Bucles

Repetir código

Recordemos el ejemplo del cálculo de nómina para empleados de la hoja de problemas. Supongamos que la empresa tiene 7 empleados y queremos calcular la nómina de todos ellos.
Una solución:

```
<<cómputo para el empleado 1>>
<<cómputo para el empleado 2>>
...
<<cómputo para el empleado 7>>
```

Y si tenemos 777 empleados?... Una solución mejor:

```
repetir 7 veces
   <<cómputo para un empleado>>
```


Repetir código

En el programa del cálculo del área de un triángulo podría ser útil hacer lo siguiente:

1. Solicitar altura
2. Comprobar que el dato introducido es válido
3. Si no es válido volver al punto 1.

Idem para solicitar la base.

Algo así como:

```
mientras <<dato no válido>>
   <<solicitar dato>>
```

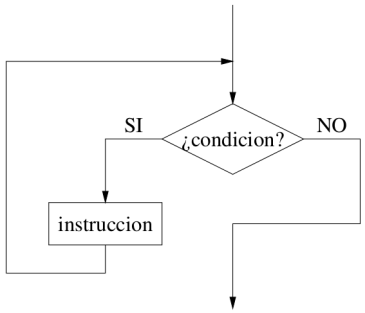

Bucle `while`

Sintaxis:

```
while (<<condición>>)
   <<código>>
```

Significado: mientras `«condición»` sea `true` ejecutar `«código»`

- `«código»` se llama cuerpo del bucle



Hola `while`: nuestro primer bucle

Un pequeño while...

```
while (true);
```

... un gran paso para el programador!

En este caso el código del `while` es vacío (instrucción vacía).

¿Qué hace este código?: cuelga el programa!!

Por primera vez en el curso podemos colgar un programa!! 

- Esto es realmente interesante para algo?
- No sería mejor "prohibir" este tipo de instrucciones? Eliminamos el `while` del repertorio de instrucciones?
- No podemos tener *otro tipo de bucle* igual de potente pero que no permita infinitas iteraciones?

Ejemplo del área de un triángulo revisitado

Solicitamos datos hasta que el usuario proporciona valores válidos. Utilizamos un booleano `pedirDato` para controlar el bucle: variable de control

```
double baseT = 0, alturaT = 0, // valor "por defecto" para evitar error
       areaT;                  // de "no-inicialización"

bool pedirDato = true; // inicialización de la variable de control
while (pedirDato) {    // bucle para pedir base
  Console.Write ("Base del triángulo: ");
  baseT = double.Parse (Console.ReadLine ());
  if (baseT >= 0 && baseT <= 10000) // si esta en rango, terminamos
    pedirDato = false; // parada del bucle
} // fin while

... // análogo para alturaT

areaT = baseT*alturaT/2;
Console.WriteLine ("El área del triángulo es: " + areaT);
```


Ejemplo del área de un triángulo revisitado

De este modo podemos ejecutar:

```
Base del triángulo: -3
Base del triángulo: 4325456
Base del triángulo: 8
Altura del triángulo: -2
Altura del triángulo: 575654
Altura del triángulo: 10
El área del triángulo es: 40
```


Ejemplo del área de un triángulo revisitado (II)

Sin booleano: damos un valor inicial no válido a `baseT` y a `alturaT`, y cambiamos la condición del bucle (utilizamos las propias `baseT` y `alturaT` como variables de control):

```
double baseT = -1, alturaT = -1, // valor "por defecto" no válido
       areaT;                    // para forzar entrada en bucle

// con condición original negada
while (!(baseT >= 0 && baseT <= 10000)) {
  Console.Write ("Base del triángulo: ");
  baseT = double.Parse (Console.ReadLine ());
} // fin while

// otra forma, utilizando equivalencias lógicas
// ojo con el manejo de opearadores lógicos!!!
while (alturaT < 0 || alturaT > 10000) {
  Console.Write ("Altura del triángulo: ");
  alturaT = double.Parse(Console.ReadLine());
} // fin while

areaT = baseT*alturaT/2;
Console.WriteLine ("El área del triángulo es: " + areaT);
```

