

# Java

## Fecha y hora

Desde Java 8 existe la API `java.time` (también llamada JSR-310). Es inmutable y thread-safe, mucho más clara y correcta que `java.util.Date` / `Calendar`.

Dentro del paquete `Java.Time` van a haber clases que sirven para manipular fechas y horas de forma separada:

- Fecha sin hora: `LocalDate`
- Hora sin fecha: `LocalTime`
- Fecha + hora sin zona: `LocalDateTime`
- Instante en timeline (UTC): `Instant`
- Fecha+hora con zona: `ZonedDateTime`
- Desplazamiento de zona: `OffsetDateTime`, `ZoneOffset`
- Además utilidades: `Period`, `Duration`, `ChronoUnit`,  
`TemporalAdjusters`, `DateTimeFormatter`, enums `DayOfWeek` y `Month`.

## DayOfWeek y Month (Enums útiles)

- `DayOfWeek` tiene constantes `MONDAY`, `TUESDAY`, ... `SUNDAY`.
  - Métodos útiles: `plus(n)`, `minus(n)`,  
`getDisplayName(TextStyle, Locale)` para obtener el nombre localizado.
- `Month` tiene `JANUARY` .. `DECEMBER`.
  - Métodos: `plus(n)`, `minus(n)`, `maxLength()`, `minLength()`,  
`getDisplayName(...)`.
  - Ejemplo: `DayOfWeek lunes = DayOfWeek.MONDAY;`  
`System.out.println(lunes.plus(8));` // devuelve `TUESDAY` (8)

```
días después) Locale arg = new Locale("es", "AR");  
System.out.println(lunes.getDisplayName(TextStyle.FULL,  
arg)); // "lunes"
```

## LocalDate

LocalDate modela una fecha calendario (día/mes/año) sin zona ni tiempo.

### Creación

- `LocalDate.of(int year, int month, int dayOfMonth)`
  - Crea una fecha específica. month puede ser Month o int (1..12).
  - Ej: `LocalDate.of(2020, 2, 29)`
  - Excepción: `DateTimeException` si los valores no forman una fecha válida (ej. mes 13 o 31/02).
- `LocalDate.ofYearDay(int year, int dayOfYear)`
  - Crea la fecha a partir del día del año (1..365 / 366).
  - Ej: `LocalDate.ofYearDay(2021, 60) // 1 de marzo (no bisiesto)`.
- `LocalDate.now() / LocalDate.now(ZoneId zone)`
  - Fecha actual del sistema o de la zona indicada.
  - Ej:  
`LocalDate.now(ZoneId.of("America/Argentina/Buenos_Aires"))`.
- `LocalDate.parse(CharSequence text)` y  
`LocalDate.parse(CharSequence text, DateTimeFormatter f)`

- Parsea un String usando formato ISO (yyyy-MM-dd) o el `DateTimeFormatter` dado.
- Excepción: `DateTimeParseException` si formato inválido.

## Obtener partes de la fecha

- `getYear()`, `getMonthValue()` //1..12 , `getMonth()` //enum Month , `getDayOfMonth()`, `getDayOfYear()`, `getDayOfWeek()` //DayOfWeek
  - Ej: `LocalDate.now().getDayOfWeek()` //devuelve MONDAY, etc.
- `lengthOfMonth()`, `lengthOfYear()`
  - Devuelven el número de días del mes //28-31 o del año //365 / 366
- `isLeapYear()` //esAñoBisiesto()
  - True si el año es bisiesto.

## Modificar/crear nuevas fechas (inmutabilidad)

- `withYear(int)`, `withMonth(int)`, `withDayOfMonth(int)`
  - Devuelven **nueva** fecha con el campo modificado, básicamente actualizan la fecha.
  - Ej: `fecha.withMonth(12)` //Cambia el mes a diciembre
- `plusDays(long)`, `plusWeeks(long)`, `plusMonths(long)`, `plusYears(long)`
  - Suman y devuelven nueva fecha.
  - Cuidado: `plusMonths` puede ajustar al último día del mes si el día original no existe en el mes destino (ej. 31/01 + 1 mes → 28/02 o 29/02).
- `minusDays(long)`, `minusMonths(long)`, `minusYears(long)`
  - Resta un número específico de días, meses o años.

- Ej: `LocalDate hace30Dias = hoy.minusDays(30);`
- `plus(long amountToAdd, TemporalUnit unit) / minus(long, TemporalUnit)`
  - Más flexible; acepta `ChronoUnit.DAYS`, `ChronoUnit.MONTHS`, etc.
- `with(TemporalAdjuster adjuster)`
  - Ajusta según un `TemporalAdjuster` (ver más abajo `TemporalAdjusters`).
  - Ej:
 

```
fecha.with(TemporalAdjusters.firstDayOfNextMonth());
```

## Comparación y orden

- `isBefore(ChronoLocalDate other), isAfter(...), isEqual(...)`
  - Comparan cronológicamente.
- `compareTo(ChronoLocalDate other)`
  - Implementa `Comparable`, devuelve negativo/0/positivo.

