```
System.out.println("**********************");
System.out.println(" C L E A N  C O D E ");
System.out.println("**********************");
System.out.println("** Paula Búrdalo  **");
System.out.println("**********************");
```

ÍNDICE

En el archivo .java de este repositorio están los ejemplos de algunos puntos, aunque van a estar las capturas en este pdf.

Bloque 1 : Nombres

1.1 Nombres con significado

Es muy importante usar nombres que tengan que ver con su función, ya sea en un método, variable, clase, etc.

En este ejemplo, vemos una variable inicializada en 12, pero no sabemos a qué se refiere ya que no tiene un nombre descriptivo. La segunda variable se llama meses y está iniciada a 12, sabemos a qué se refiere.

```java
// 1.1 Nombres con significado
int variable = 12;
int meses = 12;
```

1.2 Nombres con fácil pronunciación

Siempre se dice que los programadores pasan más tiempo leyendo código que escribiéndolo, por eso es necesario que los nombres los podamos entender, ya que si tenemos alguna duda, va a ser más complicado preguntar por una variable llamada “kjasdnuw”.

Aquí, vemos una variable que contiene mayormente consonantes y es imposible de pronunciar, la segunda variable puede ser una abreviación de algo, en caso de no ser algo directamente descriptivo pero al menos es pronunciable, como cuando decimos “sout” en vez de “system out”.

```java
// 1.2 Nombres con fácil pronunciación
String ifhy = "???";
String sout = "abreviación de system out";
```

1.3 Nombres que se puedan buscar

Este punto es muy parecido al 1.1, si usamos directamente el valor de la variable en vez del nombre como tal, puede que no nos acordemos después de qué era ese número.

En este bucle, no sabemos qué es 12, puede ser una constante o puede que la necesitemos cambiar en algún momento, sin embargo, en el segundo bucle sabemos que se refiere a la variable “meses”, la podemos buscar en el código y ver qué valor tiene.

```java
// 1.3 Nombres que se puedan buscar
for (int i = 0; i < 12; i++) {
    System.out.println(x: "no sabemos qué es 12");
}

for (int i = 0; i < meses; i++) {
    System.out.println(x: "sabemos que 12 se refiere a meses");
}
```

1.4 Nombres de clases, métodos o funciones

Los nombres de las clases deben ser verbos, pero los nombres de funciones y métodos deben ser verbos, ya que son ellos los que realizan esa acción que se describe.

Esta función nos está mostrando/pintando un texto, así que es lo que debemos poner en el nombre.

```
    // 1.4 Nombres de clases, métodos y funciones
    PintarEjemplo();

}

public static void PintarEjemplo() {
    System.out.println(x: "holaa!");
}
```

1.5 Nombres por conceptos

No necesitamos nombres diferentes en cuanto a contexto en funciones distintas si ambas van a hacer la misma función.

Por ejemplo, ya que en el 1.4 hemos usado “Pintar”, ahora vamos a hacer una función con ese verbo en vez de “Escribir” o “Mostrar”.

```
    // 1.5 Nombres por conceptos
    PintarFuncionConcepto();

}

public static void PintarEjemplo() {
    System.out.println(x: "holaa!");
}

public static void PintarFuncionConcepto() {
    System.out.println(x: "hola de nuevoo!");
}
```

Bloque 2 : Funciones

2.1 Funciones pequeñas

Una función debería hacer una sola cosa, pero eso lo trataremos más adelante en el punto 2.2.

Deben ser pequeñas para que sean más fáciles de modificar, leer, etc. Lógicamente no es lo mismo leer una función con un System Out que con 60 líneas.

```
public static void PintarEjemplo() {
    System.out.println(x: "holaa!");
}

public static void PintarFuncionConcepto() {
    System.out.println(x: "hola de nuevoo!");
}

public static void PintarFuncionLarga() {
    System.out.println(x: "hola de nuevoo!");
    System.out.println(x: ":o");
    System.out.println(x: "tampoco es muy larga");
    System.out.println(x: "pero se entiende");
    System.out.println(x: "que tiene que ser");
    System.out.println(x: "chiquita");
    System.out.println(x: ":>");
}
```

2.2 Funciones con un propósito

Como comenté antes, una función debe hacer solo una tarea, si hace más de una, tendríamos que separar cada tarea “extra” en una función diferente.