Cálculo de la nómina de "n" empleados

```
Console.Write ("Número de empleados: ");
int n = int.Parse(Console.ReadLine ());

int cont = 0; // variable de control
while (cont < n) { // se repite para cont = 0,1...(n-1) -> "n" veces
  <<computo de la nómina de un empleado>>
  cont = cont + 1;
} // end while
```

Otra forma, sin la variable `cont`:

```
Console.Write ("Número de empleados: ");
int n = int.Parse(Console.ReadLine ());

// utilizamos la propia "n" como var de control
while (n > 0) { // se repite para los valores n,n-1,...,1 -> "n" veces
  <<computo de la nómina de un empleado>>
  n = n-1;
} // end while
```


Diferencia entre *if* y *while*

```
if (<<condición>>)
   <<código>>
```

```
while (<<condición>>)
   <<código>>
```

En el primer caso «código» se ejecuta como mucho una vez (si la condición es cierta), mientras que en el segundo caso puede ejecutarse más de una vez... El código del `while` puede ejecutarse:

- 0 veces: si ya inicialmente la condición es `False`
- 1 vez: si es cierta, pero tras la primera vuelta se hace `False`
- 2 veces: ...
- ...
- ¿cuántas veces? ⇝ infinitas!!! programa que no termina!

Variables de control

Son variables (una o más) que controlan la parada del bucle. Tienen que ser:

- Inicializadas antes del bucle
- Comprobadas en cada repetición o iteración del bucle para determinar si se entra o no en el bucle (es lo que se hace en «condición»)
- Actualizadas dentro del bucle, en cada iteración

Idea: La variable (o variables) comienza con un valor inicial y en cada iteración se aproxima a un *valor final* o *valor de parada* que se controla en la condición.

También puede hacerse que la condición de parada dependa de la interacción con el usuario, como en el ejemplo del área del triángulo.

Escribiendo `n` asteriscos en pantalla

```
int n, cont;
Console.Write ("Número de asteriscos: ");
n = int.Parse (Console.ReadLine ());

cont = 0;
while (cont < n) {
  Console.Write ("*"); }
```

Incorrecto! Falta incremento de la variable de control.

```
cont = 0;
while (cont < n) {
  Console.Write ("*");
  n = n+1 ; }
```

Incorrecto! (es muy fácil colgar un programa!)

```
cont = 0;
while (cont <= n) {
  Console.Write ("*");
  cont = cont+1 ; }
```

Todavía incorrecto... hace una iteración de más.

Trazas de ejecución

Qué muestra este código en pantalla?

```
int x = 2, cont = 0;
while (cont < 3) {
  x = x * x;
  Console.WriteLine (x);
  cont = cont + 1;
}
```

Traza de ejecución (simulación manual):

	x	cont	salida
iteración 0	2	0	
iteración 1	4	1	"4"
iteración 2	16	2	"16"
iteración 3	256	3	"256"

Y si reemplazamos `cont = cont + 1;` por `cont = cont + 2;`?
Y si eliminamos la asignación `cont = cont + 1;`?
Hacer las trazas!

Sumatorio

Problema: dado $n \geq 0$ calcular $\sum_{i=1}^{n} i = 1 + 2 + \ldots + n$

```
int sum = 0,
    i = 1,
    n ;

Console.Write ("Valor de n? ");
n = int.Parse(Console.ReadLine());

while ( i<n ) {
  i = i + 1;
  sum = sum + i;
}
```

Incorrecto!!!... por qué? cómo se puede arreglar?

```
while ( i<=n ) { // recorre i=1..n
  sum = sum + i;
  i = i + 1;
}
```

Cuidado! La especificación dice "dado $n \geq 0$"... que pasa con $n = 0$??
... es correcto. Por qué?

Un inciso: operadores de incremento/decremento

- En los ejemplos anteriores hemos visto que es muy frecuente actualizar el valor de las variables sumando o restando 1... u otra cantidad.

```
...
i = i + 1;
sum = sum + i;
...
```

- C# admite una forma abreviada para este tipo de instrucciones. Podemos hacer:

```
...
i++;  // abrevia i = i+1;
sum += i; // abrevia sum = sum + i;
...
```

De manera análoga tenemos los operadores `--`, `-=`, `*=`, `/=` ...

