



Pruebas unitarias y TDD

JUnit

Implementar una suite de pruebas unitarias en lenguaje Java utilizando JUnit para asegurar el buen funcionamiento de una pieza de software

- Unidad 1: Flujo, ciclos y métodos
- Unidad 2: Arreglos y archivos
- Unidad 3: Programación orientada a objetos
- Unidad 4: Pruebas unitarias y TDD



Te encuentras aquí



¿Qué aprenderás en esta sesión?

- *Conocer las anotaciones de JUnit para saber cuándo aplicarlas.*
- *Desarrollar pruebas unitarias en un proyecto Java usando características de JUnit.*

¿Qué es JUnit?



`/* JUnit */`

JUnit

- En su versión más reciente, JUnit ha proveído de características que ayudan a integrar pruebas mediante Java y otros tipos de bibliotecas.
- Escribir pruebas unitarias con JUnit convierte la tarea en una experiencia agradable.
- Es importante destacar que JUnit es parte fundamental en el día a día de varios/as programadores/as, ya que nos sirve para comprobar códigos sin alterar significativamente el proceso final de aplicación.
- Es por esto, que a continuación veremos los detalles más importantes del cómo aplicar e implementar JUnit en Eclipse.

Antes de empezar a escribir las pruebas

Anotaciones

- @Test
- @DisplayName
- @BeforeAll
- @BeforeEach
- @AfterEach
- @AfterAll

Desde **org.mockito.Mockito** se importa el método estático `mock`, el cual crea un objeto simulado dada una clase o una interface.

Antes de empezar a escribir las pruebas

Loggers

Existen múltiples librerías que realizan este trabajo, pero con su propio log Java proporciona la capacidad de capturar los archivos del registro.

- Registro de circunstancias inusuales o errores que puedan estar ocurriendo en el programa.
- Obtener la información sobre qué está pasando en la aplicación.
- Los detalles que se obtienen de los registros varían. A veces es posible que se requieran muchos detalles respecto al problema o solo información sencilla.

Antes de empezar a escribir las pruebas

Afirmaciones

- Son métodos de utilidad para respaldar las condiciones en las pruebas.
- Son accesibles a través de la clase Assertions y se pueden usar directamente con `Assertions.assertEquals("", "")`
- Se leen mejor si se hace referencia a ellos mediante la importación estática, por ejemplo:

```
import static org.junit.Assert.assertEquals;  
assertEquals("OK", respuestaEsperadaQueDebeSerOK);
```

Dos de las afirmaciones más comunes:

- `assertEquals`
- `assertNotNull`

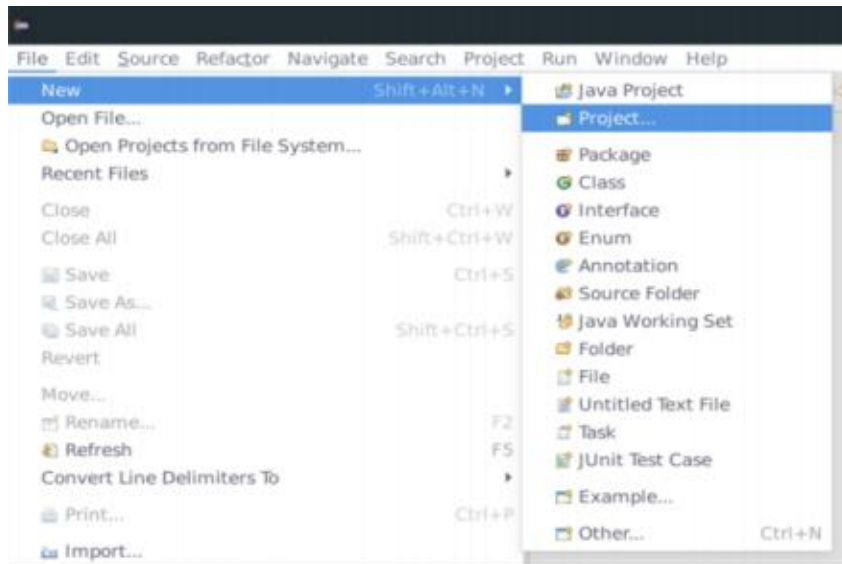
Implementación JUnit



Implementación JUnit

Paso 1

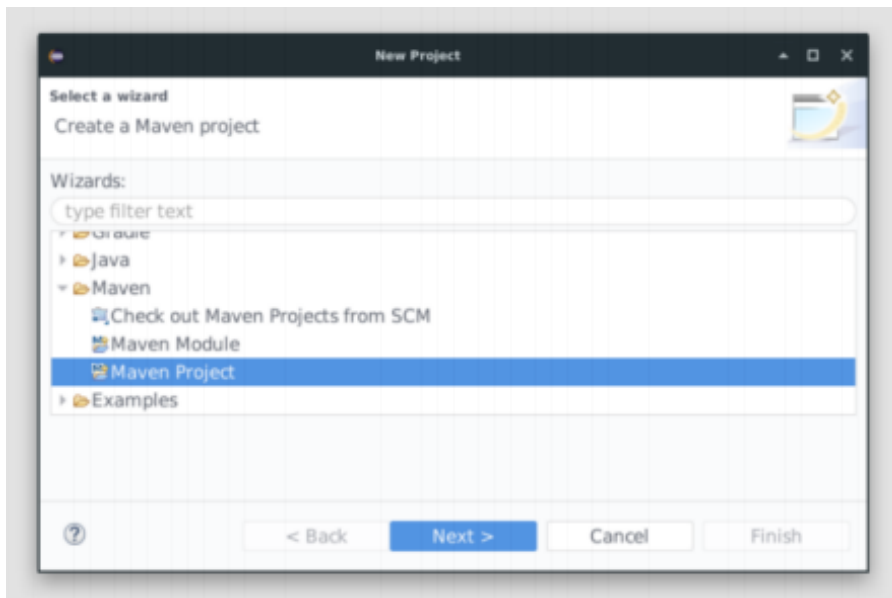
Crear un nuevo proyecto mediante File -> New -> Project.



Implementación JUnit

Paso 2

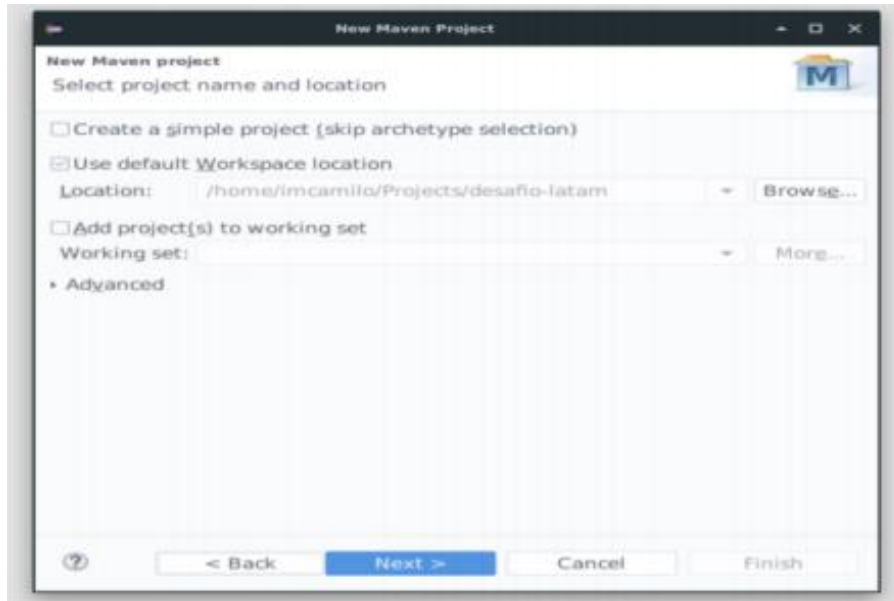
Con esto se abrirá una ventana para asistir la creación del nuevo proyecto. Buscar la carpeta llamada Maven y hacer clic sobre aquella que diga Maven Project.



