3. La abstracción procedimental

En diseño descencente...

Hemos visto que en *diseño descendente* los problemas:

- se descomponen en subproblemas más sencillos.
  Por ejemplo, para el ejercicio de los árboles de navidad hacíamos:
  - petición de altura de árbol
  - escritura de cada nivel del árbol; para cada nivel:
    - escritura de los blancos previos a los asteriscos
    - escritura de los asteriscos
    - salto de línea e incremento del contador
- cada subproblema se resuelve con independencia del resto: abstracción
  Para escribir los k blancos previos a los asteriscos nos abstraemos de la escritura de los asteriscos, o de la recogida del dato *altura* del árbol
- La solución al problema original resulta de la combinación de las soluciones a los subproblemas.

Los programas crecen: abstracción

Esta forma de abstracción puede plasmarse en la implementación mediante

subprogramas: bloques de código independientes (pero relacionados)

$\rightsquigarrow$ métodos (o funciones) en C#

Hasta ahora hemos escrito todo nuestro código en un solo método: `Main`, pero hemos hemos utilizado otros métodos (ya definidos):

`WriteLine`, `ReadLine`, `int.Parse`, `Math.Sqrt`,...

Nos hemos abstraído de cómo operan esos métodos y los hemos utilizado sin mayor problema: alguien los ha programado por nosotros.
Ahora aprenderemos a implementar nuestros propios métodos.

Ventajas?

Muchas:

- Claridad de código: nos permiten trocear grandes programas en fragmentos independientes más pequeños y fáciles programar (divide y vencerás!) $\rightsquigarrow$ organización de código.
  - Facilidad de diseño
  - Mayor facilidad de mantenimiento
  - Depuración de errores más sencilla
- Reutilización de código: los métodos tienen un nombre asociado para llamarlos (ejecutarlos, invocarlos)... y se pueden ejecutar tantas veces como queramos (también se pueden agrupar en librerías y distribuir)
  - Un buen programador es perezoso: intentará hacer cada trabajo una sola vez.
- Nuevas posibilidades: recursión... qué impide que un método se invoque a sí mismo? Nada.

Dar nombre a los bloques de código

La idea esencial de un método es sencilla:

- dar nombre a un bloque de código
- para después poder ejecutar ese código llamando (o invocando) a ese nombre

$\rightsquigarrow$ agrupar una secuencia de instrucciones y abstraerlas bajo un nombre

Otra idea esencial: la parametrización

Ejemplo

Queremos un método para limpiar la consola (dejarla en blanco, sin texto) $\rightsquigarrow$ asumiendo que tiene 25 líneas, escribir 25 saltos de línea con `Console.WriteLine ()`

Escribimos un bloque de código para hacer esa tarea

```
{
  int i;
  for (i=0; i<25; i++)
    Console.WriteLine();
}
```

...y le damos nombre (`limpiaPantalla`):

```
static void limpiaPantalla()
{
  int i;
  for (i=0; i<25; i++)
    Console.WriteLine();
}
```

Este código se pone dentro de la clase, al mismo nivel que el método `Main`. NO dentro del método `Main` (no se pueden anidar métodos).

Ejemplo (II)

Ahora, cada vez que queramos limpiar consola, llamamos:

```
limpiaPantalla();
```

En cierto modo, hemos construido una nueva instrucción `limpiaPantalla ()` que podemos utilizar cuando queramos.

Qué ocurre cuando se invoca a un método?

```
<< instrucción1 >>
<< llamada a método >>
<< instrucción2 >>
```

- Tras ejecutar `instrucción1` el flujo de ejecución salta al bloque de código correspondiente a `método`
- Se ejecutan las instrucciones de ese bloque de código
- Al teminar el código del método, el flujo vuelve al punto de llamada y continúa con `«instrucción2»`

Parámetros

En el ejemplo anterior el método `limpiaPantalla ()` escribe 25 saltos de línea... pero el número de líneas de la consola puede variar de uno a otro sistema operativo o tipo de consola.

$\rightsquigarrow$ sería más versátil implementar un método que admitiese un parámetro indicando el número de saltos de línea a escribir

. . . algo así como `saltosLinea (n)` donde `n` es un entero que indica el número de saltos de línea. Implementamos:

```
static void saltosLinea(int n) // escribe n saltos de línea
{
  int i;
  for (i=0; i<n; i++)
    Console.WriteLine();
}
```

Nota: el parámetro `n` del método se comporta como una variable de tipo `int` dentro del método.

