CAPÍTULO 3

ACTIVIDAD 3.1

Modifica el método *resta()* de la clase *Calculadora* y añade los métodos *resta2()* y *divide2()* que se exponen a continuación. Crea después los test para probar los 3 métodos. Los métodos son:

**public int resta()** {

int resul;

if(resta2())resul = num1 - num2;

else resul= num2- num1;

return resul;

}

**public boolean resta2()** {

if (num1 >= num2) return true;

else return false;

}

**public Integer divide2()** {

if(num2==0) return null;

int resul = num1 / num2;

return resul;

}

Utiliza los métodos *assertTrue(), assertFalse(), assertNull(), assertNotNull()* o *assertEquals()* según convenga.

@Test

**public** **void** testResta() {

Calculadora calcu = **new** Calculadora(5, 10);

**int** resultado = calcu.resta();

*assertEquals*(5, resultado);

}

@Test

**public** **void** testResta2() {

Calculadora calcu = **new** Calculadora(20, 30);

**boolean** resultado = calcu.resta2();

*assertTrue*("No es true" ,resultado);//,

}

@Test

**public** **void** testDivide2() {

Calculadora calcu = **new** Calculadora(20, 0);

Integer resultado = calcu.divide2();

*assertNull*("No es nulo" ,resultado);//,

}

ACTIVIDAD 3.2

Realiza pruebas parametrizadas para los métodos *suma(), resta()* y *multiplica()* de la clase Calculadora.

Carpeta 03-ACTIVIDAD 3-2

COMPRUEBA TU APRENDIZAJE

1. ¿Qué estrategias se siguen para probar el software? Si las pruebas de unidad funcionan, ¿es necesario hacer la prueba de integración?

Epígrafe 3.3

1. Considérese una aplicación bancaria, donde el usuario puede conectarse al banco a través de Internet y realizar una serie de operaciones bancarias. Una vez que ha accedido al banco con las consiguientes medidas de seguridad puede realizar una serie de operaciones. La operación que se va a gestionar requiere la siguiente entrada:

* *Código del banco:* puede estar en blanco o puede ser un número de 3 dígitos. En este último caso, el primero de ellos tiene que ser mayor que 1.
* *Código de sucursal*: número de cuatro dígitos. El primero de ellos mayor de 0.
* *Número de cuenta*: número de cinco dígitos.
* *Clave personal*: valor alfanumérico de cinco posiciones.
* *Orden*: puede estar en blanco o ser uno de los valores siguientes: “Talonario” o “Movimientos”.

El programa responde de la siguiente manera:

* Si *Orden* tiene el valor “Talonario”, el usuario recibe un talonario de cheques.
* Si *Orden* tiene el valor “Movimientos”, el usuario recibe los movimientos del mes en curso.
* Si *Orden* está en blanco, el usuario recibe los dos documentos.
* Si ocurre algún error en la entrada de datos, el programa muestra un mensaje de error sobre el dato implicado.

Se pide definir las clases de equivalencia, casos de prueba válidos y casos de prueba no válidos que cubran una sola clase no válida.

Fichero **03-SoluciónEjecicio2.pdf**, obtenido de la siguiente URL:

<http://www.lsi.us.es/docencia/get.php?id=401>

1. Rellena en la siguiente tabla los casos de prueba tomando como referencia las reglas del análisis de valores límite:

|  |  |
| --- | --- |
| **Condiciones de entrada y de salida** | **Casos de prueba** |
| Una variable toma valores comprendidos entre -4 y 4 (enteros) | Para valores: -4, 4, -3, 5 |
| El programa lee un fichero que contiene de 1 a 100 registros | Para 1, 100, 0, 101 registros |
| El programa deberá generar de 1 a 5 listados | Para 1, 5, 0, 6 listados |
| El número de alumnos para calcular la nota media es 35 | Para 0, 1, 35 y 36 |
| La función deberá devolver un array de enteros, de 1 a 10 valores. | Para 1, 10, 0 y 11 enteros. |

Realiza después un programa Java para probar la función que devuelve el array de enteros. Utiliza los casos de prueba que hayas definido.