Implementación JUnit

Paso 3

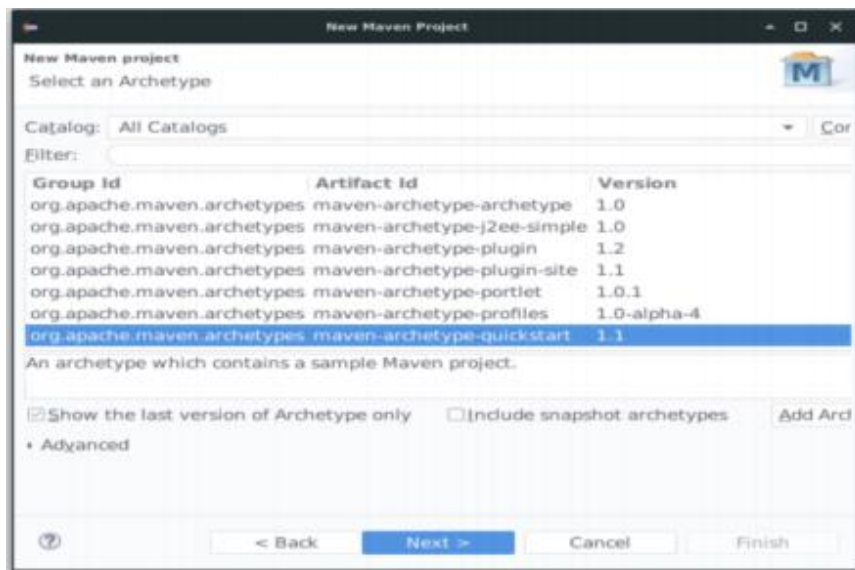
Seleccionar la ubicación del espacio de trabajo.



Implementación JUnit

Paso 4

Una vez seleccionado el espacio de trabajo, seleccionar un template (arquetipo) para el nuevo proyecto. El template maven-archetype-quickstart contiene lo necesario para iniciar un nuevo proyecto Maven, como se muestra a continuación:



Implementación JUnit

Paso 5

Definimos los parámetros del template: Group Id y Artifact Id

New Maven Project

Specify Archetype parameters

Group Id: cl.desafiolatam

Artifact Id: gs-testing

Version: 0.0.1-SNAPSHOT

Package: cl.desafiolatam

Properties available from archetype:

| Name | Value |
|------|-------|
|------|-------|

Advanced

< Back Next > Cancel Finish

Implementación JUnit

Paso 6

Modificar el archivo pom.xml para configurar maven.compiler.source y maven.compiler.target, dentro del tag properties, a continuación el detalle:

```
<properties>

    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

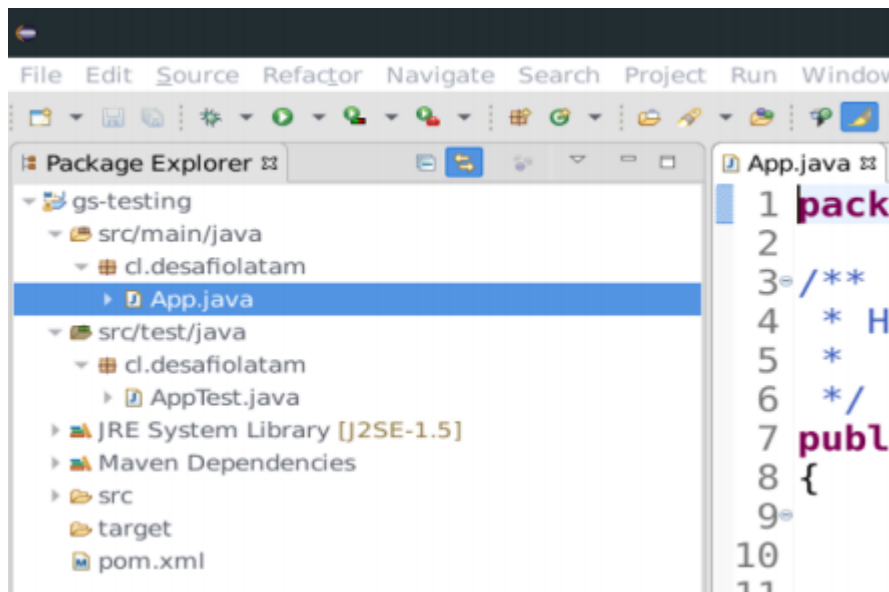
    <maven.compiler.source>1.8</maven.compiler.source>

    <maven.compiler.target>1.8</maven.compiler.target>

</properties>
```


Implementación JUnit

Finalmente, el proyecto ya está generado y se puede navegar a través de las carpetas.



