



# **DIPLOMADO PROGRAMADOR EN JAVA.**

**Instructor: Giovanni Ariel Tzec Chávez**



# Perfil de ingreso

- ✓ Conocimientos básico de: Internet, HTML y SQL estándar.
- ✓ Conocimiento de la lógica de programación.
- ✓ Conocimiento en lenguaje de programación.
- ✓ Bases de programación orientada a objetos.

# **Diplomado programador JAVA**

Java nivel 2 (Java intermedio)

Java nivel 3 (Java avanzado)

Java nivel 4 (J2EE)

Java nivel 5 (J2EE, hibernate)

Java nivel 6 (JSF)

Java nivel 7 (WebServices)

Java nivel 8 (Tecnologías JSON)

# Java Intermedio(Modulo 2)

- ✓ Estándares de Programación en JAVA
- ✓ Acceso a bases de datos desde Java – JDBC
- ✓ Clases Driver, Connection, Statement, ResultSet
- ✓ Sentencias de SQL en JAVA
- ✓ Se agregará la temática de Java Swing y POO

# Retroalimentando clase final

- ✓ POO
- ✓ Java Swing
- ✓ Listas



# Objetivo de la sesión

Desarrollar aplicaciones swing con POO.

Implementar formularios MDI

ArrayList

# Distribución de tiempo

Módulos	Duración
Java nivel 2	20 horas

# Contenido

- ✓ ArrayList
- ✓ Validaciones de datos
- ✓ Formularios MDI
- ✓ Patrones de diseño
- ✓ Práctica evaluada



# **ArrayList-Estructuras de datos dinámicas.**

## **ArrayList**

La clase ArrayList en Java, es una clase que permite almacenar datos en memoria de forma similar a los Arrays, con la ventaja de que el numero de elementos que almacena, lo hace de forma dinámica, es decir, que no es necesario declarar su tamaño como pasa con los Arrays. Un ArrayList puede contener objetos de tipos distintos.

Cuando se escribe un programa que colecciona datos, no siempre se sabe cuántos valores se tendrá.

En tal caso una lista ofrece dos ventajas significativas:

- ✓ La lista puede crecer o disminuir como sea necesario.
- ✓ La clase `ArrayList` ofrece métodos para las operaciones comunes, tal como insertar o eliminar elementos.

Las listas con una clase genérica (puede contener muchos tipos de objetos) que se encuentra en el paquete `java.util.ArrayList`

# Métodos en ArrayList

MÉTODO	DESCRIPCIÓN
size()	Devuelve el número de elementos (int)
add(X)	Añade el objeto X al final. Devuelve true.
add(posición, X)	Inserta el objeto X en la posición indicada.
get(posicion)	Devuelve el elemento que está en la posición indicada.
remove(posicion)	Elimina el elemento que se encuentra en la posición indicada. Devuelve el elemento eliminado.
remove(X)	Elimina la primera ocurrencia del objeto X. Devuelve true si el elemento está en la lista.
clear()	Elimina todos los elementos.
set(posición, X)	Sustituye el elemento que se encuentra en la posición indicada por el objeto X. Devuelve el elemento sustituido.
contains(X)	Comprueba si la colección contiene al objeto X. Devuelve true o false.
indexOf(X)	Devuelve la posición del objeto X. Si no existe devuelve -1

# Uso de listas

Durante la declaración se indica el tipo de los elementos.

- ✓ Dentro de < > como el tipo de “parámetro”
- ✓ El tipo debe ser una clase
- ✓ No se puede usar tipos de datos primitivos (int, double...)

Métodos útiles de ArrayList

- add: añade un elemento
- get: retorna un elemento
- remove: elimina un elemento
- set: cambia un elemento
- size: longitud del array

# Declaración de ArrayList

```
ArrayList ejemplo= new ArrayList();  
ejemplo.add("Giovanni Tzec");  
ejemplo.add("Docente");  
ejemplo.add(25);  
ejemplo.add(7.85);  
ejemplo.add('t');
```

Nombre del ArrayList **ejemplo**.

Elementos del ArrayList: **“Giovanni Tzec”, “Docente”, 25, 7.85, ‘t’**

**Hacer ejemplo.**



# Iterando colecciones

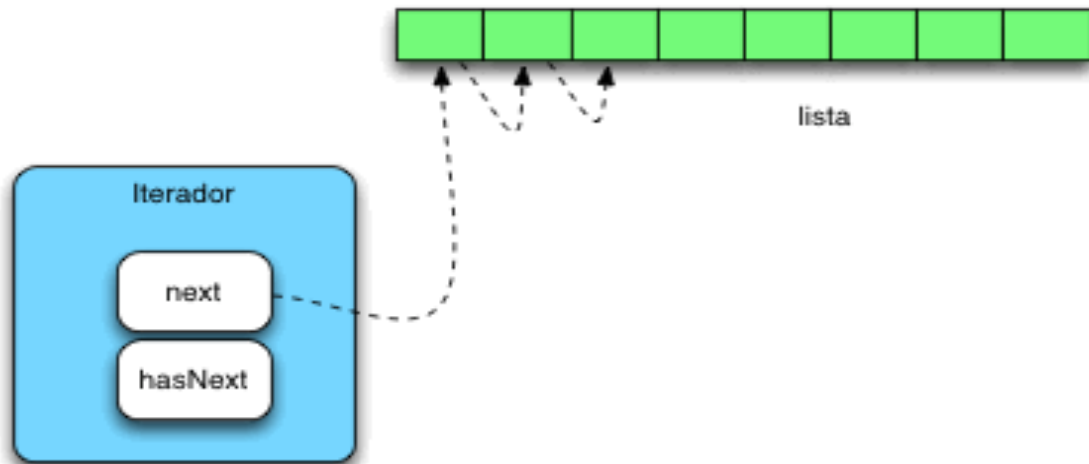
Existen dos formas de recorrer una colección.

