inside the exception object. There is nothing special about the name `xcpt`, the naming of the reference is completely up to the programmer who codes the `catch` block.

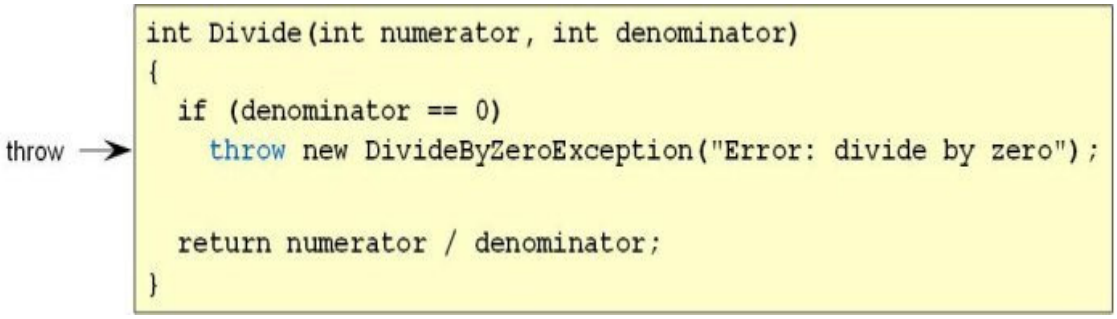
Multiple catch

A `try` block may have multiple associated `catch` blocks. Each catch block includes a parameter that specifies the type of the exception that it is designed to handle. `catch` blocks are always chosen based on the type of the exception, so if an `IndexOutOfRangeException` is active, then the catch block for that type will be executed. Only one `catch` block will be executed from the set. After a `catch` has been run, the others will be skipped and linear control flow will resume past the end of the entire `try/catch` construct.



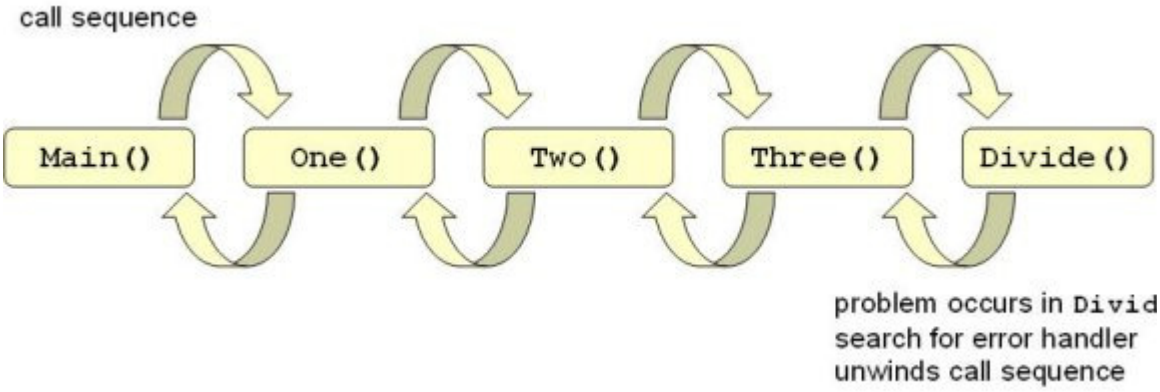
Generating an exception

The keyword `throw` is used to raise an exception. The object that is thrown must be of a derived type of the library `Exception` class. In the following `Divide` method, we generate a `DivideByZeroException` if the denominator passed to the method is zero. Notice how a string is passed as the constructor argument when the exception object is created. This string will become the error message that can be retrieved by clients using the `Message` property. When a `throw` statement is executed, normal control flow is halted and the search for a matching handler begins.



Locating a handler

Suppose our program begins execution in `Main`. The `Main` method then calls a method named `One`. `One` calls a method named `Two`. `Two` calls a method named `Three`. `Three` calls `Divide` and passes zero for the denominator argument. The `Divide` method will throw a `DivideByZeroException` and the search for a handler will begin. This situation is summarized in the figure below.

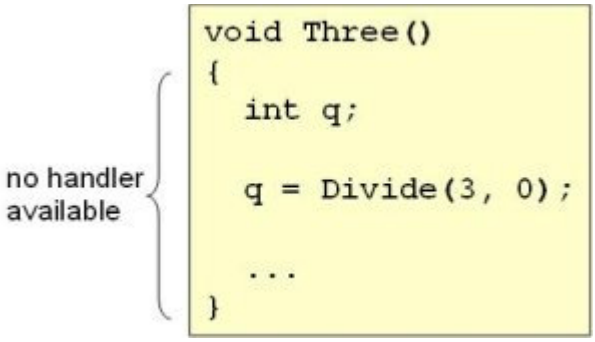


The search for a handler will begin within the `Divide` method itself. The code for `Divide` is shown below. Notice that there are no `try/catch` constructs so no handler is available. Since no handler was found, the `Divide` method will be exited and all the memory used for its execution will be reclaimed (parameters, local variables, etc.). The search for a handler will continue one level up the call chain in the method `Three`.

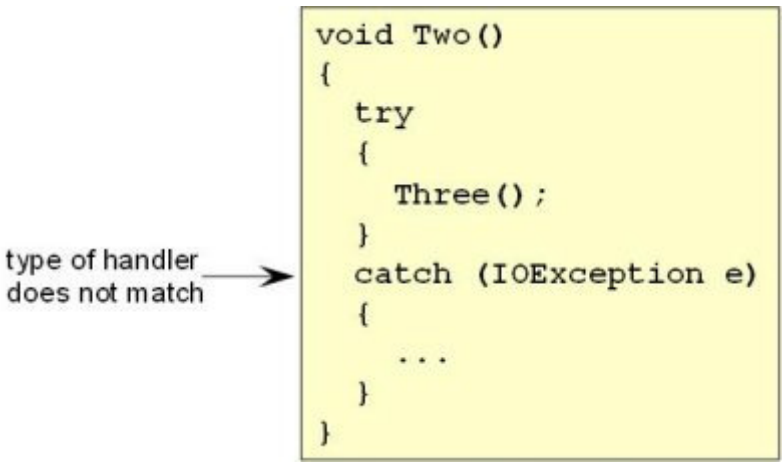
no handler available

```
int Divide(int numerator, int denominator)
{
  if (denominator == 0)
    throw new DivideByZeroException("Error: divide by zero");

  return numerator / denominator;
}
```

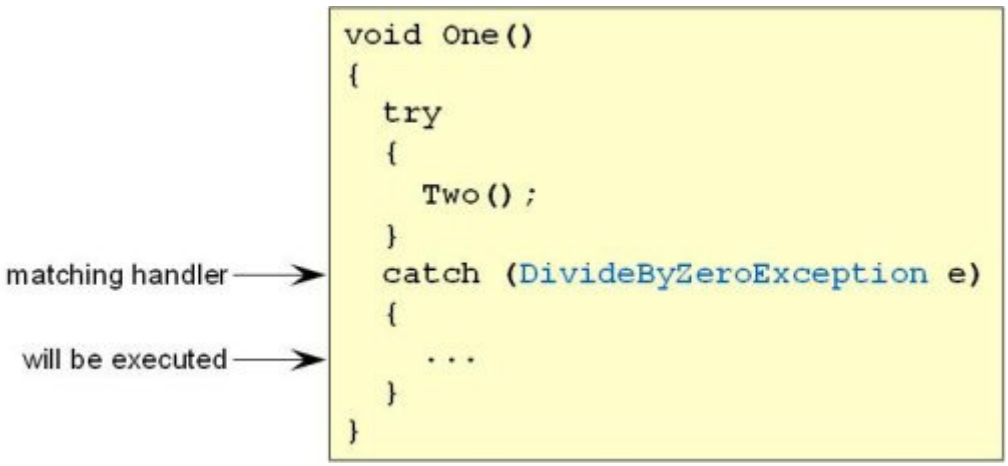
The story inside the method `Three` is similar to `Divide` in that there are no `try/catch` constructs so no handler is available. Once again the search must move up the call chain to the method `Two`.



`Two` looks more promising since it at least has a `try/catch` construct. However, the `catch` is not of the correct type so it is ignored and the search continues with the method `One`.



`One` contains a `try/catch` construct and has a `catch` of the correct type. The matching handler is executed and the exception is finished.

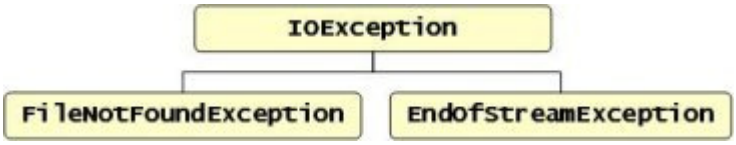


If the search had unwound the call chain all the way to the top without finding a matching handler, the program would have been terminated. Many systems have a default handler

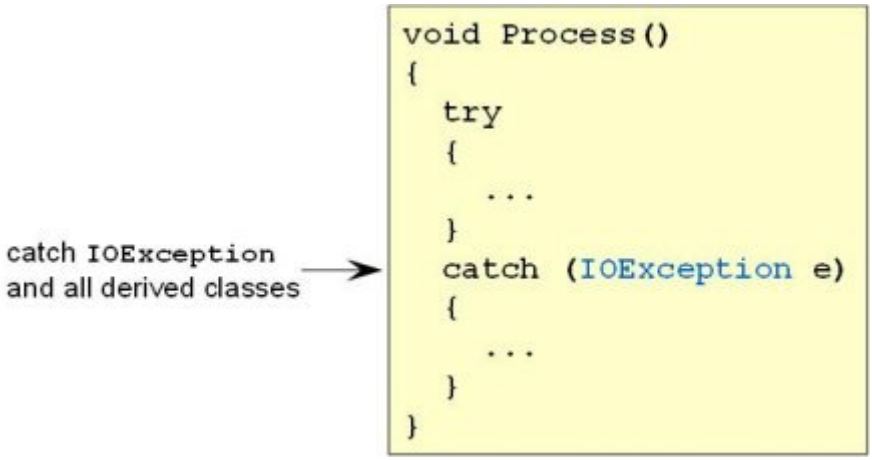
at the top level that will print out some of the information in the exception such as the message and/or the stack trace.

Catching base class

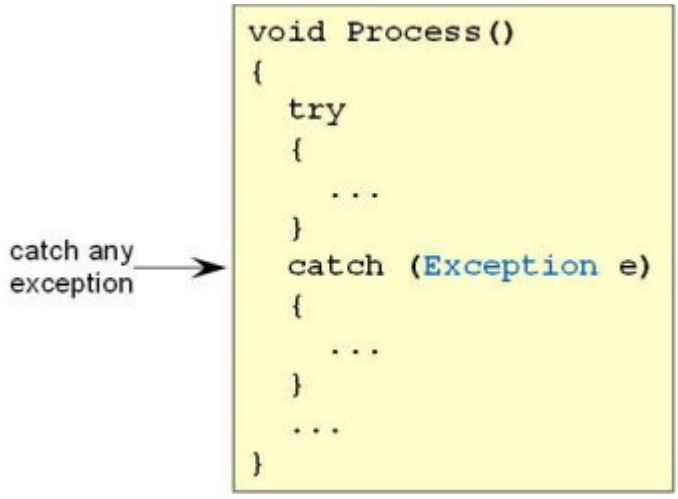
Exception classes are typically organized into inheritance hierarchies. For example, a portion of the standard exception hierarchy is shown below with the base class `IOException` and two of its derived types `FileNotFoundException` and `EndOfStreamException`.



When a `catch` clause specifies a base class, the handler will match that base class and all its derived classes. This is another example of the is-a relationship in action.

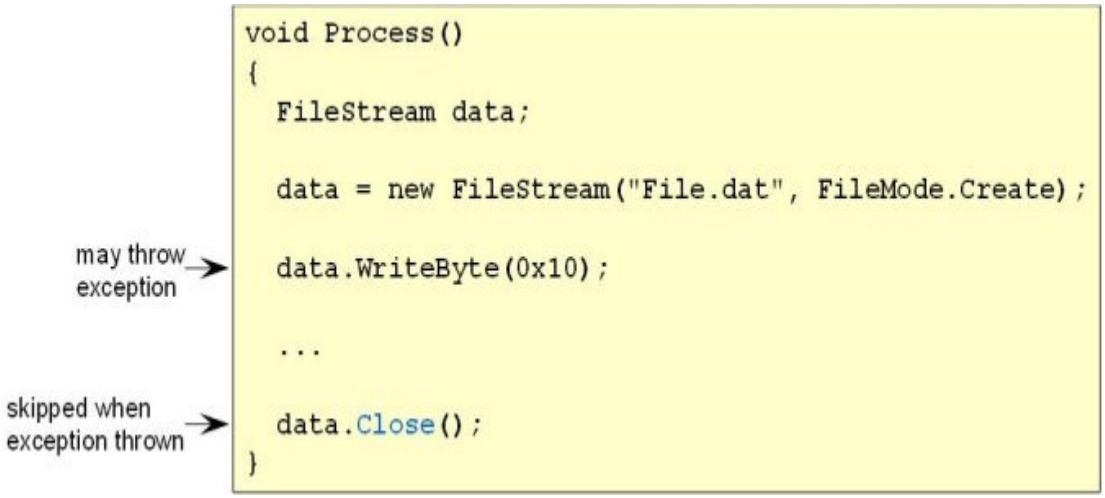


A `catch` block can be made generic by specifying an `Exception` reference as the type in the handler. Since `Exception` in the root of the exception hierarchy this will catch all exceptions.



