Recibe una cálida:

¡Bienvenida!

Te estábamos esperando 😁

alkemy

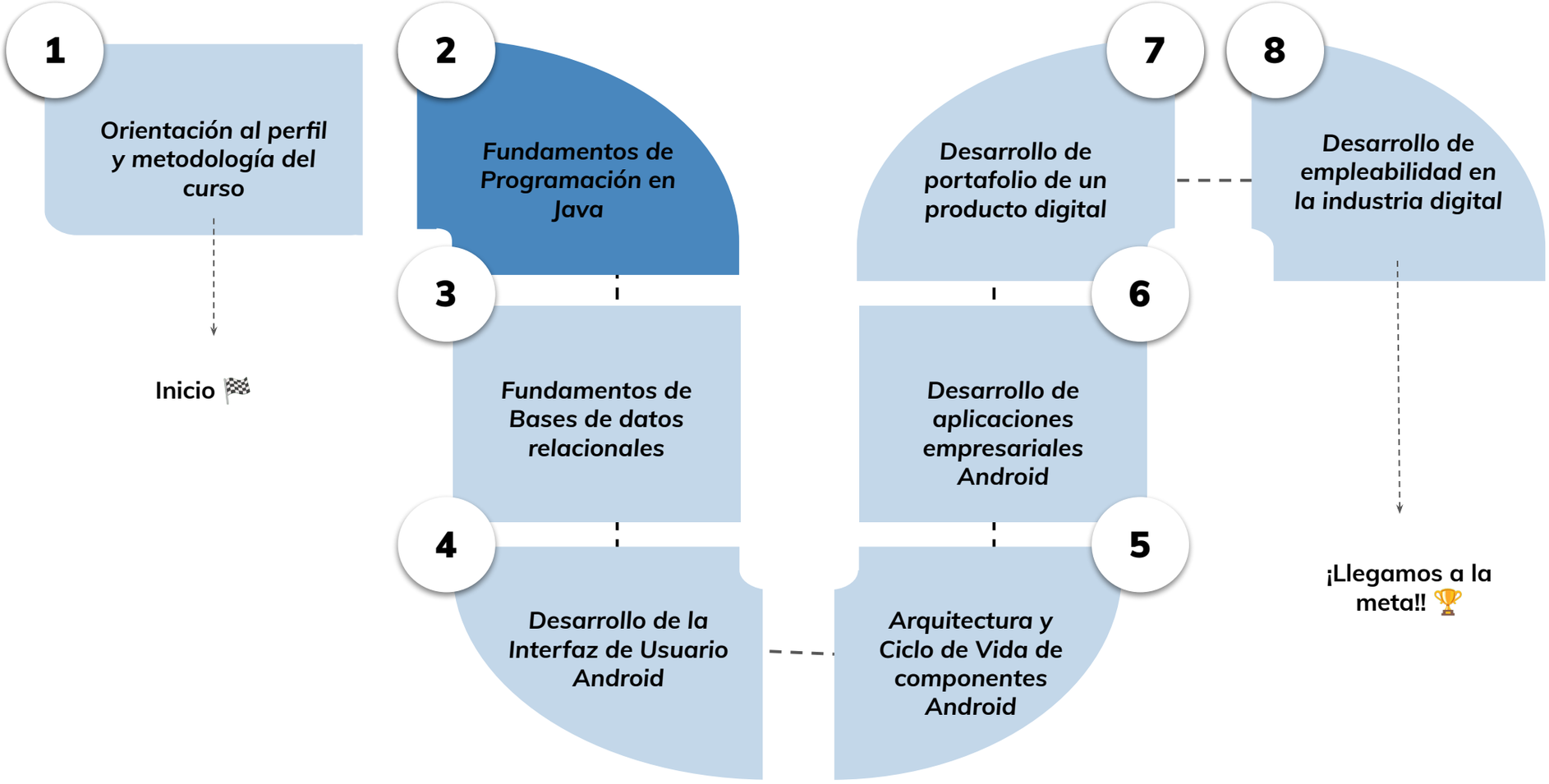
Clase grabada

Pruebas unitarias en java

AE7.1: Implementar una suite de pruebas unitarias en lenguaje java utilizando junit para asegurar el buen funcionamiento de una pieza de software.

