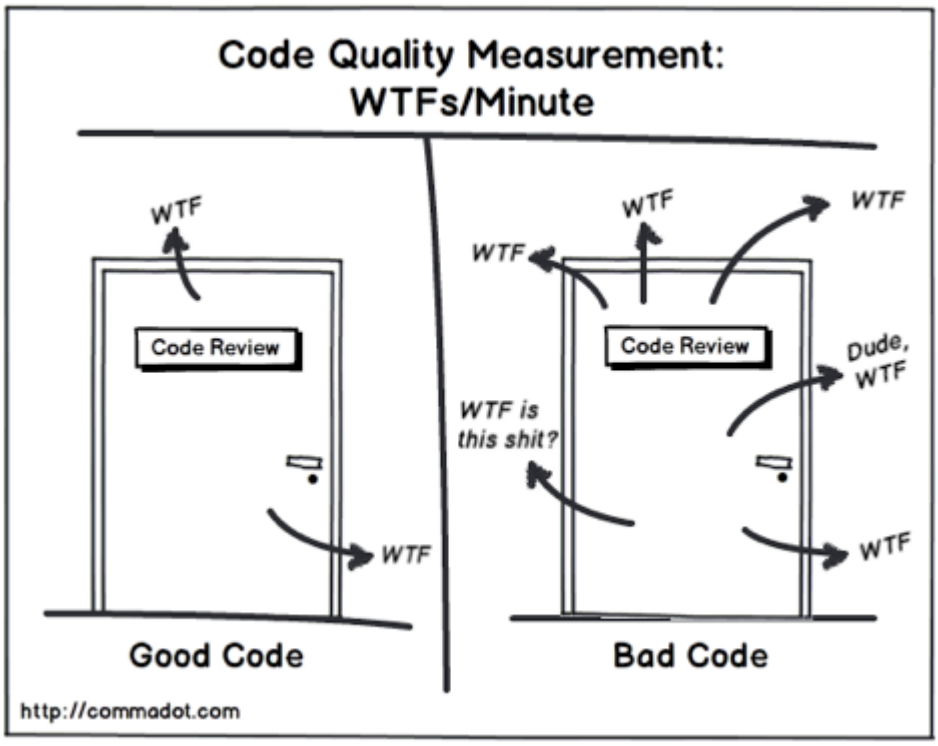
por Nacho Iborra

Entornos de Desarrollo
Bloque 1
Tema 9: Buenas prácticas de programación

1.9.1. Introducción al código limpio

Como introducción a lo que queremos que asimiléis como buenas guías de programación, en este documento se muestran algunos extractos del libro *Clean Code*, escrito por Robert C. Martin. Para empezar, aquí podemos ver una representación gráfica de cómo medir la calidad del código:



1.9.1.2. La importancia de la práctica

Hay dos partes a la hora de aprender artesanía: conocimiento y trabajo. Debes adquirir el conocimiento de los principios, patrones, prácticas y heurísticas que un artesano sabe, y debes también "machacar" esa sabiduría en tus dedos, ojos e interiorizarlo trabajando duro y practicando.

Puedo enseñarte las bases físicas de montar en bicicleta. De hecho, las matemáticas son relativamente sencillas. Gravedad, fricción, momento angular, centro de masas y todo eso, puede demostrarse en menos de una página llena de ecuaciones. Dadas esas fórmulas, puedo

demostrarte que montar en bici es práctico, y darte toda la sabiduría que necesitas para montar. Y aún así, caerías la primera vez que subieras a una bici.

Escribir código no es diferente a eso [···]

Aprender a escribir código limpio es un trabajo duro. Requiere más que sólo el conocimiento de los principios y patrones. Debes sudarlo. Debes practicar y verte fallar. Debes ver a otros practicar y fallar. Debes verles tropezar y volver sobre sus pasos. Debes verles agonizar sobre decisiones y ver el precio que pagan por hacer de esas decisiones el mal camino a seguir.

1.9.1.3. Efectos del mal código

Sé de una compañía que, a finales de los 80, escribió una aplicación innovadora. Fue muy popular, y muchos profesionales la compraron y utilizaron. Pero entonces los ciclos de entrega empezaron a estrecharse. Los errores no se reparaban de una versión a la siguiente. Los tiempos de carga crecieron, y los errores aumentaron. Recuerdo el día que cerré el producto frustado, y nunca más lo usé. La compañía quebró poco tiempo después.

Dos décadas después me encontré con uno de los primeros empleados de esa compañía, y le pregunté qué ocurrió. La respuesta confirmó mis temores. Lanzaron el producto al mercado con un gran desorden en el código. A medida que añadían más y más características, el código empeoraba hasta que simplemente no pudieron controlarlo por más tiempo. El mal código acabó con la compañía.

1.9.1.4. La imposibilidad de un gran rediseño

En algunas ocasiones, el equipo se rebela. Informan a dirección de que no pueden seguir desarrollando con esa odiosa base de código. Piden un rediseño. La dirección no quiere gastar más recursos en un nuevo rediseño, pero no pueden negar que la productividad es terrible. Así que se pliegan a las demandas de los desarrolladores y autorizan ese gran rediseño.

Un nuevo equipo es seleccionado. Todo el mundo quiere entrar en él, porque es un nuevo proyecto en blanco. Tienen que empezar de nuevo y crear algo bonito. Pero sólo los mejores son elegidos para ese equipo. Los demás deben continuar manteniendo el actual sistema.

Ahora los dos equipos están en una carrera. El equipo nuevo debe construir un nuevo sistema que haga todo lo que hace el antiguo. No sólo eso, deben actualizar los cambios que continuamente se están haciendo en el viejo sistema. La dirección no sustituirá el viejo sistema por el nuevo hasta que éste pueda hacer todo lo que hacía el viejo.