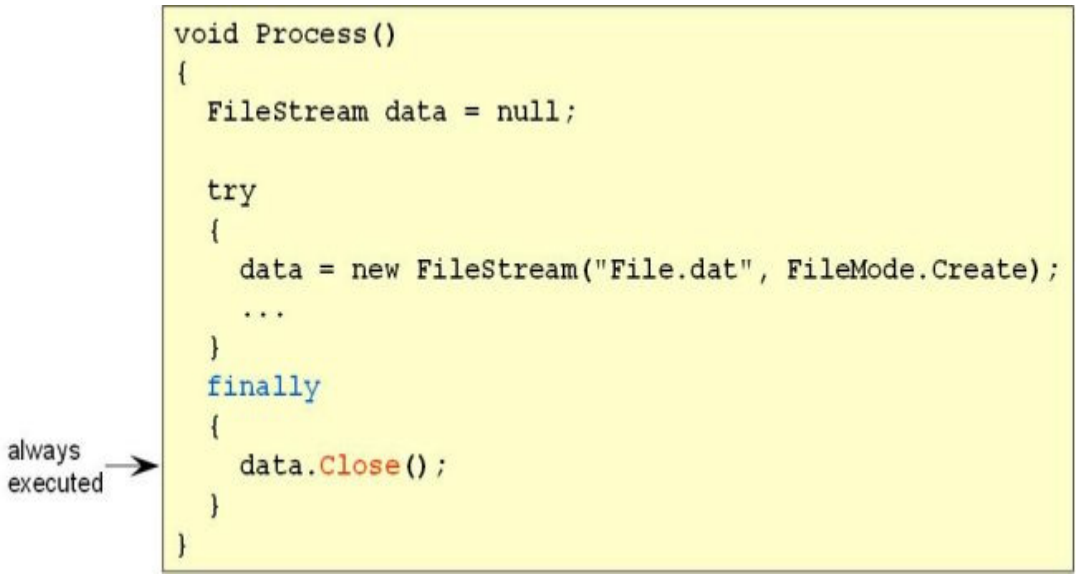
Finally

Some resources require cleanup. The classic example is a disk file where the usage pattern is the familiar open/access/close. It is sometimes difficult to use this type of class in the presence of exceptions since an exception being thrown could cause the close operation to be skipped. This situation is shown in the example code below which makes use of the .NET Framework Library class `FileStream`. Notice that there is no handler in the method so any exception that is generated will immediately leave the method in search of a handler. When the exception leaves the method, the rest of the code will be skipped and the close method of the file will not be called.



The keyword `finally` is used to create a block of code that gets executed regardless of whether or not an exception is generated. This makes it perfect for any needed cleanup code such as closing disk files. A `finally` block is placed at the end of the `try/catch` construct. The classic setup is a `try` block, several `catch` blocks, and then a `finally`

block. It is also possible to have a `try/finally` construct without any `catch` blocks. The sample code below demonstrates the syntax.



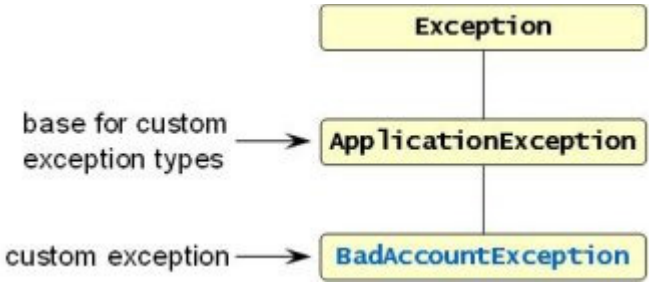

```
void Process()
{
  FileStream data = null;

  try
  {
    data = new FileStream("File.dat", FileMode.Create);
    ...
  }
  finally
  {
    data.Close();
  }
}
```

The control flow through a `try/catch/finally` construct can get a little tricky since there are a surprising number of cases to consider. The main thing to keep in mind though, is that the code in the finally block is always execute when control leaves the `try/catch` above it.

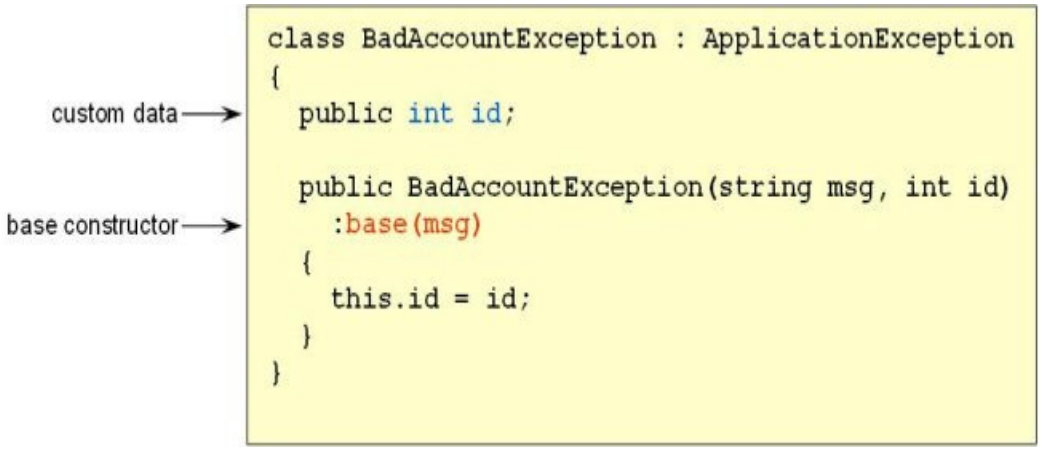
To avoid getting bogged down in too much detail, let's just examine two cases based on the `try/finally` code above. First case: the `try` block completes normally without throwing an exception. In this case, after control leaves the `try` block, it jumps down and executes the `finally`. After executing the `finally` block, normal linear control flow is resumed with whatever code follows the `finally` block. Second case: part way through the `try` block an exception is thrown. In this case, control skips the rest of the code in the `try` block and jumps down to the `finally` block. After finishing the `finally` block control will leave the method in search of an handler for the exception. There are a number of other cases, but they should be relatively simple to construct and test if needed.

Custom exception

To create a custom exception type, simply write a class derived from the library class `ApplicationException`. Convention dictates that the class name should end in `Exception`.



By convention, a custom exception should offer a constructor that takes a `string` which represents the error message. The string can be passed to the base class constructor for storage by the `Exception` class. The custom exception type can also contain any fields needed to store addition information that might be of interest to the client code that will handle the exception.



11- Exceptions Exercise

Goals:

- Practice with basic exception syntax.
- Catch exceptions generated by the CLR.
- Use a finally block to do cleanup.
- Examine the services offered by the Exception class.
- Define a custom exception type.
- Observe the effect of an uncaught exception.

Overview

Exceptions are an error notification mechanism. An exception is generated at the point an error is detected and caught and handled at a higher level.

All exception types descend from the common base class Exception defined in the .NET Framework Class Library. There are two interesting derived classes of Exception: SystemException and ApplicationException. The CLR and the library define many exceptions derived from SystemException which they use for common error conditions. Users can define custom exception types by deriving from ApplicationException. In this exercise, we will begin by using SystemException types to practice with the basics mechanics of try, catch, throw, and finally. After that we will define and use a custom exception type.

Part 1- Basic try-catch

The CLR generates exceptions for many common error conditions such as an invalid array index, insufficient memory, use of a null reference, integer divide by zero, etc. In this first lab, we will practice catching an exception generated by the CLR.

Steps:

1. Create a class called ExceptionTest and add a Main method. Allocate an array of 10 integers and attempt to assign a value into the array using an invalid index. Place the array access in a try block

and place a catch block for an IndexOutOfRangeException after the try. Put a print statement in the catch block so you can verify that the handler code is being executed. Compile and run your program and observe the behavior.

Part 2- Exception class

The root of the exception hierarchy is the class Exception. The Exception class offers several services to its clients. Here we examine two of the most useful: error message and stack trace. When the CLR creates an exception it will set the error message as appropriate. The stack trace will be filled in by the runtime when the exception is thrown.

Steps:

1. Generate and catch an IndexOutOfRangeException. In the catch block, print out the message and stack trace from the exception object using the Message and StackTrace properties. Both the message and stack trace have type string. Compile and run your program and examine the output.
2. The exception class also provides a ToString method. Pass the exception object to Console.WriteLine and observe the results.

Part 3- Multiple catch

The code in a try block may generate more than one type of exception. To facilitate handling these different conditions, a try block can have multiple catch blocks.

Steps:

1. In your main program, create a try block that may generate two different types of exceptions. For the first exception type, read an array index from the user and apply it to an array. This may generate an IndexOutOfRangeException. For the second exception type, read two integers from the user and divide them. This may generate an DivideByZeroException if the denominator is zero.
2. Place two catch blocks after the try, one for IndexOutOfRangeException and one for DivideByZeroException.

3. Run the program several times and enter values that force both exceptions to occur.

Part 4- General catch

The exception classes are all organized into an inheritance hierarchy. We have already seen one advantage of this: common services such as an error message can be provided by the base class and will then be available to all derived types. Another advantage is that a catch block specifying a base class matches that class and all derived classes. This makes it easy to catch all exceptions or specific categories of exceptions.

Steps:

1. Create a try block containing code that may generate either an IndexOutOfRangeException or a DivideByZeroException.
2. After the try block, place a single catch for type Exception. Since Exception is the root of the entire hierarchy, this catch will catch all exceptions. Print out the Message and StackTrace. Run the program several times and enter values that force both exceptions to occur. Verify that the single catch works for both exception types.
3. Modify the catch to use the "general catch clause" - i.e. a catch which does not specify any type to catch. This is a convenient shorthand way to catch all exceptions; however, there is no reference to the exception object so you can not examine the message or the stack trace. Run the program several times and enter values that force both exceptions to occur. Verify that the general catch clause is executed for both exception types.

```
try
{
  ...
}
catch // general catch clause
{
  ...
}
```

Part 5- Finally

A try block can have an optional finally block. The code in the finally block is always executed, regardless of how control leaves the try block.

One of the main uses of a finally block is to release resources. For example, a finally block is often used to close a disk file, close a database connection, close a network connection, release a lock, etc. The call to the Close or Release method is placed in a finally block to ensure it gets called even if the method terminates early by a premature return or by throwing an exception.

To make this lab realistic we need to use a resource that requires cleanup. We will use a disk file from the .NET Framework Class Library. The disk file class offers a fairly standard protocol for use: open, access, close. We will place the call to the close method in a finally block to ensure that it always gets called.

Steps:

1. The StreamWriter class from the System.IO namespace provides a convenient way to write text files. The constructor takes the name of the disk file as a string argument. Data is written to the file using the standard set of WriteLine methods. The file is closed with the Close method. The following code contains a method that opens a disk file, writes a string to the file, and closes the file. Recode the Write method so it has a try block, a catch for the IOException that might be thrown by the WriteLine call, and a finally block containing the call to Close. Place a print statement in the finally block so you can observe it being called. Run the program.

```
class ExceptionTest
{
        static void Write(string filename, string message)
        {
                StreamWriter sw = new StreamWriter(filename);

                sw.WriteLine(message);

                sw.Close();
        }

        static void Main()
        {
                Write("data.txt", "Good Morning.");
        }
```

```
}
```

Part 6- Custom exceptions (optional)

Programmers can define custom exceptions by deriving a class from System.ApplicationException. Convention dictates that the name of the new class end in "Exception". It is also common to provide a constructor that takes a string and chains to the base class constructor to set the error message property inherited from Exception. In addition, programmers can add any fields or methods that are appropriate.

Steps:

1. In this first step, we set up a bank account class that we will use to demonstrate custom exceptions. The bank account stores an int account number and a double for the account balance. The constructor sets the account number and the initial balance. A Withdraw method reduces the balance by the specified amount. Feel free to code the class yourself or take advantage of the sample code provided.

```
2.  using System;
3.
4.  namespace Exceptions
5.  {
6.          class SavingsAccount
7.          {
8.                  private int    accountNumber;
9.                  private double balance;
10.
11.                 public SavingsAccount(int accountNumber, double
    balance)
12.                 {
13.                         this.accountNumber = accountNumber;
14.                         this.balance       = balance;
15.                 }
16.
17.                 public void Withdraw(double amount)
18.                 {
19.                         balance -= amount;
20.                 }
21.         }
22.
23.         class SavingsAccountTest
24.         {
25.                 static void Main()
26.                 {
```

```
27.                             SavingsAccount account = new
   SavingsAccount(12345, 2000);
28.                             account.Withdraw(500);
29.                     }
30.             }
   }
```

31. Create a custom exception type called OverdrawnException that can be used to indicate when the account is overdrawn. The class should derive from System.ApplicationException. Provide a constructor that takes a string error message and a double for the account balance. Have the constructor chain to the base class constructor to store the message. Provide a public field to store the balance.
32. Add error checking to the Withdraw method. If the balance goes negative after the withdrawal, throw an OverdrawnException. Modify the main program to place the call to Withdraw in a try block and include a catch for the potential exception.

Part 7- Uncaught exception (optional)

A program might fail to catch an exception that is generated at runtime. When an exception is not caught and propagates all the way back to the beginning of the program, the language specification says the program will be terminated. However, it goes on to say "The impact of such termination is implementation-defined." In practice, you will likely see a dialog box notifying you of the uncaught exception - but the exact behavior may vary depending on the version of the CLR you are running and the type of your application (console, web, windows, etc.).

Two other points may be of interest. First, if your program is multithreaded and a thread throws an exception which is not caught, then only that thread is terminated, not the entire program. Second, the library class AppDomain allows programs to register to be notified when an unhandled exception occurs.

In this lab we will force an exception to occur and observe the runtime behavior when we fail to catch the exception.

Steps:

1. Allocate an array of 10 integers and attempt to assign a value into the array using an invalid index. Compile and run your program and observe the behavior. If you are using Visual Studio .NET, it might be interesting to run the application both inside and outside the debugger to see if the behavior is different.

