class Program

{

static void Main(string[] args)

{

}

}

Variable types

­­

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type Name | Description | Bytes | Data Range | Example |
| short | stores smaller integers | 2 | -32,768 to 32,767 | short score = 495; |
| int | stores medium sized integers | 4 | -2,147,483,648 to 2,147,483,647 | int score = 450000; |
| long | stores very large integers | 8 | –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | long highScore = 4043333; |
| byte | stores small unsigned (no + or -) integers | 1 | 0 to 255 | byte color = 55; |
| ushort | stores small unsigned integers | 2 | 0 to 65,535 | ushort score = 495; |
| uint | stores medium unsigned integers | 4 | 0 to 4,294,967,295 | uint score = 4500000; |
| ulong | stores large unsigned integers | 8 | 0 to 18,446,744,073,709,551,615 | ulong highScore = 4043333; |
| float | stores smaller real (floating point) numbers | 4 | ±1.5 × 10^−45 to ±3.4 × 10^38, 7 digits of accuracy | float xPosition = 3.2f; |
| double | stores larger real numbers | 8 | ±5.0 × 10^−324 to ±1.7 × 10^308, 15 to 16 digits of accuracy | double yPosition = 3.3; |
| decimal | real numbers, smaller range, higher accuracy | 16 | ±1.0 × 10^−28 to ±7.9 × 10^28, 28 to 29 digits of accuracy | decimal zPosition = 3.3; |
| char | stores a single character or letter | 2 | any single character | char myFavoriteLetter = 'c'; |
| string | stores a sequence of numbers, letters, or symbols | any | a sequence of any character of any length | string message = "Hello World!" |
| bool | stores a *true* or *false* value | 1 | *true* or *false* | bool levelComplete = true; |

Console.WriteLine("You can add strings and number like this. " + 12345);

string test1 = "";

test1 = test1 + 1 + 1; //test1 = "11"

Modulus (mod) operator. Returns the operator  
int test1 = 10 % 3; //returns the remainder, 1

int b = 7;

b -= 3; // This is the same as b = b - 3; At this point b would be 4.

b \*= 5; // This is the same as b = b \* 5; At this point b would be 20.

b /= 4; // This is the same as b = b / 4; At this point b would be 5.

b %= 2; // This is the same as b = b % 2; At this point b would be 1.

string whatTheUserTyped = Console.ReadLine();

user types, then hits enter. This is used as the string and the code continues

int aNumber = Convert.ToInt32(whatTheUserTyped);

convert the variable  
ToInt16 - short  
ToInt32 - int  
ToSingle - float  
ToDouble - double  
ToBool - bool

int a = 7;

int b = 2;

int result = a / b;

result is 3. Because it is int, any decimal gets chopped off

int a = 7;

int b = 2;

float c = 4;

float d = 3;

float result = a / b + c / d;

result = (3) + (1.333) = 4.3333

casting (typecasting)  
int a = 4049;

long b = 284404039;

long sum = a + b;

when different types, a narrower type can easily and automatically go into a wider type (int to long)

when going wide to narrow, this must be stated explicitly

long a = 3;

int b = (int)a;

long a = 3;

int b = a;

this will return an error

int a = 7;

int b = 2;

int c = 4;

// This gives you 7.5, because 'b' gets turned in to a float right away,

// and then 'a' gets implicitly converted to a float to do the division,

// then 'c' gets converted to a float to do the addition.

float result1 = a / (float)b + c;

// This gives you 7, because the a / b happens with integer division,

// giving you 3, then the addition is done with 'c', giving you 7, then

// finally, the result is turned into a floating point value of 7.

float result2 = (float)(a / b + c);

double a = double.PositiveInfinity;

float b = float.PositiveInfinity;

infinity

double a = double.NaN;

float b = float.NaN;

Not a number

Math.PI is Pi

Math.E is e

int maximum1 = Int32.MaxValue;

int maximum2 = int.MaxValue;

int minimum1 = Int32.MinValue;

int minimum2 = int.MinValue;

this will return min/max possible number in the inputted data type

int a = 3;

a++;

this will add 1 to a

int a = 3;

int b = ++a; // Both 'a' and 'b' will now be 4.

int c = 5;

int d = c++; // The original value of 5 is assigned to 'd', while 'c' is now 6.

if(condition)

{

statement;

}

else if(another condition)

{

statement;

}

else

{

statement;

}

If(a==b), if a equals b   
If(a!=b), if a does not equal b   
If(a>=b,   
If(a<=b)   
If(a>b)   
If(a<b)

If(!(a==b)), if the statement is false, not a equals b

if(a <= 0 && b <= 0)

