BLOQUE 1: NOMBRES

1. Usa nombres con significado

Cuando creas variables, funciones, clases… es importante utilizar nombres que aporten valor a la comunicación de lo que tu código hace.

Si al leer el nombre de un elemento no sabes para qué sirve, o necesitas poner un comentario para explicar lo que hace, entonces habrás fallado en este punto.

Mal:

```
public class significado {

    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        int z = x + y;

        System.out.println(z);
    }
}
```

Bien:

```
public class significado {

    public static void main(String[] args) {
        int num1 = 10;
        int num2 = 20;
        // Suma del primer y segundo número
        int suma = num1 + num2;

        System.out.println(suma);
    }
}
```

2. Usa nombres fáciles de pronunciar

Los seres humanos nos entendemos hablando, e instintivamente pronunciamos en nuestra cabeza lo que leemos, por lo que hay que ponérselo fácil al cerebro.

Además, imagínate debatiendo con tu compañero/a sobre un componente de la App que es imposible de pronunciar. La escena sería bastante cómica, pero poco productiva.

Mal:

```java
public class pronunciable {
    public static void main(String[] args) {
        int llll = 10;
        int aaaaa = 20;

        // Suma llll y aaaaa
        int zzzz = llll + aaaaa;

        System.out.println(zzzz);
    }
}
```

Bien:

```java
public class pronunciable {

    public static void main(String[] args) {
        int diez = 10;
        int veinte = 20;

        // Suma diez y veinte
        int suma = diez + veinte;

        System.out.println(suma);
    }
}
```

3. Usa nombres que puedan buscarse

Por un lado, si tienes constantes, extráelas para darles un nombre con significado.

Lo mismo para variables con nombres muy cortos. Si son variables locales, está bien usar nombres más acotados, pero si el ámbito de la variable es mayor, necesitamos explicarlo mejor.

Mal:

```java
public class constantes {

    public static void main(String[] args) {
        int x = 10;
        int y = 20;

        // Suma x y y
        int z = x + y;

        System.out.println(z);
    }
```

```
}
```

Bien:

```
public class constantes {

    public static void main(String[] args) {
        final int NUMERO_UNO = 10;
        final int NUMERO_DOS = 20;

        // Suma NUMERO_UNO y NUMERO_DOS
        int suma = NUMERO_UNO + NUMERO_DOS;

        System.out.println(suma);
    }
}
```

4. Nombres de clases y métodos (o funciones)

Los nombres de clases deben ser nombres, mientras que las funciones o métodos deben ser verbos.

Mientras que las primeras representan un elemento que en mayor o menor medida refleja un objeto del mundo real, las segundas son acciones que esos objetos realizan.

Mal:

```
public class clasemetodo {

    public static void main(String[] args) {
        ClaseClase clase = new ClaseClase();
        clase.metodo1();
        clase.metodo2();
    }

    public static class ClaseClase {
        public void metodo1() {
        }

        public void metodo2() {
        }
    }


}
```

Bien:

```
public class clasemetodo {

    public static void main(String[] args) {
        Persona Paco = new Persona();
        Paco.calcularEdad();
        Paco.saludar();
    }

    public static class Persona {
        public int edad;

        public void calcularEdad() {
        }

        public void saludar() {
        }
    }
}
```

5. Elige una sola palabra por concepto

Si realiza una tarea similar o tiene una responsabilidad parecida en tu código, debería usar el mismo concepto.

No llames a unas clases Manager, a otras Controller y a otras Driver si todas ellas solucionan un problema similar.

Mal:

```
public class drivercontrollermanager {
    public static void main(String[] args) {
        PersonaManager Tomas = new PersonaManager();
        Tomas.crearPersona();
        Tomas.actualizarPersona();
        Tomas.eliminarPersona();

        ProductoController Jacobo = new ProductoController();
        Jacobo.crearProducto();
        Jacobo.actualizarProducto();
        Jacobo.eliminarProducto();

        OrdenDriver Pedro = new OrdenDriver();
        Pedro.crearOrden();
        Pedro.actualizarOrden();
        Pedro.eliminarOrden();
    }
```

```
    public static class PersonaManager {
        public void crearPersona() {
        }

        public void actualizarPersona() {
        }

        public void eliminarPersona() {
        }
    }

    public static class ProductoController {
        public void crearProducto() {
        }

        public void actualizarProducto() {
        }

        public void eliminarProducto() {
        }
    }

    public static class OrdenDriver {
        public void crearOrden() {
        }

        public void actualizarOrden() {
        }

        public void eliminarOrden() {
        }
    }
}
```

