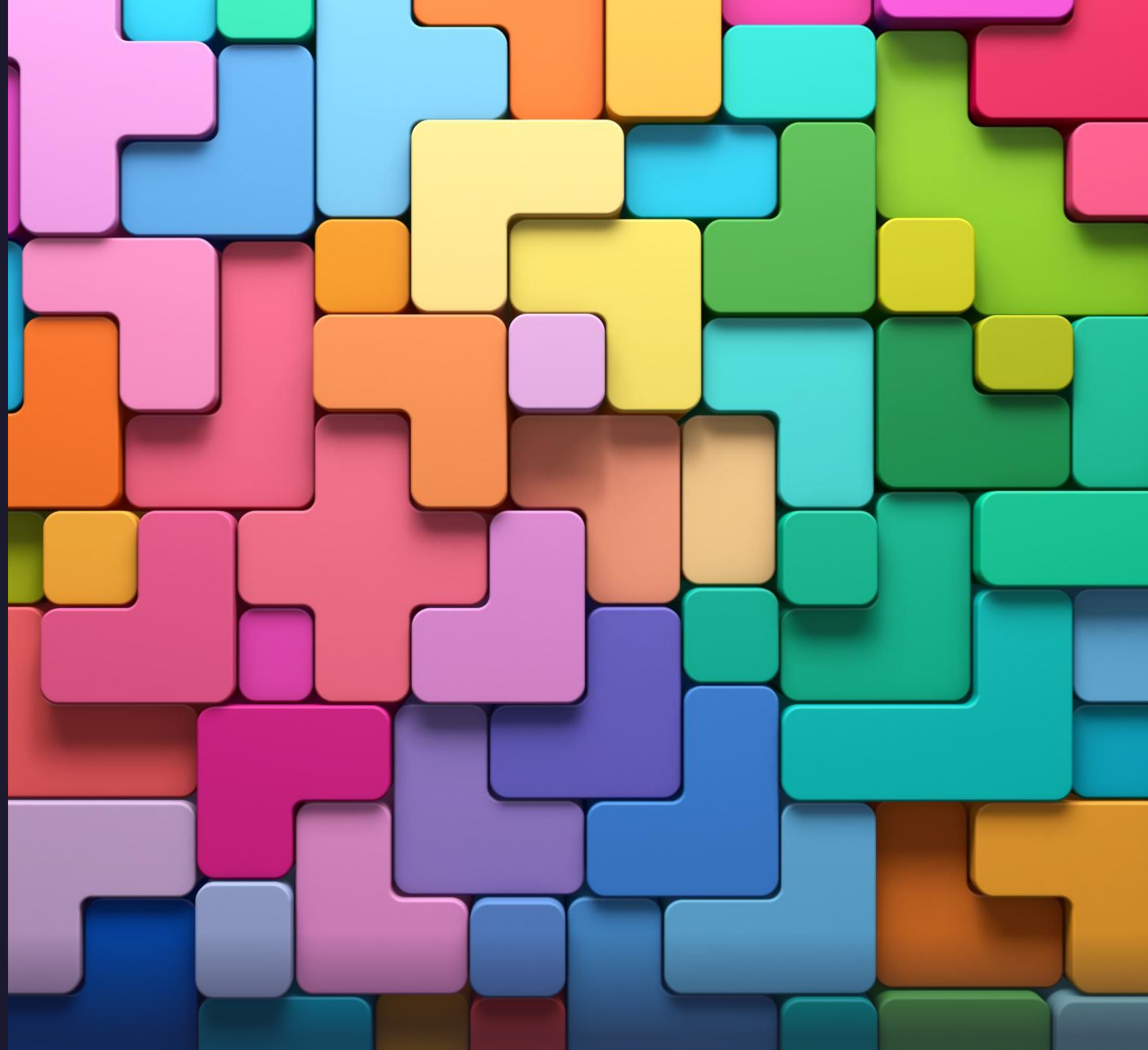
UNIT 3: Accessibility and Usability

2º DAM

IES La Mola

Enrique Laura Bernabeu



Introduction

- The concepts of usability and accessibility are intrinsically linked with user interfaces. The international standar ISO/IEC 25000 about the requirements and evaluation of software quality indicate that usability is refered to as the capacity a software has of being uderstood, learned, used and be atractive to the user.
- Accessibility is refered to as the idea that software, tools and technology is designed and developed for dissabled people to use, some of which are:
- Visual dissability: low visibility to blindness.
- Motor disability: dificulty to move the mouse or press various keys simultaneously.
- Auditory deficiencies: being able to hear sounds but not understand words to deaftness.
- Cognitive and language-related disorders: Complex or inconsisten exposition, poor choice of words.
- We have to design our UI with every user in mind.

