Clase grabada

Recibe una cálida:

# ¡Bienvenida!

Te estábamos esperando 😁

English Always | alkemy

Clase grabada

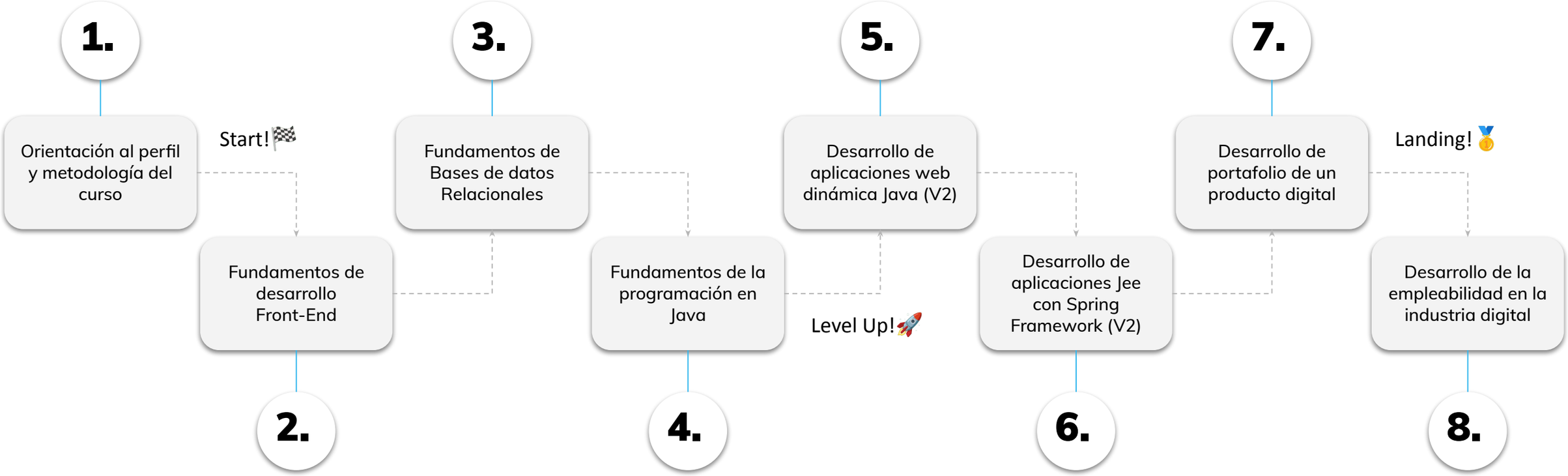
# Polimorfismo y principios básicos de diseño

**Plan formativo:** Desarrollo de Aplicaciones Full Stack Java Trainee V2.0

English Always | alkemy

# HOJA DE RUTA

¿Cuáles **skill** conforman el programa?




English Always | alkemy

# REPASO CLASE ANTERIOR

En la clase anterior trabajamos 📚:

- ✔ *Principio de Abierto/Cerrado*

English Always | alkemy

# LEARNING PATHWAY


English Always | alkemy

# OBJETIVOS DE APRENDIZAJE

¿Qué aprenderemos?

- ***Comprender la importancia e implementación del principio de Liskov en la programación orientada a objetos.***

# Rompehielo🥶

**Respondan en el chat o levantando la mano: 🙌**

1. Los caminos están hechos para que transite gente caminando, en bicicleta, con silla de ruedas. ¿Podría transitar también un automóvil?
2. Las bancas están diseñadas para que se siente una persona. ¿Podría reemplazar a una persona por un elefante?

En programación orientada a objetos tenemos la regla de que los objetos de un subtipo deben poder reemplazarse por objetos del supertipo sin afectar el comportamiento del sistema.

- ¿Que deberíamos modificar para poder implementar este principio?

# Principio de Liskov

# Principio de Liskov

**¿Qué es?**

El principio de Liskov, también conocido como el Principio de Sustitución de Liskov (Liskov Substitution Principle o LSP por sus siglas en inglés), es uno de los cinco principios SOLID de la programación orientada a objetos (POO) y fue formulado por Barbara Liskov en 1987. El principio de Liskov establece una **regla fundamental para la herencia y la abstracción** en la POO y se resume de la siguiente manera:

"**Los objetos de una clase derivada (subclase) deben ser capaces de reemplazar objetos de la clase base (superclase) sin afectar la corrección del programa**".