**public** **class** Ejemplo {

**public** **static** **void** main(String[] args) {

//generar un array de n enteros

//casos de prueba para n=0, 1, 10 y 11

**int** n=1;

**int** [] tabla =*enteros*(n);

**try** {

System.*out*.println("Número de elementos: " +

tabla.length);

**for**(**int** i=0;i<n; i++)

System.*out*.println(tabla[i]);

}

**catch**(NullPointerException e){

System.*out*.println("Array fuera de los límites");

}

}

//devuelve un array de n enteros, máximo 10 y mínimo 1

//si el array supera unos límites, la función devuelve nulo

**static** **int** [] enteros(**int** n){

**if** (n<1 || n>10)

**return** **null**;

**int**[] tabla = **new** **int**[n];

**int** j=0;

**for**(**int** i=10; j<n; i++, j++)

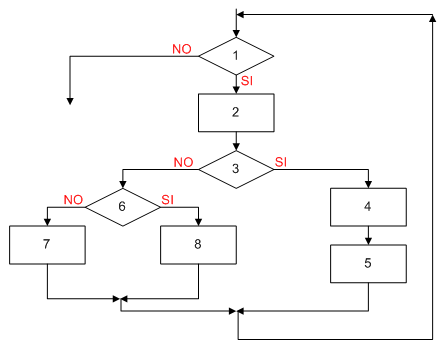
tabla[j]=i;

**return** tabla;

}

}

1. A partir del diagrama de flujo mostrado en la Figura 3.29 construye el grafo de flujo. Indica el número de nodos, aristas, regiones, nodos predicado, la complejidad ciclomática y el conjunto de caminos independientes.



**Figura 3.29.** Ejercicio 4.



Grafo de flujo: 

Nº de aristas 11, Nº de nodos 9, nodos predicado 3, Nº de regiones 4.

V(G) es 4, tenemos 4 caminos básicos. Sin mucho riesgo

Camino 1: 1 – 11

Camino 2: 1- 2,3 – 4,5 – 10 – 1 – 11

Camino 3: 1- 2,3 – 6 – 8 - 9 - 10 – 1 – 11

Camino 4: 1- 2,3 – 6 – 7 - 9 - 10 – 1 – 11

1. Realiza el grafo de flujo, calcula la complejidad ciclomática, define el conjunto básico de caminos, elabora los casos de prueba para cada camino y evalúa el riesgo para la siguiente función Java:

static int Contador1(int x, int y) {

Scanner entrada = new Scanner(System.in);

int num, c = 0;

if (x > 0 && y > 0) {

System.out.println("Escribe un número");

num = entrada.nextInt();

if (num >= x && num <= y){

System.out.println("\tNúmero en el rango");

c++;

}

else

System.out.println("\tNúmero fuera de rango");

}

else

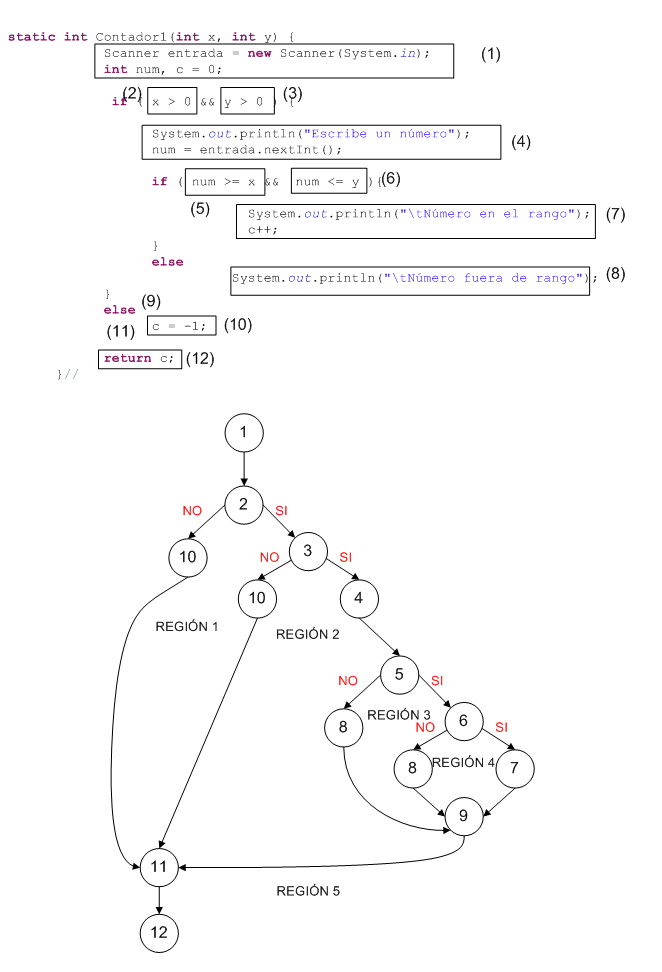
c = -1;

return c;

}//

Nº de aristas 17, Nº de nodos 12, nodos predicado 4, Nº de regiones 5.

