

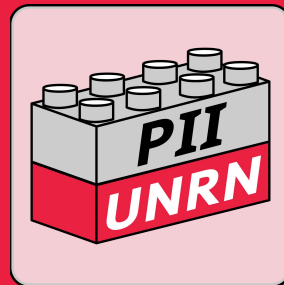
# Pruebas unitarias

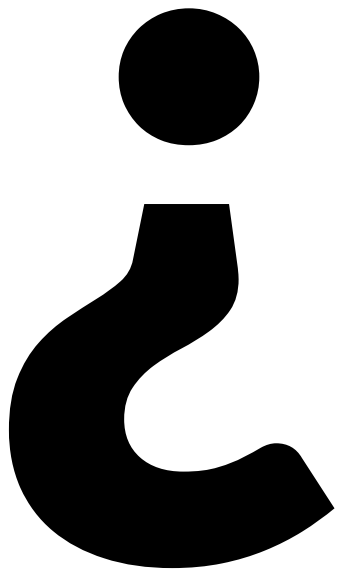
**UNRN**

Universidad Nacional  
de Río Negro

**IV**

**2024**





**Han revisado  
el TP2**



**Casi todos los  
ejercicios  
ya estan  
resueltos**

# De Programación 1





**¿Preguntas?**





o

**Abran  
hilo**

---

# Probando software

# **El camino hacia el desarrollo profesional de software**



**Y dejarnos de  
“codear”**

**Dando garantías de que  
el programa hace lo que  
se supone que debe**

# Haciendo pruebas

# Pruebas, de caja blanca / caja negra

# Caja blanca



# Usando

# la estructura del código

# Usando

# la documentación

**Usando**

**la estructura  
del código**

**la  
documentación**

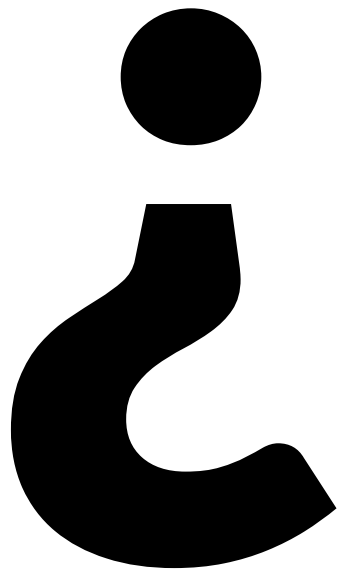
---

# ¿Que valores son tentadores para probar?

## Suma lenta de dígitos

Implementa una función que calcule la suma de los dígitos de un número entero positivo pasado como parámetro.

```
int sumaDigitos(int numero)
```



**Que valores  
fueron elegidos  
en sumaLenta**





# Suma Lenta

*La consigna*

Escribir una función que haga la suma entre dos números enteros sin hacer la operación de manera directa. Hacer sumas o restas de a uno.

# Otro ejemplo

---

# ¿Qué valores son tentadores para probar?

```
bool estaAprobado(int nota){  
    bool resultado = false;  
    if (nota > 4){  
        resultado = true;  
    }  
    return resultado;  
}
```



---

# ¿Qué valores son tentadores para probar?

```
bool estaAprobado(int nota){  
    bool resultado = false;  
    if (nota > 4){  
        resultado = true;  
    }  
    return resultado;  
}
```

- **claramente desaprobado**
- **desaprobado por poco**
- **aprobado con lo justo**
- **bien aprobado**
- **"súper" aprobado**

---

# ¿Que valores son tentadores para probar?

```
bool estaAprobado(int nota){  
    bool resultado = false;  
    if (nota > 4){  
        resultado = true;  
    }  
    return resultado;  
}
```

