Agenda

- Fundamentals
- Operators
- Flow Controls

C# Source File Structure (1 of 6)

Declaration order

Using statement

Used to reference namespace.

When class names are to be referred in the *using* directive, aliases for the classes can be used.

using *alias-name = namespace.class-name*

2. Namespace declaration

Namespaces are a way of grouping types names and reducing the chance of name collisions and can contain both namespaces and other types.

In C# you need to declare each class in a namespace. By default , namespace is automatically created with the same name as that of the project. CSharpSchool is the namespace and CSharpOne class is contained in the namespace.

3. Class declaration

A C# source file can have several classes but only one class can have the **Main** method.

```
/*
    * Created on April 11, 2011
    * First C# Program
*/
using System;
using A = System.Console;

namespace CSharpSchool
{
    public class CSharpOne
      {
          /// <summary>
          /// </summary>
          /// <param name="args"></param>
          public static void Main(string[] args )
            {
              // print a message
                 A.WriteLine("Welcome to C#!");
            }
      }

}
```

C# Source File Structure (2 of 6)

Comments

1. Single-line Comment

```
// insert comments here
```

2. Multi-line Comment

```
/*
 insert comments here
 */
```

3. Documentation Comment

```
///<summary>
///insert documentation
///<summary>
```

Whitespaces

Tabs and spaces are ignored by the compiler. They are used to improve the readability of code.

```
/*
    * Created on April 11, 2011
    * First C# Program
 */
using System;
using A = System.Console;

namespace CSharpSchool
{
    public class CSharpOne
      {
        /// <summary>
        /// </summary>
        /// <param name="args"></param>
        public static void Main(string[] args )
          {
            // print a message
              A.WriteLine("Welcome to C#!");
          }
      }

}
```

C# Source File Structure (3 of 6)

Class

Every C# program includes at least one class definition. The class is the fundamental component of all C# programs. `class` is a keyword. **CSharpOne** is a C# identifier that specifies the name of the class to be defined

A class is a construct that enables you to create your own custom types by grouping together variables of other types, methods and events.

Class body indicated by the opening and closing braces.

```
  /*
      * Created on April 11, 2011
      * First C# Program
 */
using System;
using A = System.Console;

namespace CSharpSchool
{
      public class CSharpOne
        {
              /// <summary>
              /// </summary>
              /// <param name="args"></param>
              public static void Main(string[] args )
                {
                   // print a message
                       A.WriteLine("Welcome to C#!");
                }
         }

}
```

C# Source File Structure (4 of 6)

Braces

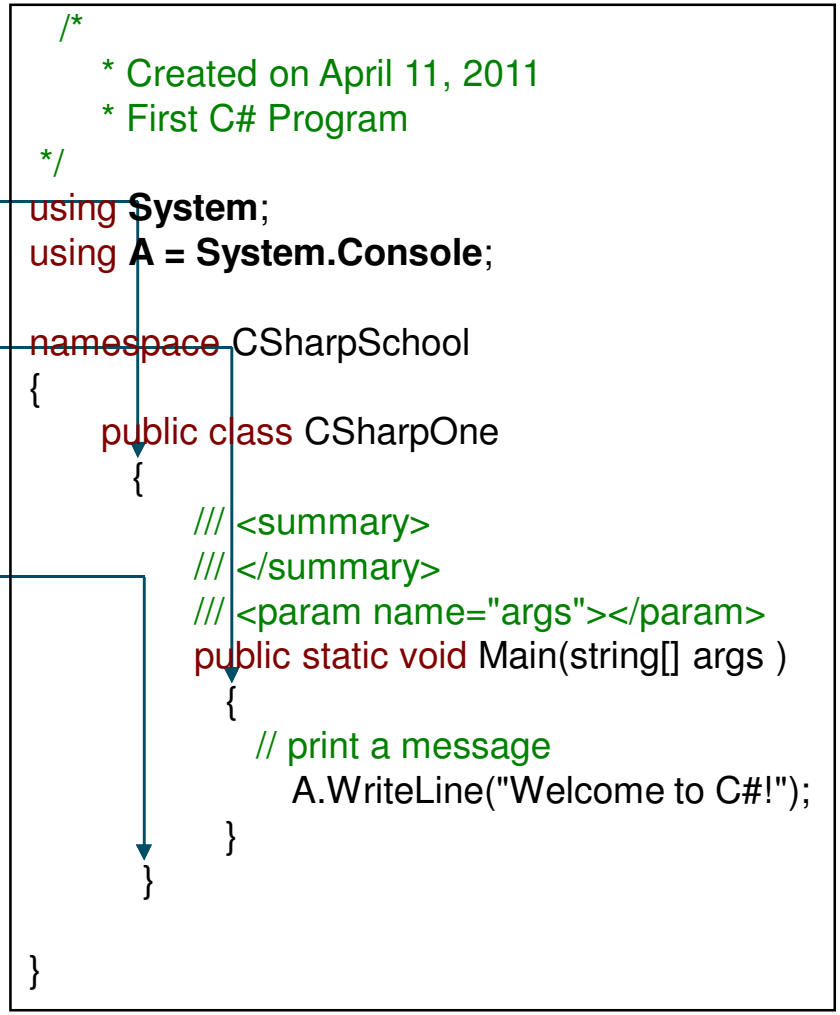
Braces are used for grouping statements or block of codes.

The left brace (**{**) indicates the beginning of a class body, which contains any variables and methods the class needs.

The left brace also indicates the beginning of a method body.

For every left brace that opens a class or method you need a corresponding right brace (**}**) to close the class or method.