Para escribir 25 saltos de línea, llamamos:

```
saltosLinea(25);
```

Devolución de valores

Los métodos pueden devolver valores (a quien los llame).
Ejemplo: pedir la altura de un triángulo entre unos valores `min` y `max` dados como parámetro y devolver dicha altura:

```
static double alturaTri(double min, double max){
  double alturaT = min - 1 ; // valor "por defecto" no válido
                             // para forzar  entrada en bucle

  while (alturaT < min || alturaT > max) { // mientras no válido
    Console.Write ("Altura del triángulo [{0},{1}]: ", min, max);
    alturaT = double.Parse (Console.ReadLine ());
  } // fin  while

  return alturaT;
}
```

- En vez de `void` ahora `double` como valor de retorno
- El valor se devuelve con `return ...`

Ahora podemos llamar a este método, con los parámetros adecuados y recoger su valor de devolución:

```
double alt
alt = alturaTri (10, 200);
```

Al ejecutar:

```
Altura triángulo [10,200]: 5
Altura triángulo [10,200]: 567
Altura triángulo [10,200]: 47
```

En la variable `alt` tendremos el valor `47`.

Generalizando: más parámetros

El código para solicitar la altura y la base de triángulo es muy parecido. Lo que cambia es:

- El texto para pedir el dato ("Base del triángulo" o "Altura del triángulo")
- Posiblemente los valores máximos y mínimos admitidos. . . ya están parametrizados en la versión anterior.

Parametrizamos el *texto* de solicitud de dato y tenemos:

```
static double pideDato(string texto, double min, double max){
  double dato = min - 1; // valor "por defecto" no válido
                         // para forzar entrada en bucle

  while (dato < min || dato > max) { // mientras no valido
    Console.Write ("{0} [{1},{2}]: ", texto, min, max);
    dato = double.Parse (Console.ReadLine ());
  } // fin while

  return dato;
}
```


Ejecutamos

```
double alt = pideDato ("Altura triángulo", 10, 200);
double bas = pideDato ("Base triángulo", 5, 300);

Console.WriteLine ("alt: " + alt + "    base: " + bas);
```

```
Altura triángulo [10,200]: 5
Altura triángulo [10,200]: 203
Altura triángulo [10,200]: 45
Base triángulo [5,300]: 0
Base triángulo [5,300]: 8
alt: 45    base: 8
```

Con el mismo código podemos hacer múltiples peticiones de datos!! (este método podríamos incluso utilizarlo para otros programas que hemos hecho).

Sintaxis

La sintaxis (todavía incompleta) de la declaración o prototipo de un método es:

static *tipoResult nombreMétodo* ($tipo_1$ $parám_1$,. . . , $tipo_n$ $parám_n$)

- **static** indica que es un método de la clase (no nos preocupa por ahora)
- *tipoResult* es el tipo de dato que devuelve el método: `void` si no devuelve nada; `int`, `double`, `bool`. . . , cualquiera de lo que conocemos
- *nombreMétodo* es un identificador cualquiera (con las mismas reglas que vimos de formación de identificadores)
- $tipo_1$,. . . ,$tipo_n$ son los tipos de los parámetros (o argumentos) del método. . . cualquier de los que conocemos
- $parám_1$,. . . ,$parám_n$ son los identificadores de los parámetros que recibe el método.

Sintaxis (II)

Después del *prototipo* del método viene un bloque de código (entre "{" y "}") que llamamos cuerpo del método:

- es un bloque de código estándar
- puede utilizar los parámetros del método, declarados en el prototipo
- y también las variables locales declaradas en este bloque
  - No puede utilizar variables de otro bloque: no puede utilizar las variables declaradas en el método `Main` u otros.
    No están en su ámbito de visibilidad ⇝ abstracción
- si el método devuelve un tipo relevante (distinto de `void`) debe contener una instrucción

  `return expresion ;`

  con una `expresion` correcta de tipo *tipoResult*. Si el tipo es `void` puede contener `return;` pero no es necesario (ni habitual).
  Nota: `return` provoca la terminación inmediata del método y la vuelta al punto de llamada al mismo (rompe flujo).

En la llamada...

Si tenemos un prototipo:

static *tipoResult nombreMétodo* ($tipo_1$ $parám_1$,..., $tipo_n$ $parám_n$)

podemos llamar al método de la forma:

$$nombreMétodo\ (val_1,\ldots,\ val_n\ )$$

donde val_1,...,val_n tienen que ser valores de tipo $tipo_1$,...,$tipo_n$.

El valor resultante de la llamada:

- se puede recoger en una variable de tipo *tipoResult* mediante una asignación `var = nombreMétodo(...`
- o se puede utilizar como valor dentro de otra expresión