&& and

if(a <= 0 || b <= 0)

|| or

Switch is useful when there are many else if statements

int menuChoice = 2;

switch (menuChoice)

{

case 1:

Console.WriteLine("You chose option #1");

break;

case 2:

Console.WriteLine("You chose option #2. I like that one too!");

break;

default:

Console.WriteLine("I don't know what to do with that number.");

break;

}

Switch statements cannot use general logic (<, >, so on)

Every case needs a break, unless the case is empty

switch (menuChoice)

{

case 1:

case 2:

Console.WriteLine("You chose option 1 or 2.");

break;

looping

while(condition) { /\* ... \*/ }

do { /\*... \*/ } while(condition);

for(initialization; condition; update) { /\* ... \*/ }

int x = 1;

while(x <= 10)

{

Console.WriteLine(x);

x++;

}

Break will exit the loop

Continue can be used to advance to the next iteration of the loop

while( condition )

{

// This code is repeated until the condition is false.

}

while(true)

{

// Depending on what goes in here, you'll never end...

}

Forever loop. Keeping going until a break

do

{

// The condition is checked at the end of the do code

}

while (/\*... \*/);

a variable used in the do while statement does not need to be initialized, unlike in a while statement

for(initial condition; condition to check; action at end of loop)

{

//...

}

Loop until to condition is false

for(int x = 1; x <= 10; x++)

{

Console.WriteLine(x);

}

The variable can be declared and initialized inside the loop

int[] array1 = new int[5] this array can hold 5 integers.  
The first integer is array1[0]

int[] array1 = new int[] { 100, 90, 80, 70, 60, 50, 40, 30, 20, 10 };

foreach(int x in array1)

{

Console.WriteLine(x);

}

Multidimensional arrays

Jagged array

int[][] matrix = new int[4][];

matrix[0] = new int[4];

matrix[1] = new int[5];

matrix[2] = new int[2];

matrix[3] = new int[6];

matrix[2][1] = 7;

square array

int[,] matrix = new int[4, 4];

matrix[0, 0] = 1;

matrix[0, 1] = 0;

matrix[3, 3] = 1;

the for each loop will go through each item in an array. The downside is that it does not keep track of the index number of the array item

int[] scores = new int[10];

// Populate data and maintain it as your program runs. The below is just an example.

scores[0] = 42;

scores[5] = -1;

foreach (int score in scores)

{

Console.WriteLine("Someone had this score: " + score);

}

For this, use a regular for loop

int[] scores = new int[10];

// Populate data and maintain it as your program runs. The below is just an example.

scores[0] = 42;

scores[5] = -1;

for(int index = 0; index < scores.Length; index++)

{

int score = scores[index];

Console.WriteLine("Score #" + index + ": " + score);

}

Enumerations are done outside of any classes

public enum DaysOfWeek { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

int dayAsInt = (int)DaysOfWeek.Sunday;

DaysOfWeek today = (DaysOfWeek)dayAsInt;

DaysOfWeek tomorrow = (DaysOfWeek)(dayAsInt + 1);

Values can be assigned to a value of an enumeration

public enum DaysOfWeek { Sunday = 5, Monday = 6, Tuesday = 7, Wednesday = 8, Thursday = 9, Friday = 10, Saturday = 11 };

methods, also called functions and procedures

an added method might look like this

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.IO;

using System.Net;

namespace MethodsTutorial

{

public class Program

{

static void Main(string[] args)

{

}

static void CountToTen()

{

for (int index = 1; index <= 10; index++)

{

Console.WriteLine(index);

}

}

}

}

The static keyword

Void, this means that the method wont give us anything back, it will do what it needs to do and be done

If the method is going to return something, change void to the variable type that is going to be returned

static int GetNumberFromUser()

{

int usersNumber = 0;

while(usersNumber < 1 || usersNumber > 10)

{

Console.Write("Enter a number between 1 and 10: ");

string usersResponse = Console.ReadLine();

usersNumber = Convert.ToInt32(usersResponse);

}

return usersNumber;

}

int usersNumber = GetNumberFromUser();

this variable will equal the number returned from the method

return can be used in void methods, it just wont return anything

return exits the method

handing data to methods

static void Main(string[] args)

{

Count(Convert.ToInt32(Console.ReadLine()));

Console.ReadKey();

}

static void Count(int countnum)

{

for (int current = 1; current <= countnum; current++)

{

Console.WriteLine(current);

}

}

static int Multiply(int a, int b)

{

return a \* b;

}

Multiple inputs can be given

static int Multiply(int a, int b)

{

return a \* b;

}

static int Multiply(int a, int b, int c)

{

return a \* b \* c;

}

Methods with the same name can exist, but they need to have different signatures

static int Multiply(int a, int b)

{

return a \* b;

}

static double Multiply(double a, double b)

{

return a \* b;

}

static int Multiply(int a, int b)

{

return a \* b;

}

static int Multiply(int c, int d) // This won't work. It has the same signature.