Operadores abreviados

Algunas abreviaturas:

Operador abreviado	Significado
`a++`	`a = a + 1`
`a--`	`a = a - 1`
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`
`a%= b`	`a = a% b`
(hay más)	...

Sentencias y valores

- En C# todas las sentencias devuelven un valor.
- Casi siempre se ignora ese valor, pero puede utilizarse:

```
int i, j;
i = (j=0);
```

La asignación `j=0`:

- asigna a `j` el valor 0
- pero además, devuelve el propio valor asignado, i.e., el `0`

Y ese valor `0`, a su vez, se asigna a la variable `i`
En resumen: `i` y `j` se inicializan simultáneamente a 0.

- Los paréntesis son opcionales (el operador = asocia por la derecha). Se puede hacer:

```
int i, j, k, l;
i = j = k = l = 0; // todas inicializadas a 0
                   // equivalente a:  i = (j = (k = (l = 0)));
```

Nota: no oscurecer el programa abusando de estas características!!

Pre-incremento y post-incremento

- El operador ++ puede utilizarse de dos formas:
  - `i++` post-incremento: primero se devuelve el valor de `i` y luego se hace el incremento
  - `++i` pre-incremento: primero se hace el incremento y luego se devuelve el valor

Tiene alguna relevancia? Qué escribe este programa?

```
int i, j, k, l;
i = j = 1;
k = i++;
l = ++j;
Console.WriteLine (i + " " + j + " " + k + " " + l);
```

⇝ `2 2 1 2`

Nota: no oscurecer el programa con estos operadores!!
En la práctica, lo más usual es el post-incremento/decremento de forma elemental:

```
i++ ;
j-- ;
```


Volvemos a los bucles: tabla de multiplicar de un número

```
int n, i;

Console.Write ("Número: ");
n = int.Parse (Console.ReadLine ());

i = 0; // inicializamos contador
while (i<=10) { // recorremos i=0,...,10
  Console.WriteLine (n + " x " + i + " = " + (n * i));
  i++; // actualizamos contador
}
```

```
Número: 6
6 x 0 = 0
6 x 1 = 6
6 x 2 = 12
6 x 3 = 18
...
6 x 10 = 60
```


Divisores de un número dado

```
int n, i;

Console.Write ("Número: ");
n = int.Parse (Console.ReadLine ());

Console.Write ("Divisores: ");

i = 1;
while (i<=n) { // recorrido i = 1..n
  if (n % i == 0)
      Console.Write (i + " ");
  i++;
}
```

```
Número: 345
Divisores: 1 3 5 15 23 69 115 345
```

Otras condiciones de parada en bucles...

Los bucles anteriores están controlados por *contadores* que llevan cuenta del número de repeticiones y terminan tras un número determinado de ellas.
Pero hay otras formas de controlar la terminación:

- condición dependiente de la entrada del usuario
- condición dependiente de la lectura de un dispositivo externo
- ...
- en general, se pueden utilizar condiciones tan complejas como se desee.

Ejemplo: una ciudad tiene 9870 habitantes y su población crece un 10 % cada año. Determinar el número de años que tardará en superar los 30.000 habitantes.

Análisis... la población es un número entero. Qué significa exactamente que se incrementa en un 10 %?
$\rightsquigarrow$ ajustamos el 10 % al entero más próximo... es esto tan importante?

Crecimiento de población

```
const double RATE = 0.1;
int habs = 9870, cont = 0;

while (habs<=30000) {
  habs = habs + (int)(Math.Round (habs * RATE));
  Console.WriteLine (habs);
  cont++;
}

Console.WriteLine ("Población al cabo de " + cont + " años: " +
                   habs + " habitantes");
```

Cuántas iteraciones hace este código?
Seguro que termina? Por qué?
Y si la población se incrementase en un 0.00001 %?
$\rightsquigarrow$ el programa no terminaría... Es importante tener esto en cuenta y documentar bien el programa.

Un pequeño motor de física: caída libre

Modelar el efecto de la gravedad sobre un cuerpo en caída libre, mediante una tabla de tiempo/altura.