Esta carrera puede seguir durante mucho tiempo. La he visto durar 10 años. Y para cuando termine, la mayoría de los miembros del nuevo equipo se habrán ido, y los miembros actuales demandarán que el nuevo sistema debe ser rediseñado porque es un desastre.

1.9.1.5. Algunos motivos por los que existe el mal código

¿Te has encontrado alguna vez con un desorden tan grande que te ha llevado semanas lo que podría haberte llevado horas? ¿Has visto alguna vez algo que podría haber supuesto un cambio en una sola línea de código, en lugar de cambiar cientos de módulos diferentes? Estos síntomas son demasiado comunes.

¿Por qué ocurre esto con el código? ¿Cómo se transforma un buen código en mal código tan rápidamente? Hay varias explicaciones para eso. Podemos quejarnos de que los requisitos del programa han cambiado de forma que han invalidado el diseño original. Podemos argumentar que los plazos eran demasiado ajustados para hacer las cosas bien. Podemos criticar a los directivos estúpidos o clientes intolerantes, o estrategias estúpidas de marketing. Pero la culpa, querido Dilbert, no está en las estrellas, sino en nosotros mismos. No somos profesionales.

Esto puede ser un trago amargo. ¿Cómo puede este desorden ser culpa nuestra? ¿Qué hay de los plazos? ¿Qué hay de los directivos estúpidos y las estrategias estúpidas de marketing? ¿No tienen ellos parte de culpa?

No. Los directores y agentes de marketing nos demandan la información que necesitan para hacer promesas y establecer compromisos; e incluso cuando no acuden a nosotros, no deberíamos ser tímidos para decirles lo que pensamos. Los usuarios acuden a nosotros para validar la forma en que sus requisitos se ajustarán al sistema. Los gestores de proyectos nos buscan para hacer cumplir los plazos. Somos cómplices de la planificación del proyecto, y por tanto tenemos una buena parte de responsabilidad en cualquier fallo, especialmente si éstos tienen que ver con el mal código. "Pero espera", diréis, "si no hago lo que mis jefes dicen, me despedirán". Probablemente no. La mayoría de jefes quieren la verdad, aunque no lo parezca. La mayoría de jefes quieren buen código, aunque estén obsesionados con los plazos, y los defiendan apasionadamente, pero es su trabajo. El tuyo es defender el código con la misma pasión.

Para poner un ejemplo más familiar, ¿qué pasaría si fueras doctor y el paciente te pidiera que dejaras de hacer toda esa preparación estúpida de lavarse las manos antes de operar, porque lleva mucho tiempo? Obviamente, en este caso el paciente es tu cliente, y aún así el doctor rechazaría esa petición. ¿Por qué? Porque el doctor sabe más que el paciente sobre los riesgos de enfermedades e infecciones. Sería poco profesional (aunque no criminal) que el doctor accediera a esa petición.

De la misma forma, también es poco profesional doblegarse a los deseos de los jefes que no entienden los riesgos de ciertos desórdenes.

> Ejercicios propuestos:

> 1.9.1.1. ¿Estás de acuerdo con estos motivos? ¿Crees que puede haber otros motivos por los que existe el mal código?

1.9.1.6. Algunas definiciones de "buen código"

Bjarne Stroustrup, inventor de C++ y autor de The C++ Programming Language

Me gusta que mi código sea elegante y eficiente. La lógica debería ser directa, para que sea difícil que haya errores, las dependencias mínimas para que sea fácil de mantener, la gestión de errores completa siguiendo una estrategia previamente articulada, y el funcionamiento cercano a lo óptimo para evitar que la gente esté tentada de desordenar el código con optimizaciones sin control. El código limpio hace una cosa bien.

Grady Booch, autor de *Object Oriented Analysis and Design with Applications*

El código limpio es simple y directo. El código limpio se lee como una prosa bien escrita. El código limpio nunca oculta las intenciones del diseñador, y está lleno de nítidas abstracciones y líneas de control directas.

Dave Thomas, fundador de OTI, padrino de la estrategia Eclipse

El código limpio puede ser leído y mejorado por un desarrollador diferente a su creador original. Tiene pruebas de unidad y aceptación. Tiene nombres significativos. Proporciona una única forma, en lugar de varias, para hacer una cosa. Tiene dependencias mínimas, explícitamente definidas, y proporciona una API clara y mínima. El código debería ser literatura, dependiendo del lenguaje, ya que no toda la información puede expresarse claramente con sólo código.

Michael Feathers, autor de *Working Effectively with Legacy Code*

Podría listar todas las cualidades que veo en el código limpio, pero hay una que lidera a todas las demás. El código limpio siempre parece que haya sido escrito por alguien que se preocupa. No hay nada obvio que se pueda hacer para mejorarlo. Todas esas cosas ya han sido pensadas antes por su autor, y si intentas imaginar mejoras, te llevarán a donde estás, sentado apreciando el código que alguien te dejó, código hecho por alguien que se preocupa profundamente por el oficio.

Ward Cunningham, inventor de Wiki, inventor de Fit, coinventor de la programación extrema. Líder del pensamiento OO y de Smalltalk. Padrino de todos los que se preocupan por el código

Sabes que estás trabajando con código limpio cuando cada rutina que lees es exactamente como esperabas. Puedes llamarlo código bonito cuando el mismo código hace que el lenguaje estuviera hecho para resolver el problema.

Y, para finalizar con este apartado, cuando hablamos de código limpio, debemos tener presente la regla del Boy Scout:

Deja el campo más limpio de lo que te lo encontraste

> Ejercicios propuestos:

> 1.9.1.2. ¿Qué definición te gusta más, o piensas que es la más acertada? ¿Por qué?

> 1.9.1.3. ¿Cómo crees que se puede aplicar la regla del Boy Scout a la escritura de código?

1.9.2. Nombres de variables

Una vez aprendemos lo que es una variable y su principal cometido (almacenar valores que pueden ser consultados o modificados a lo largo de la ejecución de un programa), debemos emplear nombres significativos para las mismas. En esta sección veremos esta característica, y otras reglas que los nombres de variables deberían seguir.