{

return c \* d;

}

Recursion, calling a method inside itself

static int Factorial(int number)

{

// We establish our "base case" here. When we get to this point, we're done. Both 0

// and 1 are defined to have a factorial of 1. Mathematics doesn't define a factorial

// for negative numbers, and we shouldn't allow it either, but we don't yet have the

// tools to appropriately account for that. (See the Exceptions tutorial.)

if(number == 0 || number == 1)

{

return 1;

}

return number \* Factorial(number - 1);

}

Overflow error will occur if nothing stops the method

Random number

Random random = new Random();

int nextRandomNumber = random.Next();

Random random = new Random();

int dieRoll = random.Next(6) + 1; // the Next method returns numbers between 0 and 5, inclusive.

The Random class also has a NextDouble method, which returns numbers between 0 and 1. This allows you to do things like random probabilities and so on.

Random random = new Random(seed);

An object is an instance of a class

"value" types, like your normal variables (int, bool, double, etc.) live on the stack, while "reference" types (objects) live in the heap, with a reference (or many references) to the object staying on the stack.

Changes to a variable in a method call do not affect the original variable. Changes to an object referenced in a method call do affect the object

Create a new class for separate object types. This helps keep organized

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace CreatingClasses

{

class Player

{

private string name;

private int score;

private int livesLeft;

}

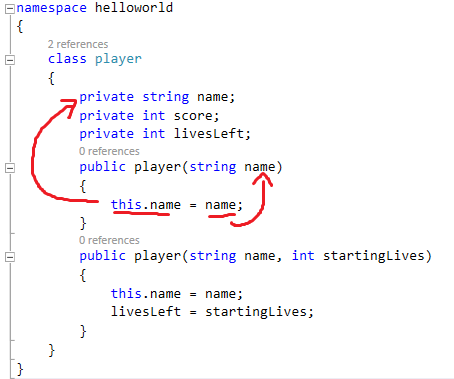
}

Private keyword indicated that the variable can only be accessed from within the class  
public keyword indicates that the variable can be seen or used from outside of the class  
private is default

Variable scope

Method scope - When a variable is used in a method, it cannot be used outside of that method, it is out of scope  
block scope - when a variable is used in a loop, it can only be used in that scope

Class scope –



If there is a variable in with method/block scope that shares a name with a variable with class scope, using this.variable will reference the class variable

Making a method public means that it can be access from outside of the class. Public methods can only be access from within the class

Static variables belong to the class as a whole, not an individual instance

A constructor is a method inside of a class that creates an instance of a class. A class must have a constructor in order to be created.

Player player = new Player();

//Class variable = new constructor(inputs);

Player player = new Player("Frankenstein",14);

List<Player> allplayers = new List<Player>();

allplayers.Add(player);

while (player.GetLivesLeft() > 0)

{

player.AddPoints(100);

player.Kill();

}

foreach (Player i in allplayers)

{

Console.WriteLine("Player name is " + i.GetName());

Console.WriteLine("Player score is " + i.GetScore());

Console.WriteLine("Player lives is " + i.GetLivesLeft());

}

Console.ReadKey();

private int score;

public int Score

{

get

{

return score;

}

set

{

score = value;

if (score < 0)

{

score = 0;

}

}

}

The score variable is private and cannot be accessed outside of the instance. However, the Score property can be accessed. Get determines what happens when the property is accessed. Set determines what happens when the property is changed. This allows for handling the input before it is applied. In this case it prevents a score from being negative.  
Both get and set do not need to be present. This would make the property read- or write-only

Player player = new Player("Frankenstein") { Score = 100, LivesLeft = 7 };

A property can be set upon initialization as well, like this

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Inheritance

{

class Square : Polygon

{

public float Size { get; set; }

public Square(float size)

{

Size = size;

NumberOfSides = 4;

}

}

}

A class can inherit properties from another class. This is a parent-child relationship

Polygon polygon = new Square(4.5f);

If a variable is declared as a parent but created as a child, you cannot see any reference to it as a child. You can check this way

Polygon polygon = new Square(4.5f);

if (polygon is Square)

{

Square square = (Square)polygon;

// We can now do what we want with the square.

