APUNTES JAVA

José Juan Urrutia Milán

Reseñas

- Curso java desde 0: https://www.youtube.com/watch?v=coK4jM5wvko&list=PLU8oAlHdN5BktAXdEVCLUYzvDyqRQJ2lk&index
- API java: https://docs.oracle.com/javase/7/docs/api/
- Instalar jdk para ejecutar desde CMD: https://www.youtube.com/watch?v=kPWezAZGPks&list=PLIAU2zQF0bqKBvYBD7gQPIqUsxcAtWVdO&index=5
- Descargar eclipse IDE: https://www.youtube.com/watch?v=MYbyzNWnrzU&t

Siglas/Vocabulario

- **API:** Interfaz de Programación para Aplicaciones (Vienen explicadas todas los paquetes, clases, interfaces, métodos y constantes de java).
- **JDK:** Java Development Kit, kit de desarrollo Java (Conjunto de herramientas para ejecutar programas en Java).
- **IDE:** Entorno de Desarrollo Integrado (App para programar).
- **JRE:** Entorno de Ejecución Java (Máquina virtual=Multiplataf).
- **POO:** Programación Orientada a Objetos.
- **OS:** Operation System
- **Layout:** Es la forma en la que se disponen los diferentes elementos en un marco o lámina (no es una sigla).
- **Cte/Ctes:** Abreviatura de Constante/Constantes.
- **Parámetro de tipo:** Tipo de dato en programación genérica (Se especifica dentro de <>).
- **BBDD:** Bases de Datos.
- **JDBC:** Java Data Base Connectivity.
- **JSP:** Java Server Pages (Páginas de Servidor Java).
- **MVC:** Modelo Vista Controlador

Leyenda

Cualquier abreviatura o referencia será subrayada.
Cualquier ejemplo será escrito en **negrita.**
Cualquier palabra de la que se pueda prescindir irá escrita en cursiva.
Cualquier abreviatura viene explicada a continuación:

Abreviaturas/Referencias:

- 123: Hace referencia a cualquier número.
- nombre: Hace referencia a cualquier palabra/cadena de caracteres.
- a: Hace referencia a cualquier caracter.
- cosa: Hace referencia a cualquier número/palabra/cadena.
- Tipo_var: Hace referencia a cualquier tipo de variable primitiva o de tipo String.
- nombre_var: Hace referencia a cualquier nombre que se le puede dar a una variable.
- código: Hace referencia a cualquier instrucción. (Se usará para indicar dónde se podrá inscribir código.)
- variable: Hace referencia a cualquier variable.
- condición: Hace referencia a cualquier condición. Entiéndase por condición, una afirmación que devuelve un true o un false. Ej: (variable == 123). *Una variable del tipo boolean puede ser usada como una condición.

Índice

Capítulo I: Programación básica

Título I: Estructura básica

Programa Java más simple (Hola mundo):

Comentarios en Java

Importar paquetes

Título II: Variables

Tipos primitivos

Operadores

Constantes

Imprimir variables float o doubles truncadas

Refundiciones y castings

Casting de int a String

Título III: Clase Math (java.lang)

Título IV: Clase String (java.lang)

Objetos String (=variable)

Métodos Clase String

Título V: Entrada de datos

Clase Scanner (java.util)

Clase JOptionPane (javax.swing)

Título VI: Condicionales

if, else if y else

switch

Condicional ternario

Título VII: Bucles

while

do while

for
for each
Uso de for each para recorrer matrices
Título VIII: Arrays/Arreglos/Matrices
Unidimensionales
Bidimensionales
Título IX: Palabras reservadas
Relacionadas con variables
Relacionadas con procedimientos
Relacionadas con clases, objetos e interfaces
Relacionadas con paquetes
Modificadores de acceso
Capítulo II: POO
Título I: Clases y Objetos
Definición
Inicializar un objeto perteneciente a una clase
Nomenclatura de variables y métodos pertenecientes a una clase
Variables públicas
Métodos
Título II: Creación de una clase
La clase en sí
Campos de clase
Método Constructor
Sobrecarga de constructores
Métodos
GETTER
SETTER

Título III: Herencia

Polimorfismo

Clases abstractas

Título IV: Interfaces

Capítulo III: Interfaces gráficas básicas (librería javax.swing)

Título I: Creación de frame: clase JFrame

Principales métodos de JFrame

setVisible(boolean x)

setSize(int x, int y)

setDefaultCloseOperation(int x)

setLocation(int x, int y)

setBounds(int x, int y, int ancho, int largo)

setResizable(boolean x)

setExtendedState(int x)

setIconImage(Image x)

dispose()

pack()

Título II: Clase Toolkit

(static) getDefaultToolkit()

beep()

getScreenSize()

getImage(String nombre_archivo)

Título III: Creación de láminas

paintComponent(Graphics g)

Limpiar láminas

Título IV: Dibujar y escribir

Principales métodos de Graphics

drawString(String str, int x, int y)

drawRect(int x, int y, int ancho, int largo)

drawLine(int x, int y, int a, int b)

Título V: Biblioteca Java 2D y clase Graphics

draw(Shape s)

clase Rectangle2D

getCenterX()

getCenterY()

clase Ellipse2D

clase Line2D

Título VI: Colorear, clase Color y objetos java2D

setPaint(Color x)

Obtener colores personalizados (clase Color)

Colorear fondo del frame coloreando la lámina

Determinar color por defecto para cualquier objeto

Título VII: Cambiar fuentes (tipos de letra)

Saber tipos de letras instalados en OS

Crear objeto perteneciente a Font

Aplicar fuente

Copiar láminas a otras láminas

Título VIII: Trabajar con imágenes

Almacenar imágenes en Image a partir de una ruta

drawImage(Image imagen, int x, int y, ImageObserver observer)

Obtener tamaño de la imagen: getWidth() y getHeight()

Capítulo IV: Eventos

Título I: Creación de botones

Título II: Creación de eventos de botón

- Clases internas anónimas

Título III: Creación de eventos de ventana

- Métodos WindowListener
  - windowActivated(WindowEvent e)
  - windowClosed(WindowEvent e)
  - windowClosing(WindowEvent e)
  - windowDeactivated(WindowEvent e)
  - windowDeiconified(WindowEvent e)
  - windowIconified(WindowEvent e)
  - windowOpened(WindowEvent e)
- Detectar cambios de estado

Título IV: Clases adaptadoras

Título V: Creación de eventos de teclado

- Saber cual es la tecla pulsada
- Métodos KeyListener
  - keyPressed(KeyEvent e)
  - keyReleased(KeyEvent e)
  - keyTyped(KeyEvent e)

Título VI: Creación de eventos de ratón

- Algunos métodos de MouseEvent
  - getX()
  - getY()
  - getClickCount()
  - getModifiersEx()

Métodos MouseListener

mouseClicked(MouseEvent e)

mouseEntered(MouseEvent e)

mouseExited(MouseEvent e)

mousePressed(MouseEvent e)

mouseReleased(MouseEvent e)

Detectar si el ratón está arrastrando o sólo moviendo

Métodos MouseMotionListener

mouseDragged(MouseEvent e)

mouseMoved(MouseEvent e)

Título VII: Creación de eventos de foco

Foco sobre componentes

Métodos FocusListener

focusGained(FocusEvent e)

focusLost(FocusEvent e)

Foco sobre Ventanas

Métodos WindowFocusListener

windowGainedFocus(FocusEvent e)

windowLostFocus(FocusEvent e)

Título VIII: Múltiples fuentes

Métodos Action

(Método heredado) actionPerformed(actionEvent e)

putValue(String clave, String valor)

getValue(String clave)

Crear una combinación de teclas como fuente

Título IX: Múltiples oyentes

Capítulo V: Layouts

Título I: FlowLayout

new FlowLayout() (default)

new FlowLayout(int alineacion)

new FlowLayout(int alineacion, int hespacio, int vespacio)

Título II: BorderLayout

new BorderLayout() (default)

new BorderLayout(int hespacio, int vespacio)

Título III: Usar distintas Layouts

Título IV: GridLayout

new GridLayout() (default)

new GridLayout(int filas, int columnas)

new GridLayout(int filas, int colums, int hespacio, int vespacio)

Título V: BoxLayout (Clase Box)

new BoxLayout(Container c, int axis)

(estáticos) createHorizontalStrut(int x), createVerticalStrut(int x)

(estático) createGlue()

Título VI: SpringLayout (en muelle)

(estático) constant(int a, int b, int c)

(estático) constant(int a)

putConstraint(String a, Component b, String c, String d,

Component e)

Título VII: Disposición libre

Crear disposiciones libres propias

layoutContainer(Container padre)

preferredLayoutSize()

preferredLayoutSize(Container padre)

addLayoutComponent(String nombre, Component c)

minimumLayoutSize(Container padre)

removeLayoutComponent(Component c)

getComponentCount()

getComponent(int x)

Capítulo VI: Formularios, componentes Swing

setBorder(Border b)

setText(String s), getText()

setSize(int a, int b), setLocation(int x, int y), setBounds(int x, int y, int a, int b)

setFont(Font f)

getStyle(), getFont()

getPreferredSize()

setEnabled(boolean b)

setMaximumSize(Dimension d), setMinimumSize(Dimension d)

setToolTipText(String x)

Título I: Botones: JButton

setEnabled(boolean b)

Título II: Etiquetas: JLabel

Aplicar tipo de texto y negrita

Título III: Líneas de texto: JTextField

setBackground(Color c)

JPasswordField

Eventos de JTextField

changedUpdate(DocumentEvent e)

insertUpdate(DocumentEvent e)

removeUpdate(DocumentEvent e)

Título IV: Campos de texto: JTextArea

setLineWrap(boolean b)

getLineWrap()

append(String s)

Título V: Checkboxes: JCheckBox

isSelected()

setSelected()

Título VI: Botones de radio: JRadioButton

Título VII: Combo box: JComboBox

setEditable(boolean b)

getSelectedItem()

removeAllItems()

Tïtulo VIII: Seleccionador deslizante: JSlider

setPaintTicks(boolean b)

setPaintLabels(boolean b)

setMajorTickSpacing(int x), setMinorTickSpacing(int x)

setOrientation(int x)

setSnapToTicks(boolean b)

getValue()

Eventos del JSlider

Título IX: Líneas de texto con aumento: JSPinner

setPreferredSize(Dimension d)

getValue(), getNextValue(), getPreviousValue()

Eventos del JSpinner

Invertir JSpinner

Título X: Menús: JMenuBar, JMenu, JMenuItem

addSeparator()

setJMenuBar(JMenuBar x)

Menús a partir de Action

Añadir imágenes a JMenuItem

new JMenuItem(String s, Icon i)

Menús con CheckBox

Clase JCheckBoxMenuItem

isSelected()

Menús con RadioButton

Clase JRadioButtonMenuItem

isSelected()

Atajos de teclado a menús

setAccelerator(KeyStroke k) (JMenuItem)

Título XI: Menús click dcho: JPopupMenu

setComponentPopupMenu(JPopupMenu c)

Título XII: Barras de herramientas: JToolBar

add(Action a)

addSeparator()

setOrientation(int x)

Título XIII: Ventanas emergentes

Cuadros de diálogo: JOptionPane

(static) showMessageDialog(Component padre, Object msg)

(static) showInputDialog(Component padre, Object msg)

(static) showConfirmDialog(Component padre, Object msg, String titulo, int a)

(static) showOptionDialog(Component padre, Object msg, String titulo, int a, int b, ICon icon, Object[] options, Object valor_inicial)

Cuadros de selección de archivos: JFileChooser

Título XIV: Crear border: BorderFactory

(static) createEtchedBorder()

(static) createTitledBorder(Border b, String titulo)
LineBorder
Título XV: Listas: JList
Barra de desplazamiento
setVisibleRowCount(int datos)
List<E> getSelectedValuesList()
Eventos de JList
Almacenar en String valores seleccionados
Título XVI: Árboles (interfaz carpetas): JTrees
Añadir barra de desplazamiento
Eventos
Título XVII: Tablas: JTable
Añadir barra de desplazamiento
setValueAt(Object value, int row, int column)
setCellSelectionEnabled(boolean n)
setSurrendersFocusOnKeystroke(boolean n)
Eventos
Imprimir (en papel)
Tablas personalizadas
Capítulo VII: Exportar programas
Título I: Applets
Título II: Archivos JAR
Capítulo VIII: Excepciones
Título I: Conceptos básicos
IOException

printStackTrace()
getMessage()
Título II: Excepciones manuales
Título III: Excepciones propias
Título IV: Capturar múltiples excepciones
Título V: Finally

Capítulo Nuevo: Debugging

Capítulo IX: Secuencias/Streams

Título I: Reader: Leer Ficheros Sencillos
Título II: Writer: Escribir en Ficheros Simples
Título III: Buffers/Filtros
Leer Ficheros
readLine()
Título IV: Streams Byte
Leer Archivos
Escribir Archivos
Título V: Serialización
Escribir objetos
Leer objetos
Conflicto actualizando programas
Título VI: File: Consultar directorios y ficheros
getAbsolutePath()
exists()
list()
isDirectory()

Título VII: File: Creación, escritura y eliminación

mkdir()

createNewFile()

delete()

Desktop.getDesktop().open(File f)

Capítulo X: Programación genérica

Título I: ArrayList

Crear un ArrayList

add(Object a)

size()

ensureCapacity(int x)

trimToSize()

set(int x, Object a)

Object get(int x)

Obtener todos los elementos con for

Copiar ArrayList en array convencional

Iteradores, intefaz Iterator

Creación de un iterador

boolean hasnext()

Object next()

remove()

Imprimir datos usando iteradores

Título II: Clases genéricas propias

Creación de métodos genéricos dentro de clases genéricas y no genéricas

Título III: Herencia en clase genérica y tipo comodín

Tipo comodín

Capítulo XI: Programación concurrente, Threads (Hilos)

static sleep(int x) (clase Thread)

Título I: Creación de hilos

detener hilo

getName()

interrupt()

boolean interrupted()

static boolean isInterrupted()

Título II: Sincronización de hilos

ReentrantLock

Condiciones en hilos

Condiciones con métodos de la clase Object

Capítulo XII: Colecciones

Título I: Tipos de colecciones

Título II: List

Características

Clases

Título III: Set

Características

Clases

TreeSet

Título IV: Map

Características

Clases

HashMap

put(K k, V v)

remove(K k)

Set entrySet()

Obtener claves y valor concreto

Título V: Queue

Características

Clases

Título VI: Iteradores, Iterator <E>

<E> next()

boolean hasnext()

remove()

Capítulo XIII: Sockets

Enviar objetos en vez de String

Obtener IP de clientes conectados al servidor

Capítulo XIV: Acceso a BBDD: JDBC

Título I: Pasos previos

Título II: Pasos para acceder a una BBDD

Título III: Primera conexión con una BBDD

Métodos de ResultSet

first()

next()

getString(String campo)

getString(int campo)

getDouble(String campo)

getDate(String campo)

Título IV: Insertar, actualizar, borrar

Título V: Consultas preparadas

Título VI: MVC

Título VII: Procedimientos almacenados

Procedimientos almacenados que reciben parámetros

Título VIII: Transacciones

Título IX: Metadatos

Clases de metadatos

Obtención de metadatos

Capítulo XV: JSP Y Servlets

Capítulo XVI: MVC

Capítulo XVII: Introspección

Obtener a qué clase pertenece un objeto

<T> getClass()

getName()

static Class<?> forName(String Name)

Capítulo XVIII: Java Beans: Programar visualmente

Capítulo I: Programación básica

Título I: Estructura básica

Java distingue mayúsculas y minúsculas.
Java es 100% orientado a objetos.
Al terminar una sentencia (línea) en Java se deberá poner un “;” .
Todo programa Java se debe guardar bajo el nombre de la clase principal (la que lleva el método main) + .java .
Todo programa Java tiene que ser compilado y ejecutado.
Al compilar, por cada clase se generará un archivo .class .
Todo programa Java debe estar dentro de una clase (mínimo).
Todo programa Java debe tener un método main, y será desde este de donde empiece a ejecutarse el programa.
Al igualar una variable a un valor, si ese valor es un número no hará falta hacer nada especial, si es un caracter sólo, habrá que aislarlo entre comillas simple (‘’) y si es una cadena de caracteres, entre comillas dobles (“”).
A la hora de usar una variable, ya sea al darle un valor o usarla dentro de otra, sólo se debe indicar el nombre de la variable.
evite el uso de caracteres extraños.

Programa Java más simple (Hola mundo):

```
public class nombre {
        public static void main(String args[]){
//println hace lo mismo que print pero añade un salto de línea.
//para añadir un salto de línea dentro de un print: “\n”.
                System.out.print(“nombre”);
        }
}
```

Comentarios en Java

(líneas que ignorará el compilador).

- En una línea:
  //cosa**;**
- En varias líneas:
  /*cosa
  cosa
  cosa*/

Importar paquetes

Para usar algunas clases que han desarrollado los desarrolladores de Java para ayudarnos a la hora de programar, es necesario importar paquetes.
Las clases Math, String y otras son pertenecientes a la clase java.lang , la cual es la por defecto de Java y no es necesario exportar, pero cuando queramos hacer uso de una clase fuera de java.lang , debemos exportar el paquete al que pertenecen (antes de la clase principal):
Los paquetes es lo primero que se escribe en un documento.
import java.lang.*;
import java.util.*; (por poner un ejemplo)
import javax.swing.*; (por poner un ejemplo)
El asterisco hace referencia a que importamos todas las clases pertenecientes a ese paquete.
Si necesitamos usar una clase perteneciente a un paquete que en otro paquete recibe el mismo nombre y necesitamos importar los dos paquetes, debemos especificar el nombre de la clase.
Ejemplo: java.util.Timer , javax.swing.Timer (queremos utilizar clases pertenecientes a los dos paquetes y solo la clase Timer del paquete java.util):
import java.util.*;
import javax.swing.*;
import java.util.Timer;

Título II: Variables

Tipos primitivos

8 tipos primitivos (int, short, long, byte, float, double, char, boolean):

- int: Enteros entre = [-2.147.483.648 , 2.147.483.647] (32 bits).
- short: Enteros entre = [-32.768 , 32.767] (16 bits).
- long: Enteros entre = [-2^63 , (2^63)-1] (64 bits), sufijo L o l.
- byte: Enteros entre = [-128 , 127] (8 bits).
- float: Flotantes con ~ 6 cifras decimales (32 bits), sufijo F o f.
- double: Flotantes con ~ 15 cifras decimales (64 bits).
- char: Representar caracteres.
- boolean: Solo admite true o false.

Declarar variables (estructura) (el sufijo solo es necesario en long y float) Cualquier forma es válida:

1. Tipo_var nombre_var = dato *sufijo*;
2. Tipo_var nombre_var;
   nombre_var = dato *sufijo*;
3. Tipo_var nombre_var1, nombre_var2, nombre_var… ;
   nombre_var1 = dato *sufijo*;
   nombre_var2 = dato *sufijo*;

int nombre = 123**;**
short nombre = 123**;**
long nombre = 123**L;**
byte nombre = 123**;**
float nombre = 123**.**123**F;**
double nombre = 123**.**123**;**
char nombre = **‘**a**’;**
boolean nombre = **true;** **boolean** nombre = **false;**

Operadores

- + : suma. - : resta. * : multiplicación. / : división.

- %: resto de división (9%2 = 1).
- > : mayor que. <: menor que. <> : mayour o menor que.
- != : distinto de. == : igual que. && : Y lógico. || : OR lógico.
- ++ : incrementa la variable indicada en 1: nombre**++;**
- **--** : reduce la variable indicada en 1: nombre--;
- +=123 : incremente la variable en 123: nombre+=123;
- -=123 : reduce la variable en 123: nombre-=123;
- + : une o concatena:
  System.out.print("Tengo " + variable_anos + " de edad.");
- **instanceof**: declara si el primer objeto es una instancia de la clase especificada a continuación. (marco es una instancia de JFrame:)
  (marco instanceof JFrame) En este caso, sería true.

