C# Programming

Notes by JACPro

Table of Contents

[C# 1.x Notes 3](#_Toc61467614)

[Classes And Structs 4](#_Toc61467615)

[Classes 4](#_Toc61467616)

[Inheritance 4](#_Toc61467617)

[Abstract Classes 4](#_Toc61467618)

[Sealed Classes 5](#_Toc61467619)

[Structs 5](#_Toc61467620)

[Instantiating Structs 6](#_Toc61467621)

[Interfaces 8](#_Toc61467622)

[Interfaces 8](#_Toc61467623)

[Interface Properties 11](#_Toc61467624)

[Delegates 12](#_Toc61467625)

[Delegates 12](#_Toc61467626)

[Events 13](#_Toc61467627)

[Events 13](#_Toc61467628)

[Properties 15](#_Toc61467629)

[Properties 15](#_Toc61467630)

[Auto-Implementation 16](#_Toc61467631)

[Operators and Expressions 18](#_Toc61467632)

[Operators and Expressions 18](#_Toc61467633)

[Statements 19](#_Toc61467634)

[Statements 19](#_Toc61467635)

[Attributes 20](#_Toc61467636)

[Attributes 20](#_Toc61467637)

[Expression-Bodied Members 21](#_Toc61467638)

[Expression-Bodied Members 21](#_Toc61467639)

# C# 1.x Notes

These notes were written when directly studying from the C# documentation page “[The History of C#](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history)”. I read through all documentation pages for the listed “major features of C# 1.0” and all relevant documentation linked by each of them.

This was done in preparation for reading Jon Skeet’s *C# In Depth*, as he teaches the reader about C#’s many features by exploring each released version of C# in order. Skeet builds off an assumed knowledge of the features released as part of C# 1.x.

## Classes And Structs

### Classes

public class Customer

{

//fields

//properties

//methods

//events

}

Customer object2;

Declares type and namespace but will be null unless an object is assigned.

Customer object2 = new Customer();

Create and assign an object.

Classes are reference types.

Customer object3 = object2;

object2 and object3 refer to the same object, so changes to one will be reflected in the other.

### Inheritance

You can only inherit from a single class.

public class Manager : Employee

{

//fields, properties, methods and events inherited from Employee

//new fields, properties, methods and events for Manager

}

### Abstract Classes

The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share.

Abstract classes cannot be instantiated.

public abstract class A

{

//Class members here

}

### Sealed Classes

Sealed classes cannot be used as base classes because they prevent derivation.

public sealed class D

{

//Class members here

}

A method, index, property or event can be declared as sealed in a derived class when overwriting a virtual member of the base class; this prevents the virtual aspect from being derived further.

public class D : C

{

public sealed override void DoWork() { }

}

### Structs

A structure type is a value type that can encapsulate data and related functionality. Use the “struct” keyword to define a structure type.

public struct Coords

{

public Coords (double x, double y)

{

X = x;

Y = y;

}

public double X { get; }

public double Y { get; }

public override string ToString() =>$“({X}, {Y})”;

}

Typically, you use structure types to design small data-centric types that provide little or no behaviour.

You can also declare readonly structs using the readonly keyword.

public readonly struct Coords

{

//can’t declare a parameterless constructor

public Coords (double x, double y)

{

//struct constructor must initialize all instance fields

X = x;

Y = y;

}

//can’t initialize instance fields or properties at declaration

//but can initialize static or const field or static property at declaration

public double X { get; }

public double Y { get; }

public override string ToString() =>$“({X}, {Y})”;

}

Structs can’t inherit from other classes or structs and they can’t be inherited from. However, the can **implement interfaces**.

### Instantiating Structs

Typically, structs are instantiated using the new keyword; in addition to explicitly declared constructors, structs have an implicit parameterless constructor, which produces the default value of the type.

If all instance fields are accessible, structs can also be instantiated without the “new” operator:

public struct Coords

{

public double x;

public double y;

}

public static void Main()

{

Coords p;

p.x = 3;

p.y = 4;

}

## Interfaces

### Interfaces

Interfaces contain definitions for a group of related functionalities that a **non-abstract** class or struct **must** implement. Interfaces may define **static** methods, which must have implementation. Since C# 8.0, interfaces can define default implementations for members. Interfaces **can’t** declare instance data (e.g. fields, auto-implemented properties and property-like events).