## Errores comunes con `LocalDate`

- Confundir zona: `LocalDate.now()` depende del reloj del sistema, no de UTC.
- Asumir que sumar meses conserva el día exacto — puede ajustarse al último día del mes destino.
- Parsear sin `DateTimeFormatter` y recibir `DateTimeParseException` si el formato no es ISO.

## LocalTime (hora sin fecha)

Representa hora del día (hora/minuto/segundo/nano) sin zona.

### Creación

- `LocalTime.of(int hour, int minute)` e overloads con segundos y nanos. Su forma es `HH:mm:ss.SSSSSSSSS` (hora, minuto, segundo y nanosegundo)
- `LocalTime.parse(CharSequence text)` o con `DateTimeFormatter`.
- `LocalTime.now()` / `LocalTime.now(Clock clock)`

### Lectura de componentes

- `getHour()`, `getMinute()`, `getSecond()`, `getNano()`
- `toSecondOfDay()` / `toNanoOfDay()` // útil para comparar o serializar.

### Modificación

- `withHour(int)`, `withMinute(int)`, `withSecond(int)`, `withNano(int)`
- `plusHours(long)`, `plusMinutes(long)`, `plusSeconds(long)`, `plusNanos(long)`
- `minus...` análogo
- `truncatedTo(TemporalUnit unit)` — trunca (ej.: `truncatedTo(ChronoUnit.SECONDS)` quita los nanos).

### Comparación

- `isBefore(LocalTime other)`, `isAfter`, `compareTo`

## Errores y recomendaciones

- `LocalTime` no conoce zona, dos `LocalTime` iguales pueden representar momentos distintos en distintas zonas cuando se combinan con `LocalDate` y `ZoneId`.
- Para medir intervalos entre horas en diferentes días usá `Duration` o calcula con `LocalDateTime`.

## `LocalDateTime` (fecha + hora sin zona)

Combina `LocalDate` y `LocalTime`.

### Creación

- `LocalDateTime.of(year, month, day, hour, minute)` o `LocalDateTime.of(LocalDate, LocalTime)`
- `LocalDateTime.now()` / `LocalDateTime.now(ZoneId)` // este último devuelve la fecha/hora local de la zona, pero sigue sin zona como objeto).
- `LocalDateTime.parse(text, formatter)`

### Conversión y utilidad

- `toLocalDate()` / `toLocalTime()` // Funciona como `toString`
- `atZone(ZoneId zone)` // Convierte a `ZonedDateTime` usando la zona (puede haber ambigüedad en cambios DST).
- `toEpochSecond(ZoneOffset offset)` // Segundos desde epoch usando el offset dado.

### Modificación

- `withYear()`, `withMonth()`, `withDayOfMonth()`, `withHour()`...
- `plusDays`, `plusHours`, `plusMinutes`, `plus(TemporalAmount)` etc.

## Comparación

- `isBefore`, `isAfter`, `isEqual`, `compareTo`

## Errores comunes

- Convertir `LocalDateTime` a `Instant` requiere **un offset o zone**; sin eso no se sabe el instante UTC.
- `LocalDateTime.now(ZoneId)` devuelve la hora "local" de la zona, pero el objeto no almacena la zona.

## DateTimeFormatter (parseo / formateo)

Clase para formatear y parsear.

### Creación

- `DateTimeFormatter.ISO_LOCAL_DATE`, `ISO_LOCAL_DATE_TIME`. // formatos estándar de la pc.
- `DateTimeFormatter.ofPattern(String pattern)` // patrón personalizado: `dd/MM/yyyy`, `yyyy-MM-dd`.
- `DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL/SHORT/MEDIUM/LONG)` // estilo localizado, por ejemplo: 2 de junio del 2005 para `LONG`
- `.withLocale(Locale)` // para nombres de mes/día en idioma deseado.

## Formatear y parsear

- `format(TemporalAccessor temporal) → String`
  - Ej:  
`fecha.format(DateTimeFormatter.ofPattern("dd/MM/yyyy"))`.

## Excepciones

- `DateTimeParseException` al parsear mal.
- `IllegalArgumentException` si el patrón es inválido.