Constantes

Variables cuyo valor no puede cambiar.
Se obtienen al usar la palabra reservada "final" delante de la declaración de una variable:

1. **final** Tipo_var nombre_var = dato;
2. **final** Tipo_var nombre_var;
   nombre_var = dato;

```
final int nombre = 123;
final char nombre = 'a';
```

Imprimir variables float o doubles truncadas

Para mostrar en pantalla una variable float o double con los decimales deseados se hará lo siguiente:

```
public class nombre {
    public static void main(String args[]){
        double nombre = 123.123123123;
        System.out.printf("%1.123f", nombre);
```

}
} (123 = nº caracteres decimales deseado)

Refundiciones y castings

Pasar de un tipo de dato a otro (siempre y cuando sea lógico el proceso; no pasar de char a int o cosas similares):

float nombre = 123,123123;
int nombre1 = (int) nombre;
double nombre2 = (double) nombre;

Existen otros métodos como por ejemplo:

int nombre1 = Integer.parseInt(nombre);

Casting de int a String

int numero = 1;
String vernumero = "" + numero;
o
String vernumero = String.valueOf(numero);

Título III: Clase Math (java.lang)

Métodos Clase Math

- **Math.sqrt(123);** Raíz cuadrada. (salida = double)
- **Math.pow(123, 123)** Potencia(base, exp.) (salida = double).
- **Math.sin(123);** Trigonometría (ángulo) (salida = double).
- **Math.round(123);** Redondea(s= double → long, float → int).
- **Math.PI;** Constante con el valor de π.(double).
- **Math.random();** Genera un número entre 0 y 1 (double).

Título IV: Clase String (java.lang)

Objetos String (= variable)

Se crea un objeto del tipo String al igual que una variable de tipo primitivo, pero usando la palabra reservada String, con S mayúscula:

String nombre **= "**nombre**";**

Métodos Clase String

- **.length();** Devuelve la longitud de la cadena de caracteres al objeto que se le aplica este método: **nombre.lenght();**
  También devuelve el número de elementos que hay en un array.
- **.charAt(**123**);** Devuelve el caracter que se encuentra en la posición especificada (Se empieza a contar desde 0).
- **.substring(**x, y**);** Extrae caracteres de un objeto String,
  x=caracter desde que se empieza a extraer.
  y=número de caracteres a extraer.
- **.equals(**"sadfsdfas"**);** Compara el objeto String con la cadena de caracteres de su interior. Devuelve true si son iguales y false si no lo son (diferencia entre mayúsculas y minúsculas).
- **.equalsIgnoreCase("**asdfasdf**");** Igual pero no diferencia mayúsculas y minúsculas.
- **.trim();** Ignora los espacios exagerados.
- **.isEmpty();** Devuelve true si el String está vacío, false si no.

Título V: Entrada de datos

Clase Scanner (java.util)

Declaramos un Objeto perteneciente a la clase **Scanner**.
A continuación, invocamos un método de ella para guarda la entrada en la variable que queramos (cambiando el método dependiendo del tipo de dato):

Scanner nombre = **new Scanner(System.in);**
String nombre1 = **nombre.nextLine();**
int nombre2 = **nombre.nextInt();**

Una vez que hemos terminado de usar la clase Scanner, es conveniente indicar que no la vamos a usar más, cerrando la conexión con la consola para así liberar los recursos que estaban siendo usados:
nombre**.close();**

Clase JOptionPane (javax.swing)

Funciona del tirón (después de importar el paquete) ya que es estático:
String nombre **= JOptionPane.showInputDialog(“**nombre1**”);**
Se abrirá una pestañita con dos botones: aceptar y cancelar y con el nombre1 de mensaje.

Título VI: Condicionales

if, else if y else

Se pone if, luego entre paréntesis una condición (normalmente, usando operadores) si esta condición es verdadera, se ejecutará el código entre llaves, si no, no.
Si queremos evaluar otra condición (no siempre hará falta), a continuación de la última llave, colocaremos else if y realizaremos el mismo proceso que con el if normal. Esta acción se puede representar las veces que se desee. (Se puede anidar un if dentro de otro).
Si queremos que una línea de código se ejecute cuando todas las condiciones a evaluar resultan falsas, usaremos el else, colocándolo al final de la última llave y poniendo entre llaves el código a ejecutar.

if (condición**){**
código;
}else if (*condición****){***
código;
}else{
código;
}

* Si queremos comprobar si una variable del tipo boolean es verdadera, no hará falta poner: variable == true , con indicar el nombre de la variable es suficiente:
if (variable**) {**código**}**

switch

Es un condicional como el if, pero se usa para si una variable vale “x” dato, ejecutar una línea de código, y si esta variable vale “y”, ejecutará otra línea diferente. el valor de las variables sólo podrá tener un dígito para poder usar este condicional.

Se usa: poniendo la palabra reservada switch, entre paréntesis la variable a comprobar, y entre llaves lo siguiente:
se pone la palabra reservada “case” seguida por un espacio del dígito que puede ocupar esta variable, dos puntos, el código a ejecutar y todo caso debe de tener un final marcado gracias a la palabra reservada “break”. Repetir este proceso las veces deseadas.

```
int var = 1;
swith (var) {
        case 1:
                código;
                break;
        case 2:
                código;
                break;
        case 5:
                código;
                break;
}
```

Al final de todos los **case**, podemos usar **default** como una especie de else.

Condicional ternario

Dentro de una instrucción como un System.out.println(); , podemos hacer una condicional si la condición a evaluar es una variable de tipo boolean:

```
boolean x = true;
System.out.println(x ? “si” : “no”);
```

Si x es true, se imprimirá en consola si, y si es false, no.

Título VII: Bucles

while

Se pone la palabra reservada “while” y a continuación una condición y unas llaves. Mientras que la condición de dentro de los paréntesis sea true, el código de entre las llaves se ejecutará infinitamente hasta que la condición sea verdadera.
*Precaución al crear un bucle while y que no sea un bucle infinito.
while (condición**) {**código**}**

do while

Usar la palabra reservada “do”, código entre corchetes y al final la palabra reservada “while” y una condición entre paréntesis.

```
do{
    código
} while (condición);
```

La diferencia con el bucle anterior es que aunque la condición sea falsa, el código se ejecutará 1 vez (si es verdadera, es exactamente igual que el bucle while).

for

Se usa la palabra reservada “for”, entre paréntesis 3 instrucciones:
la primera instrucción consistirá en inicializar una variable (esta variable sólo será accesible para el bucle en caso de que se inicialice dentro de este, por lo que fuera del bucle no se podrá usar esta (por lo general, esta variable se suele llamar i o j y es de tipo int)).
Se separa la primera de la segunda con una coma “,” y se pone la segunda, que consiste en una condición, la cual estará relacionada con nuestra variable i. Otra coma “,” y la tercera instrucción, que consiste en modificar el valor de la i, ya sea aumentándolo o disminuyéndolo (se suele usar i++ o i--, también i+=123 o i-=123) la tercera instrucción se ejecutará cada vez que el bucle de una vuelta.
Tras esto, el código a ejecutar entre llaves. El bucle se estará ejecutando mientras que la condición sea verdadera.
(Se puede anidar un for dentro de otro).

for (int i = 0, i < 123**, i++) {**código**}**

for each

Su apariencia es la misma que la del for, a veces recibe el nombre “for mejorado”. El único cambio respecto del for son las instrucciones que se introducen entre paréntesis, la cual es sólo una. Este bucle se suele usar para recorrer un array/arreglo.
Se pone la palabra reservada “for” y entre paréntesis se declara una variable del mismo tipo que el bucle a recorrer (esta variable será de una dimensión inferior al array (si el array es de una dimensión, será una variable normal, si es de 2D, la variable será una array de 1D)), se le da un nombre, se colocan dos puntos “:” y el nombre de la matriz/arreglo y entre paréntesis el código a ejecutar:
for (Tipo_var nombre **:** variable**){**código**}**

Uso de for each para recorrer matrices

```
int[] a = {1, 2, 3, 4, 5};
for (int e : a) {
        System.out.print(e + “ ”);
}
//Salida = 1 2 3 4 5

int[][] b = {{1, 2, 3, }, {4, 5, 6}}
for (int[] e : a){
        System.out.println();
        for (int z : e){
                System.out.print(z + “ ”);
        }
}
/*Salida = 1 2 3
                4 5 6*/
```

Título VIII: Arrays/Arreglos/Matrices

Unidimensionales

Son un tipo de variable la cual almacena varios datos que están relacionados entre sí. Un arreglo se inicializa de forma parecida a una variable. Hay dos formas principales de inicializarlos:

1. Se indica el tipo de dato (int, String, char...), unos corchetes de apertura y cierre "[]" (si se usa una vez, el array tendrá una dimensión, "[][]" dos dimensiones, "[][][]" tres...) el nombre del array, igual a palabra reservada "new" el tipo de dato otra vez, y unos corchetes entre los cuales se especifica el número de slots o huecos que tendrá el array;
   Luego, ir almacenando los valores de cada slot del array, indicando el slot en el que se desea guardar (Se empieza a contar desde 0):
   int[] array = new int[3];
   array[0] = 123**;**
   array[1] = 123**;**
   array[2] = 123**;**

2. Se indica el tipo de dato, corchetes, nombre, igual, entre llaves y separado por comillas, los diferentes datos que se guardarán:
   int[] array = {123**,** 123**,** 123**};**

Bidimensionales

1. **int[][] array = new int[2][2];**
   array[0][0] = 123**;**
   array[0][1] = 123**;**
   array[1][0] = 123**;**
   array[1][1] = 123**;**

2. **int[][] array = {{**123**,** 123**}, {**123**,** 123**};**

Título IX: Palabras reservadas

Relacionadas con variables

boolean, byte, char, double, float, int, long, short, final.

Relacionadas con procedimientos

break, case, do, else, for, if, return, switch, void, while.

Relacionadas con clases, objetos e interfaces

abstract, class, extends, implements, instanceof, interface, new, super, static, this.

Relacionadas con paquetes

import, package.

Modificadores de acceso

private, protected, public.

Capítulo II: POO

Programación orientada a procedimientos (batch).
Programación orientada a objetos (java).
Cada objeto, al igual que un objeto de la vida real, posee unas características y unos usos (Un coche puede poseer un color y una altura y se puede usar para ir de un sitio a otro).

Título I: Clases y Objetos

Definición

Clase: parte de código donde se redactan las características y funciones de un grupo común de objetos.
Objeto: Ejemplar perteneciente a una clase. Poseen unos atributos (variables) y unos usos (métodos (parecido a funciones)).

Inicializar un objeto perteneciente a una clase

Para ello, debemos haber creado primero una clase a nuestro gusto. Nos iremos a la clase principal e inicializamos un objeto perteneciente a la clase de la siguiente manera (es necesario que ambas clases estén en el mismo paquete y que se compilen a la vez.):
Indicamos el nombre de la clase a la que pertenece el objeto que queramos crear, introducimos el nombre que le queramos dar a ese objeto (el que sea), "=" palabra reservada "new", otra vez el nombre de la clase al que pertenece este objeto, y paréntesis "()" (entre estos paréntesis se deberán introducir los datos que recibe el método/s constructor, en su orden adecuado y separando los datos por comas, en el caso de que el método constructor de nuestra clase reciba datos, si no, se dejan los paréntesis vacíos.). Ejemplos prácticos (en el primer objeto, usamos un constructor que no recibe parámetros y en el segundo, otro en la misma clase que recibe un parámetro *Los métodos constructores se desarrollan adelante*):
class Prueba{código**}**

```
public class Main{
	public static void main(String[] args){
		int x = 4;
		Prueba Objeto1 = new Prueba();
		Prueba Objeto2 = new Prueba(x);
	}
}
```

Nomenclatura de variables y métodos pertenecientes a una clase

*Todo lo explicado a continuación sólo funcionará si ambas clases se encuentran en el mismo paquete y son compiladas a la vez.

Variables estáticas

A la hora de querer acceder a una variable estática (*palabra reservada static explicada en otro Capítulo*) de un campo de clase desde otra clase, se indicará primero el nombre de la clase en la que se encuentra la variable o el nombre de un objeto perteneciente a esta clase, punto "." y el nombre de la variable:
(La clase se llama "Clase" y la variable se llama "variable"):

Clase.variable;

O:

Clase Objeto = new Clase();
Objeto.variable;

Variables públicas

A la hora de querer acceder a una variable pública (algo poco recomendable (el que haya una variable pública)) de un campo de clase desde otra clase, se indicará primero el nombre de un objeto perteneciente a esta clase, punto "." y el nombre de la variable:
(La clase se llama "Clase" y la variable se llama "variable"):

Clase Objeto = new Objeto();
Objeto.variable;

Métodos

A la hora de llamar a un método (SETTER o GETTER) perteneciente a una clase desde otra clase, Se indicará el nombre de un objeto perteneciente a esta clase al cual se le quiere aplicar el método, punto ".", el nombre del método y paréntesis "()" (en estos paréntesis debemos poner los datos que reciba el método en correcto orden y separados por comas, siempre y cuando el método indicado reciba parámetros.):
(El objeto se llama "Objeto" y el método se llama "getMetodo" (no recibe nada.)):
Objeto.getMetodo();

Título II: Creación de una Clase

La clase en sí

Si es otra diferente a la clase principal, se escribe la palabra reservada "class" y a continuación el nombre de la clase, (*) seguido de unos corchetes (el interior de estos es la clase).

(*) Si queremos que la clase herede de otra, deberemos colocar "extends" y el nombre de la clase de la que hereda (habiendo importado el paquete anteriormente).

(**) Si queremos que la clase implemente de una interfaz, deberemos colocar "implements" y el nombre de la interfaz que implementa (habiendo importado el paquete anteriormente).

Dentro de una clase, podemos encontrar diferentes cosas (método constructor, campos de clase y métodos (GETTERS y SETTERS)). Las diferentes cosas que podemos encontrar dentro de una clase da igual el orden en el que se encuentren, funcionará de la misma manera.

Campos de clase

Es el conjunto de variables de la clase, los llamados atributos de un objeto.

Por lo general, es recomendable que todas estas lleven el modificador de acceso “private” (salvo si son variables de tipo static (las cuales se especificarán en otro título.)), para así, encapsular la clase y que estas variables no se puedan modificar ni ver desde otra clase externa (más adelante se explica cómo mostrar una variable específica a otras clases gracias a métodos GETTERS). Estas variables pueden tener un valor constante, un valor predefinido o que otra clase defina sus valores mediante un método constructor o métodos SETTERS.
ejemplo de inicialización de campos de clase:
private int altura, anchura, largo;
private String nombre, apellido;

Método Constructor

Es el encargado de asignar un valor a algunas variables del campo de clase cuando se construya un objeto perteneciente a esta clase. Si una clase carece de método constructor, sus variables obtendrán un estado inicial por defecto (null).
Un método constructor se construye de la siguiente manera:
Primero se hace uso del modificador de acceso “public”, luego se indica el nombre del método constructor, el cual debe coincidir con el nombre de la clase donde se encuentra, a continuación, se abren y se cierran paréntesis en caso de que el método constructor no vaya a recibir ninguna variable de otra clase “()” (si va a recibir variables de otra clase, entre estos paréntesis se indican qué variables, indicando el tipo de variable y su nombre, separando las variables por comas. Ej: (**int edad, String nombre**)), a continuación, se abren y cierran llaves, dentro de estas estará el código a ejecutar.
Existen dos clases principales de métodos constructores:

1. Los que asignan un valor constante a cada campo de clase (no reciben datos):
   class Clase{
   private int alto, ancho;
   public Clase(){

```
            alto = 123;
            ancho = 123;
        }
    }
```

2. Los que asignan un valor diferente a cada campo de clase, dependiendo de los datos enviados desde la clase en la que se inicializa el objeto perteneciente a nuestra clase (reciben datos):

```
class Clase{
        private int alto, ancho;
        public Clase(int altura, int anchura){
            alto = altura;
            ancho = anchura;
        }
}
```

*Es posible que las variables que recibe el método constructor y alguna variable perteneciente al campo de clase tengan el mismo nombre. En estos casos, haremos uso de la palabra reservada “this”, para indicar que la variable que lleve esta palabra reservada es la perteneciente al campo de clase:

```
class Clase{
        private int alto, ancho;
        public Clase(int alto, int ancho){
            this.alto = alto;
            this.ancho = ancho;
        }
}
```

Sobrecarga de constructores

Es posible que dentro de una misma clase haya varios métodos constructores (no hay límite), siempre y cuando estos reciban diferente tipo o número de datos, si hay dos métodos que reciben el mismo tipo de datos en el mismo orden y el mismo número de datos, dará error.

Si dentro de una misma clase hay diferentes métodos constructores, podemos hacer que estos métodos se llamen entre sí utilizando la palabra reservada “this” y dos paréntesis “()”, siempre indicando dentro de estos los datos que recibe el método constructor:

```
public MetodoConstructor (int alto){
        this.alto = alto;
}
public MetodoConstructor (int alto, int largo){
        this(alto);
        this.largo = largo;
}
```

Métodos

Un método, es una especie de “función” perteneciente a una clase.
Existen dos tipos: GETTER y SETTER.
El GETTER ofrece información y el SETTER modifica información del objeto; no es recomendable el mezclar ambos en uno, pero es posible.

GETTER

Es un método que nos permite acceder a cierta información (la que desee el creador de la clase) del campo de clase, devolviéndonos una variable a la clase donde se llama al método GETTER.
Un método GETTER se construye de la siguiente manera: palabra reservada “public”, tipo de variable que devuelve (int, String...), nombre del método (el que queramos), paréntesis “()” (Si el método necesita recibir algún dato desde la clase desde la que se le llama, entre estos paréntesis se debe indicar el tipo de variable que estos reciben y el nombre de esta, separando las diferentes variables por comas. Ej: (**int x, String nombre**)) corchetes “{}” y dentro de estos, el código a realizar (si fuese necesario), para terminar el método, debemos terminar con la palabra reservada “return”, seguida de la información a devolver (recordemos que esta información va a ser

guardada en correspondencia con el tipo de variable indicado posteriormente, así que si indicamos que es de tipo int, por ejemplo, no podemos poner una cadena de caracteres después del return):

public Tipo_var nombre () {
    código;
    return nombre_var;
}

SETTER

Este método nos permite acceder al campo de clase de un objeto desde una clase a la que este no pertenezca, ejecutando el código que se encuentre dentro de él.

Se construye de la siguiente manera: "public", "void" (ya que no devuelve ningún dato), nombre del método (el que desee), paréntesis "()" (Si el método necesita recibir algún dato desde la clase desde la que se le llama, entre estos paréntesis se debe indicar el tipo de variable que estos reciben y el nombre de esta, separando las diferentes variables por comas. Ej: (**int x, String nombre**)), corchetes "{}" y dentro el código a realizar. Ej:

public void nombre () {
    código;
}

Título III: Herencia

Toda clase que herede de otra tendrá por defecto, sus métodos constructores, sus métodos y sus campos de clase.

Eso sí, habrá que llamar al método constructor de la clase padre deseado. Para esto, llamaremos al método constructor de la clase padre con la siguiente instrucción:

super();

(*)Si el método constructor padre recibe parámetros, se deben pasar entre los paréntesis.