12-Namespaces

Goals:

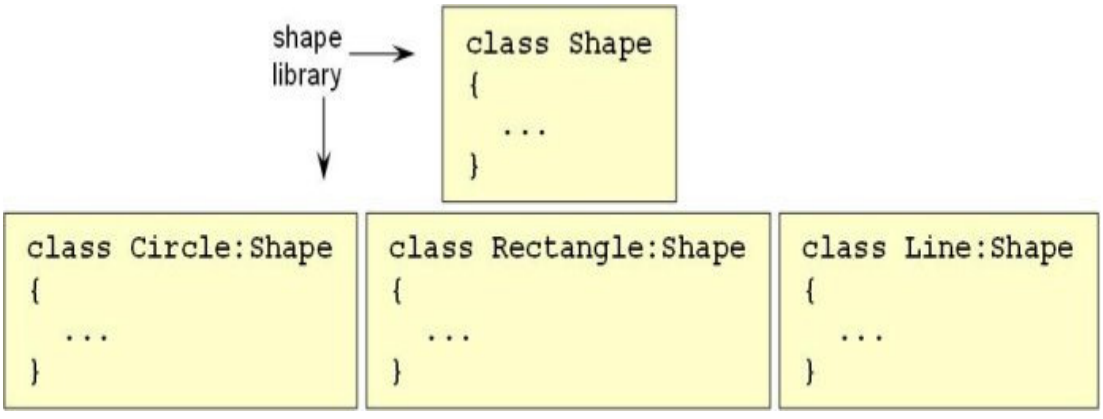
- Show how to create a namespace.
- Show how to access a type defined within a namespace.
- Discuss the advantages of organizing code into namespaces.

Overview

A namespace provides a container for a group of types. Related types are placed inside the same namespace to produce a clean and logical organization of code. Namespaces help to avoid type name collisions by creating distinct scopes.

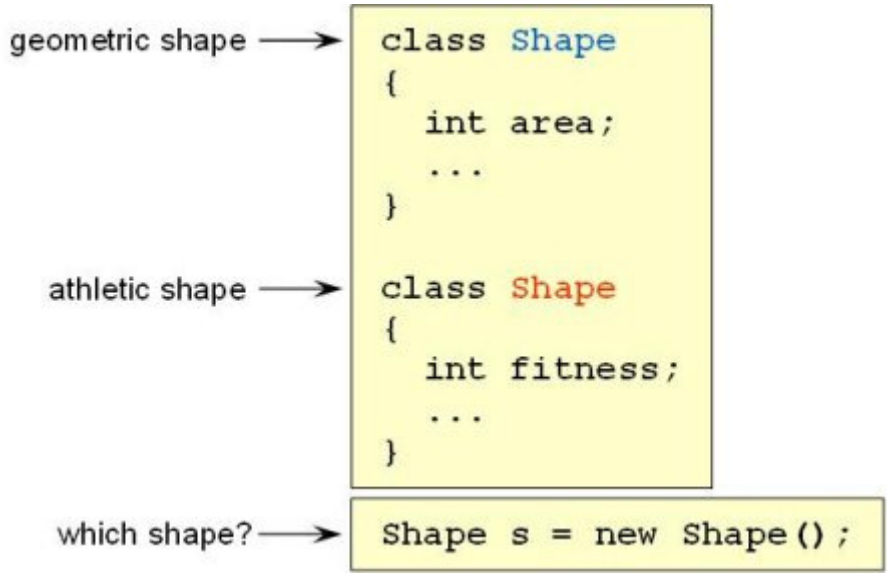
Motivation - logical grouping

It is common for a group of types to be logically related. For example, a model of people at a university would likely have types such as person, student, employee, undergraduate, etc. A drawing application would have types like shape, circle, rectangle, and line. Grouping the related types into a namespace cleanly partitions the source code. It also makes things more convenient for clients because it can greatly simplify their search for a needed class.

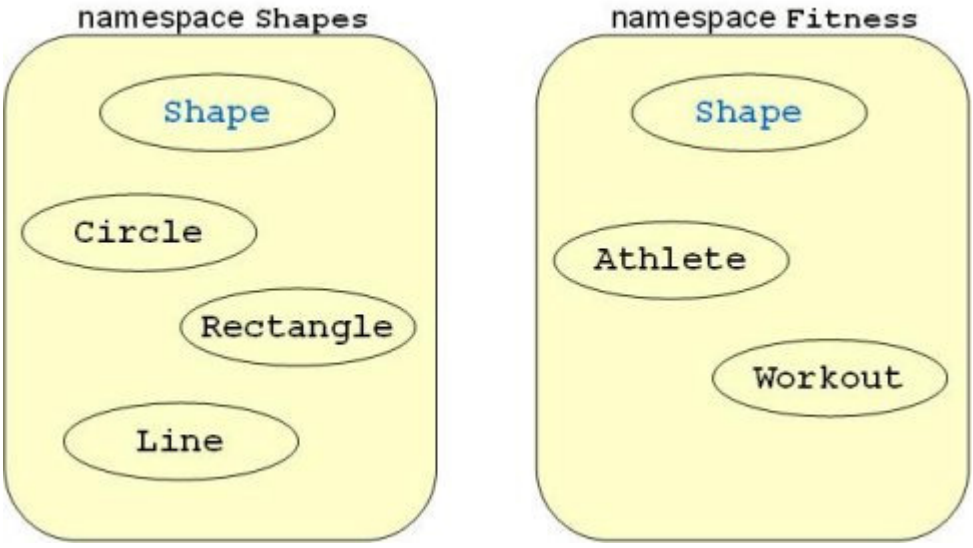


Motivation - name collision

It sometimes happens that two programmers use the same type name. The chance of this increases as the project size grows or when multiple libraries from third party vendors are utilized. Two types with the same name cannot exist in the same scope. The code below shows the problem. It will be a compile time error to use the two `Shape` classes in the same program.



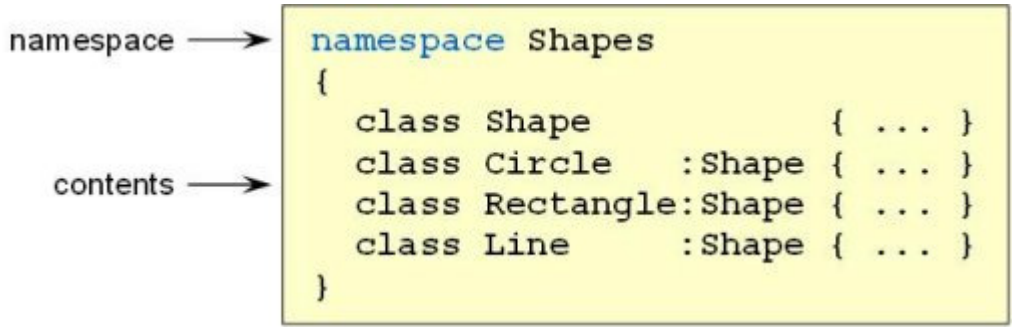
Placing the two `Shape` classes in different namespaces will resolve the issue.



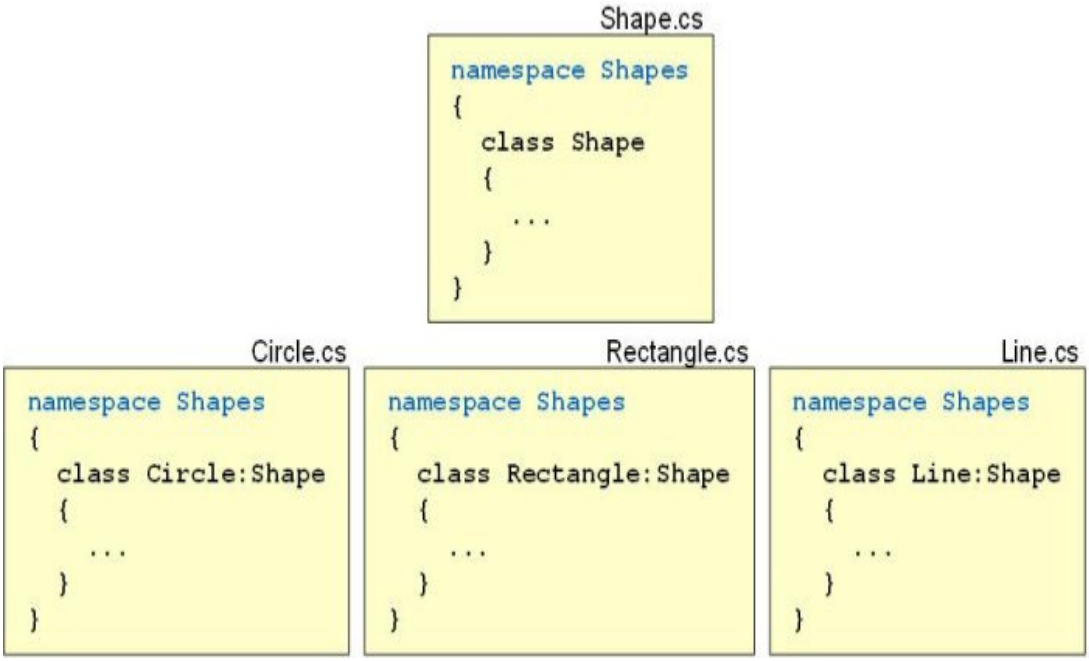
Creating a namespace

The most basic way to create a namespace is to use the keyword `namespace`, give the

name of the namespace, and then define the namespace body. The body contains a group of types inside curly braces. The figure below demonstrates the basic syntax by creating a `Shapes` namespace containing four classes.



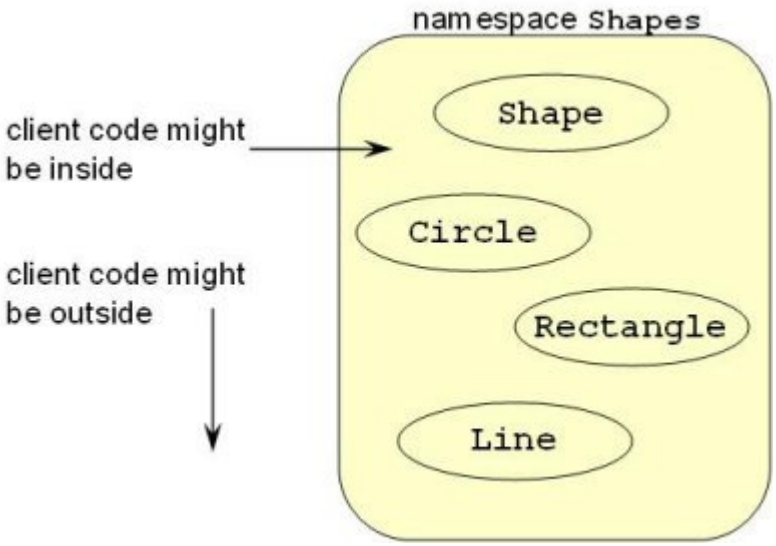
Defining an entire namespace in one place forces all the types to be defined in a single file which makes it difficult for multiple programmers to work on the code. It is more common to define the namespace in many separate parts and let the pieces be logically merged together into a single namespace by the compiler. The syntax is shown below. Notice how each class is defined in its own file and placed inside its own namespace declaration. This way of defining a namespace is logically equivalent to defining the entire namespace at once.



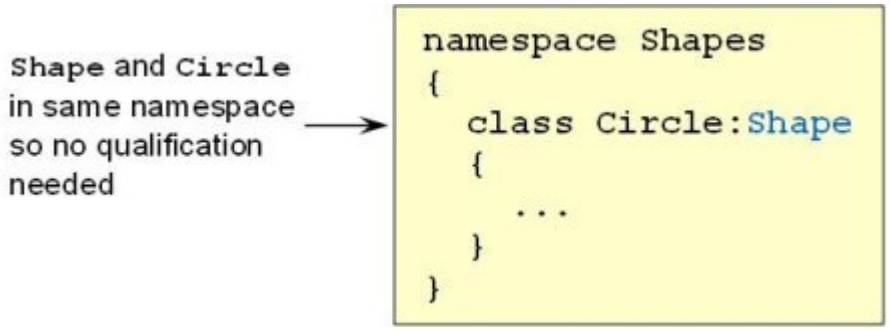
Access to namespace members

Accessing types defined inside a namespace requires just a little extra thought. There are

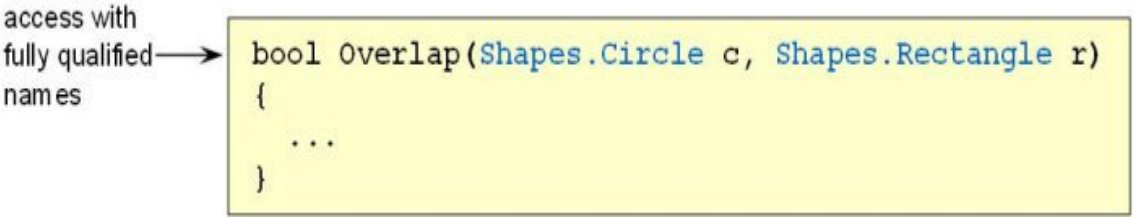
two cases to consider because the client code could either be inside the same namespace or outside. The following logical representation of the `Shapes` namespace summarizes the issue.



If the client code is inside the same namespace then there is no extra work to do and the short name of the class can be used.