## ChronoUnit

Las constantes de la enumeración `ChronoUnit` cubren un amplio rango de unidades de tiempo, desde las más pequeñas hasta las más grandes:

- Unidades basadas en tiempo: `NANOS`, `MICROS`, `MILLIS`, `SECONDS`, `MINUTES`, `HOURS`, `HALF_DAYS`.
- Unidades basadas en fecha: `DAYS`, `WEEKS`, `MONTHS`, `YEARS`, `DECADES`, `CENTURIES`, `MILLENNIA`, `ERAS`.

## Funciones principales de ChronoUnit

### Manipulación de fechas y horas

Se utiliza junto con métodos como `plus()` y `minus()` de las clases `Temporal` (`LocalDate`, `LocalDateTime`, etc.) para agregar o restar una cantidad específica de una unidad de tiempo. Ejemplo:

```
LocalDate fechaInicial = LocalDate.of(2025, 10, 6);  
LocalDate fechaFutura = fechaInicial.plus(1,  
ChronoUnit.WEEKS); // Añade 1 semana
```



## Cálculo de la diferencia

El método `ChronoUnit.between()` permite calcular la diferencia entre dos puntos temporales en una única unidad de tiempo. El resultado es un valor `long` que representa el número completo de unidades. Ejemplo:

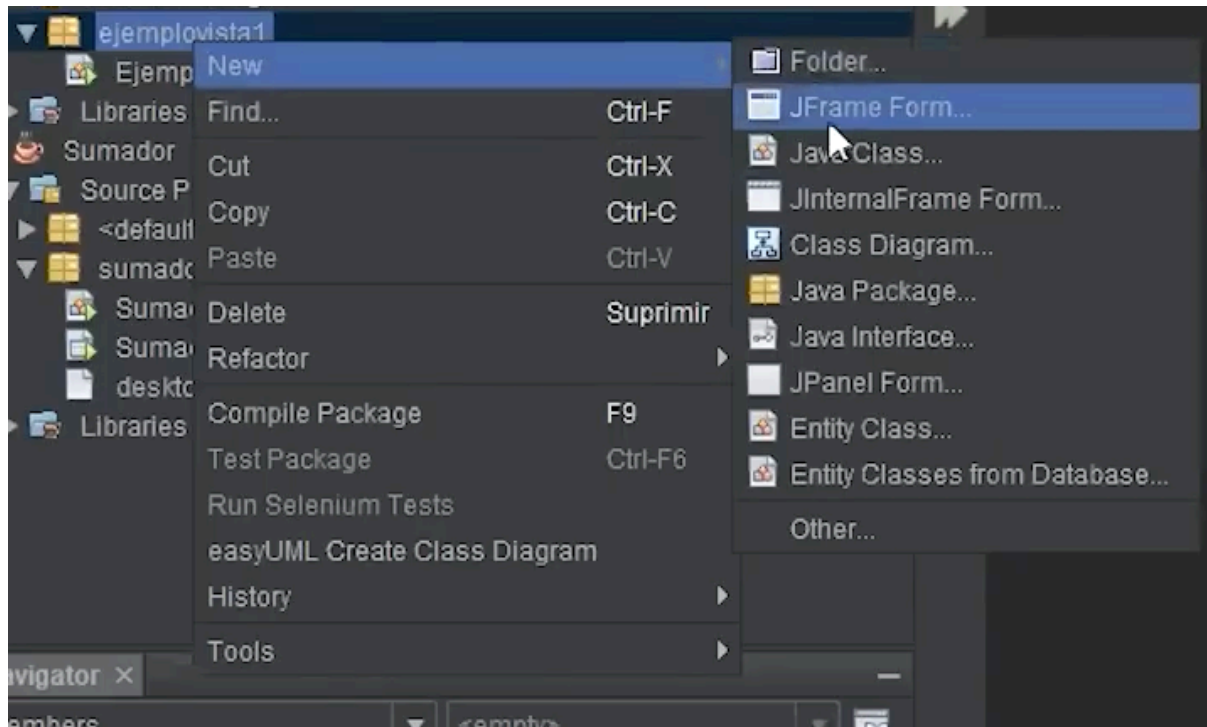
```
LocalDate inicio = LocalDate.of(2025, 1, 1);
LocalDate fin = LocalDate.of(2025, 6, 30);

long mesesEntre = ChronoUnit.MONTHS.between(inicio, fin);
long diasEntre = ChronoUnit.DAYS.between(inicio, fin);

System.out.println("Meses entre: " + mesesEntre); // Imprime:
Meses entre: 5
System.out.println("Días entre: " + díasEntre); // Imprime:
Días entre: 180
```

# Interfaz gráfica

## Creación de JFrame



## Componentes Swing

### Contenedores principales

- **JFrame** → Ventana principal.
- **JDialog** → Ventana de diálogo (emergente).
- **JApplet** → Applet (ya en desuso).
- **JInternalFrame** → Ventanas internas dentro de un JDesktopPane.
- **JPanel** → Panel contenedor.
- **JScrollPane** → Panel con barras de desplazamiento.
- **JSplitPane** → Divide la ventana en dos paneles ajustables.
- **JTabbedPane** → Pestañas.

