



que es SKU en POO

#### ◆ Ver razonamiento ▾

¡Buena pregunta! Es el punto exacto donde el mundo real de los negocios se topa con el código.

En la Programación Orientada a Objetos (POO), un **SKU (Stock Keeping Unit)** no es un **concepto** de POO (como la herencia o la encapsulación).

Un SKU es un **dato**.

Especificamente, es casi siempre un **atributo** (o propiedad) de una clase que representa un producto o un artículo en inventario.

#### ¿Qué es el SKU?

En logística y ventas, un SKU (Unidad de Mantenimiento de Stock) es un código **único** que se usa para identificar un producto **específico**.

Por ejemplo, una "Camisa" no es suficiente. Una "Camisa azul, talla M" tiene un SKU, y una "Camisa azul, talla L" tiene otro SKU. Es el identificador más granular.

#### El SKU en POO

En POO, tú modelas el mundo real usando **clases**. Si estás haciendo un sistema de inventario, tendrías una clase **Producto** (o **Articulo**, **Item**, etc.).

El SKU sería una de las propiedades de esa clase, generalmente de tipo **String**, porque es el identificador único de ese objeto en el mundo real.

Aquí tienes un ejemplo simple en Java (pero la lógica es la misma en Python, C#, PHP, etc.):

Aquí tienes un ejemplo simple en Java (pero la lógica es la misma en Python, C#, PHP, etc.):

Java



```
/* * Esta es la "plantilla" para todos los productos.
 */
public class Producto {

    // --- Atributos ---

    private String sku;          // <-- ¡Ahí está! El identificador único.
    private String nombre;
    private double precio;
    private int stock;

    // Constructor para crear un nuevo objeto Producto
    public Producto(String sku, String nombre, double precio, int stock) {
        this.sku = sku;
        this.nombre = nombre;
        this.precio = precio;
        this.stock = stock;
    }

    // --- Métodos ---

    public String getSKU() {
        return this.sku;
    }

    public void agregarStock(int cantidad) {
        this.stock += cantidad;
    }

    // ... otros getters y setters ...
}
```

Y así lo usarías:

Java



```
// Creamos "instancias" u "objetos" de la clase Producto  
  
// SKU único para la camisa mediana  
Producto camisaMediana = new Producto("CAM-AZUL-M", "Camisa de Lino Azul (M)", 49.99);  
  
// SKU único para la camisa grande  
Producto camisaGrande = new Producto("CAM-AZUL-L", "Camisa de Lino Azul (L)", 49.99);
```

**En resumen:** El SKU no es POO. Es el **dato clave** que usas *dentro* de la POO (como atributo `sku`) para conectar tu objeto de software con un artículo físico y único en un almacén.

---

¿Te gustaría ver cómo se podría modelar la diferencia entre un "Producto" (la idea general, como "Camisa de Lino") y sus "Variantes" (los SKUs específicos por talla/color)?

↶ ↷ ⏪ ⏵ :

cual es la sintaxis de un arreglo



Ver razonamiento ▾

¡Buena pregunta! La sintaxis de un arreglo (o array) **depende completamente del lenguaje de programación** que estés usando.

Dado que has estado practicando Programación Orientada a Objetos (OOP) y Java, empezaré con la sintaxis de **Java**, que es más estricta, y luego te mostraré las de Python y JavaScript, que son muy comunes.

### ⌚ Sintaxis en Java

En Java, los arreglos tienen un **tamaño fijo** y deben declarar un **tipo de dato específico** (todos los elementos deben ser del mismo tipo).