**¿Que podemos ver con los valores anteriores?**





**La especificación  
del problema  
juega fuerte.**

**Es necesario  
reforzar**

**Pero las consignas son  
bastante vagas**

**Pensar en  
precondiciones y  
postcondiciones ayuda a  
pensar la prueba**

---

# Caja negra

Esto es lo que *eventualmente*  
hará Conan

**Dada una entrada,  
¿qué es lo que sale?**



**Suele ser a un nivel  
externo, simulando  
el uso que tendría el  
programa**

**Y no una 'unidad' del  
mismo**

**Sin razonar  
puntualmente  
sobre como lo  
logra**

# En síntesis

**Ejercitar el código y ver  
cuando **no** obtenemos un  
resultado correcto**

# ¿Qué son las pruebas unitarias?

**1**

**Son un tipo de  
prueba de caja  
blanca**

**2**

**Realizadas por el  
desarrollador**

**3**

**Permite conocer si  
algún cambio *rompe*  
algo en otro lado**



# ¿Por que?

**Es mucho más fácil  
construir sobre  
código probado**



# **Facilita las modificaciones de código**



**¿Preguntas?**

# ¿Que es una 'unidad'?

**Es una delimitación  
de que será  
probado.**

# Desde funciones individuales

**Hacía funciones que  
hacen uso de las  
anteriores**



**No suelen utilizar  
recursos externos**

**Lo que probamos**  
***generalmente* no va a**  
**usar**  
**STDIN/STDOUT**

**Una buena razón  
por la cual  
printf / Scanner  
no van en las  
funciones**



**¿Preguntas?**

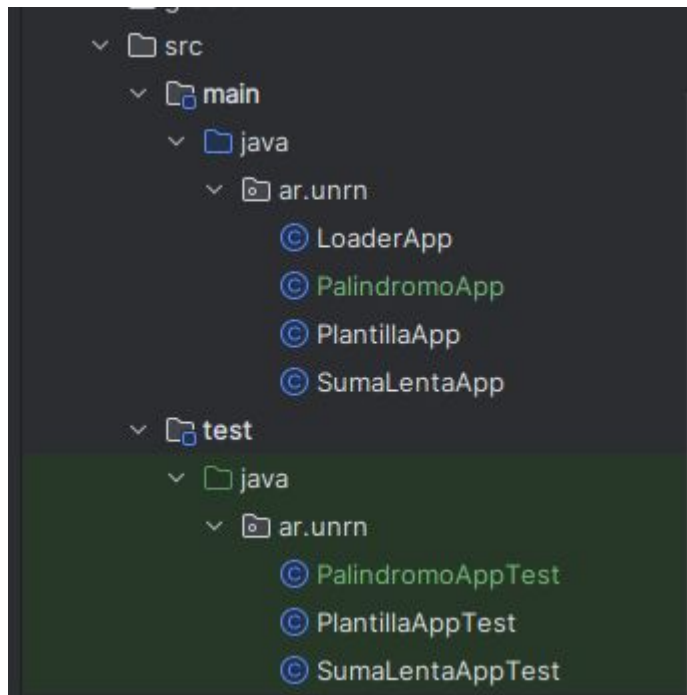
---

# Práctica planificada

La practica para esta semana, es armar tests para código y aumentar la cobertura.

El criterio de auto-corrección es el porcentaje de cobertura y la cantidad de casos de prueba.

# Testing base



El Código

Sus tests

que incluyen los  
casos

0x000D

*Cuestiones de estilo*

**Debe tener el  
mismo nombre que  
la `class` con `Test` al  
final**



# **La anatomía de un caso de prueba**

# La estructura de un caso de prueba JUnit 5

```
@test
@DisplayName("Suma positiva")
public void test_suma_positivos(){
    int argumento1 = 10;
    int argumento2 = 20;
    int esperado = 30;
    int resultado = SumaApp.suma_lenta(argumento1, argumento2);
    assertEquals(esperado, resultado, "no coincide");
}
```

Para saber que caso es

La llamada a la  
unidad

Entre más explícito, mejor

Verificamos el resultado\*

# La estructura de un test individual debe ser:

**preparar / ejecutar / comprobar / limpiar**  
**Con identificación y mensajes**

# La estructura de un caso de prueba JUnit 5

```
@test
@DisplayName("Sumamos positivos")
public void test_suma_positivos(){
    int argumento1 = 10;
    int argumento2 = 20;
    int esperado = 30;
    int resultado = SumaApp.suma_lenta(argumento1, argumento2);
    assertEquals(esperado, resultado, "no coincide");
    //limpiar en este caso no es necesario
}
```

