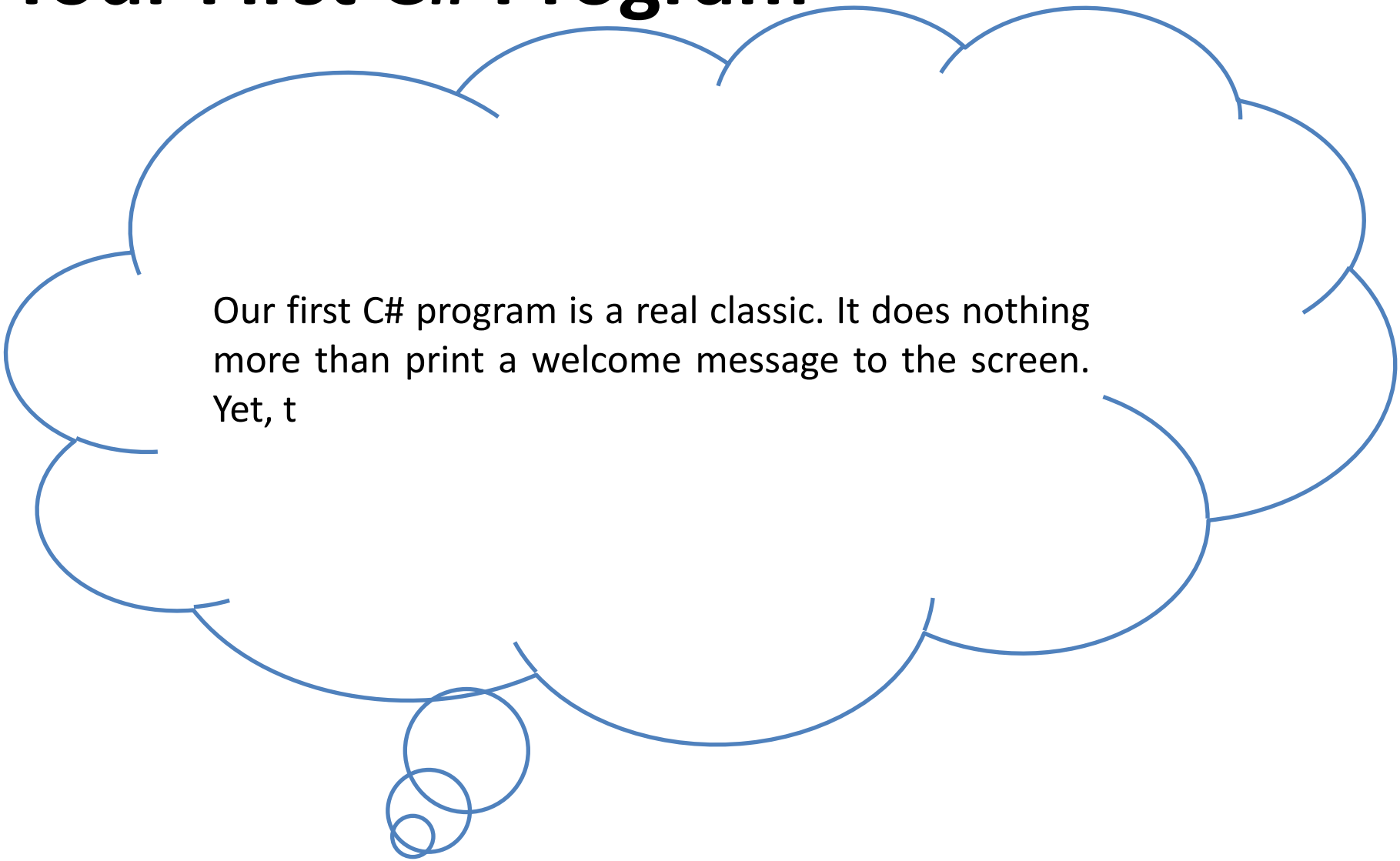
NET.C#.02
The C# language.
Fundamentals.

Your First C# Program



Our first C# program is a real classic. It does nothing more than print a welcome message to the screen. Yet, t

This program sends the simple greeting to the console

Comments

```
// Example1_1.cs
// This program sends a simple greeting to the console
//
using System;
namespace csbook.ch1
{
    class Example1_1
    {
        static void Main(String [] args)
        {
        Console.WriteLine("Welcome to C# Programming.");
        }
    }
}
```

using the System namespace

Creating namespace

Class definition

The Main method

When you execute that program and if all goes well you should have a new file in your directory named Example1_1.exe. If you run that file you should see your welcome message print to the screen.

