**POO**

BRUCE INAUGURÓ UNA LIBRERÍA Y NOS QUIERE CONTRATAR

Mister BRUCE sabe que estás aprendiendo desarrollo de software y ha decidido darte una oportunidad para desarrollar su pequeño sistema de librería. Acordaste una junta con Mister BRUCE para hablar sobre los requerimientos del sistema.

En la junta te das cuenta que BRUCE la tiene "re-clara", tanto así que te ha llevado una hoja con los:

**Requerimientos del sistema de librería**

Desarrollar un sistema en el que pueda registrar libros y consultar su información.

**Características de los libros:** autor, título, precio, stock e Id.

**Funciones:** Mostrar información

**Posdata:** Ningún usuario puede modificar los libros, eso lo dejamos para una etapa posterior del sistema

Atentamente y con amor: Mister BRUCE

**Lees los requerimientos y les das otra pasada para comenzar con el:**

**Análisis del sistema de librería**

El objeto principal serán libros. Por lo tanto, se requiere

* Modelar un libro con las propiedades mencionadas en los requerimientos
* No se pueden modificar las propiedades
* Los objetos libro deberán ser capaces de mostrar su información

**Procedimiento:** Definir una clase llamada Libro con las propiedades (autor, título, precio, stock e Id) y el método necesario para mostrar información.

Como desarrolladores generalmente pasaremos por las siguientes etapas:

1. ~~Requerimientos~~
2. ~~Análisis~~
3. Diseño / diagrama del sistema
4. Desarrollo

En un equipo de desarrollo suele haber alguien encargado de recolectar los requerimientos, de hacer el análisis e incluso de diagramar el sistema dejando al desarrollador simplemente la etapa de desarrollo. En ésta introducción hemos cubierto la etapa de requerimientos, de análisis y cubriremos también la de diagramación del sistema, sin embargo, la etapa de desarrollo es la que deberás aprender por medio de práctica.

Aprenderemos cada uno de los pilares de la POO a la par que diagramamos el sistema de librería que quiere Mister BRUCE.

**Abstracción:** el pilar donde comienza la POO.

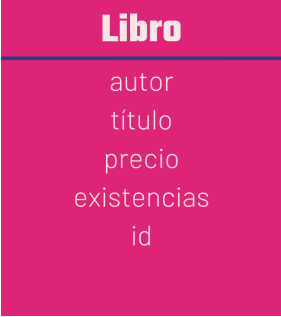
Abstraer es: **eliminar los detalles innecesarios de un objeto para solo enfocarnos en los aspectos que son relevantes para software en desarrollo**.

El sistema en cuestión se basa en objetos reales como lo son los libros (cualquier libro), el cliente fue muy específica con los datos que desea registrar de cada libro.

Sin querer queriendo BRUCE realizó un proceso de abstracción: eliminó los detalles innecesarios de un libro real para enfocarse solo en los elementos relevantes para el sistema.

Podríamos imaginar cuáles fueron estos detalles innecesarios: la portada, el número de hojas, la editorial o el año de publicación. Todo lo anterior lo ignoró y se enfocó en lo que desea: el autor, título, precio, stock e Id.

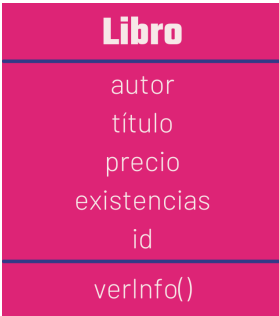
Gracias al análisis que hiciste sabes que necesitas definir una clase que a la postre te permita crear/instanciar libros. Definamos la clase con las características o propiedades requeridas:



Tenemos una clase llamada: Libro y sus propiedades definidas

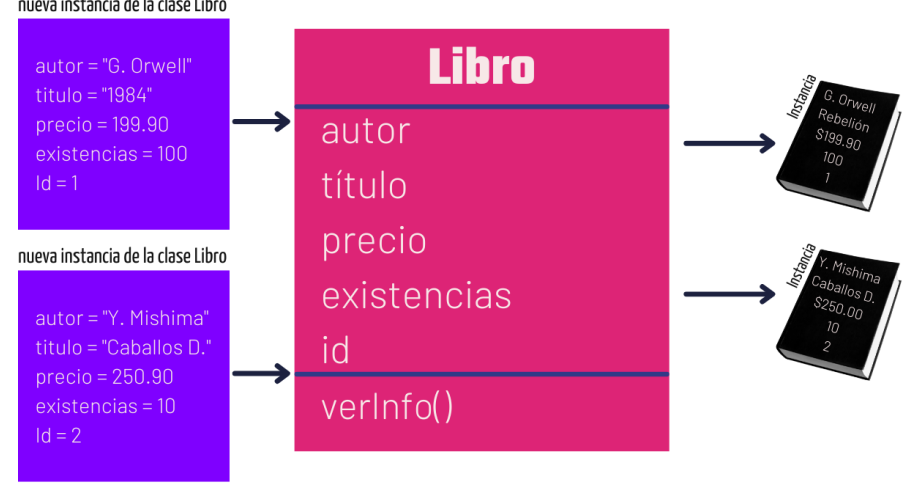
En el análisis se definió que los libros deberían ser capaces de mostrar su información. Esto es una funcionalidad y es el comportamiento que estará asociado a los objetos que se instancien de la clase Libro