Los nombres son esenciales en programación, ya que los usaremos para (casi) todo lo que añadamos a nuestros programas. En los primeros temas usamos nombres para variables, pero después los emplearemos para funciones, parámetros, clases, ficheros... etc.

1.9.2.1. Los nombres deben ser significativos

Cuando leemos un nombre de variable (o cualquier otro elemento del código), debe responder algunas preguntas básicas, como por ejemplo por qué existe, qué hace y cómo se usa. Si un nombre de variable necesita un comentario para saber qué hace, entonces no es un nombre adecuado.

Por ejemplo, si queremos almacenar en una variable la edad promedio de una lista de personas, no deberíamos hacer esto:

```
int ep;                 // Edad promedio
```

Podríamos hacer esto, en su lugar

```
int edadPromedio;
```

Excepción a la regla: bucles

Si estamos programando un bucle, probablemente necesitemos una variable entera para almacenar el número de iteraciones realizadas. Esta variable puede (debe) tener un nombre significativo, si tenemos planeado utilizarla fuera del bucle...

```
int contador = 0;
while (contador < 10)
{
        ...
```

... o podemos usar un nombre corto y típico (por ejemplo, i o n), para utilizar SÓLO en el bucle:

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

1.9.2.2. Evitar malentendidos, pequeñas variaciones y palabras ruidosas

Debemos evitar:

- Nombres que puedan llevarnos a equívoco, es decir, que parecen significar una cosa, pero en realidad significan otra. Por ejemplo, si llamamos a una variable `cuenta`, ¿para qué la utilizamos? ¿Una cuenta bancaria? ¿Una cuenta de usuario en un sitio web?
- Pequeñas variaciones en los nombres de variables. Dos variables diferentes deben tener dos nombres claramente diferenciables. Si tenemos una variable llamada `totalClientesRegistrados`, no es buena idea tener otra llamada `totalClientesNoRegistrados`, porque podemos confundirlas en el código y utilizar la que no toca. En lugar de esto, podríamos llamar a estas variables `registrados` y `anonimos`, por ejemplo.
- Palabras ruidosas, es decir, palabras que no aportan información útil al significado de la variable. Por ejemplo, si una variable se llama `nombreString`, la palabra *String* no aporta nada, porque ya podemos deducir que un nombre se va a almacenar en una cadena de texto. De la misma forma, el nombre `dinero` da la misma información que `cantidadDeDinero`.

1.9.2.3. Añadir contexto significativo

Hay algunas palabras que no son ruidosas, sino que ayudan a situar una variable en un contexto apropiado. Por ejemplo, si estamos almacenando datos sobre la dirección de una persona (nombre, apellidos, calle, ciudad, código postal...), pero sólo vemos una variable llamada `ciudad` en el código, ¿seríamos capaces de deducir que esa variable se utiliza para almacenar la ciudad de la dirección de la persona? Para estar seguros de ello, podemos añadir cierta información al nombre de la variable, y llamarla por ejemplo `ciudadDireccion` o `direccionCiudad`, lo cual es más probable que asociemos a una dirección completa que a un nombre de ciudad suelto.

1.9.2.4. Elegir una palabra por concepto

Intentemos siempre utilizar la misma palabra para expresar el mismo concepto. Por ejemplo, si implementamos varias aplicaciones para diversos clientes, y utilizamos en todas una variable para almacenar el *login* del usuario que accede, no deberíamos llamar a esa variable

`usuario` en una aplicación, `login` en otra, etc. Utilicemos siempre la misma palabra (`usuario` o `login`, en este caso).

Del mismo modo, no utilicemos la misma palabra para hablar de conceptos diferentes. Por ejemplo, no utilicemos la palabra `suma` para almacenar cualquier tipo de cálculo, salvo que efectivamente sean sumas. Es mejor utilizar `total` o `resultado` en este caso.

1.9.2.5. Otras características deseables

Además de todas las recomendaciones anteriores, los nombres de variables deben seguir otras reglas, como por ejemplo:

- Debemos asignar nombres que se puedan pronunciar, de modo que al discutir sobre una variable o su valor, podamos nombrarla. Si estamos almacenando la fecha de nacimiento de una persona, podemos llamar a la variable `fechaNacimiento`, pero no `ddmmyyyy`, por ejemplo, ya que es difícil de pronunciar.
- Si utilizamos nombres cortos de variables (por ejemplo, de una sola letra), será difícil encontrar esa variable en el código fuente, porque se mezclará con otros elementos que contendrán esa letra.
- Hace algunos años, era bastante habitual emplear cierto tipo de prefijos o sufijos en los nombres de variables que daban cierta información sobre la misma. Por ejemplo, podíamos llamar a una variable `iEdad`, donde el prefijo *i* indicaba que era una variable entera. Esta costumbre estaba promovida por ciertos lenguajes en que los tipos de datos se declaraban como prefijos, pero hoy en día hay muchos nuevos lenguajes de programación que no necesitan esta regla, y muchos IDEs que nos ayudan a averiguar el tipo de dato de una variable muy fácilmente, por lo que ya no se recomienda usar esos prefijos.

1.9.2.6. ¿Mayúsculas o minúsculas?

