Introducción al C#.

Diferencias y Similitudes con C++ y Java

Sistemas Distribuidos
Grado en Ingeniería Informática

1. Características.

- **C#** (Inglés "C Sharp" y en español "C Almohadilla"): Lenguaje de propósito general diseñado por Microsoft para su plataforma .NET. Sus principales creadores son Scott Wiltamuth y Anders Hejlsbergc (creador del lenguaje Turbo Pascal y la herramienta RAD Delphi)
- Es el lenguaje nativo de la plataforma .NET
- Su sintaxis es muy parecida al C++ para facilitar la migración de código y/o programadores.

- Entre las características que posee podemos destacar:
  - Sencillez:
    - Es autocontenido ➔ No necesita ficheros cabecera
    - El tamaño de los tipos básicos es fijo independiente del copilador ➔ No se necesita el uso del **`sizeof`**
    - No incluye macros, ni herencia múltiple

- **Moderno**: Incorpora tipos especiales de alta precisión para aplicaciones financieras
- **Es orientado a objetos puros**. No admite ni funciones ni variables globales. Todo el código y datos han de definirse dentro de los tipos de datos
- **Soporta** características propias de la programación orientada a objetos como la encapsulación, **herencia** y **polimorfismo** con importantes variaciones con respecto al c++
- **Orientación a componentes**: Definición y uso de propiedades, eventos y atributos.
- **Gestión automática de memoria**: No es necesario eliminar objetos ya que tiene su propio recolector de memoria no utilizada.

- **Seguridad de tipos**:
  - Solo hay conversiones entre tipos compatibles (un tipo y sus ancestros) y los tipos en los que explícitamente se haya definido un operador de conversión.
  - No se puede utilizar variables locales no inicializadas.
  - El compilardor inicializa valores por defecto a los campos o atributos.

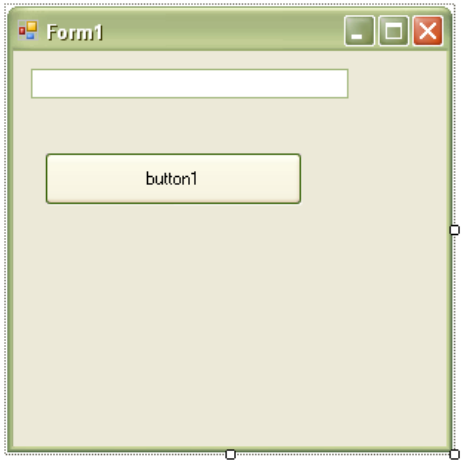
- Incluye delegados que son variables que puede contener referencias a uno o varios métodos a la vez.
- Admiten definir métodos con un número indefinido de parámetros.

- **Sistemas de tipos unificados**. Todos los tipos, incluidos los básicos, derivan de manera implícita o explicita de la clase común denominada **`System.Object`**
- **No permite el uso de punteros**. Esta restricción puede eliminarse definiendo regiones no seguras.

2. Aplicación Hola Mundo

```
using System;
class HolaMundo
{
  static void Main()
  {
     Console.WriteLine("Hola Mundo");
  }
}
```

Los métodos **`static`** (estáticos) son métodos de clase y no de objetos.




```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Pruebas
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            textBox1.Text = "Hola Mundo";
        }

        private void button1_Click(object sender, EventArgs e)
        {
            Close();
        }
    }
}
```

3. Librería de clase base (BCL)

Espacio de nombres	Utilidad de los tipos de datos que contiene
System	Tipos muy frecuentemente usados, como los tipos básicos, tablas, excepciones, fechas, números aleatorios, recolector de basura, entrada/salida en consola, etc.
System.Collections	Colecciones de datos de uso común como pilas, colas, listas, diccionarios, etc.
System.Data	Manipulación de bases de datos. Forman la denominada arquitectura **ADO.NET**.
System.IO	Manipulación de ficheros y otros flujos de datos.
System.Net	Realización de comunicaciones en red.
System.Reflection	Acceso a los metadatos que acompañan a los módulos de código.
System.Runtime.Remoting	Acceso a objetos remotos.
System.Security	Acceso a la política de seguridad en que se basa el CLR.
System.Threading	Manipulación de hilos.
System.Web.UI.WebControls	Creación de interfaces de usuario basadas en ventanas para aplicaciones Web.
System.Winforms	Creación de interfaces de usuario basadas en ventanas para aplicaciones estándar.
System.XML	Acceso a datos en formato XML.