Iniciando JUnit



Iniciando JUnit

Para empezar a utilizar JUnit debemos agregarlo al proyecto como dependencia adicional mediante sistemas de compilación como Gradle o Maven. La dependencia que viene por defecto al crear un proyecto Java es:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
```

Iniciando JUnit

Sin embargo, se debe ir al archivo pom.xml en la raíz del proyecto, se borra la dependencia de JUnit que viene por defecto y se agrega la nueva dentro del tag dependencies. El archivo pom.xml sería el siguiente:

{desafío}
latam_

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
  <project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
    <modelVersion>4.0.0</modelVersion>
    <groupId>cl.desafiolatam</groupId>
    <artifactId>gs-testing</artifactId>
    <version>1.0-SNAPSHOT</version>
```

```
    <name>gs-testing</name>
    <!-- FIXME change it to the project's website -->
    <url>http://www.example.com</url>
```

```
    <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    </properties>
```

```
    <dependencies>
      <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.4.2</version>
        <scope>test</scope>
      </dependency>
    </dependencies>
```

```
    <!--resto del archivo-->
  </project>
```

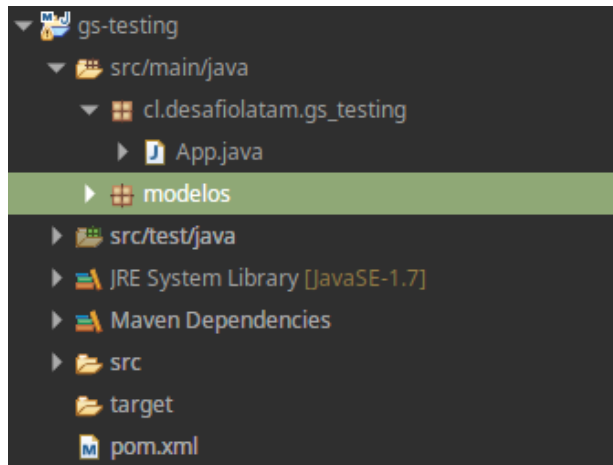
Ejercicio guiado



Pruebas

Paso 1

Agregar un nuevo package llamado modelos dentro del proyecto gs-testing.



Pruebas

Paso 2

Crear la clase Persona dentro del proyecto gs-testing y ubicarla en la carpeta src/main

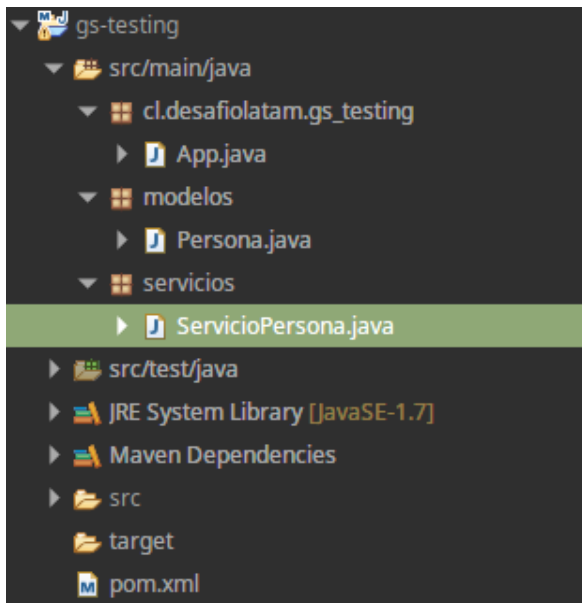
```
package modelos;
public class Persona {
    private String rut;
    private String nombre;
    public Persona(String rut, String nombre) {
        super();
        this.rut = rut;
        this.nombre = nombre;
    }
    public String getRut() {
        return rut;
    }
    public void setRut(String rut) {
        this.rut = rut;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```



Pruebas

Paso 3

Ahora, crear la clase llamada ServicioPersona en un nuevo package llamado servicios.



Pruebas

Paso 4

Al interior de la clase ServicioPersona, crear el método crearPersona(), este recibe un objeto de tipo Persona, el cual se encargará de guardarlos en el mapa llamado personasDB.