Altura en función del tiempo $alt(t)$

$$alt(t) = altIni - \frac{1}{2} * g * t^2$$

($altIni$ altura de partida; g cte. gravitatoria)

```
const double G = 9.80665; // Cte. gravitatoria universal
double altIni, // altura inicial
       alt,    // altura en funcion del tiempo
       t,      // tiempo trancurrido
       delta;  // intervalo (resolución) de tiempo

Console.Write ("Altura inicial (metros): ");
altIni = double.Parse (Console.ReadLine ());

Console.Write ("Intervalos de tiempo (segundos): ");
delta = double.Parse (Console.ReadLine ());
```


Caída libre

Construcción de la tabla

```
// Cabecera de la tabla formateada
Console.WriteLine ("{0,10} {1,-10}","Tiempo","Altura");
//                 {arg,hueco},
//                 positivo: justificación dcha; neg: just izqda

t = 0; // tiempo trancurrido
alt = altIni; // altura en cada instante de tiempo
while (alt>0) { // mientras no llege al suelo
  // fila de la tabla formateada
  Console.WriteLine ("{0,10} {1,-10}", t, alt);
  t += delta;  // abrevia t = t + delta;  incremento de tiempo
  alt = altIni - 0.5 * G * t * t;  // altura en el instante "t"
} // ojo: no saca el último par (t,alt) calculado porque puede ser alt<0

// pero sacamos con precisión el tiempo total de caída
t = Math.Sqrt (2 * altIni / G); // tiempo total de caída
Console.WriteLine ("\nTiempo total de caída: " + t);
```

Ejecutamos

```
Altura inicial (metros): 300
Intervalos de tiempo (segundos): 1
    Tiempo Altura
         0 300
         1 295,096675
         2 280,3867
         3 255,870075
         4 221,5468
         5 177,416875
         6 123,4803
         7 59,737075

Tiempo total de caída: 7,82195453698599
```


Bucles estándar

Es muy habitual repetir un número dado `n` de veces un `«código»`. Esto puede hacerse de la forma:

```
// n, i enteros

i = 0;
while (i<n) { // i recorre 0,1,...,n-1 (n vueltas)
  <<código>>
  i++ ; // instrucción de incremento
}
```

O también:

```
// n, i enteros

i = 1;
while (i<=n) { // i recorre 1,2,...,n (n vueltas)
  <<código>>
  i++ ; // instrucción de incremento
}
```


Construcción *for*

Para hacer `n` iteraciones es muy útil el bucle `for`:

```
for (i = 0 ; i<n ; i = i+1) {   // o i++
  <<código>>
}
```

Hace exactamente lo mismo que el bucle anterior:

```
i = 0;
while (i<n) { // i recorre 0,1,...,n-1 (n vueltas)
  <<código>>
  i = i+1 ; // instrucción de incremento
}
```

Nota: el incremento de la variable de control en el `for` ya está en la sección `«incremento»`, no hay que hacerlo (repetirlo) en el cuerpo del bucle!

Detalles del `for`

En general, la estructura de un bucle *for* es la siguiente:

```
for ( <<inicialización>> ; <<condición>> ; <<incremento>>)
  <<código>>
```

- `«inicialización»`: cualquier secuencia de instrucciones separadas por comas (habitualmente, una asignación simple).
- `«condición»`: cualquier condición (expresión booleana)
- `«incremento»`: cualquier secuencia de instrucciones separadas por comas (habitualmente una instrucción de incremento o similar).

Semántica (en realidad está dada por la traducción a `while`):

1. se ejecuta `«inicialización»` (solo una vez al principio)
2. se comprueba `«condición»`
   - 2.1 si es `true`:
     - 2.1.1 se ejecuta `«código»` (o cuerpo del bucle)
     - 2.1.2 se ejecuta `«incremento»`
     - 2.1.3 se vuelve al punto 2)
   - 2.2 si es `false` termina el bucle

Pinceladas de computabilidad

Nota: el bucle `while` proporciona toda la potencia expresiva necesaria para escribir cualquier programa...

- el `for` no es estrictamente necesario!!
- y de hecho, un bucle `for` es muy fácil de reemplazar por un bucle `while` (como hemos visto)

...pero el `for` es muy cómodo en muchos casos
$\rightsquigarrow$ claridad, brevedad...