While C# doesn’t support multiple inheritance, it does allow you to implement an unrestricted number of interfaces, meaning a class can use behaviour from many different places. Additionally, structs are unable to inherit from classes or other structs, but they can implement interfaces, allowing structs to include behaviour from other sources.

Interfaces can be defined using the interface keyword, as follows:

interface IEquatable<T>

{

bool Equals (T obj);

}

By convention, interface names should start with a capital I.

Any methods declared by an interface must be implemented by any class or struct that implements that interface, with a signature matching the one specified by the interface. For example, a class that implements IEquatable **must** always have a method called Equals, which takes a parameter of type T and returns a Boolean value.

The definition of IEquatable<T> doesn’t provide an implementation for Equals; it **must** be implemented by the class or struct that implements the interface.

Interfaces **can** contain:

* Instance methods
* Properties
* Events
* Indexers
* Static Constructors
* Fields
* Constants
* Operators

Interfaces **can’t** contain:

* Instance fields
* Instance constructors
* Finalizers

By default, interface members are public.

To implement an interface member, the corresponding member of the class that implements it must be public, non-static, and have an identical name and signature to the interface member.

When implementing an interface, a class or struct **must** provide an implementation for every member which is declared but not implemented by the interface. However, if a base class implements an interface, any class derived from that base class will inherit the base class’ implementations for all interface members.

You can implement interfaces as follows:

public class Car : IEquatable<Car>

{

public string Make { get; set; }

public string Model { get; set; }

public string Year { get; set; }

//Implementation of the Equals method defined by IEquatable<T>

public bool Equals (Car car)

{

return (this.Make, this.Model, this.Year)

== (car.Make, car.Model, car.Year);

}

}

If implementing two interfaces that both contain members with the same signature, both interfaces will use the same implementation for those members.

public interface IControl

{

void Paint();

}

public interface ISurface

{

void Paint();

}

public class SampleClass : IControl, ISurface

{

public void Paint()

{

Console.Writeline(“Paint method in SampleClass”);

}

}

In the above code, calls to the paint method of SampleClass, IControl and ISurface will all call the same method. In order to implement different methods of the same name each interface, you can use explicit interface implementation.

public class SampleClass : IControl, ISurface

{

void IControl.Paint()

{

Console.Writeline(“Paint method for IControl”);

}

void ISurface.Paint()

{

Console.Writeline(“Paint method for ISurface”);

}

}

Class indexers and properties can define additional accessors for properties and indexers defined by an interface. For example, if an interface defines a property with only a get accessor, the class implementing the interface can go on to declare both get and set accessors. However, if the property uses explicit implementation (see above), the accessors **must** match.

Interfaces may also inherit from an unrestricted number of interfaces; derived interfaces will inherit all members from the base interfaces, including members inherited by those base interfaces. A class inheriting a derived interface can be implicitly converted to any of its base interfaces.

A class may implement an interface multiple times by implementing a base interface and an interface derived from it; however, a class may **only provide one** implementation of any given interface.

If an interface is inherited because the base class implements it, the implementation of the interface from the base class will be provided. However, any virtual interface members can be overwritten with new implementations if required.

When interfaces declare default implementations for methods, classes implementing those interfaces will inherit those implementations; implementations defined in interfaces are **virtual**, so they implementing class can override them with a new implementation.

### Interface Properties

Unlike fields, properties can be declared in an interface.

public interface ISampleInterface

{

string Name

{

get;

set;

}

}

Interface properties don’t usually have a body; unlike in classes and structs, declaring property accessors without a body **won’t** auto-implement the property.

## Delegates

### Delegates

A delegate is a type that represents references to methods with a particular parameter list and return type. When instantiating a delegate, its instance can be associated with any method with a compatible signature and return type. The method can then be called through the delegate instance.

Delegates are used to pass methods to other methods as arguments. Event handlers are methods invoked through delegates; you create a custom method and a class can call that method when a certain event occurs.

public delegate int PerformCalculation (int x, int y);

Any accessible class or struct with a method matching the delegate type can be assigned to the delegate. Both static and instance methods can be used.

Delegates make it possible to programmatically change method calls and plug new code into existing classes.

Delegates are ideal for callback methods; for example, a method that compares two objects could be passed **by reference** as an argument to a sorting algorithm. Since the comparison code is in a separate procedure, the sorting algorithm could be written in a more general way (i.e. pass different comparison methods to the sorting algorithm so that the same algorithm can be used for sorting various different types of data).