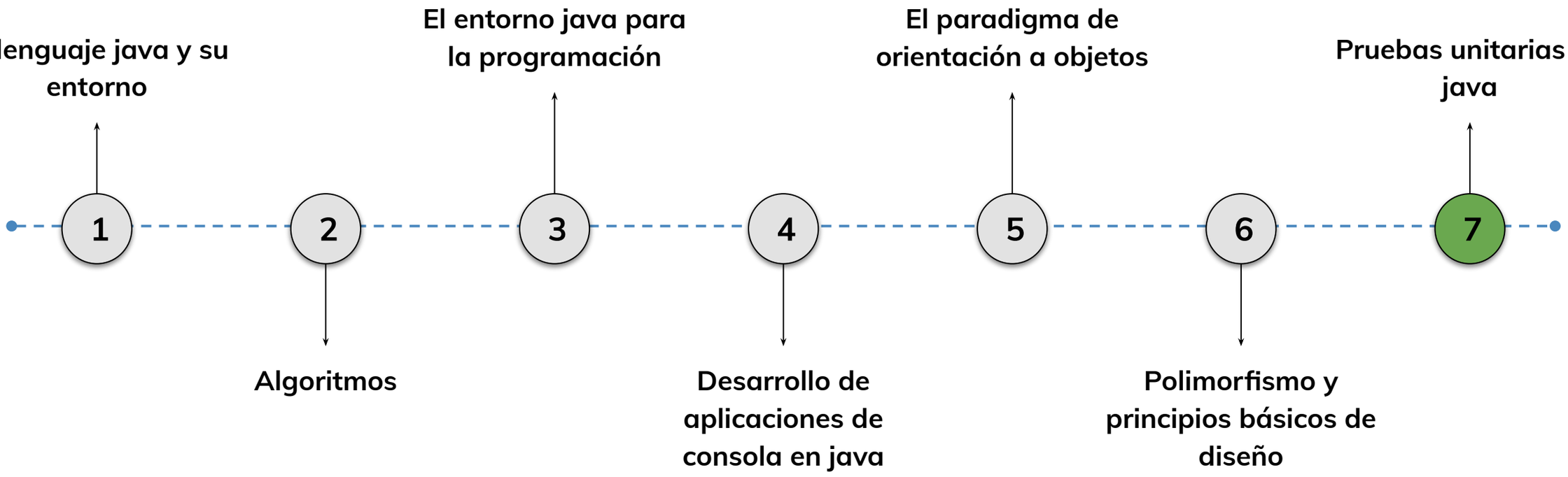
Hoja de ruta

*¿Cuáles **módulos** conforman el programa?*



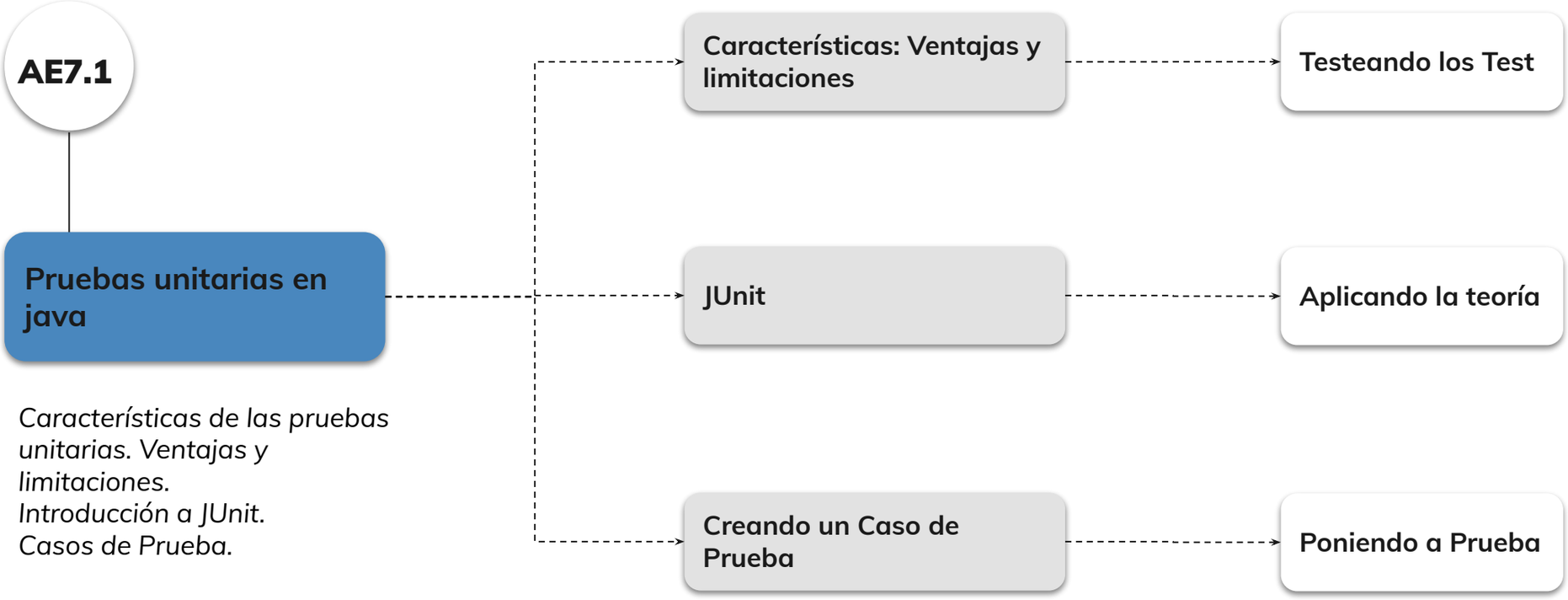
Roadmap de lecciones

*¿Cuáles **lecciones** estaremos estudiando en este módulo?*



Learning Path

¿Cuáles temas trabajaremos hoy?



Características de las pruebas unitarias. Ventajas y limitaciones.
Introducción a JUnit.
Casos de Prueba.

alkemy

Objetivos de aprendizaje

¿Qué aprenderás?

- Conocer las características de los test unitarios en Java
- Comprender las ventajas y limitaciones de los test unitarios en Java
- Comprender el concepto e implementación de la herramienta JUnit para Test Unitarios en Java
- Comprender el concepto e implementación de los Casos de Prueba

< > Rompehielo🥶

Respondan en el chat o levantando la mano: 🙌

1. ¿Cómo podríamos probar que el método sumar funciona correctamente?
2. ¿Qué beneficios obtendremos de escribir estas pruebas?
3. ¿En qué casos sería más complicado escribir pruebas automáticas?
4. ¿Creen que vale la pena el esfuerzo de construir estas pruebas? ¿Por qué?

```java
public class Calculadora {

  public int sumar(int a, int b) {
    return a + b;
  }

}
```

Test Unitarios: Características

Test Unitarios

¿Qué son?:

Los test unitarios en Java son una parte fundamental de la práctica de pruebas en el desarrollo de software. Son pequeños fragmentos de código diseñados para evaluar el comportamiento de una unidad de código aislada, por lo general una función o método, en un entorno controlado. Estas pruebas verifican si la unidad de código (como una clase o método) se comporta como se espera.

Test Unitarios

Características principales

- **Independencia:** Las pruebas unitarias deben ser independientes entre sí, lo que significa que cada prueba debe poder ejecutarse de forma aislada sin depender de otras pruebas o del entorno externo.

- **Automatización**: Las pruebas unitarias se diseñan para ser ejecutadas de forma automatizada. Se utilizan frameworks como JUnit, TestNG o Mockito para facilitar la creación y ejecución de las pruebas.

- **Aislamiento:** Cada prueba unitaria debe probar una única funcionalidad o comportamiento específico. Para lograr esto, se utilizan mocks o stubs para aislar la unidad en prueba de sus dependencias externas.

Test Unitarios

Características principales

- Cobertura o Coverage: Las pruebas unitarias deben cubrir la mayor cantidad de código posible. El objetivo es asegurar que todas las funcionalidades y caminos del código sean probados, minimizando la posibilidad de errores.

- Repetitividad: Las pruebas unitarias deben ser repetibles y producir resultados consistentes. Esto significa que deben poder ejecutarse en cualquier entorno sin depender de factores externos.

- Rapidez: Las pruebas unitarias deben ser rápidas de ejecutar, ya que se espera que se puedan ejecutar frecuentemente, incluso durante el proceso de desarrollo.

- Legibilidad: Es importante que las pruebas unitarias sean claras y comprensibles para facilitar su mantenimiento y comprensión por parte de otros desarrolladores.

Test Unitarios: Ventajas

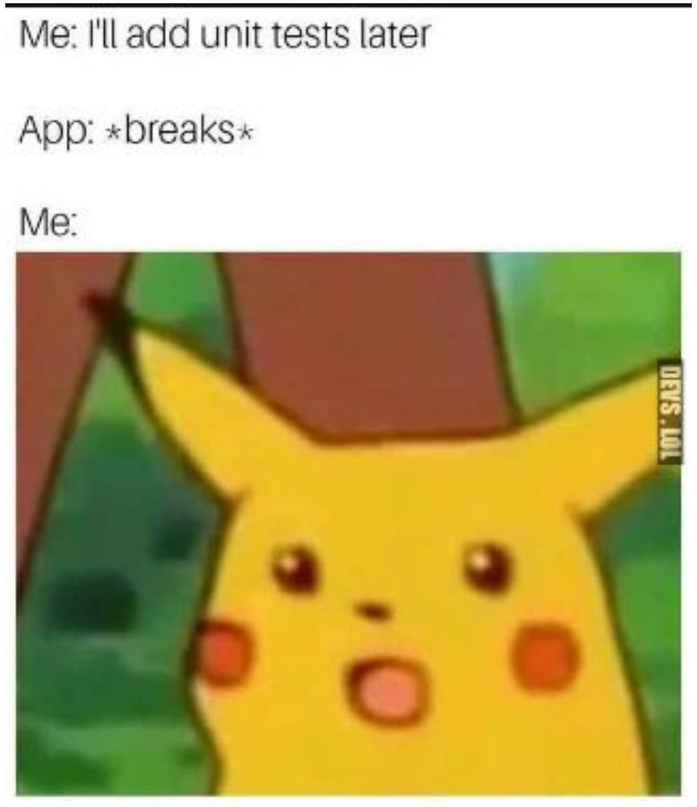
La principales ventajas de implementar test unitarios en nuestros programas son:

- Detección temprana de errores: Las pruebas unitarias te permiten identificar errores y problemas en el código de manera temprana. Al ejecutar las pruebas de forma regular, puedes capturar y corregir los errores en las primeras etapas del desarrollo, lo que facilita la solución de problemas y evita que se propaguen a otras partes del código.
- Mejora de la calidad del código: Las pruebas unitarias fomentan el desarrollo de un código de alta calidad. Al escribir pruebas, debes pensar en la funcionalidad de forma aislada y en cómo probarla adecuadamente. Esto conduce a un código más modular, desacoplado y mantenible.