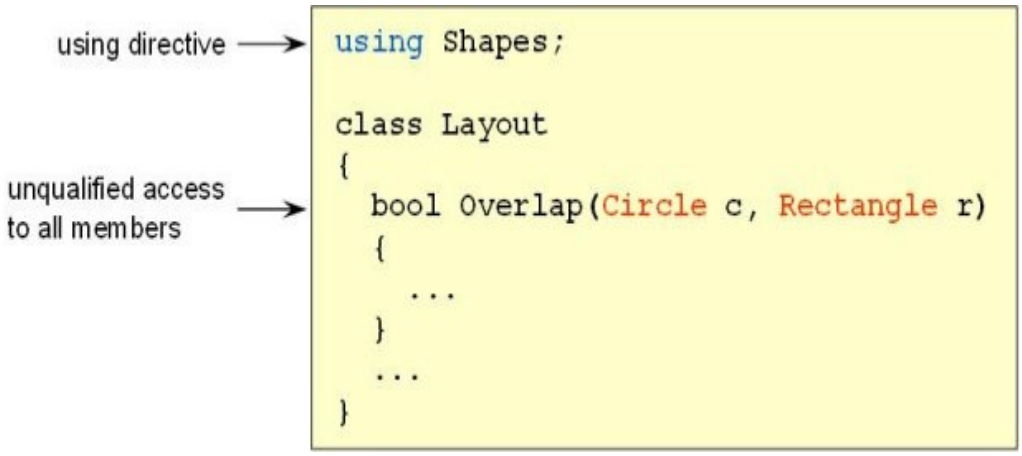


If the client code is outside the namespace, then the fully qualified name of the class is used. The fully qualified name is formed from the namespace name, the dot operator, and the short name of the class.



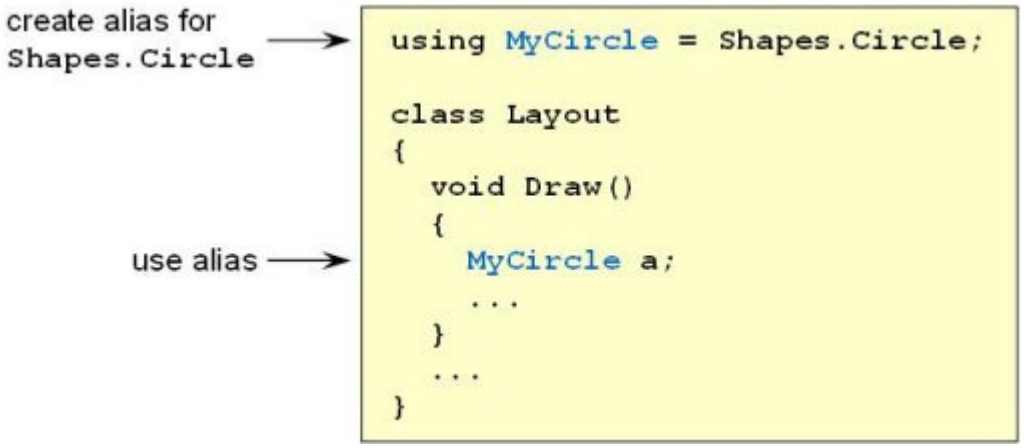
Typing fully qualified names quickly becomes tedious. A using directive eliminates the need for fully qualified names by bringing all the types from a namespace into the local

scope.

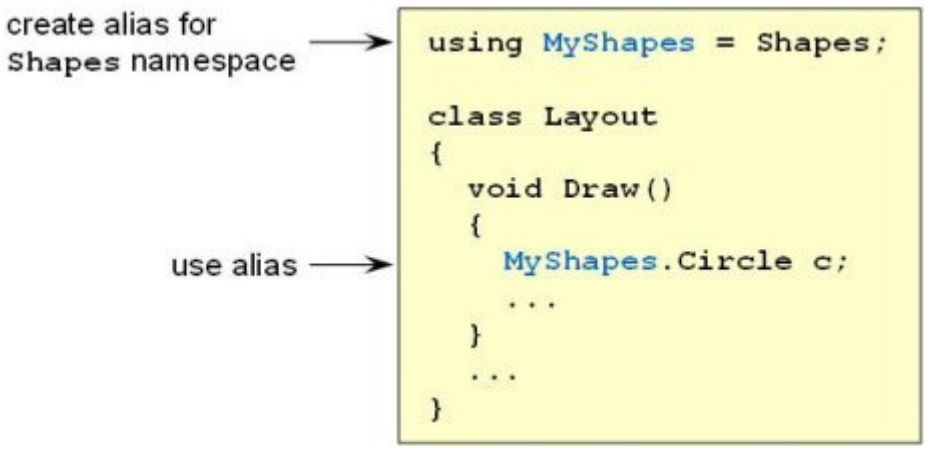


Using alias

The keyword `using` can also create an alias for either a class or a namespace. This technique is not common, but it is occasionally useful to resolve ambiguities or to provide a way to easily switch between different namespaces so it is worth a quick look. Let's first examine a using alias for a class. The syntax to create an alias is "`using newName = oldName;`". The new name can then be used in place of the old name.

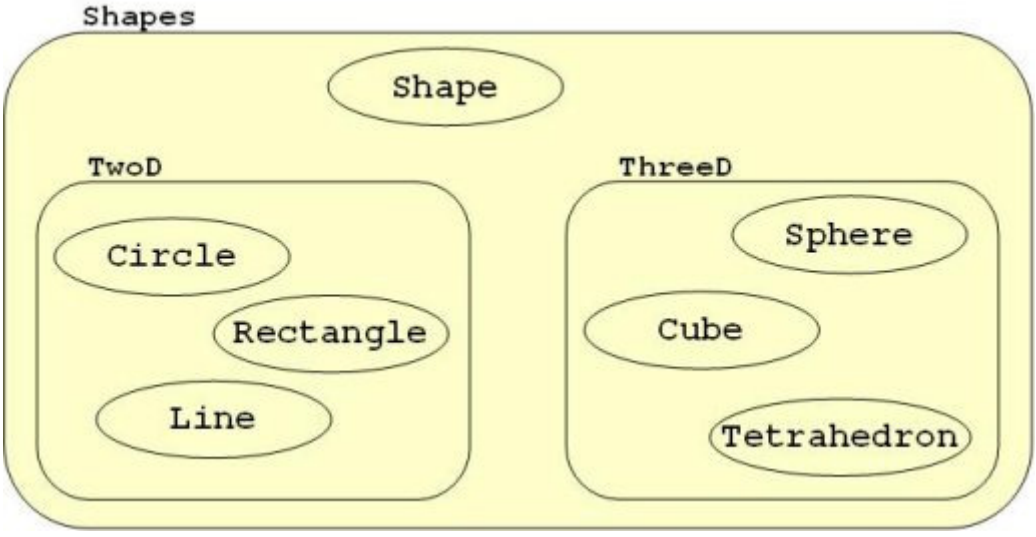


An alias for a namespace can be created with the syntax "`using newNamespace = oldNamespace;`". The new name can then be used as a namespace name. The compiler will translate the alias into the namespace name it really represents.



Nested namespace

Namespaces that are only one level deep are not generally sufficient to logically organize code. Consider the `Shapes` namespace. Should all geometric shapes be placed in the single namespace or might it be better to subcategorize the geometric types according to some other property such as whether they are a two or three dimensional shape? Furthermore, if two different programmers accidentally use the same namespace name then the names will collide. To better guarantee uniqueness and to more finely categorize their contents, namespaces can be nested.



The most basic syntax to create a nested namespace is to define one namespace inside another as shown below. This syntax is considered a bit verbose and is not used very often in practice.

nesting syntax →

```
namespace Shapes
{
  namespace TwoD
  {
    class Circle:Shape { ... }
  }
}
```

The preferred way to define a nested namespace is the shorthand syntax using the dot operator.

shorthand →

```
namespace Shapes.TwoD
{
  class Circle:Shape { ... }
}
```

The fully qualified name for a member of a nested namespace consists of each namespace name and the class name with each component separated by the dot operator. It is common to encounter five or six levels of namespace nesting so a using directive is often the better choice.

access with fully qualified names →

```
Shapes.TwoD.Circle   a;

Shapes.ThreeD.Sphere b;
```

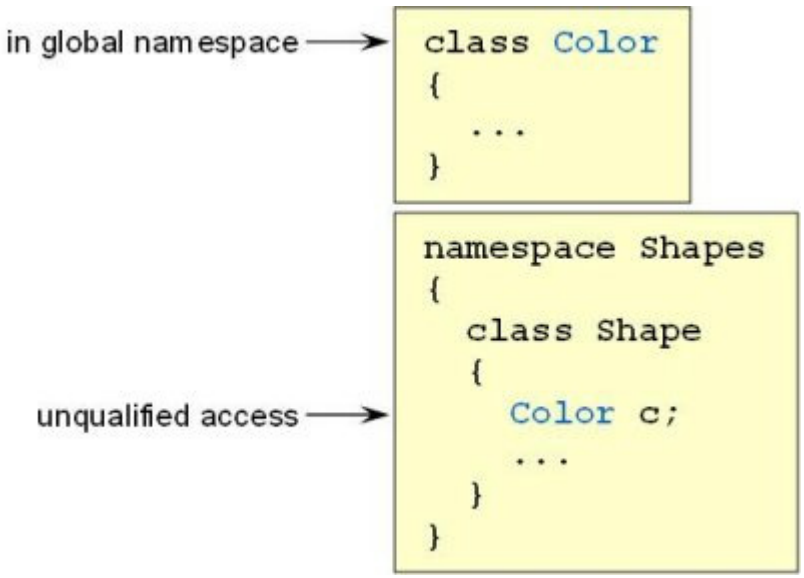
using directive →

unqualified access to all members →

```
using Shapes.TwoD;

class Layout
{
  bool Overlap(Circle c, Rectangle r)
  {
    ...
  }
  ...
}
```

Global namespace

Types that are not defined inside any namespace are placed in the "global namespace". Members of the global namespace are available for use in all other namespaces with no prefixing required. The common wisdom says to minimize your impact on the global namespace because each symbol you add to the global namespace increases the likelihood of a name collision.



12- Namespaces Exercise

Goals:

- Create top level and nested namespaces.
- Access namespace contents using fully qualified names.
- Employ a using directive to get shorthand access to an entire namespace.
- Create a using alias for a namespace.
- Create a using alias for a class.

Overview

A namespace defines a scope that can contain types and other namespaces. Partitioning code into namespaces allows related types to be grouped together into a logical unit. Grouping related classes makes life easier for users who need to navigate a large library since the types are sorted into categories. Namespaces also provide name management: two types can have the same name as long as they are in different namespaces.

To access a namespace member, user code can employ the fully qualified name. For example, to access a class named Circle in the Shapes namespace the code would be: Shapes.Circle. Typing fully qualified names quickly becomes tedious and can make code look cluttered. To help avoid the long names, programmers can employ the using keyword in several ways. First, a using directive gives short name access to the entire contents of a namespace. Second, a using alias can provide a more intuitive or concise name for a namespace. Third, a using alias can be created for a class within a namespace.

A namespace is defined with the keyword namespace and a set of curly braces. All types defined inside the curly braces become members of the namespace. The definition does not have to be contiguous; for example, namespaces with the same name defined in different files are conceptually merged together into a single namespace.

Namespaces can be nested as deeply as necessary. For example, a namespace for different types of geometric shapes could have a nested namespaces for 2D types and another one for 3D types.

Part 1- Defining namespaces

Here we create namespaces for a group of classes. The goal is to practice with the syntax for defining a namespace.

Steps:

1. Put the Photo and photo Album classes in a namespace called Photography.
2. Create a namespace called Music for music industry types. Create two nested namespaces: Works for musical works and Business for the business side of the industry. Add the music Album class to the Works namespace. Add the Agent to the Business namespace.

```
class Photo
{
        public string Title;
}

class Album
{
        public Photo[] Photos = new Photo[10];
}

class Album
{
        public string Artist;
        public string Title;
}

class Agent
{
        public string Name;
        public double Commission;
}
```

Part 2- Fully qualified names

Here we access the namespace members through their fully qualified names.

Steps:

1. Create a class called NamespaceTest in the global namespace. Add a Main method to the class.
2. Declare a Photo using only the short name. The compiler should issue an error saying that the type could not be found.
3. Declare a variable of each of the 4 classes with the fully qualified names. There is no need to actually create or manipulate objects for this exercise, declaring references and making sure the code compiles is sufficient.

Part 3- Using directive

Here we code a using directive to get shorthand access to the entire contents of a namespace.

Steps:

1. Place a using directive for the Photography namespace at the top of the file containing your test code.
2. Use the short names to declare a Photo and an Album. Test to make sure the code compiles.
3. Place a using directive for the Music.Works namespace at the top of the file containing your test code. We now have using directives for two namespaces which both contain an Album class. The two using directives do not cause a compilation error by themselves; however, attempting to use the short name to declare an Album will yield a compiler error since the compiler cannot know which Album class to use. Resolve the ambiguity with the fully qualified name for the class you would prefer and test that the code compiles.

Part 4- Using alias

A using alias can be used to create a more convenient name for either a namespace or a class. This is especially useful for deeply nested types. In addition, an alias for a class is often used a way to resolve ambiguity when multiple using directives are present.

Steps:

1. Create an alias named Works for the Music.Works namespace. Notice that we have reused the namespace name for the alias. Reuse of the name is not required and any name could have been used. Make use of the alias to declare an Album. Test to make sure the code compiles.
2. Create an alias named TheAgent for the Music.Business.Agent class. Make use of the alias to declare an agent. Test to make sure the code compiles.