A right brace always closes its nearest left brace.



```
/*
    * Created on April 11, 2011
    * First C# Program
*/
using System;
using A = System.Console;

namespace CSharpSchool
{
    public class CSharpOne
    {
        /// <summary>
        /// </summary>
        /// <param name="args"></param>
        public static void Main(string[] args )
        {
            // print a message
            A.WriteLine("Welcome to C#!");
        }
    }

}
```

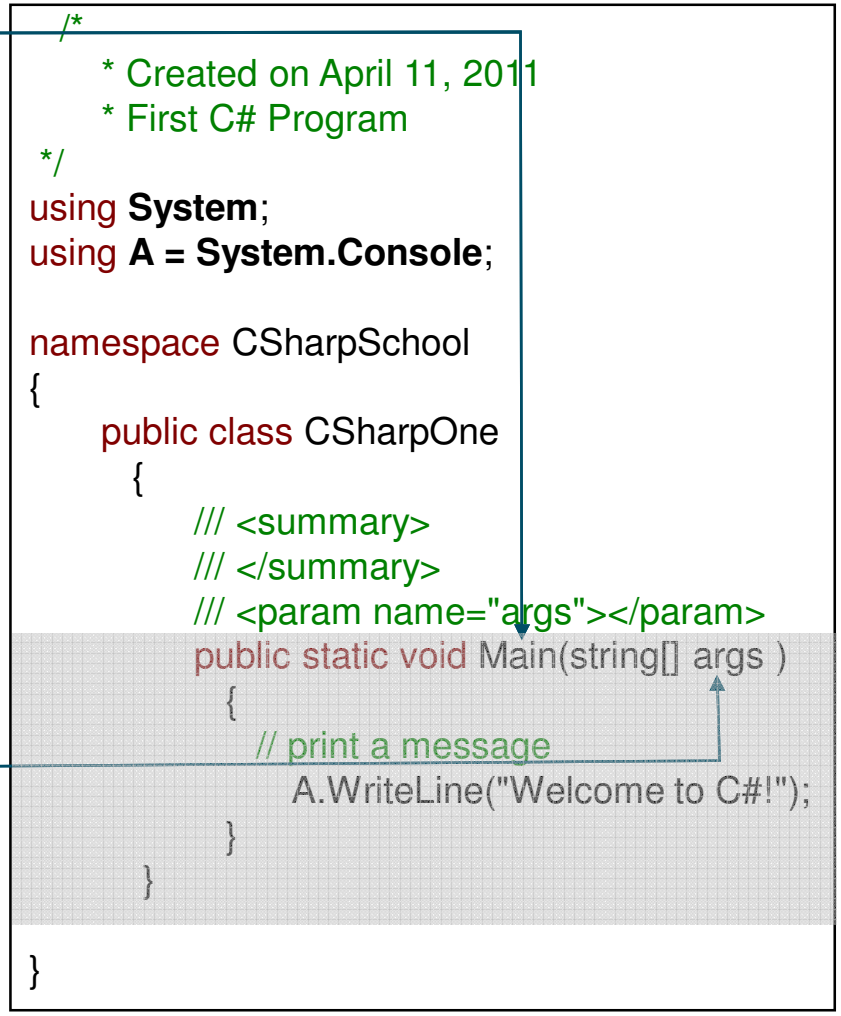
C# Source File Structure (5 of 6)

`Main()` method

This line begins the `Main()` method. This is the line at which the program will begin executing.

`string args[]`

Declares a parameter named `args`, which is an array of string. It represents command-line arguments.



C# Source File Structure (6 of 6)

C# statement

- A complete unit of work in a C# program.
- A statement is always terminated with a semicolon and may span multiple lines in your source code.

`Console.WriteLine();`

This line outputs the string "`Welcome to C#!`" followed by a new line on the screen.

Terminating character

Semicolon (`;`) is the terminating character for any C# statement.

```
/*
    * Created on April 11, 2011
    * First C# Program
 */
using System;
using A = System.Console;

namespace CSharpSchool
{
    public class CSharpOne
      {
          /// <summary>
          /// </summary>
          /// <param name="args"></param>
          public static void Main(string[] args )
           {
            // print a message
                A.WriteLine("Welcome to C#!");
           }
      }

}
```

C# Keywords

- They are an essential part of language definition as they implement specific features of the language.
- They are reserved, and cannot be used as identifiers.

abstract	as	base	bool	break	byte
case	catch	char	checked	class	const
continue	decimal	default	delegate	do	double
else	enum	event	explicit	extern	false
finally	fixed	float	for	foreach	goto
if	implicit	in	int	interface	internal
is	lock	long	namespace	new	null
object	operator	out	override	params	private
protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	struct
switch	this	throw	true	try	typeof
uint	ulong	unchecked	unsafe	ushort	using
virtual	volatile	void	while	string	

C# Keywords (cont.)

- The following keywords are contextual. They can be used as an identifier without an @symbol.

from	**get**	**global**	**descending**	**dynamic**	**equals**
join	**set**	**On**	**orderby**	**in**	**into**
select	**let**	**Value**	**group**	**partial**	**remove**
add	**ascending**	**by**	**var**	**where**	**yield**

Identifiers

- An *identifier* is the name given by a programmer to a variable, statement label, method, class, and interface.
  - An identifier must begin with a letter.
  - Subsequent characters must be letters, digits, or _ (underscore).
  - An identifier must not be a C# keyword.
  - Identifiers are case-sensitive.
  - Keywords can be used as identifiers when they are prefixed with the '@' character

Incorrect	*Correct*
`3strikes`	**`strikes3`**
`Write&Print`	**`Write_Print`**
`switch`	**`Switch`**

`printMe` is not the same as `PrintMe`

Literals

- A method of representing values that are stored in variables.
- C# Literals:
  - Numeric Literals:
    - Integer Literals
    - Real Literals
  - Boolean Literals
  - Character Literal:
    - Single Character Literals
    - String Literals

Variable and Data Types

- A variable is a named storage location used to represent data that can be changed while the program is running.
- A data type:
  - Determines the values that a variable can contain and the operations that can be performed on it.
  - Categories include:
    - Values
    - References
    - Pointers (used only in unsafe code)

Value Data Types

- Numeric types are categorized in to 3 types:
  - Integral types
  - Floating-point types
  - Decimal
- The following table shows the sizes and ranges of the integral types, which constitute a subset of simple types.

Type	Range	Size
sbyte	-128 to 127	Signed 8-bit integer
byte	0 to 255	Unsigned 8-bit integer
char	U+0000 to U+ffff	Unicode 16-bit character
short	-32,768 to 32,767	Signed 16-bit integer
ushort	0 to 65,535	Unsigned 16-bit integer
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer
uint	0 to 4,294,967,295	Unsigned 32-bit integer
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer

Value Data Types(Cont..)

