C# (pronunciado *si sharp* en inglés) es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma .NET, que después fue aprobado como un estándar por la ECMA (ECMA-334) e ISO (ISO/IEC 23270). C# es uno de los lenguajes de programación diseñados para la infraestructura de lenguaje común.

Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos de la plataforma .NET, similar al de Java, aunque incluye mejoras derivadas de otros lenguajes.

El nombre C Sharp fue inspirado por la notación musical, donde '#' (sostenido, en inglés *sharp*) indica que la nota (*C* es la nota do en inglés) es un semitono más alta, sugiriendo que C# es superior a C/C++. Además, el signo '#' se compone de cuatro signos '+' pegados.

Aunque C# forma parte de la plataforma .NET, ésta es una API, mientras que C# es un lenguaje de programación independiente diseñado para generar programas sobre dicha plataforma. Ya existe un compilador implementado que provee el marco Mono – DotGNU, el cual genera programas para distintas plataformas como Windows, Unix, Android, iOS, Windows Phone, Mac OS y GNU/Linux.

Durante el desarrollo de la plataforma .NET, las bibliotecas de clases fueron escritas originalmente usando un sistema de código gestionado llamado Simple Managed C (SMC). En enero de 1999, Anders Hejlsberg formó un equipo con la misión de desarrollar un nuevo lenguaje de programación llamado Cool (Lenguaje C orientado a objetos). Este nombre tuvo que ser cambiado debido a problemas de marca, pasando a llamarse C#. La biblioteca de clases de la plataforma .NET fue migrada entonces al nuevo lenguaje.

Hejlsberg lideró el proyecto de desarrollo de C#. Anteriormente, ya había participado en el desarrollo de otros lenguajes como Turbo Pascal, J++ y Embarcadero Delphi.

C# contiene dos categorías generales de tipos de datos integrados: tipos de valor y tipos de referencia. El término tipo de valor indica que esos tipos contienen directamente sus valores.

Tipos para definir números enteros:

Tipo de datos de enteros				
Tipo	**Equivalente BCL**	**Tamaño**	**Rango**	**Significado**
byte	System.Byte	8-bit (1-byte)	0 a 255	Entero sin signo
sbyte	System.SByte	8-bit (1-byte)	-128 a 127	Entero con signo
short	System.Int16	16-bit (2-byte)	-32.768 a 32.767	Entero corto con signo
ushort	System.UInt16	16-bit (2-byte)	0 a 65.535	Entero corto sin signo
int	System.Int32	32-bit (4-byte)	-2.147.483.648 a 2.147.483.647	Entero medio con signo
uint	System.UInt32	32-bit (4-byte)	0 a 4.294.967.295	Entero medio sin signo
long	System.Int64	64-bit (8-byte)	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	Entero largo con signo
ulong	System.UInt64	64-bit (8-byte)	0 a 18.446.744.073.709.551.615	Entero largo sin signo

Los tipos de coma flotante pueden representar números con componentes fraccionales. Existen dos clases de tipos de coma flotante: float y double. El tipo double es el más utilizado porque muchas funciones matemáticas de la biblioteca de clases de C# usan valores double. Quizá, el tipo de coma flotante más interesante de C# es decimal, dirigido al uso de cálculos monetarios. La aritmética de coma flotante normal está sujeta a una variedad de errores de redondeo cuando se aplica a valores decimales. El tipo decimal elimina estos errores y puede representar hasta 28 lugares decimales.

Tipo de datos de coma flotante				
Tipo	**Equivalente BCL**	**Tamaño**	**Rango**	**Significado**

`float`	`System.Single`	32-bit (4-byte)	±1.401298E?45 a ±3.402823E+38	Coma flotante corto
`double`	`System.Double`	64-bit (8-byte)	±4.94065645841246E?324 a ±1.79769313486232E+308	Coma flotante largo
`decimal`	`System.Decimal`	128-bit (16-byte)	?7.9228162514264337593543950335 a +7.9228162514264337593543950335	Coma flotante monetario

Los caracteres en C# no tienen un tamaño de 8 bits como en otros muchos lenguajes de programación, sino que usa un tamaño de 16 bits llamado Unicode al cual se le llama `char`. No existen conversiones automáticas de tipo entero a `char`.