V(G) es 5, tenemos 5 caminos básicos. Sin mucho riesgo



|  |  |  |
| --- | --- | --- |
| **Camino** | **Caso de prueba** | **Resultado esperado** |
| Camino 1:  1-2-10-11-12 | Escoger algún X tal que NO se cumpla la condición X>0 -  X= -1 - Contador1(-1,10) | C=-1  Devuelve -1 |
| Camino 2:  1-2-3-10-11-12 | Escoger algún X e Y tal que se cumpla la condición X > 0 y NO se cumpla Y>0  X=1, Y=-10  Contador1(1,10) | C=-1  Devuelve -1 |
| Camino 3:  1-2-3-4-5-6-7-9-11-12 | Escoger algún X e Y tal que se cumpla la condición X>0 e Y>0  Escoger algún num >=X y num<= Y  X=1, Y=10, num=5  Contador1(1,10) | Número en el rango  C=1  Devuelve 1 |
| Camino 4:  1-2-3-4-5-6-8-9-11-12 | Escoger algún X e Y tal que se cumpla la condición X>0 e Y>0  Escoger algún num >=X y num > Y  X=1, Y=10, num=15  Contador1(1,10) | Número fuera de rango  C=0  Devuelve 0 |
| Camino 5:  1-2-3-4-5-8-9-11-12 | Escoger algún X e Y tal que se cumpla la condición X>0 e Y>0  Escoger algún num <X  X=1, Y=10, num=-15  Contador1(1,10) | Número fuera de rango  C=0  Devuelve 0 |

1. Desarrolla una batería de pruebas para probar el método *DevuelveFecha()* de la clase *Fecha* que se expone a continuación. El método recibe un número entero y devuelve un String con un formato de fecha que dependerá del valor de dicho número. Si el número recibido es distinto de 1, 2 y 3 el método devuelve ERROR. La clase es la siguiente:

import java.text.SimpleDateFormat;

import java.util.Date;

**public class Fecha** {

SimpleDateFormat formato;

Date hoy;

**public Fecha()** {

hoy = new Date();

}

**public String DevuelveFecha(int tipo)** {

String cad = "";

switch (tipo) {

case 1: {

formato = new SimpleDateFormat("yyyy/MM");

cad = formato.format(hoy);

break;

}

case 2: {

formato = new SimpleDateFormat("MM/yyyy");

cad = formato.format(hoy);

break;

}

case 3: {

formato = new SimpleDateFormat("MM/yy");

cad = formato.format(hoy);

break;

}

default: {

cad = "ERROR";

}

}

return cad;

}

}//

**POSIBLE SOLUCIÓN:**

import static org.junit.Assert.\*;

import java.util.Arrays;

import java.util.Collection;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.junit.runners.Parameterized;

import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)

public class FechaTest {

private int tipo;

private String resul;

public FechaTest(int tipo, String resul) {

this.tipo = tipo;

this.resul=resul;

}

@Parameters

public static Collection<Object[]> numeros() {

return Arrays.asList(new Object[][] {

{ 1, "2014/03" },

{ 2, "03/2014" },

{ 3, "03/14"} ,

{ 3, "2014/03" },

{ 1, "03/2014" },

{ 2, "03/14"},

{ 4, "ERROR"},

{ 5, null}

});

}

@Test

public void testDevuelveFecha()

{

Fecha fec = new Fecha();

assertTrue( fec.DevuelveFecha(tipo).equals(resul) );

}

}

1. Escribe una clase de pruebas para probar el método *calculo()* de la clase *Factorial*. En el método se comprueba si el número es menor que 0, en este caso se lanza la excepción *IllegalArgumentException* con el mensaje *Número n no puede ser < 0*. Si el valor del factorial calculado es menor que 0 es que ha ocurrido un error de desbordamiento, en este caso se lanza la excepción *ArithmeticException* y se lanza el mensaje *Overflow, número n demasiado grande*. La clase a probar es la siguiente:

**public class Factorial** {

**public static int calculo(int n)** {

if (n < 0 ) {

throw new IllegalArgumentException("Número "+ n +

" no puede ser < 0");