## Events

### Events

Events are a **late binding mechanism** (i.e. the method/function being called is looked up at runtime so the method itself can be passed as an argument) built upon the language support for delegates.

Events are used by objects to broadcast to components listening for this event that something has happened.

Many graphical systems use events to report user interaction such as mouse movements or key presses, though events are also used in other scenarios.

You can define events to be raised by your classes. Note that there may not be any objects registered for any given event; code must be written so that events are not raised unless listeners are configured.

Also note that subscribing to events creates a coupling between the object of the event and the subscribed object that’s listening (**event source** and **event sink**). You **must** ensure that event sinks unsubscribe from the event source when no longer interested in events from that source.

To define an event, use the event keyword:

public event EventHandler<FileListArgs> Progress;

The type of the event must be a **delegate type**. There are certain **conventions** to follow when declaring an event:

* Event delegate type should have a void return
* Event declarations should be a **verb** or verb phrase
  + Use past tense verbs to report something that has happened
  + Use present tense verbs to report something that is about to happen.

Using present tense often indicates that the event class is customisable in some way. A common scenario is to support cancellation; for example, a Closing event may include an argument that would indicate whether the close operation should continue. Another reason is to enable callers to an event to modify the object’s behaviour by updating the properties of the event argument. You could raise an event to indicate the next action an algorithm proposes to take; the event handler may then mandate a different action by modifying the properties of the event argument.

To raise an event, call the event handlers using the delegate invocation syntax:

Progress?.Invoke(this, new FileListArgs(file));

The ?. operator makes it easy to ensure that you do not attempt to raise the event when there are no subscribers to that event.

You can subscribe to an event using the += operator:

EventHandler<FileListArgs> onProgress = (sender, eventArgs) =>

Console.WriteLine(eventArgs.FoundFile);

fileLister.Progress += onProgress;

You can unsubscribe using the -= operator:

fileLister.Progress -= onProgress;

It’s important to declare a local variable for the expression that represents the event handlers; this ensures that the unsubscribe action removes the event handler.

## Properties

### Properties

Properties behave like fields when accessed; however properties are implemented with accessors that allow additional statements to be executed when they are accessed or assigned. Put simply, properties will appear as fields to developers accessing them, however, unlike fields, you may implement them with additional functionality such as validation, custom accessibility or lazy evaluation.

The syntax for an **auto-implemented property** is as follows:

public string FirstName { get; set; }

You can initialize a property to a value other than its default type by assigning a value after the closing brace for the property. For example:

public string FirstName { get; set; } = string.Empty;

Instead of using an auto property, you may define the fields used for storage by the property manually:

public string FirstName

{

get { return firstName; }

set { firstName = value; }

}

private string firstName;

When a property implementation is a single expression, you can use **expression-bodied members** for the getter or setter:

public string FirstName

{

get => firstName;

set => firstName = value;

}

private string firstName;

The **set** accessor always has a single parameter named value. The **get** accessor must return a value that is convertible to the type of the property (e.g. in the string FirstName property, a string must be returned).

You can write code in the **set** accessor to validate property values; for example, you could ensure a string property is not null or white space:

public string FirstName

{

get => firstName;

set

{

if (string.IsNullOrWhiteSpace(value))

{

throw new ArgumentException (“First name must

not be blank”);

}

firstName = value;

}

}

private string firstName;

You can simplify the preceding example by using the thrown expression as part of the property setter **validation**:

public string FirstName

{

get => firstName;

set => firstName = (!string.IsNullOrWhiteSpace(value))?value : throw

new ArgumentException (“First name must not be blank”);

}

### Auto-Implementation

Automatic properties are used to make private variables visible outside the class by using properties with get; set; methods to expose private fields.

* Fields are normal member variables of a class
* Properties enable a class to publicly expose methods for getting and setting private fields. Because they provide a definition but no implementation, they can be used by interfaces (unlike fields).

public int SomeProperty { get; set;}

Automatic properties are used when no additional logic is required in the property accessors (i.e. using only standard get; set; accessors).

Automatic properties are shorthand for the following non-automatic property code:

private int \_someField;

public int SomeProperty

{

get { return \_someField; }

set { \_someField = value; }

}

The above code shows how you might implement a property that was declared in an interface (or how to declare and implement a property in a class).