Tipo de datos de caracteres				
Tipo	**Equivalente BCL**	**Tamaño**	**Rango**	**Significado**
`char`	`System.Char`	16-bit (2-byte)	`'\u0000'` a `'\uFFFF'`	Carácter unicode

Para los tipos de datos lógicos no existen conversiones automáticas de tipo entero a `bool`.

Tipo de datos lógicos				
Tipo	**Equivalente BCL**	**Tamaño**	**Rango**	**Significado**
`bool`	`System.Boolean`	8-bit (1-byte)	`true` o `false`	Verdadero o falso

Literales

En ocasiones, resulta más sencillo usar un sistema numérico en base 16 en lugar de 10, para tal caso C# permite especificar números enteros en formato hexadecimal, y se define anteponiendo`0x`, por ejemplo: `0xFF`, que equivale a 255 en decimal.

C# tiene caracteres denominados *secuencias de escape* para facilitar la escritura con el teclado de símbolos que carecen de representación visual.

C#, al igual que C++, define un tipo de cadena de caracteres. Dentro de la cadena de caracteres se pueden usar secuencias de escape. Una cadena de caracteres puede iniciarse con el símbolo@ seguido por una cadena entre comillas ("), en tal caso, las secuencias de escape no tienen efecto, y además la cadena puede ocupar dos o más líneas.

Enteros	
decimal	`245, [0..9]+`
hexadecimal	`0xF5, 0x[0..9, A..F, a..f]+`
entero largo	`12L`
entero largo sin signo	`654UL`
Coma flotante	
float	`23.5F, 23.5f; 1.72E3F, 1.72E3f, 1.72e3F, 1.72e3f`
double	`23.5, 23.5D, 23.5d, 1.72E3, 1.72E3D`
decimal	`9.95M`
Caracteres	
char	`'a', 'Z', '\u0231'`
Cadenas	
String	`"Hello, world"; "C:\\Windows\\", @"C:\Windows\"`
Secuencias de escape	
Alerta (timbre)	`\a`
Retroceso	`\b`
Avance de página	`\f`
Nueva línea	`\n`
Retorno de carro	`\r`
Tabulador horizontal	`\t`
Tabulador vertical	`\v`
Nulo	`\0`
Comilla simple	`\'`
Comilla doble	`\"`

Barra inversa	\\

Variables

Las variables son identificadores asociados a valores. Se declaran indicando el tipo de dato que almacenará y su identificador.

Un identificador puede:

- empezar por «_».
- contener caracteres Unicode en mayúsculas y minúsculas (sensible a mayúsculas y minúsculas).

Un identificador no puede:

- empezar por un número.
- empezar por un símbolo, ni aunque sea una palabra clave.
- contener más de 511 caracteres.

Declarar una variable:

```
int miNumero; // Declaramos la variable, pero no la inicializamos con ningún valor.
```

Para asignar un valor a una variable, se indica el identificador de la misma, seguido del símbolo igual (=) y el valor que queremos que almacene:

```
miNumero = 5; // Asignamos el valor '5'.
```

También se puede declarar y asignar un valor al mismo tiempo:

```
int miNumero = 5; // Declaramos la variable, y asignamos el valor '5'.
```

Las conversiones de tipo de variables en C# se representan en la siguiente tabla en donde la fila es el origen y la columna el destino.

Leyenda	
Rojo	Conversión incompatible (I).
Verde	Conversión automática o implícita (A).
Azul	Conversión explícita (E).

Conversiones de tipo de datos													
	byte	**sbyte**	**short**	**ushort**	**int**	**uint**	**long**	**ulong**	**float**	**double**	**decimal**	**char**	**bool**
byte		E	A	A	A	A	A	A	E	E	E	E	I
sbyte	E		A	E	A	E	A	A	E	E	E	E	I
short	E	E		E	A	A	A	A	E	E	E	E	I
ushort	E	E	E		A	A	A	A	E	E	E	E	I
int	E	E	E	E		E	A	A	E	E	E	E	I

uint	E	E	E	E	E		A	A	E	E	E	E	I
long	E	E	E	E	E	E		E	E	E	E	E	I
ulong	E	E	E	E	E	E	E		E	E	E	E	I
float	E	E	E	E	E	E	E	E		A	E	I	I
double	E	E	E	E	E	E	E	E	E		E	I	I
decimal	E	E	E	E	E	E	E	E	E	E		I	I
char	E	E	E	A	A	A	A	A	A	A	A		I
bool	I	I	I	I	I	I	I	I	I	I	I	I	