}

int fact = 1;

for (int i = 2; i <= n; i++)

fact \*= i;

if (fact < 0 ) {

throw new ArithmeticException("Overflow, número "+

n +" demasiado grande");

}

return fact;

}

}//

**POSIBLE SOLUCIÓN:**

**import** **static** org.junit.Assert.\*;

**import** org.junit.Test;

**public** **class** FactorialTest {

@Test

**public** **void** testCalculo() {

**int** n = Factorial.*calculo*(12);

*assertEquals*(479001600, n);

}

@Test(expected = java.lang.IllegalArgumentException.**class**)

**public** **void** testCalculoExcep1() {

Factorial.*calculo*(-22);

Factorial.*calculo*(-1);

}

@Test(expected = java.lang.ArithmeticException.**class**)

**public** **void** testCalculoExcep2() {

Factorial.*calculo*(18);

}

}

1. Se trata de implementar la entrada de datos desde una web para la petición de un CD de música. El cliente introduce el código del CD y una cantidad. El código deberá comenzar por un dígito, seguido de un guión y 7 dígitos más. Sólo se podrá pedir un máximo de 3 CDs por cliente. Si el código del CD no está en el catálogo se produce un error (ER1) y si la cantidad solicitada supera el stock de dicho CD se produce otro error (ER2). En el caso de que todo vaya bien se enviará el CD al solicitante con el precio mostrado en la web, salvo que tenga oferta; en este caso se le aplicará al precio la oferta.

Realiza la tabla con las condiciones de entrada y las clases de equivalencia válidas y no válidas. Muestra varios casos de prueba, en los que se valide todo correctamente y que den error y no se lleve a cabo el pedido, en este caso indica por qué no se lleva a cabo el pedido.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Condición de entrada** | **Clases de equivalencia** | **Clases Válidas** | **COD** | **Clases no Válidas** | **COD** |
| **Código** | Valor | Cualquier nº con este formato:  9-9999999 | V1 | A-1234567  1-AB12345 | NV1  NV2 |
| **Cantidad** | Valor | 1<=cantidad<=3 | V2 | Cantidad =0  Cantidad=4 | NV3  NV4 |
| **Código en catálogo?** | Lógica | Si está en catálogo | V3 | No está en catálogo | NV5 |
| **Cantidad supera stock?** | Lógica | NO supera el stock | V4 | Si supera el stock | NV6 |
| **Oferta** | Miembro de un conjunto | Oferta =”SI” | V5 | Oferta =”<” | NV7 |
| Oferta = “NO” | V6 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **CASO DE PRUEBA** | **Clases de equivalencia** | **CONDICIONES DE ENTRADA** | | | | | **Resultado esperado** |
| **Codigo** | **Cantidad** | **En catálogo?** | **Supera stock?** | **Oferta** |
| **CP1** | V1, V2, V3, V4, V5 | 2-3456789 | 1 | SI | NO | SI | Envio CD con oferta |
| **CP2** | V1, V2, V3, V4, V6 | 2-3456789 | 2 | SI | NO | NO | Envio CD sin oferta |
| **CP3** | V1, V2, NV5, V4, V5 | 2-3456789 | 2 | NO | NO | SI | ER1 |
| **CP4** | V1, V2, NV6, V3, V5 | 2-3456789 | 2 | SI | SI | SI | ER2 |
| **CP5** | NV1, V2, V3, V4, V5 | A-1234567 | 1 | SI | NO | SI | Codigo incorrecto |
| **CP6** | V1, NV3, V3, V4, V6 | 2-3456789 | 0 | SI | NO | NO | Cantidad incorrecta |
| **CP7** | V1, V2, V3, V4, NV7 | 2-3456789 | 2 | SI | NO | < | Oferta incorrecta |
| **….** | ………….. | | | | | | |

ACTIVIDADES DE AMPLIACIÓN

1. Realiza el grafo de flujo, calcula la complejidad ciclomática, define el conjunto básico de caminos, elabora los casos de prueba para cada camino y evalúa el riesgo para la siguiente función Java:

static int Contador2(int x, int y) {

Scanner entrada = new Scanner(System.in);

int num, c = 0;

if ( x > 0 && y > 0) {

System.out.println("Escribe un número");

num = entrada.nextInt();

while (num != 0) {

if ( num >= x && num <= y ) {

System.out.println("\tNúmero en el rango");

c++;

} else

System.out.println("\tNúmero fuera de rango");

System.out.println("Escribe un número");

num = entrada.nextInt();

