

Intrinsic (Simple) Data Types

- Integral
 - char examples: 'a' '3' '!'
 - short, int, long examples: 3 -5
 - bool examples: true false 1 0
- Floating-point
 - double examples: -4.1 5.23 3.12E5
 - float examples: -4.1f 5.23f

String Type

literals: examples: "1984" "I am a house."

Constants

declaration: const data-type identifier = value;
 example: const int MAX_LEN = 100;

Intrinsic and String Variable Identifier Declaration

declaration: data-type identifier[, identifier, ...];
 examples: int temp;
 int year = 1984;
 double f, c;
 string name, color="white";;

Arithmetic Operators and Expressions

- basic operators
 - addition: + example: 3 + 4 result: 7
 - subtraction: - example: 7 - 9 result: -2
 - multiplication: * example: 2.4 * 1.5 result: 3.6
 - division: / example: 7.1 / 2 result: 3.55
 - remainder: % example: 7% 3 result: 1
- order of operations:
 - highest: * / %
 - lowest: + -
- parenthesis: () changes the order of operations
 example: 3 * 2 - 4 * 5 result: -14
 3 * (2 - 4) * 5 result: -30

Assignment

operator: =
 syntax: variable = expression
 examples: (assume int i; double j; string s;)
 i = 35;
 j = 3.4 * b / c;
 s =
 int f = 72;
 string name = "Bob Cratchett";

Type Conversion

- (result-type) value example: (int) 3.1 result: 3
- string ToString() example: int i = 3;
 string s = i.ToString();
- automatic ToString string s = "" + 1984 + " was a year";
- type parse(string) example: string s = "123";
 int i = int.Parse(s);
 long l = long.Parse(s);

Block of Code (or Compound Statement)

purpose: combines multiple statements into one statement; defines scope of identifiers

syntax: {
 other statements
 }

example: int i, j;
 {
 i = 33;
 j = 33 * 4;
 }

String Manipulation

concatenation: string s = "Bob", t = "Cratchett";
 string name = s + " " + t;

Unary Increment Operator

++
 example: int i = 33;
 i++; result: i is 34
 ++; result: i is 35

Unary Decrement Operator

--
 example: int i = 33;
 i--; result: i is 32;
 --i; result: i is 31;

Common Console Output Escape characters:

\n newline
 \t tab
 \\ slash output
 \' single quote output
 \" double quote output

Standard Device Output

class access: using System;
 standard device methods: void Console.Write(string);
 void Console.Write(string,param-list);
 void Console.WriteLine(string);
 void Console.WriteLine(string,param-list);

some examples of use:

```
Console.WriteLine("Year: " + y + ", Month: " + m);
Console.WriteLine("Year: {0}, Month: {1}", y, m);
```

Standard Device Input

class access: using System;
 standard device method: string Console.ReadLine();
 example of use: string s = Console.ReadLine();

File Output

class access: System.IO;
 class: StreamWriter
 variable declaration example:

```
StreamWriter writer = new StreamWriter("output.txt", false);
```

output to file example:

```
writer.WriteLine(...);
```

closing output stream example:

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

example of output to a file:

```
StreamWriter writer = new StreamWriter("output.txt", false);
string line = "Four score and 7 years ago, ";
writer.WriteLine(line);
writer.Close();
```

File Input

class access: System.IO;
 classes: FileInfo, StreamReader
 variable declaration example:

```
FileInfo inputFile = new FileInfo("input.txt");
StreamReader reader = inputFile.OpenText();
```

input from file example:

```
string line = reader.ReadLine();
```

closing input stream example:

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

example of till-EOF:

```
FileInfo inputFile = new FileInfo("input.txt");
StreamReader reader = inputFile.OpenText();
string line;
while ((line = reader.ReadLine()) != null)
{
    ... do something with line value ...
}
reader.Close();
```

Relational Operators

== true if the left expression equals the right
 != true if the left expression does not equal the right
 < true if the left expression is less than the right
 > true if the left expression is greater than the right
 <= true if the left expression is less than or equal to the right
 >= true if the left exp. is greater than or equal to the right

Logical Operator

& true if the left and right expression are true, both expressions are evaluated
 && true if the left and right expression are true, both expressions are evaluated only if the left expression is true
 | true if the left or right expression is true, both expressions are evaluated
 || true if the left or right expression is true, both expressions are evaluated only if the left expression is false
 ! not, reverses the value of a boolean expression

If Selection

syntax: if (boolean-expression)
 statement
 if (boolean-expression)
 statement
 else
 statement
 if (boolean-expression)
 statement
 [else if (boolean-expression)
 statement]

examples: if (i > 3)
 j = 33.5;

```
if (i>3)
{
    j = 33.5;
    k = 52.7;
}
```