Test Unitarios: Ventajas

La principales ventajas de implementar test unitarios en nuestros programas son:

- Facilita el proceso de refactorización: Las pruebas unitarias actúan como una red de seguridad al realizar refactorizaciones en el código. Siempre que realices cambios en tu código, puedes ejecutar las pruebas unitarias para asegurarte de que las modificaciones no hayan introducido nuevos errores.
- Documentación y comprensión del código: Las pruebas unitarias bien escritas sirven como documentación viva de cómo se espera que funcione el código. Al leer las pruebas, otros desarrolladores pueden comprender rápidamente el comportamiento esperado de una determinada funcionalidad o clase.

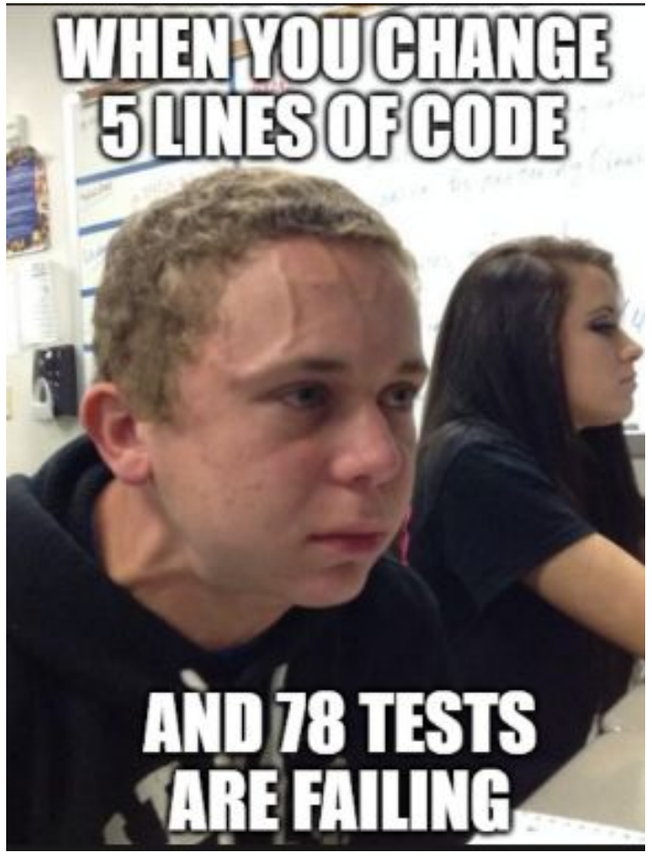


Test Unitarios: Ventajas

La principales ventajas de implementar test unitarios en nuestros programas son:

- Facilita el trabajo en equipo: Las pruebas unitarias permiten a los desarrolladores trabajar en paralelo y de manera colaborativa. Cada desarrollador puede escribir pruebas para las funcionalidades que está implementando y ejecutarlas para asegurarse de que no haya conflictos con otras partes del código.

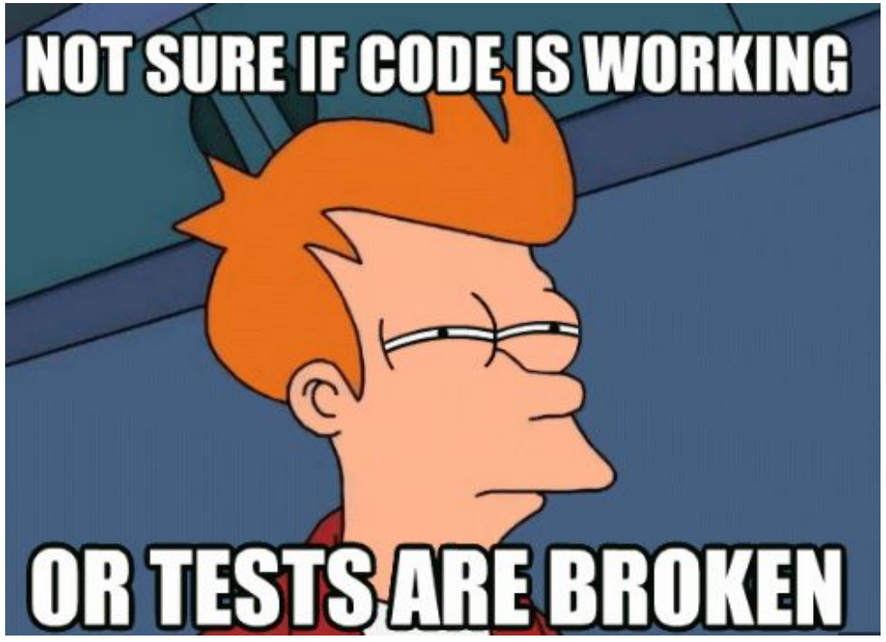
- Mejora la productividad: Aunque escribir pruebas unitarias requiere tiempo adicional, a largo plazo, puede mejorar la productividad del equipo. Al detectar y corregir errores tempranamente, se reducen los tiempos de depuración y se evitan problemas en etapas posteriores del desarrollo.



Test Unitarios: Ventajas

La principales ventajas de implementar test unitarios en nuestros programas son:

- Facilita la integración continua y entrega continua: Las pruebas unitarias son un componente clave de las prácticas de integración continua y entrega continua. Al tener un conjunto de pruebas sólidas, puedes automatizar su ejecución en los sistemas de integración continua, lo que garantiza que el código sea siempre estable y funcional.



Test Unitarios: Desventajas

Podemos encontrarnos con distintas escuelas o líneas de pensamiento que expongan “Desventajas” de las pruebas unitarias, pero la realidad es que TODAS las desventajas, pueden ser reducidas, logrando destacar así las ventajas.

Las principales desventajas a la hora de implementar Test unitarios en nuestro código y las posibles soluciones son:

- Costo de desarrollo: Es necesario invertir tiempo y recursos para escribir las pruebas y mantenerlas actualizadas a medida que el código evoluciona. Esto puede aumentar la carga de trabajo inicial y prolongar los plazos de entrega. Pero sabiendo que si son bien planteadas, otorgan la posibilidad de estabilidad de la app ante futuros cambios.

Test Unitarios: Desventajas

- Curva de aprendizaje: Para escribir pruebas unitarias efectivas, los desarrolladores deben familiarizarse con los frameworks y herramientas de pruebas, como JUnit o TestNG. Esto implica una curva de aprendizaje adicional para el equipo de desarrollo, especialmente si no están familiarizados con estas herramientas. Si bien es una desventaja, no es nada que no se pueda solucionar estudiando sobre Test Unitarios y practicando en nuestro dia a dia.
- Mantenimiento continuo: A medida que el código evoluciona, las pruebas unitarias también deben actualizarse y mantenerse. Si no se actualizan correctamente, las pruebas pueden volverse obsoletas y generar falsos positivos o falsos negativos. Esto requiere dedicar tiempo y esfuerzo para mantener el conjunto de pruebas actualizado y relevante.

Test Unitarios: Desventajas

- Cobertura limitada: Aunque las pruebas unitarias pueden cubrir una parte significativa del código, es posible que no logren cubrir todos los casos posibles. Algunos aspectos del sistema, como la interacción con bases de datos o servicios externos, pueden ser difíciles de probar mediante pruebas unitarias, lo que limita la cobertura total.

- Sobrecarga de pruebas: Si no se administra adecuadamente, el exceso de pruebas unitarias puede generar una sobrecarga en el proceso de desarrollo. Demasiadas pruebas pueden ralentizar el tiempo de compilación y ejecución, y puede ser difícil mantener todas las pruebas actualizadas y relevantes. Esto puede ser tranquilamente solucionado, planteando bien los diseños y arquitecturas de la app, con el fin de evitar monolitos saturados en múltiples módulos y tests.

