# **Motivación**

1. Entender mejor los programas.

2. Que los programas sean más fáciles de escribir y mantener

3. Que sea más fácil reutilizar los programas

Existen Conceptos que permiten desentender funcionalidades o naturalezas de datos de un programa en específico. Por ejemplo, los arraylist se usan en muchísimos contextos, pero se pueden modificar para un programa en específico para que funcione mejor: en vez de hacer más líneas de código, podríamos establecer una estructura que permita modificar la implementación de los Arraylist, ya definida dentro de un archivo que se puede invocar. Aquí los arraylist serían una superclase, y para cada funcionalidad se podría modificar esta clase (subclase) a través del concepto de Herencia [Clase Abstracta, Interfaz]. Puede darse, además, que un concepto se derive de otros a la vez (Herencia Multiple), por ejemplo, si quisiéramos aprovechar la portabilidad de los arraylist con la eficacia para encontrar datos de los HashMaps (Como una especie de diccionarios), podríamos hacer una clase que venga de los dos y aprovechar ciertas funcionalidades de cada uno, pero esto en java NO se puede hacer directamente.

**Contrato (**Concepto Clave en la delegación de trabajo ó crecimiento no lineal)**:**

El concepto que media las relaciones entre clases es el de **contrato**, y una forma de definirlos es a través de su visibilidad o encapsulamiento: public, que es el que define los contratos; protected, que es el que permite que subclases accedan a métodos de la clase madre; y private, que solo pueden ser accesibles por las instancias de esta. ¿Por qué es importante asegurar los datos? De chatgpt sabemos:

 **protected**: Permite el acceso a los miembros desde las subclases (ya sea en el mismo paquete o en otro) y también dentro del paquete en el que se define la clase. Este nivel de acceso fomenta la reutilización del código en las subclases, pero mantiene cierta protección frente a accesos externos no deseados.

 **private**: Restringe el acceso a los miembros solo dentro de la propia clase, lo que garantiza el encapsulamiento y la ocultación de detalles internos. Esto es fundamental para evitar el acceso y la modificación indebida del estado de la clase desde fuera de ella, promoviendo la integridad de los datos y la seguridad de la implementación."

El **encapsulamiento** permite ocultar los detalles de implementación de una clase y exponer solo los métodos y propiedades necesarios. Esto significa que puedes **cambiar internamente cómo funciona una clase sin que el código que interactúa con ella tenga que cambiar**.

Captura de pantalla de un celular

El contenido generado por IA puede ser incorrecto.

**Flexibilidad:**

* El **encapsulamiento** hace que el código sea más flexible porque permite que las clases cambien internamente sin afectar a las clases que las usan.
* Puedes agregar o modificar detalles internos sin romper las interfaces públicas que las clases externas utilizan.

**Mantenibilidad:**

* El **encapsulamiento** facilita el mantenimiento porque reduce el impacto de los cambios, lo que significa que puedes actualizar, mejorar o corregir una clase sin preocuparte por el código que depende de ella.
* **Modulariza el sistema**, permitiendo un mantenimiento más sencillo y menos propenso a errores.

En resumen, el **encapsulamiento** contribuye a que el código sea más fácil de **mantener**, **escalar** y **reutilizar** debido a su capacidad para ocultar detalles de implementación y permitir que los cambios se realicen de manera controlada sin afectar el sistema en su totalidad.

Detrás del concepto de **Polimorfismo**:

Debemos saber que

“

El comportamiento de un objeto depende de la implementación de sus métodos, no del tipo de la variable que lo señale.

”

**Beneficios del Polimorfismo**

1. **Código más flexible y reutilizable**: Al usar polimorfismo, puedes escribir **código más genérico** que pueda trabajar con diferentes tipos de objetos sin necesidad de modificar el código para cada tipo específico.

🔹 **Ejemplo**: Un método que acepta una lista de objetos Animal puede trabajar con **Perros**, **Gatos**, **Aves**, etc., sin necesidad de que se reescriba cada vez.

1. **Facilita la extensión del código**:  
   Si deseas agregar un nuevo tipo de animal, solo tienes que crear una nueva subclase de Animal y sobrescribir los métodos necesarios. No necesitas cambiar el código que ya usa la clase Animal, porque el polimorfismo manejará el comportamiento adecuado de cada objeto.
2. **Mayor mantenimiento y menor acoplamiento**:  
   Las clases pueden evolucionar y cambiar de manera independiente. Los cambios en las subclases no afectarán a las clases que las utilizan, lo que reduce el riesgo de **errores** y facilita el mantenimiento.

**Hay dos tipo de Polimorfismo**:

**2. Polimorfismo en tiempo de ejecución (Overriding)**

Este tipo de polimorfismo se refiere a la capacidad de una **subclase de sobrescribir (o anular) un método de la superclase**. A diferencia del **overloading**, el **overriding** permite que un objeto de la subclase **utilice una versión modificada del método de la superclase**. El tipo del objeto real determina qué versión del método se ejecuta en tiempo de ejecución.

**1. Polimorfismo en tiempo de compilación (Overloading)**

Este tipo de polimorfismo ocurre cuando se definen varios métodos con el mismo nombre pero con diferentes **firmas** (diferentes parámetros). Es decir, el compilador decide cuál método llamar según los **tipos y la cantidad de parámetros** pasados.

Detrás de estas estructuras también surgen conveniencias, como las relaciones entre las clases: Cómo interactúan entre sí. El concepto de herencia es solo una de ellas, pero hay más:

**1. Asociación**

* **Definición**: Una clase tiene una referencia a otra.
* **Ejemplo**: Un Estudiante puede estar asociado con un Curso.
* **Tipos**:
  + **Uno a Uno** (1:1): Un Pasaporte pertenece a un único Ciudadano.
  + **Uno a Muchos** (1:N): Un Profesor puede enseñar a varios Estudiantes.
  + **Muchos a Muchos** (N:M): Varios Estudiantes pueden inscribirse en varios Cursos.

**2. Agregación (Has-A)**

* **Definición**: Es un tipo de asociación donde una clase "tiene" otra, pero ambas pueden existir independientemente.
* **Ejemplo**: Un Departamento tiene varios Empleados, pero si el Departamento se elimina, los Empleados pueden seguir existiendo.

**3. Composición**

* **Definición**: Es una relación más fuerte que la agregación, donde una clase "posee" otra y no pueden existir por separado.
* **Ejemplo**: Un Coche tiene un Motor. Si el Coche se destruye, el Motor también deja de existir.

**4. Herencia (Is-A)**

* **Definición**: Una clase (subclase) hereda atributos y métodos de otra (superclase).
* **Ejemplo**: Perro es un Animal (Perro extends Animal).

**5. Implementación (Interface)**

* **Definición**: Una clase implementa una o varias interfaces y proporciona su comportamiento: sabemos que debe tener un comportamiento general y el interface obliga a que siga esos pasos. Esto permite, por ejemplo, si queremos hacer un programa que trabaje sobre una estructura de datos iterable. Al nosotros saber que debe ser iterable, sabemos que tiene una interfaz específica y podremos trabajar sobre ese set de atributos y comportamientos sin necesidad de tenerlo directamente. Esto ayuda mucho a la división de trabajo.
* **Ejemplo**: Pájaro implementa Volador (class Pájaro implements Volador).

**6. Dependencia (Usa)**

* **Definición**: Una clase usa otra temporalmente para una tarea específica.
* **Ejemplo**: Un Cliente usa un CarritoDeCompras para comprar productos.

¿Por qué asociar entre clases?