```
import modelos.Persona;
import java.util.HashMap;
import java.util.Map;
public class ServicioPersona {
    private Map<String,String> personasDB = new HashMap<>();
    public String crearPersona(Persona persona) {
        if (persona != null) {
            personasDB.put(persona.getRut(), persona.getNombre());
            return "Creada";
        } else {
            return "No creada";
        }
    }
}
```



Pruebas

Paso 5

Crear el método `actualizarPersona()` que realiza una tarea similar a `crearPersona()`, validando que el dato de entrada sea distinto de nulo y actualizando el valor de `personasDB`.

```
import modelos.Persona;
import java.util.HashMap;
import java.util.Map;
public class ServicioPersona {

    private Map<String,String> personasDB = new HashMap<>();
    public String crearPersona(Persona persona) {
        if (persona != null) {
            personasDB.put(persona.getRut(), persona.getNombre());
            return "Creada";
        } else {
            return "No creada";
        }
    }

    public String actualizarPersona(Persona persona) {
        if (persona != null) {
            personasDB.put(persona.getRut(), persona.getNombre());
            return "Actualizada";
        } else {
            return "No actualizada";
        }
    }
}
```

Pruebas

Paso 6

Agregar el método `listarPersonas()` que retorna `personasDB`, el cual es el mapa que se utiliza como almacén de datos.

```
import modelos.Persona;
import java.util.HashMap;
import java.util.Map;
public class ServicioPersona {

    private Map<String,String> personasDB = new HashMap<>();
    public String crearPersona(Persona persona) {
        if (persona != null) {
            personasDB.put(persona.getRut(), persona.getNombre());
            return "Creada";
        } else {
            return "No creada";
        }
    }

    public String actualizarPersona(Persona persona) {
        if (persona != null) {
            personasDB.put(persona.getRut(), persona.getNombre());
            return "Actualizada";
        } else {
            return "No actualizada";
        }
    }

    public Map<String, String> listarPersonas() {
        return personasDB;
    }
}
```

Pruebas

Paso 7

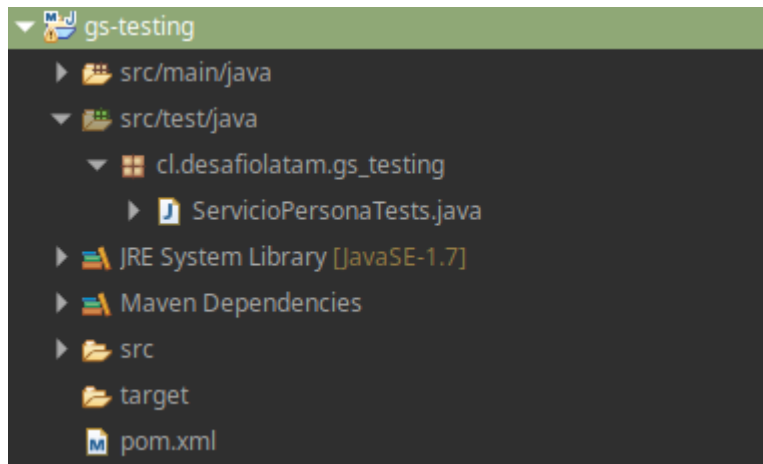
Agregar el método `eliminarPersona()` que recibe un parámetro de tipo `Persona`, valida si es nulo y procede a eliminar la persona que contenga la clave dentro del mapa `personasDB`, retornando "Eliminada".

```
import modelos.Persona;
import java.util.HashMap;
import java.util.Map;
public class ServicioPersona {
    private Map<String,String> personasDB = new HashMap<>();
    public String crearPersona(Persona persona) {
        if (persona != null) {
            personasDB.put(persona.getRut(), persona.getNombre());
            return "Creada";
        } else {
            return "No creada";
        }
    }
    public String actualizarPersona(Persona persona) {
        if (persona != null) {
            personasDB.put(persona.getRut(), persona.getNombre());
            return "Actualizada";
        } else {
            return "No actualizada";
        }
    }
    public Map<String, String> listarPersonas() {
        return personasDB;
    }
    public String eliminarPersona(Persona persona) {
        if (persona != null) {
            personasDB.remove(persona.getRut());
            return "Eliminada";
        } else {
            return "No eliminada";
        }
    }
}
```

Pruebas

Paso 8

Crear la clase ServicioPersonaTest dentro de la carpeta src/test del proyecto, como se muestra a continuación:



Pruebas

Paso 8

Esta será la clase que contendrá las pruebas de ServicioPersona, inicialmente se instancia la clase ServicioPersona. Además, se crea un logger para registrar los eventos de forma descriptiva. Para hacer que la prueba sea más legible y expresiva se agrega @DisplayName.