- Mediante for-each
- Usando iteradores

# Iterador

Un iterador es un objeto que nos permite recorrer una lista y presentar por pantalla todos sus elementos .

Dispone de dos métodos clave para realizar esta operación hasNext() y next().



```
Iterator it=ejemplo.iterator();  
while(it.hasNext())  
{  
    System.out.println(it.next());  
}
```

Los métodos **hasNext** retorna true si la iteración contiene más elementos, y el método **next** retorna el siguiente elemento en la iteración. El método **remove**, remueve el último elemento que fue retornado por **next** de la colección que está siendo iterada. El método **remove** solo puede ser llamado una vez por llamada de **next** y lanza una excepción si esta regla es violada.

Los métodos **hasNext** retorna true si la iteración contiene más elementos, y el método **next** retorna el siguiente elemento en la iteración. El método **remove**, remueve el último elemento que fue retornado por **next** de la colección que está siendo iterada. El método **remove** solo puede ser llamado una vez por llamada de **next** y lanza una excepción si esta regla es violada.

El siguiente método realiza una eliminación de elementos de la colección

```
static void filter(Collection<String> c) {  
    for (Iterator<String> it = c.iterator(); it.hasNext(); )  
        if (!cond(it.next()))  
            it.remove();  
}
```

# Bucle foreach

```
List<Integer> arreglo= new ArrayList<Integer>();
```

```
.....
```

```
for(int elemento:arreglo){  
    suma += elemento;  
}
```

Si el array contiene objetos de tipos distintos o desconocemos el tipo:

```
for(Object lista:ejemplo)  
{  
    System.out.println(lista);  
}
```



# MDI

Las ***Interfaces de Múltiples Documentos (MDI)*** son aquellas que permiten mostrar JFrames dentro de un solo JFrame principal, de modo que al minimizar o cerrar el JFrame principal todos se cierran. Es una forma de desarrollar aplicaciones que contengan varios formulario (JFrames) ordenados dentro de un mismo espacio de trabajo.

Una ventaja de este tipo de interfaces es que cada formulario se puede trabajar de forma independiente en JPanel, y en el MDI solo se llaman los formularios y todo el funcionamiento del JPanel sera el mismo.

# Procedimiento para crear MDI

```
private void Formulari1ActionPerformed(java.awt.event.ActionEvent evt) {  
    Formulari1 frm= new Formulari1();  
    this.jDesktopPanel1.add(frm);  
    frm.setVisible(true);  
}
```

# Implementación de clases

Persona
dui : Integer nombre : String edad : Integer
Persona() Persona(dui : Integer,nombre : String,edad : Integer) getDui() : Integer getNombre() : String getEdad() : Integer setNombre(nombre : String) setDui(dui : Integer) setEdad(edad : Integer) imprimir() imprimir(dui : Integer,nombre : String,edad : Integer) imprimir(edad : Integer)

# Abstracción

En el análisis de persona, sólo interesa conocer **qué** servicios presta (operaciones), no **cómo** hace para ejecutarlos. Podemos representarlo haciendo métodos específicos para ello

Persona
dui : Integer nombre : String edad : Integer
Persona() Persona(dui : Integer,nombre : String,edad : Integer) getDui() : Integer getNombre() : String getEdad() : Integer setNombre(nombre : String) setDui(dui : Integer) setEdad(edad : Integer) imprimir() imprimir(dui : Integer,nombre : String,edad : Integer) imprimir(edad : Integer)

# Encapsulamiento

La encapsulación es la **reunión** en una estructura, de todos los elementos que en un nivel de abstracción se consideran parte de una misma entidad (categoría o clase).







También es agrupar los datos y operaciones relacionados bajo una misma unidad de programación (**cohesión alta**, o bien, las características están fuertemente relacionadas).

La encapsulación **oculta lo que hace un objeto** de lo que hacen otros objetos; por eso se le llama también ocultación de datos.