13-Delegates and Events

Goals:

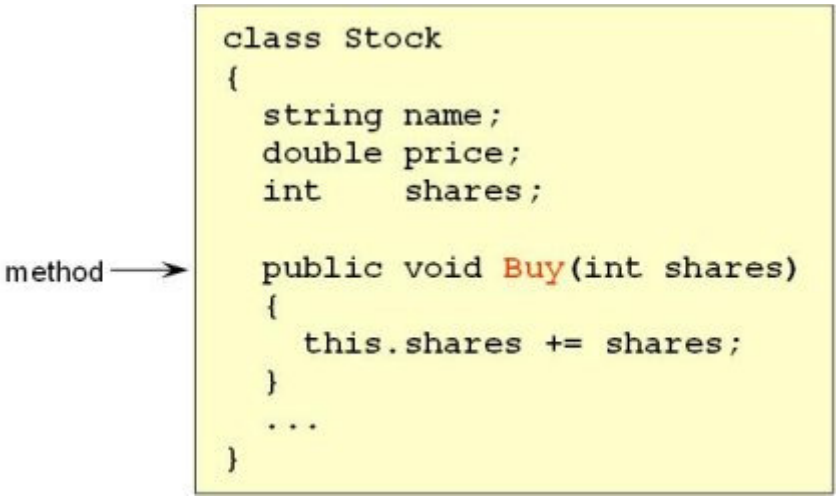
- Introduce the concept of a delegate.
- Show how to define a delegate type.
- Show how to register and unregister targets.
- Show how to handle multiple targets.
- Describe the effect of the `event` keyword.
- Show some common applications of events from the .NET Framework Class Library.

Overview

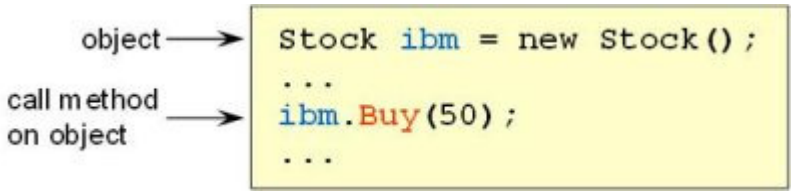
A delegate is a proxy for a method. Clients interact with the delegate and the delegate forwards the operation to the method it represents. The advantage of this indirect arrangement is that the client code is relatively independent of the target method. The client sees only the delegate and does not need to know much about the method the delegate represents. Delegates are used in many places throughout the .NET Framework Class Library. Classic examples are as callbacks in user interface programming and as the way to specify the work to be performed by a thread.

Background - object-oriented method call

Let's first do a very quick review of a really basic concept. In an object-oriented system there are two components to a method call: the object and the method. Consider the `Stock` class and its `Buy` method shown below.



We must have an object of the `Stock` class available in order to call the `Buy` method. The object will become the hidden parameter `this` inside the method.



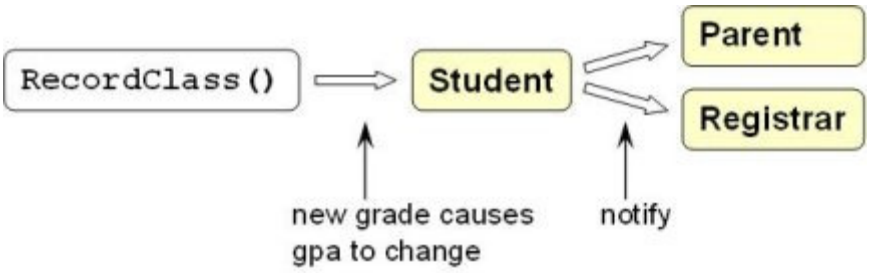
Notification

Objects typically store information in their fields. The formal way to say this is that the object maintains an internal "state". The state changes over time as user input is received or as client code calls methods. The `Student` class shown below models a student at a university and demonstrates the pattern. The student maintains a numerical rating of their scholarship in the form of a "gpa" (grade point average). Their gpa changes as they take courses and receive grades. Each time they finish a class, the `RecordClass` method will be called and their grade passed in (an A is worth 4 points, a B worth 3, and so on). A new value for the gpa will be calculated to take into account the newly received grade. In the example, we have made the simplifying assumption that each class is worth one unit.

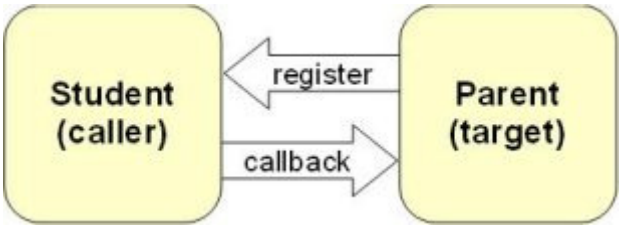
store state →
change state →

```
class Student
{
  string name;
  double gpa;
  int    units;

  public void RecordClass(int grade)
  {
    gpa = (gpa * units + grade) / (units + 1);

    units++;
  }
  ...
}
```

Various parties might be interested in tracking the student's performance. Certainly the student's parents come to mind since they might be paying the tuition fees and want to stay informed of their child's progress. Other likely candidates are the school registrar, the dean of the college, any honor society for which gpa is a criterion of membership, etc. The student will be responsible for notifying all interested parties when the gpa changes. The situation is summarized in the diagram below.

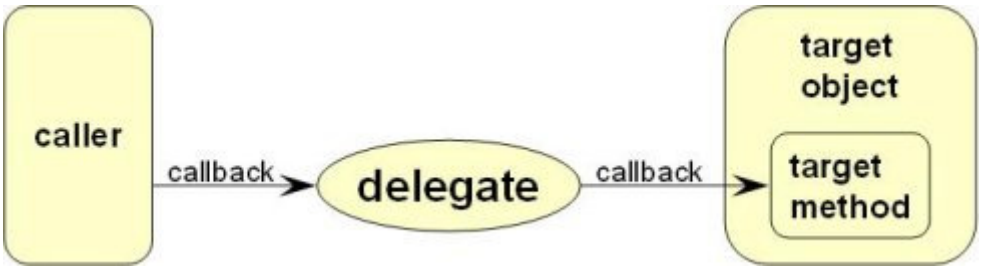


Notification typically involves "registration" and "callback". The targets that would like to receive notification must register with the caller. The caller will notify all registered clients whenever the state changes. Another name for this pattern is "publish/subscribe". In our example, the parents and the registrar would need to register with the student. The student would notify them at the end of each class when their gpa is updated. The idea is captured in the figure below.

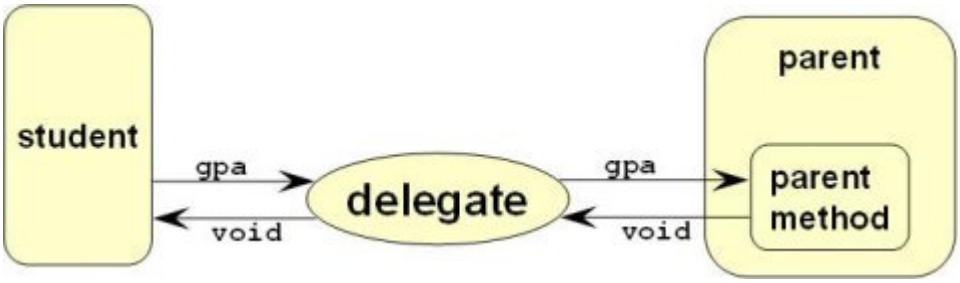


Delegate definition

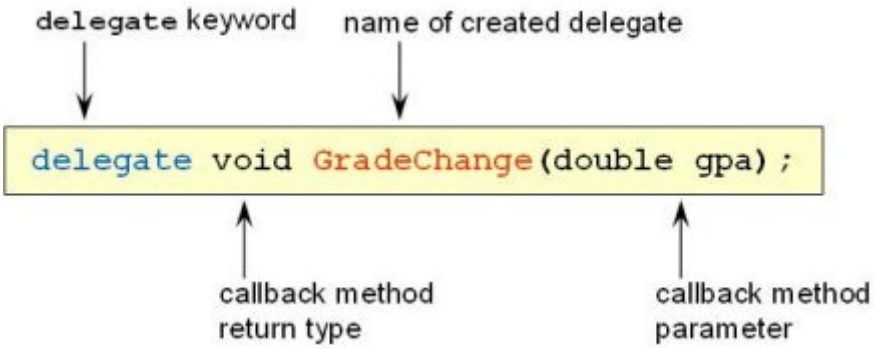
.NET uses delegates to implement callbacks. A delegate acts as an intermediary between the caller and the target. The target creates a delegate and registers it with the caller. The caller invokes the delegate which forwards the call to the target. The main advantage to this approach is that the caller and the target do not communicate directly so the caller never needs to know the type of the target or the method that it is calling.



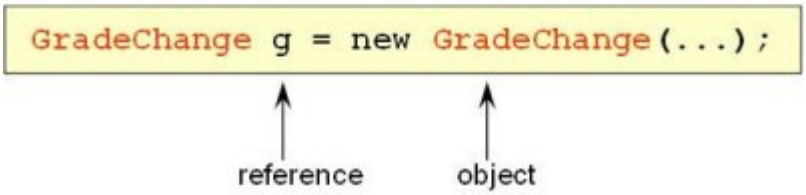
The caller and the target do need to know a little bit about each other since the caller is, in effect, making a method call on the target. The caller and target need to agree on the type of information that flows between them. In the student example, it makes sense for the student to send the new value for their gpa to a target such as a parent. We'll keep things simple and assume the parent doesn't need to send any information back to the student. If we think of these two pieces of information in terms of a method signature, we would picture a method that had an argument of type `double` (the gpa) and a return type of `void` (nothing gets passed back to the student). The information flowing between the student and their parent is described in the figure below.



For the student and parent to interact, they need a delegate that describes the information flowing between them. A delegate is defined using the C# keyword `delegate` in front of what otherwise looks like a normal method signature. The name of the delegate goes in the position normally occupied by the method name. The argument list and return type use the standard method definition syntax. A delegate for the student/parent interaction is shown below.

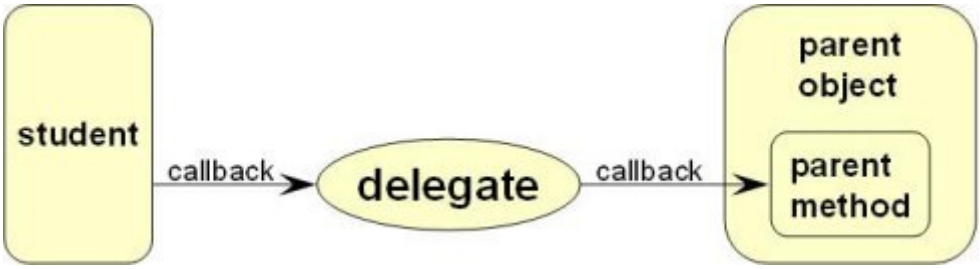


The compiler does a lot of work behind the scenes to process a delegate definition (typically generating an entire class from the delegate). The result of all the compiler's work is that a delegate name is actually a type name and we can declare references of that type and create objects.

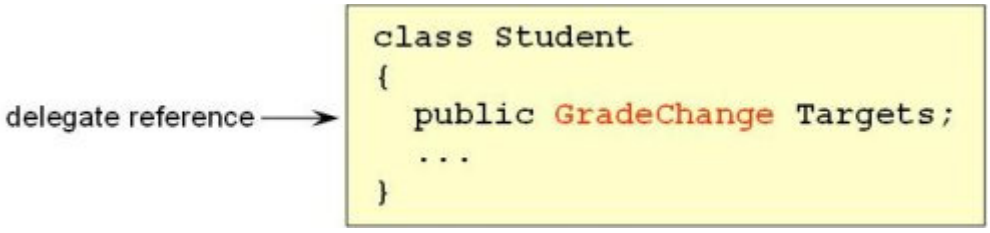


Delegate use

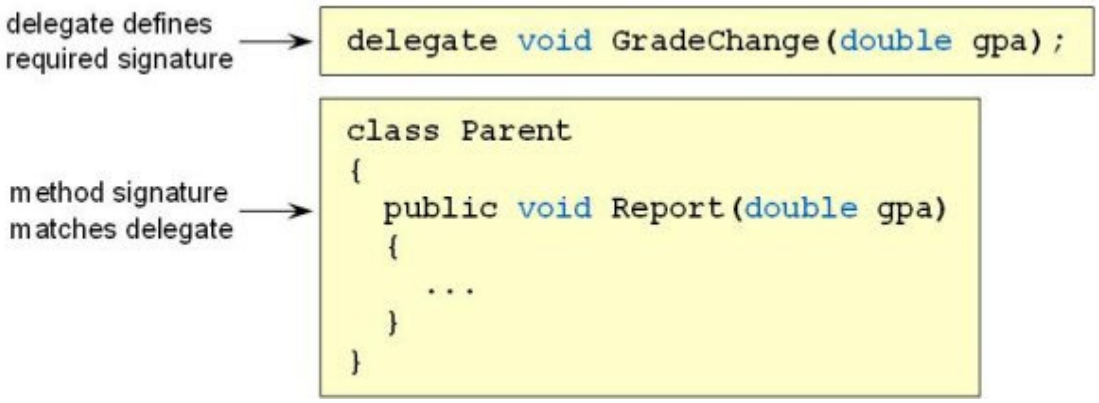
Examine the diagram below that shows the links between the student, the delegate, and the parent.