4. Aspectos Léxicos

- Palabras reservadas:

```
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, lock, is,
long, namespace, new, null, object, operator, out, override, params,
private, protected, public, readonly, ref, return, sbyte, sealed,
short, sizeof, stackalloc, static, string, struct, switch, this,
throw, true, try, typeof, uint, ulong, unchecked, unsafe, ushort,
using, virtual, void, while
```

- **Comentarios**: Iguales al c++
- **Identificadores**: Iguales. Se permite el uso de palabras reservadas como identificadores utilizando el símbolo **@**

- **Literales**: enteros, reales, lógicos y carácter y cadena iguales a c++ salvo. Los literales cadena precedidos por el símbolo @ no interpretan los literales carácter ejemplo: @"Hola\tque\ttal\n"
- **Operadores**:
  - **Aritméticos**: Iguales al C++. Se añade el par de operadores

`checked (<expresiónAritmética>)`: Está por defecto en operaciones aritméticas con constantes. Provoca un error de compilación si **`<expresiónAritmética>`** es una expresión constante y una excepción **`System.OverflowException`** si no lo es.

`unchecked(<expresiónAritmética>):` Está por defecto en operaciones aritméticas entre datos no constantes. Devuelve el resultado de la expresión aritmética truncado para modo que quepa en el tamaño esperado.

  - **Lógicas**: Iguales al C++
  - **Relacionales**: Iguales al C++
  - **Asignación**: Iguales que en c++ incluidos las versiones compuestas como +=
  - **Con cadenas**: El símbolo + se utiliza para concatenar cadenas.

- **De acceso a tablas**: Iguales que en c++.
- **Condicionales** (**?**): Igual que en c++.
- **Delegados**: Se utiliza **+** y **+=** para añadir al delegado y **–** y **-=** para eliminarlos.
- **De acceso a miembros**: Solo se utiliza el operador punto (**.**) no se utiliza la flecha →
- **Con punteros**: Iguales que en c++, pero solo se utilizan en regiones no seguras.
- **Información de tipos**:
  i. El operador **`is (<expresión> is <nombre tipo>)`** devuelve true o false si el tipo de la expresión es el mismo que **`<nombre tipo>`**
  ii. El operador **`sizeof`** es igual que en c++ pero solo se utiliza en regiones inseguras
  iii. Se puede obtener información sobre un tipo con el comando **`typeof(<nombre tipo>)`**. No devuelve gran cantidad de información sobre sus miembros, visibilidad, naturaleza, etc.
- **Creación de objetos**: Igual que en c++. Existe otro operador equivalente al **`new`** que es el **`stackalloc`** para crear tablas de objetos

- **Conversión**:
  i. Se puede utilizar el forzado de tipos del c++. Si es una conversión inválida produce una excepción **`System.InvalidCastException`**.
  ii. El operador **`as`** (**`<expresión> as <tipo Destion>`**) hace una conversión de tratamiento del tipo origen de la expresión al nuevo tipo. El operador as solo es aplicable a los tipos referencia y siempre que tengan conversiones predefinidas. Solo es una solicitud de tratamiento de la información referenciada al nuevo tipo, por ese motivo su ejecución es más rápida. Si produce una conversión inválida devuelve **`null`**.
  iii. **Declaración de Variables**: Igual que en c++ salvo que los tipos también son objetos y heredan de **`System.Object`**. Los tipos básicos al ser muy utilizados el compilador los crea automáticamente con **`new`**.