El uso de mayúsculas o minúsculas en los nombres de elementos depende del lenguaje de programación normalmente. Hay varios estándares al respecto:

- **Camel Case**: empleado en lenguajes como Java o Javascript. Cada palabra que compone el nombre de la variable, salvo la primera, empieza en mayúsculas. Por ejemplo:

```
String nombrePersona;
```

- Hay un subtipo de *camel case* llamado **Pascal Case** en el que la primera palabra del nombre también tiene su primera letra mayúscula. Este subtipo es empleado por C# para definir elementos públicos (los elementos privados o protegidos se nombran empleando *camel case*). Por ejemplo:

```
String nombrePersona;
public int EdadPersona;
```

- **Snake Case**: empleado en lenguajes como PHP. Las palabras del nombre de la variable se separan con subrayados o barras bajas:

```
$nombre_persona = "Nacho";
```

- **Kebab Case**: las palabras del nombre de variable se separan por guiones. Esta notación no es muy habitual en lenguajes de programación, porque muchos no admiten el guión como parte del nombre de variable (para no confundirlo con el operador de resta). Hay algunos pocos ejemplos, no obstante, como Lisp o Clojure:

```
(def nombre-persona "Nacho")
```

- **Mayúsculas**: en algunos lenguajes se emplean sólo mayúsculas para definir constantes. Las palabras que componen el nombre se suelen separar por subrayados o barras bajas, como en el caso de *snake case*:

```
const int TAMANO_MAXIMO = 100;
```

1.9.3. Comentarios

Los comentarios bien ubicados ayudan a entender el código que les rodea, mientras que los comentarios innecesarios o fuera de lugar pueden dañar la comprensión del mismo. Algunos programadores piensan que los comentarios son fallos, y deben evitarse todo lo posible. Una de las razones argumentadas es que son difíciles de mantener. Si cambiamos el código tras un comentario pero olvidamos actualizar el comentario, hará referencia a algo que ya no existe en nuestro código.

Otra razón para evitar comentarios es que están fuertemente ligados al mal código. Cuando escribimos mal código, normalmente pensamos que al escribir comentarios lo hacemos más comprensible, en lugar de limpiar el código propiamente dicho.

En esta sección aprenderemos dónde colocar comentarios. Primero, veremos qué tipos de comentarios pueden ser necesarios (lo que llamaremos *buenos comentarios*), y después veremos qué comentarios podemos y debemos evitar (*malos comentarios*).

1.9.3.1. Buenos comentarios

Los siguientes comentarios se consideran correctos y necesarios:

- **Comentarios legales**, como por ejemplo el copyright o autoría, siguiendo los estándares de la compañía para la que trabajemos. Este tipo de comentarios suele colocarse al principio de cada fichero fuente perteneciente al autor o compañía.
- **Comentarios de introducción**, un breve comentario al principio de cada fichero fuente que explica su principal propósito.
- **Explicación de intención**. Estos comentarios se utilizan cuando:
    - Hemos intentado obtener una mejor solución a un fragmento de código, pero no hemos podido; en ese caso, expicamos que una parte del código puede ser mejorable.
    - Hay una parte del código que no sigue el mismo patrón que el código que le rodea (por ejemplo, una variable entera entre un grupo de variables reales). En ese caso, queremos explicar el propósito de dicha variable o el uso de ese tipo de datos en concreto.
- **Comentarios *TODO***, que se colocan en partes del código por completar. Nos ayudan a recordar todas las tareas pendientes. Este tipo de comentarios ha llegado a ser tan popular que muchos IDEs los detectan automáticamente, e incluso los resaltan con colores diferentes.
- **Documentación para API**. Algunos lenguajes de programación, como Java o C#, permiten añadir comentarios en algunas partes del código con una estructura que permite después exportarlos a un formato HTML o XML, y formar así parte de la documentación del programa o librería que estemos desarrollando.

1.9.3.2. Malos comentarios

Los siguientes son ejemplos de malos comentarios que debemos evitar:

- Algunos tipos de **comentarios de información** pueden evitarse cambiando el nombre del elemento al que hacen referencia. Por ejemplo, si tenemos este comentario con esta variable:

```
// Número total de clientes registrados
int total;
```

Podemos evitar el comentario renombrando la variable de esta forma:

```
int totalDeClientesRegistrados;
```

- **Comentarios redundantes**, es decir, comentarios que son más largos de leer que el código que intentan explicar, o que simplemente son innecesarios, porque el código se explica por sí mismo. Por ejemplo, el siguiente comentario es redundante, porque el código que explica se entiende muy bien por sí mismo:

```
// Comprobamos si la edad es mayor que 18, y si es así, imprimimos un
// mensaje diciendo que "Eres lo suficientemente mayor"
if (edad > 18)
    Console.WriteLine("Eres lo suficientemente mayor");
```

- **Comentarios sin contexto**, es decir, comentarios que no van seguidos de ningún código asociado. Por ejemplo, el siguiente comentario no está completado por un código apropiado. Hay parte del código que falta. Decimos en el comentario que, si el usuario no es adulto, se le hará salir de la aplicación, pero no hay ningún código para ello tras el comentario. Puede que el código esté en otra parte del programa, pero entonces el comentario también debería estar allí

```
if (edad > 18)
{
    Console.WriteLine("Eres lo suficientemente mayor");
} else {
    // Si el usuario no es adulto, se le hace salir de la aplicación
}
```

- **Comentarios obligados**: algunas personas creen que cada variable, o cada función, por ejemplo, debe tener un comentario explicando lo que hace. Pero esta no es una buena decisión, porque muchos de esos comentarios se pueden evitar empleando nombres apropiados (significativos).

- **Comentarios de historial**: a veces se tiene un registro de cambios al principio de los ficheros fuente. Contiene una descripción de los cambios hechos en el código, indicando fecha, versión, y cambios realizados. Pero hoy en día, existen muchas aplicaciones de control de versiones, como por ejemplo GitHub, que permiten gestionar este registro fuera del código.