**1. Declarar e inicializar (Forma más común):** Se usa cuando ya sabes qué valores tendrá.

Java



```
// tipo[] nombreDelArreglo = {valor1, valor2, valor3};  
  
int[] numeros = {10, 20, 30, 40, 50};  
String[] nombres = {"Ana", "Luis", "Maria"};
```

**2. Declarar y luego inicializar (Con tamaño):** Se usa cuando sabes el tamaño, pero lo llenarás después.

Java

```
// tipo[] nombreDelArreglo = new tipo[tamaño];  
  
// Crea un arreglo para 5 enteros  
int[] puntajes = new int[5];  
// Por defecto, se llena con valores nulos (0 para int, null para String, etc.)  
// [0, 0, 0, 0, 0]
```

**3. Acceder a un elemento:** Se usa el índice numérico, que **siempre empieza en 0**.

Java

```
System.out.println(nombres[0]); // Imprime "Ana" (el primer elemento)  
System.out.println(numero[2]); // Imprime 30 (el tercer elemento)  
  
// Asignar un valor  
puntajes[0] = 100; // Cambia el primer elemento a 100
```

## 💡 Sintaxis en Python

En Python, lo que se usa comúnmente como "arreglo" se llama **lista**. Las listas son **dinámicas** (pueden cambiar de tamaño) y pueden mezclar tipos de datos. La sintaxis es mucho más simple.

### 1. Declarar e inicializar:

Python

```
# Se usan corchetes []
numeros = [10, 20, 30, 40, 50]
nombres = ["Ana", "Luis", "Maria"]
mixto = [1, "Hola", True, 2.5] # Se pueden mezclar tipos
```

## 2. Acceder a un elemento (Igual que en Java):

Python

```
print(nombres[0]) # Imprime "Ana"
```

## 3. Añadir un elemento (Diferencia clave):

Como son dinámicas, puedes añadir o quitar elementos fácilmente.

Python

```
nombres.append("Juan") # Añade "Juan" al final
print(nombres) # Imprime ["Ana", "Luis", "Maria", "Juan"]
```

## Sintaxis en JavaScript

Muy similar a Python. Los arreglos en JavaScript también son **dinámicos** y pueden mezclar tipos.

### 1. Declarar e inicializar:

JavaScript

```
// Se usan corchetes [], generalmente con 'let' o 'const'  
let numeros = [10, 20, 30, 40, 50];  
let nombres = ["Ana", "Luis", "Maria"];  
let mixto = [1, "Hola", true, 2.5];
```

## 2. Acceder a un elemento (Igual que en Java y Python):

JavaScript

```
console.log(nombres[0]); // Imprime "Ana"
```

## 3. Añadir un elemento (Similar a Python):

JavaScript

```
nombres.push("Juan"); // Añade "Juan" al final  
console.log(nombres); // Muestra ["Ana", "Luis", "Maria", "Juan"]
```

## Resumen de la Sintaxis

| Característica | Java                         | Python (Listas) | JavaScript        |
|----------------|------------------------------|-----------------|-------------------|
| Declaración    | int[] arr = {1, 2};          | arr = [1, 2]    | let arr = [1, 2]; |
| Tamaño Fijo    | Sí                           | No (Dinámico)   | No (Dinámico)     |
| Tipos Mixtos   | No (Tipo único)              | Sí              | Sí                |
| Acceso         | arr[0]                       | arr[0]          | arr[0]            |
| Añadir         | No (Hay que crear uno nuevo) | arr.append(3)   | arr.push(3)       |

 Exportar a Hojas de cálculo



y si quiero una lista?

Excelente punto. Cuando en programación moderna (especialmente en Python, Java, C#) se habla de "listas", generalmente nos referimos a una estructura más flexible y poderosa que un arreglo básico.

Si vienes de Java, la diferencia es clave:

- **Arreglo** (`int[]`): Tamaño fijo. Una vez creado, no puedes añadir o quitar elementos; solo puedes cambiar los valores que ya existen.
- **Lista** (`List`): Tamaño dinámico. Puedes añadir (`add`), quitar (`remove`) y modificar elementos libremente, y la lista ajustará su tamaño automáticamente.

La "lista" es casi siempre más cómoda de usar en la práctica, a menos que tengas requisitos de rendimiento muy específicos.