- The following table shows the precision and approximate ranges for the Floating-point types and Decimal type.

Type	Approximate range	Precision
Float	±1.5e−45 to ±3.4e38	7 digits
double	±5.0e−324 to ±1.7e308	15-16 digits
decimal	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$	28-29 significant digits

Value Data Types(Cont..)

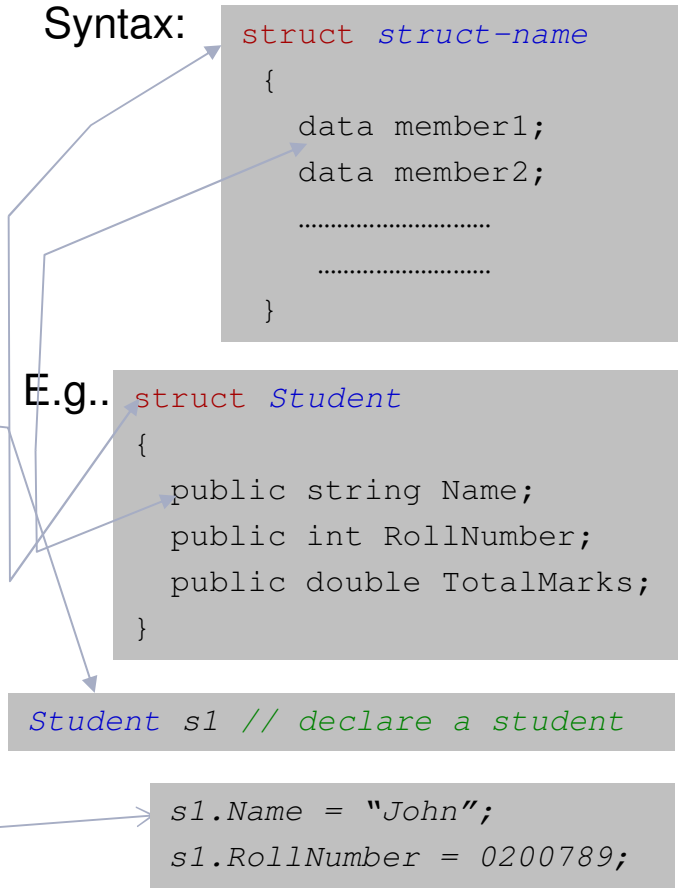
- Floating–points:
  - Used to hold numbers containing fractional parts.
  - Two types:
    - float (single precision numbers)
    - double (double precision numbers)
    - Support a special value known as Not-a-Number (NaN). NaN is used to represent results of operations such as dividing zero by zero, where an actual number is not produced.
- Decimal Type:
  - A high precision 128-byte data type, designed for use in financial and monetary calculations.
  - Can store values in the range 1.0 × 10e–28 to ±7.9 × 10e28:
    - To specify a number to be decimal type, append the character M (or m) to the value e.g 123.45M.

Value Data Types (cont.)

- Boolean Types:
  - Are declared using the keyword, bool.
  - Have two values: true or false. In languages, such as C and C++, boolean conditions can be satisfied where 0 means false and any other value means true. In C# the only values that satisfy a boolean condition is true and false, which are official keywords.
- Character Types:
  - Are declared using the keyword char.
  - char type assumes a size of two bytes but can hold only a single character.
  - Are designed to hold a 16–bit Unicode character.

Value Data Types (cont.)

- Structures (*structs*):
  - Are similar to classes.
  - Are used for simple composite data types.
  - `struct` keyword is used to declare structures.
  - Variables are known as *members* or *fields* or *elements*.
  - s1 is a variable type of structure Student.
  - Member variables can be accessed using dot notation.



Value Data Types (cont.)

- Enumerations are a user-defined integer type that provide a way to attach names to numbers.
- They help increase comprehensibility in the code.
- The `enum` keyword is used to define an enumeration. A list of words in an `enum` is separated by a comma and automatically assigned values of 0, 1, 2, etc.
- The default value of the first `enum` member is set to 0 while each subsequent member is incremented by one. It can be changed by assigning specific values to the members.

Syntax:

```
enum enum-name {word1, word2, word3 }
```

E.g.

```
enum Shape {Red,Blue,Green,Yellow}
```

```
enum Shape
{
  Red = 10,
  Blue = 20,
  Green = 100
}
```

Value Data Types (cont.)

- Default Values
  - Variables are either explicitly assigned a value or automatically assigned a default value.

Type	Default Values
All integer types	0
char type	'\x000'
float type	0.0f
double type	0.0d
decimal type	0.0m
bool type	false
enum type	0
All reference types	null

Constant Variables

- Values Variables whose values do not change during execution of a program:
  - Use the `const` keyword to initialize.
  - Constants must be declared and initialized simultaneously.
  - Constants can be initialized using an expression.
  - Constants cannot use non-const values in an expression.

```
const int age = 21;
```

```
const int ;
age = 21;
```

Is illegal

```
const int m  = 10;
const int age = m * 5
```

```
int m = 10;
const int age = m * 5
```

Error

Reference Data Types

- Reference data types represent objects.
- A reference serves as a handle to the object, it is a way to get to the object.
- C# reference data types are divided into two types:
  - User-defined (or complex) types:
    - Class
    - Interfaces
    - Delegates
    - Arrays
  - Predefined (or simple) types:
    - Object type
    - String type