# Principio de Liskov

**Algunos puntos clave:**

- **Subclases como Sustitutas**: Una subclase debe ser una sustituta válida de su superclase. Esto significa que debe proporcionar la misma interfaz (métodos) y cumplir las mismas expectativas de comportamiento que la superclase.
- **Herencia Correcta**: La herencia debe utilizarse de manera que refleje una relación "es un" entre la subclase y la superclase. Esto significa que una subclase debe ser una extensión lógica de la superclase y no debe alterar su semántica básica.
- I**nvariantes Preservadas**: Una invariante es una propiedad que se mantiene constante en el tiempo. Si la superclase tiene invariantes, la subclase no debe romperlas ni modificarlas.

# Principio de Liskov

- **Precondiciones y Postcondiciones**: Si la superclase tiene precondiciones (condiciones que deben cumplirse antes de que un método se ejecute) y postcondiciones (resultados esperados después de que un método se ejecute), entonces la subclase debe mantener esas condiciones.
- **Herencia Responsable**: La herencia debe usarse de manera responsable y reflexiva. No todas las relaciones entre clases deben modelarse como herencia, a veces es más apropiado usar composición u otras técnicas.

# Principio de Liskov

# Principio de Liskov

El principio de Liskov nos da una serie de pautas para guiar la herencia entre clases. La principal que debe cumplir si estamos realizando la herencia de una manera correcta es que cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.

**Si cuando sobreescribimos un método en la clase que hereda necesitamos lanzar una excepción o no realiza nada, entonces probablemente estamos violando el LSP**.

Esta premisa es clave para detectar fallos en la arquitectura de nuestra aplicación o posibles “code smells”.

# Evaluación Integradora ✨

¿Listos para un nuevo desafío? En esta clase comenzamos a construir nuestro....

## Trabajo Integrador del Módulo 💪🏽

Iremos completándolo progresivamente clase a clase.

English Always | alkemy

# LIVE DEMO

Ejemplo en vivo

**Aplicando principio de Liskov:**

*Vamos a ver un ejemplo donde aplicaremos el principio de Liskov para mejorar el diseño de nuestro programa.*

- *Como desarrollador de una entidad bancaria se te solicita implementar el sistema de manejo de cuentas bancarias. El cual, en la primera fase, implementará la cuenta básica y la premium. La diferencia entre ambas es que las cuentas premium generan acumulación de puntos “prefiero” cada vez que se realiza un depósito.*

**Tiempo: 30 minutos**

# LIVE DEMO

Ejemplo en vivo

```java
public abstract class CuentaBancaria {

    /**
     * Método encargado de realizar los depositos monetarios.
     * @param monto         Monto a depositar.
     */
    public abstract void depositar(double monto);

    /**
     * Método encargado de retirar el monto especificado.
     * @param monto         Monto a retirar.
     * @return              Retorna verdadero en casos exitosos,
     *                      falso en caso contrario.
     */
    public abstract boolean retirar(double monto);
}
```

*Para llevar a cabo este desarrollo, defines la siguiente clase abstracta. La clase abstracta define la obligatoriedad en las clases derivadas de definir el funcionamiento de los métodos abstractos de depositar y retirar.*

# LIVE DEMO

**Ejemplo en vivo**

*Para el manejo de las cuentas bancarias básicas y premium, se definen las siguientes clases:*

```java
public class CuentaBancariaBasica extends CuentaBancaria {

    private double saldo;

    @Override
    public void depositar(double monto) {
        this.saldo += monto;
    }

    @Override
    public boolean retirar(double monto) {
        if(this.saldo < monto)
            return false;
        else{
            this.saldo -= monto;
            return true;
        }

    }

}
```

```java
public class CuentaBancariaPremium extends CuentaBancaria {

    private double saldo;
    private int puntosPrefiero;

    @Override
    public void depositar(double monto) {
        this.saldo += monto;
        incrementarPuntosPrefiero();
    }

    @Override
    public boolean retirar(double monto) {
         if(this.saldo < monto)
            return false;
        else{
            this.saldo -= monto;
            incrementarPuntosPrefiero();
            return true;
        }
    }

    private void incrementarPuntosPrefiero(){
        this.puntosPrefiero++;
    }

}
```