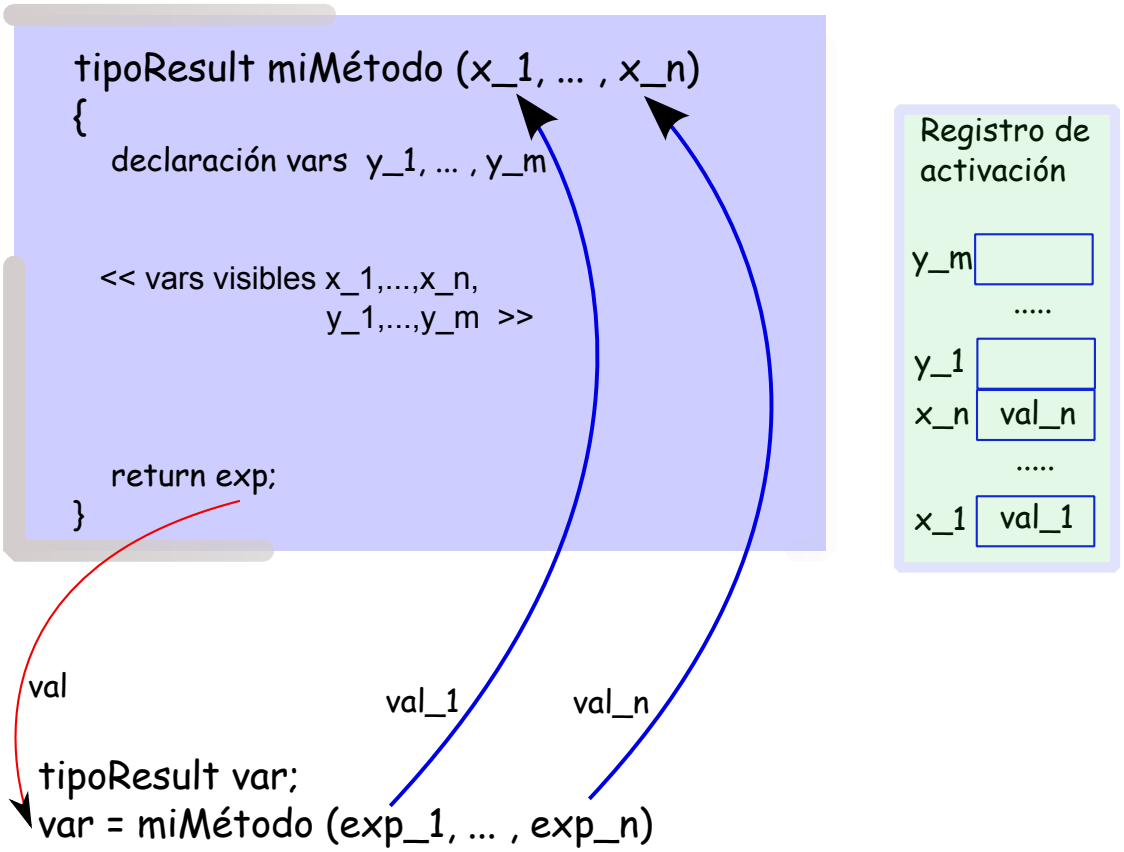
Los parámetros con los que se llama al método tienen que coincidir en número y tipo con los declarados en el prototipo del método.

Parámetros y valores de devolución

- Los métodos permiten definir bloques de código independientes, que pueden pensarse e implementarse de manera aislada.
- Los parámetros y el valor de retorno permiten compartir información con el exterior (con el método llamante).
  - los parámetros permiten que el método reciba información (datos) del exterior
  - el valor de devolución permite devolver información al exterior

  (Como veremos más adelante, los parámetros también permitirán devolver información al exterior).

La abstracción procedimental es uno de los principales mecanismos en programación, presente en todos los lenguajes comunes.



Estructura de un programa

Ahora, nuestros programas pueden contener tantos métodos como queramos. La estructura de un programa será de la forma:

```
using System;

namespace Name
{
  class MainClass
  {
    <<metodo_1>>
    ...
    <<metodo_n>>

    public static void Main ()
    {
       ...
    }
}
}
```

Cada método puede ser llamado desde cualquier otro método (no olvidemos que `Main` es el método de *entrada*, el primero que será llamado al arrancar el programa).

Estructurando programas

Revisitamos: dibujar un árbol de navidad con símbolos "*" tal como se ha visto
Método para pedir altura:

```
public static int pideAltura (int min, int max){
  int alt;

  Console.Write ("Altura del arbol: ");
  alt = int.Parse (Console.ReadLine ());

  while (alt<min || alt>max) {
    Console.Write ("Altura inválida. Debe estar " +
                   "entre {0} y {1}\n", min, max);
    Console.Write ("Altura del arbol: ");
    alt = int.Parse (Console.ReadLine ());
  }
  return alt;
}
```


Método para dibujar "n" blancos

```
public static void dibujaBlancos (int n){
  int i;
  for (i=0; i<n; i++)
    Console.Write (" ");
  }
```

Método para dibujar "n" asteriscos

```
public static void dibujaAsteriscos (int n){
  int i;
  for (i=0; i<n; i++)
    Console.Write ("*");
  }
```

El código es casi idéntico!! Se puede parametrizar la cadena a escribir?...