A Closer Look to the strucure of C# program

In C#, as in other C-style languages, every statement must end in a semicolon (;)

Statements can be joined into blocks using curly braces { }.

Single-line comments begin with two forward slash characters (//), and multi-line comments begin with a slash and an asterisk (/*) and end with the same combination reversed (*/).

The first couple of lines have to do with *namespaces*, which are a way to group together associated classes.

The *using* statement specifies a namespace that the compiler should look at to find any classes that are referenced in your code but which aren't defined in the current namespace.

All C# code must be contained within a class. The class declaration consists of the class keyword, followed by the class name and a pair of curly braces. All code associated with the class should be placed between these braces.

Every C# executable (such as console applications, Windows applications, and Windows services) must have an *entry point* — the *Main()* method

Variables in C#

We th

```
DataType variableName;
// or
DataType variableName1, variableName2;
//for exampleint
```

```
variableName = value;
```

The compiler wo iable until we have initialized it with a value, but allocates 4 bytes on the stack to hold the value.

If we declare and initialize more than one variable in a single statement, all of the variables will be of the same data type

```
int x=10, y=20;
//x and y are both ints
```

```
int x=10, bool y=true;
//This won't compile
```

Don't assign different data types within a multiple variable declaration

Initialization of Variables

Can't do this ! Need to initialize d befor use

```
public static void Main()
{ int d;
  Console.WriteLine(d);
}
```

Error message: Use of unassigned local variable 'd'