Variable Declaration and Initialization

- To declare a variable with value data type:

```
int     age  =  21;
```

primitive type — identifier name — initial value

- To declare a variable with reference data type:

```
Box   b1 = new Box();
string    name = "Jason";
```

reference type — identifier name — initial value

Value Type Declaration

declaration

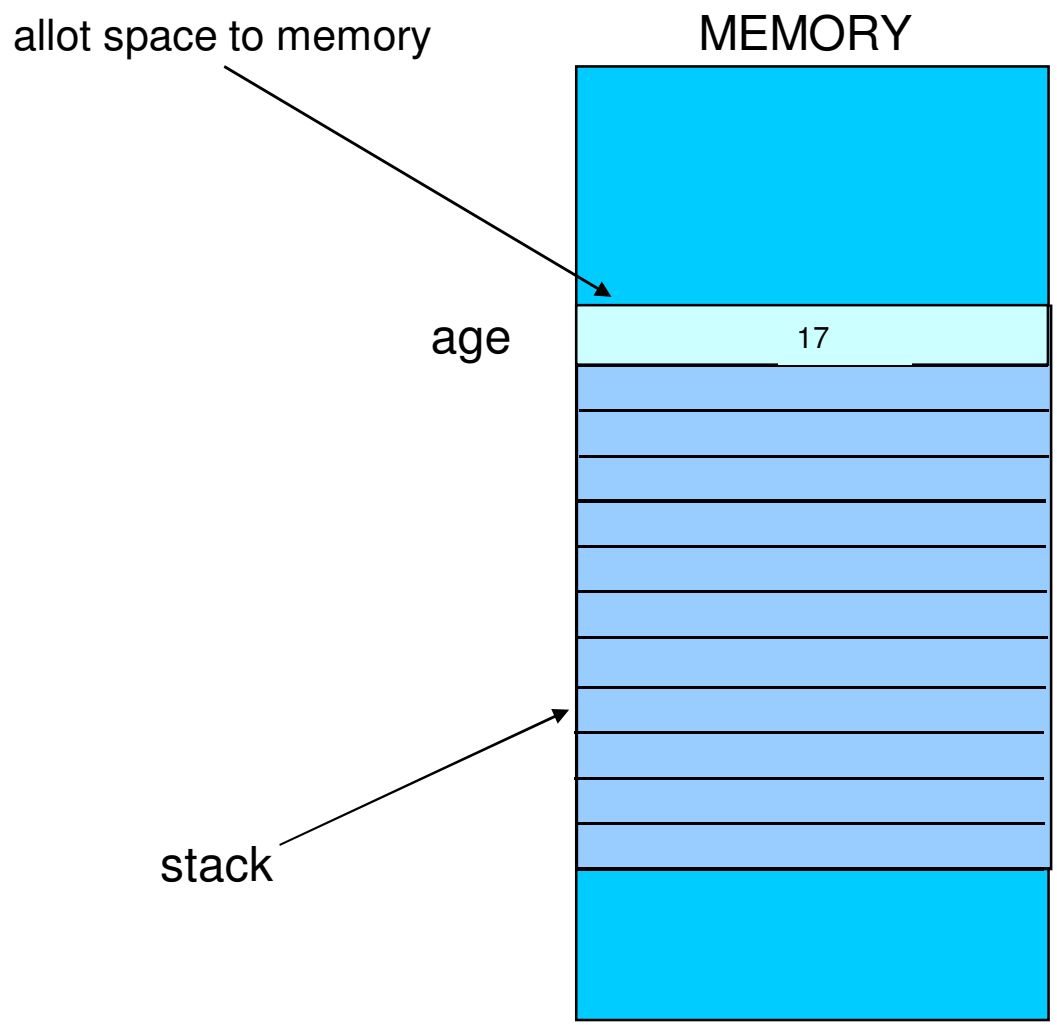
int age;

type Identifier name

initialization/assignment

age = 17;

Identifier name value

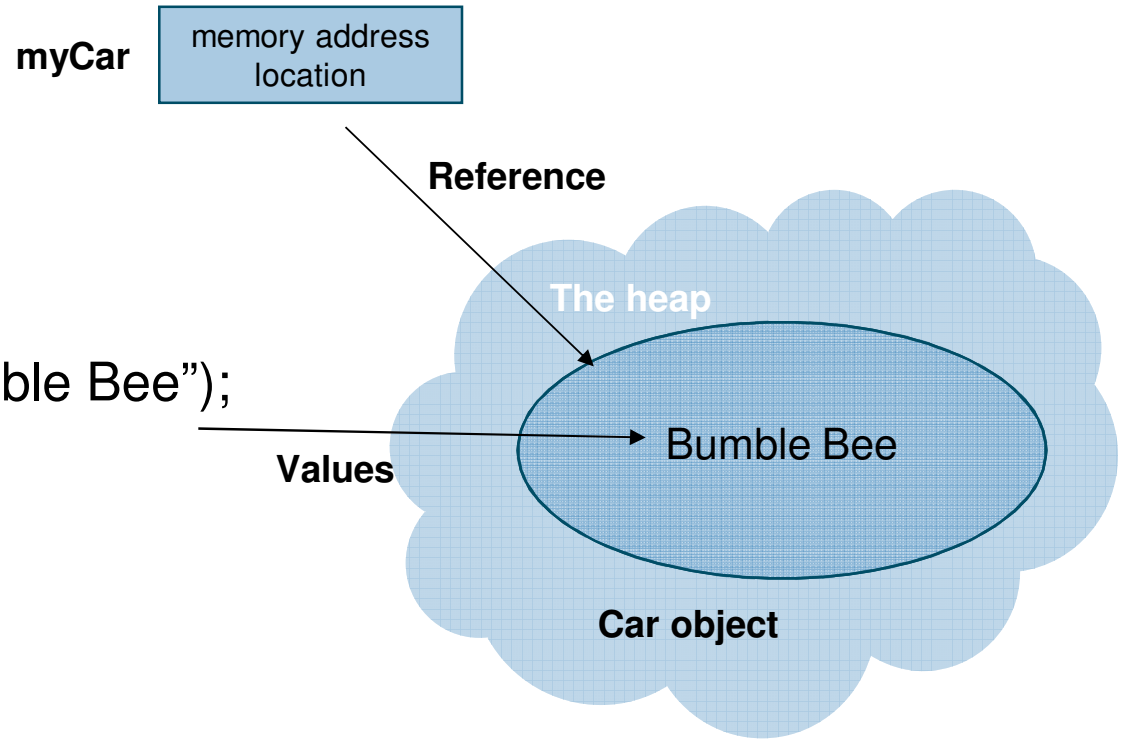


Reference Type Declaration

Car myCar;

