Variable

- Una variable es un objeto que tiene asociado un valor de un tipo determinado.
- A pesar de que al declarar cualquier variable se le asigna un valor por defecto, deben inicializarse (asignárseles un valor) antes de ser usadas por primera vez.
- Las variables definidas dentro de un bloque { } o dentro de un método sólo son visibles dentro de éstos.

Declaración variable

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

Tipos de datos

Nombre corto	Clase .NET	Tipo	Ancho	Intervalo (bits)
byte	Byte	Entero sin signo	8	0 a 255
sbyte	SByte	Entero con signo	8	-128 a 127
int	Int32	Entero con signo	32	-2.147.483.648 a 2.147.483.647
uint	UInt32	Entero sin signo	32	0 a 4294967295
short	Int16	Entero con signo	16	-32.768 a 32.767
ushort	UInt16	Entero sin signo	16	0 a 65535
long	Int64	Entero con signo	64	-9223372036854775508 a 9223372036854775507
ulong	UInt64	Entero sin signo	64	0 a 18446744073709551615
float	Single	Tipo de punto flotante de precisión simple	32	-3,402823e38 a 3,402823e38

Tipos de datos

double	Double	Tipo de punto flotante de precisión doble	64	-1,79769313486232e308 a 1,79769313486232e308
char	Char	Un carácter Unicode	16	Símbolos Unicode utilizados en el texto
bool	Boolean	Tipo Boolean lógico	8	True o false
object	Object	Tipo base de todos los otros tipos		
string	String	Una secuencia de caracteres		
decimal	Decimal	Tipo preciso fraccionario o integral, que puede representar números decimales con 29 dígitos significativos	128	±1.0 × 10e−28 a ±7.9 × 10e28

Constantes

- Una constante es otro tipo de campo. Contiene un valor que se asigna cuando se compila el programa y nunca cambia después. Las constantes se declaran con la palabra clave const;
- Son útiles para que el código sea más legible.

```
const int speedLimit = 55;
const double pi = 3.14159265358979323846264338327950;
```

Enumeraciones

- Las enumeraciones son utilizadas para agrupar constantes. El ejemplo siguiente muestra el uso de enumeradores en Visual C#.

