

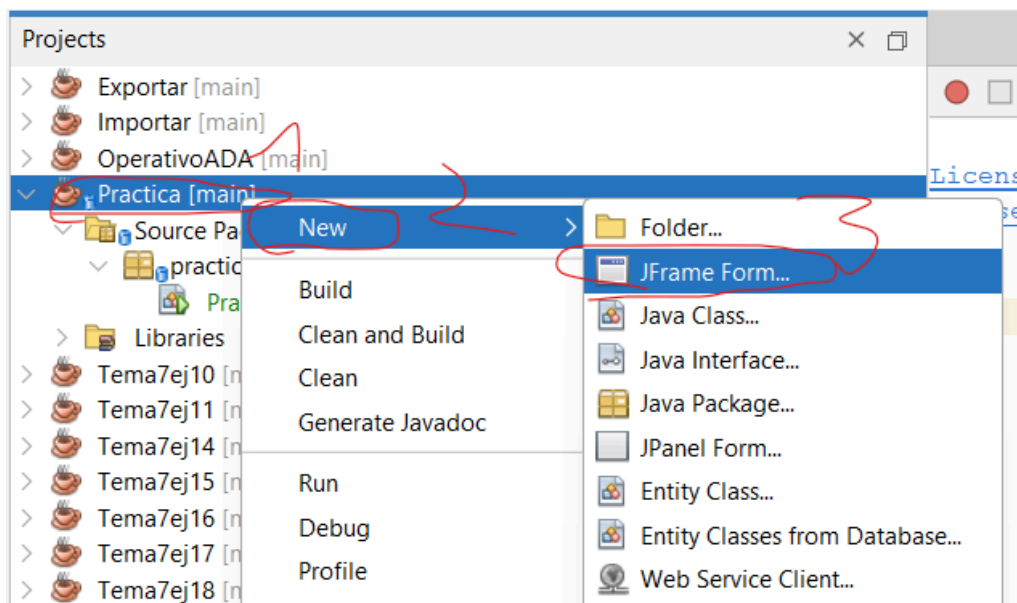
## Entornos de Desarrollo: Unidad 3 Práctica 1

### Prácticas con Java Swing

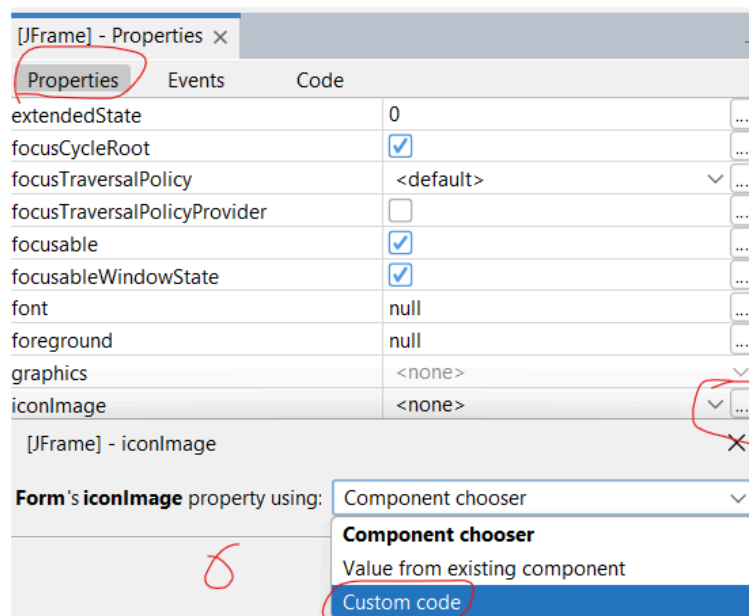
1. Práctica interfaz
2. Práctica calculadora dos datos
3. Práctica calculadora de cinco datos con porcentaje
4. Práctica calcular número aleatorio
5. Ejercicios interfaces
6. Anotaciones y JavaBeans