```
package cl.desafiolatam.gs_testing;
import org.junit.jupiter.api.*;
import java.util.logging.Logger;
@DisplayName("Tests Clase ServicioPersona")
public class ServicioPersonaTest {
    private static Logger logger =
        Logger.getLogger("cl.desafiolatam.servicios.ServicioPersonaTest");
    private final ServicioPersona servicioPersona = new ServicioPersona();
}
```



Pruebas

Paso 9

Crear el método
llamado
testCrearPersona
para el método
crearPersona(),
además de sus
respectivas
anotaciones

```
import cl.desafiolatam.gs_testing;
import org.junit.jupiter.api.*;
import java.util.logging.Logger;
import static org.junit.jupiter.api.Assertions.assertEquals;
import servicios.ServicioPersona;
import modelos.Persona;
import org.junit.Test;

public class ServicioPersonaTest {
    private static Logger logger =
        Logger.getLogger("cl.desafiolatam.servicios.ServicioPersonaTest");    private
    final ServicioPersona servicioPersona = new ServicioPersona();

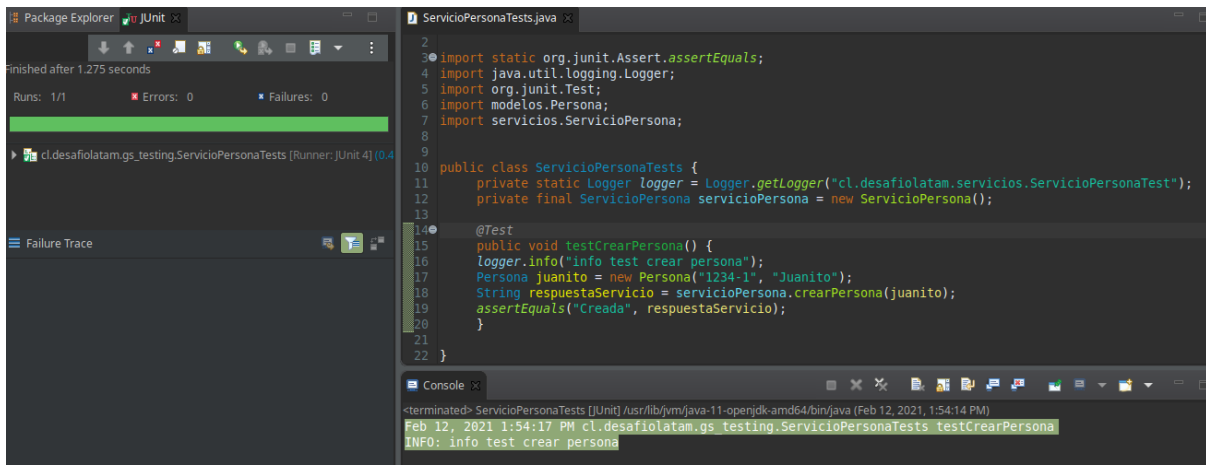
    @Test
    public void testCrearPersona() {
        logger.info("info test crear persona");
        Persona juanito = new Persona("1234-1", "Juanito");
        String respuestaServicio = servicioPersona.crearPersona(juanito);
        assertEquals("Creada", respuestaServicio);
    }
}
```

Pruebas

Paso 9

Para ejecutar la prueba `testCrearPersona`, se debe aplicar el botón Run As -> JUnit Test

Además, nos aparecerá una pantalla con una barra de color verde. Esto demuestra que el test ha salido con éxito.



The screenshot displays an IDE interface with three main panels. The top-left panel shows the Package Explorer with a green progress bar indicating a successful test run. The top-right panel shows the source code for `ServicioPersonaTests.java`, which includes imports for JUnit and logging, and a `@Test` method `testCrearPersona()` that creates a `Persona` object and calls `servicioPersona.crearPersona()`. The bottom panel shows the Console output, which includes the message `INFO: info test crear persona`.

