En la materia nos enfocamos en construir programadores “Los mejores programadores” para que puedan afrontar la vida profesional y cuenten con herramientas para la mejora continua.

Nuestro objetivo es que sepan cómo escribir código correcto y como transformar código incorrecto en correcto.

Esta filosofía de trabajo cuenta con un conjunto de normas que pretender hacer que el código sea más fácil de leer, mantener, acrecentar y menos expuesto a errores.

Si bien nos enfocamos en el lenguaje Java los conceptos que desarrollaremos se aplican a prácticamente todos los lenguajes de programación.

**Convenciones de nomenclatura generales**

1. **Los nombres de paquetes deberían estar totalmente en minúsculas**.

2. **Los nombres que representan Clases (Objetos) deben ser sustantivos y deben escribirse con mayúsculas y minúsculas iniciando con mayúscula. Idealmente una sola palabra.**

3. **Los nombres de variables deben utilizar mayúsculas y minúsculas iniciando con minúscula.**

4. **Los nombres que representan métodos deben ser verbos y escribirse con mayúsculas y minúsculas iniciando con minúscula.**

**Clases**

1. **Las clases deben declararse en archivos individuales con un nombre de archivo igual al nombre de la clase.**

2. **Las declaraciones de clase deberían organizarse.**

Esto debería hacerse de la siguiente manera:  
1.1. Documentación de la Clase.  
1.2. Sentencia Class.  
1.3. Variables de clase   
1.4. Constructores.  
1.5. Métodos (sin orden específico).  
Se trata de reducir la complejidad haciendo predecible la ubicación de cada elemento de clase.

**Tipos de datos**

1. **Las conversiones de tipos deben hacerse siempre explícitas.**Nunca caiga en conversiones implícitas.  
floatValue = (float) intValue; // NOT: floatValue = intValue \* 1.00;  
El programador debe indicar que está conciente de los diferentes tipos implicados y que la mezcla es intencional.

**Variables**

1. **Las variables deberían inicializarse donde se declaran y deberían declararse en el ámbito más pequeño posible.**

Esto asegura que las variables son válidas en todo momento. A veces es imposible inicializar una variable con un valor válido donde se le declara. En esos casos debería dejarse sin inicializar en vez de darle un valor sin sentido.

2. **Las variables nunca deben tener significado dual.** Esto mejora la legibilidad asegurando que todos los conceptos se representan de manera única. Reduce las posibilidades de errores producto de la dualidad.

3. **Las variables de clase nunca se deberían declarar public.**

El concepto de ocultamiento de información y encapsulamiento de Java es violado por las variables públicas. Use las variables privadas y acceda en vez de ello a las funciones.

4. **Las variables relacionadas del mismo tipo se pueden declarar en una sentencia común.**

Las variables no relacionadas no se deberían declarar en la misma sentencia.  
float x, y, z;  
float revenueJanuary, revenueFebrury, revenueMarch;  
El requerimiento común de tener declaraciones en líneas separadas no es útil siempre. Esto mejora la legibilidad al agrupar variables. Note que sin embargo que cuando sea posible, las variables deberían inicializarse donde se declaren.

5. **Los arreglos deben declararse con corchetes junto al tipo**.  
double[] vertex; // NOT: double vertex[];  
int[] count; // NOT: int count[];  
public double[] computeVertex()  
Esto tiene doble motivo. Primero, la característica de ser arreglo es de la clase, no de la variable. Segundo, cuando se retorna un arreglo en un método, no es posible tener los corchetes en otro lugar que no sea con el tipo.

**Repeticiones**

1. **Sólo las sentencias de control de flujo deben incluirse en la construcción del *for()***.  
sum = 0;  
for (i = 0; i < 100; i++)   
 sum += value[i];  
 NOT: for (i=0, sum = 0; i < 100; i++)  
 sum += value[i];  
Esto incrementa la legibilidad y mantenibilidad. Hace una clara distinción que *controla* y que está *contenida* en la repetición.

2. **Las variables de repetición deberían inicializarse inmediatamente antes de la repetición**.

3. **Debería evitarse el uso de ciclos do…while**.

Cualquier sentencia que se puede escribir como un *do…while* se puede igualmente escribir como una repetición *while* o una repetición *for*. La complejidad se reduce si se utiliza la menor cantidad de construcciones. La razón es legibilidad. Un ciclo con la parte condicional el final es más difícil de leer que uno con el condicional arriba.

4. **Se debería evitar el uso de *break* y *continue* en las repeticiones.**Estas sentencias sólo se deberían usar si demuestran brindar más legibilidad que sus contrapartes estructuradas.

**Comentarios**

1. **Código confuso.**

El código confuso no debería comentarse sino reescribirse. En general, el uso de los comentarios debería minimizarse haciendo el código auto documentado con elecciones apropiadas de nombres y estructuras lógicas explícitas.

2**. Unidades de código pequeñas**Las clases y métodos cortos simplifican la comprensión del código y lo hacen más mantenible  
Las funciones o clases que son muy largos se vuelven casi imposibles de entender si no estudiamos lo que hacen línea a línea, y eso alargará muchísimo el tiempo de comprensión.  
Por el contrario, si son cortos y, combinado con el punto anterior, les damos mucho significado, la lectura será directa.

3. N**o superar más de un nivel de anidamiento.**

Cada una de las ramas de un if/else por ejemplo, debería estar formado por una única línea que llame a una función.  
La idea detrás de esto es que las bifurcaciones de código se vuelven difíciles de seguir. Si lo extraes a una función, le puedes dar un nombre con significado que explique lo que se hace en ese posible camino.

4**. Las unidades de código deben hacer una única cosa**Las estructuras de código demasiado grandes probablemente estén haciendo más de una cosa.

5**. No repetir código ni dejar código que no se use**Repetir código es una acción bastante peligrosa que nos causará problemas tarde o temprano.