LIVE CODING

Imaginemos el siguiente escenario y respondamos entre todos:

Antes de permitir la creación de una CuentaBancaria, debemos crear una función que valide si una persona es mayor de edad según su edad ingresada.

1. *¿Cómo podrían probar que la función funciona correctamente antes de usarla?*
2. *¿Qué ventajas les daría crear pruebas unitarias?*
3. *¿Pero también habría desventajas o limitaciones?*

Tiempo: 20 Minutos

alkemy

LIVE CODING

https://gist.github.com/Mariocanedo/02f8606edce044c5699269c0e8179da6

github
SOCIAL CODING

alkemy

Momento:

Time-out!

⏰5 -10 min.

alkemy

Introducción a JUnit

JUnit

Ciclo de Vida de una Prueba:

JUnit sigue un ciclo de vida para las pruebas: configuración (setup), ejecución de la prueba y limpieza (teardown).

La configuración y la limpieza se realizan antes y después de cada prueba para garantizar que estén en un estado conocido.

Mensajes de Error Significativos:
Cuando una prueba falla, JUnit proporciona mensajes de error significativos que ayudan a identificar la causa del fallo.

Esto facilita la depuración y corrección de problemas.

JUnit

Herramientas de IDE:
Muchas herramientas de desarrollo integrado (IDE) como Eclipse, IntelliJ IDEA y NetBeans ofrecen soporte integrado para JUnit. Esto facilita la creación, ejecución y análisis de pruebas desde el entorno de desarrollo.

Cobertura de Código:
Algunas herramientas y complementos de JUnit pueden calcular la cobertura de código, lo que te permite saber cuántas líneas de código están siendo probadas por tus pruebas.

Pruebas Continuas (CI/CD):
Las pruebas unitarias con JUnit son esenciales en las prácticas de integración continua (CI) y entrega continua (CD), ya que ayudan a garantizar que el código nuevo no rompa funcionalidades existentes.

JUnit

¿Qué es necesario conocer antes de escribir Test con JUnit?

Programación en Java: Es fundamental tener un buen entendimiento de los conceptos básicos de programación en Java, como variables, tipos de datos, estructuras de control (if, bucles, etc.), clases, métodos y excepciones.

POO: Dado que JUnit se utiliza para probar unidades de código a nivel de clase o método, es importante tener conocimientos de programación orientada a objetos.

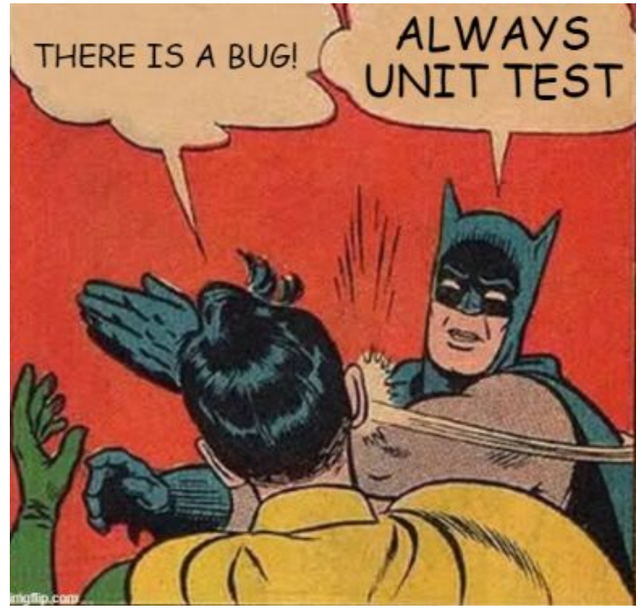
Método de prueba: Antes de utilizar JUnit, es beneficioso entender qué es un método de prueba y cómo funciona. Un método de prueba es una función que verifica el comportamiento esperado de una unidad de código específica. Debes comprender cómo escribir métodos de prueba y cómo verificar resultados utilizando aserciones (assertions).

JUnit

¿Qué es necesario conocer antes de escribir Test con JUnit?

Anotaciones: JUnit utiliza anotaciones Java para marcar los métodos de prueba y definir su comportamiento. Es importante comprender cómo funcionan las anotaciones en Java y cómo se aplican en el contexto de JUnit. Algunas anotaciones clave en JUnit son @Test, @BeforeEach, @AfterEach, @BeforeAll y @AfterAll, conceptos que más adelante explicaremos.

Aserciones (assertions): Las aserciones son declaraciones que se utilizan para verificar si una condición es verdadera o falsa. JUnit5 proporciona un conjunto de métodos de aserción que se utilizan para evaluar y validar resultados en las pruebas unitarias. Es importante conocer los diferentes métodos de aserción disponibles en JUnit5 y cómo utilizarlos correctamente.



JUnit

Para utilizar JUnit en Java, es necesario seguir los siguientes pasos:

1. Descargar e instalar JUnit en el sistema. Esto se puede hacer mediante la descarga de la biblioteca JAR de JUnit y la configuración del Classpath para incluirlo.
2. Crear una clase de prueba que contenga métodos de prueba que verifiquen la funcionalidad de las unidades de código a probar.
3. Anotar los métodos de prueba con la anotación @Test para indicar que son métodos de prueba.
4. Crear objetos de las clases a probar y llamar a sus métodos para verificar la funcionalidad. Utilice métodos de aserción para verificar que los resultados esperados se obtienen.

CREANDO PRUEBAS UNITARIAS

```
import org.junit.Test;
import static org.junit.Assert.*;
 public class MiClaseTest {
                      @Test
              public void testMetodo() {
              // Crear un objeto de la clase a probar
              MiClase clase = new MiClase();
             // Llamar al método que se quiere probar
                int resultado = clase.metodo(2, 3);
               // Verificar que el resultado sea el esperado
                    assertEquals(5, resultado);
                 }
}
```

CREANDO PRUEBAS UNITARIAS

En el ejemplo anterior, se crea una clase de prueba llamada "MiClaseTest" que tiene un método de prueba llamado "testMetodo".

El método de prueba crea un objeto de la clase "MiClase" y llama al método "metodo" con argumentos 2 y 3. Luego, utiliza la aserción "assertEquals" para verificar que el resultado devuelto por el método es 5.
Al ejecutar esta prueba con JUnit, se verifica que la clase "MiClase" funciona correctamente y se obtiene un mensaje de éxito si la prueba pasa. Si la prueba falla, se muestra un mensaje de error que indica qué aserción falló y con qué valores.

CREANDO PRUEBAS UNITARIAS

Crear pruebas unitarias

Estas líneas para la versión del compilador de java:

XML

```xml
<build>
<finalName>aritmetica</finalName>
   <plugins>
       <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-compiler-plugin</artifactId>
           <version>${maven.compiler.plugin.version}</version>
           <configuration>
               <target>${java.version}</target>
               <source>${java.version}</source>
           </configuration>
       </plugin>
   </plugins>
</build>
```

CREANDO PRUEBAS UNITARIAS

Crear pruebas unitarias

Añade, en el fichero **pom.xml** las siguiente líneas de declaración:

XML

```xml
<properties>
    <java.version>1.8</java.version>
    <junit.jupiter.version>5.0.2</junit.jupiter.version>
    <junit.platform.version>1.0.2</junit.platform.version>
    <maven.compiler.plugin.version>3.8.1</maven.compiler.plugin.version>
</properties>
```

CREANDO PRUEBAS UNITARIAS

Crear pruebas unitarias

Estas líneas para la versión del compilador de java:

XML

```xml
<build>
<finalName>aritmetica</finalName>
   <plugins>
       <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-compiler-plugin</artifactId>
           <version>${maven.compiler.plugin.version}</version>
           <configuration>
               <target>${java.version}</target>
               <source>${java.version}</source>
           </configuration>
       </plugin>
   </plugins>
</build>
```

CREANDO PRUEBAS UNITARIAS

Crear pruebas unitarias

Supongamos que queremos crear un test para esta clase.