- **Tipos de Datos:**

Tipo	Descripción	Bits	Rango de valores	Alias
SByte	*Bytes* con signo	8	-128 – 127	sbyte
Byte	*Bytes* sin signo	8	0 – 255	byte
Int16	Enteros cortos con signo	16	[-32.768, 32.767]	short
UInt16	Enteros cortos sin signo	16	[0, 65.535]	ushort
Int32	Enteros normales	32	[-2.147.483.648, 2.147.483.647]	int
UInt32	Enteros normales sin signo	32	[0, 4.294.967.295]	uint
Int64	Enteros largos	64	[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]	long

Tipo	Descripción	Bits	Rango de valores	Alias
Single	Reales con 7 dígitos de precisión	32	$[1{,}5\times10^{-45} - 3{,}4\times10^{38}]$	float
Double	Reales de 15-16 dígitos de precisión	64	$[5{,}0\times10^{-324} - 1{,}7\times10^{308}]$	double
Decimal	Reales de 28-29 dígitos de precisión	128	$[1{,}0\times10^{-28} - 7{,}9\times10^{28}]$	decimal
Boolean	Valores lógicos	32	**true**, **false**	bool
Char	Caracteres Unicode	16	[‘\u0000’, ‘\uFFFF’]	char
String	Cadenas de caracteres	Variable	El permitido por la memoria	string
Object	Cualquier objeto	Variable	Cualquier objeto	object

- **Declaración de objetos**. Todo en C# son referencias a objetos creados en tiempo de ejecución. Cuando se comparan dos objetos el c# compara la direcciones de memoria y no los contenidos de ambos objetos salvo que los objetos tengan sobrecargados los operadores de relacionales. Si se quiere siempre verificar si dos objetos apunta a la misma memoria hay que utilizar **`Object.ReferenceEquals(obj1,obj2)`**

- **Cadenas de texto constantes**: tipo **`string`** (alias del tipo **`System.String`** de la BCL)

  `string Cadena1="Hola que tal";` //Son cadenas constantes

  `string Cadena2=Cadena1;` //Apuntan a la misma constante cadena.

  `string Cadena3=System.String.Copy(Cadena1);` //Son dos cadenas distintas con el mismo contenido

  `string Caden4=Cadena2+Cadena3;` //Contiene una cadena constante resultante de concatenar ambas cadenas

  Cadena de Texto: El tipo es el **`StringBuilder`**; Ejemplo

  `StringBuilder Cadena=new StringBuilder("Hola que tal");`

- **Constantes**: igual que en C++. Se añade constantes de clase utilizando el operador **`static`** (**`<nombreClase>.<nombreConstante>`**)
- **Variables de solo lectura**: Solo pueden ser utilizadas para ser leidas sin poder cambiar su contenido más de una vez. Hay que utilizar la palabra reservada readonly antes de su declaración, ejemplo:

  `readonly int valor`;

- **Tablas**:
  - **Unidimensionales**: Igual que en java;

    ejemplo

    ```
    int[] vector; //Solo declaración del objeto tabla
    vector=new int[100] //Construcción de contenido de la tabla.
    ```

    También podría ser: `vector=new int[] {1,2,3,4};` // indicndo su valores.

    `vector.Length` nos devuelve el nº de elementos de la tabla.

  - **Dentadas o multitablas**: Son tablas que contiene tablas. Pueden ser de distinto tamaño.

    Ejemplo

    ```
    int[][] TablaDentada= {new int [2], new int[5]}; ó
    int[][] tabla1= new int [][] {new int[] {2,3}, new int[]
    {2,3,4,5}} ó
    int[][] tabla2= {new int[] {2,3}, new int[] {2,3,4,5}}
    tabla2[1][1]=10; //Acceso a una tabla dentada
    ```

- **Multidimensionales**:

```
int[,] tabla3=new int[2,5]; ó
int[,] tabla4=new int[2,3] {{1,2,3},{4,5,6}}; ó
int[,] tabla4=new int[,] {{1,2,3},{4,5,6}};
tabla4[1,1]=10; //Acceso
```

- **Tablas Mixtas**:

```
int[][,][] ejemplo={new int[3], new int[3,4], new int [5]};
ejemplo[1][1,1][2]=10; //Acceso
```

- **Operaciones**
  - **Asignación**: Igual que en c++;

- **Llamada a métodos**: Igual que en c++ para métodos no estáticos y **`<nombreTipo>.<nombreMétodo>(<parámentros>)`** para métodos estáticos. Un método estático se declara poniendo la palabra static delante de su declaración. Un método estático no está asociado a ningún objeto en particular sino a la clase, es decir, se puede utilizar sin instanciar ningún objeto de la clase.