Identificación

Preparar

ejecutar

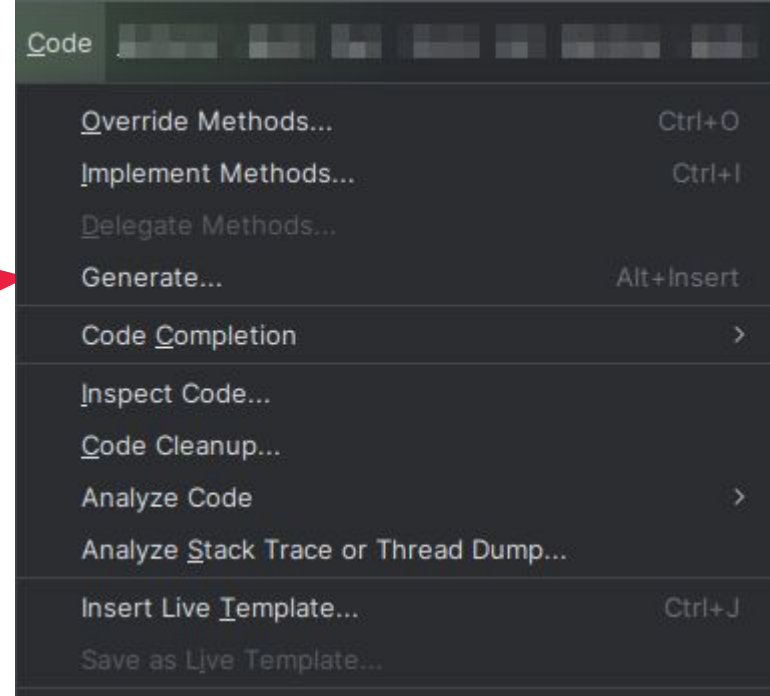
comprobar

**El código no suele  
ser más  
complicado que  
eso.**

**Tenemos una**  
*minima* **ayudita de**  
**IntelliJ IDEA**

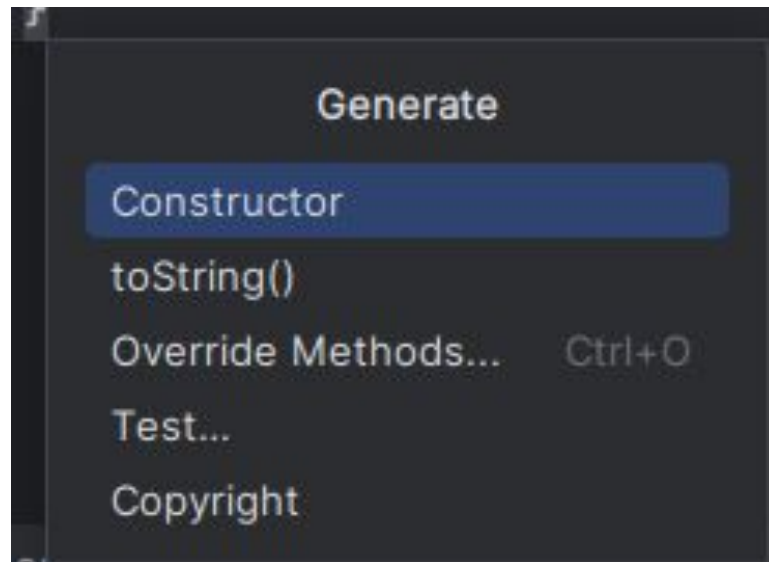
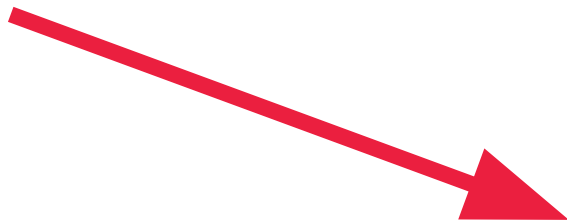
En una class  
de 'src'

1:



**Generamos  
casos**

**2:**





Para las  
funciones  
que  
indiquemos

3:



**Create Test**

Testing library: JUnit5

Class name: SumaLentaAppTest

Superclass:

Destination package: ar.unrn

Generate: ☐ setUp/@Before ☐ tearDown/@After

Generate test methods for: ☐ Show inherited methods

Member	
<input type="checkbox"/>	main(args:String[]):void
<input type="checkbox"/>	suma(sumandoUno:int, sumandoDos:int):int