Si tenemos una función que se llama “PintarMenu” que nos muestra un menú y lee una opción, ya no es fiel al nombre, deberíamos pasar la lectura de la opción a “LeerOpcion”.

```
public static void PintarMenu() {

    int opcion = 0;

    System.out.println(x: "1. Primera opción");
    System.out.println(x: "2. Segunda opción");
    System.out.println(x: "3. Tercera opción");
    System.out.println(x: "4. Cuarta opción");
    // separar de función
    System.out.println(x: "Elige una opción:");
    opcion = sc.nextInt();


}
```

2.3 Funciones con menos switches / when

Un switch podemos “limpiarlo” si en cada opción usamos una función específica, pero un when es más probable que termine haciendo más de una cosa, deberíamos buscar una solución más eficaz a este condicional.

2.4 Funciones con pocos argumentos

Una función con muchos argumentos va a ser más complicada de entender, ya que tendríamos que testear y leer todas las combinaciones posibles para saber cuál sería el resultado, una función no debería tener más de 3 argumentos.

2.5 Funciones sin “flag arguments”

Esto se refiere a funciones con booleanos, si una función recibe algo verdadero, hará una cosa y si es falso, hará otra. Esto ya “rompe” el punto 2.2, está haciendo más de una acción, además es complicado saber qué va a hacer si luego cambiamos los booleanos de valores.

2.6 Funciones sin “side effects”

Un código tiene que mostrarnos que está haciendo, no debería tener funciones colaterales que no sepamos interpretar. De nuevo, esto nos lleva al punto 2.2 otra vez, una función sólo debería hacer una cosa a la vez.

2.7 Sin repeticiones

Esta es la función básica de un método, eliminar código repetido. No necesitamos poner una línea o un bloque de código cada vez que queramos hacer algo, podemos almacenarlo en funciones y llamarlas en nuestro código.

```
    // 2.7 Sin repeticiones
    System.out.println(x: "Aquí escribo algo");
    System.out.println(x: "Aquí algo más");
    System.out.println(x: "Aquí otra cosa");
    // lo pasamos a una función
    PintarRepeticion();

}

public static void PintarRepeticion() {
    System.out.println(x: "Aquí escribo algo");
    System.out.println(x: "Aquí algo más");
    System.out.println(x: "Aquí otra cosa");
}
```

Bloque 3 : Comentarios

3.1 Comentarios obsoletos

Aquí nos referimos a comentarios que hicimos cuando estábamos escribiendo el código, puede que hayamos modificado el código y no el comentario, así que el comentario "mentiría" sobre lo que está haciendo esa línea o bloque.

```
// 3.1 Comentarios obsoletos
// Variable que muestra días de la semana
int variable_inicial = 7;
// Variable que muestra días de la semana
int variable_nueva = 30;
```

3.2 Código autoexplicativo

Un comentario no es realmente necesario si nuestro código está limpio y bien nombrado, nos está repitiendo la función de nuestro programa así que podemos omitirlos, yo no soy el mejor ejemplo de esto porque casi siempre comento todo para saber bien qué estoy haciendo :P

```
// 3.2 Código autoexplicativo
// Esta variable es para las horas de un día
int horas_dia = 24;
```

3.3 Comentarios necesarios

Estos comentarios a veces vienen bien si estamos haciendo un programa es una forma en la que nosotros lo entendemos pero puede que otros no porque no encontramos una solución mejor o más óptima, en este caso viene bien explicar un poco qué estamos haciendo para que no sea un lío.

3.3 Comentarios explicativos

Lo único que hace que sepamos cómo se funciona un bloque de código, es el código en si, ya que los comentarios sólo nos explican qué se está haciendo, no cómo. Estos comentarios deberían enseñarnos porqué hemos tomado la decisión de hacer ese bloque de código de cierta manera.