- **Marcadores de posición y divisores de código**: es muy habitual hacer comentarios que permiten identificar rápidamente una parte del código, o separar bloques de código que son algo largos. Ambos tipos de comentarios están desaconsejados, si el código se formatea correctamente.

```
// ================== VARIABLES ====================
int edad;
string nombre;
…
// ================== MAIN =========================
public static void Main()
{
    …
    ////// RESULTADO FINAL
}
```

- **Comentarios de cierre de llaves**, que se colocan tras cada llave de cierre, explicando qué elemento están cerrando (un *if*, un *while*...). Estos comentarios pueden evitarse, ya que muchos IDEs resaltan las llaves que se emparejan, de forma que podemos identificarlas fácilmente.

```
while (n > 10)
{
    if (n > 5)
    {
        …
    } // if
} // while
```

- **Advertencias**, que se emplean cuando tenemos cierto código que puede causar problemas en ciertas situaciones, y necesita ser revisado. Es muy habitual encontrar algunos bloques de código completamente comentados, a la espera de que alguien resuelva el problema. Estos comentarios deberían convertirse en comentarios "TODO", para advertir al programador de que ese código necesita ser revisado en el futuro.

> Ejercicios propuestos:

> 1.9.3.1. El siguiente programa le pide al usuario que introduzca tres números y obtiene la media de los tres. Discutid en clase qué partes del programa no están limpias o podrían mejorarse, en lo que se refiere a nombres de variables y comentarios.

```
using System;

public class MediaNumeros
{
        public static void Main()
        {
                // Variables para almacenar los tres números y la media
                int n1, n2, n3;
                int Resultado;

                // Pedimos al usuario que introduzca tres números
                Console.WriteLine("Introduce tres números:");
                n1 = Convert.ToInt32(Console.ReadLine());
                n2 = Convert.ToInt32(Console.ReadLine());
                n3 = Convert.ToInt32(Console.ReadLine());
                // El resultado es la media de los tres números
                /* Podríamos haber empleado una variable real para la med
                   mantener el programa lo más simple posible */
                Resultado = (n1+n2+n3)/3;
                Console.WriteLine("La media es {0}", Resultado);
        }
}
```

1.9.3.2. Este programa le pide al usuario que introduzca números hasta que escriba uno negativo, o hasta un total de 10 números. Cópialo en tu IDE e intenta mejorarlo con nombres de variables y comentarios apropiados.

```
/*
 * (C) IES San Vicente 2016
 */
using System;

public class IntroducirNumeros
{
        public static void Main()
        {
                // Números introducidos por el usuario
                int numero1;
                // Conteo de números
                int numero2 = 0;

                do
                {
                        // Le pedimos al usuario que escriba un número
                        Console.WriteLine("Escribe un número: ");
                        numero1 = Convert.ToInt32(Console.ReadLine());
                        numero2++;
                } while (numero2 < 10 && numero1 >= 0); // do..while

                if (numero1 < 0)
                        Console.WriteLine("Terminando. Introdujiste un nú
                else
                        Console.WriteLine("Terminando. Introdujiste 10 nú
        }
}
```

1.9.3.3. Crea un programa llamado DibujoRectangulo que le pida al usuario la base y altura de un rectángulo, y lo dibuje con las dimensiones indicadas, relleno de asteriscos. Por ejemplo, si el usuario escribe una base de 5 y una altura de 3, el programa debería mostrar esto:

```
*****
*****
*****
```

Implementa el programa siguiendo las normas explicadas en cuanto a nombres de variables y comentarios.

1.9.4. Espaciado y formateado del código

Un espaciado y formateado de código apropiados indican al lector que el programador ha prestado atención a cada detalle del programa. Sin embargo, cuando encontramos un conjunto de líneas de código incorrectamente indentadas o espaciadas, podemos pensar que esa misma falta de atención puede estar presente en otros aspectos del código.

1.9.4.1. Espaciado vertical

Veamos algunas normas sencillas para espaciar nuestro código verticalmente:

- Como cada grupo de líneas representa una tarea, estos grupos deben estar separados del resto por una línea en blanco. En un programa en C#, por ejemplo, podemos tener algo como esto (observa dónde se colocan las líneas en blanco):

```
using System;

public class Programa
{
                public static void Main()
                {
                        int edadPersona;
                        string nombrePersona;

                        Console.WriteLine("Dime tu nombre:");
                        nombrePersona = Console.ReadLine();

                        Console.WriteLine("Dime tu edad:");
                        edadPersona = Convert.ToInt32(Console.ReadLine())

                        if (edadPersona > 18)
                                Console.WriteLine("Eres adulto, {0}", nom
                }
}
```

- Los conceptos que están fuertemente relacionados deben colocarse juntos verticalmente. Por ejemplo, si declaramos dos variables para almacenar el nombre y edad de una persona, deberíamos colocar estas declaraciones juntas, sin separaciones. Esto implica que no deberíamos añadir ningún comentario en medio que rompa esa unión:

```
string nombrePersona;
/*
 * Este comentario no debería escribirse aquí
 */
int edadPersona;
```

- Las llaves de apertura se pueden poner o bien al final de la línea que las necesita (esto es habitual en lenguajes como Java o Javascript), o bien al principio de la siguiente línea, con la misma indentación que la línea anterior (típico en lenguajes como C o C#). En este último caso, las llaves de apertura pueden actuar como línea en blanco de separación entre bloques.

```
// Estilo Java (la llave de apertura NO se considera línea de separación)
if (condicion) {
        ...
}

// Estilo C# (la llave de apertura puede considerarse línea de separación
public static void Main()
{
        if (condicion)
        {
                ...
        }
...
```

En cuanto a las llaves de apertura, podemos decidir cuál de los dos formatos emplear, pero debemos:

- Aplicar siempre el mismo patrón
- Usar el mismo patrón que toda la gente de nuestro equipo de trabajo

1.9.4.2. Formateado horizontal

En cuanto al espaciado o formateado horizontal, hay también algunas reglas simples que podemos seguir:

- Una línea de código debe ser corta (como mucho 80 o 100 caracteres). Algunos IDEs muestran una línea vertical (normalmente roja) que indica la longitud "ideal" máxima de la línea. Si va a ser más larga que ese límite, deberíamos cortarla y dividir el código en múltiples líneas. También podemos aplicar otras reglas para determinar la longitud máxima: no deberíamos hacer *scroll* hacia la derecha para ver nuestro código, y al imprimirlo en una página vertical, debería tener la misma apariencia.

```
if ((edadPersona > 18 && edadPersona <= 65) ||
    (nombrePersona == "John") || (nombrePersona == "Mary"))
{
        ...
}
```