- **JToolBar** → Barra de herramientas.

## Componentes de interacción

- **JButton** → Botón.
- **JToggleButton** → Botón de encendido/apagado.
- **JCheckBox** → Casilla de verificación.
- **JRadioButton** → Botón de opción.
- **JComboBox** → Lista desplegable.
- **JList** → Lista de elementos.
- **JSpinner** → Selector de valores numéricos.
- **JSlider** → Barra deslizante.

## Campos de texto

- **JLabel** → Etiqueta de texto.
- **TextField** → Campo de texto de una línea.
- **PasswordField** → Campo de contraseña. Para tomar la contraseña se usa `.getPassword()`.
  - `jPasswordField1.setEchoChar('*')`: Enmascara la contraseña.
  - `jPasswordField1.setEchoChar((char)0)`: Desenmascara la contraseña.
- **TextArea** → Área de texto multilínea.
- **FormattedTextField** → Campo con formato (ej. fechas, números).
- **EditorPane** → Editor de texto con formato.
- **TextPane** → Similar al anterior, pero más completo.

## Tablas y listas avanzadas

- **JTable** → Tabla.
- **JTree** → Árbol de datos.

## Elementos de menú

- **JMenuBar** → Barra de menú.
- **JMenu** → Menú.
- **JMenuItem** → Elemento del menú.
- **JCheckBoxMenuItem** → Opción con casilla.
- **JRadioButtonMenuItem** → Opción con radio button.
- **JPopupMenu** → Menú emergente (clic derecho).

## Extras / Utilitarios

- **JProgressBar** → Barra de progreso.
- **JSeparator** → Separador de menús o elementos.
- **JToolTip** → Texto emergente de ayuda.
- **JFileChooser** → Selector de archivos.
- **JColorChooser** → Selector de colores.
- **JOptionPane** → Cuadros de diálogo rápidos (alertas, confirmaciones, input).

## Tomar datos de los componentes

Para tomar los datos de un `TextField` por ejemplo, es necesario usar los métodos de acceso `setText` y `getText`.

- `setText(String texto)` → Cambia el texto que se muestra en el componente.
  - `getText()` → Devuelve el texto que actualmente tiene el componente (como `String`).
- En caso de querer tomar un `INT` por ejemplo es necesario parsear el string (`Integer.parseInt(...)`), también cabe aclarar que para tomar contraseñas de un `JPasswordField` es recomendable usar `getPassword()`.

## Properties

Son los atributos configurables de cada componente gráfico (Swing), muchas son comunes a todos los componentes (background, font, enabled...), y otras son específicas (ej: model en JComboBox). Cuando las modificas, NetBeans genera código automáticamente en el método  `initComponents()`.

### Propiedades generales (compartidas por muchos componentes Swing)

Estas se heredan de las clases base (JComponent, Component) y casi siempre están disponibles:

- **background** → Color de fondo del componente.
- **foreground** → Color del texto o contenido dibujado.
- **font** → Tipo, tamaño y estilo de la letra.
- **enabled** → Si el componente está activo o deshabilitado (grisado).
- **visible** → Si se muestra o no en la interfaz.
- **opaque** → Define si el componente pinta su fondo o deja ver lo que hay detrás.
- **cursor** → El puntero del mouse que aparece al pasar sobre el componente.
- **border** → Borde que rodea al componente (línea, biselado, vacío, etc.).
- **toolTipText** → Texto de ayuda que aparece al pasar el mouse encima.
- **doubleBuffered** → Mejora el renderizado gráfico para evitar parpadeos.
- **alignmentX / alignmentY** → Posición relativa dentro de un contenedor cuando se usan algunos *layout managers*.

### Propiedades de interacción

Tienen que ver con cómo el usuario interactúa con el componente:

- **editable** → Si el usuario puede o no escribir/modificar el contenido (ejemplo: JTextField).
- **focusable** → Si puede recibir foco al tabular o hacer clic.

- **actionCommand** → Texto asociado al evento de acción (usado con ActionListener).
- **selected / selectedIndex / selectedItem** → Estado de selección en botones de opción, listas, combos, etc.
- **model** → Fuente de datos del componente (listas, tablas, botones de grupo, etc.).
- **autoscrolls** → Indica si responde a gestos de arrastre y scroll.