If Selection (continued)

```

if (i > 3)
{
    j = 33.5;
    k = 52.7;
}
else
{
    j = 0;
    k = 0;
}

if (i > 3)
{
    j = 33.5;
    k = 52.7;
}
else if (i > 6)
{
    j = -45.0;
    k = -23.9;
}
else
{
    j = 0;
    k = 0;
}

```

Switch Selection

note: switch on strings supported

typical syntax:

```

switch(expression)
{
    case value1:
        statement1
        break;
    case value2:
        statement2
        break;
    ...
    case valueN:
        statementn
        break;
    default: statement
        break;
}

```

Counters

purpose: to allow a variable to have its value changed based on its original value

syntax: variable = variable operator expression;

example: `int a = 10;`
`a = a - 3;`

Counter Shortcut Notation

operators:	<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>&=</code>
examples:	<code>a += 2;</code>		same as:	<code>a = a + 2;</code>	
	<code>a -= 2;</code>		same as:	<code>a = a - 2;</code>	
	<code>a *= 2;</code>		same as:	<code>a = a * 2;</code>	
	<code>a /= 2;</code>		same as:	<code>a = a / 2;</code>	
	<code>a %= 2;</code>		same as:	<code>a = a % 2;</code>	

While Loop

syntax: `while (boolean-expression)`
statement

examples: `int i=10;`
`while (i > 0)`
`{`
 `Console.WriteLine(i);`
 `i--;`
`}`

Do-While Loop

syntax: `do`
statement
`while(boolean-expression);`

example: `int choice;`
`do`
`{`
 `Console.WriteLine("Enter choice: ");`
 `choice = Int32.Parse(Console.ReadLine());`
`}`
`while (choice < 1 && choice > 5);`

For-Loop

syntax: `for(initial-statement; loop-condition; update-statement)`
statement

order of execution: initial-statement
loop-condition
statement
update-statement
return to loop condition

example: `for (int x=0; x<5; x++)`
`{`
 `Console.WriteLine(x * x);`
`}`

foreach-Loop

purpose: traverse a collection
 syntax: `foreach(element-type var in collection-var)`
 example: `int[] a = { 1, 2, 3, 4, 7, 8 };
 foreach (int e in a)
 Console.WriteLine(e);`

Break Statement

purpose: exits the surrounding control structure
 syntax: `break;`

Continue Statement

purpose: short circuits the execution of the statement of a loop; the loop continues
 syntax: `continue;`

Single Dimension Arrays

declaration syntax: `type[] identifier = new type[size];
 type[] identifier = { value-list};`
 element access syntax: `identifier[index-expression]`
 examples of array of intrinsic type:
`int[] a = new int[100];
 a[0] = 33;
 double[] d = { 1.1, 2.2, 3.3};
 Console.WriteLine("{0}", d[2]);`
 example of array of class type:
`Ball[] b = new Ball[10];
 for (int i=0; i<10; i++)
 b[i] = new Ball();
 Console.WriteLine(b[3].ToString());`

Two Dimension Arrays as Tables

declaration syntax: `type[,] identifier;
 type [,] identifier = { value-lists};`
 element access syntax: `identifier[index-exp][index-exp]`
 examples:
`int[,] a = new int[100, 200];
 a[0,15] = 33;
 double[,] d = { { 1.1, 2.2, 3.3},
 {4.4, 5.5, 6.6}
 };
 Console.WriteLine(d[1,2].ToString());`
 example of array of class type:
`Ball[,] b = new Ball[10,20];
 for (int i=0; i<10; i++)
 for (int j=0; j<20; j++)
 b[i,j] = new Ball();
 Console.WriteLine(b[3,4].ToString());`

Void Methods (Non-Void Class Method Functions)

prototype syntax:
`access-modifier void function-identifier(parameter-list);`
 function syntax:
`access-modifier void function-identifier(parameter-list)
 {
 statements
 }`
 function call syntax:
`function-identifier(actual-parameters);`

Non-Void Methods (Non-Void Class Method Functions)

prototype syntax:
`access-modifier type function-identifier(parameter-list);`
 function syntax:
`access-modifier type function-identifier(parameter-list)
 {
 statements
 at-least-one-return-statement
 }`
 function call syntax (use in expression):
`function-identifier(actual-parameters)`

Return Statement

purpose: exit a function; may set a value to return from the function
 syntax: `return;
 return expression;`