Generalizando con parámetros

Pasamos el texto a escribir como parámetro:

```
public static void repiteTexto (int n, string s){
  int i;

  for (i=0; i<n; i++)
    Console.Write (s);
  }
```


Generalizando con parámetros

El programa principal:

```
public static void Main ()
{
  int alt, nivel, blancos, asteriscos;

  alt = pideAltura (0, 30);

  nivel = 1;
  while (nivel <= alt) {
    // dibujamos blancos
    blancos = alt - nivel;
    repiteTexto (blancos," ");

    // dibujamos asteriscos
     asteriscos = 2*nivel - 1;
    repiteTexto (asteriscos,"*");

    Console.WriteLine ();
     nivel++;
  }
}
```

Más métodos

El cálculo del numero de blancos y de asteriscos depende de la altura del árbol y del nivel que estamos dibujando. . . y se puede calcular también en métodos:

```
public static int numBlancos (int alt, int nivel){
  return alt - nivel;
}

public static int numAsteriscos (int alt, int nivel){
  return 2*nivel - 1;
}
```

(En realidad el número de asteriscos no depende de la altura. Se podría quitar ese parámetro)

Composición funcional

Un método (función) puede tomar como argumento cualquier expresión del tipo adecuado, incluida otra llamada a función.

El programa principal ahora puede ser:

```
public static void Main ()
{
  int alt, nivel;

  alt = pideAltura (0, 30);

  nivel = 1;
  while (nivel <= alt) {
    // dibujamos blancos
    repiteTexto (numBlancos(alt,nivel)," ");

    // dibujamos asteriscos
    repiteTexto (numAsteriscos(alt,nivel),"*");

    Console.WriteLine ();
    nivel++;
  }
}
```

Cúantos métodos?

A la vista del ejemplo anterior observamos que podemos implementar un mismo algoritmo utilizando más o menos métodos

. . . cuál es el nivel adecuado de subdivisión o particionado de un problema en subproblemas más simples?

- No hay una repuesta definitiva a esta pregunta.
- Cada programador tiene su estilo. . . pero hay mejores y peores estilos!
- Es un arte, que se aprende: practicando

Algunas ideas básicas de buen estilo:

- Hacer métodos puros de entrada/salida:
  - Los métodos que hacen interacción (entrada/salida) con el usuario deben hacer solo eso. No incluir otro tipo de cómputo en los mismos.
- En general, la tarea de un método debe ser fácil de describir aisladamente

Qué ocurre en memoria?

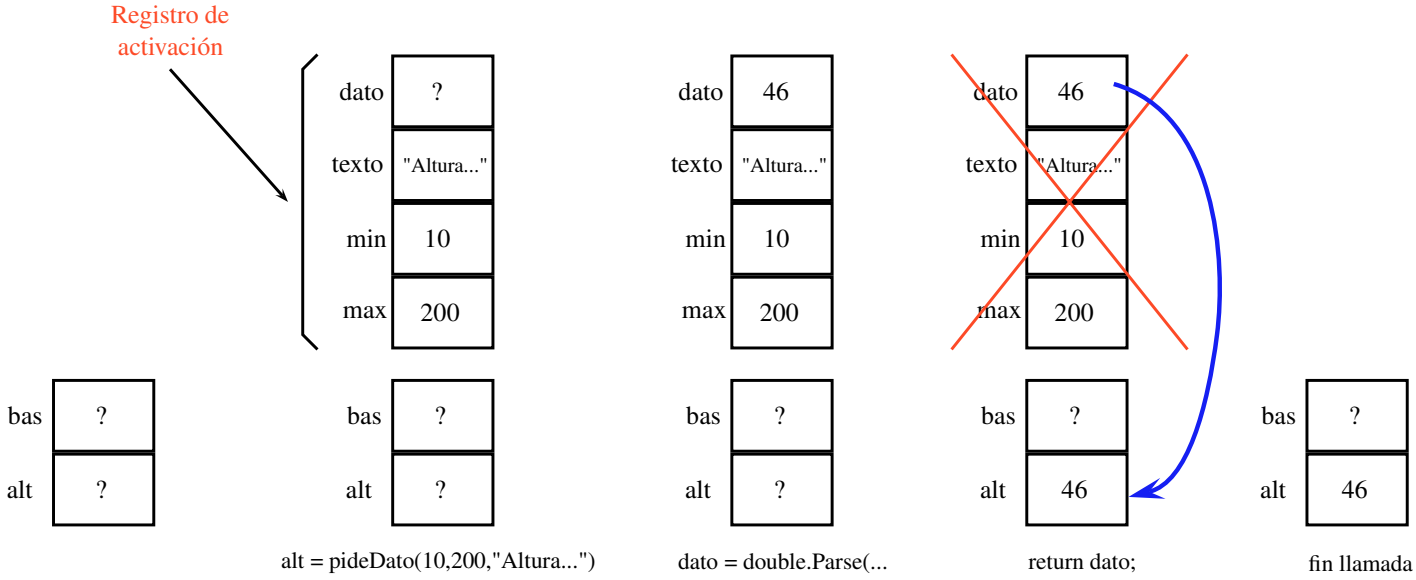
Volvamos al método `pideDato`

```
static double pideDato(string texto, double min, double max){
  double dato = min - 1; // valor "por defecto" no válido
                         // para forzar entrada en bucle

  while (dato < min || dato > max) { // mientras no valido
    Console.Write ("{0} [{1},{2}]: ", texto, min, max);
    dato = double.Parse (Console.ReadLine ());
  } // fin while

  return dato;
}
```

Y veamos la primera llamada:

```
double alt = pideDato ("Altura triángulo", 10, 200);
...
```

Qué ocurre en memoria?