JAVA

```java
public class Aritmetica {
    private float ultimoResultado;
    public float suma(float primerSumando, float segundoSumando) {
        return ultimoResultado = primerSumando + segundoSumando;
    }
    public float resta(float minuendo, float sustraendo) {
        return ultimoResultado = minuendo - sustraendo;
    }
    public float multiplicacion(float primerFactor,
        float segundoFactor) {
        return ultimoResultado = primerFactor * segundoFactor;
    }
    public float division(float dividendo, float divisor) {
        return ultimoResultado = dividendo / divisor;
    }
    public float getUltimaResultado() {
        return ultimoResultado;
  }
```

CREANDO PRUEBAS UNITARIAS

@Test: Indica que un método es una prueba de unidad. Este método debe ser público, no devolver ningún valor y no tener parámetros.

La etiqueta @Test

En el paquete **test** crea la clase **AritmeticaTest**, y dentro de esa clase crea el siguiente método:

La etiqueta **@Test** marca un método como método de prueba.

Los métodos de prueba siempre deben ser **public void**.

JAVA

```java
@Test
public void testSuma() {
    fail("Not yet implemented");
}
```

El método **fail(String)** hace fallar el test.

Ejecuta el método situando el curso sobre su nombre y pulsado Shift+Ctrl+F10, verás como falla el test.

CREANDO PRUEBAS UNITARIAS

La etiqueta @Test

Escribamos el código de prueba.

JAVA

```java
@Test
public void testSuma() {
    Aritmetica aritmetica = new Aritmetica();
    assertEquals(2, aritmetica.suma(1, 1));
}
```

1. Creamos una instancia de la clase.
2. assertEquals(valorEsperado, valorReal, error) compara el valor esperado con el real dentro de un error.
3. Ejecutamos el test.

CREANDO PRUEBAS UNITARIAS

La etiqueta @Test

Ejecuta de nuevo el test y verás como pasa.

Ahora escribe los tests de la página siguiente.

La etiqueta @Test

JAVA

```java
@Test
public void testResta() {
    Aritmetica aritmetica = new Aritmetica();
    assertEquals(3, aritmetica.resta(4, 1));
}
@Test
public void testMultiplicacion() {
    Aritmetica aritmetica = new Aritmetica();
    assertEquals(6, aritmetica.multiplicacion(2, 3));
}
@Test
public void testDivision() {
    Aritmetica aritmetica = new Aritmetica();
    assertEquals(5, aritmetica.division(10, 2));
}
```

Ejecútalos todos tests situando el cursor del ratón sobre el nombre de la clase y pulsando Shift+Crtl+F10.

CREANDO PRUEBAS UNITARIAS

La etiqueta @Test

Una de las cosas buenas de utilizar **Maven** es que no necesitas usar ningun entorno de desarrollo para compilar/probar/ejecutar nuestros proyectos, podemos hacerlo desde un terminal: **mvn test-compile test** para compilar y lanzar la ejecución de las pruebas.

CREANDO PRUEBAS UNITARIAS

@Before: Indica que un método debe ser ejecutado antes de cada prueba. Este método se utiliza comúnmente para configurar las condiciones necesarias para la prueba

La etiqueta @BeforeEach

JAVA

```java
public class AritmeticaTest {
    private Aritmetica aritmetica;
    @BeforeEach
    public void init() {
        aritmetica = new Aritmetica();
    }
    @Test
    public void testSuma() {
        assertEquals(2, aritmetica.suma(1, 1), 0);
    }
    @Test
    public void testResta() {
        assertEquals(3, aritmetica.resta(4, 1), 0);
    }
    ...
}
```

CREANDO PRUEBAS UNITARIAS

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

                public class MyTest {

                        private MyObject myObject;

                          @BeforeEach
                                public void setUp() {
                                        myObject = new
MyObject();
                          }
                              @Test
                    public void testMyObject() {
                              // Prueba de métodos de MyObject
utilizando myObject

                          }

                     }
```

@BeforeEach es una anotación en JUnit que se utiliza para indicar que un método debe ejecutarse antes de cada método de prueba en una clase de prueba. Esto se utiliza comúnmente para inicializar datos y recursos que se necesitan para las pruebas.

CREANDO PRUEBAS UNITARIAS

```
import org.junit.jupiter.api.BeforeAll;
 import org.junit.jupiter.api.Test;
public class MyTest {

 static MyObject myObject;

                                @BeforeAll
                                 public static void setUp() {
                                  myObject = new MyObject();

}
                                    @Test
     public void testMyObject() {
                    // Prueba de métodos de MyObject utilizando
myObject
                                                        }
                                     }
                        }
```

La anotación **@BeforeAll** es utilizada en JUnit para indicar que un método debe ser ejecutado una vez antes de que se inicien las pruebas en una clase de prueba. Esta anotación se utiliza comúnmente para realizar tareas de inicialización que se aplican a todas las pruebas de la clase.

CREANDO PRUEBAS UNITARIAS

La etiqueta @BeforeAll

Antes de mostrar cómo se usa la etiqueta **@BeforeAll** cambiemos la definición de la clase **Aritmética**:

JAVA

```java
public class Aritmetica {
    public float suma(float primerSumando, float segundoSumando) {
        return primerSumando + segundoSumando;
    }
    public float resta(float minuendo, float sustraendo) {
        return minuendo - sustraendo;
    }
    public float multiplicacion(float primerFactor,
        float segundoFactor) {
        return primerFactor * segundoFactor;
    }
    public float division(float dividendo, float divisor) {
        return dividendo / divisor;
    }
}
```

CREANDO PRUEBAS UNITARIAS

@AfterEach es una anotación en JUnit que se utiliza para indicar que un método debe ejecutarse después de cada método de prueba en una clase de prueba. Esto se utiliza comúnmente para liberar recursos o limpiar datos después de cada prueba.

La etiqueta @AfterEach

De igual modo, la etiqueta **@AfterEach** nos premite realizar tareas de limpieza después de realizar cada uno de los test.

JAVA

```java
public class AritmeticaTest {
    private Aritmetica aritmetica;

    @BeforeEach
    public void init() {
        aritmetica = new Aritmetica();
    }

    @AfterEach
    public void finish() {
        aritmetica = null;
    }
    ...
```

Podemos marcar con **@AfterEach** más de un método.

CREANDO PRUEBAS UNITARIAS

@AfterEach es una anotación en JUnit que se utiliza para indicar que un método debe ejecutarse después de cada método de prueba en una clase de prueba. Esto se utiliza comúnmente para liberar recursos o limpiar datos después de cada prueba.

La etiqueta @AfterEach

De igual modo, la etiqueta **@AfterEach** nos premite realizar tareas de limpieza después de realizar cada uno de los test.

JAVA

```java
public class AritmeticaTest {
    private Aritmetica aritmetica;

    @BeforeEach
    public void init() {
        aritmetica = new Aritmetica();
    }

    @AfterEach
    public void finish() {
        aritmetica = null;
    }
    ...
```

Podemos marcar con **@AfterEach** más de un método.

CREANDO PRUEBAS UNITARIAS