Formal Parameter

parameter listed in a function header

Actual parameter

parameter listed in a function call

Pass-by-Value Parameter

a copy of the actual parameter is created under the name of the formal parameter; the formal parameter becomes a separate variable and changes to it cannot change the actual parameter; by default, intrinsic (simple) types are passed-by-value

Pass-by-Reference Parameter

the address of the actual parameter is passed to the formal parameter; the formal parameter becomes a local name for the memory address of the actual parameter; changes to the formal parameter are changes to the actual parameter; by default, objects are passed-by-reference; note: even arrays of simple type are objects

Pass-By-Reference with Intrinsic (Simple) Types

placing “ref” or “out” in front of intrinsic parameters changes the default setting to pass-by-reference; “ref” dictates that the actual parameter must be initialized before call, “out” allows the actual parameter to go uninitialized at call

examples:

```
private int a = 23;
public void GetCopyA(ref int copyA)
{
    copyA = a;
}
public Program()
{
    int b = 0; // must be initialized before call
    GetCopyA(ref b);
    ...
}
```

```
private int a = 23;
public void GetCopyA(out int copyA)
{
    copyA = a;
}
public Program()
{
    int b; // does not need initialized before call
    GetCopyA(out b);
    ...
}
```

Parameter List

purpose: list of variables to be sent to a function in a call or to receive sent values or addresses in a function header

actual parameter syntax:
expression[, expression]

formal parameter syntax:
type identifier, ...
type[] identifier, ...

Array as actual Parameter

actual parameter: array-identifier

Void Method Function Example

function:

```
public void ExampleFn(int size, ref double avg, double[] c)
{
    double sum = 0;
    for (int x=0; x<size; x++)
        sum = sum + c[x];
    avg = sum / size;
}
```

call:

```
float[] f = new float[100], average;
int size = 10;
...
ExampleFn(size, ref average, f);
...
```

Non-Void Method Function Example

function:

```
public double ExampleFn(int size, double[] c)
{
    double sum = 0;
    for (int x=0; x<size; x++)
        sum = sum + c[x];
    return sum / size;
}
```

call:

```
double[] f = new double[MAX], average;
int size = 10;
...
average = ExampleFn(size, f);
...
```

Common Access Modifiers

public	members of a collection that are public may be accessed from outside of the collection object; public members may be inherited
private	members of a collection that are private may not be accessed from outside of the collection object; private members may be accessed only from inside of the collection; private members may not be inherited
protected	members of a collection that are protected may not be accessed from outside of the collection object; private members may be accessed only from inside of the collection; protected members may be inherited

Structure

a collection of members

```
syntax: access-modifier struct struct-name
{
    access-modifier type identifier;
    access-modifier type identifier;
    ....
    access-modifier type identifier;
}
```

declaration syntax:

```
struct-name identifier = new struct-name();
```

member access:

```
identifier.member
```

example:

```
public struct ExampleStruct
{
    public int i;
    public double d;
}

class Program
{
    static void Main(string[] args)
    {
        ExampleStruct b = new ExampleStruct();
        b.i = 3;
        b.d = 1.2;
    }
}
```

Class

a collection of members; members consist of member function methods and data member; member functions are implemented separately from the definition to facilitate information hiding;

definition syntax:

```
class class-name
{
    access-modifier type field-member-name;
    access-modifier type field-member-name = ...;
    ...
    access-modifier type field-member-name;

    access-modifier type method-member-name(parameter-list)
    { ... }
    access-modifier type method-member-name(parameter-list)
    { ... }
    ...
}
```

Class Constructors

members with the same name as the class; called when an object of the class type is created; constructors are always public

implementation syntax:

```
access-modifier class-name(parameter-list)
{
    statements
}
```

call syntax:

```
class-name variable-name = new class-name(param-list);
```

```
class-name variable-name;
```

```
variable-name = new class-name(parameter-list);
```

Destructors

members with the same name as the class preceded by a tilde (~); called when an object of the class type is deleted or passes out of scope; often unnecessary in C# because of automatic garbage collection

implementation syntax:

```
access-modifier ~class-name()
{
    statements
}
```

Inheritance

a collection may inherit the public and protected members of another class; the inherited members become members of the inheriting class; members inherited from the parent class may be redefined (overridden) in the child class if the parent class method has the access-modifier “virtual” and the child class method has the access-modifier “override”; a parent class with the access-modifier “abstract” has at least one method-header marked “abastract” (the method has no code) and MUST be overridden in the child class; a class with the access-modifier “sealed” cannot be inherited.

class inheritance syntax:

```
class child-class-name : parent-class-name
{
    ...
};
```

child class constructor header syntax:

```
child-class-name(p-list) : parent-class-name(p-list)
```

Note: the members of the p-list of the parent class constructor must be members of the p-list of the child class constructor

constructor usage syntax:

```
child-class-name identifier = new child-class-name(p-list);
```

Name Spaces

A name space is a named place where identifiers are created and exist. The name space is the first part of a name for an identifier.

