

BLOQUE 4 : OBJETOS Y ESTRUCTURAS DE DATOS

```
1
2 package com.mycompany.empleados;
3
4 public class Empleado {
5     public String nombre;
6     public int edad;
7     public double salario;
8
9     public Empleado(String nombre, int edad, double salario) {
10         this.nombre = nombre;
11         this.edad = edad;
12         this.salario = salario;
13     }
14
15     public Empleado() {
16         this.nombre = "";
17         this.edad = -1;
18         this.salario = 0;
19     }
20 }
```

```
1
2
3 package com.mycompany.empleados;
4
5 public class Empleados {
6
7     public static void main(String[] args) {
8         // Creación de una estructura de datos para un empleado
9         Empleado empleado1 = new Empleado("Ruth" , 19 , 26530000);
10
11         // Acceso directo a los campos de la estructura de datos
12         System.out.println("Nombre: " + empleado1.nombre);
13         System.out.println("Edad: " + empleado1.edad);
14         System.out.println("Salario: " + empleado1.salario);
15
16         // Modificación directa del salario
17         empleado1.salario = 7500.000;
18         System.out.println("Nuevo salario: " + empleado1.salario);
19     }
20 }
```

En estas imágenes se ve como utiliza estructura de datos. Crea una clase que se llama Empleado la cual tiene tres atributos que son Nombre, Edad y Salario que los hace públicos, por lo que en el Main para acceder a los datos del empleado solo utilizamos la sentencia empleado1. Nombre si es lo que queremos que nos muestre. Todas las clases tienen que tener sus atributos privados para luego en el Main acceder a ellos a través de los getters y setters

Las siguientes imágenes serán el código ya modificado con los atributos privados

```

5     private String nombre;
6     private int edad;
7     private double salario;
8
9     public Empleado(String nombre, int edad, double salario) {
10         this.nombre = nombre;
11         this.edad = edad;
12         this.salario = salario;
13     }
14
15     public Empleado() {
16         this.nombre = "";
17         this.edad = -1;
18         this.salario = 0;
19     }
20
21     public String getNombre() {
22         return nombre;
23     }
24
25     public void setnombre(String nombre) {
26         this.nombre = nombre;
27     }
28
29     public int getEdad() {
30         return edad;
31     }
32
33     public void setedad(int edad) {
34         this.edad = edad;
35     }
36
37     public double getSalario() {
38         return salario;
39     }
40
41     public void setsalario(double salario) {
42         this.salario = salario;
43     }

```

```

package com.mycompany.empleados;

public class Empleados {

    public static void main(String[] args) {
        // Creación de una estructura de datos para un empleado
        Empleado empleadol = new Empleado("Ruth" , 19 , 26530000);

        // Acceso directo a los campos de la estructura de datos
        System.out.println("Nombre: " + empleadol.nombre);
        System.out.println("Edad: " + empleadol.edad);
        System.out.println("Salario: " + empleadol.salario);

        // Modificación directa del salario
        empleadol.salario = 7500.000;
        System.out.println("Nuevo salario: " + empleadol.salario);
    }
}

```

BLOQUE 5: MANEJO DE ERRORES

En estas imágenes que te pongo del código no está escrito ningún manejo de error. Es muy importante el manejo de errores para poder construir un software confiable, seguro y fácil de

mantener. El buen manejo de errores no solo mejora la experiencia del usuario, sino que también hace que el código se menos propenso a tener errores inesperados

```
package com.mycompany.empleados;

public class Empleados {

    public static void main(String[] args) {
        // Creación de una estructura de datos para un empleado
        Empleado empleadol = new Empleado("Ruth" , 19 , 26530000);

        // Acceso directo a los campos de la estructura de datos
        System.out.println("Nombre: " + empleadol.nombre);
        System.out.println("Edad: " + empleadol.edad);
        System.out.println("Salario: " + empleadol.salario);

        // Modificación directa del salario
        empleadol.salario = 7500.000;
        System.out.println("Nuevo salario: " + empleadol.salario);
    }
}
```

```
5 private String nombre;
6 private int edad;
7 private double salario;
8
9 public Empleado(String nombre, int edad, double salario) {
10     this.nombre = nombre;
11     this.edad = edad;
12     this.salario = salario;
13 }
14
15 public Empleado() {
16     this.nombre = "";
17     this.edad = -1;
18     this.salario = 0;
19 }
20
21 public String getNombre() {
22     return nombre;
23 }
24 public void setnombre(String nombre){
25     this.nombre = nombre;
26 }
27
28 public int getEdad() {
29     return edad;
30 }
31
32 public void setedad(int edad){
33     this.edad = edad;
34 }
35 public double getSalario(){
36     return salario;
37 }
38 public void setsalario(double salario){
39     this.salario = salario;
40 }
```

En el siguiente código podemos ver como usa excepciones en vez de códigos de retorno

En nuestro código podemos observar como el empleado tiene un atributo que es el sueldo, si nosotros metemos un numero negativo en el sueldo nos debería de dar un código de error, pero como usamos excepciones nos daría una excepción llamada 'IllegalArgumentException'

```

package com.mycompany.empleados;

public class Empleados {

    public static void main(String[] args) {
        // Creación de una estructura de datos para un empleado
        Empleado empleadol = new Empleado("Ruth" , 19 , 26530000);

        // Acceso directo a los campos de la estructura de datos
        System.out.println("Nombre: " + empleadol.nombre);
        System.out.println("Edad: " + empleadol.edad);
        System.out.println("Salario: " + empleadol.salario);

        // Modificación directa del salario
        empleadol.salario = 7500.000;
        System.out.println("Nuevo salario: " + empleadol.salario);
    }
}