- Toda conversión implícita no ocasiona pérdida de información, truncamientos o redondeos.
- Es posible (mas no siempre ocurre) que en una conversión explícita haya pérdida de información, truncamientos o redondeos.
- En toda conversión implícita el tipo de dato destino es mayor que el tipo de dato origen.
- La conversión explícita se realiza indicando el tipo de dato al que se quiere convertir entre paréntesis, seguido del valor:

```
long valor = 123; // Conversión implícita
long valor = (long)123; // Conversión explícita
```

Además de realizarse dentro de una asignación, las conversiones de tipos también tienen lugar dentro de una expresión, pues en cada operación ambos operandos deben de ser del mismo tipo. Si la conversión es del tipo implícito se efectúa el siguiente algoritmo en dicho orden:

1. Si un operando es `decimal`, el otro operando se transforma a `decimal`.

2. Si un operando es `double`, el otro operando se transforma a `double`.
3. Si un operando es `float`, el otro operando se transforma a `float`.
4. Si un operando es `ulong`, el otro operando se transforma a `ulong`.
5. Si un operando es `long`, el otro operando se transforma a `long`.
6. Si un operando es `uint`, y si el otro operando es de tipo `sbyte`, `short` o `int`, los dos se transforman a `long`.
7. Si un operando es `uint`, el otro operando se transforma a `uint`.
8. Si no es ninguno de los casos anteriores, los dos operandos se transforman a `int`.

Las constantes son valores inmutables, y por tanto no se pueden cambiar.

`const`

Cuando se declara una constante con la palabra clave `const`, también se debe asignar el valor. Tras esto, la constante queda bloqueada y no se puede cambiar. Son implícitamente estáticas (`static`).

```
const double PI = 3.1415;
```

`readonly`

A diferencia de `const`, no requiere que se asigne el valor al mismo tiempo que se declara. Pueden ser miembros de la instancia o miembros estáticos de la clase (`static`).

```
readonly double E;
```

```
    E = 2.71828;
```

Operadores

Categoría	Operadores
Aritmético	+ - * / %
Lógico y a nivel de bits	^ ! ~ && ||
Concatenación	+
Incremento, decremento	++ --
Desplazamiento	<< >>
Relacional	== != < > <= >=
Asignación	= ^= <<= >>=
Acceso a miembro	.
Indexación	[ ]
Conversión	( )
Condicional	? :
Creación de objeto	new
Información de tipo	as is sizeof typeof

- Los operadores aritméticos funcionan igual que en C y C++.
- El resultado de los operadores relacionales y lógicos es un valor de tipo `bool`.
- Los operadores de cortocircuito evalúan el segundo operando solo cuando es necesario.
- Los operadores a nivel de bits no se pueden aplicar a tipos `bool`, `float`, `double` o `decimal`.

Instrucciones de control

`if-else`

```
if (i == 2) {
    // ...
}
else if (i == 3) {
    // ...
}
else {
    // ...
}
```

`switch`

```
switch (i) {
    case 1:
        ...
        break;
    case 2:
    case 3:
        ...
        break;
```

```
        default:
            ...
            break;
    }
```

for

```
    for (int i = 0; i < 10; i++) {
        // ...
    }
```

while

```
    while (i < 10) {
        // ...
    }
```

do-while

```
do {
    // ...
} while (true);
```

`foreach`

```
foreach (char c in charList) {
    // ...
}
```

- Las instrucciones `if-else`, `for`, `while`, `do-while`, `switch`, `return`, `break`, `continue` son, básicamente, iguales que en C, C++ y Java.
- La instrucción `foreach`, al igual que en Java, realiza un ciclo a través de los elementos de una matriz o colección. En este ciclo se recorre la colección y la variable recibe un elemento de dicha colección en cada iteración.
- La instrucción `goto` se sigue utilizando en C# a pesar de la polémica sobre su uso.