Polimorfismo

Se puede almacenar un objeto de una subclase siempre que se espere un objeto de una una superclase:
(Superclase = Uso, Subclase = Uso_coche):
Uso nombre **= new Uso_coche();**

Clases abstractas

Una clase abstracta es una clase que contiene ciertos métodos a usar, los cuales tendrán que ser declarados obligatoriamente en las clases que hereden de esta:

```
public abstract class Prueba{
        public abstract int Metodo();
}
```

Título IV: Interfaces

Ya que en Java no existe la heredación múltiple, se crearon las interfaces, que son una especie de "clase" la cual se puede implementar dentro de una clase.
Las interfaces solo pueden contener constantes y métodos abstractos y públicos, por lo tanto, su puede prescindir del uso de las palabras reservadas public, final y abstract.

```
interface nombre {
        int pi = 3.1415;
        int Metodo(String x);
}
```

Capítulo III: Interfaces gráficas básicas (librería javax.swing)

(Las clase de javax.swing heredan a su vez de las clases java.awt, por lo que es normal encontrar ambos paquetes nombrados a la vez en un mismo documento).
Esta clase es una de las principales para construir interfaces gráficas o frames.
Cabe destacar que estos frames aparecen invisibles y con un tamaño default de 0px x 0px, y conviene decir qué se hará cuando una de estas pestañas se cierre.

Título I: Creación de frame: clase JFrame

*JFrame conlleva una larga cola hereditaria, por lo que puede hacer uso de métodos que no se encuentren en la clase JFrame, si no de clases padre.
Para crear un frame, es necesario crear una clase que herede JFrame y posteriormente, inicializar un objeto perteneciente a esta clase dentro de la clase principal.
Para recurrir a los diferentes métodos que posee la clase JFrame, se pueden acceder a los métodos directamente indicando su nombre y parámetros dentro del método constructor de la clase que hereda de JFrame, o indicando el nombre de un objeto perteneciente a la clase que hereda de JFrame, “.” y a continuación, el método y parámetros.

Principales métodos de JFrame

*El nombre de las variables que reciben los métodos no son los reales, el tipo de variable sí. Si no recibe nada, es un SETTER.

setVisible(boolean x)

Establece si el frame será visible (true) o no (false).

Se aconseja ponerlo al final de todos los métodos, ya que hay un bug que hace que la lámina aparezca vacía y haya que redimensionarla para ver los elementos.

setSize(int x, int y)

Establece el tamaño del frame. x es para el ancho, y para el largo (medido en píxeles (px)).

setDefaultCloseOperation(int x)

Establece lo que se hará cuando se cierre un frame. (el int sólo puede tomar cuatro valores, el principal = JFrame.EXIT_ON_CLOSE (cierra la aplicación ENTERA, si queremos que solo cierre esa ventana, usamos DISPOSE_ON_CLOSE) (consultar API)).

setLocation(int x, int y)

Establece el lugar del componente padre (el medio en el que va a estar contenido este frame) en el que spawneará el nuevo frame.
x = ancho, y = largo.

setBounds(int x, int y, int ancho, int largo)

Establece la localización (x,y) y el tamaño del frame.

setResizable(boolean x)

Establece si es posible que nuestro frame sea redimensionado por el ratón del usuario.

setExtendedState(int x)

Entre otras posibilidades (consultar API), establece que el frame spawneará maximizado si se le pasa el int:
Frame.MAXIMIZED_BOTH (= 6).

setTitle(String x)

Establece el título (texto arriba a la izqda) del frame.

setIconImage(Image x)

Establece la imagen del frame, recibe un objeto de la clase Image.

dispose()

Cierra la ventana.

pack()

Indica que el tamaño del marco será el mínimo para contener a los componentes de su interior.

Título II: Clase Toolkit

(static) getDefaultToolkit()

Devuelve el medio por defecto.

beep()

Interpreta el sonido por defecto de tu OS.

Ejecutar sobre el objeto que devuelve **getDefaultToolkit()** :
Toolkit.getDefaultToolkit().beep();

getScreenSize()

Devuelve el tamaño del monitor principal (devuelve objeto del tipo Dimension (paquete java.awt)). Dentro de la clase Dimension, hay dos campos de clase (variables) que son capaces de devolver el alto (height) y el ancho (width) de la pantalla.
Ej práctico como sacar el tamaño de la pantalla de un equipo:

```
Toolkit mipantalla = Toolkit.getDefautlToolkit();
Dimension tamano = mipantalla.getScreenSize();
int altura = tamano.height;
int anchura = tamano.width;
```

*Si queremos crear un frame que se coloque justo en el medio de la pantalla, podemos hacer este procedimiento y, con la ayuda del método setLocation() de la clase JFrame, dictar que la localización de este frame (int x, int y) van a ser igual a un cuarto de altura y de anchura (**setLocation(altura/4, anchura/4)**).

getImage(String nombre_archivo);

Devuelve un objeto de tipo Image (el nombre_archivo, parte de la carpeta principal, este puede ser igual a una ruta y a la imagen (o solo a la imagen si se encuentra en la carpeta principal): carpeta/carpeta/imagen.jpg (admite gif, jpg y png)).
Ejemplo práctico poner imagen (dentro de un método constructor de una clase que herede de JFrame):
Toolkit objeto = Toolkit.getDefaultToolkit();
Image Icono = objeto.getImage(*imagen.jpg***);**
setIconImage(Icono);

Título III: Creación de láminas

Clases: JPanel y Graphics
*JPanel conlleva una larga cola hereditaria, por lo que puede hacer uso de métodos que no se encuentren en la clase JPanel, si no de clases padre.
Para usar la clase JPanel, tenemos que crear una clase que herede de JPanel, al igual que con la clase JFrame.

*El nombre de las variables que reciben los métodos no son los reales, el tipo de variable sí. Si no recibe nada, es un SETTER.

paintComponent(Graphics g);

Hace posible escribir un componente en el frame. para esto, es necesario sobreescribirlo.

Para usar este método, es necesario indicarle que haga lo que le programaron que hiciera, y luego lo que nosotros queramos (como añadir texto). Ej:

```
public void paintComponent(Graphics g){
        super.paintComponent(g);
        código
        g.drawString("Hola", 10, 10);
}
```

Para implementar nuestra lámina en un frame, deberemos inicializarla dentro de la clase a la que pertenece nuestro frame que hereda de JFrame, y finalmente le decimos que la añada. Ej:
(La clase a la que pertenece nuestra lámina se llama Lamina y estamos dentro de la clase del frame:)

```
{
        Lamina milamina = new Lamina();
        add(milamina);
}
```

Limpiar láminas

```
nombrelamina.removeAll();
nombrelamina.repaint();
```

Título IV: Dibujar y escribir
Principales métodos de Graphics

Para implementar estos métodos de la clase Graphics en una lámina, es necesario meterlos dentro del método paintComponent() de la clase JPanel, tal y como se muestra en el ejemplo anterior a este apartado.

drawString(String str, int x, int y)

Donde str es el texto que queremos mostrar, x e y posiciones relativas dentro del frame padre. (también pertenece a Graphics2D)

*se puede colorear con el método setColor() de Graphics2D.

drawRect(int x, int y, int ancho, int alto)

X e Y posiciones relativas dentro del frame padre y ancho y alto las medidas del rectángulo.

drawLine(int x, int y, int a, int b);

Donde (x, y) definen un punto y (a, b) el otro punto de la recta.

Título V: Biblioteca Java 2D y clase Graphics2D

Biblioteca Java 2D = java.awt.geom.*;

Al igual que con los métodos de la clase Graphics, cualquiera de los métodos de una clase perteneciente a la biblioteca de Java 2D, deben ir dentro del método paintComponent() de la clase JPanel.

Lo primero es crear un objeto de la clase Graphics2D dentro de nuestro método paintComponent(), esto lo hacemos mediante una refundición de g (Graphics) a Graphics2D:

Graphics2D nombre = (Graphics2D) g;

draw(Shape s)

(Basándonos en el ejemplo de la clase Graphics2D y el de la clase Rectangle2D):

nombre.draw(rect);

clase Rectangle2D

Primero, creamos un objeto perteneciente a la clase Rectangle2D.Double (clase Double interna en clase Rectangle2D):

Rectangle2D rect = new Rectangle2D.Double(x, y, a, b);

(La instanciamos así ya que la clase Rectangle2D es abstracta, por lo cual no podemos instanciarla, pero sí a una clase que herede de ella, como puede hacer una clase interna).

Donde x,y,a,b son parámetros del tipo double, x e y son las posiciones relativas dentro del frame padre, a el ancho del rectángulo y b el alto.
Ej: (estamos dentro del método paintComponent() de la clase JPanel)

```
super.paintComponent(g);
Graphics2D G2D = (Graphics2D) g;
Rectangle2D rectangulo = Rectangle2D.Double(10, 10, 200, 200);
G2D.draw(rectangulo);
```

getCenterX()

Devuelve un dato double que indica el centro del rectángulo al que se le aplica el método en el eje X.

getCenterY()

Devuelve un dato double que indica el centro del rectángulo al que se le aplica el método en el eje Y.

clase Ellipse2D

Funciona igual que Rectangle2D, pero al ser una elipse, no podemos indicar x,y,a,b, debemos hacer uso del método setFrame(Rectangle2D r) o setFrame(x, y, a, b) (x,y,a,b = double) , para dibujar una elipse inscrita en este rectángulo.
*Si hacemos la inscribimos en un cuadrado, obtendremos un círculo.
(también deberemos indicar G2D.draw(nombre) para mostrarlo):
(dentro de paintComponent())

```
super.paintComponent(g);
Graphics2D G2D = (Graphics2D) g;
Ellipse2D elipse = Ellipse2D.Double();
elipse.setFrame(10, 10, 200, 200);
G2D.draw(elipse);
```

clase Line2D

Se puede inicializar directamente dentro del draw():
(dentro de paintComponent())

G2D.draw(new Line2D.Double(100, 100, 200, 200));
(recibe los mismos parámetros que drawLine de la clase Graphics pero de tipo double).

Título VI: Colorear, clase Color y objetos java2D

setPaint(Color x)

Para indicar que las próximas figuras serán del color indicado (x).
Dentro de la clase Color, existen varias constantes que guardan los datos rgb de algunos colores. Ej (dentro del método paintComponent() para dibujar un cuadrado con el contorno rojo):

```
super.paintComponent(g);
Graphics2D G2D =(Graphics2D) g;
Rectangle2D rect = new Rectangle2D.Double(0, 0, 100, 100);
G2D.setPaint(Color.RED);
G2D.draw(rectangulo);
```

Si queremos rellenar el cuadrado de un color, sustituiríamos el draw() (dibuja el contorno solo) por un fill() (dibuja todo):

```
G2D.setPaint(Color.RED);
G2D.fill(rectangulo);
```

Obtener colores personalizados (clase Color)

Mediante el método constructor: **new Color(r, g, b);**
El ejemplo de antes: **G2D.setPaint(new Color(255, 0, 0));**

métodos brighter() y darker();

Después de la llamada al método constructor de color, se puede emplear cualquiera de estos métodos para aumentar el brillo o reducirlo:

G2D.setPaint(new Color(255, 0, 0).brighter().brighter());
*Se pueden implementar las veces que se quiera.

Colorear fondo del frame coloreando la lámina

Dentro de la clase que hereda de JFrame, podremos usar el método setBackground(Color x) sobre nuestro objeto lámina, coloreando así el fondo de esta:
(en el método contructor de la clase del frame)

```
Lamina lamina = new Lamina();
add(lamina);
lamina.setBackground(new Color(255, 255, 0));
```

También se le puede pasar el color de fondo de nuestro OS: SystemColor.window .

Determinar color por defecto para cualquier objeto

Dentro de la clase que hereda de JFrame, podremos usar el método setForeground(Color x) sobre nuestro objeto lámina, determinando el color por defecto:
(en el método contructor de la clase del frame)

```
Lamina lamina = new Lamina();
add(lamina);
lamina.setForeground(new Color(255, 0, 255));
```

Título VII: Cambiar fuentes (tipos de letra)

Saber tipos de letras instalados en OS

(dentro del método main)
(importando java.awt.GraphicsEnvironment;)

```
String[] nombres =
GraphicsEnvironment.getLocalGraphicsEnvironment().getAvailableF
ontFamilyNames();
for (String[] e: nombres){
        System.out.println(e);
}
```

Crear objeto perteneciente a Font

```
Font fuente = new Font(String x, int style, int size);
```

Font fuente = new Font(“Arial”, Font.BOLD, 16);
(Font.PLAIN para dejar el texto normal
Font.ITALIC para cursiva… consultar API).

Aplicar fuente

utilizamos el método setFont(Font fuente) sobre un objeto del tipo Graphics2D.
(setFont funciona igual que setColor).
(dentro de paintComponent() y después de refundir g):
Font fuente = new Font(“Arial”, Font.BOLD, 26);
G2D.setFont(fuente);
G2D.drawString(“Hola”, 100, 100);

Copiar láminas a otras láminas

Poseemos dos láminas llamadas LN y LS.
Para hacer que LS pueda copiar su contenido y copiarlo en LN, al invocar a la lámina LS en su clase padre, le tendremos que pasar la misma instancia que mostramos en la lámina LN.

Posteriormente, debemos crear un método en la lámina LS que lleve la instrucción paint(Graphics g) al cual le pasaremos como g el nombre que le dimos a la instancia de LN seguido del método GETTER getGraphics().

Título VIII: Trabajar con imágenes

Clases: Image, ImageIO, Graphics y File.

Almacenar imágenes en Image a partir de una ruta

Primero, dentro de la clase de nuestra lámina, crear un campo de clase privado del tipo Image.

Dentro del método paintComponent(), hacer uso del método estático .read de la clase ImageIO y pasarle como ruta un objeto del tipo File para almacenar la imagen dentro de la variable anterior.
Como el método read() lanza una excepción, tendremos que preparar el código para recibirla:

```
private Image imagen;
public void paintComponent(Graphics g){
        File archivo = new File("Carpeta_principal/asdf/imagen.png");
        try{
                imagen = ImageIO.read(archivo);
        }catch(IOException e){
                código;
        }
}
```

*Es aconsejable mover el try junto con el catch dentro del método constructor de la clase de la lámina, para así almacenar primero las imágenes que vayamos a usar en la lámina y luego, usarlas con drawImage().

drawImage(Image imagen, int x, int y, ImageObserver observer);
imagen: nuestra imagen almacenada en un objeto Image, x e y, coordenadas relativas y observer hace referencia a si queremos ver cómo va el proceso de escribir la imagen, como no queremos, le pasaremos un "null":
(después del código del apartado anterior, dentro de paintComponent:)

```
g.drawImage(imagen, 100, 100, null);
```

Obtener tamaño de la imagen: getWidth() y getHeight()
se le puede aplicar a un objeto de tipo Imagen para obtener su ancho y largo:

```
Image imagen;
imagen=imageIO.read(new Filet(ruta/ruta/image.png));
int ancho = imagen.getWidth(this);
```

```
int ancho = imagen.getHeight(this);
```

Capítulo IV: Eventos

Título I: Creación de botones

Dentro de nuestra clase lámina:
Instanciar un objeto perteneciente a la clase **JButton** como campo de clase y le pasamos al constructor el texto que queramos que tenga el botón.
Posteriormente, dentro del constructor de la lámina, añadir la instrucción add() con el nombre de nuestro botón:

```
public Lamina extends JPanel{
        JButton boton = new JButton("Púlsame");
        public Lamina(){
                add(boton);
        }
}
```

Título II: Creación de eventos de botón

Tomando como ejemplo el botón del ejemplo anterior, añadir dentro del constructor el método **addActionListener**(oyente) para indicar quién va a ser el oyente del evento producido por el botón. Como, en este caso, queremos que el oyente sea la lámina, podemos usar el operador "this" , ya que nos encontramos dentro de esta (si queremos que el oyente sea otra clase, podemos instanciar esa clase en un objeto y pasarle el nombre del objeto en lugar del "this").
(dentro del método constructor):

```
boton.addActionListener(this);
```

Ahora, toca indicarle a nuestro oyente, en este caso el objeto Lámina, que va a recibir un evento, esto lo hacemos haciendo que el objeto implemente la interfaz **ActionListener** , por lo que tendremos que sobreescribir el método **actionPerformed(ActionEvent e)** (es ActionEvent ya que es un evento de ratón).

Dentro de este método introduciríamos el código a realizar.
En el caso de que un mismo receptor reciba varios eventos de diferentes fuentes y queramos que cada fuente haga una acción diferente, deberemos indicar quién es la fuente del evento mediante el método getSource() de ActionEvent:
(dentro de actionPerformed(ActionEvent e)):
Object fuente = e.getSource();
Y hacer las comprobaciones mediante if para ver qué botón fue pulsado:
if (fuente == boton){código**}**

Clases internas anónimas

A la hora de crear clases que hereden de otras simplemente para sobreescribir un método para que podamos usar otro método en otra clase, como suele pasar mucho en los eventos, podemos crear clases internas anónimas, las cuales nos ayudan a resumir código.
Ejemplo de clase interna anónima: (tenemos un JButton llamado boton:)

```
boton.addActionListener(new ActionListener(){
        actionPerformed(ActionEvent e){
                código
        }
});
```

Al indicar "new ActionListener", indicamos la clase (o interfaz) de la que hereda (o implementa).

Título III: Creación de eventos de ventana

Al igual que en los eventos de ratón, la clase oyente debía implementar la interfaz ActionListener, la clase oyente en eventos de ventana, debe implementar la interfaz **WindowListener** (la cual posee 7 métodos) (o heredar de la clase adaptadora **WindowAdapter** (explicada en el siguiente título)).

Deberemos indicar al marco sobre el que se realiza la acción, que va a ser la fuente del evento:
Esto lo hacemos inicializando un objeto perteneciente a la clase oyente y, posteriormente, le indicamos cual es la clase oyente con la instrucción **addWindowListener()**:
(dentro del método constructor del marco):
Oyente nombre = new Oyente();
addWindowListener(nombre);

Métodos WindowListener

windowActivated(WindowEvent e)

Invocado cuando la ventana se activa (se selecciona).

windowClosed(WindowEvent e)

Invocado cuando la ventana se cierra.

windowClosing(WindowEvent e)

Invocado cuando el programa se cierra.

windowDeactivated(WindowEvent e)

Invocado cuando la ventana se deselecciona.

windowDeiconified(WindowEvent e)

Invocado cuando la ventana se des-minimiza.

windowIconified(WindowEvent e)

Invocado cuando la ventana se minimiza.

windowOpened(WindowEvent e)

Invocado cuando la ventana se abre.

Detectar cambios de estado

Al igual que para los eventos anteriores, necesitamos la creación de una nueva clase oyente la cual importará una interfaz o heredará de una clase adaptadora. En este caso, importará la interfaz **WindowStateListener**, ya que esta sólo tiene un método: **windowStateChanged(WindowEvent e)** .
Y al igual que las fuentes anteriores, tiene que existir la sentencia: **addWindowStateListener();** dentro de la clase del frame, a la cual le pasaremos un objeto perteneciente a la clase oyente.

Título IV: Clases adaptadoras

Son clases que implementan diversas interfaces. Esto nos permite crear una clase que herede de estas clases para así poder hacer uso de un método de una interfaz sin tener que reescribir todos los métodos de esa interfaz, sólo el que necesitemos.
Ejemplo de clases adaptadoras: **KeyAdapter**, **WindowAdapter**, **MouseAdapter**.

Título V: Creación de eventos de teclado

Usaremos la interfaz **KeyListener**, la cual conlleva 3 métodos.
Como en los eventos de ventana, creamos una clase oyente que implemente esta interfaz y dentro de esta, sus 3 métodos.
Y al igual que en los eventos de ventana, dentro del marco introducimos la instrucción **addKeyListener()** , a la cual le pasamos un objeto perteneciente a la clase oyente.