```
Package Explorer | JUnit |
Finished after 1.275 seconds
Runs: 1/1 | Errors: 0 | Failures: 0
cl.desafiolatam.gs_testing.ServicioPersonaTests [Runner: JUnit 4] (0.4)
Failure Trace
ServicioPersonaTests.java
import static org.junit.Assert.assertEquals;
import java.util.logging.Logger;
import org.junit.Test;
import modelos.Persona;
import servicios.ServicioPersona;

public class ServicioPersonaTests {
    private static Logger logger = Logger.getLogger("cl.desafiolatam.servicios.ServicioPersonaTest");
    private final ServicioPersona servicioPersona = new ServicioPersona();

    @Test
    public void testCrearPersona() {
        logger.info("info test crear persona");
        Persona juanito = new Persona("1234-1", "Juanito");
        String respuestaServicio = servicioPersona.crearPersona(juanito);
        assertEquals("Creada", respuestaServicio);
    }
}

Console
<terminated> ServicioPersonaTests [JUnit] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Feb 12, 2021, 1:54:14 PM)
Feb 12, 2021 1:54:17 PM cl.desafiolatam.gs_testing.ServicioPersonaTests testCrearPersona
INFO: info test crear persona
```


Pruebas

Paso 10

Crear el método de prueba `testActualizarPersona` para el método `actualizarPersona()`, además de sus respectivas anotaciones.

```
import cl.desafiolatam.gs-testing;
import org.junit.jupiter.api.*;
import java.util.logging.Logger;
import static org.junit.Assert.assertEquals;
import servicios.ServicioPersona;
import modelos.Persona;
import org.junit.Test;

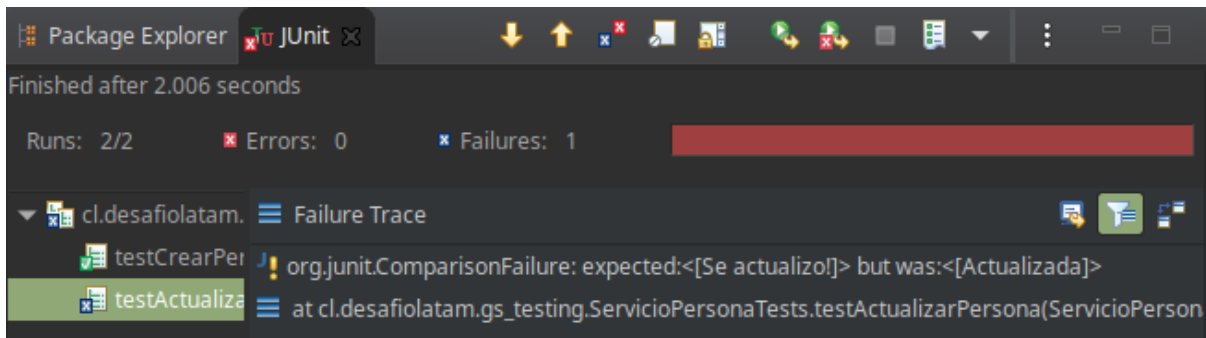
public class ServicioPersonaTest {
    private static Logger logger =
        Logger.getLogger("cl.desafiolatam.servicios.ServicioPersonaTest");
    private final ServicioPersona servicioPersona = new ServicioPersona();

    @Test
    public void testActualizarPersona() {
        logger.info("info actualizar persona");
        Persona pepe = new Persona("1234-1", "Pepe");
        String respuestaServicio = servicioPersona.actualizarPersona(pepe);
        assertEquals("Se actualizo", respuestaServicio);
    }
}
```

Pruebas

Paso 10

Al ejecutar `mvn test` se observa `AssertionFailedError`, y se detalla que `testActualizarPersona` falla en la línea 34, donde se espera "Se actualizó", pero se obtuvo "Actualizada".



Pruebas

Paso 10

Esto ocurre porque el método `actualizarPersona` retorna el String "Actualizada" y se está comparando con otra respuesta. Se observa que la prueba detona las características del método, pero falla en la aserción. Se debe corregir la prueba para comprobar si la salida es correcta.