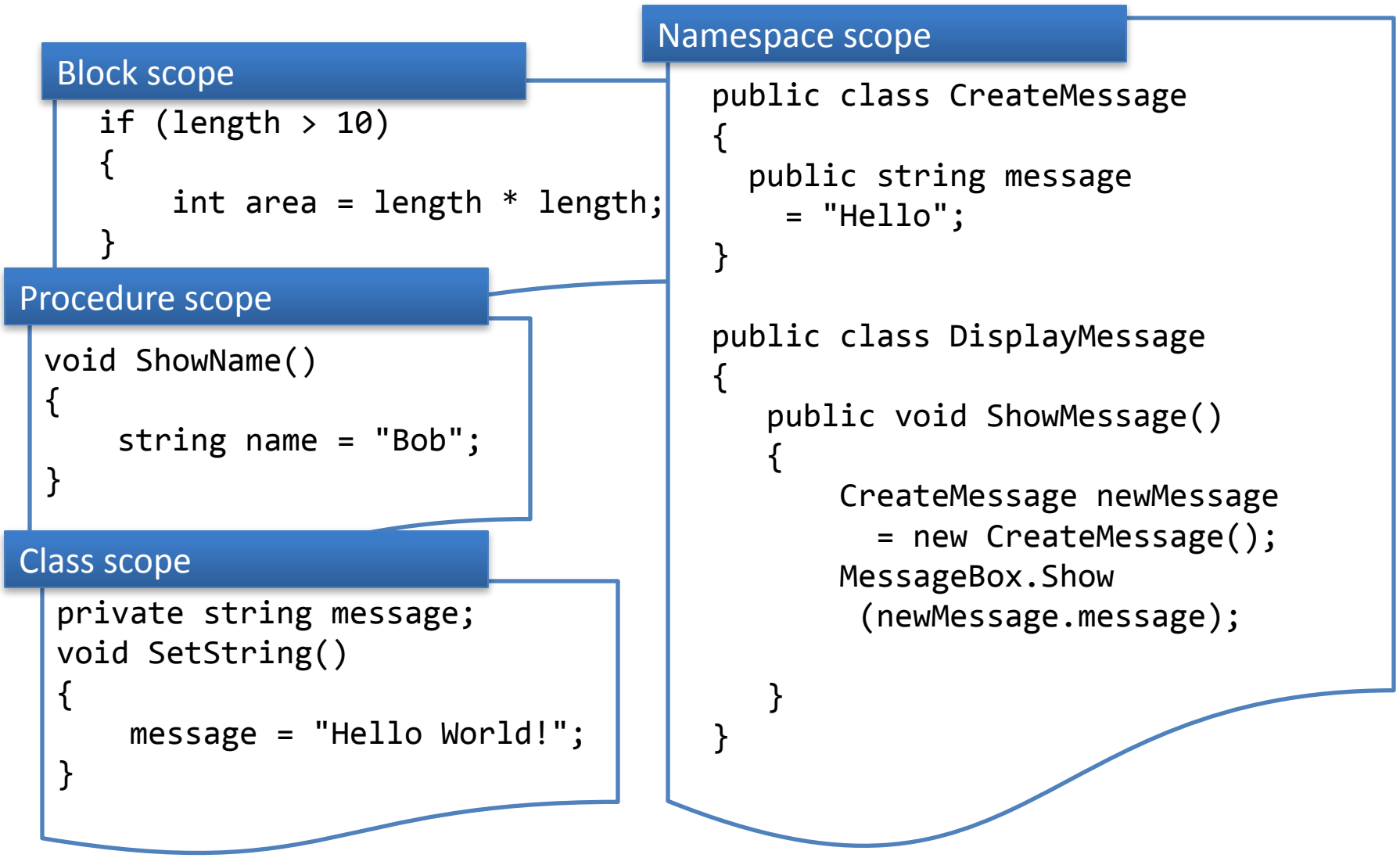
CTE

```
objSomething = new So
```

Would only create a reference for a *Something* object. But this reference does not yet actually refer to any object

Instantiating a reference object in C# requires use of the new keyword

Variable scope



Scope clashes for local variables

We can't do this:

```
int x=20;
//some code
int x=30;
```

We can't do this:

```
public static i
n
t Main(){   int j = 20;
```

We can do this:

```
public static int Main(){
f
or (int i = 0; i < 10; i
++)  {   Conso
l
e
.WriteLine(i);  } // i goes

out of scope here// We can d
ecla
re a variable named i //again, b
ecause// there's no other varia
ble with that //name i
n scope for (int i = 9; i >=

0; i
-
```

Scope clashes for fields and local variables

We can do this:

We can do this:

```
class ScopeTest2
{
  static int j = 20;  public st
a
```

What number will be displayed ?

What number will be disp ed ?

First case: the variable j in the Main() hides the class-level variable with the same name

Second case: We use the name of the class for static field j

Constants

A constant is a variable whose value cannot be changed throughout its lifetime

Constants have the following characteristics:

They must be initialized when they are declared, and once a value has been assigned, it can never be overwritten.

We can't initialize a constant with a value taken from a variable. If you need to do this, you will need to use a readonly field

Constants are always static. However, notice that we don't have to include the static modifier in the constant declaration

Statements

Programs consist of sequences of C# statements

- **Conditional statements**
- **The switch statement**
- **Loops**
- **Jump statements**

Statements allow us to control the flow of our program rather than executing every line of code in the order it appears in the program

Conditional statements

The if statement

```
if ([condition])
{
    [code to execute if condition is true]
}
else
{
    [code to execute if condition is false]
}
```

- way if

Either – or if

The conditional operator

```
Type result = [condition] ? [true expression] : [false expression]
```

The switch statement

Syntax

```
switch ([expression to check])
{
    case [test1]:
    ...
        [exit case statement]
    case [test2]:
    ...
        [exit case statement]
    default:
    ...
        [exit case statement]
}
```

Example

```
switch(a)
{
    case 0:
        // Executed if a is 0.
        break;
    case 1:
    case 2:
    case 3:
        // Executed if a is 1, 2, or 3.
        break;
    default:
        // Executed if a is any
        // other value.
        break;
}
```

Loops

C# provides four different loops

- The for loop
- The while loop
- The do … while loop
- The foreach loop

Loops allow us to execute a block of code repeatedly until a certain condition is met.

The for loop

Syntax

for ([cou])
{ [Cod

Example

```
for (int i = 0; i < 10; i++)
{
    // Code to loop, which can use i.
}
. . .
for (int i = 0; i < 10; i = +2)
{
    // Code to loop, which can use i.
}
. . .
int j;
for (j = 0; j < 10; j++)
{
    // Code to loop, which can use j.
}
// j is also available here
```

Nested loops

Example

```
static void Main(string[] args)
{
  // This loop iterates through rows...
  for (int i = 0; i < 100; i+=10)
  {
      // This loop iterates through columns...
     for (int j = i; j < i + 10; j++)
     {
       Console.Write(“ “ + j);
     }
    Console.WriteLine();
  }
}
```

The while loop

Like the for loop, while is a pre-test loop. The syntax is similar, but while loops take only one expression

Syntax

```
while ([condition])
    [Code to loop]
}
```

Example

```
double balance = 100D;
double rate = 2.5D;
double targetBalance = 1000D;
int years = 0;
while (balance <= targetBalance)
{
    balance *= (rate / 100) + 1;
    years += 1;
```

- Unlike the for loop, the while loop is most often used to repeat a statement or a block of statements for a number of times that is not known before the loop begins.