- **Condicionales**:
  i. operación **`if`** igual que en c++
  ii. operación **`switch`** igual que en c++ salvo que añade además del **`break`**, la operación **`goto case <etiqueta>|<default>;`**

- **Iterativas**:

  Operación `for`: igual que en c++

  Operación `while`:igual que en c++

  Operación do **`while`**: igual que en c++

Operación `foreach`: es una variante del **`for`** que nos permite simplificar la escritura de código cuando hemos de hacer recorridos en estructuras de datos. Formato:

```
foreach (<tipoElemento> <variable> in <colección>)
<sentencias>
```

```
int[] Datos=new int[1000];
…
int S=0;
foreach (int v in Datos)
   S+=v;
System.Console.Writeline(S);
```

- **Clases**: Para una programación orientada a objetos sencilla es igual que en c++, salvo que al igual que en Java no permiten herencia múltiple y permiten el uso e implementación de interfaces. Además los métodos virtuales y el ocultamiento debe ser indicado explícitamente.

- La sobrecarga de operadores está más limitados que en c++, es decir no todos los operadores se pueden sobrecargar.
- Permite la definición de propiedades. Una propiedad es una extensión de un atributo de una clase mediante el cual el acceso a dicho atributo se hace por un conjunto de código asociado a dicho atributo. Formato:

```
<tipoPropiedad> <NombrePropiedad> {
        set {
            código de escritura
        }
        get {
            código de Lectura
        }
}
```

- **Los Parámetros de los métodos**:
    i. **De entrada**: Se definen como en c++ un parámetro para pasar un valor por copia.

ii. **De salida**: Se añade a su declaración la palabra out. Estos siempre han de modificarse dentro del código del método. En la llamada al método también hay que indicar la palabra out en el pase de parámetros, ejemplo:

```
a.f(x, out z);
```

iii. **Por referencia**: Se añade a su declaración la palabra **`ref`**. En la llamada hay que indicar la palabra **`ref`** en el pase de parámentros.

iv. **Nº indefinidos**: Hay que indicar la palabra reservada **`params`** en su declaración. Ejemplo:

```
void F(int x, params int[] valores)
{
}
```

En la llamada puede haber 0,1 o más valores para este parámetro que serán recogido en la tabla valores. Ejemplo:

```
F(4);  //0 parámetros indefinidos
F(3,2)
F(2,3,4);
```

Si queremos que puedan ser de cualquier tipo habrá que utilizar el tipo object en su declaración. Ejemplo:

```
void F(int x, params object[] valores)
{
}
```

Java and C# Comparison

Java — Program Structure — C#

```java
package hello;

public class HelloWorld {
   public static void main(String[] args) {
      String name = "Java";

      // See if an argument was passed from the command line
      if (args.length == 1)
         name = args[0];

      System.out.println("Hello, " + name + "!");
   }
}
```

```csharp
using System;

namespace Hello {
   public class HelloWorld {
      public static void Main(string[] args) {
         string name = "C#";

         // See if an argument was passed from the command line
         if (args.Length == 1)
            name = args[0];

         Console.WriteLine("Hello, " + name + "!");
      }
   }
}
```

Java — Comments — C#

```java
// Single line
/* Multiple
   line  */
/** Javadoc documentation comments */
```

```csharp
// Single line
/* Multiple
   line  */
/// XML comments on a single line
/** XML comments on multiple lines */
```

Java — Data Types — C#

Primitive Types
boolean
byte
char
short, int, long
float, double

Reference Types
Object *(superclass of all other classes)*
String
arrays, classes, interfaces

Conversions

```
// int to String
int x = 123;
String y = Integer.toString(x);  // y is "123"

// String to int
y = "456";
x = Integer.parseInt(y);   // x is 456

// double to int
double z = 3.5;
x = (int) z;   // x is 3  (truncates decimal)
```

Value Types
bool
byte, sbyte
char
short, ushort, int, uint, long, ulong
float, double, decimal
structures, enumerations

Reference Types
object *(superclass of all other classes)*
string
arrays, classes, interfaces, delegates