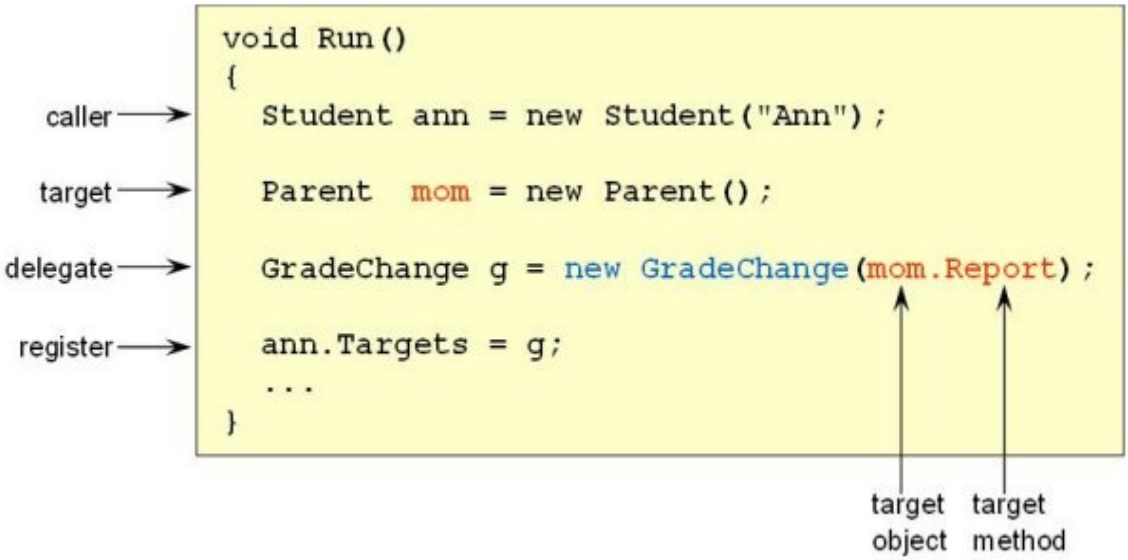
The student needs to hold a reference to the delegate. To accomplish this, we simply add a field to the student class of the delegate type.



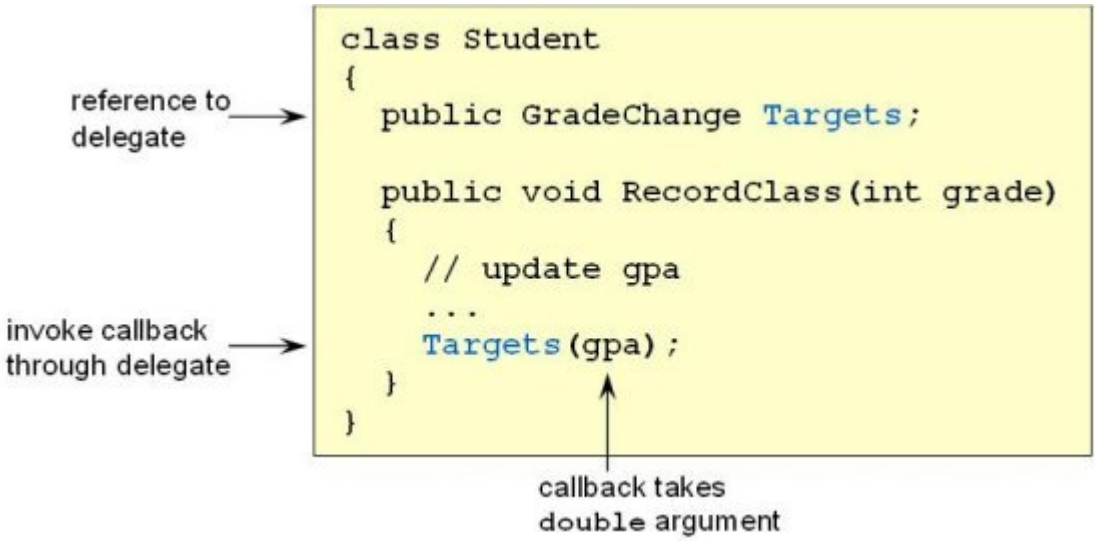
The parent needs to define a method for the student to call (indirectly through the delegate of course). The student and parent have agreed that the method should take a `double` argument and return `void` and this interaction was formalized in the delegate definition. The parent must then define a method that conforms to the required signature. The parent is free to name the method anything they like, only the parameter list and return type are constrained.



Next we write the client code that puts everything together. We need to create three objects: the student, the parent, and the delegate that links them. Creating the student and the parent are both straightforward operations. Creating the delegate is much more interesting. When a delegate is created it needs to be given the target object and the target method since both are necessary to make a method call in an object-oriented system. The target information is passed to the delegate constructor and will be stored inside the delegate object. Once the delegate is created, we register it with the student by assigning to the student's delegate field.



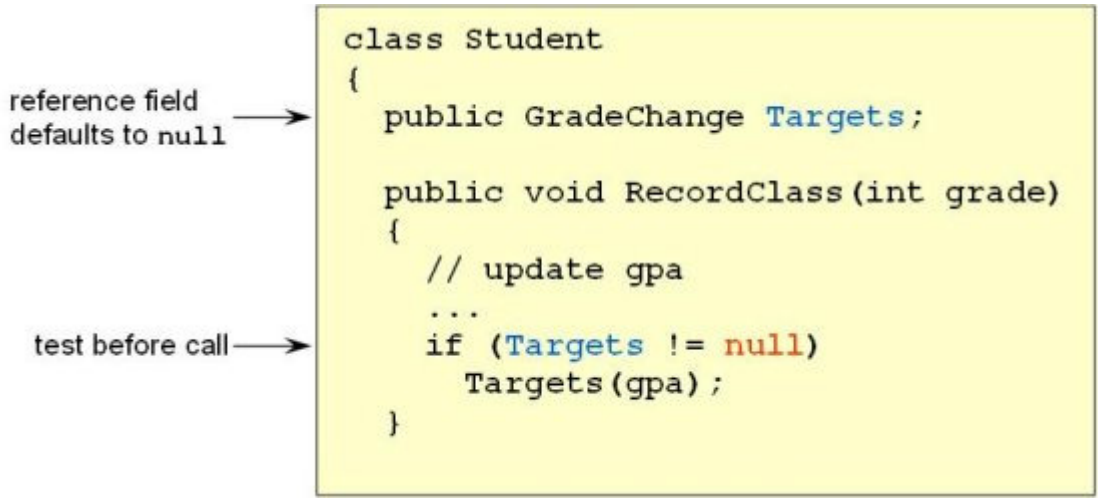
The student needs to call through the delegate whenever their gpa changes. To accomplish this, we modify the student's `RecordClass` method to include the invocation. The call is made using regular method call syntax on the delegate. Internally, the delegate calls the target method on the target object and passes along the gpa.



Null reference

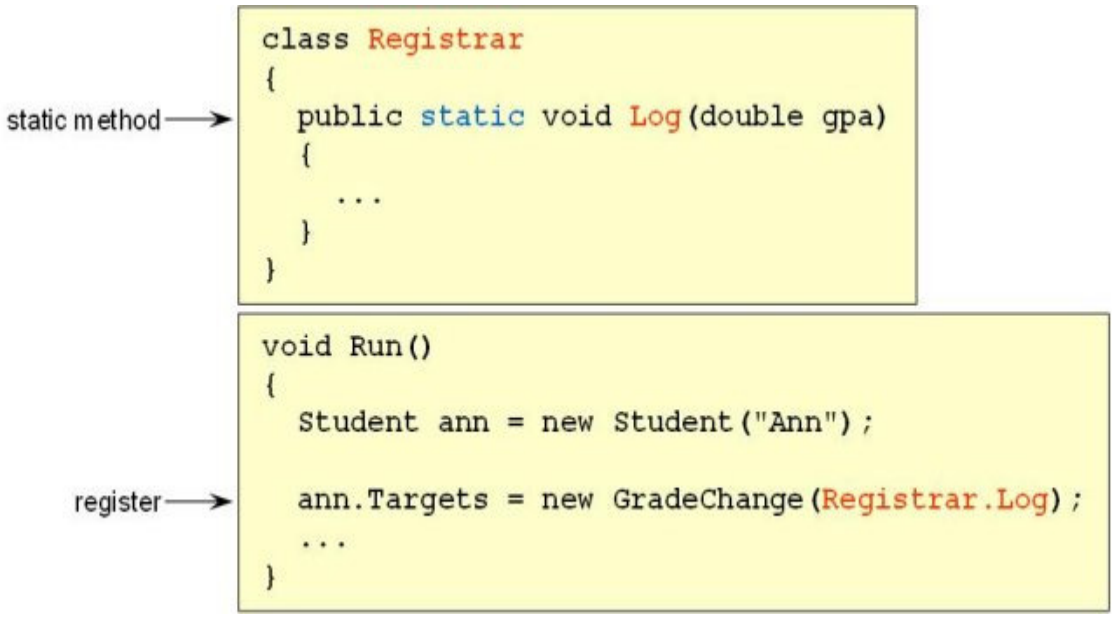
A delegate is a reference type so delegate fields will default to `null`. It is common practice to test delegate fields to make sure they are not `null` before attempting to make a callback. The CLR will throw a `NullReferenceException` if the call is made when the

reference is `null`.



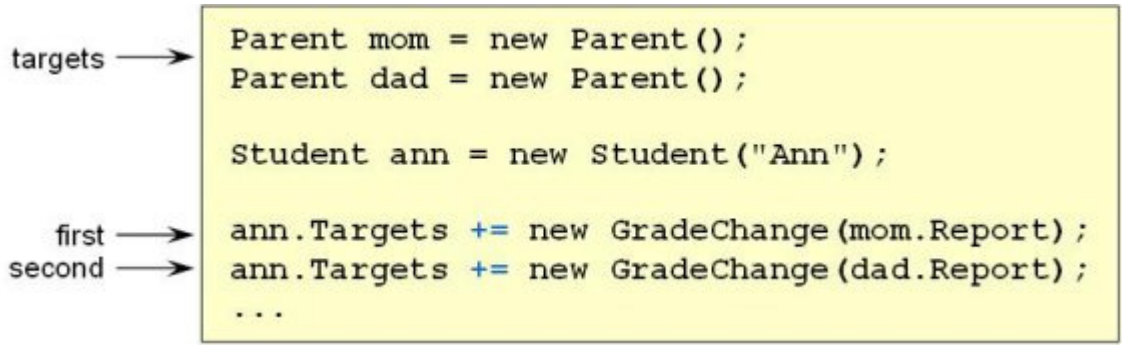
Static methods

A static method can be registered using a delegate. When the delegate is created, simply pass `ClassName.MethodName` as the constructor argument. Because static methods do not have a `this` reference, a target object is needed.

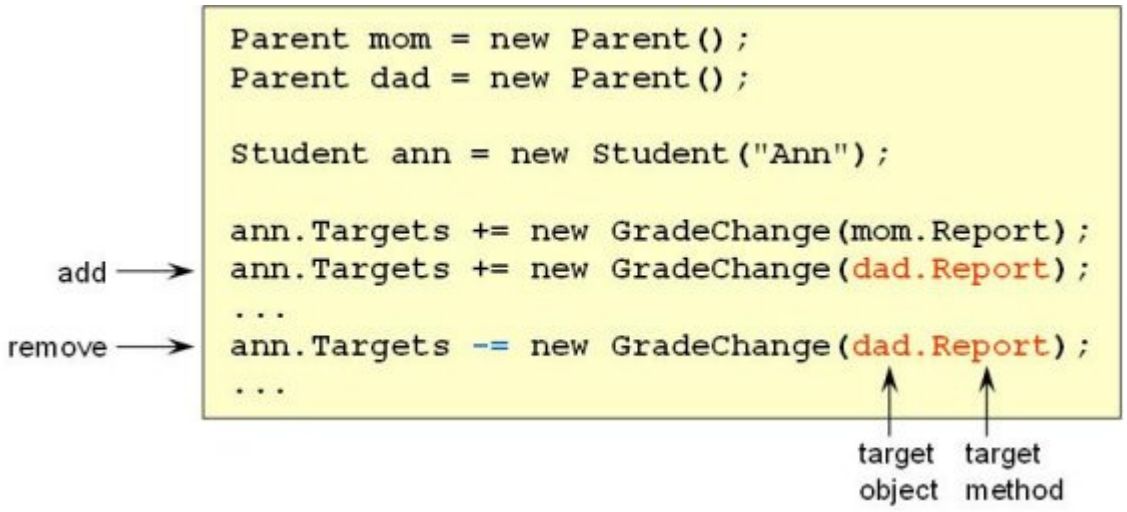


Multiple targets

Multiple targets can be registered with a delegate. Each delegate stores an "invocation list" of targets and the + and += operators are used to add new targets to the list. All targets in the list get called back when the delegate is invoked. The targets will be called in the order they were added to the invocation list. The student class might use this feature to notify both their parents whenever their gpa changes. One minor point is worth mentioning. Notice that += is used to register both targets in the example below. This is ok even though when the first target is registered the delegate will be `null`. The special case is automatically handled to help make the client code simpler.



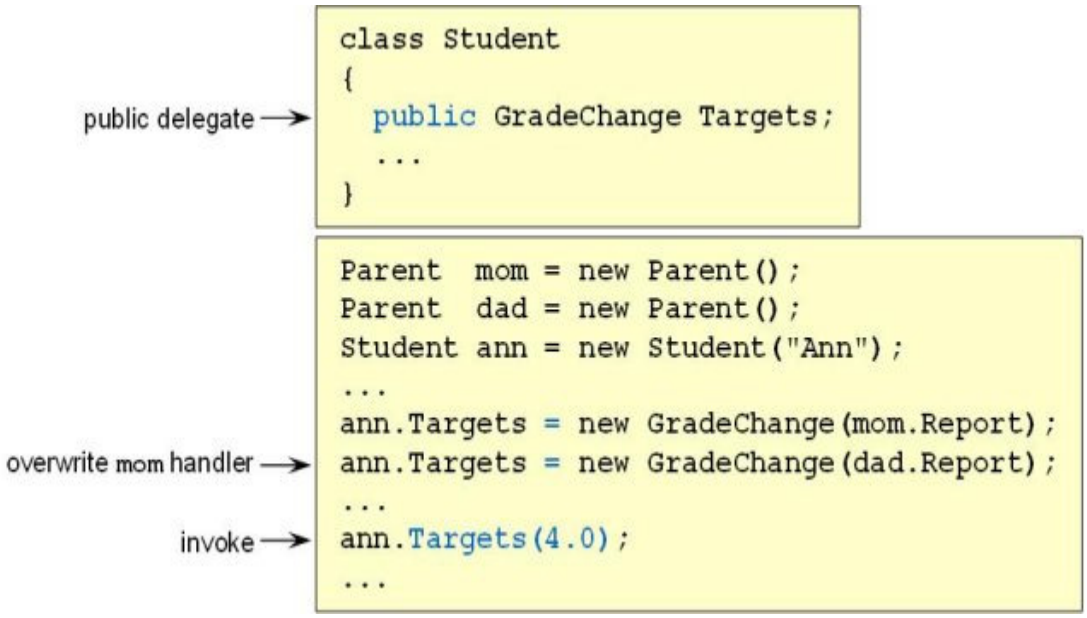
The - and -= operators are used to remove a target from an invocation list.