type Identifier name

allot space to memory

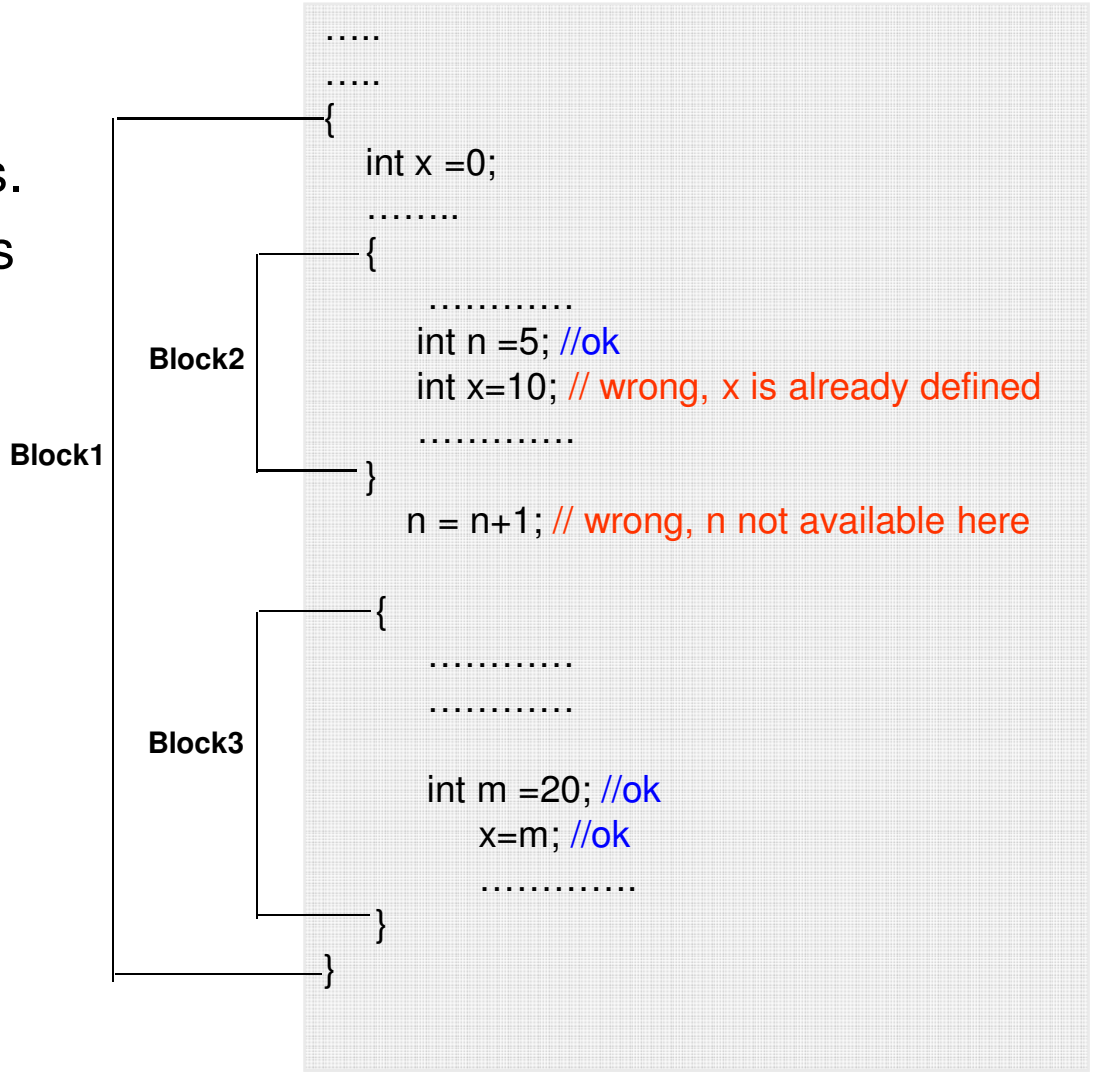


myCar = new Car("Bumble Bee");

Identifier name

Scope of Variable

- Member Variables:
  - Declared inside the class but outside of all methods.
  - Accessible by all methods of the class.
- Local Variables:
  - Available only within the method where they were declared.
  - Method parameters have local scope.



Boxing and UnBoxing

- Boxing:
  - Is a data type conversion technique that is used implicitly to convert a value type to either an object type or a reference type.

```
public static void Main( )
    {
      int x = 10;
      object obj1 =  x;
       x = 456;
      Console.WriteLine(x);
      Console.WriteLine(obj1);
    }
}
```

- UnBoxing:
  - Is the opposite of boxing. It is a data type conversion technique that is used explicitly to convert an object type to a value type.

```
public static void Main( )
    {
      int x = 10;
      object obj1 =  x;
      int y = (int) obj1;
      Console.WriteLine(y);
    }
}
```

Operators and Assignments

- Unary operators
- Arithmetic operators
- String operators
- Relational operators
- Conditional operators
- Logical operators
- Assignment operators
- Bitwise operators
- Primitive Casting

Unary Operators

- Unary operators use only one operand.

++	Increment by 1, can be prefix or postfix
--	Decrement by 1, can be prefix or postfix
+	Positive sign
-	Negative sign

Sample code:

```
int num=10;

System.Console.WriteLine("incrementing/decrementing...");
System.Console.WriteLine(++num);
System.Console.WriteLine(--num);
System.Console.WriteLine(num++);
System.Console.WriteLine(num--);

System.Console.WriteLine("setting signs...");
System.Console.WriteLine(+num);
System.Console.WriteLine(-num);
```

Sample output:

```
incrementing/decrementing...
11
10
10
11
setting signs...
10
-10
```

Arithmetic Operators

- Arithmetic operators are used for basic mathematical operations.

+	Add
–	Subtract
*	Multiply
/	Divide
%	Modulo, remainder

Sample code:

```
int num1=15, num2=10;

System.Console.WriteLine("calculating...");
System.Console.WriteLine(num1 + num2);
System.Console.WriteLine(num1 - num2);
System.Console.WriteLine(num1 * num2);
System.Console.WriteLine(num1 / num2);
System.Console.WriteLine(num1 % num2);
```

Sample output:

```
calculating...
25
5
150
1
5
```

String Operators

- The string operator (+) is used to concatenate operands.
- If one operand is string, the other operands are converted to string.