Recordemos que el comportamiento de los objetos o las funciones que pueden hacer se les llaman Métodos. Agreguemos este método al diagrama:



El método verInfo() se encargará de mostrar toda la información de un libro. Esta información no es más que las propiedades.

¡Listo! tenemos nuestra clase definida. Ahora podemos instanciar objetos de la clase Libro.



* Para instanciar un objeto primero tendrás que especificar qué quieres instanciar en este caso de la clase Libro.
* Al momento de instanciar será necesario asignar valores para cada propiedad
* Como puedes apreciar, hemos instanciado dos objetos similares (libros) pero con diferentes propiedades que hacen que cada objeto sea diferente del otro a pesar de pertenecer a la misma clase.

Es así como una clase nos permite emplear las mismas herramientas para crear objetos similares (de la misma clase) que lucen diferente.

Hemos definido el proceso de abstracción e incluso definimos nuestra primera clase e instanciamos dos objetos.

¡Genial! Vamos a aprender y a diagramar el siguiente pilar para avanzar en nuestro desarrollo:

**Encapsulamiento:** el pilar del ocultamiento

Encapsular es: **ocultar los detalles que no son relevantes para el exterior de la clase. Consta de agrupar propiedades y métodos que no sean necesarios para el exterior; de manera que el acceso sea restringido desde fuera de la clase**.

Normalmente tendremos propiedades y métodos que sirven solo para procesos internos de nuestros objetos por lo que no es necesario que el resto del programa (lo que está por fuera de la clase) tenga acceso o conocimiento de estas propiedades y métodos

**Modificadores de acceso**

Los lenguajes de programación que aceptan POO suelen contar con palabras reservadas conocidas como modificadores de acceso. Estas palabras reservadas sirven para especificar si queremos que el acceso de una propiedad o método sea público o sea privado y precisamente los dos modificadores de acceso más populares son: public y private.

Público ( public ): se puede acceder desde cualquier parte del código

Private ( private ): solo es accesible desde dentro de la clase que lo define.

BRUCE específico que no desea que nadie pueda modificar la data de sus libros y es normal, como no hay un sistema de usuarios y permisos el dejar abierto al público que se puedan modificar propiedades podría ocasionar que cualquier usuario cambiara el precio de un libro a $0.00.

En realidad, todas las propiedades son datos sensibles, no quisiéramos que cualquiera pudiera modificar el título ni el autor, podría ocasionar un caos y el resto igual. Así que especifiquemos mediante el diagrama que queremos que las propiedades sean privadas:



Todas las propiedades han sido definidas como privadas: esto limita el acceso desde fuera de la clase y ahora no se pueden acceder o modificar desde fuera de la misma.

El método verInfo() ha sido definido como público debido a que solo se limitará a mostrar la información.

* El método está definido dentro de la clase y por lo tanto tiene acceso a las propiedades privadas.
* Esta es la forma en la que el exterior podrá consultar la información de cada libro sin darle opción a modificar ninguna propiedad.

Realicemos un pequeño pseudocódigo para imaginar cómo sería intentar modificar propiedades privadas y ejecutar el método público:

* Creamos una instancia de la clase Libro, el objeto se llama: libro\_1
* La clase recibe valores para cada propiedad.

libro\_1 = nueva instancia Libro ( "autor" = "Geroge O.", "titulo" = "1984", "precio" = 199.90, "stock" = 100, "id" = 1 );

* desde el objeto se intenta modificar una propiedad privada como: precio. El precio que se asignó al objeto cuando se instanció fue de 199.90

libro\_1.precio = 0.00

El resultado sería un error como este: ERROR, no puede modificar una propiedad privada

* Ahora intenta ejecutar el método verInfo() que es público:

libro\_1.mostrarInfo()

Simulamos que el resultado es algo como

"""

Autor: G. Orwell Título: 1984

Precio: $199.90

En stock: 100

Id: 1

Como dato curioso: no todos los lenguajes cuentan con modificadores de acceso formales como r y private . Python, por ejemplo, en su lugar emplea una sintaxis especial para replicar este comportamiento.