}//fin while

}

else

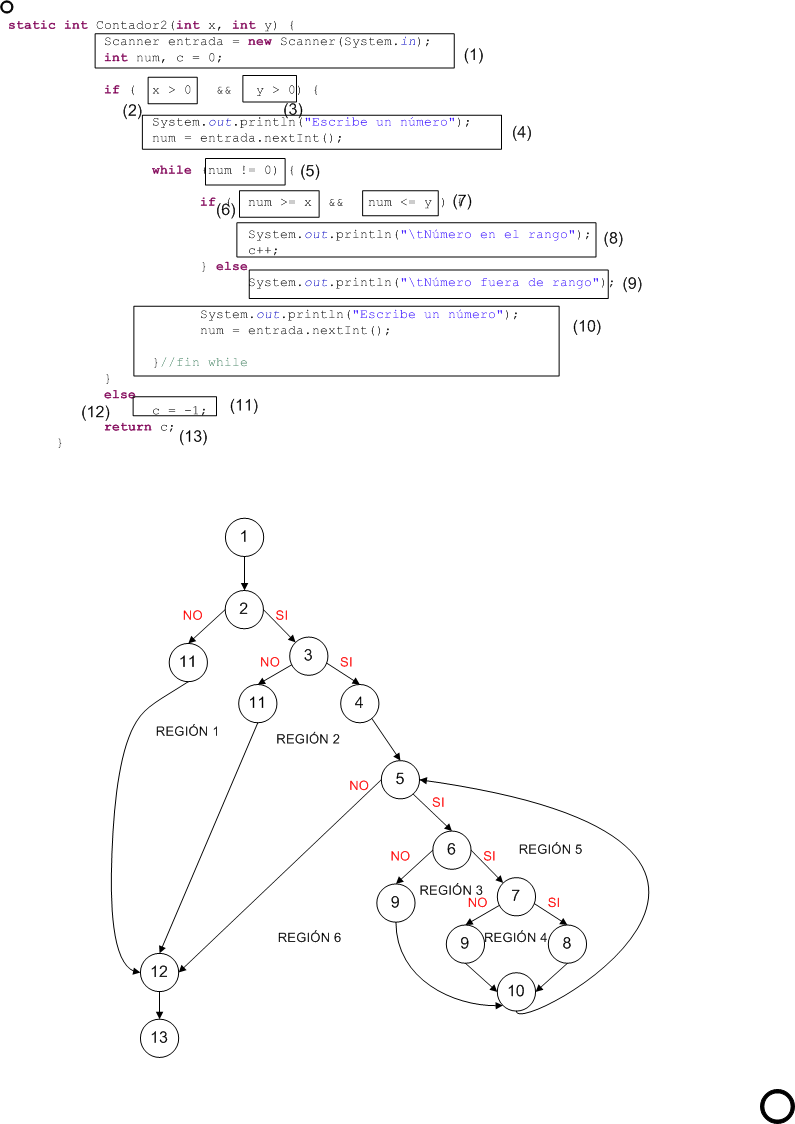
c = -1;

return c;

}

Nº de aristas 19, Nº de nodos 15, nodos predicado 5, Nº de regiones 6.

V(G) es 6, tenemos 6 caminos básicos. Sin mucho riesgo.



|  |  |  |
| --- | --- | --- |
| **Camino** | **Caso de prueba** | **Resultado esperado** |
| Camino 1:  1-2-11-12-12 | Escoger algún X tal que NO se cumpla la condición X>0 -  X= -1 - Contador2(-1,10) | C=-1  Devuelve -1 |
| Camino 2:  1-2-3-11-12-13 | Escoger algún X e Y tal que se cumpla la condición X > 0 y NO se cumpla Y>0  X=1, Y=-10  Contador2(1,10) | C=-1  Devuelve -1 |
| Camino 3:  1-2-3-4-5-12-13 | Escoger algún X e Y tal que se cumpla la condición X>0 e Y>0  Escoger algún num tal que sea =0  X=1, Y=10, num=0  Contador2(1,10) | Sale del bucle  Devuelve C |
| Camino 4:  1-2-3-4-5-6-9-10-5 | Escoger algún X e Y tal que se cumpla la condición X>0 e Y>0  Escoger algún num NO >=X  X=1, Y=10, num=-5  Contador2(1,10) | Número fuera de rango  No suma 1 a C |
| Camino 5:  1-2-3-4-5-6-7-8-10-5 | Escoger algún X e Y tal que se cumpla la condición X>0 e Y>0  Escoger algún num >=X y num <= Y  X=1, Y=10, num=5  Contador2(1,10) | Número en el rango  suma 1 a C |
| Camino 6:  1-2-3-4-5-6-7-9-10-5 | Escoger algún X e Y tal que se cumpla la condición X>0 e Y>0  Escoger algún num >=X y num NO <= Y  X=1, Y=10, num=15  Contador2(1,10) | Número fuera de rango  No suma 1 a C |