- Se crea un registro de activación: zona de memoria con
  - los parámetros (valores) recibidos por el método (`texto, min, max`)
  - las variables locales del método (`dato`)
- Cuando acaba el método se destruye el registro de activación
  - pero el valor de retorno de `dato` (46) se devuelve al método llamante (y queda asignado a la variable `alt`)

Máximo de 4 enteros leídos de teclado

Método para leer entero:

```
static int leeNum(string s) {
  Console.Write(s);
  return int.Parse (Console.ReadLine ());
}
```

Método para calcular el máximo de 2 enteros:

```
static int max(int i, int j){
  int m;
  if (i >= j)  m = i;
  else m = j;
  return m;
}
```

Otra forma:

```
static int max(int i, int j){
  if (i >= j)
    return i;
  else
    return j;
}
```

Paso de parámetros

Método para calcular el máximo de 4 enteros:

```
static int max4(int i, int j, int k, int l){
  return max(max(max(i,j),k),l);
}
```

Composición funcional! (anidamiento de llamadadas)

Método `Main`

```
public static void Main ()
{
  int a, b, c, d;

  a = leeNum ("Primer num: ");
  b = leeNum ("Segundo num: ");
  c = leeNum ("Tercer num: ");
  d = leeNum ("Cuarto num: ");

  Console.WriteLine ("El maximo es: {0}", max4(a,b,c,d));
}
```


Paso de parámetros

Hemos visto el paso de parámetros por valor (o por copia):

- el método recibe el valor del parámetro, más precisamente, una copia en memoria de ese valor.

Qué significa esto?

```
static void sumaUno(int i){
  i = i+1;
  Console.WriteLine ("Valor: " + i);
}
```

Y ahora en el método `Main` hacemos:

```
int val = 3;
sumaUno (val);
Console.WriteLine ("Valor: " + val);
```

Qué vemos en consola?

```
Valor: 4
Valor: 3
```

Explicación: el modelo de memoria de la diapositiva anterior!

- ¿Cómo hacemos que un método reciba un dato y lo modifique?

Por ejemplo, un método `sumaUno(i)` que incremente en 1 el valor de `i`, i.e., que cambie el valor de la variable `i` sumándole 1.

- ¿Cómo hacemos que un método devuelva más de un valor?

Por ejemplo, un método `pideDatosTriangulo` que recoja la base y la altura de un triángulo, y devuelva ambos valores... el `return` de un método solo puede devolver un valor.

El paso de parámetros por valor es *muy seguro* porque el método no puede modificar las variables externas (las que pueda enviar el método llamante)... pero tiene limitaciones citadas...

Nota: el problema de devolver varios valores podría resolverse *indirectamente* devolviendo un *valor que agrupe valores* (una tupla)

Pero la solución directa a estos problemas es el

paso de parámetros por referencia

Paso de parámetros por referencia

Idea: en vez de pasar el valor de una variable, se pasa una referencia a esa variable, la dirección de esa variable.

- Dentro del método se utiliza a la propia variable, en vez de *solo* su valor:
  - se tiene acceso a su valor
  - pero también se puede modificar ese valor!
- Todo lo que le pasa a la variable dentro del método le pasa fuera de este (queda reflejado cuando acabe el método).
- En realidad, se pasa la dirección de memoria de la variable.

Esto se consigue añadiendo la palabra reservada `ref` al paso de parámetros, tanto en la declaración como en la llamada.

Paso de parámetros por referencia

En el ejemplo anterior:

```
static void sumaUno (ref int i){
  i = i+1;
}
```

Y ahora en el método `Main` hacemos:

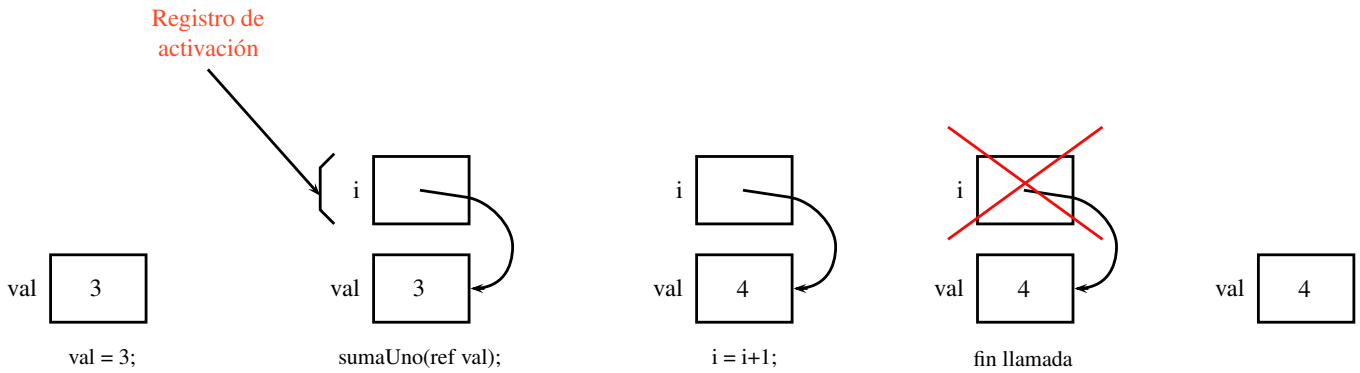
```
int val = 3;
sumaUno (ref val);
Console.WriteLine ("Valor: " + val);
```

Qué vemos en consola?

```
Valor: 4
```

Se ha cambiado el valor de la variable dentro del método!

Qué ocurre en memoria?



- Se crea un registro de activación que contiene:
  - los parámetros recibidos por el método: referencia a `val`, con el nombre `i`
  - (en este caso no hay variables locales)
- Cuando acaba el método se destruye el registro de activación
  - pero como efecto lateral se ha modificado el valor de `val`

Paso de parámetros: por valor y por referencia

Hay una diferencia muy importante dependiendo de cómo se pasa un parámetro (aparte de la palabra `ref`):

- por valor: en la llamada puede pasarse cualquier expresión del tipo correspondiente.

```
// metodo saltosLinea(int n) : escribe "n" saltos de línea
int i = 5, j = 7;
saltosLinea((6*i - j) % 2)
```

- por referencia: en la llamada tiene que pasarse una variable del tipo correspondiente (para recoger el valor de retorno).

```
int val = 6;
sumaUno (ref val);
```


Paso de parámetros: variantes

En C# el paso de parámetros esencialmente puede ser de los dos modos vistos:

- por valor
- por referencia (con la palabra reservada `ref`)

. . . hay una variante del paso por referencia: el modificador `out` para parámetros de salida.

Diferencias?

	parámetro **ref**	parámetro `out`
Inicialización	Valor asignado antes del método	Valor asignado en el método
Modificación	El método puede o no modificarlo	El método tiene que asignarle valor

Parámetros por referencia con modificador `out`

Método para pedir base y altura de un triángulo (utilizando el método `pideDato` visto):

```
public static void pideDatosTriangulo(
        double minBase,   double maxBase,    out double baseT,
        double minAltura, double maxAltura, out double alturaT){

  baseT   = pideDato ("Base del triángulo",    minBase,    maxBase);
  alturaT = pideDato ("Altura del triángulo", minAltura, maxAltura);
}

//llamada
double bas, alt;

pideDatosTriangulo (10, 200, out bas, 30, 800, out alt);
```


Paso de parámetros: resumen

Cómo pasamos un parámetro a un método:

• entrada: paso por valor (sin modificador) ⇝ el método recibe un valor y lo usa pero no lo modifica (si se pasa una variable, su valor no se ve alterado).

- llamada: con cualquier expresión del tipo adecuado
- el método puede modificar localmente un parámetro pasado por valor, pero esa modificación no tiene efecto fuera del método