Principles

- Control Identification and Distribution
- UI simplicity
- Frecuent control access
- Keyboard use
- Undo option
- Assitance from the UI

Look and Feel

Look and Feel

- The *look and feel* of a Java GUI is responsible of the aspect of the graphic components. Because Java is a multiplatform language, it uses a default aesthetic independently from the Operating System.
- To modify the aesthetic we use the class **UiManager** together with it's method `setLookAndFeel()`, qthat receives a `String` in which we indicate the *lookandfeel* that we want to use.
- In case we modify the *look and feel* of an application, **we have to stablish it at the beggining of the *main()* method that starts the application**.

Look and Feel: Dependent/Independent from the OS

- The *look and feel* aesthetic is, by default, known as *cross-platform look and feel, also called metal look and feel*. But in some cases we may be interested in using the OS *look and feel*.
- `//Usar cross-platform L&F, ó metal L&F`
- `UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());`
-
- `//Usar el look and feel del sistema anfitrión`
- `UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName();`
- *The method `setLookAndFeel()` throws exceptions that we must control by using a try-catch block.

Other Look and Feel

- Using the String we call the function with, we can modify the *look and feel* .
- **Metal or Cross-Platform**: Class`javax.swing.plaf.metal.MetalLookAndFeel`
- **Windows**: Class `com.sun.java.swing.plaf.windows.WindowsLookAndFeel`
- **Windows Classic**: Class `com.sun.java.swing.plaf.windows.WindowsClassicLookAndFeel`
- **Nimbus**: Class `javax.swing.plaf.nimbus.NimbusLookAndFeel`
- **CDE/Motif**: Class `com.sun.java.swing.plaf.motif.MotifLookAndFeel`

Other Look and Feel

```
try {
    UIManager.setLookAndFeel("com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel");
    //UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
    //UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
    //UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");

//UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsClassicLookAndFeel"
);
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (InstantiationException e) {
```

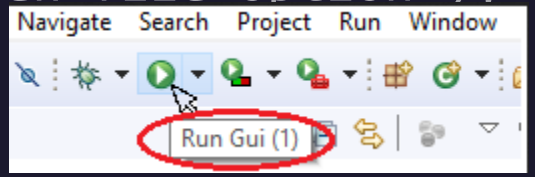
Intuitive Icons

Intuitive Icons

- The buttons of most applications are buttons that do not contain text, just an icon, the same way the elements of a menu also use an icon in addition to the text.
- We must select images for our icons with a representative form related to their function.

ToolTip Labels

- A **tooltip** is a visual help that allows us to insert a text that functions as an explanation about a component or control element of our application. This text will appear when we hover the mouse cursor over the element.
- `JButton btnMessage = new JButton("Open file");`
-
- `btnMessage.setToolTipText("Open file option");`



User Input Error Control

- We can warn the user when an error occurs with JOptionPane dialogs. It is very useful to notify of input errors when interacting with our application:
  - When Introducing an erroneous data type
  - Submit a new object without all its data introduced
  - Deleting an object that does not exist
  - Exiting without saving
  - Loading a file that does not exist

  For our interface to be robust, we can't allow uncontrolled errors that imped the use of our application

Keyboard Access

- As mentioned at the beginning of this section, a graphical interface should make keyboard handling easier, both for accessibility and also for speed.

Mnemonic Access

- I can perform actions using mnemonic keys. These keys are used by holding down the **ALT** key + mnemonic key. This is the simplest way of accessing by keyboard and can be associated with menu items and buttons.
- To use mnemonic access, the button or menu option must be visible. If it is in another section of the application and not visible, its mnemonic access cannot be used.
- They can be added through code or via WindowBuilder properties:

```
JButton btnNewUser = new JButton("New User");
```

```
btnNewUser.setMnemonic(KeyEvent.VK_N);
```

Now I can trigger the button action with **ALT + N**. Java underlines the letter of the mnemonic access.

Menu Accelerators

- For nested options inside a `JMenu`, we have another, more practical option than mnemonic keys: menu accelerators. Although documentation uses that name, they are basically keyboard shortcuts, but can only be applied to menu elements (`JMenuItem`, `JMenuCheckBoxItem`, etc.).
- The difference from mnemonics is that accelerators can trigger the action from anywhere in the application, regardless of whether the menu is expanded. Once configured, an indication appears next to the `JMenuItem`.
- They can be added either from WindowBuilder or via code:

```
// Pressing CTRL + N

mntmNewUser.setAccelerator(

KeyStroke.getKeyStroke(KeyEvent.VK_N, InputEvent.CTRL_DOWN_MASK));
```