Métodos

- Todo método debe ser parte de una clase, no existen métodos globales (funciones).
- Por defecto, los parámetros se pasan por valor. (Nótese que las listas y otras colecciones son variables *por referencia* (referencias al espacio reservado para esa

lista en la pila) y que se pasa por valor al método la referencia , pero el espacio reservado para la lista es común, por lo que si elimina un elemento lo hace también de la original)

- El modificador `ref` fuerza a pasar los parámetros por referencia en vez de pasarlos por valor y obliga a inicializar la variable antes de pasar el parámetro.
- El modificador `out` es similar al modificador `ref`, con la diferencia de que no se obliga a inicializar la variable antes de pasar el parámetro.
- Cuando `ref` y `out` modifican un parámetro de referencia, la propia referencia se pasa por referencia.
- El modificador `params` sirve para definir un número variable de argumentos los cuales se implementan como una matriz.
- Un método debe tener como máximo un único parámetro `params` y éste debe ser el último.
- Un método puede devolver cualquier tipo de dato, incluyendo tipos de clase.
- Ya que en C# las matrices se implementan como objetos, un método también puede devolver una matriz (algo que se diferencia de C++ en que las matrices no son válidas como tipos de valores devueltos).
- C# implementa *sobrecarga de métodos*, dos o más métodos pueden tener el mismo nombre siempre y cuando se diferencien por sus parámetros.
- El método `Main` es un método especial al cual se refiere el punto de partida del programa.

`ref`

```
void PassRef(ref int x) {
    if (x == 2) {
        x = 10;
    }
}
```

```
    int z = 0;
    PassRef(ref z);
```

out

```
    void PassOut(out int x) {
        x = 2;
    }

    int z;
    PassOut(out z);
```

params

```
    int MaxVal(char c, params int[] nums) {
        // ...
    }

    int a = 1;
    MaxVal('a', 23, 3, a, -12); // El primer parámetro es
    obligatorio, seguidamente se pueden poner tantos números
```

```
    enteros como se quiera
```

Sobrecarga de métodos

```
int Suma(int x, int y) {
    return x + y;
}
int Suma(int x, int y, int z) {
    return x + y + z;
}

Suma(1, 2); // Llamará al primer método.
Suma(1, 2, 3); // Llamará al segundo método.
```

Main

```
public static void Main(string[] args) {
    // ...
```

```
}
```

Matrices

- En C# las matrices se implementan como objetos.
- Los índices de las matrices comienzan en 0.
- Ya que C# implementa las matrices como objetos, cada matriz tiene una propiedad `Length` que contiene el número de elementos que puede alojar o tiene alojados.

Declarar una matriz:

```
int[] intArray = new int[5];
```

Declarar e inicializar una matriz (el tamaño de la matriz se puede omitir):

```
int[] intArray = new int[] {1, 2, 3, 4, 5};
```

Acceder a un elemento:

```
intArray[2]; // Retornará el valor '3'
```

Declarar una matriz multidimensional:

```
int[,] intMultiArray = new int[3, 2]; // 3 filas y 2 columnas
```

Declarar e inicializar una matriz multidimensional (el tamaño de la matriz se puede omitir):

```
int[,] intMultiArray = new int[,] { {1, 2}, {3, 4}, {5, 6} };
```

Acceder a un elemento de una matriz multidimensional:

```
intMultiArray[2, 0]; // Retornará el valor '5'
```

Más información en: Tutorial de matrices (C#) (en inglés).

Clases y objetos

- Una variable de objeto de cierta clase no almacena los valores del objeto sino su referencia (al igual que Java).
- El operador de asignación no copia los valores de un objeto, sino la referencia al mismo (al igual que Java).
- Un constructor tiene el mismo nombre que su clase y es sintácticamente similar a un método.
- Un constructor no devuelve ningún valor (ni siquiera void).
- Al igual que los métodos, los constructores también pueden ser sobrecargados.
- Si no se especifica un constructor en una clase, se usa uno por defecto que consiste en asignar a todas las variables el valor `0`, `null` o `false` según corresponda.
- Cuando un objeto no es referenciado por ninguna variable, el recolector de basura ejecuta el destructor de dicha clase y libera la memoria utilizada.
- El destructor de una clase no se llama cuando un objeto sale del ámbito.
- Todos los destructores se llamarán antes de que finalice un programa.
- La palabra clave `this` es una referencia al mismo objeto en el cual se usa.
- La palabra clave `base` es una referencia a la clase padre del objeto en la que se usa (por defecto, Object).
- La palabra clave `static` hace que un miembro pertenezca a una clase en vez de pertenecer a objetos de dicha clase. Se puede tener acceso a dicho miembro antes de que se cree cualquier objeto de su clase y sin referencias a un objeto.
- Un método `static` no tiene una referencia `this`.
- Un método `static` puede llamar sólo a otros métodos `static`.
- Un método `static` sólo debe tener acceso directamente a datos `static`.
- Un constructor `static` se usa para inicializar atributos que se aplican a una clase en lugar de aplicarse a una instancia.
- C# permite la sobrecarga de operadores (+, -, *, etc.) con la palabra clave `operator`.
- Al comparar objetos (==, !=, <, >, <=, >=) se comprueba si hacen referencia al mismo objeto.