- El espaciado horizontal nos ayuda a asociar cosas que están relacionadas, y a desasociar cosas que no lo están. Por ejemplo, los operadores deberían separarse de los elementos con los que operan con un espacio en blanco:

```
int media = (numero1 + numero2) / 2;
```

- No alinear los nombres de variables verticalmente. Esta práctica era bastante habitual en algunos lenguajes antiguos, como el ensamblador, pero no tiene sentido en lenguajes modernos, donde hay muchos tipos de datos diferentes. Si hacemos esto, corremos el riesgo de tender a leer los nombres de variables sin prestar atención al tipo de dato que tienen:

```
StringBuilder   textoLargo;
int                             tamanoTexto;
string                  textoABuscarYReemplazar;
```

- La indentación es importante, porque establece una jerarquía. Hay elementos que pertenecen al fichero fuente completo, y otros que pertenecen a un bloque concreto. La indentación nos ayuda a determinar el ámbito de un grupo de instrucciones. En este sentido:
  - El nombre de la clase no se indenta
  - Las funciones y otros elementos dentro de clases se indentan un nivel
  - La implementación de estas funciones se indenta dos niveles
  - Los bloques dentro de cada función (*if* o *while*, por ejemplo) se indentan tres niveles
  - ··· etc.

```
public class MiClase
{
        public static void Main()
        {
                Console.WriteLine("Hola");
                if (...)
                {
                        Console.WriteLine("Dentro de un if");
                }
        }
}
```

1.9.5. Funciones

Las funciones son una forma de organizar el código de nuestros programas. Nos ayudan a dividirlo en diferentes módulos, y llamar/utilizar cada uno siempre que lo necesitemos. Sin embargo, si se escriben de forma incorrecta, pueden ser parte del problema, en lugar de ser parte de la solución. EN esta sección aprenderemos cómo deben organizarse las funciones y algunas normas básicas para escribir código limpio en ellas.

1.9.5.1. El tamaño importa

La regla que quizá sea más importante para desarrollar una función es que ésta debe ser pequeña. Pero el adjetivo *pequeña* es algo difuso... ¿cuántas líneas se considera "pequeño"? Bueno, algunos años atrás los expertos decían que una función no debía ser más larga que una página impresa. Pero los tipos de letra entonces eran muy limitados, y sólo permitían unas pocas líneas por página; hoy en día podemos reducirlos hasta conseguir unas 100 líneas en una sola página. ¿Significa esto que una función puede tener 100 líneas de código? La respuesta es NO... una función debería tener en torno a 20 líneas de código como mucho.

Para cumplir esto, debemos dividir nuestro código en funciones, y hacer llamadas entre ellas correctamente. Esto implica que el nivel de indentación de una función no debería exceder de uno o dos (contando con alguna que otra estructura como `if` o `while` o similar).

1.9.5.2. Tareas simples

¿Cuál es la mejor forma de dividir nuestro código de forma que cada función no ocupe más de 20 líneas? Debemos dividir el programa en tareas, y estas tareas en subtareas, si es necesario, y así sucesivamente, de forma que cada función implemente una única tarea.

Consideremos el siguiente programa: debemos validar el login de un usuario, y entonces, imprimir su perfil en pantalla. El programa principal estaría compuesto de dos tareas principales:

1. Validar usuario
2. Imprimir perfil

Debemos definir una función para cada una de estas dos tareas, y así entramos en la labor de dividir cada tarea en subtareas. Por ejemplo, para la primera función o tarea:

1. Validar usuario 1.1. Imprimir el mensaje "Login usuario:" en pantalla 1.2. Obtener el login del teclado 1.3. Imprimir el mensaje "Password usuario:" en pantalla 1.4. Obtener el password del teclado 1.5. Comprobar el usuario en la base de datos 1.6. Si es correcto, finalizar la función 1.7. Si no es correcto, volver al paso 1.1

Casi todas estas subtareas son lo suficientemente simples como para implementarse en una o dos líneas de código, por lo que no necesitamos más niveles... salvo para la subtarea 1.5 que necesitaría un subnivel más:

1.5. Comprobar el usuario en la base de datos 1.5.1. Conectar a la base de datos 1.5.2. Buscar el login y password introducidos 1.5.3. Determinar si existe o no el usuario

Si trasladamos este esquema a código, podríamos tener algo así:

```
public static void Main()
{
        ValidarUsuario();
        ImprimirPerfilUsuario();
}

public static void ValidarUsuario()
{
        bool resultado = false;
        do
        {
                Console.WriteLine("Login usuario:");
                string login = Console.ReadLine();
                Console.WriteLine("Password usuario:");
                string password = Console.ReadLine();
                resultado = ComprobaUsuarioEnBD(login, password);
        } while (!resultado);
}

public static void ComprobarUsuarioEnBD(string login, string password)
{
        ...
}

...
```

1.9.5.3. Nombres de funciones

Del mismo modo que las variables deben tener nombres apropiados, las funciones también deben tener nombres que expliquen claramente lo que hacen. Debemos intentar aplicar el mismo patrón de nombrado a funciones que sean similars, y utilizar un verbo en el nombre, ya que las funciones HACEN algo. Por ejemplo, si tenemos un grupo de funciones para validar que los datos que introduce el usuario son correctos, podemos llamar a estas funciones `validarNumero`, `validarCodigoPostal`, `validarTelefono`... En todos estos nombres empleamos un verbo (*validar*), y el mismo patrón (*validarAAA*, *validarBBB*...). Sería un error llamar a una función `validarNumero`, a otra `comprobarCodigoPostal`, etc.