Sample code:

```
string fname = "Henry" ;
string lname = "Ford";
string mi = "D";
string fullName = lname + ", " + fname + " " + mi + ".";
string nickName = "Henry";
int age = 21;

System.Console.WriteLine("My full name is: " + fullName);
System.Console.WriteLine("You can call me " + nickName + "!");
System.Console.WriteLine("I'm " + age + " years old.");
```

Sample output:

```
My full name is: Ford, Henry D.
You can call me Henry!
I'm 21 years old.
```

Relational Operators (1 of 2)

- Relational operators are used to compare values.
- `boolean` values cannot be compared with non-`boolean` values.
- Only object references are checked for equality, and not their states.
- Objects cannot be compared with `null`.
- `null` is not the same as “”.

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equals
!=	Not equals

Relational Operators (2 of 2)

- Sample code:

```
string name1 = "Marlon"; int weight1=140, height1=74;
string name2 = "Katie"; int weight2=124, height2=78;

boolean isLight = weight1 < weight2, isLightEq = weight1 <= weight2;
System.Console.WriteLine("Is " + name1 + " lighter than " + name2 + "? " + isLight);
System.Console.WriteLine("Is " + name1 + " lighter or same weight as " + name2 + "? " +
    isLightEq);
boolean isTall = height1 > height2, isTallEq = height1 >= height2;
System.Console.WriteLine("Is " + name1 + " taller than " + name2 + "? " + isTall);
System.Console.WriteLine("Is " + name1 + " taller or same height as " + name2 + "? " +
    isTallEq);
boolean isWeighEq = weight1 == weight2, isTallNotEq = height1 != height2;
System.Console.WriteLine("Is " + name1 + " same weight as " + name2 + "? " + isWeighEq);
System.Console.WriteLine("Is " + name1 + " not as tall as " + name2 + "? " + isTallNotEq);
System.Console.WriteLine("So who is heavier?");
System.Console.WriteLine("And who is taller?");
```

Sample output:

```
Is Marlon lighter than Katie? false
Is Marlon lighter or same weight as Katie? false
Is Marlon taller than Katie? false
Is Marlon taller or same height as Katie? false
Is Marlon same weight as Katie? false
Is Marlon not as tall as Katie? true
So who is heavier?
And who is taller?
```

Conditional Operators

Syntax:

```
exp1 ? exp2 : exp3;
```

- The ternary operator (`?:`) provides a handy way to code simple `if-else()` statements in a single expression, it is also known as the conditional operator.
- The operator `?:` works as follows: `exp1` is evaluated first
  - If `exp1` is `true`, then `exp2` is returned as the result of operation.
  - If `condition` is `false`, then `exp3` is returned as the result of operation.
- It can be nested to accommodate chain of conditions.

Sample code:

```
int yyyy=1981;
int mm=10;
int dd=22;
String mmm = mm==1?"Jan":mm==2?"Feb":mm==3?"Mar":mm==4?"Apr":mm==5?"May":mm==6?"Jun":
    mm==7?"Jul":mm==8?"Aug":mm==9?"Sep":mm==10?"Oct":mm==11?"Nov":mm==12?"Dec":"Unknown";
System.Console.WriteLine("I was born on " + mmm + " " + dd + ", " + yyyy);
```

Sample output:

```
I was born on Oct 22, 1981
```

Relational Operators

- Logical operators are used to compare `boolean` expressions.
- ! inverts a boolean value.
- `&` `|` evaluate both operands.
- `&&` `||` evaluate operands conditionally.
- `&&` `||` are used to form compound conditions by combining two or more relations.

!	logical NOT
&	bitwise logical AND
|	bitwise logical OR
^	bitwise logical NOT
&&	logical AND
||	logical OR

Truth Table:

Op1	Op2	!Op1	Op1 & Op2	Op1 | Op2	Op1 ^ Op2	Op1 && Op2	Op1 || Op2
false	false	true	false	false	false	false	false
false	true	true	false	true	true	false	true
true	false	false	false	true	true	false	true
true	true	false	true	true	false	true	true

Logical Operators

Sample output:

```
Are you a candidate for promotion? true
Will you be promoted as a regular employee? false
Will you be promoted as a supervisor? false
Will you be promoted as a manager? true
Will you be paid more and work less? false
I hope you won't be demoted, are you? false
```

Sample code:

```
int yrsService=8;
double perfRate=86;
double salary=23000;
char position='S';
// P-probationary R-regular, S-supervisor, M-manager, E-executive, T-top executive

boolean forRegular, forSupervisor, forManager, forExecutive, forTopExecutive;
forRegular = yrsService>1 & perfRate>80 & position=='P' & salary<10000;
forSupervisor = yrsService>5 & perfRate>85 & position=='R' & salary<15000;
forManager = yrsService>7 & perfRate>85 & position=='S' & salary<25000;
forExecutive = yrsService>10 & perfRate>80 & position=='M' & salary<50000;
forTopExecutive = yrsService>10 & perfRate>80 & position=='E' & salary<75000;
boolean isPromoted = forRegular||forSupervisor||forManager||forExecutive||forTopExecutive;
boolean isLuckyGuy = forExecutive ^ forTopExecutive;

System.Console.WriteLine("Are you a candidate for promotion? " + isPromoted);
System.Console.WriteLine("Will you be promoted as a regular employee? " + forRegular);
System.Console.WriteLine("Will you be promoted as a supervisor? " + forSupervisor);
System.Console.WriteLine("Will you be promoted as a manager? " + forManager);
System.Console.WriteLine("Will you be paid more and work less? " + isLuckyGuy);
System.Console.WriteLine("I hope you won't be demoted, are you? " + !isPromoted);
```

Assignment Operators

- Assignment operators are used to set the value of a variable.