The do … while loop

The do...while loop is the post-test version of the while loop

Syntax

```
    [Code to loop]
} while ([condition]);
```

Example

```
string userInput = "";
do
{
    userInput = GetUserInput();
    if (userInput.Length < 5)
    {
  // You must enter at least 5 characters.
    }
} while (userInput.Length < 5);
```

This means that the loop's test condition is evaluated after the body of the loop has been executed. This loop will at least execute once, even if Condition is false.

The foreach loop

The foreach loop allows us to iterate through each item in a collection. The Collection is an object that contains other objects.

Syntax

```
each (type [ item
] in [ collection ]{
```

Example for C# arrays

```
foreach (int temp in arrayOfInts) { Console.WriteLine(temp);}
```

We can't change the value of the item in the collection, so code such as the following will not compile:

CTE

```
foreach (int temp in arrayOfInts) { temp++; Console.WriteLine(temp);}
```

Jump statements

C# provides the number of statements that allow us to jump to another line in the program

The break statement

```
ak can be used to
exit from for, foreach
```

```
The return statement i
s
u
sed to exit a method of a class
, returning control to the ca
```

The return statement

```
eturn type,
retur
n must return a value of this t
```

Classes and Structs

Classes and structs are templates form wich we can create objects. Each object contain data and has methods to manipulate and access data.

Classes are reference type stored in the heap.

Struct are value type stored on the stack.

Simple Class

```
class PhoneCustomer
{
public const string
DayOfSendingBill = “Monday”;
public int CustomerID;
public string FirstName;
public string LastName;
}
```

Simple Struct

```
struct PhoneCustomer
{
public const string
DayOfSendingBill = “Monday”;
public int CustomerID;
public string FirstName;
public string LastName;
}
```

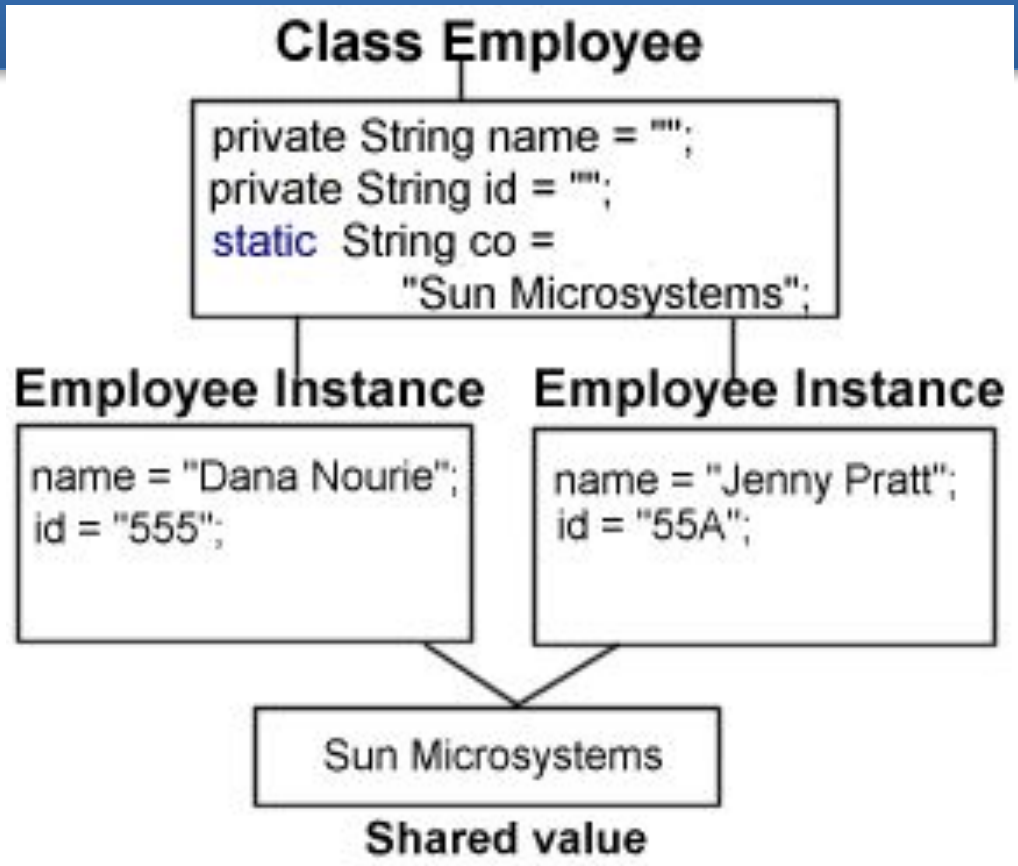
```
PhoneCustomer myCustomer = new PhoneCustomer();// works for a class

PhoneCustomer myCustomer = new PhoneCustomer(); // works for a struct
```