Declarar una clase:

```
class Clase {
}
```

Iniciar una clase (también llamado *crear un objeto de la clase* o *instanciar una clase*):

```
Clase c = new Clase();
```

Constructor (como si fuera un método, pero con el nombre de su clase):

```
class Clase {
    Clase() {
        // ...
    }
}
```

Destructor (como si fuera un método, precedido del símbolo '~'):

```
class Clase {
    ~Clase() {
        // ...
    }
}
```

this:

```
class Clase {
    int i = 1;
    Clase() {
         this.Arrancar(); // Llamará al método 'Arrancar' del
objeto
    }
    void Arrancar() {
        // ...
    }
}
```

static:

```
class Clase {
    static int i = 1;
}
Clase.i; // Retornará el valor '1'. No hace falta crear un
objeto, ya que al ser 'static', pertenece a la clase.
```

operator:

```
class Clase {
    static int operator +(int x, int y) {
        // Sobrecarga el operador '+'
        // ...
    }
    static int operator -(int x, int y) {
        // Sobrecarga el operador '-'
        // ...
    }
    static int operator int(byte x) {
        // Sobrecarga la conversión de tipo 'byte' a 'int'
        // ...
    }
```

```
}
```

Comparación de objetos:

```
class Clase {
}
Clase c1 = new Clase();
Clase c2 = new Clase();
bool b = c1 == c2; // Retornará 'false', ya que son dos
objetos distintos
```

Cadenas de caracteres

- El tipo de dato para las cadenas de caracteres es `string`.
- Realmente la palabra clave `string` es un alias de la clase `System.String` de la plataforma .NET.
- En C# las cadenas son objetos y no una matriz de caracteres; aun así, se puede obtener un carácter arbitrario de una cadena por medio de su índice (mas no modificarlo).
- Las cadenas son inmutables, una vez creadas no se pueden modificar, solo se pueden copiar total o parcialmente.
- El operador == determina si dos referencias hacen referencia al mismo objeto, pero al usar dicho operador con dos variables tipo `string` se prueba la igualdad del contenido de las cadenas y no su referencia. Sin embargo, con el resto de los

operadores relacionales, como < y >=, sí se comparan las referencias.
- Se pueden concatenar (unir) dos cadenas mediante el operador +.
- Las cadenas se pueden usar en las instrucciones `switch`.

Declarar una cadena de caracteres (como si fuera una variable de un tipo de dato como `int` o `double`):

```
string texto = "Cadena de caracteres";
string texto = new System.String("Cadena de caracteres"); //
Equivalente al anterior
```

Longitud de una cadena:

```
string texto = "Cadena de caracteres";
int i = texto.Length; // Retornará '20'
```

Comparar dos cadenas:

```
bool b = "texto" == "texto"; // Retornará 'true', ya que ambas
```

```
    cadenas contienen "texto"
```

Concatenar cadenas:

```
    string texto = "Cadena de" + " caracteres"; // Nótese el
    espacio antes de "caracteres", si no se pusiera, aparecería
    junto: "decaracteres"
```

La clase `System.String`, y una instancia de la misma, `string`, poseen algunos métodos para trabajar con cadenas, como:

`static string Copy(string str)`: devuelve una copia de `str`.

```
    string texto1 = "abc";
    string texto2 = "xyz";
    string texto2 = System.String.Copy(texto1); // 'texto2' ahora
    contiene "abc"
```

`int CompareTo(string str)`: devuelve menor que cero si la cadena que llama es menor que str, mayor que cero si la cadena que llama es mayor que str, y cero si las cadenas son

iguales.

```
string texto = "abc";
int i = texto.CompareTo("abc"); // Retornará '0'
```