=	Assign
+=	Add and assign
-=	Subtract and assign
*=	Multiply and assign
/=	Divide and assign
%=	Modulo and assign
&=	AND and assign
|=	OR and assign
^=	XOR and assign

Sample code:

```
double unitPrice=120, qty=2, salesAmount;
double discRate=15, discAmount, vatRate=10, vatAmount;

// compute gross sales
salesAmount = unitPrice * qty;
System.Console.WriteLine("Gross Sales: " + salesAmount);
// compute tax
vatRate /= 100;
vatAmount = salesAmount * vatRate;
salesAmount += vatAmount;
System.Console.WriteLine("Tax: " + vatAmount);
// compute discount
discRate /= 100;
discAmount = salesAmount * discRate;
salesAmount -= discAmount;
System.Console.WriteLine("Discount: " + discAmount);
System.Console.WriteLine("Please pay: " + salesAmount);
```

Sample output:

```
Gross Sales: 240.0
Tax: 24.0
Discount: 39.6
Please pay: 224.4
```

Casting (Type Conversion)

- Casting is conversion from one data type to another which include:
  - Implicit casting
  - Explicit casting
- Implicit Conversion is the conversion of one data type to another data type without any loss of data.

Data Type	Permissible Implicit Data Type Conversion
int	decimal, long, double, and float
long	decimal, double and float
short	int, long, decimal, double, and float
sbyte	short,int,long,decimal,double,and float
byte	int, uint, long, ulong, short, ushort, decimal, double, and float
ushort	int, ulong, decimal, double, and float
uint	long, ulong, decimal, double, and float
ulong	decimal, double, and float
float	double
char	int, uint, long, ulong, ushort, decimal, double, and float

Casting (Type Conversion) (cont.)

- Explicit Conversion is the conversion of one data type to another data type with loss of data.

Data Type	Permissible Explicit Data Type Conversion
int	uint, byte, short, ushort, char, and ulong
long	int, uint, byte, sbyte, short, ushort, char, and ulong
short	uint, ushort, byte, sbyte, char, and ulong
sbyte	byte, uint, ulong, char, and ushort
byte	char and sbyte
ushort	short, byte, sbyte, and char
uint	int, short, ushort, byte, sbyte, and char
ulong	byte, sbyte, int, uint, short, ushort, long, and char
float	int, uint, long, ulong, char, decimal, short, ushort, byte, and sbyte
double	int, uint, short, ushort, byte, sbyte, long, ulong, char, float, and double
decimal	byte, sbyte, int, uint, short, ushort, ushort, long, ulong, char, float, and double
char	short, byte, and sbyte

- Explicit conversions can be carried out using the 'cast' operator.

Syntax:

```
Type variable1 = (type) variable2;
```

E.g..

```
float amount = 50;
long totAmount = (long) amount;
```

```
int m = 50;
Byte n = (byte) m;
```

Summary of Operators

- Evaluation order of operators in C# is as follows:
  - Unary (++ -- + - ~ ())
  - Arithmetic (* / % + -)
  - Shift (<< >> >>>)
  - Comparison (< <= > >= == !=)
  - Bitwise (& ^ |)
  - Logical Operators (&& || !)
  - Conditional (?:)
  - Assignment (= += -= *= /=)

Flow Controls

- `if-else()` statement
- `switch()` statement
- `while()` statement
- `do-while()` statement
- `for()` statement
- `foreach()` statement
- `break` statement
- `goto` statement
- `continue` statement
- `label` statement

if-else()

- `if-else()` performs statements based on two conditions.
- `condition` should result to a `boolean` expression.
- If `condition` is true, the statements following `if` are executed.
- If `condition` is false, the statements following `else` are executed.
- `if-else()` can be nested to allow more conditions.

Syntax:

```
if (condition) { // braces optional
    // statement required
}
else {    // else clause is optional
    // statement required
}
```

Example:

```
int age=10;
if (age < 10) {
    System.Console.WriteLine("You're just a kid.");
}else if (age < 20){
    System.Console.WriteLine("You're a teenager.");
} else {
    System.Console.WriteLine("You're probably
    old...");
}
```

Output:

```
You're a teenager.
```

switch()

- `switch()` performs statements based on multiple conditions.
- `exp` can be `char byte short int`, `val` should be a unique constant of exp.
- `case` statements falls through the next `case` unless a `break` is encountered.
- `default` is executed if none of the other cases match the `exp`.

Syntax:

```
switch (exp) {
case val:
     //  statements here
case val:
     //  statements here
default:
     //  statements here
}
```

Example:

```
char sex='M';
switch (sex){
    case 'M':
         System.Console.WriteLine("I'm a
    male."); break;
    case 'F':
          System.Console.WriteLine("I'm a
    female."); break;
    default:
          System.Console.WriteLine("I am what I
    am!"); break;
}
```

Output:

```
I'm a male.
```

switch() (cont.)

- C# does **not** allow automatic *"fall-through" (where the control moves to the next case block with no break statement).*
- *"Fall-through"* is allowed only if the case block is **empty**.
- For two consecutive case blocks to be executed continuously, we have to force the process by using the `goto` statement.

Error in the below code

```
switch (m){
    case 1:
          x = y;
    case 2:
          x = y + m;
    default:
          x = y - m;
}
```

Error in the below code

```
switch (m){
    case 1:
    case 2:
          x = y + m;
    default:
          x = y - m;
}
```

```
switch (m){
    case 1:
         x = y;
         goto case 2;
    case 2:
          x = y + m;
    default:
          x = y - m;
}
```

while()

- `while`() performs statements repeatedly while `condition` remains true.

Syntax:

```
while (condition) { // braces optional
    // statements here
}
```

Example:

```
int ctr=10;
while (ctr > 0) {
    System.Console.WriteLine("Timer: " + ctr--);
}
```

Output:

```
Timer: 10
Timer: 9
Timer: 8
Timer: 7
Timer: 6
Timer: 5
Timer: 4
Timer: 3
Timer: 2
Timer: 1
```

do-while()

- `do-while()` performs statements repeatedly (at least once) while `condition` remains true.