**Declarar los atributos privados para establecer encapsulamiento.**



```
private int dui;  
private String nombre;  
private int edad;
```

```
22  
23  public int getDui() {  
24     return dui;  
25 }  
26  
27  public void setDui(int dui) {  
28     this.dui = dui;  
29 }  
30  
31  public String getNombre() {  
32     return nombre;  
33 }  
34  
35  public void setNombre(String nombre) {  
36     this.nombre = nombre;  
37 }  
38  
39  public int getEdad() {  
40     return edad;  
41 }  
42  
43  public void setEdad(int edad) {  
44     this.edad = edad;  
45 }
```

## Encapsulamiento y Ocultación de Datos

- De este modo los clientes de un componente (clase) solo necesitan conocer cuales son los servicios de su interfaz (métodos) y como utilizarlos(necesita parámetros o no). No necesitan conocer como se implementan.
- Así, se puede modificar la implementación de la interfaz en una clase sin afectar a las restantes clases relacionadas con ella, solo es necesario mantener o conservar la interfaz.
- Por lo que, la interfaz indica que se puede hacer con el objeto (caja negra).
- La interfaz pública es estable, pero la implementación se puede modificar.

# Polimorfismo

Es la propiedad que permite a una operación (función) tener el mismo nombre en clases diferentes y actuar de modo distinto en cada una de ellas.

Una misma operación puede realizar diferentes acciones dependiendo del objeto sobre el que se aplique.

En cada caso se realiza una operación diferente.

```
46 //Imprimir1
47 public void imprimir()
48 {
49     JOptionPane.showMessageDialog(null, "Técnicas de programación");
50 }
51 //Imprimir2
52 public void imprimir(int dui, String nombre, int edad)
53 {
54     JOptionPane.showMessageDialog(null, "Datos persona \n Nombre:" + nombre+
55     "\n DUI "+ dui+ "\n Edad:" + edad);
56 }
57 //Imprimir3
58 public void imprimir(int edad)
59 {
60     JOptionPane.showMessageDialog(null, "Edad: "+ edad);
61 }
```

# Polimorfismo

Desde un punto de vista práctico de ejecución del programa, el polimorfismo se realiza en tiempo de ejecución, ya que durante la compilación no se conoce qué tipo de objeto y por consiguiente que operación ha sido invocada.

En Java, el polimorfismo permite que un objeto determine en tiempo de ejecución la operación a realizar.

Si se desea imprimir, la clase tiene un método distinto para hacerlo.

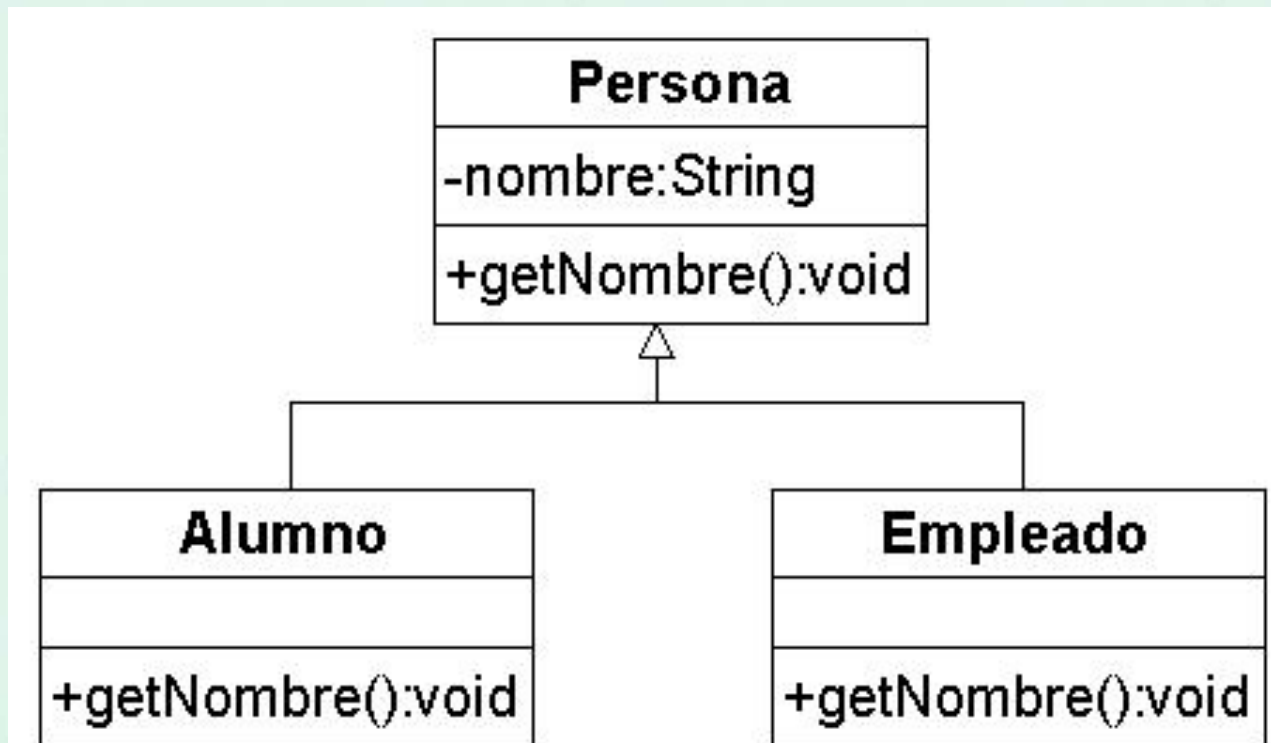
Entonces, el polimorfismo permite definir una única función **imprimir**, cuya implementación es diferente en la clase.