Code is created inside of a name space. This allows the same identifier to be reused in a name space, even though it may be in use in another namespace.

```
Syntax:      namespace namespace-identifier { ... }
Example:     namespace ConsoleApplication8
              {
                .... code here ....
              }
```

If an identifier exists in a name space, it may be accessed via its full name, i.e. name-space.identifier

```
Example:     System.Console.WriteLine("Hello, World!");
```

If the using statement is placed prior to a name space, the identifiers of the name space named in the using statement become local to the new name space.

```
Example:
            using System;
            namespace ConsoleApplication8
            {
                class Program
                {
                    static void Main(string[] args)
                    {
                        Console.WriteLine("Hello, World!");
                        ...
                    }
                }
            }
```

Using some Common APIs and Classes

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using System.Text;
using System.Xml;
using System.IO;
using System.Security.Cryptography;
```

Typical Creation of Random Numbers

The value of "i" in the following examples will be a random whole number between 1 and 2,147,483,647, j will contain a random number between 0 and 99.

```
int i = RandomClass.next();
int j = i % 100;
```

Pointers

There are no pointers in C#. :>

Reference Variables

All object variables with identifiers are reference variables, i. e. can contain the address of an actual, but unnamed variable of the same type or the value null.

null

value that does not point to any memory address; any reference variable can be assigned the constant null; any reference variable can be tested against the constant null

Relational Operator Overloading for Classes

declare public; return boolean value

header syntax:

```
type operator operator (type identifier, type identifier);
```

example overloaded method implementation:

```
public class SimpleBook
{
    private string title;
    public static bool operator ==(SimpleBook sb1,
                                   SimpleBook sb2)
    {
        return (sb1.title == sb2.title);
    }
    public static bool operator !=(SimpleBook sb1,
                                   SimpleBook sb2)
    {
        return (sb1.title != sb2.title);
    }
    ...
}
```

Note: Overloading the == operator requires overloading the != operator. The raison d'être of the operator cannot be changed.

Generics

Allows a type to be passed to a class when an object of that class is created. The type value is used where ever the type variable appears in the class.

Class Header Syntax:

```
access-modifier class class-identifier<type-variable>
```

Example of One Inside the Class Syntax:

```
type-variable identifier = new type-variable();
```

Example of class and class objects:

```
public class List<Type> ...
    // "Type" is used wherever the type for the
    // data element is used
...
List<Item> inventory = new Stack<Item>;
// Item is the value of Type in class Stack

List<Bird> sitings = new List<Bird>;
// Bird is the value of Type in class Stack
```

Scope

Scope is the region where an identifier is said to be visible. Another way to think of this is that scope is the property that determines if a variable is usable or accessible in a specific context. All identifiers are said to have scope and only identifiers have scope. The scope of an identifier is said to be local, global or out of scope.

Identifiers that are local or global may be accessed. Identifiers that are out of scope can not be accessed.

An identifier declared within a compound statement is local to that compound statement. This means that it cannot be accessed outside of that compound statement, as it would be out of scope. However, the same variable may be global to another compound statement if that second compound statement is within the compound statement in which the identifier was declared. An identifier that is global can be accessed.

Identifiers declared in the header of a method function are considered local to that method function. Identifiers declared in a For statement are considered local to the body of the For Loop.

Identifiers that are declared outside of a method function are considered global within the block of code that they are declared in.

Access modifiers (public, protected and private) can be used to modify the access rights of identifiers declared at class level.

Private and protected identifiers cannot be accessed from outside of a class. Since class and struct are types, an object must be created in order for any of their members to exist and the member access notation [dot (.) operator] must be used to access public members of an object of a struct or class type.

Exceptions: Throw and Catch within a Method

allows program execution to continue

- try – block of code to attempt
- catch – block of code to handle exceptions
- finally – block code that must be done when exception occurs

syntax:

```
try block
[catch (parameter) block
[catch (parameter) block
[...]]]
[catch block]
[finally block]
```

example:

```
int f = 0, g, h;
for (g = -1; g < 2; g++)
    for (h = -1; h < 2; h++)
    {
        try
        {
            Console.WriteLine("{0} / {1}", f, g);
            f = g / h;
        }
        catch (DivideByZeroException e)
        {
            Console.WriteLine("Exception:\n" + e);
            f = 0;
        }
        finally
        {
            Console.WriteLine("result is {0}\n",f);
        }
    }
```