```
import org.junit.jupiter.api.AfterEach;
 import org.junit.jupiter.api.Test;

                public class MyTest {
                  private MyObject myObject;

                    @BeforeEach
                      public void setUp() {
            myObject = new MyObject();

                   }
```

```
            @Test
public void testMyObject() { //    Prueba de
métodos de MyObject utilizando myObject
}
            @AfterEach
      public void tearDown() {
            myObject = null;
                  }


}
```

CREANDO PRUEBAS UNITARIAS

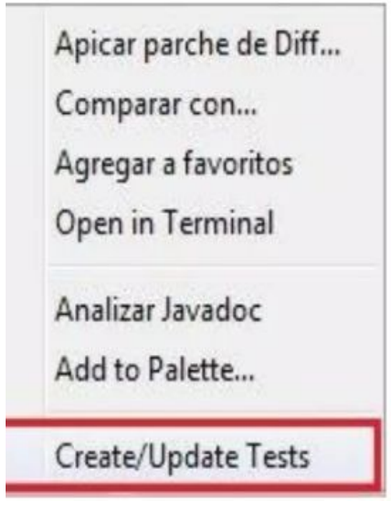
```
import org.junit.Test;
import org.junit.Before;
 import org.junit.After;
import org.junit.BeforeClass;
 import org.junit.AfterClass;
import static org.junit.Assert.*;

 public class MiClaseTest {
 @BeforeClass
       public static void setupClass() {
// Configuración para todas las pruebas en
la clase }
 @Before
         public void setup() {
// Configuración para cada prueba
 }
```

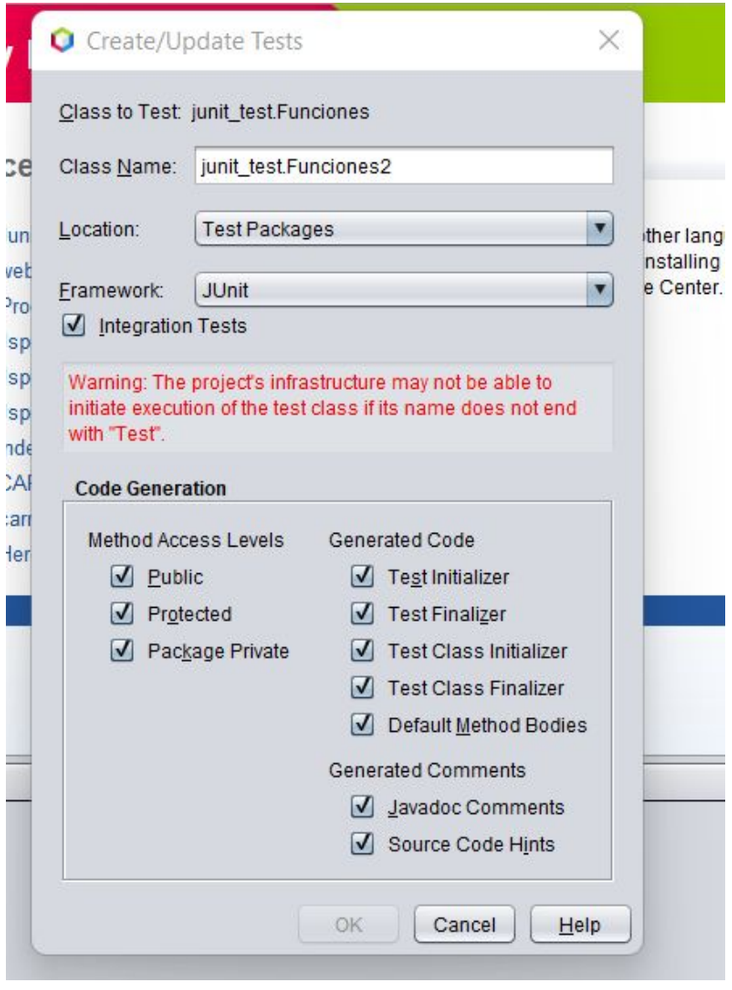
```
@Test
            public void testMetodo() {
        // Prueba de unidad
 }
 @After
             public void tearDown() {
   // Limpiar después de cada prueba
}
 @AfterClass
              public static void tearDownClass() {
// Limpiar después de todas las pruebas en la
clase
                              }
            }
```

CREANDO PRUEBAS UNITARIAS

1. Hacemos click derecho sobre la clase que vamos hacer su clase de prueba

2. Seleccionar Crear o Actualizar Prueba

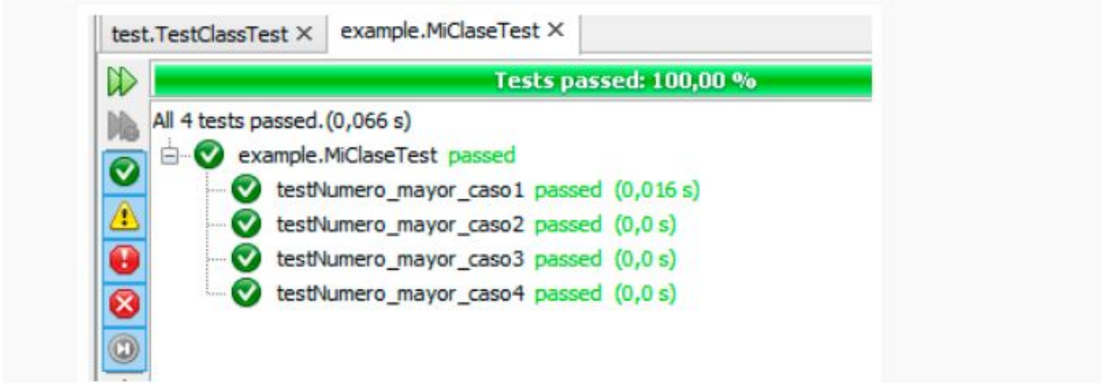


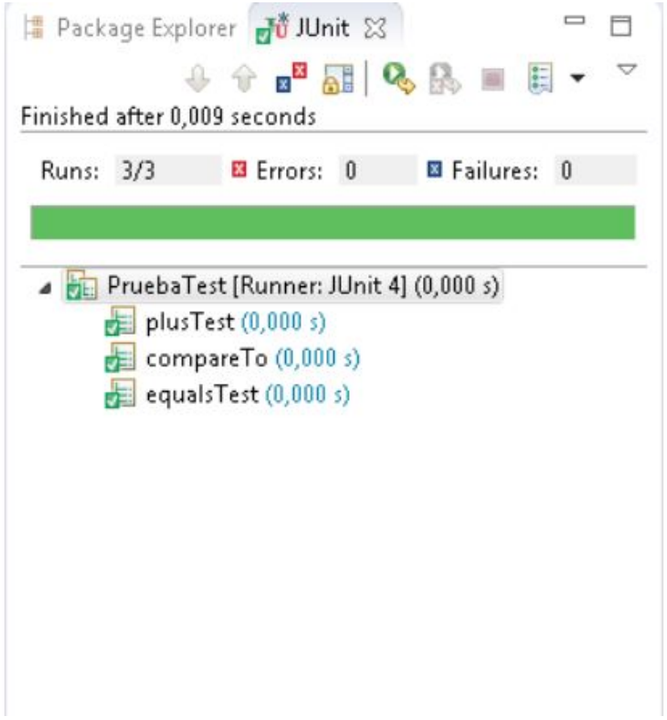


alkemy

CREANDO PRUEBAS UNITARIAS

Click derecho sobre la clase de prueba y la ejecutamos





Como mencionamos en un principio, este plugin nos permite ver más a detalle lo que pasa con el código y los caminos que este toma. Para ejecutarlo, clic derecho sobre el proyecto -> Test with JaCoCoverage. Se abrirá una pagina en el navegador que tengamos configurado por defecto.

Ejemplo:
https://www.jc-mouse.net/java/ejemplo-de-pruebas-unitarias-en-java

alkemy

PRUEBAS UNITARIAS