En cuanto al uso de mayúsculas y minúsculas, las funciones siguen la misma regla que las variables. En Java, por ejemplo, todas las funciones (salvo los constructores) empiezan por minúscula, y cada palabra interna del nombre empieza por mayúscula (*camel case*):

```
public void miFuncion() { … }
```

En C#, esta regla se aplica sólo a elementos que no sean públicos. Si una función es pública, también empieza por mayúscula (*pascal case*):

```
public void MiFuncion() { … }
int otraFuncion() { … }
```

1.9.5.4. Funciones y espaciado

Debemos prestar atención a las reglas de espaciado vistas antes para formatear y espaciar el código de nuestras funciones. Para empezar, debe haber una linea en blanco entre cada par de funciones:

```
public void Funcion1()
{
        ...
}

public int Funcion2()
{
        ...
}
```

Las funciones que están fuertemente relacionadas deben ponerse juntas en el código, de forma que no tengamos que explorar demasiado para buscar una de ellas desde otra. Esta afinidad puede ser debida a dependencia (que una función utilice la otra), o a funcionalidad similar.

- En cuanto a la dependencia, si una función A llama o utiliza otra llamada B, entonces A debe ir antes que B.
- En cuanto a la funcionalidad, consideremos un grupo de funciones que tienen el mismo prefijo o sufijo (`nuevoNombre`, `nuevoCliente`...). Estas funciones deberían estar cercanas en el código.

1.9.5.5. Otras características

Hay otras características que son deseables en una función

- **Evitar efectos laterales**: el código de una función no debe modificar nada externo a dicha función directamente, ni hacer nada que no se deduzca implícitamente de su

nombre. Por ejemplo, una función que se llame `validarUsuario` no debería almacenar nada en un fichero.

- **Seguir la regla de la programación estructurada**: según la programación estructurada, cada bloque de código (por ejemplo, un bucle, o una función), debe tener sólo un punto de entrada y un punto de salida. Esto implica que cada función debe tener una única instrucción `return` que devuelva su valor, y cada bucle debe tener un único punto de salida (debemos evitar el uso de `break` o `continue` en los bucles).

> Ejercicios propuestos:

> 1.9.5.1. El siguiente ejemplo le pregunta al usuario que indique cuántos nombres va a introducir, inicializa un array con ese tamaño y almacena los nombres en él. Finalmente, ordena los nombres alfabéticamente y los imprime. Echa un vistazo al código e intenta mejorarlo en términos de limpieza de código.

```
using System;

public class OrdenarNombres
{
        static void ImprimirNombres(string[] nombres)
        {
                Console.WriteLine("Esta es la lista de nombres en orden a
                foreach(string nombre in nombres)
                {
                        Console.WriteLine(nombre);
                }
        }
        public static void Main()
        {
                int totalNombres;
                string[] nombres;
                string aux;
                Console.WriteLine("Indica el total de nombres a leer:");
                totalNombres = Convert.ToInt32(Console.ReadLine());
                nombres = new string[totalNombres];
                Console.WriteLine("Introduce los {0} nombres:", totalNomb
                for(int i = 0; i < totalNombres; i++)
                {
                        nombres[i] = Console.ReadLine();
                }
                for (int i = 0; i < totalNombres - 1; i++)
                        for (int j = i+1; j < totalNombres; j++)
                                if (nombres[i].CompareTo(nombres[j]) > 0)
                                {
                                        aux = nombres[i];
                                        nombres[i] = nombres[j];
                                        nombres[j] = aux;
                                }
                ImprimirNombres(nombres);
        }
}
```

1.9.5.2. El siguiente código muestra algunas funciones que devuelven valores, y un *Main* que utiliza dichas funciones. Intenta mejorar el código en términos de limpieza:

```
using System;

// Devuelve si un número es par o no
public static boolean esPar(int numero)
{
        if (numero % 2 == 0)
                return true;
        else
                return false;
}

// Busca una palabra en un array de cadenas y devuelve la posición donde
// Si la palabra no se encuentra, devuelve -1
public static int buscar(string[] array, string palabra)
{
        for (int i = 0; i < array.Length; i++)
        {
                if (array[i] == palabra)
                        return i;
        }
        return -1;
}

public static void Main()
{
        Console.WriteLine("Introduce un número:");
        int numero = Convert.ToInt32(Console.ReadLine());
        bool par = esPar(numero);
        if (par)
                Console.WriteLine("El número es par");
        else
                Console.WriteLine("El número es impar");
        string[] palabras = {"hola", "adiós", "uno", "dos", "tres", "cuat
        Console.WriteLine("Introduce una palabra:");
        string palabra = Console.ReadLine();
        int resultado = buscar(palabras, palabra);
        if (resultado >= 0)
                Console.WriteLine("Palabra encontrada en posición " + res
        else
                Console.WriteLine("Palabra no encontrada");
}
```

1.9.6. Otros elementos que considerar

Para finalizar con este tema, vamos a echar un vistazo rápido a otros elementos que deben ser tratados cuidadosamente, como las clases o el tratamiento de errores.

1.9.6.1. Utilizar clases

Aunque puede que no sepas qué es una clase por ahora, quizá necesites conocer algunos conceptos útiles para escribirlas correctamente:

- **Estructura estándar de una clase**: los estándares dicen que una clase debe tener una lista de variables o atributos propios, y después, los constructores y métodos. En cuanto a los atributos, normalmente indicamos primero las constantes, y luego las variables. Por ejemplo, este podría ser el código para una clase llamada *Persona* que gestiona el nombre y la edad de la gente:

```
class Persona
{
        string nombre;
        int edad;

        public Persona(string unNombre, int unaEdad)
        {
                nombre = unNombre;
                edad = unaEdad;
        }

        public void DecirHola()
        {
                Console.WriteLine("Hola {0}, tienes {1} años", nombre, ed
        }
}
```