English Always | alkemy

# LIVE DEMO

Ejemplo en vivo

```java
public class ServicioRetiros {

    public static final double MONTO_GASTO_ADMON = 25.00;

    public void cargarDebitarCuentas(){

        CuentaBancaria ctaBasica = new CuentaBancariaBasica();
        ctaBasica.depositar(100.00);

        CuentaBancaria ctaPremium = new CuentaBancariaPremium();
        ctaPremium.depositar(200.00);

        List<CuentaBancaria> cuentas = new ArrayList();

        cuentas.add(ctaBasica);
        cuentas.add(ctaPremium);

        debitarGastosAdmon(cuentas);

    }

    private void debitarGastosAdmon(List<CuentaBancaria> cuentas){
        cuentas.stream()
                .forEach(cuenta -> cuenta.retirar(ServicioRetiros.MONTO_GASTO_ADMON));
    }
}
```

*A las cuentas básicas y premium se les realiza un cobro administrativo anual de $25.00. Para implementar esta política se implementa la clase de servicio de retiros.*

# LIVE DEMO

Ejemplo en vivo

*Una vez finalizada la implementación de las cuentas básicas y premium, se te solicita la implementación de las cuentas a largo plazo. Las características de este tipo de cuentas son las siguientes:*

- *Las cuentas bancarias de depósito a largo plazo están exentas de cobros administrativos.*
- *Las cuentas bancarias de depósito a largo plazo no permiten retiros manuales. Si un cliente desea retirar lo abonado a su cuenta, lo debe de realizar mediante una gestión diferente en las sucursales del banco.*

*Como desarrollador encargado del sistema de cuentas bancarias,* ***decides utilizar la misma clase abstracta definida en los ejemplos anteriores****.*

# LIVE DEMO

Ejemplo en vivo

*En este punto es donde resalta la violación al **principio de sustitución de Liskov**, ya que las cuentas a largo plazo no deben de extender el funcionamiento de la operación retirar. Esto significa que si intentamos forzar la extensión de la clase debemos de dejar el método retirar como un método vacio o que lance una excepción de método no soportado.*

*Además, se generaría un problema en el servicio de debito de gastos administrativos, ya que si por error se incluyera una cuenta a largo plazo se daría una excepción de ejecución.*

```java
public class CuentaLargoPlazo extends CuentaBancaria{

    private double saldo;

    @Override
    public void depositar(double monto){
        this.saldo += monto;
    }

    @Override
    public boolean retirar(double monto){
        throw new UnsupportedOperationException("No permite la operación de debito");
    }
}
```

# LIVE DEMO

Ejemplo en vivo

*Se le podría dar la vuelta al error e incluir una condicional en el método de "debitarGastosAdmon" para que ignore a las cuentas a largo plazo.*

```
private void debitarGastosAdmon(List<CuentaBancaria> cuentas){
        for(CuentaBancaria cuenta : cuentas){
            if(cuenta instanceof CuentaLargoPlazo)
                continue;
            else
                cuenta.retirar(ServicioRetiros.MONTO_GASTO_ADMON));
        }
    }
```

*Sin embargo, la inclusión del condicional viola el principio de abierto/cerrado debido a que cada vez que se incluya una cuenta bancaria con diferente funcionamiento, habrá que modificar el código para condicionarlo acorde a la clase.*

# LIVE DEMO

Ejemplo en vivo

**Implementación del principio de Liskov:**

*El problema integral del ejemplo anterior es que **la cuenta bancaria a largo plazo no es una cuenta regular**, al menos no del tipo que se definió en la clase abstracta.*

*Existe una prueba de razonamiento inductivo que se puede utilizar en estos casos "Si grazna como un pato, camina como un pato y se comporta como un pato, entonces, ¡seguramente es un pato!". La prueba del pato es importante bajo el concepto del principio de sustitución de Liskov debido a que **la cuenta bancaria a largo plazo, luce como una cuenta bancaria regular, pero no se comporta como una cuenta regular, ya que no permite retiros**. Por lo que inferimos que las cuentas a largo plazo no son cuentas bancarias regulares.*