Saber cual es la tecla pulsada

La clase KeyEvent tiene un método **getKeyCode()** que nos devuelve un número entero dependiendo de qué tecla hayamos pulsado. (consultar API).
Esta clase también tiene un método **getKeyChar()** que nos devuelve una variable char con el nombre de la tecla (a,b,c,1...) (consultar API).

Métodos KeyListener

keyPressed(KeyEvent e)

Invocado cuando una tecla es presionada.

keyReleased(KeyEvent e)

Invocado cuando una tecla es soltada.

keyTyped(KeyEvent e)

Invocado cuando una tecla es presionada y soltada.

Título VI: Creación de eventos de ratón

Haremos uso de la interfaz **MouseListener** o de la clase adaptadora **MouseAdapter** (según convenga).
Esta interfaz o clase deberán ser aplicadas a la clase oyente.
Al igual que en eventos de teclado y de ventana, tenemos que crear un objeto perteneciente a la clase oyente dentro de la clase fuente para pasarle este como parámetro en la instrucción: **addMouseListener();**

Algunos métodos de MouseEvent

getX()

Nos muestra el valor del eje x donde hemos pulsado.

getY()

Nos muestra el valor del eje y donde hemos pulsado.

getClickCount()

Nos muestra la cantidad de clicks seguidos que hemos realizado.

getModifiersEx()

Nos devuelve un int dependiendo de qué botón hemos pulsado (izqda, dcha, rueda...) (consultar API) (ej: MouseEvent.BUTTON1(2,3)_DOWN_MASK).

Métodos MouseListener

mouseClicked(MouseEvent e)

Invocado cuando el ratón ha sido pulsado (y levantado)

mouseEntered(MouseEvent e)

Invocado cuando el ratón pasa por encima de un componente.

mouseExited(MouseEvent e)

Invocado cuando el ratón sale de un componente.

mousePressed(MouseEvent e)

Invocado cuando el ratón pulsa un componente (sin soltar).

mouseReleased(MouseEvent e)

Invocado cuando el ratón suelta un componente.

Detectar si el ratón está arrastrando o sólo moviendo

Al igual que con eventos de teclado o ventana, creamos una clase oyente que implemente la interfaz **MouseMotionListener** , la cual implementa 2 métodos.
En la clase fuente, deberemos indicarle la instrucción **addMouseMotionListener()** y pasarle un objeto perteneciente a la clase oyente.

Métodos MouseMotionListener

mouseDragged(MouseEvent e)

Invocado cuando el ratón está arrastrando algo (se repite en bucle hasta hasta su fin).

mouseMoved(MouseEvent e)

Invocado cuando el ratón se está moviendo por encima del componente fuente (se repite en bucle mientras se mueve).

Título VII: Creación de eventos de foco

Foco sobre Componentes

El foco es si una ventana está seleccionada o no por el usuario.
Usaremos la interfaz **FocusListener**, o en su defecto, la clase adaptadora **FocusAdapter**.
Para indicar cual va a ser la clase oyente, instanciamos esta clase y la indicamos con la instrucción **addFocusListener()** .

Métodos FocusListener

focusGained(FocusEvent e)

Se desencadena cuando el foco se gana.

focusLost(FocusEvent e)

Se desencadena cuando el foco se pierde.

Foco sobre Ventanas

Usaremos la interfaz **windowFocusListener**, o la clase adaptadora **WindowAdapter**.
Indicaremos cual es la clase oyente con la instrucción **addWindowFocusListener()** .

Métodos WindowFocusListener

windowGainedFocus(FocusEvent e)

Se desencadena cuando el foco se gana.

windowLostFocus(FocusEvent e)

Se desencadena cuando el foco se gana.

Título VIII: Múltiples Fuentes

Para esto, haremos uso de la interfaz **Action** , o hacer uso de la clase **AbstractAction**, la cual nos implementa los 6 métodos y nos obliga a implementar el método **actionPerformed(ActionEvent e).**
Crearemos nuestra clase oyente, que hereda de **AbstractAction**, en el constructor, indicamos mediante putValue , alguna información sobre el botón, como el nombre, qué es lo que hace (esto mediante la constante Action.SHORT_DESCRIPTION, la cual hace que si colocamos el cursor sobre el botón, nos aparezca el valor de lo que hayamos puesto con putValue())(Action.NAME para establecer el texto del botón)...
También, crearemos el método **actionPerformed()** , ya que esta clase será la clase oyente.
Luego, instanciaremos nuestra clase, pasándole al constructor los datos necesarios. A continuación, crearemos nuestros botones, usando el constructor JButton(Action a), al cual le pasaremos una instancia de nuestra clase.

Métodos Action

(Método heredado) actionPerformed(ActionEvent e)

Se llama a este objeto en cuanto se pulsa un botón.
Dentro de él introduciremos qué pasará al pulsar ese botón.
Si queremos obtener información de los botones, usaremos el método getValue();

putValue(String clave, String valor)

Se usa sobre el método constructor de la clase a la que pertenece, permite almacenar información mediante Clave: valor, donde la clave son variables finales estáticas pertenecientes a la clase Action (usaremos una u otra dependiendo de qué queramos almacenar) (o nos las inventamos como si fueran un String (ej: “colorboton”)), y a continuación una coma y lo que almacenaremos.

Si usamos la variable final estática Action.SMALL_ICON y le pasamos una imagen, al crear nuestro botón, saldrá la imagen antes que el texto del botón.
(constantes principales (consultar API) **Action.NAME, Action.SHORT_DESCRIPTION**)

getValue(String clave)

Este método lo podemos usar como el putValue(), pero este lo usaremos principalmente dentro del método actionPerformed.
Este método lo podemos usar para rescatar en una variable el valor de la clave que le indiquemos.

Crear una combinación de teclas como fuente

Para ello, usaremos las clases: **KeyStroke, InputMap, JPanel y ActionMap** y sus respectivos métodos:
static getKeyStroke(String s), put(KeyStroke, Object), getInputMap(int condición) y put(Object, acción)

Para hacer la combinación de teclas, seguiremos estos pasos:

1. Crear mapa de entrada: indicar qué objetos tendrán el foco.
   C: InputMap y JComponent
   M: getInputMap(int condición)
2. Crear la combinación de teclas: indicarla.
   C: KeyStroke
   M: static getKeyStroke(String s)
3. Asignar las combinaciones de teclas a un objeto.
   C: InputMap
   M: put(KeyStroke, Object)
4. Asignar nuestro objeto a la acción.
   C: ActionMap
   M: put(Object, acción)

1. Dentro del constructor de la lámina, instanciamos un objeto perteneciente a la clase **InputMap** (con constructor default), que será nuestro mapa de entrada, y lo igualamos a el método **getInputMap** que recibirá: **JComponent.WHEN_IN_FOCUSED_WINDOW** . Hacemos esto para almacenar el output del método getInputMap.
InputMap mapa_input= getInputMap(JComponent.WHEN...);
Con esto le indicaremos que el oyente va a estar dentro de la lámina que tiene el foco (cambiar cte. JComponent.WHEN… para cambiar esto).

2. Todavía, dentro del constructor de la lámina, creamos un objeto perteneciente a la clase **KeyStroke** (con constructor default) y lo igualamos al método **Keystroke.getKeyStroke(String s)** en este String s, le indicaremos la combinación de teclas (consultar API para ver como se expresan).
KeyStroke micombinacion = KeyStroke.getKeyStroke(“ctrl A”); “(shift A); “(alt A); “(A); …

3. Asignar nuestra combinación de teclas al mapa de entrada, para ello usaremos un objeto perteneciente a la clase **InputMap**, usaremos el creado anteriormente y usaremos el método **put(KeyStroke, Object)** , indicando en KeyStroke el objeto perteneciente a la clase KeyStroke que hemos creado anteriormente y en object, crearemos un nuevo objeto:
mapa_input.put(micombinacion, “cambiar_color_fondo”);

4. Primero, creamos un objeto perteneciente a la clase **ActionMap** y lo igualamos a su método **getActionMap()** para ahora asignar nuestro objeto a nuestra acción, usamos el método **put(Object, acción)** sobre el objeto perteneciente a la clase ActionMap que acabamos de crear: (nuestra acción creada anteriormente se llama accioncambiacolor)
ActionMap mapaAccion = getActionMap();
mapaAccion.put(“cambiar_color_fondo”, accioncambiacolor);

Título IX: Múltiples Oyentes

Para esto, haremos que en verdad, haya un oyente pero este oyente va a ser una clase de la que tendremos diversos objetos.
Esto hace que todos los objetos pertenecientes a esta clase interactúen con el evento.

Capítulo V: Layouts

Disposiciones/Layouts por defecto de Java: **FlowLayout**(default), **BorderLayout** y **GridLayout**. (por cada una hay una clase homónima).
Para indicar el layout que se va a utilizar, dentro del constructor de nuestra lamina, creamos un objeto perteneciente a la clase del layout que queramos usar y posteriormente, le indicamos con el método **setLayout()** qué Layout vamos a usar (esto también es totalmente válido en el constructor del marco para establecer un layout al marco):

```
FlowLayout disposicion = new FlowLayout();
setLayout(disposicion);
```

*Todos los Layouts se pueden aplicar a láminas que estén dentro de láminas con distinto layout que estén dentro de otras láminas con distinto layout…
*Esta introducción funciona con los Layouts básicos (Títulos I,II, IV).

Título I: FlowLayout

Layout por default que centras los componentes arriba y en el centro.
Este layout no afecta al tamaño de los componentes.
La sobrecarga de constructores nos permite personalizar el Layout:

new FlowLayout() (default)

Establece los elementos en el centro arriba de la lámina con una separación entre objetos y contenedor de 5px (consultar API).

new FlowLayout(int alineacion)

Establece los elementos dependiendo de la alineación que le indiquemos mediante constantes estáticas de la clase FlowLayout. (FlowLayout.CENTER, FlowLayout.RIGHT, FlowLayout.LEFT).
Con una separación entre objetos y contenedor de 5px.

new FlowLayout(int alineacion, int hespacio, int vespacio)

Igual que el anterior, pero le podemos indicar la distancia horizontal (hespacio) y la vertical (vespacio) con respecto a los demás componentes y la distancia mínima a los bordes.

Título II: BorderLayout

Divide el espacio en 5 zonas: north, south, west, east, center.
Los componentes que están en cada zona, se ensanchan lo máximo posible como para abarcarla completamente.
Al usar este constructor, después de indicar con add() el componente que añadimos, (como un botón), debemos poner una coma e indicar el espacio (north, south...) que este ocupará mediante constantes estáticas pertenecientes a esta misma clase:
add(new JButton("Botón"), BorderLayout.CENTER);
Existen 2 constructores:

new BorderLayout() (default)

Los componentes se ensanchan lo máximo posible como para abarcarla completamente sin dejar espacio entre ellos.

new BorderLayout(int hespacio, int vespacio)

Deja una separación horizontal (hespacio) y vertical (vespacio) en píxeles entre los componentes (los componentes no se separan de los márgenes).

Título III: Usar distintas Layouts

Para usar distintas Layouts en el mismo contenedor o marco, debemos crear tantas láminas como Layouts estemos dispuestos a usar.
Creamos las dos láminas con sus respectivos layouts, y las añadimos a nuestro marco con el método add() , pero después de añadirlas, colocamos una coma e indicamos con constantes de clases de BorderLayout qué espacio ocupará cada lámina:

add(lamina1, BorderLayout.NORTH);
add(lamina2, BorderLayout.SOUTH);

Título IV: GridLayout

Este Layout, divide la lámina en filas y columnas creando celdas cuadradas (como si fuera una hoja de excel). Y hace que el componente que se encuentra en cada celda, la rellene completamente.
Este Layout, también cuenta con varios constructores:

new GridLayout() (default)
Establece todos los elementos en la misma fila pero en diferentes columnas.

new GridLayout(int filas, int columnas)
Establece el número de filas y columnas que tendrá el layout.

new GridLayout(int filas, int colums, int hespacio, int vespacio)
Establece el número de filas y columnas que tendrá el layout y la separación entre estos.

Disposiciones/Layouts avanzados a partir de aquí.

Título V: BoxLayout (Clase Box)

Crear cajas cuyos componentes interiores estarán ordenados vertical u horizontalmente.
Este layout no hace que se redimensionen los componentes.
Para utilizar este layout, primero tenemos que crear un BoxLayout con métodos estáticos: **createHorizontalBox** o **createVerticalBox** (pertenecientes a Box) para crear un BoxLayout horizontal o vertical. posteriormente, debemos introducir los componentes que estarán en nuestro BoxLayout con la instrucción add() sobre nuestro objeto Box. (Se puede meter un BoxLayout dentro de otro con add())

Al final, añadir nuestro BoxLayout a la lámina con add().

new BoxLayout(Container c, int axis)

Crea un box layout.
c = componente padre (this la mayoría de veces).
axis = BoxLayout.Y_AXIS o BoxLayout.X_AXIS para disponer los objetos de dentro de la caja en vertical u horizontal.

(estáticos) createHorizontalStrut(int x), createVerticalStrut(int x)

(pertenecen a Box) Crean un componente invisible de los píxeles especificados (x) para separar los componentes de un BoxLayout (se deben usar sobre nuestro objeto de tipo Box con la instrucción add()).

(tenemos dos JButton que se llaman boton1 y boton2)

```
Box BoxLayout = Box.createHorizontalBox();
BoxLayout.add(boton1);
BoxLayout.add(Box.createHorizontalStrut(20));
BoxLayout.add(boton2);
```

(estático) createGlue()

(perteneciente a Box) adecúa el espacio entre dos objetos en correspondencia con el tamaño del marco (cuando lo redimensionamos), dejando los dos componentes en los extremos.

```
BoxLayout.add(Box.createGlue());
```

Título VI: SpringLayout (en muelle)

Permite alargar y encoger la distancia entre los componentes al redimensionar el marco como si de un muelle se tratase.
Para crear un SpringLayout, primero hay que instanciarlo.
A continuación, la instrucción **setLayout(Obj o)** con nuestro objeto.
(Se implementa igual que los Layouts básicos).

Para que este layout funcione correctamente, tendremos que establecer “un muelle” entre los componentes:

(estático) constant(int a, int b, int c)

(Perteneciente a Spring). (Más usado).
Crea muelles con un valor mínimo (a), preferido (b), y máximo (c). Este muelle tiene capacidad elástica de rebote y estiramiento. (ir probando valores aleatorios).
Spring muelle1 = Spring.constant(0, 10, 100);

(estático) constant(int a)

(Perteneciente a Spring)
Crea un muelle con un tamaño, creando una especie de muelle fijo (no se ensancha ni encoge), ideal para unir componentes que no queramos que se separen o junten al redimensionar. (a = nº píxeles).
Spring muelle2 = Spring.constant(10);

A continuación, estableceremos dónde van los muelles:

putConstraint(String a, Component b, Spring c, String d, Component e)

(sobre nuestro objeto SpringLayout)
a: lugar final del muelle (en el componente b) (al west, al east...).
b: componente más a la dcha de los 2 que unimos con el muelle.
c: nuestro muelle.
d: donde empieza el muelle (en el componente e) (al west…).
e: componente más a la izqda de los 2 que unimos con el muelle.

ej: (tenemos 2 JButton: boton1, boton2)

```
SpringLayout miLayout = new SpringLayout();
setLayout(miLayout);
add(boton1); add(boton2); add(boton3);
Spring muelle = Spring.constant(0, 10, 100);
```

miLayout.putConstraint(SpringLayout.WEST, boton1, muelle, SpringLayout.WEST, this); *//this para hacer referencia a la lámina.*
miLayout.putConstraint(SpringLayout.WEST, boton2, muelle, SpringLayout.EAST, boton1);
miLayout.putConstraint(SpringLayout.WEST, this, muelle, SpringLayout.EAST, boton2);

Título VII: Disposición libre

Para que nuestra lámina tenga una disposición libre, que viene siendo lo mismo a no tener layout, le indicaremos a la lámina:
setLayout(null) .
Utilizar una disposición libre nos obliga a usar el método setBounds(x, y, a, b) con todos los componentes que incorporemos.
*Si no indicamos el setBounds, los componentes no aparecerán.

Crear disposiciones libres propias

Para esto, deberemos crear una clase que herede de la interfaz **LayoutManager** , la cual implementa 5 métodos.
Es conveniente crear la clase en un documento aparte por si en un futuro proyecto queremos usar el mismo Layout.
Dentro del campo de clase, creamos dos variables (int x, y) y las iniciamos con las coordenadas que queremos que tenga el primer componente.
Una vez terminado todo, debemos indicarle a la lámina o marco, que usaremos nuestro layout con la instrucción setLayout() y le pasaremos una instancia de nuestra clase.

layoutContainer(Container padre)

Dentro de este método, crearemos dos variables int, una que nos lleve una cuenta y otra que recoja los componentes que añadimos (esta segunda será igual a padre.getComponentCount()).

creamos un bucle for que recorra los valores de nuestra segunda variable, creando un objeto de tipo Container que será igual a padre.getComponent(i) y después, estableceremos las coordenadas de los componentes con el método setBounds() , modificando posteriormente los valores de x e y para el siguiente componente. ej:

```
public void layoutContainer(Container padre){
        int contador = 0;
        int n = padre.getComponentCount();
        for(int i = 0; i<n; i++){
                contador++;
                Component c = padre.getComponent(i);
                c.setBounds(x, y, 100, 20);
                x+=100
                if(contador%2==0){
                        x=20;
                        y+=40;
                }
        }
}
```

preferredLayoutSize(Container padre)

addLayoutComponent(String nombre, Component c)

minimumLayoutSize(Container padre)

removeLayoutComponent(Component c)

Clase Container:

getComponentCount()

Devuelve (int) el número de componentes que hemos añadido.

getComponent(int x)

Devuelve el componente indicado (empieza a contar desde 0, como un arreglo).

Capítulo VI: Formularios, componentes Swing

Los componentes swing los agregaremos sobre la lámina.
Prácticamente todos los componentes se agregan igual:
Creamos un objeto perteneciente a la clase correspondiente y lo agregamos con la instrucción add() :

JButton boton = new JButton();
add(boton);

setBorder(Border b)

Establece un borde al componente (consultar título XIX).

setText(String s), getText()

Prácticamente, todos los componentes disponen de estos dos métodos, los cuales nos permiten cambiar el texto de un componente (setText()) o recoger el texto de un componente (getText()).

setSize(int a, int b), setLocation(int x, int y), setBounds(int x, int y, int a, int b)

Prácticamente, todos los componentes disponen de estos tres métodos, establecen tamaño, posición o ambas (Aunque es mejor hacer uso de un Layout y no de estos métodos).

setFont(Font f)

Es aplicable a prácticamente aplicable a todos los componentes, cambia la fuente, el estado (negrita) o el tamaño del texto (consultar capítulo III, título III, Cambiar fuentes (tipos de letra)).

getStyle(), getFont()

getStyle devuelve el estilo (int (constantes de clase)) (negrita, cursiva...), getFont devuelve la fuente (Font) (Arial, Times...).

getPreferredSize()

Devuelve (Dimension) el tamaño que tiene un componente.

setEnabled(boolean b)

Hace que se pueda interaccionar (true) o no (false) con el componente.

setMaximumSize(Dimension d), setMinimumSize(Dimension d)

Establecen el tamaño máximo de un componente.

setToolTipText(String s)

Establece el texto que saldrá en una pestaña inferior al componente cuando el usuario sitúe el ratón encima de este.

Título I: Botones: JButton

La sobrecarga de constructores nos permite crear botones con texto, imágenes o relacionados con acciones (consultar capítulo IV, título VIII) (consultar constructores en la API).
Los botones nos permiten que podamos sacar eventos de ellos (capítulo IV).

setEnabled(boolean b)

Activa (true) o desactiva (false) el botón.