}

Base Class – the parent

Derived Class – the child

Polymorphism – a base class with multiple different derived classes. Ex: a player base class that has HumanPlayer and AIPlayer derived classes

In a base class, a method prefixed by **virtual** means that it can be overridden by a derived class

public virtual MoveDirection MakeMove()

In a derived class, a method prefixed by **override** can override a method in the base class. This happens by default

public override MoveDirection MakeMove()

public override MoveDirection MakeMove()

{

MoveDirection moveDirection = base.MakeMove();

if(moveDirection == MoveDirection.None)

{

return MoveDirection.Up;

}

}

Using the base keyword, access to a method in the base class can be accessed

abstract class Player

{

public virtual MoveDirection MakeMove();

}

The abstract keyword prevents a class from being able to be created

public abstract MoveDirection MakeMove();

abstract methods can be created as well. These do not have any content and serve only to by overridden

public new MoveDirection MakeMove()

{

//...

}

You can create a new method in a derived class. This means that this method has nothing to do with the method of the same name in the base class. Why?

<http://rbwhitaker.wikidot.com/c-sharp-interfaces> - fairly confusing, revisit this later

using System.IO;

to write information to a file and read it back in, add the using System.IO statement to the top alongside the other using statements

string informationToWrite = "Hello persistent file storage world!";

File.WriteAllText("C:/Users/RB/Desktop/test1.txt", informationToWrite); // Change the file path here to where you want it.

string[] arrayOfInformation = new string[2];

arrayOfInformation[0] = "This is line 1";

arrayOfInformation[1] = "This is line 2";

File.WriteAllLines("C:/Users/RB/Desktop/test2.txt", arrayOfInformation);

The first method will write the string to the file, the second will write multiple lines to the file

string fileContents = File.ReadAllText("C:/Users/RB/Desktop/test1.txt");

string[] fileContentsByLine = File.ReadAllLines("C:/Users/RB/Desktop/test2.txt");

this will read the data, populating variables

HighScore[] highScores = new HighScore[10];

// populate and maintain the list of high scores in here as needed...

string allHighScoresText = "";

foreach(HighScore score in highScores)

{

allHighScoresText += score.Name + "," + score.Score + "**\n**";

}

File.WriteAllText("highscores.csv", allHighScoresText);

A method to translate variables into a string to be written into a file

\ is used to indicate special characters

\n indicated a new line

To read, parse the data by reading it, then splitting it

string[] highScoresText = File.ReadAllLines("highscores.csv");

HighScore[] highScores = new HighScore[highScoresText.Length];

for(int index = 0; index < highScoresText.Length; index++)

{

string[] tokens = highScoresText[index].Split(',');

string name = tokens[0];

int score = Convert.ToInt32(tokens[1]);

highScores[index] = new HighScore(name, score);

}

Split is used to break strings into multiple parts. This can be used to populate an array and fill variables

FileStream fileStream = File.OpenWrite("C:/Users/RB/Desktop/test3.txt");

TextWriter textWriter = new StreamWriter(fileStream);

textWriter.Write(3);

textWriter.Write("Hello");

textWriter.Flush();

textWriter.Close();

This allows text to be written a bit at a time, rather than all at once. Flushing indicated the code to be writted (nothing is written until flushed). Closing the textwriter disconnects the connection to the file.

<http://rbwhitaker.wikidot.com/c-sharp-file-io> - some info was confusing, revisit later

List<string> strings = new List<string>();

strings.Add("text1");

strings.Add("text2");

lists are good for unknown size arrays

strings.Insert(0, "text3");

listOfStrings.ElementAt(0);

returns the item at the given index number

string secondItem = listOfStrings[1];

this returns the item at index 1

listOfStrings[0] = "This message replaces whatever was already at index 0";

listOfStrings.RemoveAt(2);

listOfStrings.Clear();

int itemsInList = listOfStrings.Count;

List<int> someNumbersInAList = new List<int>();

someNumbersInAList.Add(14);

someNumbersInAList.Add(24);

someNumbersInAList.Add(37);

int[] numbersInArray = someNumbersInAList.ToArray();

foreach(int number in someNumbersInAList)

{

// ...

}

Dictionaries associate a value to a key

Dictionary<string, int> phoneBook = new Dictionary<string, int>();

phoneBook["Gates, Bill"] = 5550100;

phoneBook["Zuckerberg, Mark"] = 5551438;

int billsNumber = phoneBook["Gates, Bill"];

<http://rbwhitaker.wikidot.com/c-sharp-generics> - generics and making your own

Console.Write("Enter a number: ");

string userInput = Console.ReadLine();

try

{

// The code that may have problems goes in here...

int number = Convert.ToInt32(userInput);

// It is important to point out that if there was an error in

// the line above, the program would jump down to the

// catch block below, and any code right here would not

// be executed.