# LIVE DEMO

Ejemplo en vivo

**Para cumplir con el principio de sustitución de Liskov se deben de tomar en cuenta las siguientes consideraciones:**

1. *Los tres tipos de cuentas permiten la acción de depósito.*
2. *Únicamente se permite realizar retiros en cuentas bancarias básicas y premium. Esto significa que existen dos tipos de cuentas, las cuentas bancarias retirables y las no retirables.*
3. *Se define una clase abstracta con el único método de depositar.*
4. *Se definirá una segunda clase abstracta que extenderá el comportamiento de la primera clase abstracta e incluirá la acción de retirar.*
5. *Las cuentas bancarias del tipo básico y premium serán del tipo de cuenta bancaria retirable. Mientras que las cuentas bancarias a largo plazo serán del tipo cuenta bancaria, la cual únicamente permite la acción de depósito.*

# LIVE DEMO

Ejemplo en vivo

*Para llevar a cabo los cambios de estructura se define la clase abstracta CuentaBancaria, la cual es la clase básica para todas las cuentas y tiene un único método para realizar la acción de depósito.*

```java
public abstract class CuentaBancaria {

    /**
     * Método encargado de realizar los depositos monetarios.
     * @param monto          Monto a depositar.
     */
    public abstract void depositar(double monto);
}
```

# LIVE DEMO

Ejemplo en vivo

*Se extiende el funcionamiento de la clase de CuentaBancaria para definir el comportamiento de las cuentas bancarias que permiten retiros. La clase CuentaBancariaRetirable es también una clase abstracta que define el método retirar.*

```
public abstract class CuentaBancariaRetirable extends CuentaBancaria {

    /**
     * Método encargado de retirar el monto especificado.
     * @param monto         Monto a retirar.
     * @return              Retorna verdadero en casos exitosos,
     *                      falso en caso contrario.
     */
    public abstract boolean retirar(double monto);
}
```

# LIVE DEMO

Ejemplo en vivo

*Las cuentas bancarias básicas y premium extenderán el uso de la clase de CuentaBancariaRetirable, la que a su vez extiende la clase CuentaBancaria. Esto permite que las clases de cuentas básicas y premium tengan las acciones de deposito y retiro.*

```
public class CuentaBancariaBasica extends CuentaBancariaRetirable {

    private double saldo;

    @Override
    public void depositar(double monto) {
        this.saldo += monto;
    }

    @Override
    public boolean retirar(double monto) {
        if(this.saldo < monto)
            return false;
        else{
            this.saldo -= monto;
            return true;
        }
```

# LIVE DEMO

Ejemplo en vivo

*Las cuentas bancarias básicas y premium extenderán el uso de la clase de CuentaBancariaRetirable, la que a su vez extiende la clase CuentaBancaria. Esto permite que las clases de cuentas básicas y premium tengan las acciones de deposito y retiro.*

```java
public class CuentaBancariaBasica extends CuentaBancariaRetirable {

    private double saldo;

    @Override
    public void depositar(double monto) {
        this.saldo += monto;
    }

    @Override
    public boolean retirar(double monto) {
        if(this.saldo < monto)
            return false;
        else{
            this.saldo -= monto;
            return true;
        }
```

```java
public class CuentaBancariaPremium extends CuentaBancariaRetirable {

    private double saldo;
    private int puntosPrefiero;

    @Override
    public void depositar(double monto) {
        this.saldo += monto;
        incrementarPuntosPrefiero();
    }

    @Override
    public boolean retirar(double monto) {
         if(this.saldo < monto)
            return false;
        else{
            this.saldo -= monto;
            incrementarPuntosPrefiero();
            return true;
        }
    }

    private void incrementarPuntosPrefiero(){
        this.puntosPrefiero++;
    }
```

# LIVE DEMO

Ejemplo en vivo

*Debido a que las cuentas a largo plazo únicamente permiten depósitos, se extenderá el uso de la clase CuentaBancaria.*

```
public class CuentaBancariaLargoPlazo extends CuentaBancaria {

    private double saldo;

    @Override
    public void depositar(double monto) {
        this.saldo += monto;
    }

}
```

