



Recibe una cálida:

¡Bienvenida!

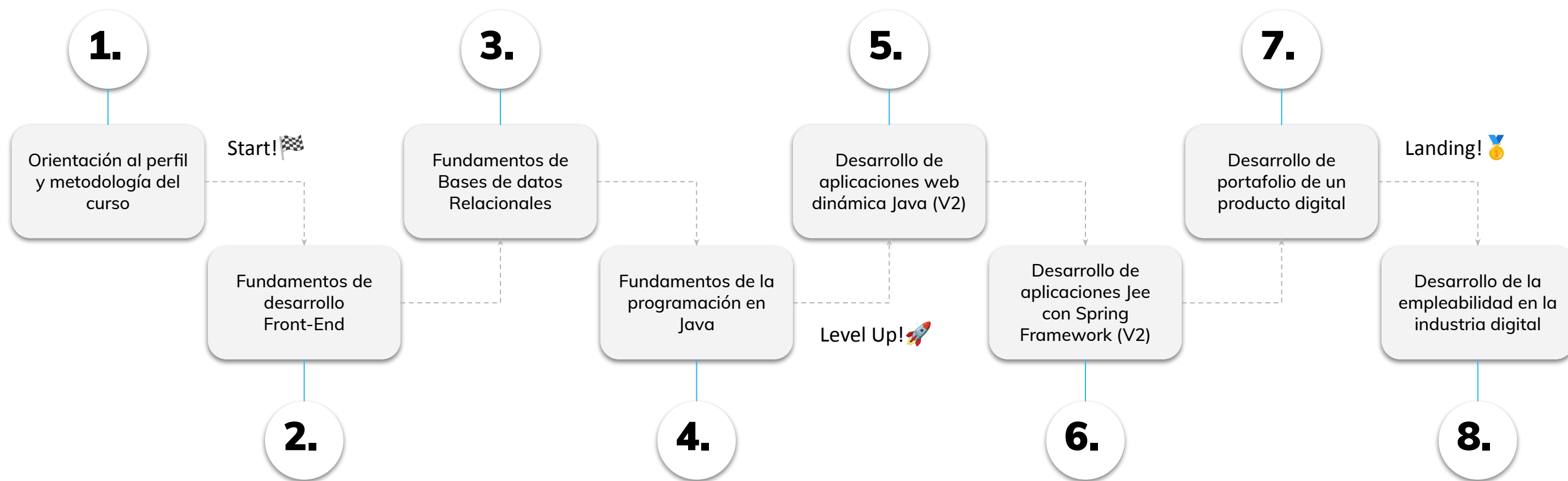
Te estábamos esperando 😊 

➤ Pruebas Unitarias en Java

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

HOJA DE RUTA

¿Cuáles **skill** conforman el programa?



REPASO CLASE ANTERIOR

En la clase anterior trabajamos :

- ✓ Integración de JUnit en Eclipse IDE

LEARNING PATHWAY

4.7

Start! 🏁

Pruebas Unitarias en Java

Utilización de Fixtures en las unidades de prueba

Utilización de Fixtures en las unidades de prueba

@Anotando

OBJETIVOS DE APRENDIZAJE

¿Qué aprenderemos?



Comprender la implementación de fixtures en las unidades de prueba



Aprender la correcta utilización de las anotaciones de JUnit



› Utilización de Fixtures en las unidades de prueba




Utilización de Fixtures en las unidades de prueba




¿Qué son los Fixtures?:

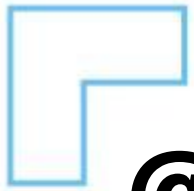
Son objetos o estructuras de datos que se utilizan para **establecer un estado** inicial predefinido **antes** de ejecutar las pruebas y **restaurar el estado** original **después** de que se completen las pruebas.



Los fixtures son esenciales para garantizar que las pruebas se ejecuten de manera **aislada y consistente**.

En JUnit, los fixtures se definen utilizando **anotaciones** y métodos especiales que se ejecutan antes y después de cada prueba o de la suite de pruebas.





@Anotaciones

En JUnit5 encontraremos diversas anotaciones, las cuales empiezan con “@” y nos facilitan la escritura de los Test. Veamos algunas de ellas:

- **@Test:** Esta anotación se utiliza para marcar un método como una prueba unitaria. Los métodos anotados con @Test deben ser públicos y no devolver ningún valor.
- **@BeforeEach:** Esta anotación se utiliza para marcar un método que se ejecuta antes de cada prueba. Puedes usarlo para realizar configuraciones o inicializaciones necesarias antes de cada prueba.
- **@AfterEach:** Esta anotación se utiliza para marcar un método que se ejecuta después de cada prueba. Puedes usarlo para limpiar recursos o realizar acciones posteriores a cada prueba.

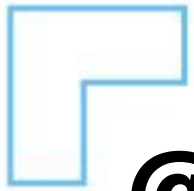
```
class EjemploTest {  
    // private MiClase miObjeto;  
  
    @BeforeEach  
    public void configuracionInicial() {  
        // Metodo que se correra antes de cada @Test  
        // miObjeto = new MiClase();  
    }  
  
    @AfterEach  
    public void limpiezaFinal() {  
        // Accion final que se ejecutara al final de cada @Test  
        // miObjeto = null; liberar el objeto  
    }  
  
    @Test  
    void test1() {  
        int expected = 1;  
        int actual = 1;  
        assertEquals(expected, actual);  
    }  
  
    @Test  
    void test2() {  
        int unexpected = 2;  
        int actual = 1;  
        assertNotEquals(unexpected, actual);  
    }  
}
```



@Anotaciones

- **@BeforeAll:** Esta anotación se utiliza para marcar un método que se ejecuta una sola vez antes de todas las pruebas en una clase. El método anotado debe ser estático.
- **@AfterAll:** Esta anotación se utiliza para marcar un método que se ejecuta una sola vez después de todas las pruebas en una clase. El método anotado debe ser estático.