```java
import org.junit.Test;
import static org.junit.Assert.*;
import java.util.ArrayList;

public class ArrayListTest {

    @Test
    public void testAdd() {
        ArrayList<String> list = new ArrayList<String>();
        list.add("Hello");
        list.add("World");
        assertEquals(2, list.size());
    }
```

En este ejemplo, estamos probando algunas operaciones comunes en un ArrayList que contiene Strings. En el primer método de prueba, testAdd, creamos una nueva instancia de ArrayList, agregamos dos cadenas a la lista y luego comprobamos que la longitud de la lista es 2 utilizando el método assertEquals de JUnit.

PRUEBAS UNITARIAS

```java
@Test
public void testRemove() {
    ArrayList<String> list = new ArrayList<String>();
    list.add("Hello");
    list.add("World");
    list.remove("Hello");
    assertEquals(1, list.size());
    assertEquals("World", list.get(0));
}
```

En este método de prueba, **testRemove**, también creamos una nueva instancia de ArrayList y agregamos dos cadenas a la lista. Luego eliminamos la cadena "Hello" y comprobamos que la longitud de la lista es 1 y que la cadena "World" sigue siendo el primer elemento en la lista.

PRUEBAS UNITARIAS

```
    @Test
    public void testClear() {
        ArrayList<String> list = new ArrayList<String>();
        list.add("Hello");
        list.add("World");
        list.clear();
        assertEquals(0, list.size());
    }
}
```

En el tercer método de prueba, testClear, creamos una nueva instancia de ArrayList, agregamos dos cadenas a la lista y luego llamamos al método clear para eliminar todos los elementos de la lista. Finalmente, comprobamos que la longitud de la lista es 0 utilizando el método assertEquals de JUnit.

Ejercicio N° 1

Testeando los Test

Testeando los Test

Contexto: 🙌

Debes definir cuáles de las siguientes características de los Test Unitarios son PROS y cuáles son CONTRAS.

Consigna: ✍️

En sala reducidas, respondan con pro o contra:

- Detecta errores temprano
- No prueban la UI ni la DB
- Requieren mantenimiento
- Facilitan el refactoring
- Aíslan el código bajo prueba
- Aumentan la carga de desarrollo inicial
- Automatización de las pruebas
- Simplicidad y rapidez de ejecución
- Sobrecarga si no se administran bien
- Requiere aprender frameworks de testing
- Independencia entre pruebas
- Complejidad de mockear dependencias

Al finalizar, tendremos una puesta en común en la sala principal

Tiempo🕐: *20 minutos*

LIVE CODING

Vamos a crear las primeras clases Test para el proyecto de BilleteraVirtual:

1. *Crear una clase de pruebas CuentaTest para probar la clase Cuenta.*
2. *Crear una clase de pruebas TransaccionTest para probar la clase Transaccion.*
3. *Crear una clase de pruebas MonedaTest para probar la interfaz Moneda.*
4. *Explicar el uso de* ***assertEquals*** *para verificar el resultado.*

Tiempo: 30 Minutos

alkemy

Ejercicio N° 2

Aplicando la teoría

Aplicando la teoría

A responder levantando la mano o en el chat (justifica la respuesta): 🙌

Vamos a responder con V o F las siguientes afirmaciones:

1. JUnit sirve para crear pruebas unitarias automatizadas.
2. Las clases de prueba deben llamarse IgualQueLaClaseOriginalTest.
3. JUnit solo sirve para probar código Java.
4. Las pruebas se crean en métodos anotados con @Test.
5. JUnit permite aislar el código a probar de dependencias externas.
6. assertEquals() compara resultados esperados vs actuales.

Tiempo🕐: *10 minutos*

Aplicando la teoría

A investigar: 🙌

En los próximos 15 minutos investiguen sobre las siguientes anotaciones y su función principal en los test. Tendremos una puesta en común al finalizar el ejercicio!

1. @Test
2. @Before
3. @After
4. @BeforeClass
5. @AfterClass
6. @Ignore

Tiempo🕐: *15 minutos*

Casos de Prueba

Caso de Prueba

¿Qué son?:

Son unidades de trabajo que se utilizan para verificar el comportamiento de una parte específica de un programa. Representan situaciones de entrada, ejecución y resultados esperados de una función o método en el código.

Los casos de prueba son una parte fundamental del proceso de prueba unitaria y se utilizan para asegurarse de que las unidades de código, como clases o métodos, funcionen correctamente.

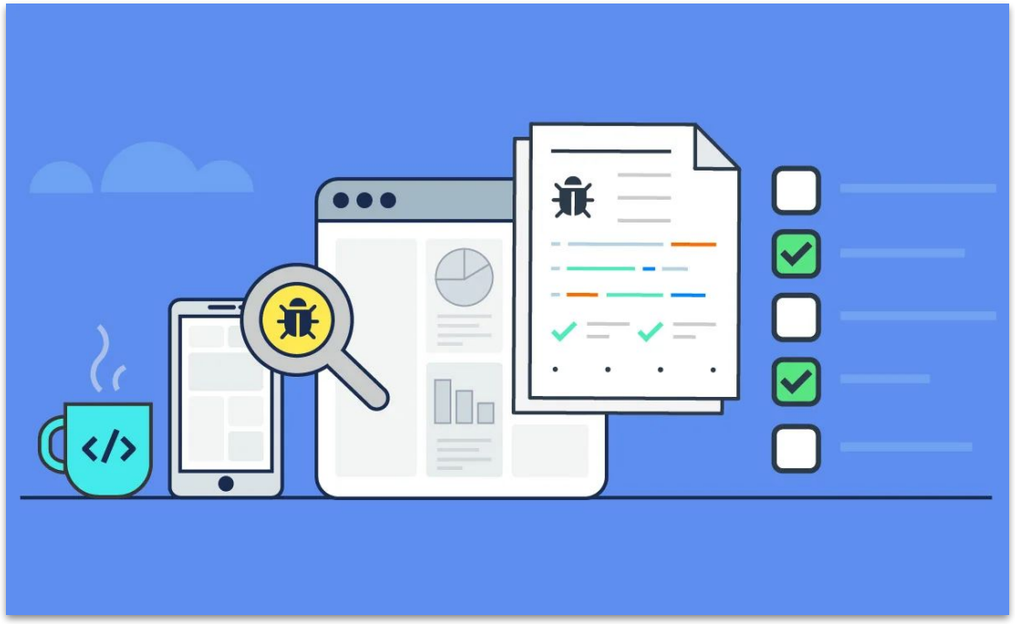
Caso de Prueba

Cobertura (coverage):

A la hora de crear los casos de prueba deberemos tener en cuenta que porcentaje de coverage o cobertura estableceremos:

- más del 70% es el valor mínimo aceptable
- entre el 85% y 95% es muy bueno
- más del 95% es excelente.

Una vez definido este valor, deberemos considerar todas las clases a cubrir con test y que casos posibles vamos a contemplar en los test.

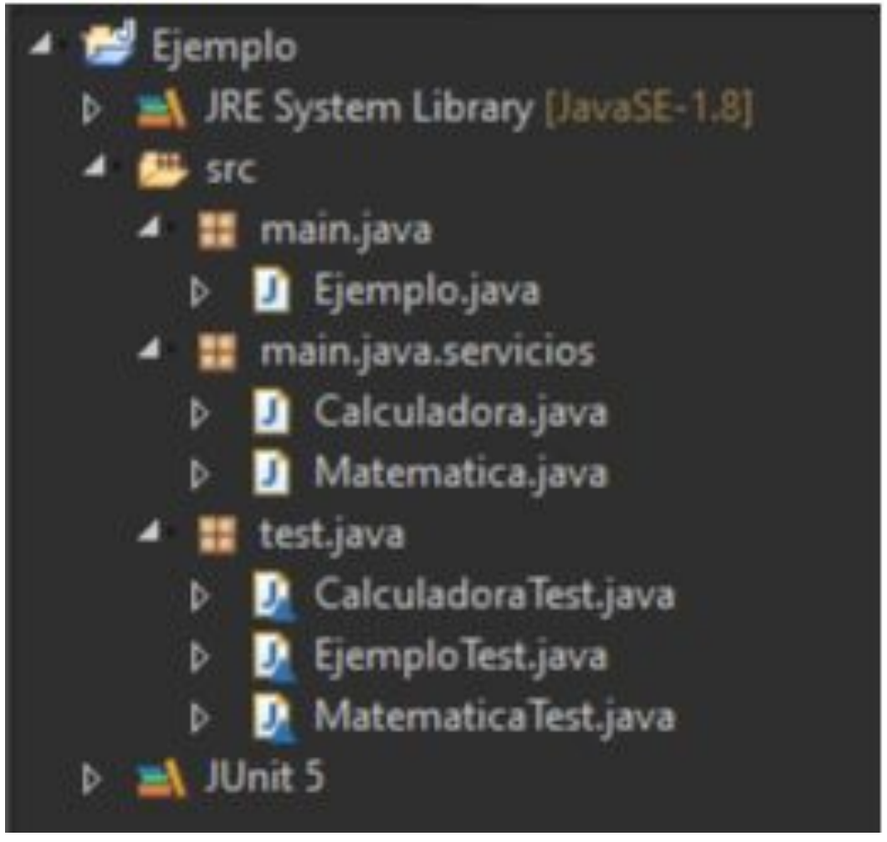


Caso de Prueba

Algunos criterios para decidir cómo organizar nuestros tests.

1. Tantas clases Test como Clases a testear:
   Se debe generar una ClaseTest por cada Clase que se desea testear.

   Un error común al iniciarse en la creación de los test unitarios es que dentro de una ClaseTest, busquemos testear tanto una Clase1 como una Clase2. Si deseamos testear ambas clases deberemos generar dos Clases Test distintas para cubrirlas.



Caso de Prueba

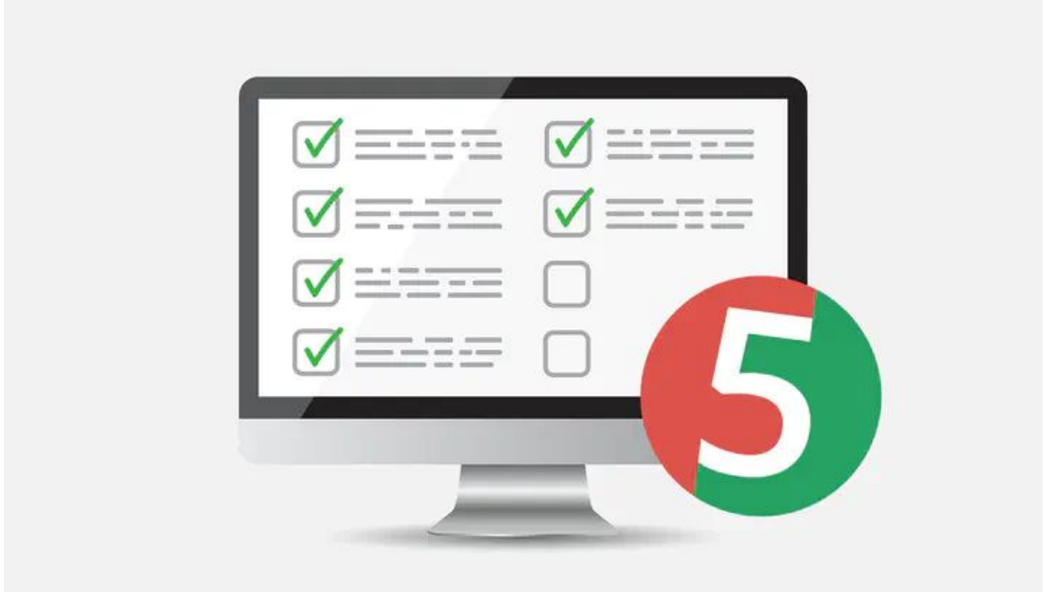
2. Funciones a testear: Si la Clase a Testear tiene cinco funciones, la ClaseTest deberá tener un mínimo de cinco @Test. Nunca se debe integrar en un @Test más de una función a testear.
Algo que sí es posible dentro de un @Test es intentar testear dos posibles salidas, un ejemplo de eso sería la siguiente función, parImpar, que puede devolver dos posibles valores (el @Test posee dos assertEquals).