}

catch(Exception e)

{

// The code to handle the problem goes here.

// Notice that if nothing goes wrong, this code

// never gets executed.

Console.WriteLine("You must enter a number.");

}

If the variable isn’t going to be used, you don’t need to name it

try

{

//...

}

catch(Exception) // this Exception variable has no name

{

//...

}

try

{

int number = Convert.ToInt32(userInput);

}

catch (FormatException e)

{

Console.WriteLine("You must enter a number.");

}

catch (OverflowException e)

{

Console.WriteLine("Enter a smaller number.");

}

catch (Exception e)

{

Console.WriteLine("An unknown error occurred.");

}

|  |  |
| --- | --- |
| **Exception Type** | **What it Does** |
| NotImplementedException | If a method has been defined, but not implemented yet, you can use this exception. If Visual C# Express automatically generates methods for you, it will put this in the body. |
| IndexOutOfRangeException | You tried to access an array or something at an index that is beyond how big the array is. |
| InvalidCastException | You tried to cast something to another type, but the type you tried to cast to wasn't the right kind of object. |
| FormatException | The text you had is not in the right format for converting to something else (like letters in a string that is supposed to be turned into a number). |
| NotSupportedException | You tried to do an operation that wasn't supported. For instance, make a method call at a time that didn't allow it. |
| NullReferenceException | A variable for any kind of object actually contained *null* instead of an actual object, but whatever you were doing required something besides *null*. |
| StackOverflowException | You see this all the time when you run out of space on the stack from calling too many methods. This is usually a result of recursion that went bad. |
| DivideByZeroException | You tried to divide by zero and got caught. |
| ArgumentException | One of the parameters or arguments you sent to a method didn't match what was required by the method. |
| ArgumentNullException | One of the arguments or parameters that you gave to a method was null, but the method requires something besides null. |
| ArgumentOutOfRangeException | One of the arguments contained a value that the method couldn't intelligently do something with. For instance, if a method required a number between 1 and 10, but you gave it -13, you might see this kind of exception. |

public static void TestFinally(bool throwException)

{

try

{

Console.WriteLine("in try");

if (throwException)

{

throw new Exception();

}

return;

}

catch (Exception)

{

Console.WriteLine("in catch");

}

finally

{

Console.WriteLine("in finally");

}

}

Finally block code always gets executed, even if there is a return in the try block

Delegates – use a “variable” to assign a method to run data through

public delegate float MathDelegate(float a, float b);

public static float Add(float a, float b)

{

return a + b;

}

public static float Subtract(float a, float b)

{

return a - b;

}

public static float Power(float baseNumber, float exponent)

{

return (float)Math.Pow(baseNumber, exponent);

}

MathDelegate mathOperation = Add;

float a = 5;

float b = 7;

float result = mathOperation(a, b);

<http://rbwhitaker.wikidot.com/c-sharp-delegates>  
more practical application

threading – using multiple threads to process more data at once. Useful for long monotonous tasks that will not interfere with other tasks

<http://rbwhitaker.wikidot.com/c-sharp-threading>

operator overloading – you can define how things behave when using operators. For example, how should two custom classes be added to each other?

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace OperatorOverloading

{

public class Vector

{

public double X { get; set; }

public double Y { get; set; }

public Vector(double x, double y)

{

X = x;

Y = y;

}

public static Vector operator +(Vector v1, Vector v2)

{

return new Vector(v1.X + v2.X, v1.Y + v2.Y);

}

public static Vector operator +(Vector v, double scalar)

{

return new Vector(v.X + scalar, v.Y + scalar);

}

public static Vector operator -(Vector v)

{

return new Vector(-v.X, -v.Y);

}

public static bool operator ==(Vector v1, Vector v2)

{

return ((v1.X == v2.X) && (v1.Y == v2.Y));

}

public static bool operator !=(Vector v1, Vector v2)

{

return !(v1 == v2);

}

}

}

In this vector class, various operators are defined

The same can be done for indexing

public double this[int index]

{

get

{

if(index == 0) { return this.X; }

else if(index == 1) { return this.Y; }

else { throw new IndexOutOfRangeException(); }

}

set

{

if (index == 0) { this.X = value; }

else if (index == 1) { this.Y = value; }

else { throw new IndexOutOfRangeException(); }

}

}

With this done, you can use indexing however it is defined for this class

Vector v = new Vector(5, 2);

double xComponent = v[0]; // we can now use the indexing operator!

double yComponent = v[1];

you cannot create an instance of a static class