? OK Cancel

# El encabezado

```
package unrn.programacion;  
  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.*;
```

# Nos da el Test, como esqueleto

**(la plantilla  
es más  
completa)**

```
class SumaLentaAppTest {  
  
    @Test  
    void suma() {  
    }  
}
```

# Un assert 'genérico'

Lo que se supone que sale

Lo que se sale

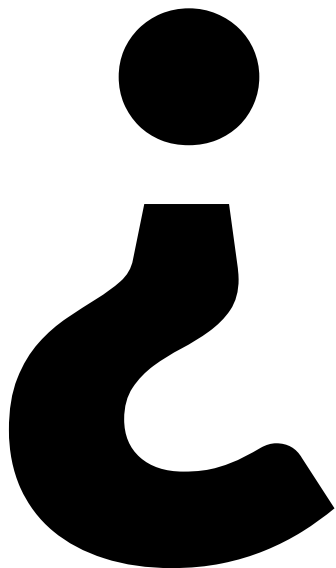
```
assert??(esperado, recibido, mensaje);
```

Cuando no obtenemos lo esperado



**El message en sí es opcional.**

**Describir cuál es la idea del caso  
de prueba no.**



**Que asserts  
tenemos**



# Tipos de assert

```
assertEquals(primitivo expected, primitivo actual, String message)
```

```
assertEquals(float expected, float actual, float delta, String message)
```

```
assertTrue(boolean condition, String message)
```

```
assertFalse(boolean condition, String message)
```

```
fail(String message)
```

# assertEquals

```
@Test
public void igualdadTest() {
    String esperado = "UNRN";
    String resultado = "UNRN";

    assertEquals(esperado, resultado, "mensaje");
}
```



# assertEquals - con margen de error para decimales

```
@Test
public void igualdadTest() {
    float esperado = 3.1416f;
    float resultado = 3.14f;
    float delta = 2f;
    assertEquals(esperado, resultado, delta);
}
```

0x000F

*Cuestiones de estilo*

# Una llamada a función en cada caso de prueba\*\*

\*salvo que estén *fuertemente* conectadas  
como en DivisionLenta

# assertNull / assertNotNull

```
@Test
public void testNull() {
    Object cosa = null;

    assertNull("la cosa debe ser null", null);
}

@Test
public void testNotNull() {
    Gato michi = new Gato();

    assertNotNull("El gato debe de existir!", michi);
}
```

# assertSame / assertNotSame

```
@Test
public void whenAssertingNotSameObject_thenDifferent() {
    Object michi = new Gato();
    Object pichichu = new Perro();

    assertNotSame(michi, pichichu);
}
```

# assertTrue / assertFalse

```
@Test
public void testBooleano() {
    assertTrue(5 > 4, "5 es mayor que 4");
    assertFalse(5 > 6, "5 no es mayor que 6");
}
```



**¿Preguntas?**

---

# Probando nuestro código

**UNRN**

Universidad Nacional  
de Río Negro

# Suma de dígitos de un número

```
unsigned long long int suma_digitos(long long int numero)
{
    long long int suma = 0;
    numero = valor_absoluto(numero);
    while (numero > 0)
    {
        suma = suma + (numero % 10);
        numero = numero / 10;
    }
    return suma;
}
```



# Su documentación

```
/**  
 * Calcula la suma de los dígitos de un número natural.  
 *  
 * @param número que se calculará la suma de sus dígitos.  
 * @return La suma de los dígitos del número.  
 *  
 * @pre El número debe ser un natural, los números negativos  
serán tratados como positivos.  
 * @post El valor devuelto es la suma de los dígitos del  
número, el signo será siempre positivo.  
 */
```

# iA la terminal!



# La estructura de una función de pruebas (junit)

```
@test
public void test_suma_positivos(){
    int argumento1 = 10;
    int argumento2 = 20;
    int esperado = 30; // argumento1 + argumento2 esta bien
    int resultado = suma_lenta(argumento1, argumento2);
    assertEquals(esperado, resultado, "no coincide");
}
```

**unrn.edu.ar**

**UNRN**

Universidad Nacional  
de **Río Negro**



| **unrionegro**