## JOptionPane

El JOptionPane es una clase de Java dentro de la biblioteca Swing que permite crear cuadros de diálogo estándar para interactuar con el usuario en aplicaciones gráficas.

Para usarlo hay que importar la siguiente línea:





```
import javax.swing.JOptionPane;
```

Sintaxis:

```
JOptionPane.showMessageDialog(this, "Texto del cuadro",  
"Titulo", JOptionPane.INFORMATION_MESSAGE); //this hace referencia a  
que se abre en "esta" ventana. Se pone null si no se va a abrir sobre una ventana
```

### Tipos de mensajes (Message Types)

Estos determinan el **icono** y el estilo del mensaje:

- `JOptionPane.ERROR_MESSAGE` →  Mensaje de error.
- `JOptionPane.INFORMATION_MESSAGE` →  Mensaje informativo.
- `JOptionPane.WARNING_MESSAGE` →  Mensaje de advertencia.
- `JOptionPane.QUESTION_MESSAGE` →  Pregunta (ej. confirmar algo).
- `JOptionPane.PLAIN_MESSAGE` → Mensaje simple (sin icono).

## Tipos de opciones (Option Types)

Estos determinan los **botones** que aparecen:

- `JOptionPane.DEFAULT_OPTION` → Botón por defecto (ej. "Aceptar").
- `JOptionPane.YES_NO_OPTION` → Botones **Sí / No**.
- `JOptionPane.YES_NO_CANCEL_OPTION` → Botones **Sí / No / Cancelar**.
- `JOptionPane.OK_CANCEL_OPTION` → Botones **Aceptar / Cancelar**.

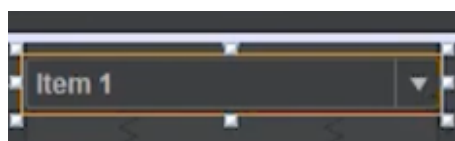
## Tipos de diálogos

Son los distintos métodos estáticos que se usan:

- `showMessageDialog(...)` → Muestra un mensaje (info, error, warning, etc.).
- `showConfirmDialog(...)` → Pide confirmación (Sí/No, etc.).
- `showInputDialog(...)` → Pide un dato al usuario (campo de texto o lista).
- `showOptionDialog(...)` → El más flexible, permite personalizar botones, iconos y texto.

## JComboBox

Un JComboBox es un componente de interfaz de usuario en Java Swing que combina un cuadro de texto o botón con una lista desplegable oculta. Permite a los usuarios seleccionar un valor de una lista predefinida o, si es editable, introducir uno nuevo. Este componente es muy útil para presentar opciones a los usuarios de manera compacta y organizada.



En los properties del JComboBox, en la parte de “model” se tiene que eliminar los ítems por defecto que trae el componente, para luego en la sección de “code” hay que ajustar en “Type Parameters” qué ítems va a mostrar, ya sea String o algún objeto.

### **Agregar items al Combo Box**

Para esto es necesario poner el nombre del JComboBox seguido del método `.addItem(. . .)`, por ejemplo, en caso de querer agregar un alumno al combo debería escribir: `jcbAlumnos.addItem(alumno)`.

Es recomendable crear una función para llenar el combobox y que este sea llamada posterior al `initcomponent`.

Para recuperar los datos del ítem seleccionado hay que irnos al `ActionPerformed` del combobox, seguido a esto usamos el método `getSelectedItem()`. En caso de que sea un objeto tipo `Alumno` por ejemplo, hay que castear el dato de forma explícita.

```
Alumno alumnoSelect = (Alumno) jcbAlumno.getSelectedItem();
```

### **JTable**

Un `JTable` en Java es un componente de la biblioteca Swing que sirve para mostrar y editar datos bidimensionales en formato de tabla, similar a una hoja de cálculo, organizando la información en filas y columnas. Es una herramienta flexible para presentar datos de manera clara y permite personalizar la apariencia y el comportamiento de la tabla para adaptarse a las necesidades de una aplicación gráfica.

Matrícula	Apellido	Nombre
123	Lopez	Maria



JTable se compone de varias clases, como DefaultTableModel, TableColumnModel y TableCellRenderer, que trabajan juntas para mostrar y manipular los datos de la tabla.

Para poder modelar la tabla hay que crear un objeto de tipo DefaultTableModel que representa el modelo de la tabla, y luego importar.

```
private DefaultTableModel modelo = new DefaultTableModel();
```

## Crear columnas

Se debe crear un método que sirva para rellenar la cabecera de la tabla (ej. `armarCabecera()`), y de ahí llamamos al método `addColumn(" . . . ")` para ir creando las columnas. Para terminar la función de llama a la tabla y se usa el método `.setModel(modelo)`.