Keyboard Shortcuts for Actions

- If we want the same functionality as accelerators but for buttons, we can define actions using the `Action` and `AbstractAction` classes. This is common when our application allows the same actions from a menu and from toolbar buttons.

```
JButton button = new JButton();

Action buttonAction = new AbstractAction("Refresh") {

    @Override

    public void actionPerformed(ActionEvent evt) {

        System.out.println("Refreshing...");    }};

    String key = "Refresh";

    button.setAction(buttonAction);

    buttonAction.putValue(Action.MNEMONIC_KEY, KeyEvent.VK_R);

    button.getInputMap(JComponent.WHEN_IN_FOCUSED_WINDOW).put(

    KeyStroke.getKeyStroke(KeyEvent.VK_F5, 0), key);

button.getActionMap().put(key, buttonAction);
```

Default Buttons

- Default buttons are those that are selected when using an application and respond to a key (usually **Enter**).
- There can only be one default button in each top-level container.

```
JButton btnAdd = new JButton("Add");

// Set default button

getRootPane().setDefaultButton(btnAdd);
```

Text Selection

- It is common that when clicking on a text field, its contents are selected so it is easier to modify: the selected text is deleted when typing any character. The same happens when pressing the **Tab** key to move from one text field to another.
- To implement this behavior, we must use a `FocusListener`. Focus refers to the property that makes something selected. When we click a text field, it gains focus; when clicking another, the previous one loses it.

Text Selection

- public class Controller implements FocusListener {
- myTextField.addFocusListener(this);
- @Override
- public void focusGained(FocusEvent evt) {
- if(evt.getSource() == myTextField){
- myTextField.selectAll(); } }
- @Override
- public void focusLost(FocusEvent evt) { } }

Undo Actions

- The **undo** feature is common in applications. We can undo the last change, or a list of changes.
- We need a data structure to store the change made (e.g., creation or deletion of an object). When performing the action, we must store the type of change and the object affected.
- That way, if we want to undo, we can recover the previous state of the application.
- Modifications can be stored in a list or we can keep only the last one.

Undo Actions

```
// Example class to store modifications
public class Modification {
    private String action;
    private Object modifiedObject;
    public Modification(String action, Object object) {
        this.action = action;
        this.modifiedObject = object;   } }
```

User Access Control

- If we want our application to control user access, we must create a login window. Usually, it consists of a small dialog where a username and password can be entered.
- The logic is simple: check if the username matches the password. We can support one or multiple users (e.g., stored in a list).
- To improve functionality, we can allow creating users with roles (e.g., admin vs. standard user).

Guidelines for creating a login window

- Use `JDialog` instead of `JFrame`
- Make the dialog modal
- Use a `JTextField` for the username and `JPasswordField` for the password
- Close (dispose) on success
- Show error and keep the window on failure
- Exit app on Cancel
- Remove title bar and buttons (`undecorated`)

Guidelines for creating a login window

```
public class VentanaLogin extends JDialog{
 //Declaro dos constantes con las credenciales
 private static final String USUARIO = "admin";
 private static final String PASS = "admin";
 public VentanaLogin(){
 //cuando es modal, la ejecución se detiene
 //cuando se cierra el dialogo, continua la ejecución del
programa
 setModal(true);
 //inicialización de componentes gráficos
 //elimino la barra de título
 setUndecorated(true);
 //Listener para botón de Login
 loginButton.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent e) {
 //Si la contraseña coincide, cierro la ventana
 if(comprobarUsuario(txtUser.getText(),
passwordField.getPassword())) {
 dispose();
 } else {
```

```
//Si no coincide lo indico, pero dejo la ventana
 JOptionPane.showMessageDialog(ventana, "Login incorrecto",
"Error", JOptionPane.ERROR_MESSAGE);
 } }  });
 //Listener boton cancelar
 cancelButton.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent arg0) {
 //Salgo de la aplicacion
 System.exit(0);
 }
 });
 }
 //Devuelvo true si la comprobación es correcta
 private boolean comprobarUsuario(String usuario, char[]
pass){
 if(USUARIO.equals(usuario) && PASS.equals(pass.toString){
 return true;
 }
 return false;
 }
}
```

Internationalization (I18n)

- Internationalizing an application makes it accessible for different regions by providing translations for text, images, etc., without rebuilding or recompiling.
- We can use **ResourceBundle files** to store visible texts (labels, fields, buttons, menus, etc.). The application will use the correct file depending on the `Locale`.