```
public String parImpar(int num) {
    if (num % 2 == 0) {
        return "Par";
    } else {
        return "Impar";
    }
}
```

```
@Test
void parImpar() {
    String expected = "Par";
    String actual = this.matematica.parImpar(2);
    assertEquals(expected, actual);

     expected = "Impar";
     actual = this.matematica.parImpar(1);
    assertEquals(expected, actual);
}
```

Caso de Prueba

3. Casos de prueba: Una vez definida la clase que queremos cubrir con los test, se deberá entender que funciones es necesario cubrir, qué parámetros de entrada necesitan las mismas, que parámetros de salida tendrán y qué posibles errores pueden ocurrir al usarlas.



LIVE CODING

Probando: Vamos a realizar algunas pruebas para poner en práctica lo aprendido.

1. *Crear casos de prueba para depositar(), retirar() y getSaldo().*
2. *Explicar cómo se parametrizan las pruebas con diferentes casos y datos de entrada*

github
SOCIAL CODING

https://gist.github.com/Mariocanedo/713594081fd6b6617618592d7686448d

Tiempo: 20 Minutos

alkemy

Ejercicio N° 3

Poniendo a Prueba

Poniendo a Prueba

Practiquemos lo aprendido: En salas reducidas analizar lo siguiente 🙌

Imaginen que tenemos una clase de prueba CalculadoraTest y debemos pensar posibles casos de prueba para el método dividir().

Luego, en la sala principal compartir conclusiones según cada experiencia:

- ¿Qué tipo de pruebas crearían? ¿Por qué?

Tiempo🕐: *20 minutos*

Resumen

¿Qué logramos en esta clase?

- ✓ ***Comprender la definición y características de los Test Unitarios***
- ✓ ***Reconocer las ventajas y desventajas de los Test Unitarios***
- ✓ ***Reconocer las implementaciones de los casos de prueba y sus características***
- ✓ ***Comprender el concepto e implementación de la herramienta JUnit para Test Unitarios en Java***

¿Alguna consulta?

alkemy

¡Ponte a prueba!

Momento de ejercitación

Te invitamos a aprovechar esta última sección del espacio sincrónico para realizar de manera individual las ***actividades disponibles en la plataforma.*** *Estas propuestas son clave para afianzar lo trabajado y* ***forman parte obligatoria del recorrido de aprendizaje.***

💪 Análisis de caso ———— 💪 Selección Múltiple

💪 Comprensión lectora

Si al resolverlas surge alguna duda, compartela o tráela al próximo encuentro sincrónico.

alkemy

< ¡Muchas gracias! >