Encapsular nos permite tener un sistema más seguro y menos agobiante para el usuario, al cual la mayoría de las veces no le interesa cómo funcionan internamente las cosas, solo quiere poder hacer lo que sabe que puede hacer el objeto en cuestión. Así como una persona promedio que sabe manejar un automóvil sin la necesidad de saber cómo trabaja internamente el motor, incluso si el motor tuviera expuesto todo su cableado y componentes internos podría ser peligroso para el usuario curioso y el automóvil mismo, por eso es mejor encapsular los detalles que no son relevantes.

**Herencia**

Herencia es**: el mecanismo por el cual la POO nos permite reutilizar código sin necesidad de hacer copy-paste. Nos permite relacionar clases de tal forma que una clase hereda propiedades y métodos de otra clase.**

Así de simple como suena, una clase hereda o extiende su funcionalidad desde otra clase. La clase que hereda u obtiene desde otra clase se le conoce como subclase o clase hija y la clase que proporciona su código es conocida como superclase o clase padre.

La herencia permite crear objetos más específicos partiendo de una clase general.

BRUCE necesita agregar un objeto a su sistema

Haz recibido una llamada de Mister BRUCE para solicitarte una nueva feature para el sistema.

Su negocio está creciendo y ahora además de vender libros también venderá Comics.

BRUCE no deja nada al azar y ha puesto manos a la obra de una nueva hoja de requerimientos:

**Requerimientos del sistema de librería: Integrar registro de Comics**.

Integrar al sistema el registro de Comics, los cuales al igual que los libros se pueden consultar, pero no modificar.

**Características de comics:**

* las mismas que en los libros
* ilustradores
* volumen

**Funciones:**

* Mostrar información

Después de leer y analizar los nuevos requerimientos te has percatado que las propiedades entre libros y comics son casi las mismas salvo que los comics cuentan con dos más: ilustradores y volumen, además la funcionalidad también es la misma: verInfo()

Podrías pensar que la solución es tan fácil como:

definir una nueva clase llamada: Comic

copiar y pegar propiedades y métodos de la clase Libro en la clase Comic

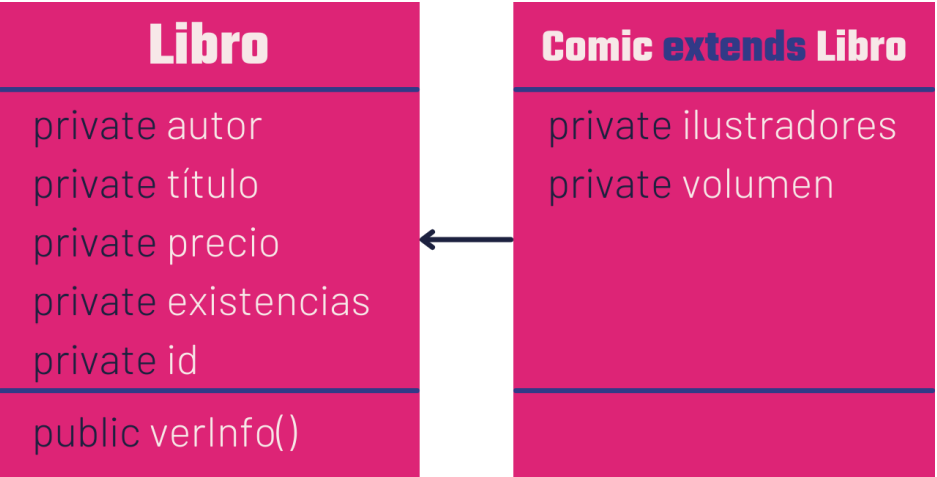
y solo agregar las dos nuevas propiedades que hacen falta para un comic.

Y si, efectivamente tu sistema funcionaría bien si lo haces así, pero copiar y pegar código rompe uno de los principios de la programación: No te repitas (DRY: don't repeat yourself)

Si tu forma de desarrollo se basa en copiar y pegar código de diferentes partes y tener mucho código repetido, poco a poco tu sistema se volverá muy difícil de darle mantenimiento y de hacerlo crecer hacía otras funcionalidades, porqué probablemente siempre tengas que ir a copiar la nueva funcionalidad de un lado a otro y eso se volvería insostenible en un sistema mediano-grande.

En cambio, la herencia, nos permite reutilizar código sin la necesidad de hacer copypaste.