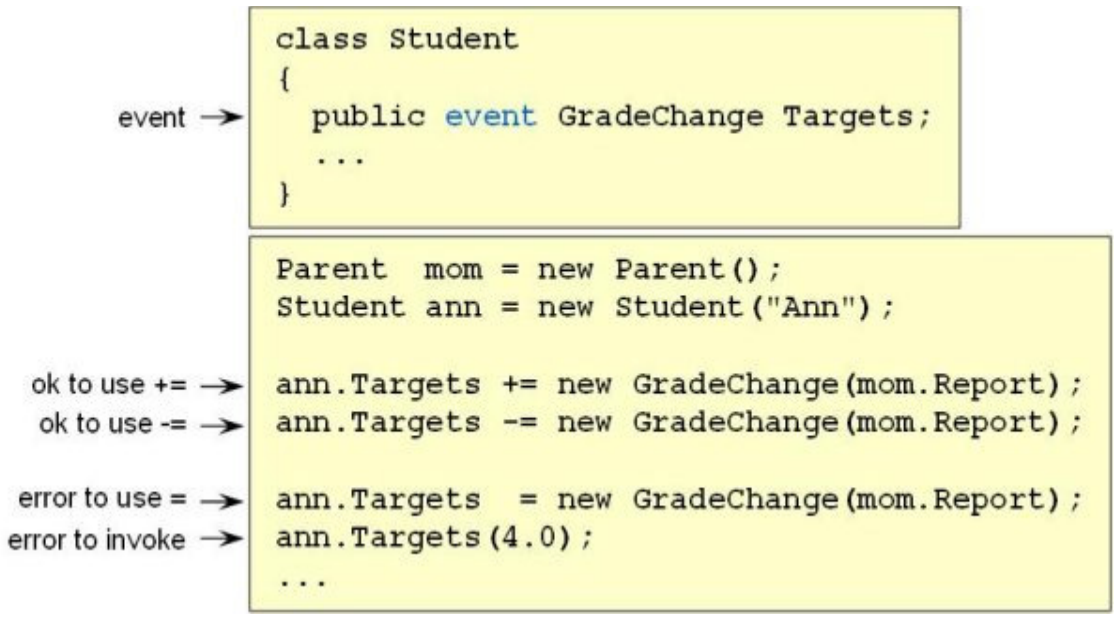
Events

There are several operations that can be performed on a delegate: assignment, registering a target with +=, removing a target with -=, and invocation. The common wisdom says

that external code should be able to register a new target or unregister an existing target and should have no additional control over a delegate field. We do not want to allow external code to assign to a delegate because a careless client could accidentally remove all previously registered targets simply by using `=` instead of `+=`. We do not want to allow invocation because the external code is not in a position to know when conditions are right for a callback. The object internally should make this decision whenever its state changes. So we have a problem. If we make the delegate field public, the client code gets full control including the power of assignment and invocation. On the other hand, if we make the delegate field private the client code has no access at all and cannot even register or unregister targets.



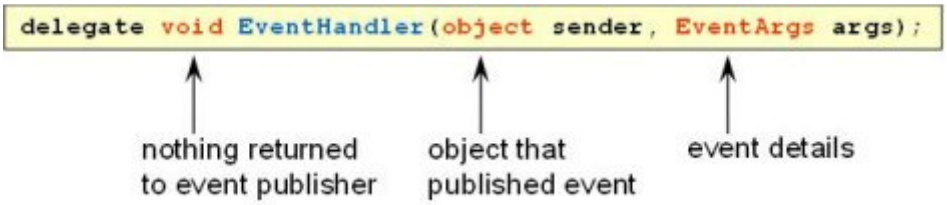
This discussion is reminiscent of the private data / public access method pattern. The class designer could make the delegate field private and code public `Register` and `Unregister` methods. This would give the desired interface since clients could only add and remove targets and would have no additional power. The designers of C# decided that requiring programmers to constantly code that pattern was too much busywork. To solve the problem they introduced the `event` keyword. Adding `event` to a public delegate field gives exactly the desired behavior. External code can still use the `+=` and `-=` operators but invocation and direct assignment will not compile.



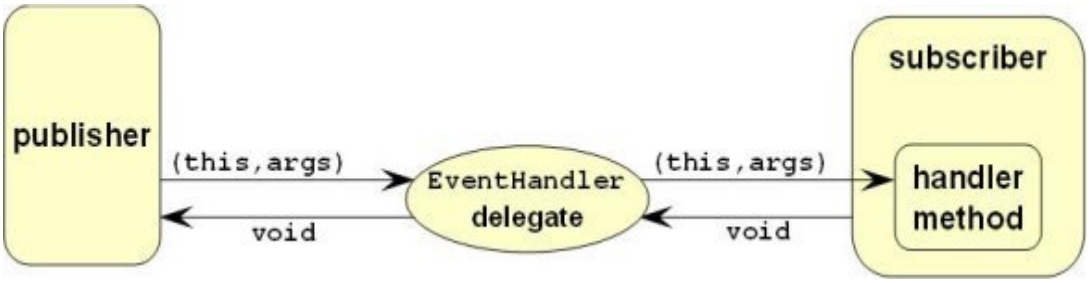
Applications

The .NET Framework Class Library makes extensive use of events. For example, when creating a new `Thread`, the client must pass a `ThreadStart` delegate to the constructor. The delegate is used to specify the work that the new thread should perform. Another common application is in user interface programming. Controls such as buttons, text boxes, list boxes, etc. offer events. Clients create delegates and register them with the controls. When the event is triggered, the client will be called back through the delegate. To supply a concrete example of delegates and events in use, we will conclude our discussion here by taking a closer look at a few of the library delegate types and their use with GUI programming.

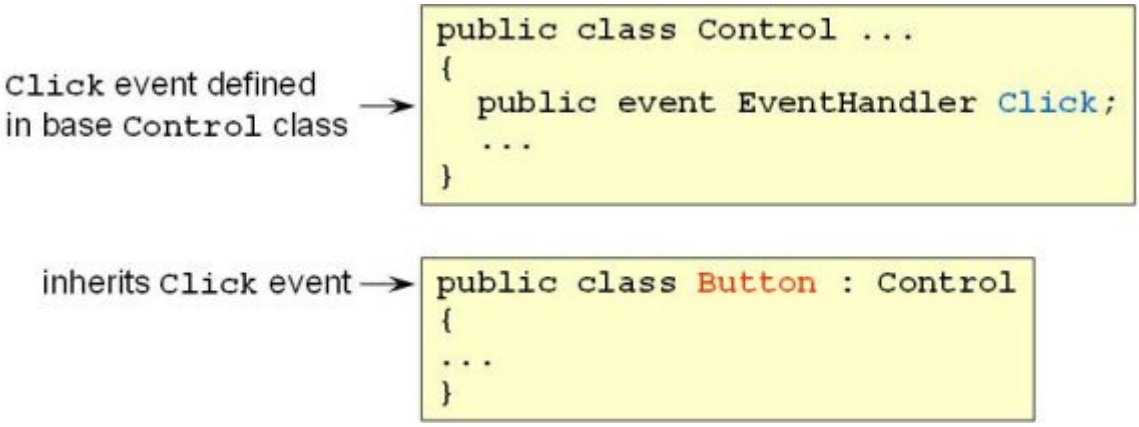
The .NET Framework Class Library defines a delegate named `EventHandler` which is used throughout the library. The `EventHandler` type is actually defined in the `System` namespace which indicates it is intended to be applicable to many areas of the library. The definition of the `EventHandler` delegate is shown below. Note how the delegate captures the information flowing between a publisher and a subscriber. A publisher passes itself as the first argument and an `EventArgs` object as the second argument. Passing itself to the handler method lets the client code easily determine which object fired the event. The `EventArgs` object is used to pass details of the event.



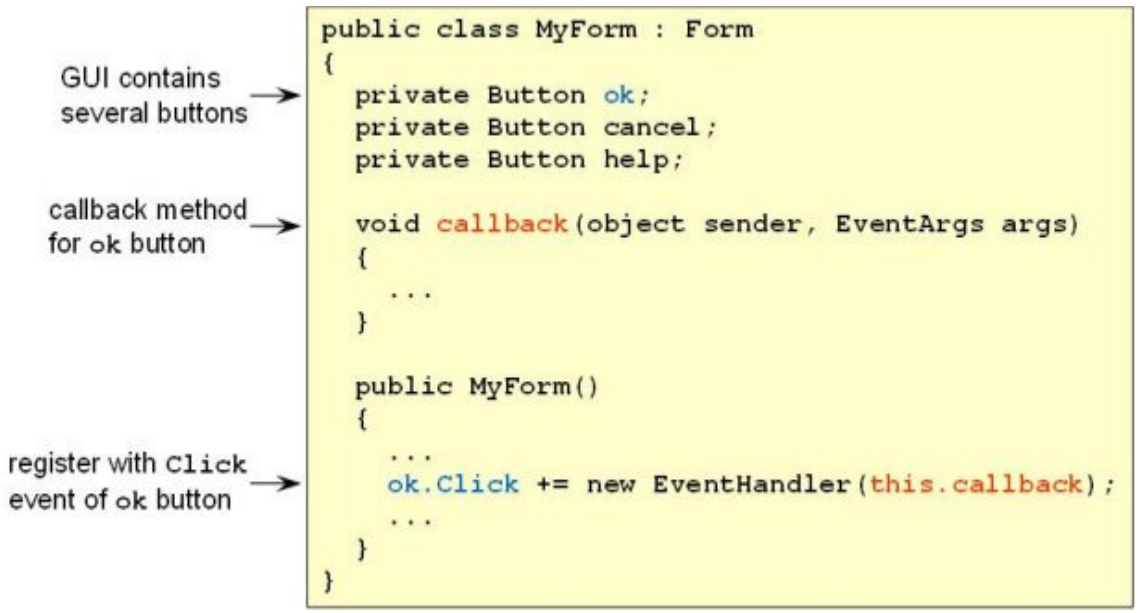
The data flow through an `EventHandler` delegate is summarized in the figure below.



The controls in the `System.Windows.Forms` namespace use an `EventHandler` for their `Click` event. The event is declared in the base class `Control` and is inherited by all derived controls such as the `Button` class. The figure below shows the definition of the event.

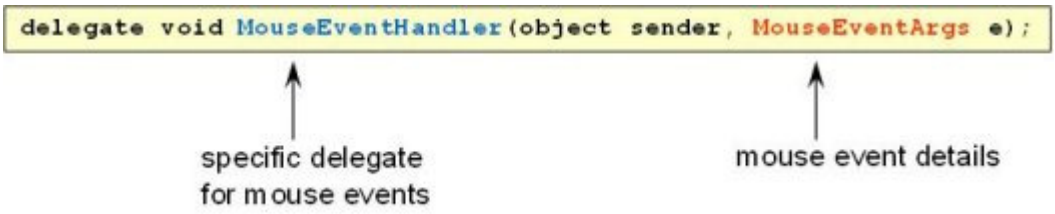


Clients that would like to register for the `Click` event must define a method with the appropriate signature: a first parameter of type `object`, a second parameter of type `EventArgs`, and a return type of `void`. They then create an `EventHandler` delegate and use the `+=` operator to register with the event. The registration process is shown in the figure below. When the user clicks on the `ok` button in the GUI, the `callback` method will be invoked.



The `Click` event is a good introduction to the topic of GUI event handling but it is a little too simple to show the full power of the model. The main reason the `Click` event is not representative is that there is generally no need to pass any information from the control to the event handler for something as simple as a `Click`. That is, the only thing the handler needs to know is that the button was clicked and there is typically no need for more detailed data to be sent. In fact, the `EventArgs` class does not even contain any data so the publisher could not pass any additional information if it wanted to.

Other types of events do need to pass additional information. For example, a mouse event would likely need to send the coordinates of the mouse click and the identity of the mouse button the user pressed. To accommodate this type of interaction there are more specific delegate types that pass derived types of `EventArgs` as parameters. The derived types contain the detailed event information. A delegate definition for a mouse event is shown in the figure below. Notice how the second parameter is now `MouseEventArgs` rather than the more general `EventArgs`.



The `MouseEventArgs` class is derived from `EventArgs` and adds information specific to mouse events. Part of the class definition is shown below.

info specific to mouse event →

```
class MouseEventArgs : EventArgs
{
  public MouseButtons Button { get; }
  public int          Clicks { get; }
  public int          X      { get; }
  public int          Y      { get; }
  ...
}
```

The `Control` class offers mouse events such as the `MouseDown` event shown in the next figure. Notice that the type of the event is `MouseEventHandler` rather than the generic `EventHandler`.

mouse event →

```
public class Control : Component ...
{
  public event MouseEventHandler MouseDown;
  ...
}
```

A client that wishes to subscribe to the `MouseDown` event must code a method that fits the `MouseEventHandler` pattern. The process is shown in the figure below.

control offering mouse events →
callback method →
event details →
register →

```
public class MyForm : Form
{
  private Panel drawingSurface;

  void mouseCallback(object sender, MouseEventArgs args)
  {
    ...
    int xCoord = args.X;
    int yCoord = args.Y;
    ...
  }

  public MyForm()
  {
    ...
    drawingSurface.MouseDown += new MouseEventHandler(this.mouseCallback);
    ...
  }
}
```

There are other interesting events as well such as those for keyboard activity and painting. They all fit the same basic pattern: a specific delegate type with a derived class of `EventArgs` to transmit the event details.