Convertions

```
// int to string
int x = 123;
String y = x.ToString();  // y is "123"

// string to int
y = "456";
x = int.Parse(y);   // or x = Convert.ToInt32(y);

// double to int
double z = 3.5;
x = (int) z;   // x is 3  (truncates decimal)
```

Java — Constants — C#

```
// May be initialized in a constructor
final double PI = 3.14;
```

```
const double PI = 3.14;

// Can be set to a const or a variable. May be initialized in a constructor.
readonly int MAX_HEIGHT = 9;
```

Java	Enumerations	C#

```
enum Action {Start, Stop, Rewind, Forward};

// Special type of class
enum Status {
  Flunk(50), Pass(70), Excel(90);
  private final int value;
  Status(int value) { this.value = value; }
  public int value() { return value; }
};

Action a = Action.Stop;
if (a != Action.Start)
  System.out.println(a);            // Prints "Stop"

Status s = Status.Pass;
System.out.println(s.value());      // Prints "70"
```

```
enum Action {Start, Stop, Rewind, Forward};

enum Status {Flunk = 50, Pass = 70, Excel = 90};

No equivalent.

Action a = Action.Stop;
if (a != Action.Start)
  Console.WriteLine(a);             // Prints "Stop"

Status s = Status.Pass;
Console.WriteLine((int) s);         // Prints "70"
```

Operators

Java

Comparison
== < > <= >= !=

Arithmetic
+ - * /
% *(mod)*
/ *(integer division if both operands are ints)*
Math.Pow(x, y)

Assignment
= += -= *= /= %= &= |= ^= <<= >>= >>>= ++ --

Bitwise
& | ^ ~ << >> >>>

Logical
&& || & | ^ !

Note: && and || perform short-circuit logical evaluations

String Concatenation
+

C#

Comparison
== < > <= >= !=

Arithmetic
+ - * /
% *(mod)*
/ *(integer division if both operands are ints)*
Math.Pow(x, y)

Assignment
= += -= *= /= %= &= |= ^= <<= >>= ++ --

Bitwise
& | ^ ~ << >>

Logical
&& || & | ^ !

Note: && and || perform short-circuit logical evaluations

String Concatenation
+

Java	Choices	C#

```
greeting = age < 20 ? "What's up?" : "Hello";

if (x < y)
  System.out.println("greater");

if (x != 100) {
  x *= 5;
  y *= 2;
}
else
  z *= 6;

int selection = 2;
switch (selection) {     // Must be byte, short, int, char, or enum
  case 1: x++;           // Falls through to next case if no break
  case 2: y++;   break;
  case 3: z++;   break;
  default: other++;
}
```

```
greeting = age < 20 ? "What's up?" : "Hello";

if (x < y)
  Console.WriteLine("greater");

if (x != 100) {
  x *= 5;
  y *= 2;
}
else
  z *= 6;

string color = "red";
switch (color) {                    // Can be any predefined type
  case "red":    r++;   break;      // break is mandatory; no fall-through
  case "blue":   b++;   break;
  case "green": g++;   break;
  default: other++;     break;      // break necessary on default
}
```

Loops

Java

```
while (i < 10)
  i++;

for (i = 2; i <= 10; i += 2)
  System.out.println(i);

do
  i++;
while (i < 10);

for (int i : numArray)  // foreach construct
  sum += i;

// for loop can be used to iterate through any Collection
import java.util.ArrayList;
ArrayList<Object> list = new ArrayList<Object>();
list.add(10);    // boxing converts to instance of Integer
list.add("Bisons");
list.add(2.3);    // boxing converts to instance of Double

for (Object o : list)
  System.out.println(o);
```

C#

```
while (i < 10)
  i++;

for (i = 2; i <= 10; i += 2)
  Console.WriteLine(i);

do
  i++;
while (i < 10);

foreach (int i in numArray)
  sum += i;

// foreach can be used to iterate through any collection
using System.Collections;
ArrayList list = new ArrayList();
list.Add(10);
list.Add("Bisons");
list.Add(2.3);

foreach (Object o in list)
  Console.WriteLine(o);
```