```

Cuando se crea un manejo de errores se escribe un código try-catch-finally , por lo que este código dice que en cualquier momento salta al catch por lo que el programa tiene que estar atento de cuando eso ocurra seguir

En este código se ve cómo primero prueba a insertar un numero de sueldo negativo que es el -2800.00 y lo haces para provocar una excepción , al salir la excepción el código salta al catch por lo que hay te salta la excepción de `IllegalArgumentException`

```

package com.mycompany.empleados;

public class Empleados {

    public static void main(String[] args) {
        // Creación de una estructura de datos para un empleado
        Empleado empleadol = new Empleado("Ruth" , 19 , 26530000);

        // Accediendo a los atributos y métodos del objeto Empleado
        System.out.println("Nombre: " + empleadol.getNombre());
        System.out.println("Edad: " + empleadol.getEdad());
        System.out.println("Salario: " + empleadol.getSalario());


        // Modificación directa del salario
        empleadol.setSalario(7500.000);
        System.out.println("Nuevo salario: " + empleadol.getSalario());

        // Modificando el salario del empleado
        try {
            empleadol.setSalario(-2800.00); // Intente establecer un salario negativo para provocar una excepción
            System.out.println("Nuevo salario: " + empleadol.getSalario());
        } catch (IllegalArgumentException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}

```

BLOQUE 6: TEST UNITARIOS

Aquí se puede ver como en el código no hay ningún test unitario escrito



```
4  import org.junit.jupiter.api.AfterAll;
5  import org.junit.jupiter.api.BeforeEach;
6  import org.junit.jupiter.api.BeforeAll;
7  import org.junit.jupiter.api.Test;
8  import static org.junit.jupiter.api.Assertions.*;
9
10 /**
11  *
12  * @author ruthd
13  */
14 public class EmpleadosTest {
15
16     public EmpleadosTest() {
17     }
18
19     @BeforeAll
20     public static void setUpClass() {
21     }
22
23     @AfterAll
24     public static void tearDownClass() {
25     }
26
27     @BeforeEach
28     public void setUp() {
29     }
30
31     @AfterEach
32     public void tearDown() {
33     }
34
35 }
```

Ahora tenemos el código con test unitarios creados

```

import com.mycompany.empleados.Empleado;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class EmpleadosTest {

    public EmpleadosTest() {
    }

    @BeforeAll
    public static void setUpAll() {
        System.out.println("Esto es una configuracion inicial antes de ejecutar todas las pruebas");
    }

    @BeforeEach
    public void setUp() {
        System.out.println("Esto se inicia antes de ejecutar cada prueba");
    }

    @Test
    public void testConstructor() {
        Empleado empleado = new Empleado("Ruth", 30, 2500.00);
        assertEquals("Ruth", empleado.getNombre());
        assertEquals(30, empleado.getEdad());
        assertEquals(2500.00, empleado.getSalario(), 0.01);
    }

    @Test
    public void testSetSalario() {

```

```

        @Test
        public void testSetSalario() {
            Empleado empleado = new Empleado("Ruth", 30, 2500.00);

            // Probar establecer un salario positivo
            empleado.setSalario(2800.00);
            assertEquals(2800.00, empleado.getSalario(), 0.01);

            // Probar establecer un salario igual a cero
            empleado.setSalario(0.00);
            assertEquals(0.00, empleado.getSalario(), 0.01);
        }

        @AfterEach
        public void tearDown() {
            System.out.println("Este metodo realiza una limpieza despues de ejecutar cada una de las pruebas");
        }

        @AfterAll
        public static void tearDownAll() {
            System.out.println("Este metodo realiza una limpieza despues de ejecutar todas las pruebas de los test realizados");
        }
    }
}

```

En este código podemos ver como se crean dos Test Unitarios, el primero de ellos el TestConstructor que verifica si el constructor de 'Empleado' establece correctamente los atributos indicados en la clase

Y el segundo Test Unitario es el TestSetSalario que lo que hace este test es probar el método setsalario(double salario) para asegurarse de que se puede establecer un salario positivo correctamente y si se puede establecer un salario igual a cerro

BLOQUE 7: CLASES

Las clases tienen una organización ya que como en este proyecto solo tenemos una clase que es la de empleados, la clase en si es una clase publica, pero sus atributos son privados. Esta clase es pequeña ya que solo realiza un método que es el de validar si el salario que estamos introduciendo es válido o no. En esta clase de empleado el código sí que está preparado para realice algún cambio ya que nosotros mismos podemos crear un método nuevo y nuestro

código no se verá afectado y además también podemos añadir o eliminar alguno de los atributos privados que la clase no se va a ver afectada.

```
1 package com.mycompany.empleados;
2
3
4 public class Empleado {
5     private String nombre;
6     private int edad;
7     private double salario;
8
9     public Empleado(String nombre, int edad, double salario) {
10         this.nombre = nombre;
11         this.edad = edad;
12         this.salario = salario;
13     }
14
15     public Empleado() {
16         this.nombre = "";
17         this.edad = -1;
18         this.salario = 0;
19     }
20
21     public String getNombre() {
22         return nombre;
23     }
24
25     public void setnombre(String nombre){
26         this.nombre = nombre;
27     }
28
29     public int getEdad() {
30         return edad;
31     }
32
33     public void setedad(int edad){
34         this.edad = edad;
35     }
36
37     public double getSalario(){
38         return salario;
39     }
40
41     public void setsalario(double salario){
42         this.salario = salario;
43     }
44
45     public void setSalario(double salario) {
46         // Validar que el salario no sea negativo
47         if (salario < 0) {
48             throw new IllegalArgumentException("El salario no puede ser negativo");
49         }
50     }
51 }
```