#### 1. Práctica interfaz

- 1) Click derecho en el proyecto
- 2) New
- 3) JFrame Form

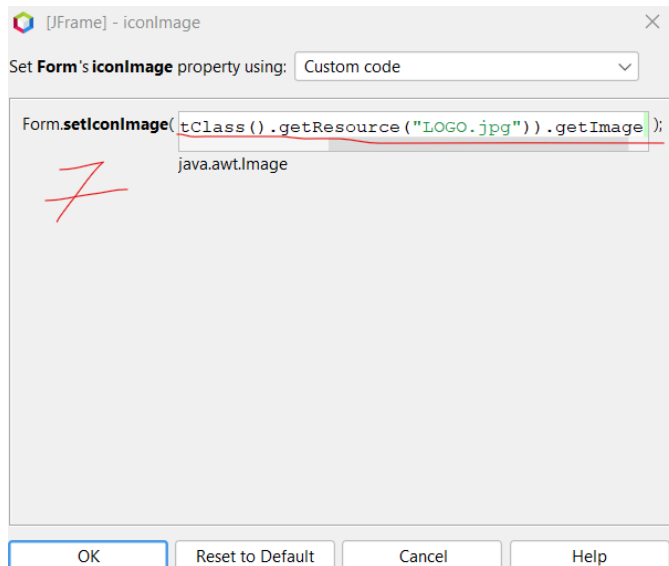


- 4) Properties
- 5) ImageIcon
- 6) Custom code



- 7) En `Form.setIconImage` copiamos **new**  
**`ImageIcon(this.getClass().getResource("LOGO.jpg")).getImage()`**

LOGO.jpg es un logo que nos hemos descargado de Internet y hemos insertado en los archivos del programa



- 8) Ahora hacemos el **import** de **`javax.swing.ImageIcon`**

`import javax.swing.ImageIcon;`

- 9) Hacemos visible con **`new JFrame().setVisible(true);`**

- 10) Ahora debería ser visible

```

}
//</editor-fold>

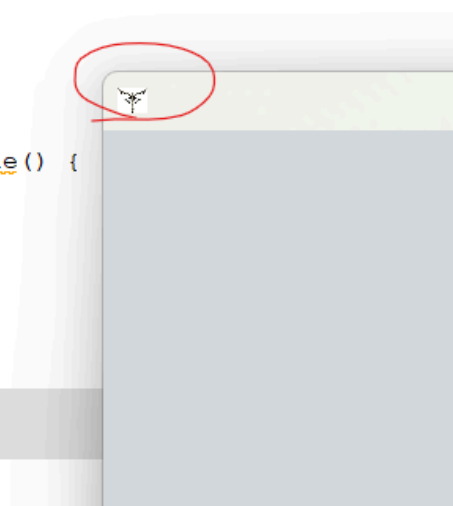
/* Create and display the form */
java.awt.EventQueue.invokeLater(new Runnable() {
    public void run() {
        new JFrame().setVisible(true);
    }
});
}

```

```

// Variables declaration - do not modify
// End of variables declaration

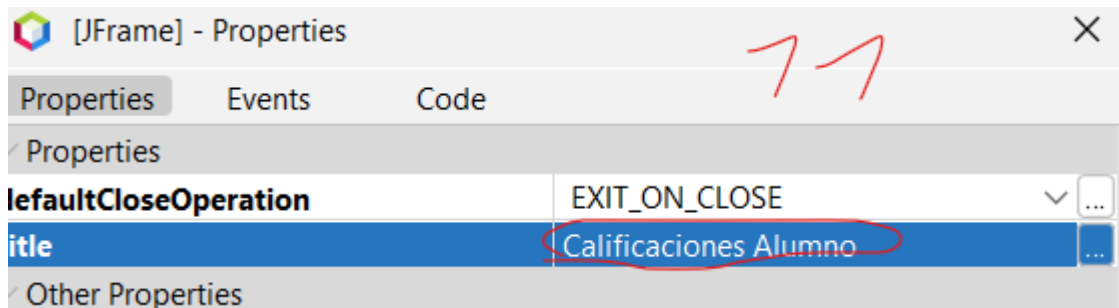
```