- **Encapsulación**: es también recomendable que los atributos de la clase se mantengan privados (o protegidos, en caso de herencia). Las clases y objetos ocultan sus datos y exponen funciones para trabajar con ellos. Es lo que se conoce como *encapsulación*.
- **Cohesión y clases pequeñas**: cuantos más atributos emplee un método de una clase, más cohesión tiene ese método con la clase. Una clase donde todos sus atributos se manipulan por todos su métodos está óptimamente cohesionada. Nuestro objetivo es tener una cohesión lo más alta posible, lo que significa que los métodos y los atributos de la clase dependen unos de otros, y actúan como un todo lógico. Además, para intentar mantener esta cohesión, será deseable que las clases sean pequeñas y con pocos elementos (cuantos más tengan, más complicado será mantener la cohesión).
- **Nombres de clases**: una clase se nombra normalmente con un nombre que típicamente va en singular. Por ejemplo, *Persona*, *Forma*, *Animal*, *StringBuffer*... Además, el nombre de la clase debe dar suficiente información sobre el propósito de la misma, y sus responsabilidades. Por ejemplo, una clase llamada *String* nos da a entender que gestionará todos los métodos para manipulación de textos, como por ejemplo extraer fragmentos del texto, buscar por una frase en un texto, reemplazar un texto por otro, etc.

1.9.6.2. Gestión de errores

La gestión de errores es importante en nuestros programas, porque el usuario puede cometer errores al introducir la información, o algunos dispositivos o elementos pueden fallar (por ejemplo, cuando intentemos leer o escribir datos en un archivo). Sin embargo, si no utilizamos la gestión de errores adecuadamente, nuestro código puede verse dañado, y su lógica principal escondida por la propia gestión de errores. En esta sección veremos algunas técnicas que nos ayudan a escribir código que es limpio y, a la vez, robusto.

1.9.6.2.1. Excepciones y códigos de error

En lenguajes de programación antiguos no existían las excepciones, y la única forma de detectar errores era crear una variable booleana para almacenar si cierta operación era correcta, o devolver un valor numérico que represente el error producido (dependiendo del valor numérico, se averiguaba el error asociado a dicho número).

Sin embargo, si un lenguaje de programación permite tratar con excepciones, es mejor utlizarlas en lugar de emplear códigos de error. Siempre que un bloque de código pueda causar un error, deberemos intentar capturarlo (*catch*) o propagarlo para que se capture más adelante (*throw*). Pero hay algunos aspectos que debemos tener en cuenta cuando trabajemos con excepciones:

- **Proporcionar contexto**: cuando obtengamos el mensaje de error de una excepción, éste debe ser lo suficientemente claro como para dejarnos ver cuál ha sido el error. Por lo tanto, cuando lancemos una excepción en el código, debemos procurar acompañarla de un mensaje de error apropiado. Algo típico es indicar la operación que falló, y el tipo de fallo: "*Error leyendo de fichero. El fichero 'datos.txt' no existe en /home/user*".

- **Escribir los bloques *try..catch* primero**: puede ser una buena idea escribir los bloques `try..catch` antes de comenzar con las líneas de código que irán dentro de dicho bloque. El bloque *try* define un ámbito, como lo hacen `if` o `while`, y cuando lo añadimos, debemos ser conscientes de que la ejecución puede romperse en cualquier punto dentro de ese *try*, y entonces deberemos dejar el programa en un estado consistente para poder continuar. Por otra parte, comenzar una función que puede provocar una excepción con un bloque *try..catch** le indica al usuario que ahí se espera una excepción, no importa cuál, o cuándo.

- **Diferenciar tipos de excepciones si se requiere**: en muchos programas, puede ser conveniente mostrar mensajes de error diferente para excepciones diferentes. Por ejemplo, un mensaje de error si no se encuentra el fichero a leer, y otro diferente si no se ha podido leer un dato del mismo.

1.9.6.2.2. El uso de null

En las APIs oficiales, podemos encontrar muchos métodos que devuelven `null` cuando algo ha ido mal. Por ejemplo, hay métodos para leer datos de un fichero que devuelven

`null` cuando ya no hay nada más que leer.

Sin embargo, devolver `null` puede ser una fuente de nuevos errores, porque obligamos al programador a comprobar si un dato es nulo antes de continuar:

```
String valor = miObjeto.metodoQuePuedeDevolverNull();

if (valor != null)
{
        // Resto del código que realmente importa
}
```

Observad que, si lo hacemos así, estamos indentando un bloque de código con un nivel extra que realmente no le corresponde. Además, hay otros elementos que no se han comprobado. ¿Qué pasaría si `miObjeto` fuera nulo? No deberíamos tener que hacer esta comprobación en ninguna parte de nuestro código.

Pero, ¿qué podemos hacer para evitar devolver `null`? Tenemos algunas alternativas:

- Lanzar una excepción con un mensaje apropiado
- Devolver un valor especial que represente el error. Por ejemplo, un número negativo si estamos buscando un elemento en un array, y no lo encontramos.

Del mismo modo, no es buena idea pasar `null` como parámetro a ningún método o función, porque podríamos lanzar una excepción involuntariamente.

> Ejercicios propuestos

> 1.9.6.1. El siguiente ejercicio le pide al usuario que introduzca un número de edades, y después calcula la media de las mismas. Corrige el código en términos de limpieza y gestión de errores para que sea limpio y robusto.

```
using System;

public class MediaEdades
{
        public static void Main()
        {
                int numeroEdades;
                int[] edades;

                Console.WriteLine("Dime cuántas edades vas a introducir:"
                numeroEdades = Convert.ToInt32(Console.ReadLine());
                edades = new int[numeroEdades];

                for (int i = 0; i < numeroEdades; i++)
                {
                        edades[i] = Convert.ToInt32(Console.ReadLine());
                }
                int total = 0;
                for (int i = 0; i < numeroEdades; i++)
                        total += edades[i];

                int media = total / numeroEdades;
                Console.WriteLine("La edad promedio es " + media);
        }
}
```