Classes Members

Data members are those members that contain the data for the class – fields, constants, and events.

Data members can be static (associated with the class as a whole).

Data members can be instance (each instance of the class has its own copy of the data).



Classes Members

Fields are any variables associated with the class. We can access these fields using the Object.FieldName syntax

```
PhoneCustomer Customer1 = new PhoneCustomer();
Customer1.FirstName = “Simon”;
```

Constants can be associated with classes in the same way as variable. If it is declared as a public, it will be accessible from outside the class.

```
public const string DayOfSendingBill = “Monday”;
```

Events are class members that allow an object to notify a caller whenever something happens, such as a field property changing or something else.

The client can have a code known as an event handler that reacts to the event.

Function Members

Function members are those members that provide some functionality for manipulating the data in the class. They include:

- **Methods**
- **Properties**
- **Constructors**
- **Finalizers**
- **Operators**
- **Indexers**

Methods are functions that are associated with a particular class. They can be instance or static methods (like the Console.WriteLine() method).

Declaring methods

The definition of the method consists: method modifiers, the type of return value, the name of the method, a list of arguments, the body of the method

```
[modifiers] return_type MethodName([parameters])
{
// Method body
}
```

```
public bool IsSquare(Rectangle rect)
{
  return (rect.Height == rect.Width);
}
```

A method can contain as many return statements as required:

```
public bool IsPositive(int value)
{
   if (value < 0) return false;
   return true;
}
```

Passing parameters to methods

By reference

By value

```
class ParameterTest{ s
ta
t
ic void SomeFunction(int[] ints
, int i)  {     ints[0] = 100
;  i
= 100;  }public static int Main(){   int i = 0;
int[]
 ints = { 0, 1, 2, 4, 8 };  //
```

Ref parameters

Passing variables by value is the default. We can, however, force value parameters to be passed by reference. To do so, we use the ref keyword. If a parameter is passed to a method, and if the input argument for that method is prefixed with the ref keyword, then any changes that the method makes to the variable will affect the value of the original object

```
// define method stati
c
v
oid SomeFunction(int[] ints, re
f int i)   {     ints[0] = 10
0;  i
 = 100;   }  // We will also need to add the ref
 when
 we invoke the method
```

Any variable must be initialized befor it is passed into a method, whether it is passed in by value or reference

Out parameters

C# requires that variables be initialized with a starting value before they are referenced. But if the method arguments is prefixed with out, that method can be passed a variable that has not been initialized. The variable is passed by reference

```
// define method stati
c
v
oid SomeFunction(out int i)   {
     i = 100;   }public stati
c voi
d Main()   {     int i; // i is declared but not
 init
ialized     SomeFunction(out i)
```

Properties

The idea of a property is that it is a method or pair of methods that are dressed to look like a field

Suppose you have the following code:

```
// mainForm is of type of System.Windows.Forms
 mainForm.Height = 400;
```

On executing this code, the height of the window will be set to 400 and you will see the window resize. The code looks like we are setting a field, but in fact we are calling a property accessor that contain code to resize the form.

To define a property…

We use the following syntax:

```
public string SomeProperty
{
    get
     {
        return “This is the property value”;
          }
        set
         {
           // do whatever needs to be done to set the property
         }
}
```

Get accessor

Set accessor

The get accessor takes no parameters and must return the same type as the declared property.

To define a property…

Example:

Class field

Property

```
private string foreNam
e;
p
ublic string ForeName{ get  { r
eturn foreName;  } set   {
  if
(value.Length > 20)        // code here to take
error
 recovery action        // (eg.
```

set works

get works

Thanks for your attention