Arrays

Java

```
int nums[] = {1, 2, 3};   or   int[] nums = {1, 2, 3};
for (int i = 0; i < nums.length; i++)
  System.out.println(nums[i]);

String names[] = new String[5];
names[0] = "David";

float twoD[][] = new float[rows][cols];
twoD[2][0] = 4.5;

int[][] jagged = new int[5][];
jagged[0] = new int[5];
jagged[1] = new int[2];
jagged[2] = new int[3];
jagged[0][4] = 5;
```

C#

```
int[] nums = {1, 2, 3};
for (int i = 0; i < nums.Length; i++)
  Console.WriteLine(nums[i]);

string[] names = new string[5];
names[0] = "David";

float[,] twoD = new float[rows, cols];
twoD[2,0] = 4.5f;

int[][] jagged = new int[3][] {
    new int[5], new int[2], new int[3] };
jagged[0][4] = 5;
```

Functions

Java

```
// Return single value
int Add(int x, int y) {
   return x + y;
}

int sum = Add(2, 3);
```

```
// Return no value
void PrintSum(int x, int y) {
   System.out.println(x + y);
}

PrintSum(2, 3);
```

C#

```
// Return single value
int Add(int x, int y) {
   return x + y;
}

int sum = Add(2, 3);
```

```
// Return no value
void PrintSum(int x, int y) {
   Console.WriteLine(x + y);
}

PrintSum(2, 3);
```

Functions

Java

```
// Primitive types and references are always passed by value
void TestFunc(int x, Point p) {
   x++;
   p.x++;       // Modifying property of the object
   p = null;    // Remove local reference to object
}

class Point {
   public int x, y;
}

Point p = new Point();
p.x = 2;
int a = 1;
TestFunc(a, p);
System.out.println(a + " " + p.x + " " + (p == null) );  // 1 3 false
```

```
// Accept variable number of arguments
int Sum(int ... nums) {
  int sum = 0;
  for (int i : nums)
    sum += i;
  return sum;
}

int total = Sum(4, 3, 2, 1);   // returns 10
```

C#

```
// Pass by value (default), in/out-reference (ref), and out-reference (out)
void TestFunc(int x, ref int y, out int z, Point p1, ref Point p2) {
   x++;  y++;  z = 5;
   p1.x++;       // Modifying property of the object
   p1 = null;    // Remove local reference to object
   p2 = null;    // Free the object
}

class Point {
   public int x, y;
}

Point p1 = new Point();
Point p2 = new Point();
p1.x = 2;
int a = 1, b = 1, c;   // Output param doesn't need initializing
TestFunc(a, ref b, out c, p1, ref p2);
Console.WriteLine("{0} {1} {2} {3} {4}",
   a, b, c, p1.x, p2 == null);   // 1 2 5 3 True
```

```
// Accept variable number of arguments
int Sum(params int[] nums) {
  int sum = 0;
  foreach (int i in nums)
    sum += i;
  return sum;
}

int total = Sum(4, 3, 2, 1);   // returns 10
```

Strings

Java

```
// String concatenation
String school = "Harding ";
school = school + "University";   // school is "Harding University"

// String comparison
String mascot = "Bisons";
if (mascot == "Bisons")    // Not the correct way to do string comparisons
if (mascot.equals("Bisons"))   // true
if (mascot.equalsIgnoreCase("BISONS"))   // true
if (mascot.compareTo("Bisons") == 0)   // true

System.out.println(mascot.substring(2, 5));   // Prints "son"

// My birthday: Oct 12, 1973
java.util.Calendar c = new java.util.GregorianCalendar(1973, 10, 12);
String s = String.format("My birthday: %1$tb %1$te, %1$tY", c);

// Mutable string
StringBuffer buffer = new StringBuffer("two ");
buffer.append("three ");
buffer.insert(0, "one ");
buffer.replace(4, 7, "TWO");
System.out.println(buffer);     // Prints "one TWO three"
```