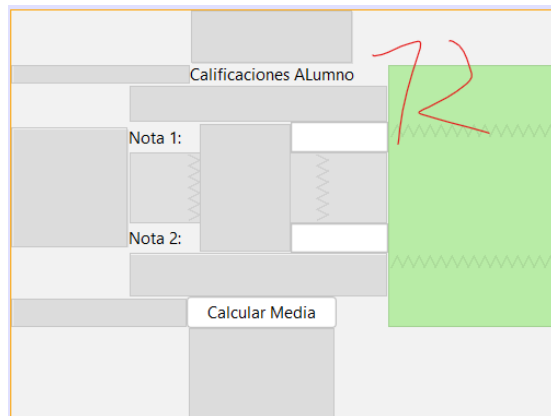
## Calificaciones Alumno

### 2. Práctica calculadora de dos datos

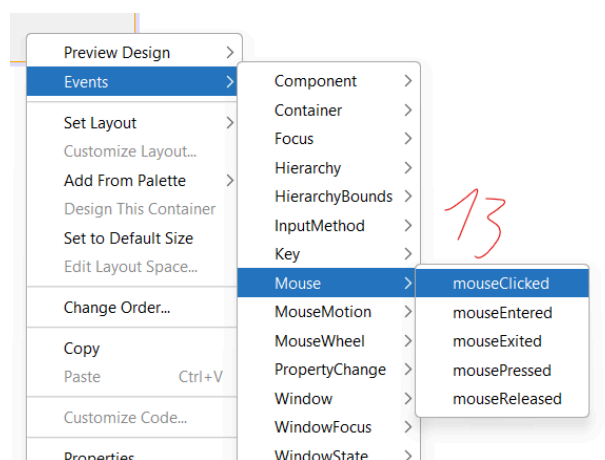
- 11) En **Properties**, cambiamos Title a Calificaciones Alumno



- 12) Añadimos Labels, botones y cuadros de texto para diseñar la calculadora



- 13) En el botón de Calcular Media, click derecho > **Events** > **Mouse** > **mouseClicked**



Para que la media calcule correctamente, el código debería verse así:

```
private void jButton1MouseClicked(java.awt.event.MouseEvent evt) {
    String s1, s2;

    float n1, n2;

    float res;

    s1 = jTextField1.getText();
    s2 = jTextField2.getText();

    n1 = Integer.parseInt(s1);
    n2 = Integer.parseInt(s2);

    res = (n1 + n2) / 2;

    System.out.println("La nota media es: " + res);

    jLabel4.setText("La media es: " + res);

}
```

### 3. Práctica calculadora de cinco datos con porcentaje

A continuación, añadimos los cuadros de texto y códigos necesarios para convertirla en una calculadora con cinco notas, cada una de las cuales cuenta por un porcentaje distinto.

Como se puede ver en **14)** hay una condición que si se cumple, aparece el mensaje APTO. De lo contrario, saldrá NO APTO. La calculadora debería verse como en **15)**

```
private void jButton1MouseClicked(java.awt.event.MouseEvent evt) {
    String s1, s2, s3, s4, s5;

    float n1, n2, n3, n4, n5;

    float res;

    s1 = jTextField1.getText();
    s2 = jTextField2.getText();
    s3 = jTextField3.getText();
    s4 = jTextField4.getText();
    s5 = jTextField5.getText();

    n1 = Integer.parseInt(s1) * 1.15f;
    n2 = Integer.parseInt(s2) * 1.23f;
    n3 = Integer.parseInt(s3) * 1.26f;
    n4 = Integer.parseInt(s4) * 1.12f;
    n5 = Integer.parseInt(s5) * 1.15f;

    res = (n1 + n2 + n3 + n4 + n5) / 5;

    System.out.println("La nota media es: " + res);
    System.out.println("APTO");
    System.out.println("NO APTO");

    jLabel4.setText("La media es: " + res);

    if(res <= 5.0f){
        jLabel17.setText("NO APTO");
    }else{
        jLabel11.setText("APTO");
    }

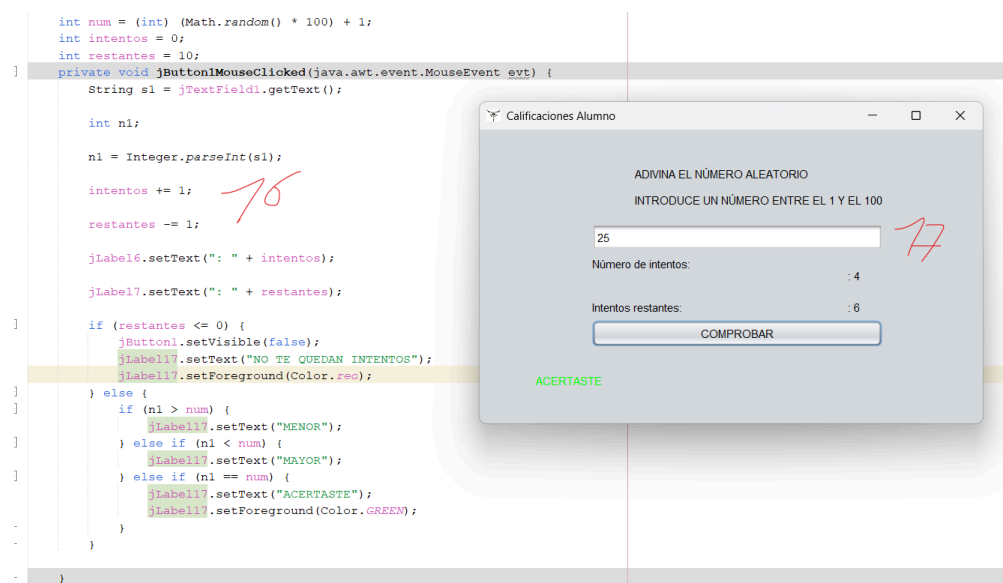
}
```