Aquí tienes la sintaxis de las **Listas** en los mismos tres lenguajes:

### ⌚ Sintaxis en Java (Usando `ArrayList`)

En Java, "Lista" (o `List`) es una **interfaz**. La clase más común que se usa para implementarla es `ArrayList`. Necesitas importar la clase para usarla.

#### 1. Declarar e inicializar:

Java



```
// Importar la clase al inicio de tu archivo
import java.util.ArrayList;
import java.util.List; // Opcional, pero buena práctica

// Crear una lista vacía
// Se usan "genéricos" < > para definir el tipo de dato
List<String> nombres = new ArrayList<>();
List<Integer> numeros = new ArrayList<>(); // ¡Ojo! Se usa 'Integer', no 'int'
```

```
// Crear lista con valores iniciales  
List<String> ciudades = new ArrayList<>(List.of("Tokio", "París", "Roma"));
```

**2. Añadir un elemento:** Se usa el método `.add()`.

Java

```
nombrres.add("Ana"); // [Ana]  
nombrres.add("Luis"); // [Ana, Luis]  
nombrres.add("Maria"); // [Ana, Luis, Maria]
```

**3. Acceder a un elemento:** Se usa el método `.get(indice)`.

Java

```
System.out.println(nombrres.get(0)); // Imprime "Ana"
```

**4. Quitar un elemento:** Se usa el método `.remove()`.

Java

```
nombrres.remove(1); // Quita el elemento en el índice 1 ("Luis")  
// La lista ahora es: [Ana, María]
```

**5. Obtener el tamaño:** Se usa el método `.size()`.

Java

```
System.out.println(nombres.size()); // Imprime 2
```

### 💡 Sintaxis en Python (Las Listas son el estándar)

Como mencioné antes, en Python, lo que llamas "arreglo" es en realidad una "lista" por defecto. La sintaxis es la misma que te mostré:

#### 1. Declarar e inicializar:

Python

```
# Los corchetes [] crean una lista
nombres = [] # Lista vacía
nombres = ["Ana", "Luis", "Maria"] # Lista con valores
```

#### 2. Añadir un elemento: Se usa el método `.append()`.

Python

```
nombres.append("Juan") # [Ana, Luis, Maria, Juan]
```

#### 3. Acceder a un elemento: Se usa `[indice]`.

Python

```
print(nombres[0]) # Imprime "Ana"
```

Python

```
print(nombres[0]) # Imprime "Ana"
```

**4. Quitar un elemento:** Hay varias formas, la más común es `.remove()` (por valor) o `del` (por índice).

Python

```
nombres.remove("Luis") # Busca "Luis" y lo borra  
# La lista ahora es: [Ana, Maria, Juan]
```

**5. Obtener el tamaño:** Se usa la función `len()`.

Python

```
print(len(nombres)) # Imprime 3
```

### Sintaxis en JavaScript (Los Arreglos son Listas)

En JavaScript, los "Arreglos" (`Array`) que vimos antes ya funcionan como listas dinámicas. No hay una sintaxis separada para una "lista" vs. un "arreglo" como en Java.

La sintaxis del arreglo que te mostré (con `.push()` para añadir) es la que se usa para listas dinámicas.

JavaScript



```
let nombres = ["Ana", "Luis", "Maria"];  
  
// 1. Añadir  
nombres.push("Juan"); // [Ana, Luis, Maria, Juan]  
  
// 2. Acceder  
console.log(nombres[0]); // "Ana"  
  
// 3. Quitar (El método .splice() es muy común)  
// .splice(indice_inicio, cantidad_a_borrar)  
nombres.splice(1, 1); // Empezando en índice 1, borra 1 elemento ("Luis")  
  
// La lista ahora es: [Ana, Maria, Juan]  
  
// 4. Obtener el tamaño  
console.log(nombres.length); // 3
```

### Resumen: ¿Arreglo o Lista?

| Lenguaje   | "Arreglo" (Tamaño Fijo)                | "Lista" (Tamaño Dinámico)         |
|------------|--|-----------------------------------|
| Java       | String[] miArreglo                     | ArrayList<String> miLista         |
| Python     | (Menos común, array del módulo array ) | mi_lista = [] (Es el estándar)    |
| JavaScript | (No existe nativamente)                | let miLista = [] (Es el estándar) |

Exportar a Hojas de cálculo



**En resumen:** Si estás en Java, usa `ArrayList` si quieres una lista dinámica. Si estás en Python o JavaScript, ¡lo que llamas "arreglo" (`[]`) ya es una lista dinámica!