Título II: Etiquetas: JLabel

Agregar texto a nuestras láminas.
(Consultar sobrecarga de constructores en la API).
*Un constructor nos permite indicarle la posición horizontal de la etiqueta ((estáticos) .LEFT, .RIGHT, .CENTER).
Mediante las constantes estáticas de la clase JLabel:
JLabel etiqueta = new JLabel(JLabel.CENTER);

Aplicar tipo de texto y negrita

Usaremos la instrucción **setFont(Font f)** sobre nuestro objeto de tipo JLabel, indicando el tipo de fuente que queremos (consultar capítulo III, título III, Cambiar fuentes (tipos de letra)).

Título III: Líneas de texto: JTextField

Cuadro de texto de una línea.
(Consultar API para sobrecarga de constructores (columns = ancho)).
Si no indicamos texto inicial ni tamaño, aparecerá con un tamaño inútil de 1px.

setBackground(Color c)

Cambia el fondo del cuadro de texto al color indicado. (consultar capítulo III, título III, Colorear, clase Color para frame y objetos java2D)

JPasswordField

Es prácticamente igual que JTextField, pero en vez de mostrarnos el texto escrito, nos muestra asteriscos; ideal para contraseñas.
Si no le indicamos tamaño, aparecerá con un tamaño inútil de 1px.
Esta clase no dispone del método getText() , en su lugar se encuentra getPassword() , que viene a ser lo mismo pero devuelve un arreglo de tipo char.
JPasswordField también acepta setBackground().

Eventos de JTextField

Para crear nuestra fuente, tendremos que aplicar el método **getDocument()** sobre nuestro objeto de tipo JTextField, este método nos devolverá un objeto de tipo Document, al cual le tenemos que indicar la clase oyente con el método **addDocumentListener(clase) .**

(nuestro objeto JTextField se llama texto y nuestra clase oyente, oyente.)

Document doc = texto.getDocument();
doc.addDocumentListener(oyente);

La clase oyente debe implementar la interfaz **DocumentListener** , la cual tiene 3 métodos:

changedUpdate(DocumentEvent e)

Se ejecuta cuando el texto cambia (por ejemplo, de negrita).

insertUpdate(DocumentEvent e)

Se ejecuta cuando se introduce texto.

removeUpdate(DocumentEvent e)

Se ejecuta cuando se borra texto.

Título IV: Campos de texto: JTextArea

(Clase JTextPane es similar).
Cuadro de texto de infinitas líneas.
(Consultar API para sobrecarga de constructores).
Si no le indicamos tamaño, aparecerá con un tamaño inútil de 1px.
Si creamos un JTextArea default, este no tiene límites, se ensancha y alarga en función del texto que introduzca el usuario.

setLineWrap(boolean b)

Establece que el cuadro de texto no se pueda ensanchar (true), o permite que se ensanche (false) (por defecto, lo permite).

getLIneWrap()

Devuelve (true/false) el estado de setLineWrap().

Para solucionar el bloqueo vertical, la única forma de evitarlo es crear una barra que nos permita subir y bajar por el campo. Esto lo haremos creando una segundo lámina perteneciente a la clase **JScrollPane** que será la que incorpore la barra, y contendremos ahí nuestro campo de texto: (dentro del constructor de la lámina principal)

```
JTextArea campo = new JTextArea(10, 20);
JScrollPane laminabarra = new JScrollPane(campo);
add(laminabarra);
```

Si nuestro objeto tiene el setLineWrap en true, no pasará nada extra, pero si está en false, el campo no se ensanchará, sino que al estar en una lámina con barra, aparecerá una barra horizontal.
*Si usamos la instrucción getText() sobre un campo de texto, se conservan los saltos de línea.

append(String s)

Añade el texto indicado al JTextArea indicado al final de todo el texto.

Título V: Checkboxes: JCheckBox

Debido a la sobrecarga de constructores, podemos crear un checkbox que lleve texto detrás sin necesidad de crear un checkbox y luego un JLabel al lado (consultar API sobrecarga de constructores).
Al igual que con los botones, los checkboxes pueden ejecutar un evento cada vez que son presionados gracias a la instrucción **addActionListener(Object a)** y a la clase oyente que implementa la interfaz **ActionListener**.

isSelected()

Devuelve true si el checkbox está activado, false si no.

setSelected()

Activa el botón (true) o lo desactiva (false).

Título VI: Botones de radio: JRadioButton

Usaremos la clase JRadioButton para instanciar cada uno de los botones que queramos, y la clase **ButtonGroup** , para que sólo uno de ellos pueda estar seleccionado a la vez.
JRadioButton, al igual que la mayoría de componentes swing, presenta sobrecarga de constructores (consultar API).
para crear los botones, instanciamos los botones que queremos por separado, posteriormente, instanciamos un objeto de tipo ButtonGroup y luego, añadimos los botones al grupo con la instrucción **add()** sobre nuestro objeto de tipo ButtonGroup:

```
JRadioButton boton1 = new JRadioButton();
JRadioButton boton2 = new JRadioButton();
ButtonGroup grupo = new ButtonGroup();
grupo.add(boton1);
grupo.add(boton2);
add(boton1);     //Para añadir los botones sobre nuestra lámina.
add(boton2);
```

A los radio button, al igual que a los botones y los checkboxes, se les puede implementar eventos.

Título VII: Combo box: JComboBox

Presenta sobrecarga de constructores (consultar API) el más usado es el default.
Instanciaremos un objeto perteneciente a esta clase.
Posteriormente, sobre nuestro objeto, usaremos el método **addItem(String s)** para añadir las diferentes opciones de nuestro combo box.

setEditable(boolean b)

Nos permite “convertir” el combo box en un TextField, permitiendo al usuario escribir.
Si nuestro combo box tiene un evento, para hacer que este funcione con la cosa que hayamos escrito, tenemos que presionar ENTER después de escribirlo.

getSelectedItem()

Debe implementarse sobre un objeto de tipo JComboBox. Nos devuelve lo que ha seleccionado el usuario, pero nos lo devuelve como un object. es recomendable hacer un casting delante para recibir un String: (nuestro JComboBox se llama caja)
System.out.print((String)caja.getSelectedItem());

removeAllItems()

Borra todos los items del JComboBox.

A los combo boxes, al igual que a los botones y los checkboxes, se les puede implementar eventos.

Título VIII: Seleccionador deslizante: JSlider

Presenta sobrecarga de constructores (consultar API). Con esta sobrecarga, podemos cambiar la orientación del slider (default=horizontal) mediante constantes (consultar método setOrientation() más adelante:).
Primero, inicializarlo y el último paso, añadirlo a la lámina.
Por defecto, el slider puede adoptar valores intermedios entre las rayas menores indicadas, para evitar esto, aplicar **setSnapToTicks(boolean b)** .

setPaintTicks(boolean b)

Indica si se verán (true) o no (false) las rayas de debajo del JSlider.

setPaintLabels(boolean b)

Indica si se verán (true) o no (false) los números de debajo del JSlider (los números sólo aparecen en las marcas mayores).

setMajorTickSpacing(int x), setMinorTickSpacing(int x)

Indica la separación entre las rayas grandes (setMajorTickSpacing()) o las pequeñas (setMinorTickSpacing()).

setOrientation(int x)

Cambia la orientación del slider (mediante constantes (SwingConstants.HORIZONTAL o SwingConstants.VERTICAL)).

setSnapToTicks(boolean b)

Hace que el slider sólo pueda estar en las rayas marcadas, haciendo imposible valores intermedios.

getValue()

Nos devuelve el valor al que está señalando el Slider en ese momento.

El Slider es muy personalizable (consultar API).

Eventos del JSlider

La interfaz que usaremos es **ChangeListener ,** la cual implementa el método **stateChanged(ChangeEvent e)** , que se pone a la escucha con el método **addChangeListener(oyente)** .

Título IX: Líneas de texto con aumento: JSpinner

Presenta sobrecarga de constructores (consultar API).
El constructor por defecto (no recibe parámetros), nos permite hacer un Spinner numérico, pero si queremos hacer un Spinner con parámetro en forma de lista, como una especie de combo box,

deberemos usar el segundo constructor, el cual recibirá una clase específica para lo que queramos hacer:

- Para fechas y horas:

JSpinner Spinner = new JSpinner(new SpinnerDateModel());

- Para datos en lista:

String[] lista = {“a”, “b”, “c”};
JSpinner Spinner = new JSpinner(new SpinnerListModel(lista));

- Para personalizar el JSpinner numérico:

SpinnerNumberModel modelo = new SpinnerNumberModel();
(consultar API sobrecarga de constructores).
JSpinner Spinner = new JSpinner(modelo);

Por defecto, los JSpinner se adaptan al tamaño del primer elemento.

setPreferredSize(Dimension d)

Establece unas dimensiones al Spinner: (nuestro JSpinner se llama s:)
s.setPreferredSize(new Dimension(100, 20));
(consultar API constructores para Dimension).

getValue(), getNextValue(), getPreviousValue()

Los tres métodos devuelven un objeto, al cual le podemos aplicar un casting para String o para int, según convenga.

Eventos del JSpinner

La interfaz que usaremos es **ChangeListener ,** la cual implementa el método **stateChanged(ChangeEvent e)** , que se pone a la escucha con el método **addChangeListener(oyente)** .

Invertir JSpinner

Si queremos invertir el control de las flechas (la de abajo arriba y viceversa), lo tendremos que hacer nosotros mismos creando una clase interna:
creamos una clase interna que herede de **SpinnerNumberModel** .
Dentro del constructor, tendremos que pasarle los datos al constructor de la clase padre, que lo haremos con el método **super()** .
A continuación, para invertir el aumentar por disminuir y viceversa, sobreescribimos los métodos **getPreviousValue()** y **getNextValue()** con el método contrario. Ya sólo quedaría añadir un objeto de nuestra clase a el constructor de nuestro JSpinner:

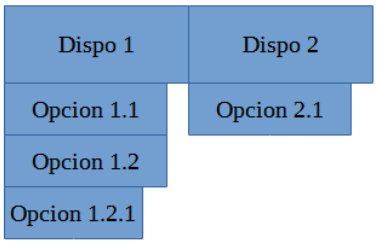
```
class lamina extends JPanel{
	public lamina(){
		ModeloSpinner modelo = new ModeloSpinner(1,1,1,1);
		JSpinner Spinner = new JSpinner(modelo);
		add(Spinner);
	}

	private class ModeloSpinner extends SpinnerNumberModel{
		public ModeloSpinner(int a, int b, int c, int d){
			super(a, b, c, d);
		}
		public Object getPreviousValue(){
			return super.getNextValue();
		}
		public Object getNextValue(){
			return super.getPreviousValue();
		}
	}
}
```

Título X: Menús: JMenuBar, JMenu, JMenuItem

JMenuBar gestiona la barra principal, JMenu las disposiciones de la barra principal, y JMenuItem las distintas opciones de cada disposición.
-Para crear la barra, con instanciar un objeto ya basta (la instrucción add() la añadimos después de añadir todos los JMenu) (sólo presenta constructor default).
-Para crear las disposiciones del menú, instanciamos tantos JMenu como sea necesario (consultar API sobrecarga de constructores). Añadimos las disposiciones a la barra con la instrucción add() sobre la barra (antes de ello, debemos haber añadido todos los JMenuItem).
-Para crear las opciones del menú, instanciamos tantos JMenuItem como sea necesario. Añadimos las opciones a las disposiciones con la instrucción add() sobre las disposiciones.
Si queremos que desde una opción salgan más opciones, la opción principal será un JMenu añadido a otro JMenu, y las demás opciones serán JMenuItem. (dentro del constructor de la lámina):
Para añadir funcionalidad, deberemos añadir un evento al menú, como si de un botón se tratase.

```
JMenuBar menu = new JMenuBar();
JMenu dispo1= new JMenu(“Dispo 1”);
JMenu dispo2= new JMenu(“Dispo 2”);
JMenuItem opcion1_1 = new JMenuItem(“Opcion 1.1”);
JMenu opcion1_2 = new JMenu(“Más”);
JMenuItem opcion1_2_1 = new JMenuItem(“Opcion 1.2.1”);
JMenuItem opcion2 = new JMenuItem(“Opción 2.1”);
opcion1_2.add(opcion1_2_1);
dispo1.add(opcion1_1);
dispo1.add(opcion1_2);
dispo2.add(opcion2);
menu.add(dispo1);
menu.add(dispo2);
add(menu);
```

Así conseguiremos lo siguiente:



addSeparator()

Añade una línea separatoria sobre el objeto (sólo sobre JMenu).

setJMenuBar(JMenuBar x)

Si queremos añadir directamente los menús sobre el marco de la ventana, usaremos esta instrucción en el constructor del marco, pasándole como argumentos, nuestra instancia de JMenuBar.

Menús a partir de Action

Para ello, creamos un JMenu que esté en un JMenuBar y a este JMenu, le añadimos mediante add() las instancias de las diferentes acciones, y se crearan automáticamente JMenuItem con las acciones y nombres correspondientes.

Añadir imágenes a JMenuItem

new JMenuItem(String s, Icon i)

Icon es una interfaz, por lo que necesitaremos una clase que implemente esa interfaz, como puede ser **ImageIcon** , la cual contiene un método constructor en el cual le podemos indicar la ruta de nuestra imagen.

JMenuItem opcion = new JMenuItem(“Opcion”, new ImageIcon(“Imagenes/imagen.png”));

(partimos de la carpeta principal de nuestro proyecto). Para poner primero el texto a las imágenes: (nuestro JMenuItem se llama opcion)

opcion.setHorizontalTextPosition(SwingConstants.LEFT);

Menús con CheckBox

Clase JCheckBoxMenuItem

Construye un JMenuItem acompañado de un JCheckBox.
Presenta sobrecarga de constructores (consultar API).
Se aplica igual que un JMenuItem.
Debemos añadir un evento para que funcione, como si de un botón se tratase.

isSelected()

Devuelve (boolean) true si está activado, false si no lo está.

Menús con RadioButton

Clase JRadioButtonMenuItem

Al igual que con los RadioButton, si quieres que sólo uno a la vez esté activado, debes agruparlos en un grupo instanciando un objeto perteneciente a ButtonGroup y añadir los JRadioButtonMenuItem con la instrucción add();
Funciona igual que los JMenuItem.
Al añadir los JMenuItem al JMenu correspondiente, no añadimos el grupo, sino los botones.
Debemos añadir un evento para que funcione, como si de un botón se tratase.

isSelected()

Devuelve (boolean) true si está activado, false si no lo está.

Atajos de teclado a menús

setAccelerator(KeyStroke k) (JMenuItem)

Combina una combinación de teclas con el oyente de un JMenuItem.
Usaremos el método estático **getKeyStroke(int keycode, int modif)** perteneciente a KeyStroke que devuelve un objeto KeyStroke.
(Queremos establecer Ctrl + A:)

keycode = código de la letra (A = KeyEvent.VK_A).
modif = código del modificador (Ctrl = InputEvent.CTRL_DOWN_MASK).
(ctes. estáticas de clase de KeyEvent (letra) y InputEvent (modif)).
(Al aplicar un atajo de teclado, se especifica automáticamente en el JMenuItem el atajo de teclado).

Primero, sobre el JMenuItem (o JCheckBoxMenuItem...) , le aplicamos el método **setAccelerator()**, al cual le pasamos el método estático **getKeyStroke()**, y le pasamos dos parámetros, los cuales son ctes estáticas de clase de KeyEvent y InputEvent:
(nuestro JMenuItem se llama opcion1 y queremos establecer Ctrl + A)
opcion1.setAccelerator(KeyStroke.getKeyStroke (KeyEvent.VK_A, InputEvent.CTRL_DOWN_MASK));

Título XI: Menús click dcho: JPopupMenu

Presenta sobrecarga de constructores; el más usado es el default.
Primero, instanciaremos la clase. instanciamos todos los JMenuItem que queramos y los añadimos a nuestro objeto JPopupMenu con add().
Los JPopupMenu no se agregan a la lámina con add() se agregan con el método setComponentPopupMenu().
Debemos añadir un evento para que funcione, como si de un botón se tratase.

setComponentPopupMenu(JPopupMenu c)

Se usa para añadir el menú emergente, si lo hacemos sobre la lámina, el PopupMenu saldrá en toda la lámina, si lo hacemos sobre un botón, sólo sobre el botón: (el botón se llama boton).
boton.setComponentPopupMenu(menu_emer);

JPopupMenu menu_emer = new JPopupMenu();

JMenuItem opcion1 = new JMenuItem(“Opcion 1”);
menu_emer.add(opcion1);
setComponentPopupMenu(menu_emer); *// al no indicar nada, lo agregamos sobre toda la lámina ya que estamos en el constructor de la lámina.*

(Si agregamos el PopupMenu sobre la lámina y en esa lámina hay un JTextArea (o lo que sea) deberemos indicar con la instrucción anterior que también debe funcionar sobre el JTextArea). (se llama area).
setComponentPopupMenu(menu_emer);
area.setComponentPopupMenu(menu_emer);

Título XII: Barras de herramientas: JToolBar

Primero, necesitamos crear nuestra clase oyente, la cual heredará de **AbstractAction** (consultar capítulo IV, título VIII).
JToolBar presenta sobrecarga de constructores (consultar API) uno de ellos, nos permite darle orientación a la barra. El default es de los más usados.
Primero, instanciamos nuestra barra, posteriormente, con add() sobre nuestro objeto JToolBar, añadiremos acciones a la barra y se crearán automáticamente botones asociados a estas acciones.
(también podemos pasar por add() una instancia de JButton (así, no tendríamos que crear una clase que herede de AbstractAction, sino implementar un evento al botón mediante addActionListener() sobre el botón)).
Posteriormente, añadimos nuestra barra con la instrucción add().
*Si no indicamos la posición de la barra, esta aparecerá arriba y se mantendrá fija, pero si le indicamos las posiciones: BorderLayout.NORTH, BorderLayout.EAST, BorderLayout.WEST o BorderLayout.SOUTH, la barra se creará en la posición indicada, dejando la posibilidad de que el usuario la arrastre a cualquiera de las

otras 3 posiciones (Si la arrastramos al centro del marco, se generará una ventana con esta, aunque seguirá funcionando).

add(Action a)

Añade una acción a los botones de la barra de herramientas

addSeparator()

Añade una separación entre los botones (se implanta sobre el objeto JToolBar, haciendo que el siguiente add(Action a) aparezca separado).

setOrientation(int x)

Cambia la orientación de la barra (0=horizontal, 1=vertical).

Título XIII: Ventanas emergentes

-Selección de ficheros: Clase JFileChooser.
-Selección de color: Clase JColorChooser.

Cuadros de diálogo: JOptionPane

(static) showMessageDialog(Component padre, Object msg)

(Presenta sobrecarga de métodos para cambiar la ventana (consultar API)).
-padr: clase padre (si nos encontramos en una clase interna de una lámina, le indicaremos el nombre de la lámina .this) (null si no quieres añadir).
-**msgj:** Mensaje a mostrar (String).

(static) showInputDialog(Component padre, Object msg)

(Presenta sobrecarga de métodos para cambiar la ventana (consultar API)).
Igual que el anterior.

(static) showConfirmDialog(Component padre, Object msg, String titulo, int a)

(Presenta sobrecarga de métodos para cambiar la ventana (consultar API)).
Igual que el anterior.
-titulo: el título de la ventana emergente.
-a: Cambia los botones que muestra (0, 1, 2).