- el `for` presenta a la vez la inicialización, la condición de parada y el incremento (en el `while` a veces cometemos el error de olvidar el incremento)

Algunos `while`'s revisitados

Escribir `n` asteriscos en pantalla:

```
int n, cont;
Console.Write ("Número de asteriscos: ");
n = int.Parse (Console.ReadLine ());

for (cont = 0; cont < n; cont++)
  Console.Write ("*");
```

Calcular los divisores de un número dado:

```
int n, i;

Console.Write ("Número: ");
n = int.Parse (Console.ReadLine ());

Console.Write ("Divisores: ");

for (i = 1; i<=n; i++)
  if (n % i == 0)
      Console.Write (i + " ");
```


Más bucles: bucles sobre `char`

Es posible iterar sobre el tipo `char`: tiene *comparación* y también incremento (apoyado en la codificación numérica interna).

Problema: escribir el código ASCII (o unicode) de las letras de la `'A'` a la `'Z'`

```
char c;

for (c='A'; c<='Z'; c++) // c recorre 'A','B',...,'Z'
   Console.WriteLine ((int) c); // escribimos su código
```


Otras condiciones de parada

Calcular la media de una secuencia de enteros leída de teclado (la secuencia termina con un negativo):

```
int n = 0,    // numero de datos
    suma = 0, // suma acumulada
    dato;     // valor leido

Console.WriteLine ("Introduce valores (negativo para terminar)");

// recogida de dato
Console.Write ("Dato: ");
dato = int.Parse(Console.ReadLine ());

while (dato >= 0) {
  suma = suma + dato; // acumulamos valor
  n++ ; // incrementamos contador

  // recogida de dato
  Console.Write ("Dato: ");
  dato = int.Parse (Console.ReadLine ());
} // fin while
if (n == 0) Console.WriteLine ("Sin datos de entrada. Media: 0");
else Console.WriteLine("Num. datos: "+n+". Media: "+((float) suma)/n);
```

Cuántas vueltas damos?

Problema: determinar si un entero dado es primo.

```
int n, i;
bool primo;

n = -1;
while (n < 2) {
  Console.Write ("Número (mayor o igual que 2): ");
  n = int.Parse (Console.ReadLine ());
}

primo = true; // suponemos que es primo hasta que encontremos un divisor
for (i=2; i<n; i++) {   // i = 2,3,...n-1
  if (n%i==0) primo = false;
  i = i+1;
}

if (primo)
  Console.WriteLine("El número " + n + " es primo.");
else
  Console.WriteLine("El número " + n + " no es primo.");
```

Incorrecto!!! Sobra el incremento `i=i+1;` en el cuerpo del `for`

Cuántas vueltas damos?

Corregimos el error anterior:

```
int n, i;
bool primo;

n = -1; // para forzar entrada en el bucle
while (n < 2) {
  Console.Write ("Número (mayor o igual que 2): ");
  n = int.Parse (Console.ReadLine ()); }

primo = true; // suponemos que es primo hasta que encontremos un divisor
for (i=2; i<n; i++)     // i = 2,3,...n-1
  if (n%i==0) primo = false;

if (primo) Console.WriteLine("El número " + n + " es primo.");
else Console.WriteLine("El número " + n + " no es primo.");
```

Es correcto? Es eficiente? Es mejorable?
Si ponemos el número 2147483646... cuántas vueltas da? Es tan costoso ver que este número NO es primo?

Cuántas vueltas damos? Búsquedas y recorridos

En este caso el bucle `for` no es apropiado: no es necesario hacer un recorrido de todo el intervalo de candidatos comprobando todos los posibles divisores `i = 2,3...n-1`

En el ejemplo anterior, 2 divide a 2147483646 $\rightsquigarrow$ no es primo y no hay que hacer ninguna comprobación más.
Debemos hacer una búsqueda de divisores `i = 2,3...` hasta

- encontrar un divisor
- o alcanzar el tope *i=n-1*

La diferencia entre hacer *recorrido* y *búsqueda*

- recorrido: se prueban *todos* los posibles candidatos (divisores potenciales) $\rightsquigarrow$ se suele utilizar for
- búsqueda: en cuanto se encuentra uno, el algoritmo termina $\rightsquigarrow$ bucle while

Muy importante diferenciarlos e implementar el que corresponda!!