Luego como puedo imprimir los datos de mi lista asignados a un ID

Ver razonamiento ▾

¡Excelente pregunta! Este es un punto clave en la programación y es justo la diferencia entre una estructura simple y una estructura más avanzada.

La respuesta corta es: Una **Lista** o **Arreglo** simple no usa "IDs" (como el SKU "CAM-AZUL-M"), usa **índices numéricos** (0, 1, 2, 3...).

Si quieras obtener datos usando un "ID" (que generalmente es un texto), no estás buscando una **Lista**, estás buscando un **Mapa** (o **Diccionario**).

Vamos a ver las dos formas de hacer esto.

---

**Contexto:** Nuestra Clase **Producto**

Recordemos la clase **Producto** de la primera pregunta. La usaremos en todos los ejemplos.

Java



```
// Clase de ejemplo
class Producto {
    String sku;      // Este es nuestro "ID"
    String nombre;

    public Producto(String sku, String nombre) {
        this.sku = sku;
        this.nombre = nombre;
    }
}
```

```
    public String getSKU() { return this.sku; }
    public String getNombre() { return this.nombre; }
}
```

### Solución 1 (La mejor): Usar un Mapa (o HashMap / Diccionario )

Esta es la estructura de datos *diseñada* para lo que pides. Un Mapa guarda pares de Clave-Valor. Es como un diccionario del mundo real: la "Clave" es la palabra que buscas (tu ID), y el "Valor" es la definición (tu objeto).

Es la forma más rápida y eficiente de obtener algo por su ID.

#### Java (Usando HashMap )

```
Java
```

```
import java.util.HashMap;
import java.util.Map;

// 1. Crear los productos
Producto camisaM = new Producto("CAM-AZUL-M", "Camisa Azul Mediana");
Producto camisaL = new Producto("CAM-AZUL-L", "Camisa Azul Grande");

// 2. Crear el Mapa (Clave: String, Valor: Producto)
Map<String, Producto> inventario = new HashMap<>();

// 3. Asignar los productos al Mapa usando su SKU (ID)
inventario.put(camisaM.getSKU(), camisaM); // "CAM-AZUL-M" -> (Objeto Producto M)
inventario.put(camisaL.getSKU(), camisaL); // "CAM-AZUL-L" -> (Objeto Producto L)
```

```
// --- 4. IMPRIMIR DATOS USANDO EL ID ---
String idBuscado = "CAM-AZUL-M";
Producto productoEncontrado = inventario.get(idBuscado); // ¡La magia está aquí!

if (productoEncontrado != null) {
    System.out.println("Encontrado: " + productoEncontrado.getNombre());
} else {
    System.out.println("Producto no encontrado.");
}
```

**Resultado:** Encontrado: Camisa Azul Mediana

#### 🐍 Python (Usando dict)

Python

```
# (Asumiendo que la clase Producto existe en Python)
camisaM = Producto("CAM-AZUL-M", "Camisa Azul Mediana")
camisaL = Producto("CAM-AZUL-L", "Camisa Azul Grande")

# 2. Crear el diccionario (Mapa)
inventario = {}

# 3. Asignar usando el SKU (ID) como clave
inventario[camisaM.sku] = camisaM
inventario[camisaL.sku] = camisaL

# --- 4. IMPRIMIR DATOS USANDO EL ID ---
id_buscado = "CAM-AZUL-L"
producto_encontrado = inventario.get(id_buscado) # .get() es más seguro

if producto_encontrado:
    print(f"Encontrado: {producto_encontrado.nombre}")
else:
    print("Producto no encontrado.")
```

**Resultado:** Encontrado: Camisa Azul Grande

### Solución 2 (La literal): Buscar en una `Lista`

Tal vez *realmente* tienes los datos en una `Lista` y lo que quieras es **buscar** dentro de ella. Esto es totalmente válido, pero es más lento que usar un `Mapa` (la computadora tiene que recorrer la lista uno por uno hasta encontrarlo).