Uses of properties include:

* Validating data before allowing a change
* Transparently exposing data in a class where that data is actually retrieved from some other source (e.g. a database)
* They can take an action when data is changed, such as raising an event or changing the value of other fields

### Read-Only Properties

You may assign different accessibility levels to get and set accessors, rather than leaving them as public; for example, you could have the set accessor as private so that only other methods in the same class are able to change its value.

public string FirstName { get; private set; }

While the value in the example above can still be viewed from anywhere, it can now only be changed from within the class the contains the property.

You can place any access modifier on the accessors of a property (i.e. private, protected, internal, protected internal). However, access modifiers on accessors must be more restrictive than that on the property itself (e.g. you can’t declare a private property with public accessors).

You may also restrict modifications to a property so that a value can only be set in an initializer or constructor:

public Person (string firstName) => this.FirstName = firstName;

public string FirstName { get; }

The above feature is primarly used for initializing collections that are exposed as read-only properties:

public class Measurements

{

public ICollection<DataPoint> points { get; } = new

List<DataPoint>();

}

### Computed Properties

Properties may return computed values; they are not restricted to only returning the value of a member field:

public class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

public string FullName { get {return $”{FirstName} {LastName}”; } }

}

The above example uses the $ character for string interpolations; this allows for interpolation expressions (such as {FirstName}) and so provides a more readable and convenient syntax for writing formatted strings.

You could instead use an **expression-bodied member** to achieve the same effect in a more succinct way. Expression-bodied members use the lambda expression syntax and define methods that contain a single expression only.

public class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

public string FullName => $”{FirstName} {LastName}”;

}

### Cached Evaluated Properties

You can store a computed property to create a **cached evaluated property**. For example, the FullName property in the above examples could be cached after being evaluated, meaning the string formatting will only occur the first time the property is accessed:

public class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

private string fullName;

public string FullName

{

get

{

if (fullName == null)

{

fullName = $”{FirstName} {LastName}”;

}

return fullName;

}

}

}

In practice, the above code would be impractical; while the fullName field may be accurate the first time it’s accessed, it won’t change if the first or last name is changed. For example, if the full name is “John Smith” when the FullName getter is first called, the correct name will be returned. However, if the first name is then changed to “Tom”, the fullName field won’t be updated to reflect this, meaning all future calls to the FullName getter will still return “John Smith”.

This can be amended by having the set accessors for FirstName and LastName reset the value of the fullName field to null (meaning it will be evaluated again the next time it’s called):

public class Person

{

private string firstName;

public string FirstName

{

get => firstName;

set

{

firstName = value;

fullName = null;

}

}

private string lastName;

public string LastName

{

get => lastName;

set

{

lastName = value;

fullName = null;

}

}

private string fullName;

public string FullName

{

get

{

if (fullName == null)

{

fullName = $”{FirstName} {LastName}”;

}

return fullName;

}

}

}

In this version of the code, the value of fullName is only calculated when it’s needed. Note that while these changes have made the code cleaner and more efficient, developers that use this class do not need to know the details about its implementation and so they will use the Person class in the same way regardless of which of the implementations listed above is used. This is the primary reason for using properties to expose data members of an object.

### Attaching attributes to auto-implemented properties

In auto-implemented properties, you can still attach field attributes to the compiler generated backing field by using NonSerializedAttribute; this can only be attached to fields, not properties:

[field:NonSerialized]

public int Id { get; set; }

### Implementing INotifyPropertyChanged

You may need to write code in a property accessor to support the INotifyPropertyChanged interface, which is used to notify **data binding clients** when a property’s value changes. When such a change occurs, the INotifyPropertyChanged.PropertyChanged event will be raised, meaning the data binding libraries will then be able to appropriately update display elements based on that change. This could be implemented for the FirstName property of the example Person class in the following way:

public class Person : INotifyPropertyChanged

{

public string FirstName

{

get => firstName;

set

{

if (value != firstName)

{

PropertyChanged?.Invoke(this, new

PropertyChangedEventArgs(nameof(firstName)));

}

firstName = value;

}

}

private string firstName;

public event PropertyChangedEventHandler PropertyChanged’

}

**NOTE:** The ?. operator is the **null conditional operator**; this checks for and handles NullReferenceExceptions before evaluation the right side of the operator. In the case of our example, a NullReferenceException would be thrown if the PropertyChanged event is raised while there are no subscribers; however, the ?. stops the event from ever being raised under these circumstances.