Syntax:

```
do
    // statements here
while (condition);
```

Example:

```
int ctr=0;
do
    System.Console.WriteLine("Timer: " + ctr++);
while (ctr < 10);
// next statement
```

Output:

```
Timer: 0
Timer: 1
Timer: 2
Timer: 3
Timer: 4
Timer: 5
Timer: 6
Timer: 7
Timer: 8
Timer: 9
```

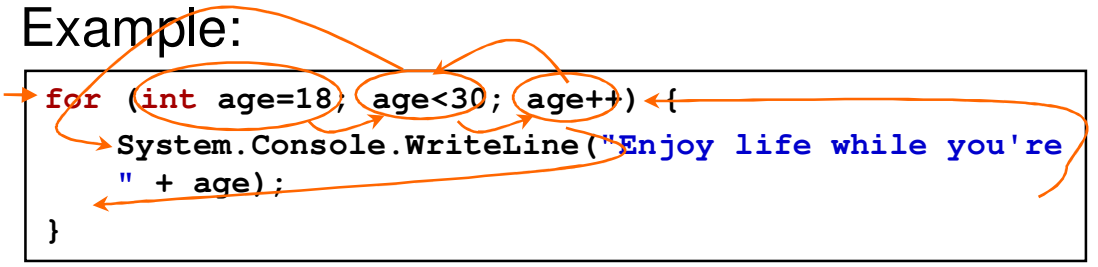
for()

- `for()` performs statements repeatedly based on a condition.
- `init` is a list of either declarations or expressions evaluated first and only once.
- `condition` is evaluated before each iteration.
- `exp` is a list of expressions evaluated after each iteration.
- All entries inside () are optional, `for(;;)` is an infinite loop.

Syntax:

```
for (init; condition; exp) { // braces optional
    // statements here
}
```

Example:



```
for (int age=18; age<30; age++) {
    System.Console.WriteLine("Enjoy life while you're
    " + age);
}
```

Output:

```
Enjoy life while you're 18
Enjoy life while you're 19
Enjoy life while you're 20
Enjoy life while you're 21
Enjoy life while you're 22
Enjoy life while you're 23
Enjoy life while you're 24
Enjoy life while you're 25
Enjoy life while you're 26
Enjoy life while you're 27
Enjoy life while you're 28
Enjoy life while you're 29
```

foreach()

- `foreach`() is similar to a `for` statement but implemented differently.
- `type and variable` declare the iteration variable. During execution, the iteration variable represents the array element (or collection element in case of collections) for which an iteration is currently being performed.
- `in` is a keyword.
- `exp` must be an array or collection type and an explicit conversion must exist from the element type of the collection to the type of the iteration variable.

Syntax:

```
foreach (type variable in exp) { //
    braces optional
    // statements here
}
```

Example:

```
int[] arrayint ={11,22,33,44};

 foreach(int m in arrayint)
  {
   System.Console.WriteLine(" " + m);
  }
   System.Console.WriteLine( );
```

Output:

```
11
22
33
44
```

break

- `break` exits loops and `switch()` statements.

Syntax:

```
break;
```

Example:

```
boolean isEating=true;
int moreFood=5;

while (isEating) {
    if (moreFood<1) break;
    System.Console.WriteLine("Uhm, yum, yum...");
    moreFood--;
}
System.Console.WriteLine("Burp!");
```

Output:

```
Uhm, yum, yum...
Uhm, yum, yum...
Uhm, yum, yum...
Uhm, yum, yum...
Uhm, yum, yum...
Burp!
```

continue

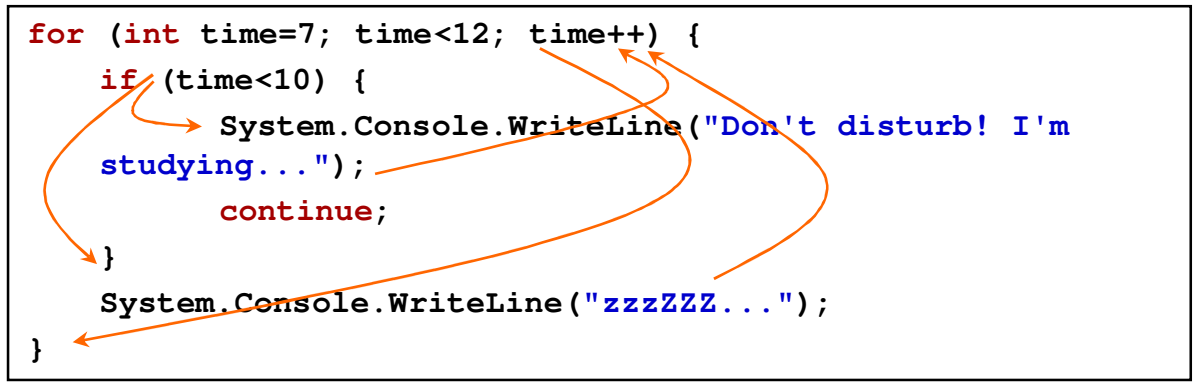
- `continue` is used inside loops to start a new iteration.

Syntax:

```
continue;
```

Example:

```
for (int time=7; time<12; time++) {
    if (time<10) {
        System.Console.WriteLine("Don't disturb! I'm
    studying...");
        continue;
    }
    System.Console.WriteLine("zzzZZZ...");
}
```



Output:

```
Don't disturb! I'm studying...
Don't disturb! I'm studying...
Don't disturb! I'm studying...
zzzZZZ...
zzzZZZ...
```

label and goto

- label and goto are used in combination.
- Labels can be used anywhere in the program and goto is used inside loops to start a new iteration.

Syntax:

```
Goto labelname;
```

Syntax:

```
labelname :
```

Example:

```
for (int i =0; i<10 ; i++)
{
   while (x<5)
     {
        y = i * x;
        Console.WriteLine(y);
            if (y>10)
                goto out1;
         x = x + 1;
     }
 }
out1:
  Console.WriteLine ("Out of loop");
```

Output:

```
1
2
3
4
5
2
4
6
8
10
3
6
9
12
Out of loop
```

return

- The return branching statement is used to exit from the current method. There are two forms:
  - return <value>;
  - return;

Example 1:

```
public int sum(int x, int y) {
        return x + y;
}
```

Example 2:

```
public int sum(int x, int y) {
        x = x + y;
        if (x < 100){
             return x;
        }else{
             return x + 5;
        }
}
```

Example 2:

```
public void getSum(int x) {
        System.Console.WriteLine(x);
return;
}
```