#### Java (Recorriendo un `ArrayList`)

Java



```
import java.util.ArrayList;
import java.util.List;

Producto camisaM = new Producto("CAM-AZUL-M", "Camisa Azul Mediana");
Producto camisaL = new Producto("CAM-AZUL-L", "Camisa Azul Grande");

// 1. Crear la LISTA
List<Producto> productos = new ArrayList<>();
productos.add(camisaM);
productos.add(camisaL);

// --- 2. IMPRIMIR DATOS BUSCANDO EL ID ---
String idBuscado = "CAM-AZUL-L";
Producto productoEncontrado = null; // Variable para guardar el resultado

// 3. Recorrer la lista para buscar
for (Producto p : productos) {
    if (p.getSku().equals(idBuscado)) { // ¿El SKU de este producto es el que busco
        productoEncontrado = p;
    }
}
```

```
// 4. Imprimir el resultado
if (productoEncontrado != null) {
    System.out.println("Encontrado en la lista: " + productoEncontrado.getNombre())
} else {
    System.out.println("Producto no encontrado en la lista.");
}
```

**Resultado:** Encontrado en la lista: Camisa Azul Grande

### 🐍 Python (Recorriendo una `list`)

Python



```
camisaM = Producto("CAM-AZUL-M", "Camisa Azul Mediana")
camisaL = Producto("CAM-AZUL-L", "Camisa Azul Grande")

# 1. Crear la LISTA
productos = [camisaM, camisaL]

# --- 2. IMPRIMIR DATOS BUSCANDO EL ID ---
id_buscado = "CAM-AZUL-M"
producto_encontrado = None

# 3. Recorrer la lista para buscar
for p in productos:
    if p.sku == id_buscado:
        producto_encontrado = p
        break

# 4. Imprimir el resultado
if producto_encontrado:
    print(f"Encontrado en la lista: {producto_encontrado.nombre}")
else:
    print("Producto no encontrado en la lista.")
```

**Resultado:** Encontrado en la lista: Comisa Azul Mediana

**Resumen: ¿Cuál usar?**

- **Usa un Mapa (Solución 1):** Si tu tarea principal es "dame el dato para este ID". Es la forma más rápida y limpia.
- **Usa una Lista (Solución 2):** Si tu tarea principal es "guardar una colección de cosas" y ocasionalmente necesitas buscar una por su ID.



La cafetería del campus quiere desarrollar una aplicación para que los estudiantes y profesores puedan realizar sus pedidos directamente desde su celular o computadora. La aplicación debe contar con un módulo de lógica que permita crear pedidos, calcular totales y ver un resumen básico del consumo de cada cliente.

En la cafetería existen distintos productos del menú (ejemplo: empanadas, café americano, pollo a la parmesana, galletas, etc.), cada uno tiene un nombre, un SKU y un precio base.

Cada persona que compra se considera un cliente. El sistema solo necesita guardar nombre del cliente y un identificador (por ejemplo, matrícula o un código de acceso). Un cliente puede hacer muchos pedidos a lo largo del día o de la semana, y la cafetería quiere poder ver todos los pedidos de un cliente específico.

Un pedido está formado por varias líneas de pedido. Cada línea de pedido corresponde a un producto específico del menú y una cantidad. El precio total de cada línea es:

*precio del producto x cantidad*

El total del pedido es la suma de todas sus líneas de pedido.

Si un pedido se elimina, sus líneas ya no tienen sentido: no existen líneas "sueltas" sin pedido. En cambio, un cliente sigue existiendo aunque hoy no tenga pedidos, y un producto del menú existe aunque nadie lo pida.