Bien:

```
public class drivercontrollermanager {

    public static void main(String[] args) {
        GestorPersonas Tomas = new GestorPersonas();
        Tomas.crearPersona();
        Tomas.actualizarPersona();
        Tomas.eliminarPersona();

        GestorProductos Jacobo = new GestorProductos();
```

```
        Jacobo.crearProducto();
        Jacobo.actualizarProducto();
        Jacobo.eliminarProducto();

        GestorOrdenes Pedro = new GestorOrdenes();
        Pedro.crearOrden();
        Pedro.actualizarOrden();
        Pedro.eliminarOrden();
    }

    public static class GestorPersonas {
        public void crearPersona() {
        }

        public void actualizarPersona() {
        }

        public void eliminarPersona() {
        }
    }

    public static class GestorProductos {
        public void crearProducto() {
        }

        public void actualizarProducto() {
        }

        public void eliminarProducto() {
        }
    }

    public static class GestorOrdenes {
        public void crearOrden() {
        }

        public void actualizarOrden() {
        }

        public void eliminarOrden() {
        }
    }
}
```

6. Las funciones deben ser pequeñas

Cuanto más largas sean, más posible es que esté incumpliendo el Principio de Responsabilidad Única.

Se vuelve más difíciles de leer, de modificar, de testear…

Otra forma de limitar la longitud es que la indentación sea máximo de 1 ó 2 niveles. Si va a ser más, mejor extraer esa parte a otra función.

En cuanto a la longitud de las líneas, no uses más de 100-120 caracteres. Muchos IDEs permiten configurar una línea vertical para ayudarte con ello.

Mal:

```
public static void calcularSueldo(Empleado empleado) {
        double sueldoBase = empleado.getSueldoBase();
        double plus = empleado.getAntiguedad() * 100;
        double sueldoBruto = sueldoBase + plus;
        double impuestos = sueldoBruto * 0.21;
        double sueldoNeto = sueldoBruto - impuestos;

        System.out.println("El sueldo de " + empleado.getNombre() + " es
" + sueldoNeto);
    }
```

Bien:

```
public static void calcularSueldo(Empleado empleado) {
        double sueldoNeto = calcularSueldoNeto(empleado);

        System.out.println("El sueldo de " + empleado.getNombre() + " es
" + sueldoNeto);
    }

    private static double calcularSueldoNeto(Empleado empleado) {
        double sueldoBase = empleado.getSueldoBase();
        double plus = empleado.getAntiguedad() * 100;
        double sueldoBruto = sueldoBase + plus;
        double impuestos = sueldoBruto * 0.21;

        return sueldoBruto - impuestos;
    }
```

7. Haz una única cosa

Las funciones deberían hacer una única cosa y hacerla bien.

Relacionado con esto, si tu función tiene más de un nivel de abstracción, entonces mejor divídela.

Mal:

```
public static void mostrarInformacion(Persona persona) {
        System.out.println(persona.nombre);
        System.out.println(persona.edad);
        System.out.println(persona.direccion);
    }
```

Bien:

```
public static void mostrarNombre(Persona persona) {
        System.out.println(persona.nombre);
    }

    public static void mostrarEdad(Persona persona) {
        System.out.println(persona.edad);
    }

    public static void mostrarDireccion(Persona persona) {
        System.out.println(persona.direccion);
    }
```

8. No abuses de los switch/when

Es muy difícil que una función con un control de este tipo haga una única cosa.

Si necesitas usarlos, usa polimorfismo para ocultarlo y no tener que repetirlo si es necesario volver a usarlo.

Mal:

```
public static String saludar(String nombre) {
        switch (nombre) {
            case "Juan":
                return "Hola, Juan";
            case "María":
                return "Hola, María";
            default:
                return "Hola, desconocido";
        }
    }
```

Bien:

```
public static String saludar(String nombre) {
        if (nombre.equals("Juan")) {
            return "Hola, Juan";
        } else if (nombre.equals("María")) {
```

```
            return "Hola, María";
        } else {
            return "Hola, desconocido";
        }
    }
```

9. ¿Cuántos argumentos debe tener una función?

Si una función tiene muchos argumentos, es difícil razonar sobre ella, ya que hay muchas combinaciones de argumentos de entrada, y es difícil comprender qué va a ocurrir con cada posible combinación.

Esto llevado al testing es lo mismo: hay muchas posibilidades, y por tanto probar la función con todos sus posibles estados se puede volver inmanejable.

Mal:

```
public static String sumarYMultiplicar(int numero1, int numero2, int
numero3, int numero4, int numero5) {
        int suma = numero1 + numero2 + numero3 + numero4;
        int producto = suma * numero5;
        return String.valueOf(producto);
    }
```

Bien:

```
public static String calcular(int[] numeros) {
        int suma = 0;
        int producto = 1;
        for (int numero : numeros) {
            suma += numero;
            producto *= suma;
        }
        return String.valueOf(producto);
    }
```

10. Evita los “flag arguments”

Normalmente son booleanos que identifican que si el valor viene a false se hace una cosa y si viene a true se hace otra.

Para empezar, porque esto ya significa que la función está haciendo 2 cosas, y no una.

Y para seguir, porque a nivel de comprensión, leer por ejemplo una función que sea render(true) no nos explica nada.

Es mejor tener 2 funciones que no reciban argumento y que su nombre indique exactamente lo que hacen.