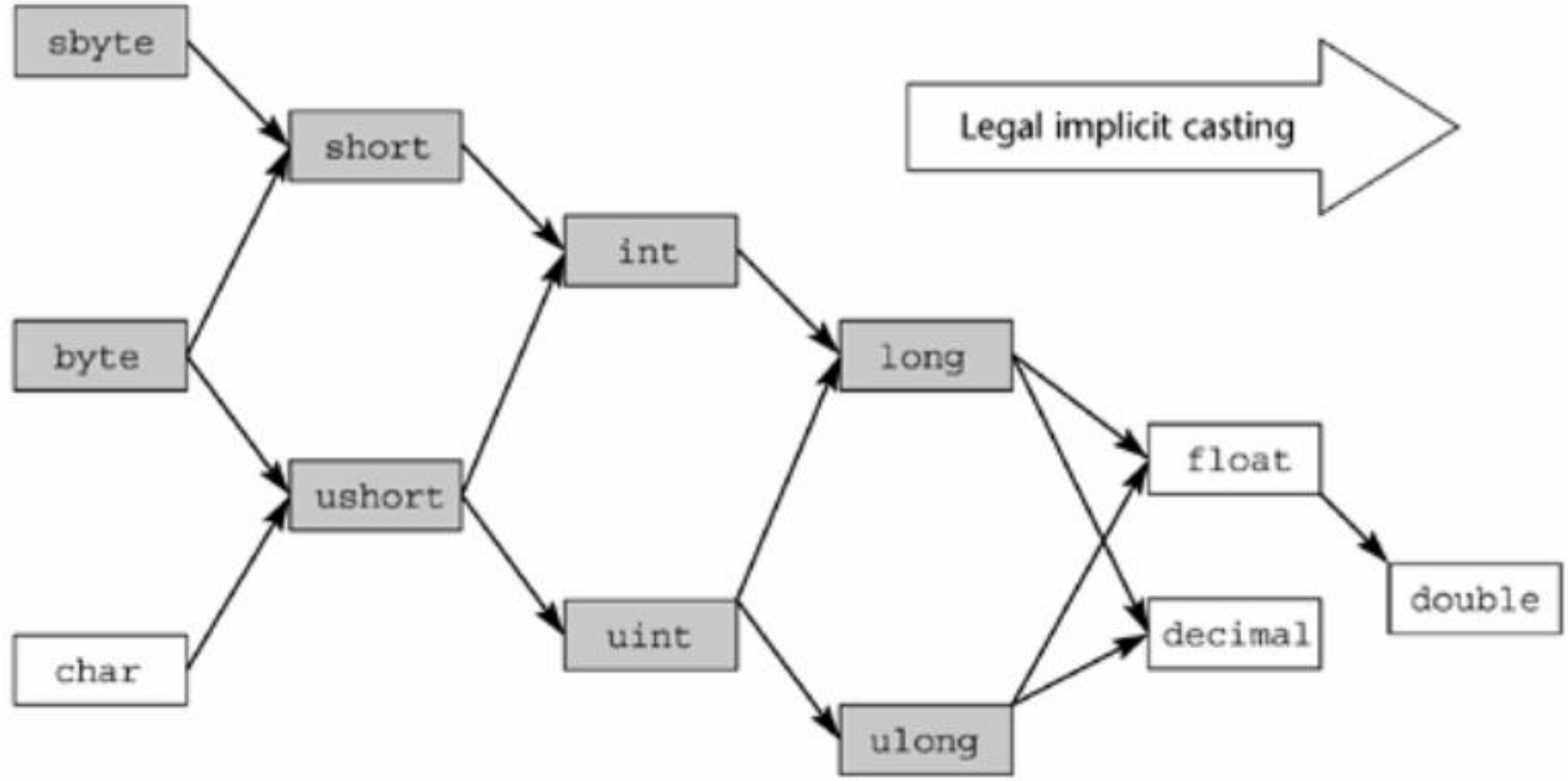
```
public enum Color
{
    Green,   //defaults to 0
    Orange,  //defaults to 1
    Red,     //defaults to 2
    Blue     //defaults to 3
}
```

```
public enum Color2
{
    Green = 10,
    Orange = 20,
    Red = 30,
    Blue = 40
}
```

Conversiones

- Objetos de un tipo pueden convertirse a otro tipo, implícita o explícitamente.
- Las conversiones implícitas entre tipos básicos son automáticas: el compilador las realiza y garantiza que no haya pérdida de información.
- Las conversiones explícitas ocurren cuando se especifica un nuevo tipo entre paréntesis delante de una expresión. Puede existir pérdida de información

Conversiones implícitas validas



Conversión Implícita

```
// Implicit conversion. num long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

Conversión Explicita

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

Operadores Aritméticos

Operador	Descripción	Ejemplo	Resultado
+	Suma	int a=6+2;	8
-	Resta	int a=6-2;	4
*	Multiplica	int a=6*2;	12
/	Divide	int a=6/2;	3
%	Modulo (cociente de la división)	int a= 6%2;	0

Operadores relacionales y de tipo

Operador	Descripción	Ejemplo	Resultado
<	Menor que	5<7	true
>	Mayor que	5>7	false
<=	Menor o igual que	5<=7	true
>=	Mayor o igual que	5>=7	false

Operadores de igualdad

Operador	Descripción	Ejemplo	Resultado
==	Igualdad	(5==5)	true
¡=	Desigualdad	(6=!5)	true

Operadores lógicos y condicionales

Operador	Descripción	Ejemplo	Resultado
&& (AND)	Evalúa dos operadores o expresiones	(a && b)	Según el resultado de a (si es true) evalúa b.
|| (OR)	Evalúa dos operadores o expresiones	(a || b)	Evalúa a o b buscando cuál de las dos es true

Estructuras de control

- **Estructuras de decisión.**
- **Estructuras de bucles.**

Instrucciones Condicionales