## Polimorfismo.

- Ejemplos.

No importa que tipo de objeto sea, puede ser Persona, Alumno o Empleado, cuando invoque el método **“getNombre()”** llamará al propio de cada subclase, pero el objeto no deja de ser Persona también, y puede estar almacenado en una variable de tipo Persona.



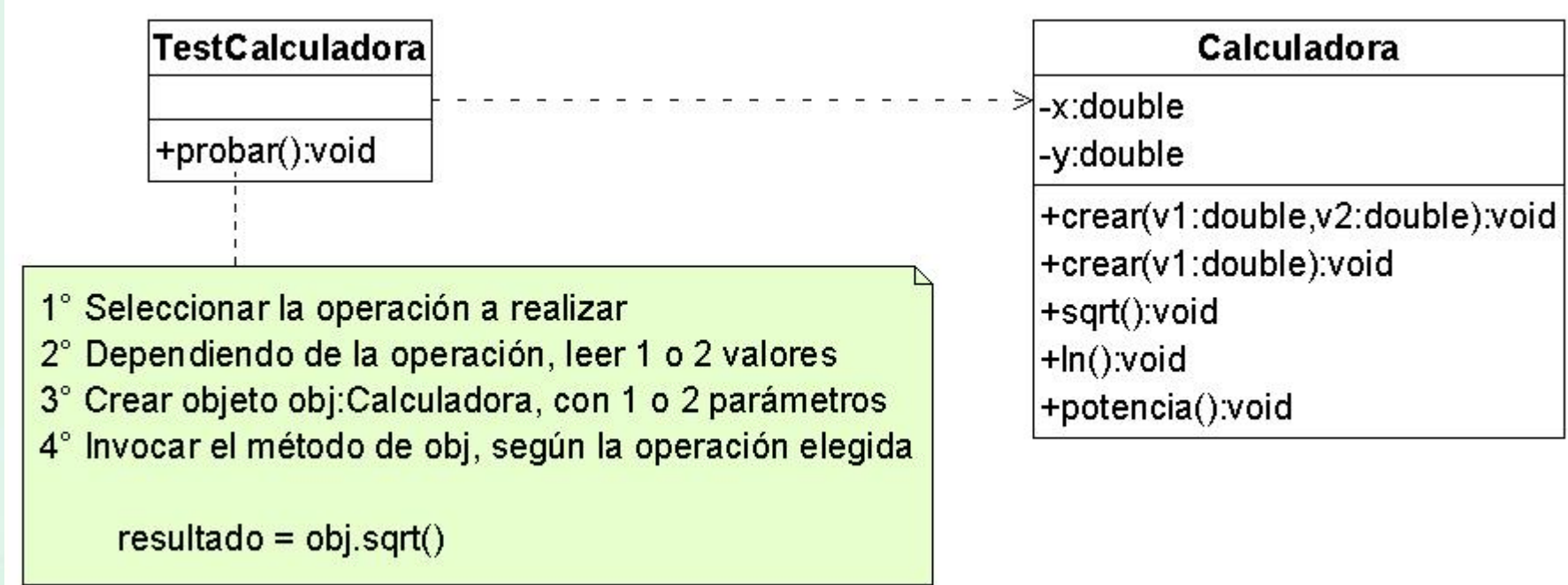
## Sobrecarga de operadores y funciones o métodos.

- Es un tipo especial de polimorfismo.
- La sobrecarga básica de operadores existe siempre.
- El operador + sirve para sumar números enteros o reales, y concatenar cadenas; pero si desea sumar números complejos, hay que sobrecargar el operador + para que permita realizar esta suma.
- Invocar en una clase métodos con el mismo nombre

### Ejemplo:

```
52      public void llenar()  
53      {  
54          obj.setDui(Integer.parseInt(this.jTextField2.getText()));  
55          obj.imprimir();  
56          obj.imprimir(obj.getDui(), obj.getNombre(), obj.getEdad());  
57          obj.imprimir(obj.getDui());  
58      }  
59
```

- Ejemplo de sobrecarga de métodos
- La clase Calculadora realiza las siguientes operaciones:
  - Sqrt: aplicado en el atributo **x**
  - Ln: aplicado en el atributo **x**
  - Potencia: aplicado así, atributo **x** elevado al atributo **y**
- Dependiendo de la operación a realizar se crea un **obj:Calculadora** con 1 o 2 parámetros



# Herencia

Trata de modelar la herencia como en la vida real.

La idea es que las clases hijas (subclases) comparten características con la clase padre (superclase o clase principal).