#### 4. Práctica calcular número aleatorio:

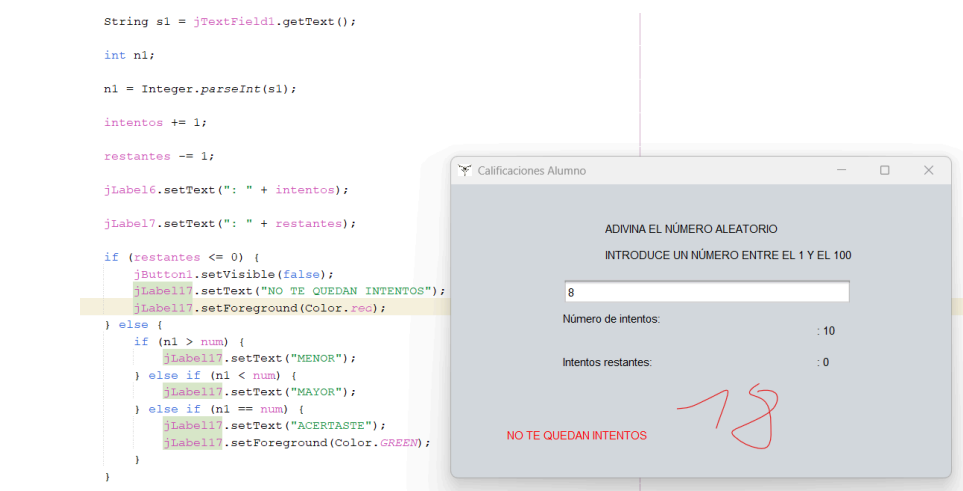
La siguiente práctica consiste en diseñar un juego para adivinar un número entre el 1 y el 100, con un número limitado de 10 intentos. Si el usuario acierta, aparecerá un texto en verde diciendo "ACERTASTE". De lo contrario, aparecerá un texto en rojo indicando que ya no quedan intentos.

#### 16) El código del programa

#### 17) Así es como se ve



#### 18) El programa si fallas todos los intentos



## 5. Ejercicios interfaces

**Cuestión 1: ¿Es posible incluir métodos privados en una interfaz? Razone la respuesta.**

Incluir métodos privados en las interfaces es contrario a la naturaleza de la propia interfaz: es decir, ser una especie de plantilla en blanco que contiene código reutilizable para otras clases. Si los métodos de la interfaz no fuesen públicos, la interfaz no serviría para más que ocupar espacio con código que no se va a utilizar.

Lo mismo ocurre con no hacer abstractos sus métodos, pero ahora responderemos a eso.

**Cuestión 2: ¿Que es una clase abstracta y como se puede usar en las interfaces?**

Una clase abstracta es una cuyos métodos no tienen cuerpo para que estos puedan ser implementados y reutilizados desde otras clases. Si una interfaz se implementa con **implements**, la clase abstracta se implementa con **extends**. Ambas funcionan de forma muy parecida, ya que los métodos de las interfaces son abstractos, y en ambos casos se implementan utilizando la anotación **@Override**.


Sin embargo las clases abstractas cuentan con una limitación que solo hace posible a otras clases implementar una a la vez, problema que las interfaces no tienen: una clase puede implementar tantas interfaces como necesite.

Como comentábamos en la pregunta anterior, los métodos de las interfaces deben ser **PÚBLICOS** y **ABSTRACTOS** por necesidad; de no ser abstractos no podrían sobrecribirse y modificarse de acuerdo con el comportamiento deseado.