(static) showOptionDialog(Component padre, Object msg, String titulo, int a, int b, Icon icon, Object[] options, Object valor_inicial)

(No presenta sobrecarga de métodos).
Igual que los anteriores.
-b: Cambia el icono (interrogación, exclamación...).
-icon: Icono que se muestra en vez de mensaje (null si no quieres añadir).
-options: Arreglo de objetos (JButton, por ejemplo) que aparecerán en el cuadro (null si quieres que se usen los especificados en **a**).
-valor_inicial: Valor por defecto (null si no quieres añadir).

Cuadros de selección de archivos: JFileChooser

Esta clase nos permite seleccionar ficheros dentro de nuestro dispositivo.
1. Primero, creamos un objeto perteneciente a esta clase con el constructor por defecto, el cual empieza buscando en la carpeta Documentos (sustituir constructor para hacer que empiece por otra ruta).
2. Podemos construir un filtro con la ayuda de la clase **FileNameExtensionFilter**, para que desde el JFileChooser sólo se vean los archivos con las extensiones especificadas. Para ello, usamos el constructor: **FileNameExtensionFilter(String description, String… extensions)**

3. Agregamos nuestro filtro a nuestro objeto **JFileChooser** con el método **setFileFiltrer(FileNameExtensionFilter filter)**
4. Creamos una variable **int** que igualamos a nuestro objeto **JFileChooser** seguido del método **showOpenDialog(Component parent)**, al que le tenemos que especificar un componente padre.
La ventana pausará el hilo de ejecución.
5. Evaluamos si el usuario pulsó en aceptar o cancelar, comparando la variable **int** con la cte de clase: **JFileChooser.APPROVE_OPTION**.
6. Almacenamos en una variable **File** nuestro archivo seleccionado con la ayuda del método **getSelectedFile()** sobre nuestro objeto **JFileChooser**.

```
JFileChooser ventana = new JFileChooser();
FileNameExtensionFilter filtro = new
FileNameExtensionFilter("Archivo de texto", "txt");
//Añadir tantas extensiones como se desee:
//"txt", "gif", "jpg"... , acepta múltiples cadenas
ventana.setFileFiltrer(filtro);
int var = ventana.showOpenDialog(null);
if (var==JFileChooser.APPROVE_OPTION){
        File archivo = ventana.getSelectedFile();
}
```

Título XIV: Crear bordes: BorderFactory

Para todo tipo de componentes.

(static) createEtchedBorder()

Crea un borde con el color de la lámina padre.

(static) createTitledBorder(Border b, String titulo)

Crea un borde.

(En el constructor de la lámina):
setBorder(BorderFactory.createTitledBorder(BorderFactory.createEtchedBorder(), "título"));

LineBorder

Creamos un objeto de tipo **LineBorder** con el constructor:
public LineBorder(Color c, int thickness, boolean roundedCorners);
Posteriormente, agregamos nuestro borde al contenedor que queramos con **setBorder(Border e);**

- **Componentes Swing avanzados:**

Título XV: Listas: JList

Lista de elementos a la que podemos agregar una barra lateral de desplazamiento.

A la hora de crear un **JList**, deberemos indicarle qué tipo de objetos almacenará.
JList <E> nombre = new JList <E>(); (sustituir E por tipo de clase)
Lo normal es pasarle a este **JList** un array del dato que almacenará.
String[] asd = {"a", "s", "d"};
nombre = new JList<String>(asd);

*Se pueden seleccionar varios objetos del **JList** a la vez con la tecla control.

Barra de desplazamiento

Para añadir una barra de desplazamiento al **JList** deberemos crear un **JScrollPane** al que le pasaremos como parámetro en el método constructor nuestro objeto **JList**. Agregar el objeto **JScrollPane**.

```
String[] asd = {“a”, “s”, “d”};
JList <String> lista = new JList <String> (asd);
JScrollPane barra = new JScrollPane(lista);
add(barra);
```

setVisibleRowCount(int datos)

Donde especificas el número de elementos que se verán en el JList sin usar la barra de desplazamiento.

List<E> getSelectedValuesList()

Devuelve un objeto **List<E>** (E = E del objeto **JList**) que contiene a los elementos seleccionados en el **JList**.

Eventos de JList

Para ello, debemos agregar el método **addListSelectionListener(ListSelectionListener e)** , crear una clase oyente que implemente la interfaz **ListSelectionListener** y sobreescribir el método **valueChanged(ListSelectionEvent e)**

Almacenar en String valores seleccionados

```
List <String> valores = lista.getSelectedValuesList();
StringBuilder texto = new StringBuilder(“”);
for(String elemento: valores){
        String palabra = elemento;
        texto.append(palabra);
        texto.append(“, ”);
}
String resultado = texto.toString();
```

Título XVI: Árboles (interfaz carpetas): JTrees

Un árbol está compuesto por nodos. Cada nodo es un nivel en el árbol, siendo el nodo padre el nodo raíz, seguido de los nodos hijos y finalmente los nodos hojas que no son inextensibles.
Un árbol se puede expandir infinitamente.

Usaremos las clases **JTree** y **DefaultMutableTreeNode** donde en la segunda crearemos cada nodo indicando qué tipo de nodo es y los agrupamos en una clase **JTree**.

Para ello, creamos cuantos **DefaultMutableTreeNode** como necesitemos y vamos agregando unos a otro con el método **add()**, como si de unos **JMenu** se tratase: **(DefaultMutableTreeNode = DMTN** sólo para este ejemplo**)**

```
DMTN raiz = new DMTN("Mundo");
DMTN hijo1 = new DMTN("España");
DMTN hijo2 = new DMTN("Madrid");
DMTN hoja = new DMTN("Móstoles");
raiz.add(hijo1);
hijo1.add(hijo2);
hijo2.add(hoja);
```

Una vez tenemos todos los nodos ordenados, creamos un objeto perteneciente a **JTree** al que le pasamos el nodo raíz en la llamada al método constructor. Y añadimos el árbol a la lámina (conveniente que esta tenga un BorderLayout() o un GridLayout(1, 1)).

```
JTree arbol = new JTree(raiz);
add(arbol);
```

Añadir barra de desplazamiento

Para añadir una barra de desplazamiento al **JTree**, deberemos crear un **JScrollPane** al que le pasaremos como parámetro en el método constructor nuestro objeto **JTree**. Agregar el objeto **JScrollPane**.

JTree arbol = new JTree(raiz);
JScrollPane barra = new JScrollPane(arbol);
add(barra);

Eventos

La clase **JTree** puede enviar varios eventos (consultar métodos API).

Título XVII: Tablas: JTable

JTable permite crear tablas. Presenta sobrecarga de constructores (consultar API).
Constructor principal: **JTable(Object[][] data, Object[] nomcolum)** donde indicamos los datos de la tabla (data (estos datos pueden estar compuestos por boolean en el hueco 0, String en el hueco 1...)) y el nombre de cada columna (nomcolum).

Añadir barra de desplazamiento

Para añadir una barra de desplazamiento al **JTable**, deberemos crear un **JScrollPane** al que le pasaremos como parámetro en el método constructor nuestro objeto **JTable**. Agregar el objeto **JScrollPane**.
JTable tabla = new JTable(data, nomcolum);
JScrollPane barra = new JScrollPane(tabla);
add(barra);

Imprimir (en papel)

Usar el método **print()** sobre nuestro objeto **JTable** y se nos abrirá el asistente de impresión.

setValueAt(Object value, int row, int column)

Especifica el valor **value** en la casilla especificada.

setCellSelectionEnabled(boolean n)

Hace que al pinchar en una casilla se seleccione sólo esa casilla (**true**) o la fila entera (**false**).

setSurrendersFocusOnKeystroke(boolean n)

Hace que si las celdas del JTable tengan la propiedad de ser editadas (**isCellEditable()**), que podamos acceder a estas con un sólo click en lugar de dos.

Eventos

Los JTable pueden detectar cuando ha habido un cambio en su estructura. Para ello:

Método: getModel().addTableModelListener(TableModelListener e)

Interfaz: TableModelListener

Método evento: public void tableChanged(TableModelEvent e)

*Podemos identificar la casilla actualizada con los métodos **getColumn** y **getRow** sobre nuestro objeto **TableModelEvent** (e).

Tablas personalizadas

Para ello, usaremos la clase abstracta **AbstractTableModel**, la clase **TableModel** y los 3 métodos que esta implementa: **getColumnCount()**, **getRowCount()**, **getValueAt(int rowIndex, int columnIndex)**.

- **getColumnCount():** indicamos el nº de columnas.
- **getRowCount():** indicamos el nº de filas.
- **getValueAt();** indicamos qué elemento se verá en cada casilla.
- **isCellEditable();** indicamos si cierta celda es editable o no.

Sobreescribimos todos estos métodos y creamos nuestra tabla: (ModeloTabla es el nombre de la clase que hemos creado que hereda de **AbstractTableModel**):

TableModel modelo = new ModeloTabla();

JTable tabla = new JTable(modelo);

add(tabla);

*Dentro de la clase que hereda de **AbstractTableModel** también podemos sobreescribir el método **getColumnName(int c)** para nombrar nuestras columnas.

Capítulo VII: Exportar programas

Título I: Applets

Programas Java que se ejecutan en otros programas (un navegador).
Los applets no llevan método main, en su lugar tienen el método init.
Todas las clases que forman un applet deben heredar de **JApplet**.
Programa applet más sencillo:

```
import javax.swing.*;
public class Main extends JApplet{
	public void init(){
		JLabel etiqueta = new JLabel(“Hola mundo”);
		add(etiqueta);
	}
}
```

Aunque eclipse lleva una herramienta para visualizar el applet (bastante sencilla), como la finalidad del applet es usarlo en un navegador, lo podemos meter dentro de una página en html.
Para ello, creamos nuestro archivo de página web en la carpeta padre (donde se encuentra el documento .applet correspondiente a nuestro programa).

```
<html>
<body>
<applet code=”ruta/Main.class” width=”10” height=”10”>
</applet>//los números pueden variar al gusto, es para ver cuánto
</body>    ocupará el applet en la página web
</html>
```

***Consultar vídeo 132, minuto 16:30 (seguridad).**

Un applet no puede tener marco, no se le pueden dar coordenadas ni ancho o largo (lo hacemos desde html) y el método setVisible no hace falta ponerlo, ya que siempre un applet será visible. Tampoco puede tener título un applet.
Sí se pueden insertar láminas dentro de un applet.

Se pueden hacer cosas interesantes, como un botón en nuestra página web que al pulsarlo se abra el programa que queramos en una ventana diferente, esto lo hacemos creando un botón que se verá en el applet con un oyente que ejecuta el programa.

Se pueden pasar datos entre html y java.
Esto lo hacemos indicando en html:
<param name=”variable1” value=”5”>

y en java dentro del método init()
String num = getParameter(“variable1”);

Título II: Archivos JAR

Consultar vídeos 137 - 141.
(137 - 140 son teóricos, hablando sobre cómo crear un jar normal y cómo firmarlo digitalmente)
(140 ya explica cómo crear un JAR ejecutable)
(141 explica cómo crear un enlace para ejecutar un archivo jar remótamente)

Capítulo VIII: Excepciones

Título I: Conceptos básicos

En Java, todo son objetos, por lo que las excepciones, son objetos que heredan de la clase **Exception**. A partir de aquí, las excepciones pueden ser de dos tipos:

-IOException (excepciones comprobadas): No son fallo del programador (Imagen no se encuentra en carpeta, por ejemplo).
Si es una excepción que hereda de esta clase, estaremos obligados a crear un bloque try catch (explicado posteriormente).

-RuntimeException (“ no “): Errores por parte del programador (recorrer hasta 5 una variable de 4 valores, por ejemplo). Si obtenemos una excepción que hereda de esta clase, deberemos implementar algo en el código para prevenir que no ocurra no estamos obligados a implementar un try catch, aunque podamos hacerlo (Es preferible depurar el código antes de utilizar throws try catch).

IOException

Las IOException hacen que sea necesario prever una excepción ya que es probable que algo falle. Ej (método read() de ImageIO):
Declaración del método:

public static BufferedImage read (File x) throws IOException{...}

el throws IOException final hace que si por alguna razón, el método no llega a funcionar, se lance una excepción y esto nos obliga a que cada vez que usemos este método, tengamos que montar una estructura try catch, donde se intenta usar este método, pero si falla que haya un código suplente:

```
try{
        Image ejemplo = ImageIO.read(new File(“C:/Fotos/x.png”));
}catch(IOException e){
        código suplente;
}
```

(El “código suplente” sólo se ejecutará si el código de try falla y por lo tanto, lanza su excepción correspondiente. Si el try funciona correctamente, el código suplente no se ejecuta).

Si en nuestro programa aparece una excepción, podemos hacer que lo que lanza la excepción sea un método que lanza la excepción correspondiente y cuando la llamemos, usar la estructura try catch.

printStackTrace()

Si lo aplicamos al objeto de nuestro catch, imprime en consola la excepción que ha ocurrido:

```
try{
      código;
}catch(IOException e){
      e.printStackTrace();
}
```

getMessage()

Si lo aplicamos al objeto de nuestro catch, imprime en consola la excepción que ha ocurrido:

```
try{
      código;
}catch(IOException e){
      System.out.println(e.getMessage());
}
```

Título II: Excepciones manuales

Lanzadas por el programador por la cláusula throw.
Esto se puede hacer instanciando una clase que maneje excepciones (como ArrayIndexOutOfBoundsException) y con la cláusula throw:
throw new ArrayIndexOutOfBoundsException();

Si esto lo hacemos dentro de un método, debemos indicarle al método que puede lanzar una excepción con (en este ejemplo) **throws ArrayIndexOutOfBoundsException** .

Título III: Excepciones propias

Se pueden crear excepciones personalizadas creando una clase que herede de **Exception**, **IOException** o **RuntimeException** .
(Nuestras preferencias cambian dependiendo de para qué queramos nuestra excepción, si para una excepción comprobada o no comprobada).
Lo recomendado es crear dos constructores en esta clase, el por defecto y otro que reciba un String, el cual se pasará al constructor padre (Este String se imprimirá en consola junto con la excepción si la clase hereda de RuntimeException).

Título IV: Capturar múltiples excepciones

Se nos puede dar la ocasión de que un método lance diferentes tipos de excepciones, y que dependiendo de la excepción lanzada, queramos hacer una cosa u otra. Por esto, es por lo que podemos atrapar diferentes excepciones con un solo try:

```
try{
        código;
}catch(ArithmeticException e){
        código;
}catch(NumberFormatException e){
        código;
}...
```

Título V: Finally

Complementa al bloque try catch, lo que introduzcamos en finally se ejecutará siempre, independientemente de si se lanzó o no una excepción.

```
try{
        código;
}catch(IOException e){
        código;
}finally{
        código;
}
```

La cláusula finally se suele usar para cerrar conexiones entre cosas externas al programa (como bases de datos) con el programa, para liberar los recursos usados.

Capítulo Nuevo: Debugging

Consultar vídeos 150, 151.

Capítulo IX: Secuencias/Streams

Un Stream establece una conexión entre el programa y un archivo externo, útil para escribir o leer información en ficheros externos. Esto se hace utilizando clases que heredan de las clases abstractas **Reader** y **Writer** .

Título I: Reader: Leer Ficheros Sencillos

Lo más cómodo es crear una clase que se dedique a leer. Esta clase contará con un método que leerá la información. Dentro de aquí: Primero, indicaremos la ruta de nuestro fichero instanciando un objeto de la clase **FileReader** (el cual puede lanzar una excepción **FileNotFoundException** que hereda de IOException, por lo que deberemos crear una instrucción try catch).
Para leer el fichero, usaremos la instrucción read() sobre nuestro objeto perteneciente a la clase FileReader, el cual nos devuelve un int que se corresponde al código del carácter (si el texto termina, devuelve un -1). Si a este le realizamos un casting a char, volvemos a obtener el carácter. Cuando terminemos de leer, es importante finalizar el "puente" creado entre el fichero y el programa para no consumir recursos extra, esto lo haremos con la instrucción close() sobre nuestro objeto FileReader. Ejemplo de lectura:

```
class Lectura{
	public void leer(){
		try{
			FileReader archivo;
			archivo= new FileReader("C:/Carpeta/asd.txt");
			int c = 0;
			while(c!=-1){//Cuando el texto termina
				c = archivo.read();
				char letra = (char) c;
				if(c!=-1){//-1 = ? , así nos quitamos la ?
```

```
                    System.out.print(letra);
                }
            }
            archivo.close();
        }catch(IOException e){
            System.out.println("Archivo no encontrado.");
        }
    }
}
```

Título II: Writer: Escribir en Ficheros Simples

Para escribir en un fichero, haremos algo similar a leer, creamos una clase que contenga un método en el cual escribiremos lo siguiente: Primero, indicaremos la ruta de nuestro fichero instanciando un objeto de la clase **FileWriter** (el cual puede lanzar una excepción **FileNotFoundException** que hereda de IOException, por lo que deberemos crear una instrucción try catch).
A continuación, escribiremos en el fichero carácter por carácter nuestro texto y al final, cerraremos el puente creado entre el programa y el fichero: (dentro de nuestro método escribir():)

```
String texto = "Ejemplo de texto";
try{
    FileWriter archivo = new FileWriter("C:/Carpeta/asd.txt");
    for(int i=0; i<texto.length(); i++){
        archivo.write(texto.charAt(i));
    }
    archivo.close();
}catch(IOException e){
    System.out.println("Archivo no encontrado");
}
```

*Si hacemos esto, se creará un archivo con el nombre indicado y con el texto indicado. Si queremos añadir texto a un fichero ya existente, en el constructor del FileWriter, indicaremos un true:
FileWriter archivo = new FileWriter("C:/Carpeta/asd.txt", true);

Título III: Buffers/Filtros

Son memorias internas entre nuestro programa y el archivo con el que vamos a interactuar, las cuales hacen que esto sea más eficiente.
Se suelen usar en programas de streaming.
Clases: **BufferedReader**, **BufferedWriter**.

Leer Ficheros
readLine()

Devuelve un String que corresponde a cada línea del fichero. Si se queda sin líneas, devuelve null.

Reescribiendo el código del título I:

```
try{
	FileReader archivo;
	archivo= new FileReader("C:/Carpeta/asd.txt");
	BufferedReader buffer = new BufferedReader(archivo);
	String texto = "";
	while(c!=null){//Cuando el texto termina
		texto = buffer.readLine();

		if(texto!=null)   //Para evitar que imprima el null.
		System.out.println(texto);
	}
	archivo.close();
}catch(IOException e){
	System.out.println("Archivo no encontrado.");
}
```

Título IV: Streams Byte

Con estos streams, podemos leer y escribir bytes, permitiéndonos cosas diferentes, como copiar imágenes.
Clases: **FileInputStream**, **FileOutputStream**. (Se usan igual que Reader y Writer).

Leer Archivos

```
try{
      FileInputStream archivo = new FileInputStream("C:/a.jpg");
       boolean final_ar = false;

       while(!final_ar){
            int byte_in = archivo.read();
            if(byte_in==-1)
            final_ar = true;
            System.out.println(byte_in);
       }
       archivo.close();
}catch(IOException e){código}
```

Escribir Archivos

```
int[] datos = {...};
try{
FileOutputStream archivo = new FileOutputStream("C:/ap.jpg");
       for(int i=0; i<datos.length; i++){
            archivo.write(datos[i]);
       }
       archivo.close();
}catch(IOException e){código}
```

Título V: Serialización

La serialización nos permite convertir en bytes un objeto de nuestro programa para compartirlo y "desconvertirlo" de bytes para obtenerlo de nuevo tal y como estaba cuando se serializó.
A usar: **Serializable** (Interfaz que no implementa nada, solo indica que los objetos pertenecientes a esta clase son susceptibles de ser serializados), **ObjectOutputStream → writeObject()** (Para escribir), **ObjectInputStream→ readObject()** (Para leer).