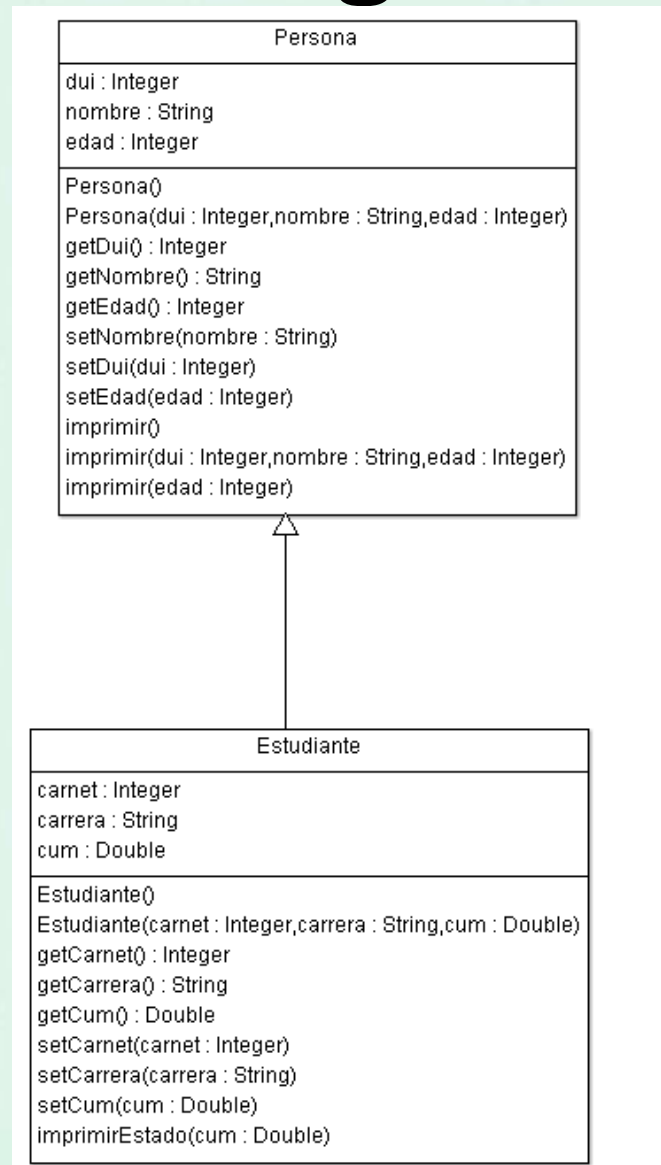
Además de las características compartidas, cada subclase tiene sus propias características.

La clase padre también se le conoce como: principal, superclase o base.

Las clases hijas también se les conoce como: subclase o derivada.



# Codificar la siguiente herencia





# Súper clase y clase derivada

Palabra reservada Extends

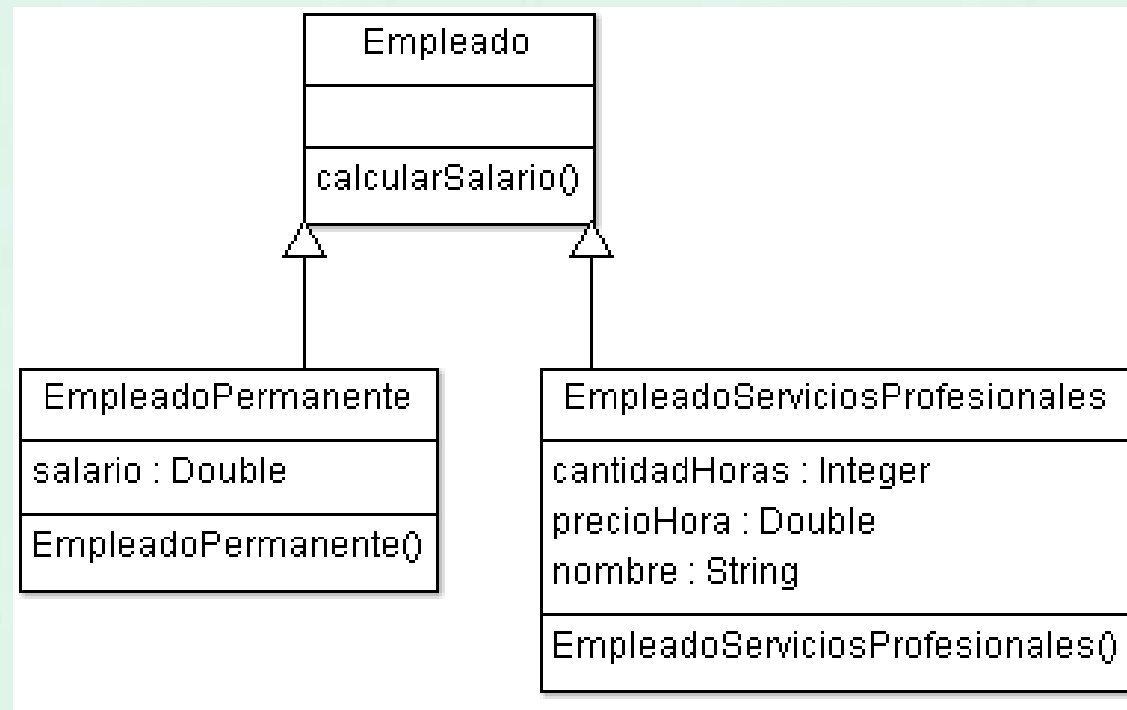
## **Superclases y Subclases**

- ✓ Las clases con propiedades comunes se organizan en superclases.
- ✓ Una superclase es una generalización de las subclases.
- ✓ Una subclase representa una especialización de la superclase.
- ✓ La subclase hereda atributos y comportamientos de la superclase.
- ✓ Subclase: hija, derivada
- ✓ Superclase: padre, base

# Clases abstractas

Son aquellas clases de las cuales no se puede instanciar un objeto, solo implementar, no contiene atributos solo operaciones

Desarrollar un ejemplo.