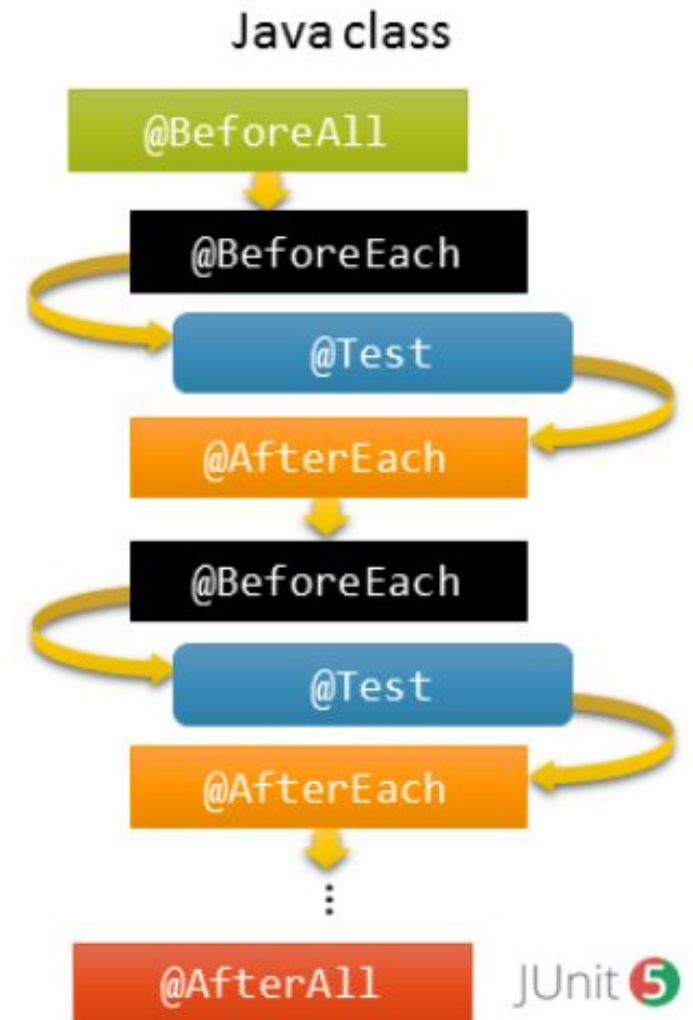
```
class EjemploTest {  
  
    // private MiClase miObjeto;  
  
    @BeforeAll  
    public void configuracionInicial() {  
        // Metodo que se corra una unica vez antes de que se corran los @Test  
        // miObjeto = new MiClase();  
    }  
  
    @AfterAll  
    public void limpiezaFinal() {  
        // Accion final que se corra cuando se corra el ultimo @Test  
        // miObjeto = null; Liberar el objeto  
    }  
  
    @Test  
    void test1() {  
        int expected = 1;  
        int actual = 1;  
        assertEquals(expected, actual);  
    }  
  
    @Test  
    void test2() {  
        int unexpected = 2;  
        int actual = 1;  
        assertNotEquals(unexpected, actual);  
    }  
}
```



@Anotaciones

- **@DisplayName:** Se utiliza para proporcionar un nombre descriptivo para una prueba o una clase de prueba.
- **@Disabled:** Se utiliza para deshabilitar una prueba o una clase de prueba temporalmente sin eliminarla por completo.
- **@ParameterizedTest:** Permite ejecutar una misma prueba con diferentes conjuntos de parámetros.
- **@RepeatedTest:** Se utiliza para marcar un método como una prueba repetida y puedes especificar el número de repeticiones.
- **@Timeout:** Se utiliza para establecer un límite de tiempo para una prueba.

En la imagen podremos ver el orden de ejecución de los test:



Evaluación Integradora ✨

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

Trabajo Integrador del Módulo 💪

Iremos completándolo progresivamente clase a clase.



LIVE CODING

Ejemplo en vivo

¡Vamos a colocar las anotaciones!:

Es hora de anotar los métodos de nuestra clase Test

1. *Crear los métodos (sin implementar) y escribir las anotaciones necesarias de las clases Test que creamos en la clase anterior.*

 **Tiempo: 30 minutos**



Ejercicio N° 1

@Anotando



@Anotando



Consigna: 🖋️ **Vamos a generar nuestras primeras @notaciones**

Ahora vamos a escribir la anotaciones necesarias en los métodos que creas pertinentes implementar en las clases Test generadas en el ejercicio anterior (de las clases de Polimorfismo).



Recuerda que puedes declarar los métodos a testear, pero aún no es necesario que pienses en su implementación. Vamos a hacer hincapié en las anotaciones a utilizar.



Tiempo 🕒: 30 minutos

○

¿Alguna consulta?

+



RESUMEN

¿Qué logramos en esta clase?

- ✓ **Comprender la implementación de *fixtures* a través de anotaciones en JUnit5**



#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. *Lectura Módulo 4, Lección 7: páginas 10 - 12*
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 🙌



Momento: ✚

Time-out!

🕒 5 min.