C#

```
// String concatenation
string school = "Harding ";
school = school + "University";   // school is "Harding University"

// String comparison
string mascot = "Bisons";
if (mascot == "Bisons")    // true
if (mascot.Equals("Bisons"))   // true
if (mascot.ToUpper().Equals("BISONS"))   // true
if (mascot.CompareTo("Bisons") == 0)    // true

Console.WriteLine(mascot.Substring(2, 3));    // Prints "son"

// My birthday: Oct 12, 1973
DateTime dt = new DateTime(1973, 10, 12);
string s = "My birthday: " + dt.ToString("MMM dd, yyyy");

// Mutable string
System.Text.StringBuilder buffer = new
System.Text.StringBuilder("two ");
buffer.Append("three ");
buffer.Insert(0, "one ");
buffer.Replace("two", "TWO");
Console.WriteLine(buffer);     // Prints "one TWO three"
```

Exception Handling

Java

```
// Must be in a method that is declared to throw this exception
Exception ex = new Exception("Something is really wrong.");
throw ex;

try {
  y = 0;
  x = 10 / y;
} catch (Exception ex) {
  System.out.println(ex.getMessage());
} finally {
  // Code that always gets executed
}
```

C#

```
Exception up = new Exception("Something is really wrong.");
throw up;

try {
  y = 0;
  x = 10 / y;
} catch (Exception ex) {      // Variable "ex" is optional
  Console.WriteLine(ex.Message);
} finally {
  // Code that always gets executed
}
```

Namespaces

Java

```
package harding.compsci.graphics;
```

```
import harding.compsci.graphics.Rectangle;  // Import single class
import harding.compsci.graphics.*;   // Import all classes
```

C#

```
namespace Harding.Compsci.Graphics {
  ...
}
or
namespace Harding {
  namespace Compsci {
    namespace Graphics {
      ...
    }
  }
}
```

```
// Import all class. Can't import single class.
using Harding.Compsci.Graphics;
```

Java — Classes / Interfaces — C#

Accessibility keywords
public
private
protected
static

// Inheritance
```
class FootballGame extends Competition {
  ...
}
```

// Interface definition
```
interface IAlarmClock {
  ...
}
```

// Extending an interface
```
interface IAlarmClock extends IClock {
  ...
}
```

// Interface implementation
```
class WristWatch implements IAlarmClock, ITimer {
   ...
}
```

Accessibility keywords
public
private
internal
protected
protected internal
static

// Inheritance
```
class FootballGame : Competition {
  ...
}
```

// Interface definition
```
interface IAlarmClock {
  ...
}
```

// Extending an interface
```
interface IAlarmClock : IClock {
  ...
}
```

// Interface implementation
```
class WristWatch : IAlarmClock, ITimer {
   ...
}
```

Constructors / Destructors

Java

```
class SuperHero {
  private int mPowerLevel;

  public SuperHero() {
    mPowerLevel = 0;
  }

  public SuperHero(int powerLevel) {
    this.mPowerLevel= powerLevel;
  }

  // No destructors, just override the finalize method
  protected void finalize() throws Throwable {
    super.finalize();    // Always call parent's finalizer
  }
}
```

C#

```
class SuperHero {
  private int mPowerLevel;

  public SuperHero() {
     mPowerLevel = 0;
  }

  public SuperHero(int powerLevel) {
    this.mPowerLevel= powerLevel;
  }

  ~SuperHero() {
    // Destructor code to free unmanaged resources.
    // Implicitly creates a Finalize method.
  }
}
```

Java	Objects	C#

```
SuperHero hero = new SuperHero();

hero.setName("SpamMan");
hero.setPowerLevel(3);

hero.Defend("Laura Jones");
SuperHero.Rest();  // Calling static method

SuperHero hero2 = hero;   // Both refer to same object
hero2.setName("WormWoman");
System.out.println(hero.getName());  // Prints WormWoman

hero = null;   // Free the object

if (hero == null)
  hero = new SuperHero();

Object obj = new SuperHero();
System.out.println("object's type: " + obj.getClass().toString());
if (obj instanceof SuperHero)
  System.out.println("Is a SuperHero object.");
```

```
SuperHero hero = new SuperHero();

hero.Name = "SpamMan";
hero.PowerLevel = 3;

hero.Defend("Laura Jones");
SuperHero.Rest();   // Calling static method

SuperHero hero2 = hero;   // Both refer to same object
hero2.Name = "WormWoman";
Console.WriteLine(hero.Name);   // Prints WormWoman

hero = null ;   // Free the object

if (hero == null)
  hero = new SuperHero();

Object obj = new SuperHero();
Console.WriteLine("object's type: " + obj.GetType().ToString());
if (obj is SuperHero)
  Console.WriteLine("Is a SuperHero object.");
```