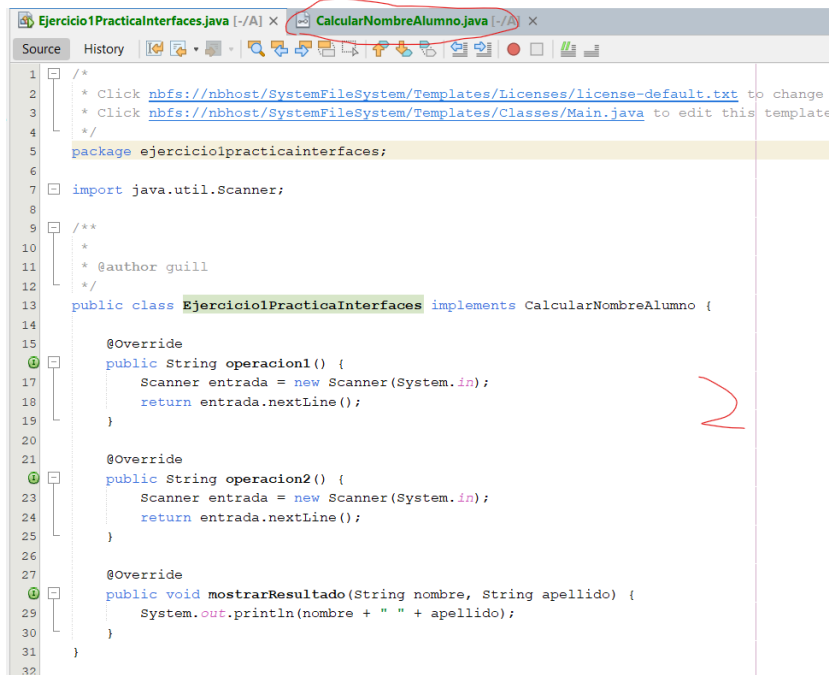
**Ejercicio1: Implemente una interfaz llamada calcular nombrealumno que tenga varios métodos: operacion1, operacion2, mostrar resultado, e implementar una clase de ejemplo que los implemente.**

**1)** La interfaz con los métodos:

```
5 package ejercicio1practicainterfaces;
6
7 /**
8  *
9  * @author guill
10  */
11 public interface CalcularNombreAlumno {
12
13     public String operacion1();
14
15     public String operacion2();
16
17     public void mostrarResultado(String nombre, String apellido);
18 }
19
```



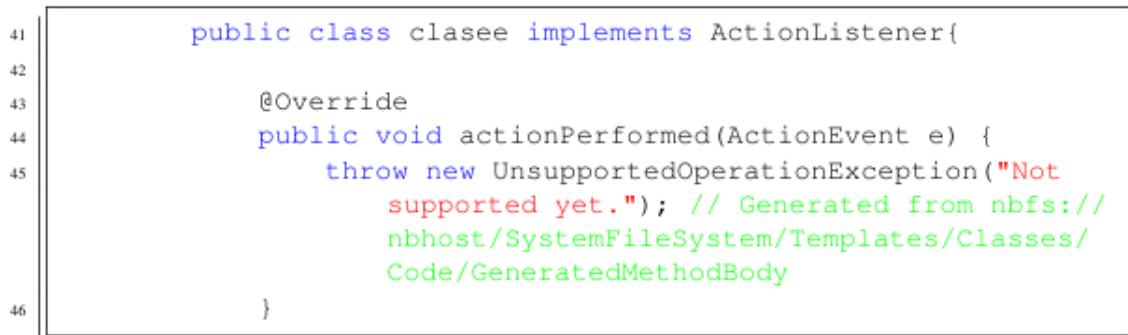
## 2) La clase principal con la interfaz implementada



```

1  /*
2  * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change
3  * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Main.java to edit this template
4  */
5  package ejercicio1practicainterfaces;
6
7  import java.util.Scanner;
8
9  /**
10   *
11   * @author guill
12   */
13  public class Ejercicio1PracticaInterfaces implements CalculaNOMBREAlumno {
14
15      @Override
16      public String operacion1() {
17          Scanner entrada = new Scanner(System.in);
18          return entrada.nextLine();
19      }
20
21      @Override
22      public String operacion2() {
23          Scanner entrada = new Scanner(System.in);
24          return entrada.nextLine();
25      }
26
27      @Override
28      public void mostrarResultado(String nombre, String apellido) {
29          System.out.println(nombre + " " + apellido);
30      }
31  }
32
  
```

**Ejercicio2:** En la siguiente imagen se usa la cláusula `implements` en las clases swing que se vieron anteriormente, ¿qué significan?



```

41      public class clasee implements ActionListener{
42
43          @Override
44          public void actionPerformed(ActionEvent e) {
45              throw new UnsupportedOperationException("Not
46                  supported yet."); // Generated from nbfs://
47                  nbhost/SystemFileSystem/Templates/Classes/
48                  Code/GeneratedMethodBody
49          }
50      }
51
  
```

Está implementando un método llamado **actionPerformed** desde una interfaz llamada **ActionListener**. Asumimos que esto forma parte de un programa más largo de varias clases que requieren usar este método pero con comportamientos definidos para cada clase. Vemos que para sobrescribir el código original (ya que está vacío) utiliza la anotación “**@Override**”.