Mal:

```
public void render(boolean conTitulo) {
        if (conTitulo) {
            System.out.println("Título:");
        }

        System.out.println("Contenido");
    }
```

Bien:

```
public void renderTitulo() {
        System.out.println("Título:");
    }

    public void renderContenido() {
        System.out.println("Contenido");
    }
```

11. No generes “Side Effects” o efectos colaterales

Si una función pide algo, pero, además, hace algo que no está representado en el nombre de la función, nos está ocultando información.

Imagina que tienes una función que pide un listado de elementos, pero también los guarda en una base de datos, o modifica esos datos de alguna forma.

Pueden ocurrir cosas inesperadas de las que la función no nos está informando.

Por lo general, una función se debe encargar de hacer algo o devolver algo, pero nunca ambas cosas.

Mal:

```
public class doble {
    public List<String> getListado() {
    // Obtiene el listado de elementos
    List<String> listado = ...;

    // Guarda el listado en una base de datos
    guardarListado(listado);

    return listado;
}
```

Bien:

```
public List<String> getListado() {
    // Obtiene el listado de elementos
    return ...;
```

```
}

public void guardarListado(List<String> listado) {
    // Guarda el listado en una base de datos
}
```

12. Don’t Repeat Yourself (No te repitas)

Un principio muy conocido dentro de la programación. Repetir código implica que si la lógica de ese código cambia, nos tenemos que acordar de modificarla en todas las partes donde se encuentre la duplicación.

Extrae el código en una función y reutilízalo donde corresponda.

Mal:

```
public void imprimirListado(List<String> listado) {
        for (String elemento : listado) {
            System.out.println(elemento);
        }
    }

    public void imprimirListadoConEspacios(List<String> listado) {
        for (String elemento : listado) {
            System.out.println(" " + elemento);
        }
    }
```

Bien:

```
public void imprimirListado(List<String> listado) {
        imprimir(listado, false);
    }

    public void imprimirListadoConEspacios(List<String> listado) {
        imprimir(listado, true);
    }

    private void imprimir(List<String> listado, boolean conEspacio) {
        for (String elemento : listado) {
            if (conEspacio) {
                System.out.println(" " + elemento);
            } else {
                System.out.println(elemento);
            }
        }
    }
```

BLOQUE 3: COMENTARIOS

13. Los comentarios mienten

El gran problema que tienen los comentarios es que no compilan, por lo que si el código asociado se modifica y el comentario queda obsoleto, nadie se enterará.

A partir de ese momento, efectivamente el comentario miente, y por tanto quien lo lea tendrá un conocimiento equivocado del código. Es casi mejor no entender el código que entenderlo mal.

Mal:

```
public void calcularSuma(int a, int b) {
        // Calcula la suma de dos números
        int suma = a + b;
        return suma;
    }
```

Bien:

```
public int sumar(int a, int b) {
        return a + b;
    }
```

14. Usa código autoexplicativo

En vez de crear un comentario que explique lo que el código hace, ¿por qué no escribes el código de tal forma que él mismo explique lo que hace?

Muchas veces es tan sencillo como dar nombres más descriptivos, y crear funciones con buenos nombres que encapsulen detalles del código.

Siempre que necesitemos un comentario para explicar un código confuso, plantéate si no puedes mejor reescribir el código para que se entienda.

Mal:

```
public void calcularSuma(int a, int b) {
        // Calcula la suma de dos números
        int suma = a + b;
        return suma;
    }
```

Bien:

```
public int sumar(int a, int b) {
        return a + b;
    }
```

15. A veces los comentarios son necesarios

Para algunos casos sí tiene sentido escribir comentarios.

Si por ejemplo necesitas escribir un código un poco rebuscado que no eres capaz de resolver de otra forma, o te encuentras trabajando con una API o framework que obliga a hacer algo de una manera muy especial, o tienes que resolver algún problema externo a tu código con un hack, estas son situaciones razonables.

Ejemplo Python:

```python
def calcular_suma(a, b):
    # Calcula la suma de dos números
    return a + b

# Usar un comentario para explicar el código rebuscado

def calcular_suma_rebuscada(a, b):
    # Calcula la suma de dos números utilizando un algoritmo rebuscado
    return a + b if a > b else b + a
```

16. Los comentarios dicen qué hace el código, no cómo lo hace

A modo de resumen, si tienes un comentario que dice qué pasos se están dando en el código, ese comentario no debería existir.

Un comentario debería explicar por qué se ha tomado cierta decisión que pueda generar cierta controversia, o que a simple vista no sea evidente.

Mal:

```java
public void calcularSuma(int a, int b) {
        // Calcula la suma de dos números
        int suma = a + b;

        // Si la suma es mayor que 100, se imprime un mensaje de error
        if (suma > 100) {
            System.out.println("La suma es mayor que 100");
        }
    }
```

Bien:

```java
public void calcularSuma(int a, int b) {
        // La suma de dos números no debe ser mayor que 100
        int suma = a + b;
        if (suma > 100) {
            throw new IllegalArgumentException("La suma es mayor que
100");
        }
    }
```