Escribir objetos

El primer paso, es buscar la clase a la que pertenece nuestro objeto y hacer que esta clase implemente la interfaz **Serializable** .
A continuación, creamos dentro de un try/catch un objeto perteneciente a **ObjectOutputStream** utilizando el constructor que recibe un **FileOutputStream**.
A continuación, escribiremos nuestro objeto con el método correspondiente y cerraremos el stream.

```
Object objeto = new Object();
try{
        String ruta = "C:/Users/.../archivo.dat";
        ObjectOutputStream escribir = new
        ObjectOutputStream(new FileOutputStream(ruta));
        escribir.writeObject(objeto);
        escribir.close();
}catch(Exception e){
        código;
}
```

Leer objetos

Igual que para escribir el objeto, en este caso, iniciaremos un objeto perteneciente a **ObjectInputStream** .

También, debemos crear un array del mismo tipo que el objeto que hemos almacenado en el fichero .dat, para poder almacenarlo en nuestro programa.
Como el método **readObject()** devuelve un **Object**, tendremos que realizar un casting a la clase que queramos obtener.
Importante cerrar el stream.

```
try{
        String ruta = "C:/.../archivo.dat";
        ObjectInputStream leer = new ObjectInputStream(new
FileInputStream(ruta));
        JFrame[] array = (JFrame[])leer.readObject();
        leer.close();
}catch(Exception e){
        código;
}
```

Conflicto actualizando programas

Consultar vídeo 158
Para crear una serialVersionUID:
private static final long serialVersionUID = 1L;
(la L es por ser una variable de tipo long)

Título VI: File: Consultar directorios y ficheros

Presenta sobrecarga de constructores, el que usaremos es el que recibe un String que indica la ruta.
(Si no indicamos ruta (C:/.../.../..), toma por defecto la carpeta de nuestro programa)
Otro constructor bastante usado es el que recibe dos String, de los cuales uno es la ruta padre y otro el hijo, el padre será igual a todos los directorios de donde se encuentra: C:/.../… y el hijo al archivo: archivo.txt (Consultar vídeo 159, minuto 17:30).

getAbsolutePath()

Devuelve un String, el cual nos indica la ruta absoluta (C:/...) de nuestro objeto perteneciente a File.

exists()

Devuelve un boolean, true si el archivo existe, false si no.

list()

Crea un listado (String[]) de todos los archivos/ficheros que se encuentran en la ruta especificada.

isDirectory()

Devuelve un boolean, true si el archivo especificado es una carpeta, false si no lo es.

Título VII: File: Creación, escritura y eliminación

mkdir()

Crea un directorio en la ruta especificada. Devuelve un boolean.
Para crearlo, nuestro objeto File tiene que llevar el nombre del directorio que se crea:

File archivo = new File("C:/Users/Public/Nueva Carpeta");
archivo.mkdir();

Y así se creará la carpeta Nueva Carpeta dentro de la carpeta Public.

createNewFile()

Crea un archivo en la ruta especificada. Devuelve un boolean.
El archivo lo crea vacío, y sólo lo crea si no existe, si existe no hace nada. Lanza una IOException.
Una vez creado el archivo, podemos modificarlo con todos los métodos utilizados en este capítulo.

delete()

Borra el archivo/directorio especificada en el constructor del objeto File.

Desktop.getDesktop().open(File f)

Abre (con la aplicación predeterminada) el archivo f.

Capítulo X: Programación genérica

Es genérico todo aquello que trabaja con un objeto sin especificar la clase a la que pertenece.
Su objetivo principal es el poder reutilizar el código.
Una ventaja es no tener que crear clases para modificar cosas, esto lo podemos hacer desde un ArrayList modificando el parámetro de tipo (lo de dentro de <>).

Título I: ArrayList

Permite crear arrays cuyo tamaño se adapta a los datos que introduzcamos de forma dinámica.
Los ArrayList permiten almacenar objetos, pero no datos de tipo primitivo (int, long, char...).
Como los arrays normales, los elementos se empiezan a colocar desde la posición 0.

Crear un ArrayListt

ArrayList <JFrame> lista = new ArrayList <JFrame> ();
Dentro de <> le introducimos el tipo de dato que almacena.

add(Object a)

Permite almacenar objetos en nuestro ArrayList.
lista.add(new JFrame());

size()

Devuelve un int con la cantidad de datos del array.

ensureCapacity(int)

Establece el tamaño default del ArrayList (se amplía dinámicamente si añadimos un elemento más), útil para no consumir más recursos de los necesarios.

trimToSize()

Recorta el espacio sobrante de la memoria usada, optimización de recursos. Se utiliza cuando ya tenemos todos los espacios llenos.

set(int x, Object a)

Establece en la posición x del ArrayList el objeto a.

Object get(int x)

Devuelve (Object) el objeto que se encuentra en la posición especificada (x).
Si queremos que nos devuelva un método que se encuentra en esta posición, indicar el método fuera del paréntesis. ej:
(JFrame se llama Marco)

```
ArrayList <JFrame> lista = new ArrayList <JFrame> ();
lista.add(Marco);
System.out.println(lista.get(0).getBounds);
```

(este código imprimirá las coordenadas de Marco de tipo JFrame).

Obtener todos los elementos con for

```
for(int i=0; i<lista.size(); i++){
        JFrame e = lista.get(i);
}
```

Copiar ArrayList en array convencional

```
JFrame[] arraylista = new JFrame[lista.size()];
lista.toArray(arraylista);
```

Iteradores, interfaz Iterator

Creación de un iterador

```
Iterator <JFrame> iterador = lista.iterator();
```

boolean hasnext()

Devuelve true si existe un elemento siguiente al que estamos analizando ahora mismo; false si no.

Object next()

Devuelve el siguiente objeto

remove()

Borra el elemento con el que estamos trabajando actualmente.

Imprimir datos usando iteradores

```
Iterator <JFrame> iterador = lista.iterator();
while(iterador.hasnext()){
        System.out.prinlnt(iterador.next().getBounds());
}
```

Título II: Clases genéricas propias

Para definir una clase genérica, después del nombre, le tenemos que indicar que va a ser genérica con corchetes angulares y una letra (por convección entre programadores suele ser mayúscula y suele ser la T, U o la K).

```
class generica <T> {}
```

Después, tenemos que crear el espacio de tipo genérico (T) donde almacenaremos los objetos y hacer que este sea nulo.
Y posteriormente, un método que establezca el valor de T.
De paso, estableceremos un GETTER que nos devuelva el dato introducido.
(Dentro de la clase:)

```
private T primero;
public generica(){
        primero = null;
```

```
}
public void setPrimero (T valor){
        primero = valor;
}
public T getPrimero(){
        return primero;
}
```

Creación de métodos genéricos dentro de clases genéricas y no genéricas

Ejemplo de GETTER que devuelve un String a partir de un array genérico que recibe:

```
public <T> String getNoseque(T[] a){
        return código;
}
```

A la hora de llamar al método podemos hacerlo de dos maneras:

```
… .getNoseque(new String[])                 //El compilador detecta
… .<String>getNoseque(new String[])         //el tipo de objeto
```

*Si usamos un método genérico perteneciente a una interfaz (como el método compareTo de la interfaz Comparable), para que el código funcione deberemos hacer que el método implemente la interfaz:

```
public <T extends Comparable> getNoseque(){}
```

(Comparable, o cualquier otra interfaz)

Título III: Herencia en clase genérica y tipo comodín

A la hora de realizar un polimorfismo (explicado a continuación)
(X hereda de Y)
X = new X();
Y = new Y();
Y = new X();
X = new Y(); NO

Si hacemos esto con clases genéricas (una clase genérica <X> y una <Y>), la primera y la segunda opción sí que la podemos hacer pero la tercera no.

Tipo comodín

Si queremos hacer un método que reciba una clase genérica con un parámetro de tipo especificado y que este sea válido para todas las clases que heredan de una superclase, haremos los siguiente:
public void nombre(ClaseGenerica<? extends JFrame> p){}

De esta forma, podremos pasar al método nombre todas las clases genéricas pertenecientes a ClaseGenerica que tengan como parámetro de tipo JFrame o cualquier clase que herede de esta.

Capítulo XI: Programación concurrente, Threads (Hilos)

Todos los programas creados anteriormente eran programas monotarea, capaces de realizar solo una tarea a la vez.

static sleep(int x) (clase Thread)

Detiene el programa durante x milisegundos.
Lanza una InterruptedException que nos veremos obligados a capturar.
Esta excepción es porque hemos intentado interrumpir el hilo mientras se producía un sleep y el sleep bloquea al programa, por lo que bloquea la interrupción.
*Si queremos interrumpir un hilo con un sleep (dentro del catch:)
Thread.currentThread().interrupt();

Título I: Creación de hilos

Para ello, seguiremos los siguientes pasos:
1. Crear clase que implemente la interfaz **Runnable** (método run()).
2. Escribir el código de la tarea dentro del run().
3. Instanciar la clase creada y almacenar la instancia en variable de tipo Runnable
4. Crear instancia de la clase **Thread** pasando como parámetro al constructor de Thread el objeto Runnable anterior.
5. Poner en marcha el hilo de ejecución con el método **start()** de la clase Thread.
(consultar vídeo 168 min 21:50)

class nombre implements Runnable{public void run(){código}}
(en otro lado:)
Runnable nom = new nombre();
Thread hilo = new Thread(nom);

hilo.start();

O bien, seguiremos estos pasos:
1. Crear clase que herede de la clase **Thread**.
2. Sobreescribir el método **start()**.
3. Crear una instancia perteneciente a nuestra clase.
4. Ponerla en marcha con la instrucción **start()**.

class nombre extends Thread{public void start(){código}}
(en otro lado:)
nombre hilo = new nombre();
hilo.start();

detener hilo

Si el hilo lleva un bucle while(true), sustituir este por while(!Thread.currentThread().isInterrupted()){} y añadir instrucción interrupt().

getName()

Devuelve un String con el nombre del Thread que java asigna automáticamente. (Thread-0 , Thread-1 … (como si fuera un arreglo))

interrupt()

Sobre nuestro objeto de tipo Thread (hilo) creado y puesto en marcha anteriormente. Este método lo detiene siempre y cuando el método anterior no lleve el método sleep, lo cual conllevará una excepción.

boolean interrupted()

Devuelve true si el hilo actual se encuentra interrumpido, false si no.

static boolean isInterrupted()

(Perteneciente a Thread) Realiza la misma función que interrupted pero este método se centra en el hilo actual.
Thread.currentThread().isInterrupted();

Título II: Sincronización de hilos

Sincronizar hilos es hacer que uno no empiece hasta que el otro no acabe y esto lo haremos con el método **join()** , que lanza una excepción.
Esto lo hacemos de la siguiente manera (dados los hilos 1 y 2):

```
Thread hilo1 = new Thread();
Thread hilo2 = new Thread();
hilo1.start();
try{
        hilo1.join();
}catch(Exception e) {
        código;
}
hilo2.start();
```

Esto lo podemos dejar más claro si dentro del método **run()** del objeto hilo2 introducimos el try catch con el **join()** haciendo referencia al hilo1. Así podemos llamar a ambos hilos que hasta que no se termine de ejecutar el 1, no empieza el 2.

ReentrantLock

Si queremos que en programación concurrente una parte del código sólo pueda ser usada por un hilo a la vez, deberemos usar métodos pertenecientes a la interfaz **Lock** de la que hereda la clase **ReentrantLock**.

```
Lock bloqueo = new ReentrantLock();
bloqueo.lock();
```

```
try{
        código
}finally{
        bloqueo.unlock();
}
```

*Es importante desbloquear el código una vez que el hilo salga de él, por eso el código se mete en un try y el desbloqueo en un finally.

Condiciones en hilos

Para establecer que algunos hilos se mantengan a la espera y que dejen a otros hilos pasar hasta poder realizar su trabajo, necesitamos crear una condición y añadirla a nuestro objeto **ReentrantLock**.
Condition condicion1 = bloqueo.newCondition();

Ahora pondremos a la espera a los hilos con el método **await()** sobre nuestra condicion para relacionar a los hilos que se encuentran a la espera de alguna forma.
condicion1.await();

Para avisar a los hilos que se encuentran a la espera de que otro hilo ha pasado por el código y que puede que ahora se cumpla la condición, usaremos el método **signalAll()**.
condicion1.signalAll();

Condiciones con métodos de la clase Object

Podemos obtener un resultado similar a la de ReentrantLook con condiciones con métodos pertenecientes a la clase Object.

Para esto, haremos que las líneas de código que queramos bloquear se encuentren dentro de un método el cual porta la palabra reservada **syncronized** . Dentro de la condición que hará que nuestro código se

bloquee, ponemos **wait()** y al final de todo el código, ponemos **notifyAll()**.

Con la clase Object sólo podemos establecer un bloqueo y una condición a la vez, con la clase ReentrantLock podemos establecer varios bloqueos con varias condiciones.

Capítulo XII: Colecciones

Las colecciones son “arrays” dinámicos que se ensanchan y encogen a medida que vamos introduciendo y eliminando elementos.
Un ejemplo de una colección son los ArrayList.
Al igual que los arrays, empiezan a contar en 0.
No pueden almacenar datos de tipo primitivo (String sí).

Para instancia cualquier colección invocamos a esta y la igualamos a la clase a usar. (Queremos usar la clase HashSet:)
Set <E> nombre = new HashSet<E>();
(Sustituir E por el tipo de objeto que queramos almacenar (JFrame...))

La mayoría de métodos que pueden usar cada clase están recogidos en el API de JAVA en su interfaz respondiente.

*Para indicar que dos objetos con el mismo valor en una variable son iguales, deberemos sobreescribir los métodos hashSet() y equals() en la clase padre (vídeos 181 y 182). Eclipse lo hace de forma automática en: Source>Generate hashCode() and equals() y activaremos las respectivas check boxes conforma a las variables que queramos que se comparen a la hora de comparar si son iguales o no.

Título I: Tipos de colecciones

(Cada tipo es una interfaz)
-List: Podemos almacenar objetos repetidos. Podemos acceder a ellos de forma random (como la RAM). Sustituto de los arrays.
-Set: Solo permite almacenar objetos diferentes, ninguno repetido y no podemos acceder de forma numérica a estos.
-Map: Permite accesos repetibles que se indexan por una clave única arbitraria.

-Queue: No permiten acceso aleatorio, sólo se puede acceder a objetos que están al principio o al final de la cola.

Cada interfaz la podemos ver recogida en al menos una clase de la API de Java, las cuales usaremos para utilizar las colecciones.

removeAll(Collection<E> c)

Elimina de una colección la colección indicada c, si dentro de la primera colección hay elementos que están en la segunda colección.

Título II: List

Características

+ Acceso aleatorio.
+ Están ordenadas.
+ Añadir / Eliminar sin restricción.
+ ListIterator modifica en cualquier dirección.
+ Sintaxis similar a arrays.
- Bajo rendimiento en algunas operaciones (usar otras interfaces)

Clases

- **ArrayList:** Rápida leyendo elementos, muy frecuente, poco eficiente eliminando.
- **LinkedList:** Listas enlazadas, eficiente agregando y eliminando.
- **Vector:** Obsoleta, operaciones concurrentes.
- **CopyOnWriteArrayList** Programas concurrentes, eficiente en lectura pero no en escritura.

Título III: Set

Características

+ No permite elementos duplicados.
+ Uso sencillo del método **add()**.
- Sin acceso aleatorio.

- Poca eficiencia a la hora de ordenar elementos.

Clases

- **HashSet:** Rápida, no permite orden.
- **LinkedHashSet:** Orden por entrada, eficiente escribiendo, poco eficiente leyendo
- **TreeSet:** Ordenado (si la clase implementa la interfaz **Comparable**), poco eficiente.
- **EnumSet:** La mejor para tipos enumerados.
- **CopyOnWriteArraySet:** Para operaciones concurrentes, eficiente lectura, poco eficiente escritura, poco eficiente al eliminar.
- **ConcurrentSkipListSet:** Para operaciones concurrentes, admite orden, no eficiente con muchos elementos.

TreeSet

Crea una colección Set ordenada según el método **compareTo()** de la interfaz **Comparable** que implementa la clase a la que pertenecen los objetos a almacenar. De esta forma, si se crea un **TreeSet** de String, se ordenará alfabéticamente.

Si la clase a ordenar es una clase creada por nosotros, podemos hacer que esta implemente la interfaz **Comparable** y sobreescribir el método **compareTo()** para establecer el criterio del orden. (**compareTo()** devuelve un negativo, cero o positivo en función de la relación que tengan ambos objetos).

Si la clase a ordenar no es una clase creada por nosotros, la clase **TreeSet** tiene un constructor al que se le puede pasar un objeto de tipo **Comparator<E>** , el cual será el criterio por el que se ordenen los objetos introducidos. Esta interfaz tiene un método llamado

compare(T o1, T o2) , que compara dos argumentos y devuelve los mismos valores que **compareTo()** .
Para hacer todo esto, crearemos una clase que implementará la interfaz **Comparator**<E> y sobreescribir el método **compareTo()**.
Posteriormente, le pasaremos a nuestro **TreeSet** una instancia de esta clase.
Todo esto se puede simplificar usando clases internas anónimas.

Título IV: Map

Características

+ Asociación clave-valor.
+ No hay claves iguales.
- Poca eficiencia comparado con las demás.

Clases

- **HashMap:** No ordenada, eficiente.
- **LinkedHashMap:** Ordenada por inserción, permite orden por uso, eficiente lectura, no eficiente al escribir.
- **TreeMap:** Ordenado por clave, poco eficiente en todo.
- **EnumMap:** Permite enum como claves, muy eficiente.
- **WeakHashMap:** Crear elementos que el sistema borra si no se utilizan, poco eficiente.
- **HashTable:** Obsoleto, operaciones concurrentes.
- **CurrentHashMap:** Utilizado en concurrencia, no permite nulos.

HashMap

Para crear un HashMap (o un mapa en general), hacemos lo siguiente:
HashMap <K, V> nombre = new HashMap <K, V>();
(donde K es el tipo de dato que será la clave y donde V es el tipo de dato que se almacenará como el valor de su respectiva clave).

*Si introducimos otro elemento con la misma clave que otro, este se sustituirá por el segundo.

put(K k, V v)

Pone en la colección indicada el valor v de tipo V (indicado a la hora de instanciar la colección) con respecto a la clave k de tipo K (“).

remove(K k)

Elimina de la colección el elemento indicado en la clave k de tipo K.

Set entrySet()

Devuelve una colección Set que se copia a partir de nuestro Map.

Obtener claves y valor concreto

```
for(Map.Entry<K, V> nombre: nombremapa.entrySet()){
        K key = nombre.getKey();
        V valor = nombre.getValue();
}
```

Título V: Queue

Características

+ Muy rápidas accediendo al primer y último elemento.
+ Crea colas muy eficientes (LIFO/FIFO).
- Acceso lento a elementos intermedios.

Clases

- **ArrayDeque**
- **LinkedBlockingDeque**
- **LinkedList**
- **PriorityQueue**
- **PriorityBlockingQueue**

Título VI: Iteradores, Iterator <E>

Es una interfaz que construye un objeto iterador que recorre una colección de principio a fin determinando en cada caso qué hacer con el objeto que está evaluando actualmente.
El iterador copia el valor del elemento en el que se sitúa, por lo que podemos modificar estos elementos modificando el iterador en un momento determinado.