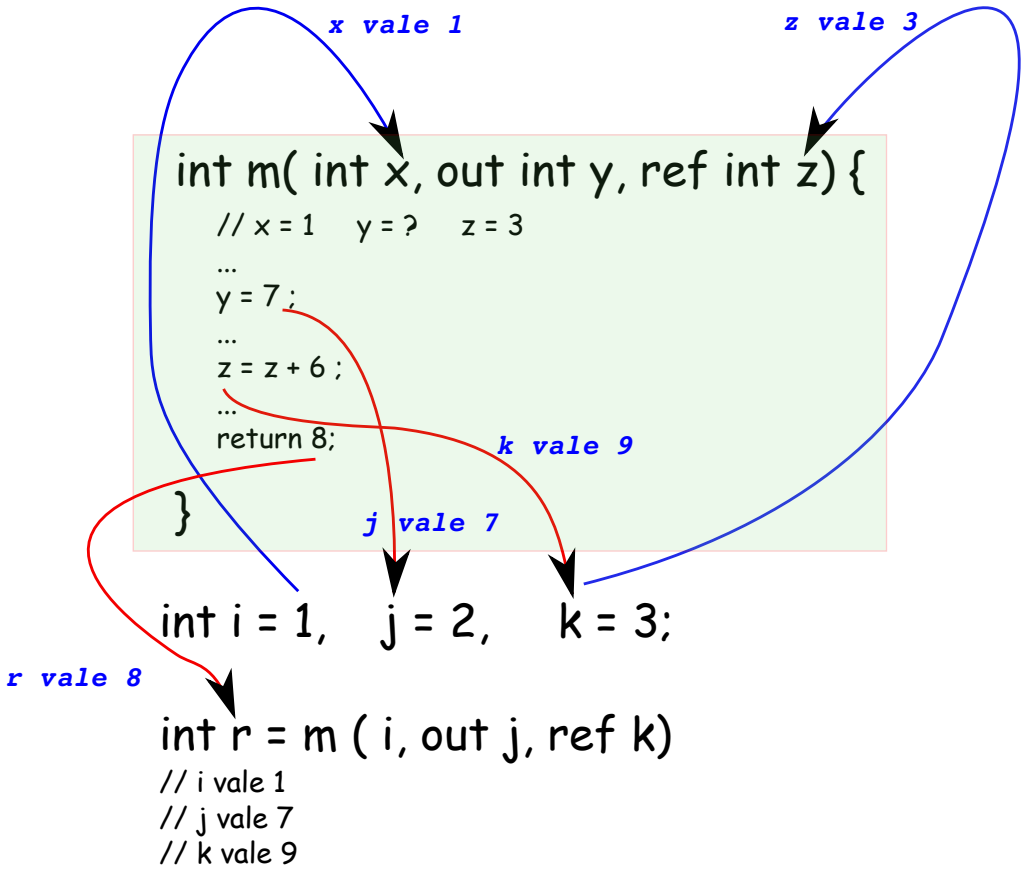
• salida: paso por referencia con modificador `out` ⇝ el método recibe una variable sin valor previo (si lo tiene es irrelevante) en la que devolverá un valor.

- llamada: tiene que ser con una variable
- el parámetro correspondiente a esa variable tiene que asignarse dentro del método

• entrada-salida: paso por referencia con modificador `ref` ⇝ el método recibe una variable con valor previo que puede utilizarse dentro del método y que se utilizará para devolver otro valor.

- llamada: tiene que ser con una variable
- esa variable tiene que asignarse antes del método

Paso de parámetros: "in", `out`, `ref`



Diseño de métodos

A la hora de diseñar un programa conviene hacer un análisis cuidadoso, descomponerlo en subproblemas y decidir los métodos a implementar. A la hora de diseñar un método:

- Debemos hacer una descripción clara y concisa de su cometido (especificación)

  Qué recibe de entrada? Qué hace? Qué devuelve?

- Siguiente paso . . . si el anterior no está *absolutamente claro* volver atrás.
- Determinar los parámetros concretos que utiliza el método y el valor de retorno:
  - Determinar si son de entrada, salida o entrada salida
- Comentar el método (antes de implementarlo!)
- Siguiente paso . . . si el anterior no está *completo* volver atrás.
- Abstraerse del resto del programa, declarar las variables locales necesarias, escribir el código del método, asegurarse de que se devuelve el valor correspondiente,. . .

Correcto o incorrecto?

```
public static void m(ref int x){
  x = 4;
}
// llamada
int i = 3;
m (ref i);
```

Correcto. Valor de `i`? 4

```
public static void m(ref int x){
}
// llamada
int i = 3;
m (ref i);
```

Correcto. Valor de `i`? 3


```
public static void m(ref int i){
  i = 4;
}
// llamada
int i;
m (ref i);
```

Incorrecto! `i` tiene que estar asignada antes de llamar a `m`

```
public static void m(ref int x){
  x = x+1;
}
// llamada
int i = 3;
m (ref i);
```

Correcto. Valor de `i`? 4

Aquí se utiliza el valor de `i` y se modifica.

```
public static void m(out int i){
  i = i+1;
}
// llamada
int i = 3;
m (out i);
```

Incorrecto! `i` tiene que asignarse en el método antes de utilizar su valor en el lado derecho de la asignación `i = i+1`

Este código es equivalente a:

```
public static void m(out int x){
  x = x+1;
}
// llamada
int i = 3;
m (out i);
```

Aquí se ve más claro que `x` no tiene valor asignado antes de su uso en la asignación `x = x+1;`


```
public static void m(int x){
  x = 4;
}
// llamada
int i = 3;
m (i);
```

Correcto! Valor de `i`? 3

```
public static void m(int x){
  x = 4;
}
// llamada
int i = 3;
m (ref i);
```

Incorrecto. La llamada no coincide con la declaración del método.

```
public static void m(out int x){
  int i = 3;
}
// llamada
int i = 3;
m (out i);
```

Incorrecto! el método tiene que asignar valor a `x`

```
public static void m(ref int i, out int j){
  j = 6;
  j = i + j;
}
// llamada
int i = 1, j = 2;
m (ref i, out j);
```

Correcto. Valor de `i j` tras el método? `1 7`

Variables y parámetros formales

```
public static void resta(int x, int y, out int z){
  z = x-y;
}


public static void Main ()
{
  int x = 3, y= 4, z = 5;

  resta (z, x, out y);
  Console.WriteLine ("x: {0}   y: {1}   z: {2}", x, y, z);
}
```

Es correcto? Qué escribe?