Menos vueltas

```
...
primo = true; // suponemos que es primo hasta que encontremos un divisor
i = 2;
while (i < n && primo) {
  if (n % i == 0)
    primo = false;
  i++;
}
...
```

Más simplificado, sin variable booleana:

```
i = 2;
while ((i < n) && (n % i !=0)) i++;

if (i==n) Console.WriteLine("El número " + n + " es primo.");
else Console.WriteLine("El número " + n + " no es primo.");
```

Más simplificado aún

```
i = 2;
// buscamos el primer divisor >=2
while (n % i !=0) i++;
// si es primo, el primer divisor será el propio n

if (i==n) Console.WriteLine("El número " + n + " es primo.");
else Console.WriteLine("El número " + n + " no es primo.");
```

¿Más eficiente? Algo de álgebra al rescate:

- Necesitamos llegar a comprobar hasta el propio `n` o podemos parar antes?
- Necesitamos comprobar con el 2,3,4,5,6,... o hay alguna forma sencilla de eliminar candidatos?
- ... mucho para pensar. Para curiosos: ver criba de Eratóstenes.

Entrada salida con formato

Afinando la escritura en pantalla

Las cadenas de texto que escribimos con `Console.Write(...)` pueden formatearse de varios modos para obtener una salida más pulida:

- "Huecos" posicionales:

```
int i = 150 ;
double x = 1234.56789 ;
Console.WriteLine("i: {0}  x: {1}", i, x);
Console.WriteLine("i: {1}  x: {0}", x, i);
```

Escribe:

```
i: 150  x: 1234.56789
i: 150  x: 1234.56789
```

La parte `{n}` se refiere al parámetro `n`-ésimo, contando desde 0.

- En el primer caso `{0}` se refiere a la `i` y {1} a la `x`
- En el segundo caso `{0}` se refiere a la `x` y {1} a la `i`

- Número de dígitos y precisión en los reales

```
int i = 345;
double x = 123.456789;
Console.WriteLine ("i: {0:0000}  x: {1:00000.00}", i, x);
```

Escribe:

```
i: 0345  x: 00123,46
```

- La especificación `{0:0000}` escribe el valor de `i` en 4 huecos, i.e., los 0's indican el número de dígitos, rellenando con 0's por la izquierda. Si el número no cabe, se escribe entero (se obvia el formato).
- La especificación `{1:00000.00}` escribe el valor de `x` con 5 huecos para la parte entera y 2 decimales.

Como especificador de formato, `0` significa: reemplazar con el correspondiente dígito si lo hay; si no poner "0"

- Más control sobre los dígitos:

```
int i = 345;
double x = 123.456789;
Console.WriteLine ("i: {0:#.##}  x: {1:####.00#}", i, x);
```

Escribe:

```
i: 345  x: 123,457
```

- La especificación `{0:#.##}` escribe el valor de `i` en un hueco, i.e., los `#`'s indican el número de dígitos, pero no rellena con nada por la izquierda. Si el número no cabe, se escribe entero (se obvia el formato).
- La especificación `{1:####.00#}` escribe el valor de `x` con 4 huecos para la parte entera, 2 decimales forzados (pone ceros si no los tiene) y un decimal más, si lo hay (nada en otro caso).

Como especificador de formato, `#` significa: reemplazar con el correspondiente dígito si lo hay; si no, no poner nada (el anterior rellenaba con `0`).

- Escribiendo en columnas, justificación y huecos reservados:

```
int i = 345;
double x = 123.456789;
Console.WriteLine ("i: {0,6:0}  x: {1,10:0.00}", i, x);
Console.WriteLine ("i: {0,6:0}  x: {1,10:0.00}", 0, 0);
```

Escribe:

```
i:    345  x:     123,46
i:      0  x:       0,00
```

- La especificación `{0,6:0}` escribe `i` en 6 columnas, justificando a la derecha
- La especificación `{1,10:0.00}` escribe `x` en 10 columnas, con al menos una posición entera y exactamente dos posiciones decimales (alinea las comas decimales), justificando a la derecha.
- Para poner la justificación a la izquierda, basta con indicar el número de huecos *en negativo*: `{0,-6:0}` y `{1,-10:0.00}`