```
if (expresión)
     instrucción1;

if (expresión)
     instrucción1;
else
     instrucción2;
```

- La parte de la expresión debe retornar un valor booleano (*no* un entero)
- Aplica la evaluación por corto circuito
- Puede usarse `else if` para abreviar
- Todas las instrucciones de control de flujo pueden ponerse dentro de bloques { }, pero son obligatorios si se tiene más de una instrucción
- Pueden anidarse

If

```
01 if(x < 10)     //Hacer algo se ejecuta si x es menor que 10
02 HacerAlgo();
03
04 //******
05
06 if(x < 10)    // En este caso es igual al anterior,
07 {                //  pero al ser más de una instrucción,
08  HacerAlgo();   // debemos colocarlos entre llaves ("{" "}")
09  HacerAlgo2();
10 }
11
12 //******
13
14 if(x < 10)
15 {              //es indistinto el uso de llaves
16 Hacer();       //en este ejemplo ya que es solo una linea
17 }
18 else           //Si la condicion anterior no es verdadera  (x < 10)
19 {              //se ejecuta  Hacer2() y no Hacer()
20 Hacer2();
21 }
```

Switch

```
01 switch(entero)
02 {
03   case 1:
04           //código que queremos ejecutar en caso de que entero sea igual a 1
05           break; //Esta instruccion hace que salgamos del switch
06   case 2:
07           //código
08           break;
09   default:
10           //codigo que querramos hacer en caso de que el "entero" sea
11           //distintos a los anteriores casos
12           break;
13
14 }
```

Instrucciones de Iteración

```
while (expresión)
     instrucción1;

do
     instrucción1;
while (expresión);

for ([inicializador,]; [expresión]; [iterador])
     instrucción1;
```

- Adicionalmente, se pueden implementar ciclos usando `goto`
- Un `do … while` entra al ciclo al menos una vez
- Las variables de inicialización declaradas en un `for` sólo pueden ser usadas dentro de éste
- Se puede usar `break` para salirse prematuramente de un ciclo
- Se puede usar `continue` para saltarse una iteración dentro de un ciclo
- `break` y `continue` crean múltiples puntos de salida dentro de una instrucción de iteración. Deben usarse con cuidado

For

```
1 for(int i=0;i<10;i++)  //partes: declaración(i = 0) ;
2 {                       //prueba o condición (i < 10);
3  //codigo               //acción(i++), aumenta i en 1
4 }
```

Foreach

```
1  string[] nombre = new string[10];
2  //Esto lo veremos más adelante
3  foreach (string auxNombre in nombre)  //Foreach permite recorrer arreglos
4  {                                     // y colecciones
5     //auxNombre es un elemento de nombre
6  }
```

While

```
1  bool condicion = true;
2
3  while(condicion == true)
4  {
5  //codigo
6  //en algún momento poner condicion = false;
7  // o si no será un buecle infinito
8  }
```

Do While

```
01 bool condicion = true;
02
03 do
04 {
05 //codigo
06 //en algún momento poner condicion = false;
07 // o si no será un buecle infinito
08 }while(condicion==true);
09
10 //La unica diferencia entre while y do-while,
11 //es que en while la condicion debe ser verdadera para entrar
12 //en el bucle. Por ejemplo si cuando llega al while y condicion es falso,
13 //el programa salta todo el while. O sea no entra.
14 //En cambio en el do-while, el programa entra al menos una vez, y al final
15 //inspecciona la condicion
```

Vectores y matrices

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

```
string[] stringArray = new string[6];
```

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

```
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

Operaciones de cadena

Función	Descripción
Concat	Permite concatenar dos cadenas en una sola.
Contains	Devuelve un valor indicando si una cadena especificada se encuentra dentro de otra cadena.
Replace	Reemplaza una cadena buscada dentro de otra, tantas veces como aparezca.
StartsWith	Determina si el principio de una cadena coincide con una cadena especificada.
Substring	Recupera una subcadena a partir de otra cadena.
ToString	Convierte una instancia de objeto en un objeto String.
Remove	Elimina un número de caracteres especificado en una cadena.
ToUpper	Convierte a mayúscula una cadena especificada.