Java	Properties	C#

```
private int mSize;

public int getSize() { return mSize; }
public void setSize(int value) {
  if (value < 0)
    mSize = 0;
  else
    mSize = value;
}


int s = shoe.getSize();
shoe.setSize(s+1);
```

```
private int mSize;

public int Size {
  get { return mSize; }
  set {
    if (value < 0)
      mSize = 0;
    else
      mSize = value;
  }
}

shoe.Size++;
```

Java	Structs	C#

No structs in Java.

```
struct StudentRecord {
  public string name;
  public float gpa;

  public StudentRecord(string name, float gpa) {
    this.name = name;
    this.gpa = gpa;
  }
}

StudentRecord stu = new StudentRecord("Bob", 3.5f);
StudentRecord stu2 = stu;

stu2.name = "Sue";
Console.WriteLine(stu.name);    // Prints "Bob"
Console.WriteLine(stu2.name);   // Prints "Sue"
```

Console I/O

Java	C#

```
java.io.DataInput in = new java.io.DataInputStream(System.in);
System.out.print("What is your name? ");
String name = in.readLine();
System.out.print("How old are you? ");
int age = Integer.parseInt(in.readLine());
System.out.println(name + " is " + age + " years old.");
```

```
Console.Write("What's your name? ");
string name = Console.ReadLine();
Console.Write("How old are you? ");
int age = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("{0} is {1} years old.", name, age);
// or
Console.WriteLine(name + " is " + age + " years old.");
```

```
int c = System.in.read();   // Read single char
System.out.println(c);      // Prints 65 if user enters "A"
```

```
int c = Console.Read();  // Read single char
Console.WriteLine(c);    // Prints 65 if user enters "A"
```

```
// The studio costs $499.00 for 3 months.
System.out.printf("The %s costs $%.2f for %d months.%n", "studio",
499.0, 3);
```

```
// The studio costs $499.00 for 3 months.
Console.WriteLine("The {0} costs {1:C} for {2} months.\n", "studio",
499.0, 3);
```

```
// Today is 06/25/04
System.out.printf("Today is %tD\n", new java.util.Date());
```

```
// Today is 06/25/2004
Console.WriteLine("Today is " + DateTime.Now.ToShortDateString());
```

Java — File I/O — C#

```
import java.io.*;

// Character stream writing
FileWriter writer = new FileWriter("c:\\myfile.txt");
writer.write("Out to file.\n");
writer.close();

// Character stream reading
FileReader reader = new FileReader("c:\\myfile.txt");
BufferedReader br = new BufferedReader(reader);
String line = br.readLine();
while (line != null) {
  System.out.println(line);
  line = br.readLine();
}
reader.close();

// Binary stream writing
FileOutputStream out = new FileOutputStream("c:\\myfile.dat");
out.write("Text data".getBytes());
out.write(123);
out.close();

// Binary stream reading
FileInputStream in = new FileInputStream("c:\\myfile.dat");
byte buff[] = new byte[9];
in.read(buff, 0, 9);   // Read first 9 bytes into buff
String s = new String(buff);
int num = in.read();   // Next is 123
in.close();
```

```
using System.IO;

// Character stream writing
StreamWriter writer = File.CreateText("c:\\myfile.txt");
writer.WriteLine("Out to file.");
writer.Close();

// Character stream reading
StreamReader reader = File.OpenText("c:\\myfile.txt");
string line = reader.ReadLine();
while (line != null) {
  Console.WriteLine(line);
  line = reader.ReadLine();
}
reader.Close();

// Binary stream writing
BinaryWriter out = new BinaryWriter(File.OpenWrite("c:\\myfile.dat"));
out.Write("Text data");
out.Write(123);
out.Close();

// Binary stream reading
BinaryReader in = new BinaryReader(File.OpenRead("c:\\myfile.dat"));
string s = in.ReadString();
int num = in.ReadInt32();
in.Close();
```