# JList

- Muestra un conjunto de **ítems** de texto, gráfico o ambos.
- Permite tres tipos de selección:
  - Ítem único
  - Rango Simple
  - Rango Múltiple

## **Ejemplo:**

```
private String [] contenidos = {"uno", "dos", "tres"};  
private JList lista = new JList(contenidos);
```

```
14  L      */
15  [-     public Listas() {
16  |         initComponents();
17  |         this.jList1.setModel(lista());
18  |     }
19  private DefaultListModel lista()
20  [- {
21  |     DefaultListModel<String> modelo= new DefaultListModel<>();
22  |     modelo.addElement("A1");
23  |     modelo.addElement("A1");
24  |     modelo.addElement("A1");
25  |     return modelo;
26  | }
27  [- /**
```

# JPasswordField

Ocultar los caracteres introducidos por el usuario.

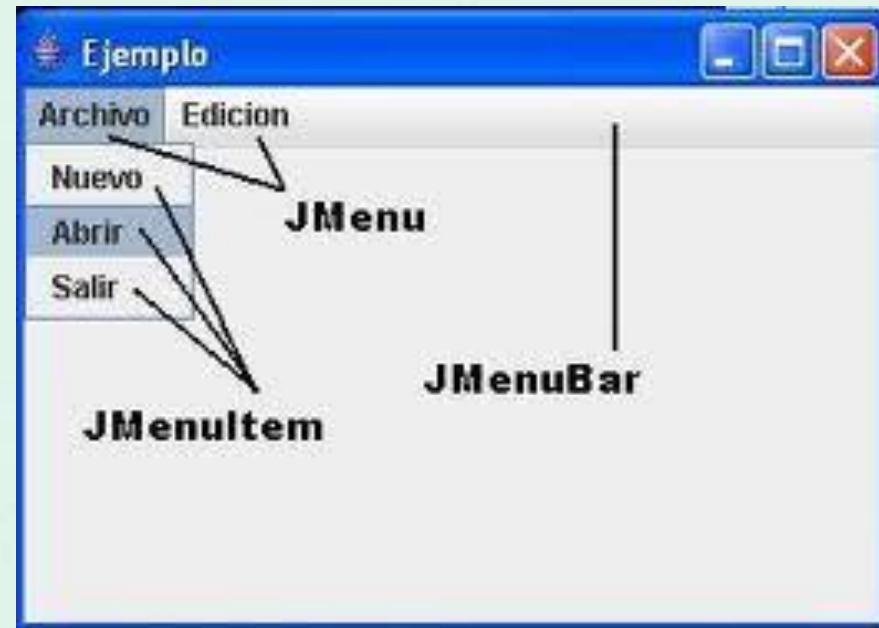
Algunos métodos:

- **setEchoChar('char')** indica el carácter de máscara.
- **getPassword()** recupera el password introducido.



# Menu

- Los menús van dentro de la ventana principal de la aplicación.
- Es posible asignarles un gráfico (ícono).
- Pueden ser de tres tipos:
  - Drop-Down (**JMenuBar**)
  - Submenu (**JMenu**)
  - Contextuales (**JPopupMenu**)



# Menú

- Los menú “Drop-Down” son los que saldrán , por ejemplo, hacer click en Archivo.
- Los submenús son aquellos que salen como un grupo de un elemento de menú.
- Los menús contextuales, (clase **JPopupMenu**) son aplicables a la región en la que está localizado el puntero del ratón.
- Son las clases **JMenuBar**, **JMenu** y **JMenuItem**.

