Clase grabada

Recibe una cálida:

¡Bienvenida!

Te estábamos esperando 😁

English Always | alkemy

Clase grabada

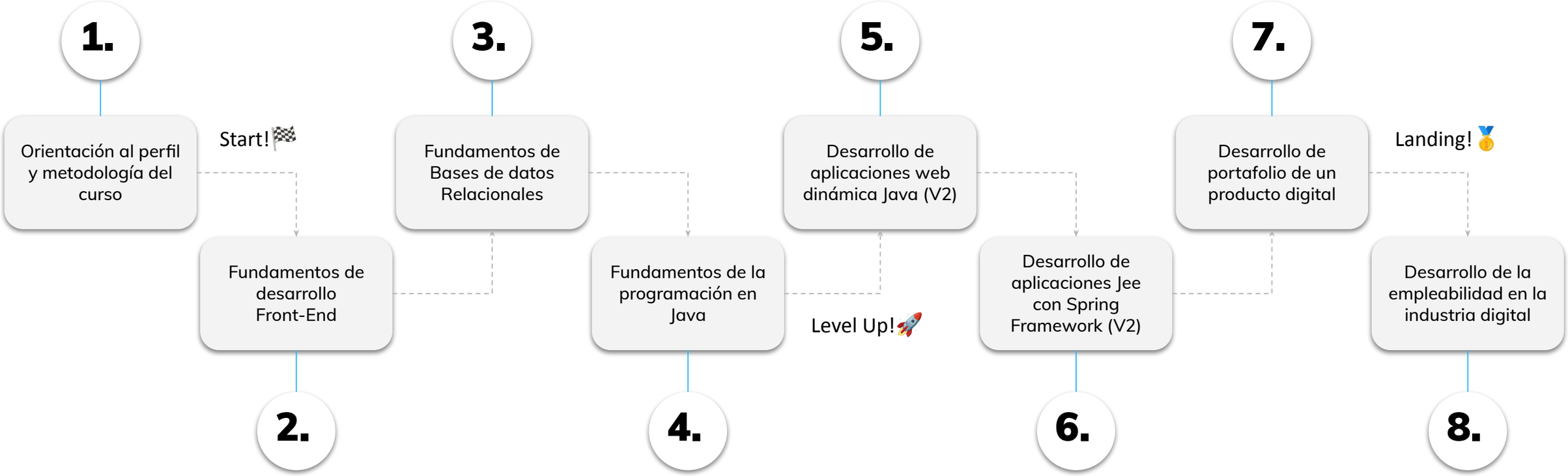
Polimorfismo y principios básicos de diseño

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

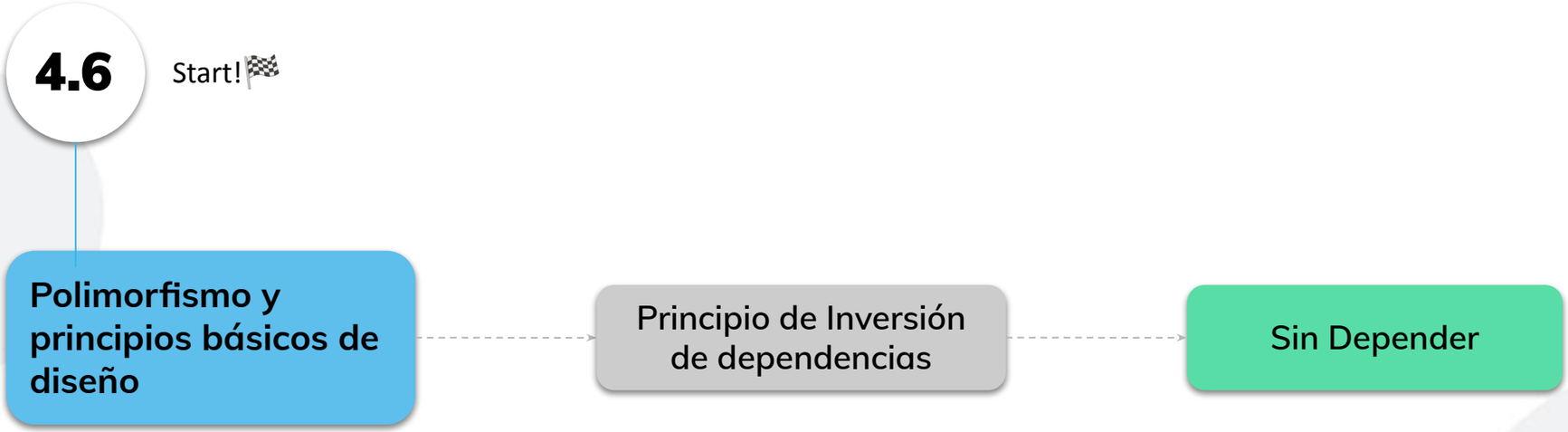
REPASO CLASE ANTERIOR

En la clase anterior trabajamos 📚:

- ✓ *Principio de segregación de interfaces*

English Always | alkemy

LEARNING PATHWAY



English Always | alkemy

OBJETIVOS DE APRENDIZAJE

¿Qué aprenderemos?

- *Comprender la aplicación del Principio de Inversión de dependencias*

Rompehielo🥶

Respondan en el chat o levantando la mano: ✍️

- ¿De qué depende la casa para mantenerse en pie?
- Si la base se debilita ¿qué sucede?
- ¿Es mejor depender de fundamentos sólidos o cambiantes?
- ¿Que decisión tomarían ustedes, crear cimientos como la primera o la segunda casa?