```
import cl.desafiolatam.gs-testing;
import org.junit.jupiter.api.*;
import java.util.logging.Logger;
import static org.junit.Assert.assertEquals;
import servicios.ServicioPersona;
import modelos.Persona;
import org.junit.Test;
public class ServicioPersonaTest {
    private static Logger logger =
        Logger.getLogger("cl.desafiolatam.servicios.ServicioPersonaTest"); private final ServicioPersona
        servicioPersona = new ServicioPersona();

    @Test

    public void testActualizarPersona() {
        logger.info("info actualizar persona");
        Persona pepe = new Persona("1234-1", "Pepe");
        String respuestaServicio = servicioPersona.actualizarPersona(pepe); assertEquals("Actualizada",
        respuestaServicio);
    }
}
```

`/* TestFixtures */`

TestFixture

Si existen pruebas que tienen necesidades parecidas o sus características son iguales, estas características se pueden agrupar en una **TestFixture** o, en términos más simples, escribiendo las tareas en la misma clase con el objetivo de reutilizar código y eliminar código duplicado.

De esta forma, al estar en la misma clase, se pueden empezar a crear métodos que todas las pruebas puedan consumir. JUnit brinda anotaciones útiles que se pueden usar para reutilizar código, facilitar su desarrollo y claridad para inicializar objetos.

@BeforeAll

Se utiliza para indicar que el método anotado debe ejecutarse antes de todas las pruebas, el cual puede ser utilizado para inicializar objetos, preparación de datos de entrada o simular objetos para la prueba. Además, los métodos deben tener un tipo de retorno nulo, no deben ser privados y deben ser estáticos por defecto.

```
package cl.desafiolatam.servicios;
import org.junit.jupiter.api.*;
//imports

@DisplayName("Tests Clase ServicioPersona")
public class ServicioPersonaTest {

    @BeforeAll
    static void setup() {
        logger.info("Inicio clase de prueba");
    }
    //resto de la clase
}
```

@BeforeEach

Se utiliza para indicar que el método anotado debe ejecutarse antes de cada método que esté anotado con @Test en la clase de prueba actual, puede utilizarse para inicializar o simular objetos específicos para cada prueba.

Los métodos @BeforeEach deben tener un tipo de retorno nulo, no deben ser privados y no deben ser estáticos.

```
package cl.desafiolatam.servicios;
import org.junit.jupiter.api.*;
//imports

    @DisplayName("Tests Clase
ServicioPersona")
    public class ServicioPersonaTest {

        @BeforeEach
        void init() {
            logger.info("Inicio metodo de prueba");
        }
        //resto de la clase
    }
```

@AfterEach

Se usa para indicar que el método anotado debe ejecutarse después de cada método anotado con @Test en la clase de prueba actual.

Los métodos @AfterEach deben tener un tipo de retorno nulo, no deben ser privados y no deben ser estáticos.

```
package cl.desafiolatam.servicios;
import org.junit.jupiter.api.*;
//imports

@DisplayName("Tests Clase ServicioPersona")
public class ServicioPersonaTest {

    @AfterEach
    void tearDown() {
        logger.info("Metodo de prueba finalizado");
    }
    //resto de la clase
}
```


@AfterAll

Se utiliza para indicar que el método anotado debe ejecutarse después de todas las pruebas en la clase de prueba actual, donde es idóneo liberar los objetos creados.

Los métodos @AfterAll deben tener un tipo de retorno nulo, no deben ser privados y deben ser estáticos por defecto.

```
package cl.desafiolatam.servicios;
import org.junit.jupiter.api.*;
//imports
@DisplayName("Tests Clase ServicioPersona")
public class ServicioPersonaTest {

    @AfterAll
    static void done() {
        logger.info("Fin clase de prueba");
    }
    //resto de la clase
}
```

Test para el método listarPersona()

```
public class ServicioPersonaTest {  
    private static Logger logger =  
Logger.getLogger("cl.desafiolatam.servicios.ServicioPersonaTest");  
    private final ServicioPersona servicioPersona = new ServicioPersona();  
  
    @Test  
  
    public void testListarPersona() {  
        logger.info("info listar persona");  
        Map<String, String> listaPersonas = servicioPersona.listarPersonas();  
        assertNotNull(listaPersonas);  
    }  
}
```

¿Qué hace un
TestFixture?



¿Cuáles son las anotaciones TestFixtures?





Próxima sesión...

- *Guia de ejercicios*

{desafío}
latam_

*Academia de
talentos digitales*