## Crear filas

Para crear filas se usa el método `.addRow()` de esta forma:

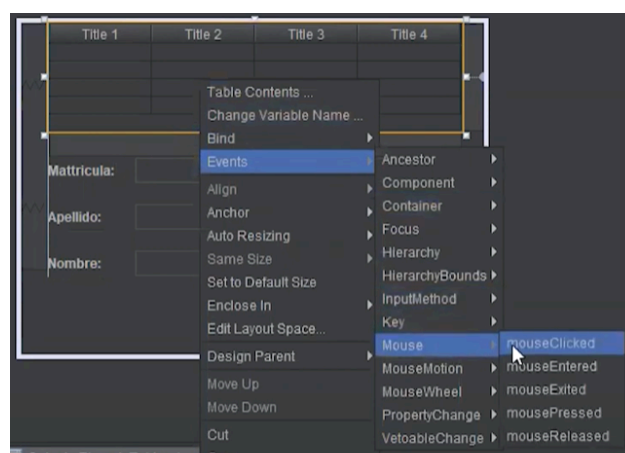
```
modelo.addRow(new Object [] {item1, item2, . . . });
```

## Eliminar filas

Para eliminar una fila se llama primero al método `getSelectedRow()` que es utilizado en Java para obtener el índice de la fila seleccionada (en caso de no detectar una fila seleccionada devuelve -1.), luego se llama al otro metodo `removeRow()` para eliminar la fila según el índice.

## Capturar datos

Para capturar los datos de una fila primero hay que crear el evento de "mouseClicked" desde la JTable:



Esto se hace para que el método se ejecute cuando el usuario haga click en la fila. Una vez hecho el evento hay que guardar la fila seleccionada con `getSelectedRow()` en alguna variable (devuelve -1 si no hay fila seleccionada). Luego para traer finalmente los datos se usa `getValueAt(fila, columna)`, este método devuelve un objeto por lo que hay que castearlo al dato que queramos capturar.

### **Editar celdas**

Por default se puede editar cada celda del `JTable`, pero en el caso de que no queramos que esto suceda es necesario que lo definamos en la creación del objeto `DefaultTableModel`.

```
private DefaultTableModel modelo = new DefaultTableModel();
```

Posterior a los paréntesis se tienen que abrir unas llaves (`{}`) para luego crear el la clase anónima `public boolean isCellEditable(int fila, int columna)`, de la siguiente forma:

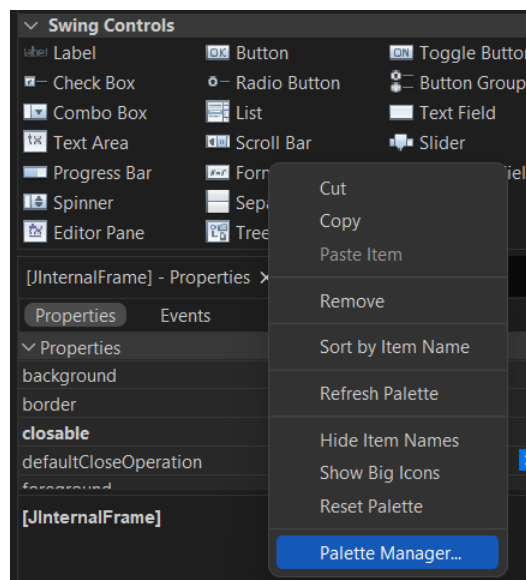
```
private DefaultTableModel modelo = new DefaultTableModel(){
    public boolean isCellEditable(int fila, int columna){
        return false;
        if (columna==2){
            return true;
        }
    }
};
```

Si a este método le retornamos false ninguna celda va a ser editable, en caso contrario con true todas lo van a ser. También como se ve en el ejemplo, es posible que solo una celda pueda ser editable y las otras no.