13- Delegates and Events Exercise

Goals:

- Define a delegate type.
- Use a delegate to implement a simple callback.
- Register both instance and static callback methods.
- Use composition to create a delegate with multiple targets.
- Use the event keyword to impose well know access semantics on a delegate.
- Manually invoke the targets of a delegate.
- Code a delegate with a return value.

Overview

A delegate serves as a proxy for a method. Invoking the delegate causes it to invoke the method it represents. Similar concepts are implemented using function pointers in C/C++ and interfaces in Java. The classic application for this technique comes from graphical user interface programming. Controls such as buttons, trees, lists, etc. publish events to which interested parties subscribe. User actions such as clicking a button trigger the event and the control notifies all its subscribers that the event occurred.

Part 1- Basics

In this lab we explore the basic definition and use of delegates and events. There will be three main pieces: the delegate definition, the publisher that invokes through the delegate, and the subscriber that receives the call.

Steps:

1. Define a delegate named MessageHandler for methods that take a string argument and return void. Recall that the delegate definition syntax is similar to a method signature, the generic pattern is shown below.

   ```
   delegate returnType delegateName(arguments);
   ```

2. Define a class Publisher which will publish string messages by invoking through a delegate. Add a public field of type MessageHandler to store the delegate. Add a public method called Dispatch that invokes the delegate. Recall that a delegate field is a reference with a default value of null so it is typical to place the call inside a conditional to ensure the delegate is valid. The delegate specifies a string argument so you can pass a message of your choice.
3. Define a class Subscriber which will subscribe to the messages published by the Publisher. Add an instance method to the subscriber that matches the signature of the delegate; that is, it takes a string argument and returns void. Name the method CallMe and have it simply print out a message containing the string it receives from the publisher.
4. Define a class called Driver and add a Main method. Create a publisher and a subscriber. Create a delegate for the SubscriberCallMe method and install it on the publisher. Call the publisher's Dispatch method and verify that the delegate invokes the subscriber method.
5. Add an instance method named MeToo to the subscriber class. Make sure the method signature conforms to what is required by the MessageHandler delegate. Modify the driver code to register both the CallMe and the MeToo methods. Be sure to use the += operator to do the registration. Run the program and observe the output: notice that both callbacks are invoked and they are called in the order they were registered.
6. Add a static method named AndMe to the subscriber class. Make sure the method signature conforms to what is required by the MessageHandler delegate. Modify the driver code to register all three subscriber methods: CallMe, MeToo, and AndMe. Use the class name when registering the static method. Run the program and observe that all three methods are called.
7. Currently, the delegate field in the Publisher class is public so it can be accessed by the driver code. Because the delegate is public, the driver code can assign to the delegate or invoke it.

   Use the assignment operator to assign directly to the delegate from the driver code and verify that the program compiles. Direct assignment is considered bad style since it clobbers over any previously registered callbacks. The preferred alternative is the += operator which simply adds a new callback without disturbing the existing ones. If the delegate is public we have no contol over how the delegate is used by the client.

Invoke the delegate from the driver code. Again, this is allowed because the delegate is public. This is also considered poor style since the callbacks should only be made when conditions in the publisher warrant it.

Add the keyword event to the delegate declaration in the publisher. Making it an event modifies the accessibility of the delegate. It now exposes the += and -= operators to the user code but denies direct access to the underlying delegate. Code inside the declaring class still has full access to the delegate. Verify that the event cannot be directly assigned to nor invoked from the driver code.

Part 2- Invocation List (optional)

A delegate can have multiple targets. Invoking the delegate invokes all the registered targets. This automatic invocation is typically convenient and desirable; however, there are a few cases where more control is required.

Suppose a target were to throw an exception. The default behavior is to stop processing the targets and propagate the exception to the publisher. The publisher might prefer to invoke each target individually, catch any exception that is thrown, and continue the processing of the remaining targets.

Another case to consider is a delegate with a return type other than void. The default behavior is to return the result from the last target in the sequence and discard the others. The publisher might prefer to obtain the result from each target.

To allow a publisher more control over target invocation, delegates offer a GetInvocationList method that returns the individual targets. The targets are returned as an array of delegates - each element in the array represents one target. The publisher can iterate through the array and invoke each target individually.

In this lab, we recode the Publisher class Dispatch method from the last exercise so it manually invokes each target.

Steps:

1. In the PublisherDispatch method, remove the invocation of the delegate. In its place, retrieve the invocation list from the delegate, step through the array, and invoke each target individually.

   ### Notes:

   - The type of each array element is the system type Delegate; however, the array is actually filled with MessageHandler objects. As you step through the array, downcast each element of the array to MessageHandler.
   - Use standard method call syntax to invoke the individual targets after the downcast.
   - A foreach loop is probably the simplest way to accomplish the goal here since it will both step through the array and downcast each element.

Part 3- Delegate return value (optional)

Delegates often return void. This is not surprising since their most common use is as callbacks where the publisher is notifying subscribers of an event. Typically, the subscribers do not need to return any data to the publisher.

The use of delegates does not need to be limited to callbacks. Delegates provide a clean way of decoupling the actors in many other design patterns. For example, a delegate might be used as a data source for a consumer where the consumer invokes the delegate to obtain needed data. This design insulates the consumer from the data source: the data location, the name of the class supplying the data, and even the name of the method used to retrieve the data are all hidden from the consumer by the delegate. Some delegates might draw from a database, others from a network connection, and still others from a disk file. The data source can easily be changed by supplying a different delegate without the need to modify the consumer code.

In this lab we will code a histogram class that uses a delegate for the data source of each entry. To keep things simple we will make a

number of simplifications in the drawing. It should be straightforward to remove these limitations in a more realistic implementation.

Notes:

- Each entry will have only a magnitude, text labels will not be supported.
- Magnitude will be an integer.
- Drawing will be done using text: the magnitude will be represented by the corresponding number of "*" characters.
- The histogram will be drawn "sideways" with each entry on one line.

Steps:

1. Define a delegate named HistogramSource that takes no arguments and returns an int. The int return value will be the magnitude of one histogram entry.
2. Create a class named Histogram and add a public event of type HistogramSource. Add a public method called Draw that invokes the delegates in the event. You will need to invoke the delegates manually in order to capture the return value from each call. Each delegate in the event represents one entry in the histogram and the return value is the magnitude of that entry. Output a line of "*" characters representing the magnitude.
3. Create one or more classes with methods to serve as the data sources. Feel free to choose any algorithm you would like to generate the magnitude of the entries in the histogram. To save time, you can use the class shown below which generates random values.

```
4.    class RandomSource
5.    {
6.              private int    max;
7.              private Random r;
8.
9.              public RandomSource(int max)
10.             {
11.                     this.max = max;
12.
13.                     r = new Random();
14.             }
15.
16.             // callback method - generates a random number between 0 and
    max-1
17.             public int Magnitude()
18.             {
19.                     return r.Next(max);
```

```
20.             }
    }
```

21. Create a driver program. Create a histogram object. Register a few data sources. Draw the histogram.

Lab: Versioning Assemblies

Estimated time for completion: 40 minutes

Goals of this activity:

- Learn how to make strong name assemblies
- Learn how to install assemblies into the GAC
- Learn how to version assemblies

Overview

In this sample you will be creating two projects within a single solution in VS.NET. One project will be a console application as before while the other will be a library application that we will version and deploy in the Global Assembly Cache (GAC).

---

part 1- Creating the projects

Create a console application with VS.NET and then add another project making sure to specify library application. We will then assign a strong name to the library project.

Criteria:

- Create a console application and add a library application to the solution. Feel free to name them whatever you want.
- Notice that in both projects the VS.NET templates used provide several files for you, one of them is the "AssemblyInfo.cs" file. If you open this file up on the library project and take a look inside you will notice a bunch of attributes. We will be modifying the AssemblyKeyFile and the AssemblyVersion attributes.
- In the library application create a simple function that returns some data, then reference this project from your console application and add code in the Main method to create an instance of this library and call the function.

  Test your application to verify everything works as expected. The code for the library and the client should look something like the following:

```
//client code
  class Client
```

```
  {
          static void Main(string[] args)
          {
                  TheWidget wg = new TheWidget();
                  Console.WriteLine(wg.Foo());
          }
  }
//library code
  public class TheWidget
  {
          public string Foo()
          {
                  return "data from foo";
          }
  }
```

Once you have tested your application close the console application and open only the library project. Make a change to the data returned in the library method and then recompile the library and place the new dll in the same folder as the client .exe file. The exe can be found under projectname\bin\debug. Run the client.exe from a command prompt and notice that the client exe now displays the new data from the library dll without being recompiled! In fact there is no version checking or anything being performed on the client.exe. This is a problem! Now we want to put version information on our library and make sure that this type of upgrade can't happen by mistake.

Notes:

- This part of the sample demonstrates calling an assembly that has no versioning information on it and can be easily changed right under the client with no complaints.

---

part 2- Putting version information on an assembly

Learning how to add attributes to an assembly and create a public/private key file to use when signing an assembly.

Criteria:

- Open the library application we have been working with and in the solution explorer open the AssemblyInfo.cs file. We need to modify the AssemblyKeyFile attribute found near the bottom of the file. Add a filename to the empty string in the attribute so that it looks something like the following `[assembly: AssemblyKeyFile("mykey.snk")]`. This instructs the compiler to look for a file called mykey.snk for key information to use in order to sign the

assembly when compiling. We now need to create this file and place it somewhere the compiler can find it.

- Open a the Visual Studio Command Prompt, located on the programs menu with VS.NET, and then navigate to your client project folder. We need to generate the mykey.snk file.

  Type `sn.exe -k mykey.snk` this will invoke the strong name utility and tell it to generate a public/private key pair in a file named mykey.snk.

  Now that we have generated the file we need to compile our library application. VS.NET by default looks in the `projectfolder\obj\debug` folder to locate mykey.snk. So, we can either put the file there or we can instruct VS.NET to look in the root by changing the attribute to the following: `[assembly: AssemblyKeyFile(@"..\..\mykey.snk")]`, you decide what works best for you. Just realize that if the compiler can't find mykey.snk you will get an error and your assembly will not be signed.

- Once you are sure everything is compiling fine with the library go ahead and open the client and recompile it, being sure to use the new signed assembly.

  Now test the client.exe by running it from a command prompt, it should work fine. Then go back and change your library component and recompile ONLY the library. Drop the new library.dll in the same folder as the client.exe as we did earlier, over writting the one currently there, and then test the client.exe again. It should fail this time as the client.exe is expecting/demanding the older signed version of the assembly which is no longer available. You now have version control.

- Where did the version numbers come from? If you look in the AssemblyInfo.cs file of the library application you will notice that there is an AssemblyVersion attribute that looks like the following: `[assembly: AssemblyVersion("1.0.*")]`, this tells VS.NET to simply increment the build and revision numbers everytime you compile, which gives us a new version each time. If you want to manually control when you get a new version simply change the attribute to something you want like: `[assembly: AssemblyVersion("1.0.0.0")]`.

Notes:

- You might also inspect the library.dll and client.exe with ILDASM.EXE before and after setting a strong name on your assemblies to see the added information in the manifest when signing.

part 3- Deploying to the GAC

Install an assembly in the Global Assembly Cache.

Criteria:

- Now that you have created a strong named assembly you can deploy it in the GAC. To install the assembly in the GAC use the gacutil.exe tool.
- Open the VS.NET Command Prompt and navigate to the folder where your library.dll is located and then type gacutil -i yourlibrary.dll. This will install the dll in the GAC. Open up explorer.exe and look under Windows\Assembly to find your dll. Once you have verified it installed into the GAC go to your client.exe and remove any local copies of the dll found in the folder with the client.exe and then try running your client.exe. You should notice that the client.exe works fine because it is using the assembly found in the GAC
- Now go build another version of your library.dll and install it in the GAC. How do we get our client application to use this new version in the GAC? If you delete the old version from the GAC the client application will simply stop working. What we need to do is build an application configuration file.
- Create an xml file with the same name as your client.exe and tac .config to the end of it. Your file name should look something like `client.exe.config`. In the xml file you need to add an assembly redirect. The following sample will help you:

```
<?xml version="1.0" ?>
<configuration xmlns:asm="urn:schemas-microsoft-com:asm.v1" >
  <runtime>
    <asm:assemblyBinding>
       <asm:dependentAssembly>
        <asm:assemblyIdentity name="Widget"
                              publicKeyToken="f496b453d02f293f"
/>
        <!-- one bindingRedirect per redirection -->
        <asm:bindingRedirect oldVersion="1.0.0.0"
newVersion="2.0.0.0" />
       </asm:dependentAssembly>
    </asm:assemblyBinding>
  </runtime>
</configuration>
```

- You will need to make sure the name, publicKeyToken, oldVersion and newVersion attributes in the application configuration file match your information in order for things to work.

Notes:

- This sample is a bit tedious with respect to putting versions of files in the right place, so if things don't work right away don't be suprised it's easy to thing configured wrong. You want to make sure your configuration file is correct and that the right version you expect is in the GAC and that the folder with the client.exe doesn't contain any versions of the dll to test clearly.