`int IndexOf(string str)`: devuelve el índice de la primera coincidencia de la subcadena especificada en `str`, o -1 en caso de error.

```
string texto = "abcdefabcdef";
int i = texto.IndexOf("e"); // Retornará '4'
int j = texto.IndexOf("def"); // Retornará '3', que es donde
se encuentra el carácter 'd', seguido de 'e' y 'f'
```

`int LastIndexOf(string str)`: devuelve el índice de la última coincidencia de la subcadena especificada en `str`, o -1 en caso de error.

```
string texto = "abcdefabcdef";
int i = texto.LastIndexOf("e"); // Retornará '10'
int j = texto.LastIndexOf("def"); // Retornará '9', que es
donde se encuentra el último carácter 'd', seguido de 'e' y
```

```
    'f'
```

`string ToLower`: devuelve una copia de la cadena en minúsculas.

```
    string texto = "ABC";
    string texto = texto.ToLower(); // Retornará "abc"
```

`string ToUpper`: devuelve una copia de la cadena en mayúsculas.

```
    string texto = "abc";
    string texto = texto.ToUpper(); // Retornará "ABC"
```

`string Substring`: devuelve una subcadena, indicando la posición de inicio y la longitud que se desea.

```
    string texto = "Cadena de caracteres";
```

```
string texto = texto.Substring(10, 8); // Retornará "caracter"
```

Más información en: String (Clase) (System) (en inglés).

Ejemplos

Ejemplo básico «Hola mundo»:

```
using System;

public class Ejemplo {
    public static void Main(string[] args) {
        Console.WriteLine("Hola mundo");
    }
}
```

Suma y concatenación:

```
using System;

public class Ejemplo {
    public static void Main(string[] args) {
```

```
            int x = 10;
            int y = 20;
            Console.WriteLine("El resultado es: " + (x + y)); //
    Imprimirá en pantalla: "El resultado es: 30"
        }
    }
```

Uso de clases, métodos, propiedades y sobrecarga:

```
    using System;

    public class Coche {
        private int numPuertas;
        public int NumPuertas {
            get {
                return this.numPuertas;
            }
            set {
                  this.numPuertas = value; // 'value' es una
    variable que se asigna automáticamente al asignar un valor a
    la propiedad, para poder trabajar con dicho valor.
            }
        }

        public Coche(int numPuertas) {
            this.NumPuertas = numPuertas;
```

```
    }

    // Sobrecarga: si se instancia la clase sin indicar ningún
parámetro, se inicializa 'numPuertas' con el valor '2'
    public Coche() : this(2) {
    }
}

public class Ejemplo {
    public static void Main(string[] args) {
         Coche coche = new Coche(); // Se usa el segundo
constructor
        coche.NumPuertas = 4;
         Console.WriteLine("El número de puertas es: " +
coche.NumPuertas);
    }
}
```

Compiladores

En la actualidad existen los siguientes compiladores para el lenguaje C#:

- Microsoft .NET Framework 2.0 (SDK) incluye un compilador de C#, pero no un IDE.
- Microsoft Visual Studio, IDE por excelencia de este lenguaje.
- SharpDevelop, IDE libre para C# bajo licencia GNU LGPL, con una interfaz muy similar a Microsoft Visual Studio.
- Mono, es una implementación con licencia GNU GPL de todo el entorno .NET desarrollado por Novell. Como parte de esta implementación se incluye

un compilador de C#.
- Delphi 2006, de Borland Software Corporation.
- DotGNU Portable.NET, de la Free Software Foundation.

Metas del diseño del lenguaje

El estándar ECMA-334 lista las siguientes metas en el diseño para C#:

- Lenguaje de programación orientado a objetos simple, moderno y de propósito general.
- Inclusión de principios de ingeniería de software tales como revisión estricta de los tipos de datos, revisión de límites de vectores, detección de intentos de usar variables no inicializadas, y recolección de basura automática.
- Capacidad para desarrollar componentes de software que se puedan usar en ambientes distribuidos.
- Portabilidad del código fuente.
- Fácil migración del programador al nuevo lenguaje, especialmente para programadores familiarizados con C, C++ y Java.
- Soporte para internacionalización.
- Adecuación para escribir aplicaciones de cualquier tamaño: desde las más grandes y sofisticadas como sistemas operativos hasta las más pequeñas funciones.
- Aplicaciones económicas en cuanto a memoria y procesado.