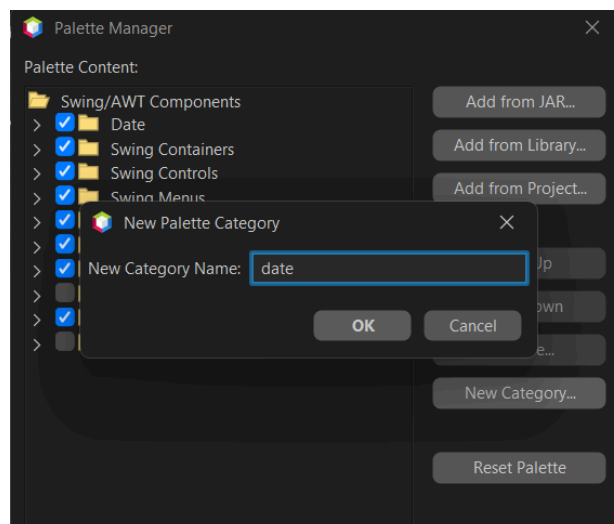
## JCalendar

### Instalación

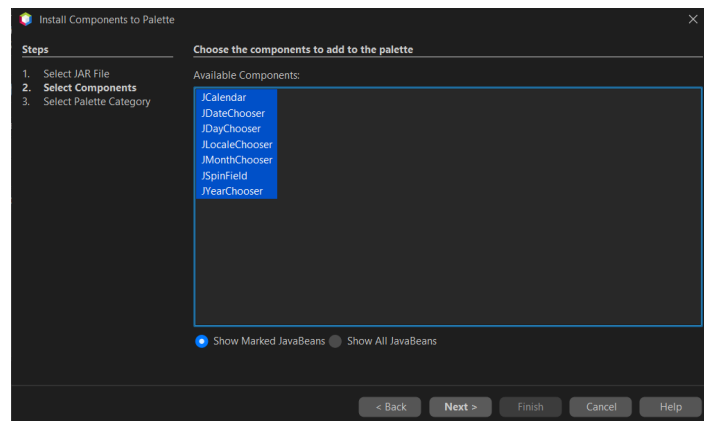
Para poder agregar el .jar del JCalendar hay que hacer click derecho en la Palette de la derecha, y después en “Palette Manager”.



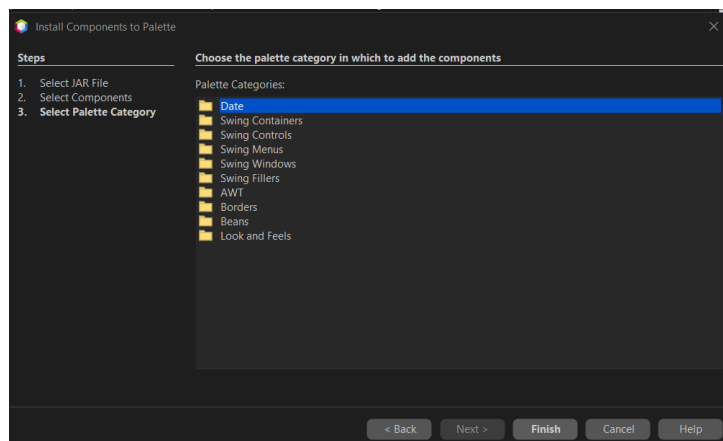
Dentro del Pallete Manager hay que ir a “New Category...”, y ponerle nombre al nuevo desplegable.



Una vez creado hay que ir a “Add from JAR...” y buscar el Calendar.jar, seleccionamos todos y le damos a next



Por último seleccionamos la carpeta que creamos antes y le damos a finish



## JDBC - Java Database Connectivity

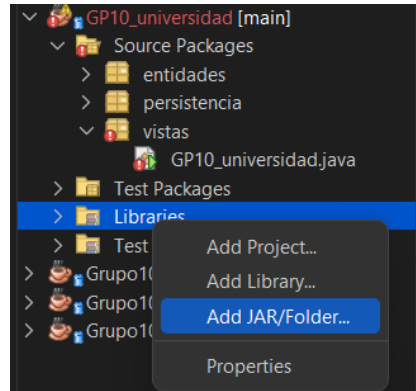
Es una interfaz de programación de aplicaciones que define como una aplicación java puede acceder a una base de datos. Proporciona métodos y clases que permiten establecer conexiones con la base de datos, y enviar consultas para obtener resultados.

Los componentes del JDBC son:

- El gestor de drivers (java.sql.DriverManager) //Administra los controladores de la base de datos
- La conexión con la base de datos (java.sql.Connection) //Envía y recibe resultados de la base de datos a través de las peticiones
- La sentencia a ejecutar (java.sql.Statement) //Es la interfaz SQL que nos va a permitir enviar consultas a la base de datos (INSERT, DELETE, etc). Excepto el SELECT
- El resultado (java.sql.ResultSet) //Es el objeto que tiene los resultados de tipo SELECT de nuestra base de datos, y nos permite iterar a través de los registros devueltos por la consulta para acceder a los datos específicos de cada consulta

## Instalación de la conexión

Para instalar el .jar tenemos que darle click derecho a la carpeta de “Libraries”, y después ir a “add JAR/Folder...”. Después buscamos el driver.



## Pasos para utilizar JDBC en aplicaciones Java

### 1. Cargar / Registrar el Driver JDBC

Dentro del main, se hace referencia a esta línea para poder llamar a la clase driver. La siguiente línea corresponde a MariaDB.

```
Class.forName("org.mariadb.jdbc.Driver");
```

Pero también se debe poner en un try catch, ya que es posible que no encuentre el driver correspondiente y debemos capturar la excepción.