La cafetería quiere que la aplicación:

1. Registre clientes.
2. Registre productos del menú.
3. Cree un nuevo pedido para un cliente existente, agregando varias líneas de pedido (producto + cantidad).
4. Calcular el total de cada pedido.
5. Mostrar un resumen de todos los pedidos de un cliente, incluyendo el total acumulado gastado por ese cliente.

y como hago un UML de esto?

¡Ah, perfecto! Ese texto es un ejercicio clásico de modelado de POO. Te pide exactamente que traduzcas esos requisitos a una estructura de clases.

Para hacer el UML de esto, no usas un solo diagrama, sino (generalmente) dos:

1. **Diagrama de Casos de Uso:** Para mostrar qué puede hacer el usuario.
2. **Diagrama de Clases:** Para mostrar cómo está construido el sistema (las clases, sus datos y cómo se relacionan).

Basado en tus preguntas anteriores, el **Diagrama de Clases** es el que te interesa para la POO. Es el "plano" de tu código.

---

### **1. Diagrama de Clases (El más importante aquí)**

Este diagrama identifica las "cosas" (clases), sus datos (atributos) y sus acciones (métodos).

Primero, identificamos los "sustantivos" principales en tu texto:

- Cliente
- Producto
- Pedido
- LineaPedido

Así es como se desglosan y se conectan:

#### **Clases y Atributos:**

- Cliente

### Clases y Atributos:

- `Cliente`
  - `identificador: String` (Este es tu ID de cliente, la "matrícula")
  - `nombre: String`
- `Producto`
  - `sku: String` (¡Ahí está nuestro SKU!)
  - `nombre: String`
  - `precioBase: double`
- `Pedido`
  - `fecha: Date` (El texto no lo pide, pero es obvio que lo necesitarás)
  - `total: double` (Este se puede calcular)
  - `calcularTotal(): double` (Este sería un método)
- `LíneaPedido`
  - `cantidad: int`
  - `calcularSubtotal(): double` (Este sería un método: `cantidad * producto.precioBase`)

### Relaciones (La parte clave):

Aquí es donde conectamos todo, usando la cardinalidad (cuántos se conectan):

- `Cliente` y `Pedido` (**Asociación 1 a N**)
  - Un `Cliente` puede tener **muchos** `Pedidos` (1...\*).
  - Un `Pedido` pertenece a **un solo** `Cliente` (1).
  - *En el código:* Tu clase `Cliente` tendría una `List<Pedido>`.

- **LíneaPedido y Producto (Asociación N a 1)**
  - Una `LíneaPedido` se refiere a **un solo** `Producto` (1).
  - Un `Producto` puede estar en **muchas** `LíneasPedido` (\*).
  - *En el código:* Tu clase `LíneaPedido` tendría una variable de tipo `Producto`.

## 2. Diagrama de Casos de Uso (El contexto)

Este es más simple. Muestra lo que los actores pueden hacer.

- **Actor:** `Cliente` (representa a Estudiantes y Profesores)
- **Actor:** `Cafeteria` (representa al administrador del sistema)

Y los "casos de uso" (las acciones, de tu lista 1-5):

- `Cliente` puede:
  - Crear nuevo pedido
  - Ver resumen de mis pedidos
- `Cafeteria` (Admin) puede:
  - Registrar clientes
  - Registrar productos del menú

## Conectando todo (El resumen)

El **Diagrama de Clases** es tu mapa para el código POO. Te dice exactamente qué clases necesitas crear.

El **Diagrama de Casos de Uso** te dice qué *botones y funciones* necesita tu aplicación.

Cuando implementes esto, usarás las estructuras que preguntaste:

1. Para guardar los productos y buscarlos por SKU, usarías un `Map<String, Producto>`.
2. Para guardar los pedidos de un cliente, usarías una `List<Pedido>` dentro de tu clase `Cliente`.

¿Te gustaría ver cómo se verían esas clases y relaciones principales escritas en código Java?