Veamos cómo quedaría nuestro diagrama si creamos una clase Comic que herede de la clase Libro:



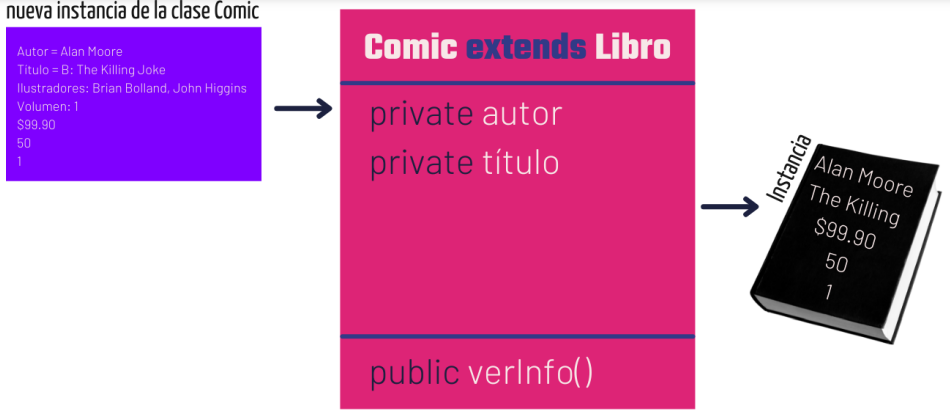
Cuando hablamos de herencia en POO hablamos de extender la funcionalidad de una clase partiendo de otra, por eso la palabra más común para relacionar clases en los lenguajes de programación será la palabra **extends.**

* Ahora la clase Comic extiende de Libro

1. La clase Comic hereda propiedades y métodos de la clase Libro por lo que ya no necesitamos definirlos de nuevo en la clase Comic .
2. Gracias a la herencia ahora solo necesitamos definir las propiedades y métodos que diferencian a un objeto comic de un libro: ilustradores y volumen.

Podemos observar en el diagrama que la clase Comic queda más vacía y en código pasará igual dado que automáticamente estamos reutilizando el código de la clase Libro al momento de heredar.

Ahora podemos instanciar objetos Comic:



**Como dato curioso:** hay lenguajes de programación como Java y Python que aceptan herencia múltiple y otros como PHP que solo permiten heredar de una sola clase. El concepto en sí funciona igual.

En conclusión, la herencia nos permite reutilizar código y crear objetos más especializados.

Polimorfismo

Polimorfismo **es un apodo despampanante para un concepto muy simple probablemente, el más simple de los cuatro pilares. Aunque su descripción también puede traicionarnos sino vemos un ejemplo claro de cómo se aplica.**

La definición literal de polimorfismo es: Muchas formas.

Y en POO: **es la capacidad de que un mismo método obtenga resultados diferentes dependiendo de la subclase desde la cual se ejecute.**

Es confuso porqué... ¿Cómo logramos que una misma función/método obtenga diferentes resultados?

Respuesta: sobrescribiendo el método en la subclase y ajustando un poco.

En nuestro desarrollo tenemos un pequeño problema: el método verInfo() está hecho para mostrar toda la información relacionada con un Libro y no está tomando en cuenta las dos propiedades que se agregaron en la clase Comic (ilustradores y volumen).

Si por ejemplo, simulamos ejecutar el método desde un objeto Comic ahora mismo resultaría en algo como:

"""

Autor = Alan Moore

Título = B: The Killing Joke

$99.90

50

1

"""

Nos mostraría la información de las propiedades que coinciden entre las dos clases, pero hace falta las dos propiedades específicas de la clase Comic.

Esto pasa porqué en nuestra clase comic no hemos definido ningún método verInfo() por lo cual automáticamente está empleando el método que hereda desde la clase Libro.

Si queremos que la funcionalidad este completa tal y como la quiere BRUCE, debemos emplear polimorfismo para implementar el mismo método verInfo() desde la clase Comic y ajustar la información que debe de mostrar:



* Definimos el método verInfo() en la clase Comic

Ahora cuando ejecutemos verInfo() desde un objeto Comic podremos mostrar toda la información, por ejemplo:

"""

Autor = Alan Moore

Título = B: The Killing Joke

Ilustradores: Brian Bolland, John Higgins

Volumen: 1

$99.90

50

1

"""

* Ahora si contamos con la información de los ilustradores y el volumen

El polimorfismo no es más que sobrescribir un método que proviene de la superclase y adaptarlo para obtener el resultado deseado

Ahora podríamos crear tantos objetos libros y comics como Mister Bruce requiera.

Y colorín colorado este desarrollo se ha terminado y quedamos atentos a los nuevos requerimientos de Mister Bruce que seguramente a futuro querrá que apliques.

Infografía:

Mini Book: Entiende POO ya! - YouDev