Locale Class

- Represents the geographical region and language. Java apps use system defaults but we can override them.
- Modify Localization:

```
public static void main(String[] args) {


    System.out.println(Locale.getDefault()); //Muestra es_ES

    Locale.setDefault(Locale.UK);

    System.out.println(Locale.getDefault()); //Muestra en_GB

}
```

Locale Class

- Create a Locale element
- `Locale locale = Locale.getDefault();`
-
- `locale = new Locale("es", "ES");`
-
- `locale = Locale.US;`
-
- `locale = new Locale.Builder().setLanguage("fr").setRegion("CA").build();`
-
- `Locale.setDefault(locale);`

Locale Class

- Region and Contry codes

Código de región	Región
ES	España
US	Estados Unidos
UK	Reino Unido
AU	Australia
BR	Brasil
CA	Canada
CN	China
DE	Alemania
FR	Francia
IN	India
RU	Rusia

Código de idioma	Idioma
es	Español
de	Aleman
en	Ingles
fr	Frances
ru	Ruso
ja	Japones
jv	Javanes
ko	Koreano
zh	Chino

.properties Files

- Properties files are plain text files. Their contents is organized in key-value pairs.

EtiquetasBundle.properties

```
# Fichero de idioma por defecto (español)
lblNombre = Nombre:
btnNuevoUsuario = Nuevo
menuItemGuardar = Guardar
 . . .
```

EtiquetasBundle_en.properties

```
# Fichero de idioma inglés
lblNombre = Name:
btnNuevoUsuario = New
menuItemGuardar = Save
 . . .
```

ResourceBundle Class

- This class allows us to load different files depending on the properties of the Locale object.
- Through the static method getBundle(), we stablish the properties of the localization.
  - GetBundle("myResourceBundle", [optional]locale)

    Without the optional parameter, it returns the current Locale object (the one in Locale.getDefault())

Replacing Static Text

- All static texts in code must be replaced by `bundle.getString("key")` calls so the application shows text according to the selected language.
-
- `JButton btnGuardar = new JButton("Guardar");`
- `JMenu menu = new JMenu("Archivo");`
- `JLabel lbl1 = new JLabel("Nombre:");`
-

- `ResourceBundle bundle = ResourceBundle.getBundle("EtiquetasBundle", locale);`
- `JButton btnGuardar = new JButton(bundle.getString("textoGuardar"));`

IntelliJ Internationalization Options

- Supports multiple `ResourceBundle` languages
- Allows adding keys directly from code or GUI form
- Helps translate `.properties` files

Splash Screen

- A splash screen is useful when an application needs time to initialize (e.g., DB connections, loading data). It hides delays and shows a loading indicator.

Splash Screen

SplashScreen.java

```java
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.GridLayout;
import javax.swing.ImageIcon;
import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JProgressBar;
import javax.swing.SwingConstants;

public class SplashScreen2 extends JDialog{

   private JProgressBar barraProgreso;

   public SplashScreen2() {
      setBounds(100, 100, 637, 566);
      JPanel contentPane = new JPanel();
      contentPane.setLayout(new BorderLayout());
      setContentPane(contentPane);

      //Creo una etiqueta con la imagen en el centro
      JLabel lblImagen = new JLabel();
      //Indico la imagen que quiero mostrar en la label
      lblImagen.setIcon(new ImageIcon(SplashScreen.class.getResource("/gui/splash.jpg")
      contentPane.add(lblImagen, BorderLayout.CENTER);

      //Creo un panel al sur con una barra de carga y una label para el autor
      JPanel panelInferior = new JPanel();
      panelInferior.setLayout(new GridLayout(2, 1, 0, 0));
      barraProgreso = new JProgressBar();
      //Muestra el % de carga
      barraProgreso.setStringPainted(true);
      panelInferior.add(barraProgreso);
      JLabel lblFersoft = new JLabel("FerSoft 2020");
      lblFersoft.setForeground(Color.BLUE);
      lblFersoft.setHorizontalAlignment(SwingConstants.CENTER);
      panelInferior.add(lblFersoft);

      //Anado el panel inferior al principal
      contentPane.add(panelInferior, BorderLayout.SOUTH);

      setResizable(false); //Impedir redimensionar la ventana
      setUndecorated(true); //Eliminar la barra de título y sus botones
      setLocationRelativeTo(null); //Mostrar en el centro
      setVisible(true);

      try {
         iniciarBarraCarga();
      } catch (InterruptedException e) {
         throw new RuntimeException(e);
      }
      //Al terminar la carga cierro la ventana
      dispose();
   }

   private void iniciarBarraCarga() throws InterruptedException {
      for(int i = 0; i <= 100; i++){
         Thread.sleep(20);
         actualizarBarraProgreso(i);
      }
   }

   private void actualizarBarraProgreso(int valor) {
      SwingUtilities.invokeLater(new Runnable() {
         @Override
         public void run() {
            progressBar1.setValue(valor);
         }
      });
   }
}
```

Help and User Manuals

- Every application usually has a Help section. In our case, we must provide at least user manuals.
- Simplest approaches:
  - Open a website in the system browser
  - Display a PDF inside the application
  - Display a web page in a window