`x: 3  y: 2  z: 5`

Ejemplo

Diseñar un método para calcular el cociente y el resto de la división entera:

```
public static void cocienteYresto(int i, int j, out int c, out int r){
  c = i / j;
  r = i % j;
}
```

Diseñar un método que intercambie el valor de dos variables de tipo `int`:

```
public static void swap(out int i, out int j){
  int aux = i ; // var auxiliar para intercambio
  i = j ;
  j = aux ;
}
```

Por qué está mal? son parámetros de entrada-salida
$\rightsquigarrow$ `ref` en vez de `out`!

Valor de retorno o variable de salida?

Cuando un método tiene que devolver un solo resultado tenemos dos opciones:

- devolverlo como valor de retorno del método
- devolverlo como variable de salida (`out`)

¿Qué es mejor? ¿Alguna diferencia importante?

Desde el punto de vista de la eficiencia, es esencialmente igual... hay una ventaja en devolverlo como valor de retorno del método:

- la llamada al método devolverá un valor (del tipo correspondiente) que puede utilizarse directamente en una expresión $\rightsquigarrow$ anidamiento de llamadas.

Comparemos...

Máximo de 4 números revisitado

```
// como valor de retorno del método
static int max(int i, int j){
  if (i >= j) return i;
  else return j; }

static int max4(int i, int j, int k, int l){
  return max(max(max(i,j),k),l); } // anidamiento de funciones

// llamada
int m = max4(a,b,c,d));
```

```
// Como variable de salida
static void maxOut(int i, int j, out int m){
  if (i >= j) m = i;
  else m = j; }

static void max4Out(int i, int j, int k, int l, out int m){
  maxOut(i,j,out m); // no podemos anidar
  maxOut(m,k,out m);
  maxOut (m, l, out m); }

// llamada
int m;
max4Out (a, b, c, d, out m);
```


Cociente y resto revisitado

Calcular el cociente y el resto de la división entera sin utilizar la división y el módulo de C#:

```
static void cocienteYresto(int a, int b, out int c, out int r){
  c = 1;
  while (a >= b) {
    a -= b;
    c++;
  }
  r = a;
}
```

Es correcto? Trazar (depurar) la ejecución de `cocienteYresto(7,3,out c,out r)`

Incorrecto! `c` debe inicializarse a 0.

Algoritmo de Euclides para el MCD

Calcular el MCD de dos números utilizando el algoritmo de Euclides:

```
static int mcd(int n, int m){
  int r = 1; //
  while (m != 0) { // (n,m) = (m,n%m) con variable auxiliar
    r = n % m;
    n = m;
    m = r;
  }
  return n;
}
```

Es correcto? Que ocurre si $n < m$? Por ejemplo, en la llamada `mcd(6,15)`. Trazar la ejecución.

Es correcto!

Ámbito de las variables

- Las variables declaradas tienen un ámbito de existencia en el que son visibles (pueden ser utilizadas). Fuera de él *no existen*.
- El ámbito de una variable corresponde al bloque en el que está declarada.
  - Las variables tienen que ser declaradas antes de ser utilizadas.

Ya hemos visto bloques: el bloque principal del método `Main`, bloques en el *if-else*, bloque del cuerpo del *while*, bloques de métodos declarados,...$\rightsquigarrow$ bloques de código entre `"{"` y `"}"`

- Ya hemos visto que en C# se pueden crear bloques anidados y cada uno de ellos puede tener sus propias variables declaradas.

```
{
  int i;
  // i es visible
  {
    int j;
    // i, j son visibles
  }
  // solo i es visible, j = 3 es incorrecto
}
```

Otros lenguajes permiten *redefinir* una variable (crear dos variables con el mismo nombre), ocultando la del bloque externo. C# no permite la redefinición:

```
// no compila en C#
{
  int i;
  i = 3;
  {
    int i;
    i = 8;
  }
  // cuanto valdría i?
}
```


Pero sí permite la reutilización de un nombre de variable en diferentes bloques:

```
{
  int i;
  i = 3;
  Console.WriteLine(i); // escribe 3
}
{
  int 8;
  i = 8;
  Console.WriteLine(i); // escribe 8
}
// cuanto valdría i en este punto?
```

Está permitido y es muy habitual declarar variables *contador* en los bucles `for`, en la propia inicialización del bucle:

```
for (int i=0 ; i<10 ; i++)
{
  int j = 3;
  Console.WriteLine("{0}x{1} = {2} ", i, j, i*j);
}
```

Si justo después de este bucle hacemos:

```
Console.WriteLine(i);
Console.WriteLine(j);
```

que valores escribirá?