Ésta analogía nos permite entender que en programación es mejor depender de abstracciones estables que de implementaciones volátiles.

Principio de Inversión de Dependencias

Principio de Inversión de Dependencias

¿Qué es?:

El principio de inversión de dependencias indica que **las clases de un sistema deben depender de las abstracciones/interfaces y no de las implementaciones concretas**. Esto significa que las clases no deben depender directamente de clases específicas, sino de interfaces o clases abstractas. Esto lo haremos inyectando dependencias en el constructor de la clase, pero estas dependencias serán interfaces o clases abstractas, no clases finales.

Este principio se centra en la inversión de la dirección de las dependencias en un sistema. En lugar de que los módulos de alto nivel dependan de los detalles de los módulos de bajo nivel, ambas partes deben depender de abstracciones, lo que promueve un diseño más flexible y mantenible.

Principio de Inversión de Dependencias

Algunos conceptos clave:
Los **módulos de alto nivel** suelen ser componentes o clases que encapsulan la lógica principal de una aplicación. Los **módulos de bajo nivel** suelen ser componentes o clases que realizan tareas específicas y están destinados a ser utilizados por los módulos de alto nivel.

- Los módulos de alto nivel no deben depender de los módulos de bajo nivel, **ambos tiene que depender de abstracciones**.

- Las abstracciones no deben depender de los detalles. **Los detalles deben depender de abstracciones**.

Principio de Inversión de Dependencias

Principales ventajas:

- **Flexibilidad**: Al depender de abstracciones en lugar de detalles concretos, el sistema se vuelve más flexible y menos propenso a cambios drásticos cuando se modifican los módulos de bajo nivel.
- **Mantenibilidad**: Cambiar o actualizar los módulos de bajo nivel no debería afectar los módulos de alto nivel si se siguen las abstracciones adecuadas.
- **Reutilización**: Las abstracciones permiten la reutilización de código a través de la implementación de diferentes módulos de bajo nivel que cumplen con la misma interfaz.
- **Pruebas** y **Mocking**: Facilita las pruebas unitarias y el uso de objetos simulados (mocks) para pruebas, ya que se pueden crear implementaciones de abstracciones que se comporten de manera controlada.

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 Inversión de Dependencias:

Supongamos que tenemos una clase para realizar el acceso a datos, y lo hacemos a través de una BBDD.

```
class DatabaseService{
    //...
    void getDatos(){ //... }
}

class AccesoADatos {

    private DatabaseService databaseService;

    public AccesoADatos(DatabaseService databaseService){
        this.databaseService = databaseService;
    }

    Dato getDatos(){
        databaseService.getDatos();
        //...
    }
}
```

Tiempo: 25 minutos

LIVE DEMO

Ejemplo en vivo

Imaginemos que en el futuro queremos cambiar el servicio de BBDD por un servicio que conecta con una API. Detengámonos un minuto a pensar qué habría que hacer... ***¿Ven el problema?*** *Tendríamos que* ***ir modificando todas las instancias de la clase AccesoADatos, una por una.***

Esto es debido a que nuestro módulo de alto nivel (AccesoADatos) ***depende de un módulo de más bajo nivel*** *(DatabaseService), violando así el principio de inversión de dependencias.* ***El módulo de alto nivel debería depender de abstracciones****.*

Tiempo: 25 minutos

LIVE DEMO

Ejemplo en vivo

Para arreglar esto, podemos hacer que el módulo AccesoADatos dependa de una abstracción más genérica. Así, sin importar el tipo de conexión que se le pase al módulo AccesoADatos, ni este ni sus instancias tendrán que cambiar, por lo que nos ahorraremos mucho trabajo.

```
interface Conexion {
    Dato getDatos();
    void setDatos();
}

class AccesoADatos {

    private Conexion conexion;

    public AccesoADatos(Conexion conexion){
        this.conexion = conexion;
    }

    Dato getDatos(){
        conexion.getDatos();
    }
}
```

LIVE DEMO

Ejemplo en vivo

Ahora, cada servicio que queramos pasar a AccesoADatos deberá implementar la interfaz Conexion. Así, tanto el módulo de alto nivel como el de bajo nivel dependen de abstracciones, por lo que cumplimos el principio de inversión de dependencias. Además, esto nos forzará a cumplir el principio de Liskov, ya que los tipos derivados de Conexion (DatabaseService y APIService) son sustituibles por su abstracción (interfaz Conexion).

```
class DatabaseService implements Conexion {

    @Override
    public Dato getDatos() { //... }

    @Override
    public void setDatos() { //... }
}

class APIService implements Conexion{

    @Override
    public Dato getDatos() { //... }

    @Override
    public void setDatos() { //... }
}
```

Ejercicio N° 1

Sin Depender

Sin Depender

Vamos a aplicar la inversión de dependencias en la clase FiguraGeometrica: 🙌

Para poner en práctica lo aprendido, vamos a modificar las clases de geometría que venimos trabajando en ejercicios anteriores para que cumpla con lo requerido en la consigna!

Consigna: ✍️

Dada una clase Geometria con método calcularÁrea() que tiene dependencia directa de Figura (es decir, que hereda de). Modificar para que la dependencia sea hacia una interfaz FiguraGeométrica implementada por Figura.

Tiempo🕐: *20 minutos*

¿Alguna consulta?

RESUMEN

¿Qué logramos en esta clase?

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

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.