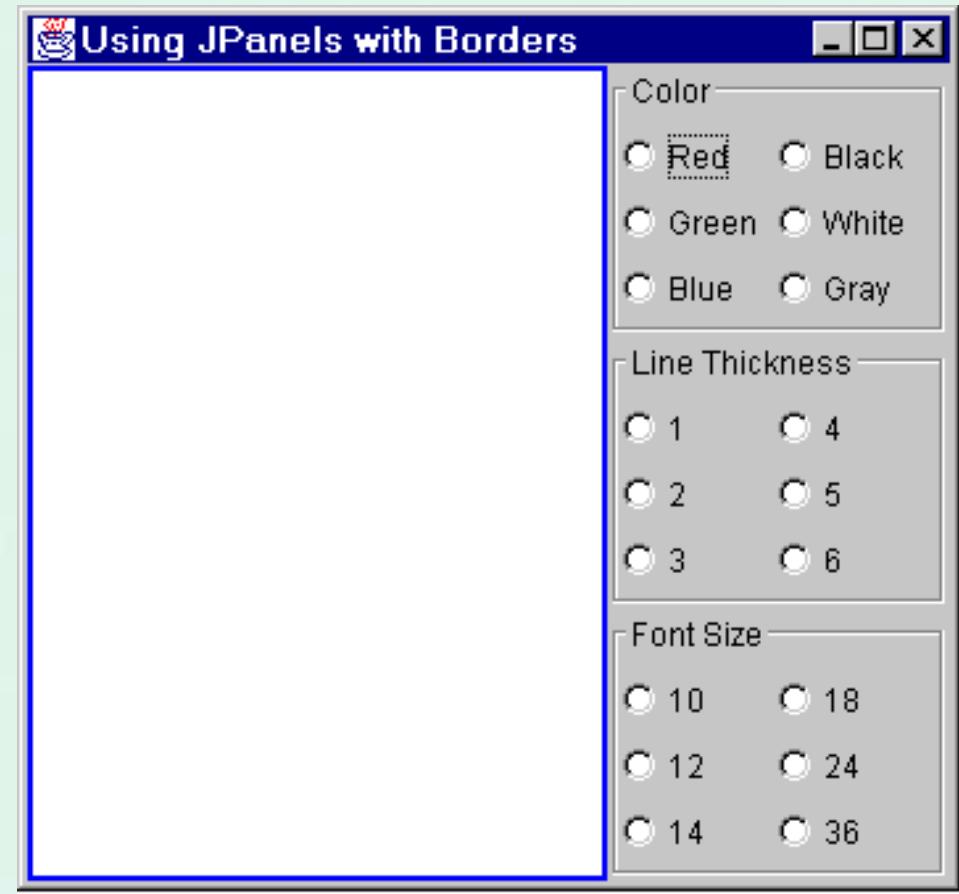
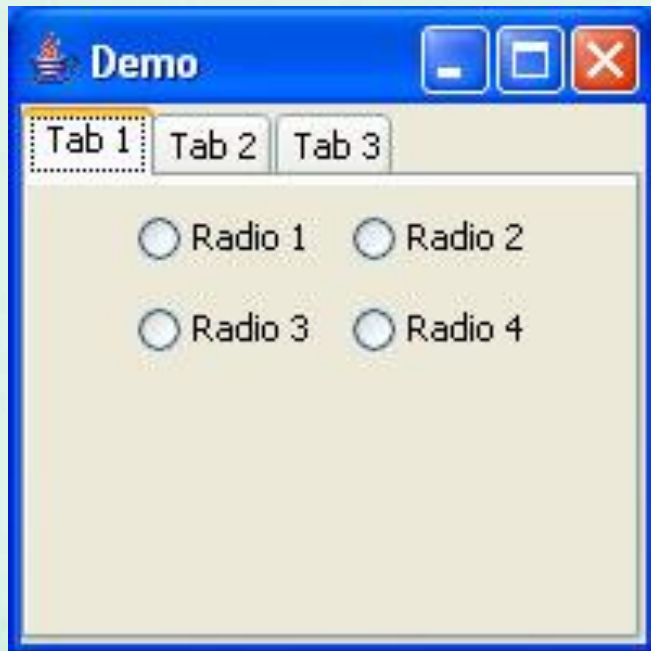
# Componentes Contenedores swing

- Sirven para organizar a los demás componentes y a su vez pueden contener otros objetos del mismo tipo.
  - **JPanel**: simplemente un marco.
  - **JFrame**: una ventana independiente.
  - **JDialog**: una ventana dependiente, no se puede ver en la barra de tareas, ni posee los botones comunes de maximizar o minimizar, como en el JFrame.

# Componentes Contenedores swing

- Clase **JPanel**:

Es un contenedor que agrupa componentes dentro de una ventana.



- Clase **JTabbedPane**:

Es un contenedor que permite tener varios componentes separados por pestañas.



# Ejercicio evaluado 1

Crear un formulario que capture el código empleado (2 letras y 6 dígitos), nombre, edad, salario, genero, intereses, dui, nit, #de afiliado de ISSS, # de teléfono, departamento , cargo, sueldo final.

Al seleccionar el departamento de Finanzas se habilitara (Recursos humanos, compras, cajas, contabilidad)

Desarrollo (Programadores, analistas, DBA)

Redes( soporte técnico, administrador de servidores, administrador de red)

Atención al cliente (Ejecutivos de venta, call center)

Validar los campos necesarios.

Imprimir los datos del empleado.

Tomar en cuenta que si el sueldo del usuario es  $> 0$  y  $< 300$ . Aumento del 10%.

Sueldo  $\geq 300$  y  $< 600$ . Aumento del 12%

Sueldo superior o igual a 600. Aumento del 14%

Crear un formulario principal el cual llame al formulario que capturara los datos del empleado.

Agregar un icono representativo a la opción que llamara el formulario, crear una combinación de tecla para el mismo.

Descontar del salario + aumento (ISSS 6%, AFP 6.25%, ISR 10%, FSV 7%) imprimir todos los datos mas el salario final del empleado, usar formatos numéricos.



# Conexión a datos con MySql y JavaSwing.



# Elementos importantes

Al momento de conectar un gestor de base de datos con MySql con JAVA se deben de considerar los siguientes aspectos:

Descargar MySql para Windows y trabajarlo con la IDE MySQLWorkbench.

Descargara el conector Mysql JDBC Driver el cual se debe de incluir como librería del proyecto.