Operaciones matemáticas

Función	Descripción
Log10	Devuelve el logaritmo en base 10 de un número especificado.
Max(Int32, Int32)	Devuelve el mayor de dos números enteros de 32 bits.
Min(Int32, Int32)	Devuelve el menor de dos números enteros de 32 bits.
Pow()	Devuelve la potencia de un número.
Sin()	Devuelve el seno de un ángulo especificado.
Sqrt()	Devuelve la raíz cuadrada de un número.
Tan()	Devuelve la tangente de un ángulo especificado.

Operaciones de fecha

Función	Descripción
DateTime.Now	Devuelve la fecha y hora actual del sistema.
DateAddDays	Devuelve una fecha a la cual se le agrego un intervalo de tiempo en días.
DateAddMonths	Devuelve una fecha a la cual se le agrego un intervalo de tiempo en meses
Date.ToShortDateString();	Devuelve la fecha corta del sistema en formato dd/mm/yyyy

Excepciones

- No confundir: defectos, errores, excepciones.
- En C#, las condiciones anormales pueden ser manejadas por medio de excepciones: objetos que almacenan información sobre situaciones inusuales ocurridas durante la ejecución de un programa.
- Todas las excepciones son no-chequeadas y lanzadas en tiempo de ejecución

Excepciones

- En otras palabras: los métodos no las declaran explícitamente como parte de su firma usando throws, ni es necesario capturarlas.

- Si una excepción en un método no es capturada localmente, es propagada al contexto desde donde se llamó al método y así sucesivamente hasta que sea capturada o llegue hasta el manejador de excepciones por defecto, que termina el programa

Lanzando Excepciones

```
try { }
[ catch [(tipo [variable])] { } ]
[ finally { } ]
```

- Para señalar una condición anormal en un programa, se lanza una excepción usando la palabra clave `throw` en *cualquier* parte del código de un método, aún en bloques `catch` o `finally`

  ```
  throw new Exception();
  ```

- Todas las excepciones son instancias de `System.Exception` o una de sus sub-clases
- Las excepciones se manejan así:
  1. Se rodea el código que puede lanzar una excepción dentro de un bloque `try`,
  2. Se capturan las excepciones dentro de bloques `catch` y/o
  3. Se declara un bloque `finally`
- Los bloques `try` pueden anidarse unos dentro de otros, dentro de un bloque `catch` o dentro de un bloque `finally`

Capturando Excepciones

- La forma más general de capturar una excepción lanzada desde un bloque `try`, es usando un `catch` *solo*. Si se captura `Exception`, es posible que no se capturen excepciones en lenguajes distintos a C#
- No es necesario declarar una variable cuando no se necesita usar la excepción capturada en un bloque `catch`. Si debe ser lanzada nuevamente, se usa un `throw` *solo*
- Es posible capturar sólo algunas de las excepciones lanzadas en un bloque `try` y permitir que las otras se propaguen
- El orden en que se capturan las excepciones *sí* importa: se deben capturar primero las que están más abajo en la jerarquía de herencia, de lo contrario serían inalcanzables

Instrucción finally

- Usada cuando hay una acción que se debe ejecutar en un método, se lance o no una excepción (Ej.: cerrar un archivo)
- Se garantiza que los bloques `finally` siempre se ejecutan
- Se necesita definir un bloque `try` para usar un bloque `finally`, pero no se necesita que hayan bloques `catch`
- Las siguientes instrucciones *no* se pueden usar en un bloque `finally`: `break`, `continue`, `return` o `goto`.

```
class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new System.DivideByZeroException();
        return x / y;
    }
    static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result = 0;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException e)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

CamelCase

CamelCase es un estilo de escritura que se aplica a frases o palabras compuestas.

Existen dos tipos de CamelCase:

- **UpperCamelCase**, cuando la primera letra de cada una de las palabras es mayúscula.
  Ejemplo: *EjemploDeUpperCamelCase*.
- **lowerCamelCase**, igual que la anterior con la excepción de que la primera letra es minúscula.
  Ejemplo: *ejemploDeLowerCamelCase*.