Los caminos son:

Camino 1: 1-2-11-12-12

Camino 2: 1-2-3-11-12-13

Camino 3: 1-2-3-4-5-12-13

Camino 4: 1-2-3-4-5-6-9-10-5-12-13

Camino 5: 1-2-3-4-5-6-7-8-10-5-12-13

Camino 6: 1-2-3-4-5-6-7-9-10-5-12-13

En los caminos 4, 5 y 6 no hace falta probar los nodos 5-12 y 13 ya que se prueban en los caminos anteriores.

En cada camino se trata de explorar una arista o nodo no explorado anteriormente.

1. Escribe una clase de pruebas para probar los métodos de la clase *TablaEnteros*. En esta clase de prueba crea un método con la anotación **@BeforeClass** en el que inicialices un array de enteros para usarlo en las pruebas de los métodos. El método *sumaTabla()* suma los elementos del array y devuelve la suma. El método *mayorTabla()* devuelve el elemento mayor de la tabla. Y el método *posicionTabla()* devuelve la posición ocupada por el elemento cuyo valor se envía.

En el constructor se comprueba si el número de elementos de la tabla es nulo o 0, en este caso se lanza la excepción *IllegalArgumentException* con el mensaje *No hay elementos*. El método *posiciónTabla* también lanza la excepción, *java.util.NoSuchElementException*, en el caso de que no se encuentre el elemento en la tabla. Hay que añadir otros dos métodos de prueba para probar estas excepciones. La clase a probar es la siguiente:

**public class TablaEnteros** {

private Integer[] tabla;

**TablaEnteros(Integer[] tabla)** {

if (tabla == null || tabla.length == 0)

throw new IllegalArgumentException("No hay elementos");

this.tabla = tabla;

}

//devuelve la suma de los elementos de la tabla

**public int sumaTabla()** {

int suma = 0;

for (int i = 0; i < tabla.length; i++)

suma += tabla[i];

return suma;

}

//devuelve el mayor elemento de la tabla

**public int mayorTabla()** {

int max = -999;

for (int i = 0; i < tabla.length; i++)

if (tabla[i] > max)

max = tabla[i];

return max;

}

//devuelve la posición de un elemento cuyo valor se pasa

**public int posicionTabla(int n)** {

for (int i = 0; i < tabla.length; i++)

if (tabla[i] == n)

return i;

throw new java.util.NoSuchElementException("No existe:" + n);

}

}//

**POSIBLE SOLUCIÓN:**

**import** **static** org.junit.Assert.\*;

**import** org.junit.BeforeClass;

**import** org.junit.Test;

**public** **class** TablaEnterosTest {

**private** **static** Integer[] *tabla*;

@BeforeClass

**public** **static** **void** inicioPruebas() {

*tabla* = **new** Integer[5];

**for** (Integer i = 0; i < 5; i++)

*tabla*[i] = i \* 30 + 3;

// 3,33,63,93,123

}

@Test(expected = java.lang.IllegalArgumentException.**class**)

**public** **void** testTablaEnterosNull() {

TablaEnteros T = **new** TablaEnteros(**null**);

}

@Test(expected = java.util.NoSuchElementException.**class**)

**public** **void** testPosicionTablaExcepcion() {

TablaEnteros T = **new** TablaEnteros(*tabla*);

*assertEquals*(1, T.posicionTabla(693));

}

@Test

**public** **void** testSumaTabla() {

TablaEnteros T = **new** TablaEnteros(*tabla*);

*assertEquals*(315, T.sumaTabla());

}

@Test

**public** **void** testMayorTabla() {

TablaEnteros T = **new** TablaEnteros(*tabla*);

*assertEquals*(123, T.mayorTabla());

}

@Test

**public** **void** testPosicionTabla() {

TablaEnteros T = **new** TablaEnteros(*tabla*);

*assertEquals*(3, T.posicionTabla(63));

}

}