Si se tiene instalado XAMPP solo se debe de levantar el servicio de MySql e iniciar el IDE MySql Workbench.

# Driver de acceso

Los drivers para poder acceder a cada SGBD no forman parte de la distribución de Java por lo que deberemos obtenerlos por separado. ¿Porqué hacer uso de un driver?. El principal problema que se nos puede plantear es que cada SGBD dispone de su propio API (la mayoría propietario), por lo que un cambio en el SGBD implica una modificación de nuestro código. Si colocamos una capa intermedia, podemos abstraer la conectividad, de tal forma que nosotros utilizamos un objeto para la conexión, y el driver se encarga de traducir la llamada al API. El driver lo suelen distribuir las propias empresas que fabrican el SGBD.

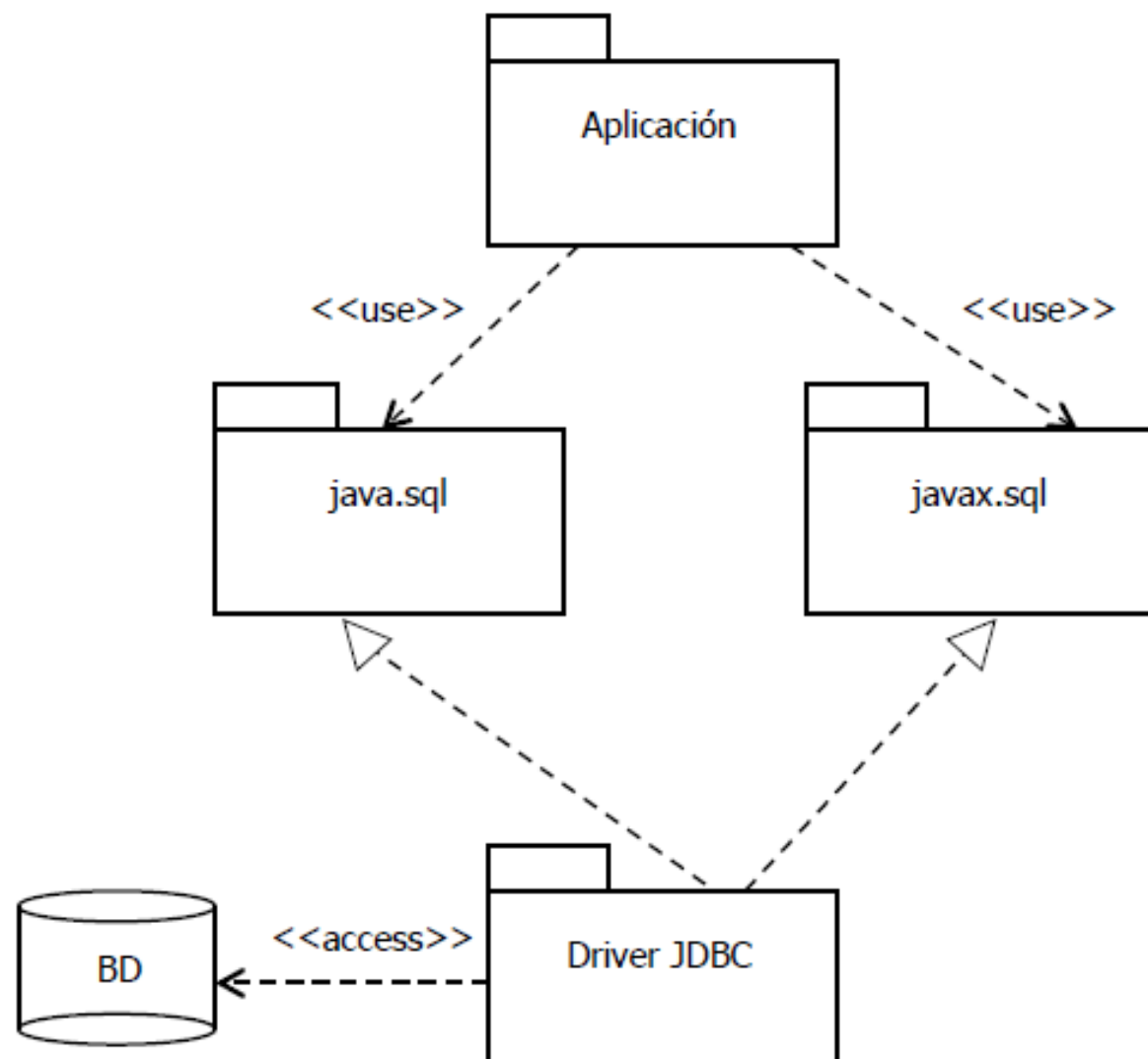
# Drivers

Con el driver instalado, podremos cargarlo desde nuestra aplicación simplemente cargando dinámicamente la clase correspondiente al driver:

- ✓ `Class.forName("com.mysql.jdbc.Driver");`
- ✓ `Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");`
- ✓ `Class.forName("org.postgresql.Driver");`



# Driver JDBC





# Bibliografía.

Piensa en Java 2 Edicion Spanish

Aprenda Java Desde Cero.

# Preguntas y respuestas