Los iteradores se crean con la interfaz **Iterator** , igualandola al método **iterator()** de nuestra colección (método que cada colección hereda de su interfaz) (marco es una colección que almacena JFrame):
Iterator <JFrame> nombre = marco.iterator();

*Si queremos que el iterador creado anteriormente vuelva a su posición inicial:
nombre = marco.iterator();

<E> next()

Devuelve el siguiente elemento a evaluar (<E>). Devuelve una excepción **NoSuchElementException** si no existe ningún elemento más y se invoca este método, por lo que es conveniente usarlo con **hasnext()**.

boolean hasnext()

Devuelve un boolean, true si hay un siguiente objeto, false si no.

remove()

Borra de la colección el objeto que se está examinando actualmente.

Existen otros tipos de iteradores, como el **ListIterator <E>**.
Este otro iterador nos permite añadir elementos (con su método **add()**) en la zona actual donde se encuentre ese iterador.

Capítulo XIII: Sockets

Los sockets son una especie de “puentes virtuales” que permiten conectar dos equipos (normalmente un ordenador cliente y un servidor) para transmitir información entre ambos.
Usaremos la clase **Socket**.

Para construir un socket necesitamos un programa en el ordenador, un programa en el servidor, la dirección del servidor (con una IP o un dominio) y un puerto de recepción. También necesitaremos el uso de OutputStream y InputStream.

Los sockets se suelen utilizar dentro del evento de un botón.

1. Para usar sockets, primero deberemos crearlos.
Para ello, usaremos el constructor que recibe una dirección y un puerto **Socket(InetAddress address, int port)** lanza una excepción.
Podemos introducir como **InetAddress** un String.
Socket socket = new Socket(“192.168...”, 9999);

2. A continuación, instanciaremos un **DataOutputStream** al que le pasaremos como parámetro el **OutputStream** de nuestro socket, que obtenemos con el método **getOutputStream()** sobre nuestro objeto socket.
*Si en vez de texto, queremos enviar un objeto, usar la clase **ObjectOutputStream**.
DataOutputStream stream = new DataOutputStream(socket.getOutputStream());

3. Ahora, le indicaremos al Stream que circulará por él (en este caso, el texto de un **JTextField** de nombre texto) y cerrando a su vez el socket y el stream. Una vez hecho esto, la parte del cliente ya estaría terminada.

*Si enviamos un objeto, **writeObject(Obj o)**.
stream.writeUTF(texto.getText());
stream.close();
socket.close();

---

4. Ahora, nos toca poner el servidor a la escucha del cliente:
El servidor va a permanecer constantemente a la escucha del cliente, por lo que para que pueda seguir realizando otros procesos, haremos que un hilo se encargue de esta tarea.

5. Una vez tengamos el hilo, creamos una instancia de **ServerSocket** dentro de este a la que le especificamos el mismo puerto que le especificamos al cliente. Este proceso lanza una excepción.
ServerSocket servidor = new ServerSocket(9999);

6. Posteriormente, aceptamos las conexiones que vengan de este puerto (como no sabemos en qué momento van a llegar, creamos un bucle infinito en el que integraremos los siguientes pasos), creando una instancia de **Socket** e igualándola al método **accept()** sobre nuestro **ServerSocket**:
Socket socket = servidor.accept();

7. Ahora crearemos un flujo de datos de entrada: Creamos una instancia de **DataInputStream** / **ObjectInputStream** a la cual le pasamos como parámetros el InputStream de nuestro socket con el método **getInputStream()** sobre nuestro socket. Posteriormente, leeremos nuestro socket con la instrucción **readUTF() / readObject()** (debemos hacer un casting ya que devuelve un objeto de tipo **Object**) igualandola a cualquier variable para almacenar la información.
DataInputStream stream = new DataInputStream(socket.getInputStream());

String texto = stream.readUTF();

8. Finalmente, cerraremos el socket y el stream.
stream.close();
socket.close();

Del punto 6 al 8 sería convenientes meterlos dentro de un bucle infinito para estar constantemente leyendo este socket.
(Creación de chat: 192 - 200 (vídeos)).

A la hora de usar sockets para transmitir distintos tipos de datos, es conveniente crear una clase que enviemos a través del socket.

Enviar objetos en vez de String

Los pasos a realizar por parte del programa cliente son los mencionados anteriormente, sustituyendo **DataOutputstream** por **ObjectOutputStream** , **writeUTF(String s)** por **writeObject(Obj o)** y haciendo que la clase que enviemos implemente la interfaz **Serializable**.

Los pasos para crear el servidor son similares a los mencionados anteriormente, a diferencia de que necesitamos crear una instancia de la misma clase enviada para poder construir el objeto serializado que recibimos por la red.
Sustituimos **DataInputStream** por **ObjectInputStream** y **readUTF()** por **readObject()**, el cual almacenaremos en la instancia mencionada anteriormente, mediante la realización de un casting.
readObject() lanza una excepción.

Obtener IP de clientes conectados al servidor

ServerSocket socketserver = new ServerSocket(9999);
Socket socket = socketserver.accept();
String IP = socket.getInetAddress().getHostAddress();

Capítulo XIV: Acceso a BBDD: JDBC

Título I: Pasos previos

JDBC (Java Data Base Connectivity) es un driver, un "puente" que conecta un programa Java y un programa gestor de BBDD. Este driver proporciona una conexión con la BBDD y manipula sus datos. El driver JDBC lo proporciona el fabricante del programa que gestiona las BBDD.

Consultar para este capítulo los apuntes de SQL, ya que es necesario tener conocimientos sobre este tema.

Software necesario: (vídeo 202).

1. Instalar mysql
2. Instalar driver jdbc mysql
3. Agregar el driver al classpath

Paquetes a manejar: **java.sql.**
Clases principales a usar: **DriverManager** .
Interfaces principales: **ResultSet** , **Connection** , **Statement** , **DataSource** .

Título II: Pasos para acceder a una BBDD

Proceso a la hora de acceder a BBDD:

1. Establecer la conexión con la BBDD.
2. Crear objeto **Statement**.
3. Ejecutar sentencia SQL.
4. Leer el **ResultSet**.

1. Para ello, necesitaremos un String que almacena la cadena de conexión a la base de datos (driver, protocolo del driver, detalles de la conexión del driver) y a veces un usuario y una contraseña.
Las conexiones pueden lanzar excepciones.

2. Para ello, usaremos el método **createStatement()** sobre nuestro objeto conexión para obtener un objeto **Statement.**

3. El objeto del paso 2 nos permitirá ejecutar el método **executeQuery(String Sentencia_SQL)** que nos devolverá un resultset (objeto donde se guarda la información solicitada por la sentencia SQL).

4. Leer y almacenar el resultset del punto 3 con los métodos **getString()** y **next()** y con la ayuda de bucles.

Título III: Primera conexión con una BBDD

1. Primero, debemos crear la conexión con la BBDD, la cual puede lanzar una excepción así que la capturaremos con un try catch. La conexión la creamos almacenando en un objeto de tipo **Connection** lo que nos devuelve el método estático **DriverManager.getConnection(String url, String user, String password)**

En url, indicaremos lo siguiente:
jdbc:mysql://ruta:puerto/nombreBBDD , donde si trabajamos en local, la ruta será localhost , y el puerto por defecto suele ser 3306 en Mysql. Sustituir el nombre de la BBDD por el correspondiente (el puerto no es necesario indicarlo siempre).
En usuario indicaremos **root** por defecto.
La contraseña por defecto suele estar vacía, por lo que : “”.

2. Creamos el objeto de tipo Statement: Creamos un objeto **Statement** que igualamos al método **createStatement()** que ejecutamos sobre nuestro objeto de tipo **Connection** .
3. Ejecutamos nuestra sentencia sql. Para ello, creamos un objeto de tipo **ResultSet** donde almacenaremos el resultado del query. Este objeto lo igualamos al método **executeQuery(String sql)** que ejecutamos sobre nuestro objeto **Statement** .
4. Leeremos el objeto de tipo **ResultSet** con la ayuda de algunos métodos que esta clase nos proporciona.
5. Cerramos nuestro objeto **ResultSet** para liberar memoria.
6. Cerramos nuestro objeto **Connection** para liberar memoria.

Métodos de ResultSet

first()

Mueve el cursor a la primera fila del resultado.

next()

Pasa a la siguiente fila del resultado.

getString(String campo)

Devuelve un String con el valor del campo especificado en **campo** en la fila en la que se encuentra el cursor (**objeto.getString("Nombre")**)

getString(int campo)

Hace exactamente lo mismo que **getString(String campo)**, salvo que indicamos la posición de la columna con un int (**objeto.getString(1)**) En este caso, se empieza a contar desde 1.

getDouble(String campo)

Hace exactamente lo mismo que **getString()** , salvo que nos devuelve un **double**.

getDate(String campo)

Hace exactamente lo mismo que **getString()** , salvo que nos devuelve un **Date**.

(Queremos obtener el código, el nombre y el precio de todos los artículos que hay en una tabla llamada productos en la BBDD curso_sql:)

```
try{
	Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost:3306/curso_sql", "root", "");

	Statement statement = conexion.createStatement();

	ResultSet resultado = statement.executeQuery("select * from productos");

	while(resultado.next()){
		System.out.println(resultado.getString("CODIGO") + ", " + resultado.getString("NOMBRE") + ", " + resultado.getString("PRECIO"));
	}
	resultado.close();
	conexion.close();
}catch(Eception e){
	e.printStackTrace();
}
```

Título IV: Insertar, actualizar y borrar

Al no querer ver ningún dato (ningún **select**), no es necesario recorrer el objeto **ResultSet**, por lo que no es necesario que tengamos uno.

El método **executeQuery(String sql)** devuelve un objeto **ResultSet** pero en este caso, no nos hace falta, ya que no vamos a recibir ningún parámetro, por lo que crearemos la conexión con la BBDD con nuestro objeto **Connection** y crearemos nuestro objeto **Statement**, pero en este caso, usaremos el método **executeUpdate(String sql)**, el cual nos permite insertar, actualizar o borrar información:

1. Primero, debemos crear la conexión con la BBDD, la cual puede lanzar una excepción así que la capturaremos con un try catch. La conexión la creamos almacenando en un objeto de tipo **Connection** lo que nos devuelve el método estático **DriverManager.getConnection(String url, String user, String password)**
2. Creamos el objeto de tipo Statement: Creamos un objeto **Statement** que igualamos al método **createStatement()** que ejecutamos sobre nuestro objeto de tipo **Connection** .
3. Ejecutamos nuestra sentencia sql. Para ello, ejecutamos el método **executeUpdate(String sql)** que ejecutamos sobre nuestro objeto **Statement** .
4. Cerramos nuestro objeto **ResultSet** para liberar memoria.
5. Cerramos nuestro objeto **Connection** para liberar memoria.

Título V: Consultas preparadas

Estas consultas nos permiten pasar parámetros a las sentencias SQL, por lo que nos da un mejor rendimiento ya que son sentencias precompiladas y reutilizables.
Las consultas preparadas son consultas que podemos usar repetidas veces cambiando los criterios.
Usaremos el método **prepareStatement(String sql)** de la clase **Connection** .

1. Para crear una consulta preparada, primero debemos crear la conexión con nuestra BBDD. Posteriormente, creamos un objeto de la interfaz **PreparedStatement**, que igualamos al método **prepareStatement(String sql)** sobre nuestro objeto **Connection**. Dentro del código sql, indicamos el código que queremos repetir, indicando unos "?" en los parámetros que variarán. ej:
select * from tabla where id=?
2. Posteriormente, indicamos el valor de todos los ? que tendrá esta consulta preparada, con el método **setString(int index, String value)** sobre nuestro objeto **PreparedStatement** para indicar qué valor tendrá cada interrogante (si en la BBDD es un int, **setInt(int index, int value)...**).
3. Finalmente, creamos nuestro objeto **ResultSet** que igualamos al método **executeQuery()** sobre nuestro objeto **PreparedStatement**. (Dentro del try de un try catch:)
4. Cerramos nuestro objeto **ResultSet** para liberar memoria.
5. Cerramos nuestro objeto **Connection** para liberar memoria.

```
Connection conexion = DriverManagar.getConnection(...);
PreparedStatement sentencia =
conexion.prepareStatement("select * from tabla where seccion=?
and pais=?");
sentencia.setString(1, "deportes");
sentencia.setString(2, "España");
ResultSet rs = sentencia.executeQuery();
while(rs.next()){
        código;
}
rs.close();
conexion.close();
```

Título VI: MVC

El MVC es un patrón de arquitectura que nos permite separar la parte lógica (modelo) de la interfaz de usuario (vista) y de las comunicaciones (o eventos) (controlador).
El esquema básico es: el controlador manda información al modelo, que manda un output a la vista.
Para dividir un programa en mvc, creamos 3 paquetes: un modelo, una vista y un controlador. (Se pueden crear más, como uno principal que contenga el método main).

Al trabajar con diferentes paquetes, tenemos que importarlos todos, para acceder a las clases.
Es recomendable encapsular todos los parámetros del modelo para acceder a él sólo con GETTERS y SETTERS.

Título VII: Procedimientos almacenados

Los procedimientos almacenados son una especie de "funciones" que se almacenan dentro de la BBDD para que accedamos a ellos desde diferentes programas. Los procedimientos almacenados pueden tener bucles y condicionales en su interior.
Para ejecutar procedimientos almacenados, tenemos que utilizar la clase **CallableStatement**.
Para ejecutar un procedimiento almacenado:
1. Crear la conexión a nuestra BBDD.
2. Crear un objeto **CallableStatement** con la ayuda del método **prepareCall(String call)** sobre nuestro objeto conexión en el que indicamos la llamada al procedimiento almacenado (**conexion.prepareCall("{call procedimiento}")**).
*La llamada se hace entre corchetes.
**En caso de que nuestro procedimiento no reciba parámetros, no es necesario indicar paréntesis.
3. Ejecutamos la sentencia con el método **executeQuery()** sobre nuestro objeto **CallableStatement**, el cual podemos igualar a un

ResultSet en caso de que este procedimiento almacenado nos devuelva algún dato.
*Si nuestra sentencia no nos devuelve nada, podemos usar directamente el método **execute()**
4. Cerramos el **ResultSet**.
5. Cerramos la conexión con la BBDD.

*Los procedimientos almacenados los creamos desde la BBDD.

```
Connection conexion = DriverManager.getConnection(...);
CallableStatement sentencia = conexion.prepareCall("{call procedimiento}");
ResultSet rs = sentencia.executeQuery();
//sentencia.execute();
```

Procedimientos almacenados que reciben parámetros

En este caso, debemos indicar los paréntesis y los parámetros que recibe los sustituimos por interrogantes: "?"
Posteriormente, debemos indicar los valores de estos interrogantes con los métodos **setInt(int index, int value)**, **setString(int index, String value)...** sobre nuestro objeto **CallableStatement** .
*En index indicamos a qué interrogante sustituirá la variable indicada (1º, 2º, 3º...).

```
Connection conexion = DriverManager.getConnection(...);
CallableStatement sentencia = conexion.prepareCall("{call procedimiento(?, ?)}");
sentencia.setInt(1, 2);
sentencia.setString(2, "España");
ResultSet rs = sentencia.executeQuery();
```

Título VIII: Transacciones

Las transacciones son una serie de sentencias SQL que se ejecutan en serie.
Si algo en una transacción falla, no se ejecuta ninguna sentencia, o se ejecuta todo o no se ejecuta nada.
Si algo se ejecuta pero algo falla después, los resultados afectados por sentencias vuelven a su estado anterior.

1. Para convertir varias sentencias sql en una transacción, después de establecer la conexión con la BBDD, debemos indicarle a esta con el método **setAutoCommit(boolean state)** que todas las sentencias serán parte de una transacción (esto lo hacemos indicando un **false**).
2. Posteriormente, después de ejecutar todas las sentencias, debemos ejecutar el método **commit()** sobre nuestro objeto **Connection** para confirmar los cambios.
3. Finalmente, como todo el código relacionado con BBDD se introduce dentro de try catch, debemos indicar dentro del **catch** de las transacciones el método **rollback()** sobre nuestro objeto **Connection** para que, en caso de que algo falle, todo vuelva a como estaba antes. (debemos indicar un try catch dentro del catch para esta línea de código).

```
try{
	//declaramos la conexión con la bbdd
	conexion.setAutoCommit(false);
	//ejecutamos todas las sentencias que queramos
	conexion.commit();
}catch(Exception e){
	try{
		conexion.rollback();
	}catch(Exception e2){
		código;
	}
}
```

Título IX: Metadatos

Los metadatos son los datos que describen cómo es una BBDD o alguna de sus partes. Existen varias clases de metadatos.
Todos estos metadatos los podemos obtener desde código Java.

Clases de metadatos

-Relativos a la BBDD: Información acerca del gestor de BBDD, la versión del gestor, el driver de conexión, la versión del driver…

-Relativos a un conjunto de resultados: Información acerca de resultados que podemos almacenar en un **ResultSet**: nombres de las tablas, nombres de los campos, tipos de datos en los campos, propiedades de campos...

-Relativos a parámetros de sentencias preparadas: Información acerca de sus parámetros: tipo de datos...

Obtención de metadatos

Para obtener los metadatos, primero debemos crear un objeto de tipo **Connection** sobre el que ejecutaremos el método **getMetaData()**, el cual nos devuelve un objeto **DatabaseMetaData**, desde el que podemos extraer bastantes metadatos ejecutando sobre él métodos como: **getDatabaseProductName()** (nombre del gestor BBDD) **, getDatabaseProductVersion() , getDriverName() , getDriverVersion() , getTables() , getColmns(), getString()** (este método se ejecuta sobre el **ResultSet** que devuelve **getTables()** o **getColumns()** , a este método le podemos indicar “TABLE_NAME” para que nos devuelva el nombre de las tablas o “COLUMN_NAME” para el nombre de los campos)**, …**
Estos tres últimos métodos devuelven un **ResultSet**.
(consultar API).

Connection conexion = DriverManager.getConnection(...)
DatabaseMetaData metadata = conexion.getMetaData();
…

Capítulo XV: JSP Y Servlets

JSP o Páginas de Servidor Java es un código Java que se ejecuta en un servidor el cual lee las acciones del usuario desde formularios HTML y devuelve una página HTML que se genera de forma dinámica (en tiempo real) dependiendo de las acciones del usuario.

Software necesario: (vídeo 228)

- Java JEE IDE
- Servidor web (Tomcat)

(Vídeos 228-245 JSP) (vídeos 246, 247 servlets) (necesario otra clase de eclipse y simulador de servidor web)

Capítulo XVI: MVC

MVC o Modelo Vista Controlador
(Vídeos 248-264)

Los capítulos XIV-XVI fueron saltados
(vídeos 202-264). Necesario: curso de BBDD mysql

Capítulo XVII: Introspección

La introspección es la capacidad de un programa para cambiar sus métodos en tiempo de ejecución.
Usaremos las clases:

- **Class<T>:** Para determinar a qué clase pertenece un objeto así como para crear nuevas instancias en tiempo de ejecución.
- **Modifier:** Para averiguar los modificadores de acceso.
- **Constructor<T>:** Para determinar cuántos constructores y de qué tipo tiene una clase.
- **Field:** Para determinar cuántos campos de clase y de qué tipo tiene una clase.

Obtener a qué clase pertenece un objeto
<T> getClass()

(De la clase **Object**) devuelve un genérico que será la clase a la que pertenece dicho objeto. Podemos igualar este método a un objeto perteneciente a la clase **Class**: (ventana es un objeto de tipo **JFrame**)
Class prueba = ventana.getClass();

getName()

Devuelve (String) el nombre de la entidad.
String x = prueba.getName();

static Class<?> forName(String Name)

Devuelve el objeto asociado con la clase que ha sido dada en **Name**. Lanza una excepción.

Consultar vídeos 271-273

Capítulo XVIII: Java Beans: Programar visualmente

Vídeo 274: qué son, cómo usarlos y cómo añadir nuevos Java Beans.