## 6. Anotaciones y JavaBeans

### Anotaciones en Java

Java posee una característica integrada llamada “annotations” consistente en metadatos que amplían información sobre el código (clases, métodos, variables, etc) sin afectar a la lógica directamente pero que, a diferencia de los comentarios, sí pueden influir en su comportamiento. Las anotaciones pueden ser procesadas en tiempo de compilación o ejecución por una serie de herramientas, frameworks o incluso el propio compilador.

Para definir una anotación se escribe una `@` delante del nombre de la misma (ahora veremos algún ejemplo). Como ya se ha dicho, su propósito es el de configurar comportamientos SIN AFECTAR A LA LÓGICA, también generar código auxiliar como frameworks (es decir, bibliotecas, herramientas o convenciones para facilitar códigos reutilizables) o incluso validar código y generar advertencias.

Los ejemplos más característicos son:

`@Override`: Indica que un método sobrescribe uno de la superclase.

`@Deprecated`: Marca elementos obsoletos.

`@SuppressWarnings`: Suprime advertencias del compilador.

`@FunctionalInterface`: Define interfaces funcionales (Java 8+).

`@SafeVarargs`: Suprime advertencias de varargs en métodos genéricos.

Las anotaciones se pueden utilizar con “retenciones”. Las retenciones forman parte del paquete `java.lang.annotation` y son de tres tipos:

**SOURCE**: Solo en el código fuente (ej: `@Override`).

**CLASS**: En el bytecode, pero no accesible en runtime (por defecto).

**RUNTIME**: Disponible en runtime (ej: `@Test` de JUnit).

Ejemplo de anotaciones:

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
```

```
@Retention(RetentionPolicy.RUNTIME)
public @interface MiAnotacion {
    String valor() default "Predeterminado";
    int numero() default 0;
}
```

```
@MiAnotacion(valor = "Ejemplo", numero = 5)
public class Ejemplo {
    // ...
}
```

**Procesamiento de anotaciones**



Para leer anotaciones en runtime, se usa reflection :

```
public static void main(String[] args) {
    Ejemplo ejemplo = new Ejemplo();
    Class<?> clazz = ejemplo.getClass();

    if (clazz.isAnnotationPresent(MiAnotacion.class)) {
        MiAnotacion anotacion = clazz.getAnnotation(MiAnotacion.class);
        System.out.println("Valor: " + anotacion.valor());
    }
}
```

## JavaBeans

Un JavaBean es una clase Java que sigue convenciones específicas para ser reutilizable y manejable por herramientas, como los frameworks anteriormente mencionados, o los IDE's.. Son la base de muchos patrones en Java, como MVC o componentes en Spring.

Un JavaBeans debe utilizar un constructor público sin parámetros, sus propiedades son privada e implementan la interfaz Serializable.

Cuentan además con los getter y setter para acceder y modificar las propiedades.

A continuación un ejemplo de JavaBeans

```
import java.io.Serializable;

public class Persona implements Serializable {
    private String nombre;
    private int edad;

    // Constructor sin argumentos
    public Persona() {}

    // Getters y setters
    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }

    public int getEdad() { return edad; }
    public void setEdad(int edad) { this.edad = edad; }
}
```

Las anotaciones y JavaBeans suelen combinarse en frameworks modernos. Por ejemplo:

En **Spring**, un JavaBean puede marcarse con `@Component` para ser gestionado por el contenedor. **Hibernate** usa anotaciones como `@Entity` para mapear JavaBeans a tablas de bases de datos. Ambos conceptos son fundamentales para el desarrollo Java orientado a frameworks y buenas prácticas de diseño. Sin embargo, hay una serie de diferencias a tener entre las anotaciones y JavaBeans. Como ya se ha dicho, las anotaciones proporcionan metadatos, mientras que el propósito de JavaBeans es definir los componentes reutilizables. JavaBeans es una clase propiamente dicha con sus getters y setters mientras que las anotaciones, en cuanto a estructura, se parecen más a una interfaz. Las anotaciones se emplean en la compilación mientras que JavaBeans se emplea en tiempo de ejecución.