# LIVE DEMO

Ejemplo en vivo

*La clase encargada de proveer el servicio de retiros utilizará únicamente las clases que extiendan el funcionamiento de CuentaBancariaRetirable.*

```
public class ServicioRetiros {

    public static final double MONTO_GASTO_ADMON = 25.00;

    public void cargarDebitarCuentas(){

        CuentaBancariaRetirable ctaBasica = new CuentaBancariaBasica();
        ctaBasica.depositar(100.00);

        CuentaBancariaRetirable ctaPremium = new CuentaBancariaPremium();
        ctaPremium.depositar(200.00);

        List<CuentaBancariaRetirable> cuentas = new ArrayList();

        cuentas.add(ctaBasica);
        cuentas.add(ctaPremium);

        debitarGastosAdmon(cuentas);

    }

    private void debitarGastosAdmon(List<CuentaBancariaRetirable> cuentas){
        cuentas.stream()
                .forEach(cuenta -> cuenta.retirar(ServicioRetiros.MONTO_GASTO_ADMON));
    }
}
```

# LIVE DEMO

Ejemplo en vivo

***Conclusión:***

*El método encargado de debitar los gastos administrativos ahora recibe un listado de objetos que extienden el funcionamiento de la clase CuentaBancariaRetirable. Es decir, nosotros podemos sustituir los objetos de CuentaBancariaRetirable por cuentas básicas o premium, siempre y cuando estas extiendan el uso de la clase abstracta CuentaBancariaRetirable, pero no podemos hacer la sustitución por objetos que solo extienden la clase CuentaBancaria, tal como el caso de las cuentas a largo plazo.*

*El diseño de la estructura de nuestras clases también refuerza los requerimientos funcionales dados por el cliente. Ya que las cuentas a largo plazo, al extender el funcionamiento de la clase CuentaBancaria permiten que únicamente se pueda realizar depósitos en este tipo de cuentas.*

## Ejercicio N° 1

# Sustituyendo

# Sustituyendo

**Volvemos a trabajar con figuras geométricas:** 🙌

En este caso nuestro objetivo será diseñar una jerarquía de clases que cumpla con el Principio de Liskov y permita el cálculo de áreas de figuras geométricas.

**Consigna:** ✍️

1. Crea una clase base abstracta llamada FiguraGeometrica que contenga los siguientes métodos abstractos:

   **double calcularArea()** y **String obtenerNombre():** Un método que devuelve el nombre de la figura (por ejemplo, "Círculo" o "Rectángulo").

2. Deriva tres clases concretas de FiguraGeometrica: Circulo, Rectangulo y Triangulo. Cada una de estas clases debe implementar los métodos abstractos calcularArea() y obtenerNombre() de la clase base.
3. En el Main crea instancias de objetos de cada una de las subclases y calcula y muestra el área de cada figura utilizando el método calcularArea() y también muestra el nombre de cada figura utilizando el método obtenerNombre().

# Sustituyendo

**Pistas:** ⚙️

1. Investiga las fórmulas para calcular el área de un círculo, un rectángulo y un triángulo.
2. Asegúrate de que las subclases proporcionen implementaciones válidas para calcular estas áreas.
3. Utiliza polimorfismo para llamar a los métodos calcularArea() y obtenerNombre() en un bucle que recorre una lista de objetos de tipo FiguraGeometrica.

**Tiempo🕐:** *25 minutos*

# ¿Alguna consulta?

# RESUMEN

¿Qué logramos en esta clase?

- ✓ ***Comprender el principio de Liskov y su importancia e implementación***

English Always | alkemy

# #WorkingTime

Continuemos ejercitando

**¡Antes de cerrar la clase!** Te invitamos a: 👇👇👇

1. Repasar nuevamente la grabación de esta clase
2. Revisar el material compartido en la plataforma de Moodle (lo que se vio en clase y algún ejercicio adicional)
   a. Material 1 *(Foro)*
   b. *Lectura Módulo 4, Lección 6: página 9*
3. Traer al próximo encuentro, todas tus dudas y consultas para verlas antes de iniciar nuevo tema.

# ¡Muchas Gracias!

Nos vemos en la próxima clase 👋🏻

English Always | alkemy

Momento:

# Time-out!

⏰5 min.