**NOTE:** The **nameof** operator is used above to convert the property name symbol to a string representation. It’s common to use nameof to avoid errors where a property name may be mistyped or changed.

## Operators and Expressions

### Operators and Expressions

C# provides many operators, most of which are supported by the built-in types (bool, int string etc.), which allow you to perform basic operations with the values of those types. The supported operators include:

* Arithmetic operators – perform arithmetic operations with numeric operands
* Comparison operators – compare numeric operands
* Boolean logical operators – perform logical operations with bool operands
* Bitwise and shift operators – perform bitwise or shift operations with operands that are one of the integral types
* Equality operators – check if the given operands are equal or not

These operators can typically be overloaded.

**NOTE:** Operator overloading is where a type provides a custom implementation of an operation in the case that one or both operands are of that type. With this, you can define custom structures to represent data in a useful way (e.g. to represent functions).

As in maths, you can change the order in which operators in an expression are performed by using parentheses. This allows you to manipulate operator precedence.

Other kinds of expressions that C# provides include:

* Interpolated string expressions – these provide a convenient syntax for creating formatted strings:

var r = 2.3;

var message = $”The area of a circle with radius {r} is {Math.Pi \* r \*

r:F3}.”;

* Lambda expressions – these allow you to create anonymous functions:

int[] numbers = {2, 3, 4, 5};

var maxSquare = numbers.Max(x => x \* x);

Console.WriteLine(maxSquare); //calculates largest square from numbers in array

* Query expressions – these allow you to use query capabilities (similar to SQL) directly in C#:

var scores = new[] {90, 97, 78, 68, 85};

IEnumerable<int> highScoresQuery =

from score in scores

where score > 80

orderby score descending

select sore;

Console.WriteLine(string.Join(“ “, highScoresQuery);

//Output: 97, 90, 85

Operator Precedence

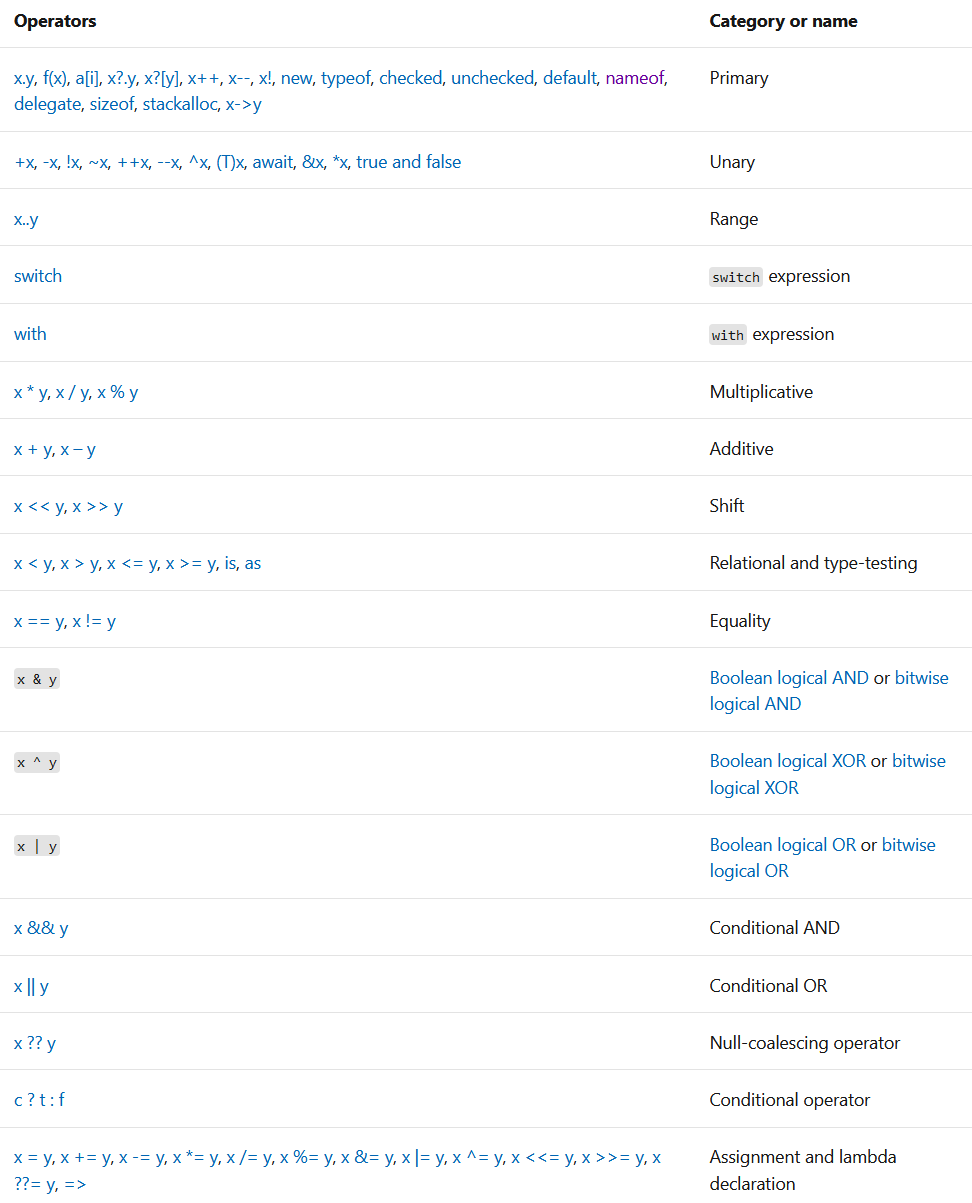
In expressions with multiple operators, operators will be evaluated in order of precedence. In mathematical operations, the BODMAS precedence order applies, meaning that in a sum with both addition and multiplication, the multiplication will be performed first:

var a = 2 + 2 \* 2; //a = 6

As with BODMAS, you can change the order of precedence using brackets:

var a = (2+2) \* 2; //a = 8

The following table lists the C# operators in order of precedence from highest to lowest:



### Operator Associativity

When two operators in an expression have the same precedence, associativity determines the order in which they will be performed.

* Left-associative operators are evaluated from left to right. Excluding assignment operators and null-coalescing operators, all binary operators are left-associative:  
  i.e. a + b + c will be calculated as (a + b) + c
* Right-associative operators are evaluated from right to left. Assignment operators, null-coalescing operators and the ?: conditional operator are right-associative:  
  i.e. x = y = z will be calculated as x = (y = z)

### Operand Evaluation

Operands in an expression are evaluated from left to right, ignoring operator precedence and associativity. For example:

|  |  |
| --- | --- |
| **Expression** | **Order of Evaluation** |
| a + b | a, b, + |
| a + b \* c | a, b, c, \*, + |
| a / b + c \* d | a, b, /, c, d, \*, + |
| a / (b + c) \* d | a, b, c, +, /, d, \* |

While all operands are usually evaluated, some are only evaluated conditionally; certain operators may cause only certain operands to be evaluated. These operands are: the logical AND and OR, null-coalescing operators ?? and ??=, null-conditional operators ?. and ?[], and conditional operator ?:.

## Statements

### Statements

Statements are the actions that a program takes. They encompass many things in C# code, but common actions include declaring variables, assigning values, calling methods, looping through collections and branching to other code blocks given certain conditions.

The order in which these statements are executed is known as the flow of control/flow of execution. The flow of control may vary each time a program is run, as different user inputs can cause different conditions to be met, causing branches to lead to other code blocks.

Statements can consist of single lines of code that end in a semi-colon or a series of single-line statements in a block. Statement blocks are enclosed in {} brackets and can contain nested statement blocks.

The following code demonstrates various types of statements and their names:

public class SimpleStatements

{

static void Main()

{

int counter; //declaration statement

counter = 1; //assignment statement

//declaration statements with initializers

int[] radii = { 15, 32, 108, 74, 9 };

const double pi = 3.14159;

//foreach statement block

foreach (int radius in radii)

{

double circumference = pi \* (2 \* radius);

//expression statement (method invocation)

System.Console.WriteLine(“Radius of circle #{0} is {1}.

Circumference = {2:N2}”, counter, radius,

circumference);

//expression statement (postfix increment)

counter++;

} //end of foreach statement block

} // end of Main method body

} // end of SimpleStatements class

### Types of Statements

## Attributes

### Attributes

## Expression-Bodied Members

### Expression-Bodied Members