## 2. Obtener la conexión

Después de llamar al driver, se tiene que obtener la conexión mediante la url de la base de datos. Para esto es necesario crear un objeto de tipo Connection.

```
Connection conexion =  
DriverManager.getConnection(url,usr,pwd);
```

En la url va a ser donde vaya a estar alojado el servidor, después de colocar "jdbc:mysql://" vamos a colocar el host, en nuestro caso como lo estamos levantando con xampp tenemos que poner nuestra IP o directamente "localhost:puerto" (generalmente el puerto es 3306. si no se pone nada automáticamente se conecta ahí). Después colocamos un / para seguirlo con el nombre de la base de datos. Ejemplo:

```
"jdbc:mysql://localhost:3306/obrador2023"
```

En el segundo parámetro donde dice "usr" hace referencia al usuario de conexión de la base de datos, por defecto este es "root".

El tercer parámetro "pwd" hace referencia a la contraseña de este usuario, como recién estamos viendo la conexión con bases de datos colocamos comillas simples "".

Toda la línea completa debería quedar de esta forma:

```
DriverManager.getConnection("jdbc:mysql://localhost:3306/  
obrador2023","root","");
```

Para terminar hay que asegurarse de manejar de excepción SQL

```
try {
    Class.forName("org.mariadb.jdbc.Driver");
    conexion = DriverManager.getConnection(
} catch (ClassNotFoundException ex) {
    JOptionPane.showMessageDialog(null, "E
    System.out.println("No se conecto a la Bas
} catch (SQLException ex) {
    JOptionPane.showMessageDialog(null, "E
}
```

### 3. Crear el comando SQL

```
conn.createStatement();
conn.prepareStatement(sql);
conn.prepareCall(sql);
```

### 4. Ejecutar el comando SQL

```
stmt.executeQuery();
stmt.executeUpdate();
```

En los casos anteriores se uso executeUpdate()

### 5. Procesar los resultados

```
(create, alter, drop) DDL
(insert, update, delete) DML
(select)
```

### 6. Liberar recursos.

```
stmt.close();
conn.close();
```

## Consultas

### INSERT

Para insertar tuplas se usan exactamente los mismos comandos que en sql, en este caso el INSERT. Esta misma línea de sql se debe guardar en un string con los datos que queramos. Luego, para crear el comando necesitamos crear el objeto de tipo PreparedStatement para usar el método de la conexión "prepareStatement(string)".

```
PreparedStatement ps = conexion.prepareStatement(INSERT...);  
//Dentro iria el comando sql con los datos que queramos agregar a la base de datos
```

Para poder ejecutar la sentencia anterior tenemos que llamar al método ps.executeUpdate(), pero esta devuelve un entero que representa las filas afectadas, por lo que es necesario guardarlo en un int.

```
int registro = ps.executeUpdate(); //Ejecuta
```

### UPDATE

El update funciona igual que el insert, necesitamos guardar en un string la consulta y luego usar el método ps.executeUpdate().

```
PreparedStatement ps = conexion.prepareStatement(UPDATE...);  
//Dentro iria el comando sql con los datos que queramos modificar de la base de datos
```

```
int registro = ps.executeUpdate(); //Ejecuta
```



## DELETE

Para eliminar es exactamente igual que los anteriores.

```
PreparedStatement ps = conexion.prepareStatement(DELETE.);  
//Dentro iria el comando sql con los datos que queramos modificar de la base de  
datos
```

```
int registro = ps.executeUpdate(); //Ejecuta
```

## SELECT

Para poder usar el select igual que en los anteriores creamos un string con la consulta sql, luego creamos el objeto PreparedStatement ps = conexion.prepareStatement(sql). Una vez hecho esto usamos el método ps.executeQuery(), que devuelve el resultado de la consulta como si fuese una matriz de tipo ResultSet.

```
//Obtener todos los empleados.  
String sql="SELECT * FROM empleado";  
PreparedStatement ps=conexion.prepareStatement(sql);  
ResultSet resultado=ps.executeQuery();
```

Ahora para poder acceder a los datos del objeto podemos usar el método resultado.next(), devuelve true en caso de encontrar una fila para recorrer o false cuando ya no hay más. Se usa en un while para poder recorrer el objeto.

```
while(resultado.next()){  
    System.out.println("ID "+resultado.getInt("idEmpleado"))  
}
```

Existe un get para cada tipo de dato, por lo que primero se pone get y en los paréntesis el nombre exacto de la columna de la base de datos.