




# UNIDAD 7

## DISTRIBUCION DE APLICACIONES

DESARROLLO DE INTERFACES

Ángel Gómez Fernández

- 
- Introducción
  - Paquetes autoinstalables
  - Herramientas para crear instaladores
  - Generación de paquetes de instalación



## Introducción

Un **instalador** simplifica el proceso de instalación de un programa en el ordenador del usuario:

- Se encarga de copiar todos los archivos a la ubicación necesaria
- Realiza todas las tareas de configuración
- Crea accesos directos
- ...
- Todo ello, de manera lo más transparente posible para el usuario final



## Introducción

Los pasos que va a realizar un instalador son los siguientes:

- Verificación de la compatibilidad: comprueba que se cumplen tanto los requisitos hardware como software
- Verificación de la integridad: se verifica que el paquete de software es el original
- Creación de los directorios requeridos
- Creación de los usuarios requeridos: cada grupo de usuarios puede usar un determinado software



## Introducción

- Copia, desempaquetado y descompresión de los archivos desde el paquete de software.
- Compilación y enlace con las bibliotecas requeridas
- Configuración
- Definición de las variables de entorno requeridas
- Registro de la aplicación ante el autor o autora de la aplicación



## Paquetes autoinstalables

Un **paquete autoinstalable**, es un único archivo que contendrá todos los archivos y directorios que forman la aplicación

En Windows será un **.exe** o un **.msi**

En Linux depende de la distribución:

- Debian, Ubuntu, etc → **.deb**
- Red Hat, Suse, etc → **.rpm**

## Herramientas para crear instaladores

Existen multitud de herramientas para crear instaladores:

- InstallAnywhere, Jexpress, InstallBuilder, Windows Installer, InstallShield, InstallAware, Wise Installation Studio, MSI Studio, NSIS, IzPack, Lift-Off, etc.

En esta unidad vamos a utilizar dos:

- **NSIS** (Nullsoft Scriptable Install SysTem)

Software libre, solo para Windows

- **IzPack**

Software libre, disponible para Windows y Linux

## Generación de paquetes

A la hora de generar paquetes instalación de una aplicación en Java, disponemos de varias alternativas:

- **Distribuir sin empaquetar.** Distribución del Jar con la aplicación y las librerías relacionadas (solo es necesario el entorno de desarrollo)
- **Paquete autoinstalable.** Es necesario el uso de herramientas externas como NSIS o IzPack.

En Linux también tenemos la opción de crear un paquete para una distribución (.deb por ejemplo)



## Netbeans - Entorno de desarrollo

En NetBeans, el código compilado y preparado para la distribución se encuentra en la carpeta **dist** del proyecto

Junto al jar con nuestro programa existe un directorio **lib** con todas las librerías necesarias para su ejecución si hemos marcado la opción en:

Para ejecutar la aplicación tendríamos dos opciones:

- Hacer doble click sobre el jar de la aplicación
- En la consola ejecutar: `java -jar nombreJar.jar`

## Netbeans - Entorno de Desarrollo (II)

Depende de la versión de Netbeans podemos encontrarnos que no copia las librerías a la carpeta dist aunque la opción esté marcada. En ese caso, accedemos al fichero build-impl.xml y comentamos la línea:

```
<condition property="do.mkdist">
  <and>
    <isset property="do.archive"/>
    <isset property="libs.CopyLibs.classpath"/>
    <not>
      <istrue value="${mkdist.disabled}"/>
    </not>
    <!--<not>
      <istrue value="${modules.supported.internal}"/>
    </not>-->
  </and>
</condition>
```

Si queremos un único .jar que nos empaquete librerías, clases, informes... podemos crear un paquete de almacenamiento. Para ello definimos un nuevo target en el fichero build.xml donde incluiremos:

```
<target name="nombre-del-objetivo" depends="jar">
  <property name="store.jar.name" value="nombre-del-jar"/>
  <property name="store.dir" value="store"/> <!-- nombre del directorio donde se guardará -->
  <property name="store.jar" value="${store.dir}/${store.jar.name}.jar"/>
  <echo message="Empaquetando ${application.title} en un solo JAR en ${store.jar}"/> <!-- Mensaje que aparecerá-->
  <delete dir="${store.dir}"/>
  <mkdir dir="${store.dir}"/>
  <jar destfile="${store.dir}/temp_final.jar" filesetmanifest="skip"> <!-- incluimos los directorios a empaquetar-->
    <zipgroupfileset dir="dist" includes="*.jar"/>
    <zipgroupfileset dir="dist/lib" includes="*.jar"/>
    <manifest>
      <attribute name="Main-Class" value="${main.class}"/> <!--Clase principal-->
    </manifest>
  </jar>
  <zip destfile="${store.jar}">
    <zipfileset src="${store.dir}/temp_final.jar"
      excludes="META-INF/*.SF, META-INF/*.DSA, META-INF/*.RSA"/>
  </zip>
  <delete file="${store.dir}/temp_final.jar"/>
</target>
```

## Netbeans - Entorno de Desarrollo(III)

Una vez hecho esto, tras hacer clean and build, sobre el fichero build.xml, con clic derecho, encontraremos la opción de Run Target y en Others aparecerá el que acabamos de crear en el xml.

Una vez hecho clic veremos en la consola cómo se van empaquetando los archivos. Al finalizar, en la carpeta seleccionada como destino, veremos un único jar que contiene todos los recursos necesarios para que nuestra aplicación funcione.

## Netbeans - Entorno de Desarrollo(IV)

En la carpeta dist por defecto al hacer el Clean and Build lo que veremos es el .jar de nuestra aplicación basado en los .class de nuestra aplicación y una carpeta /lib con las librerías .jar que necesita nuestra aplicación para funcionar.

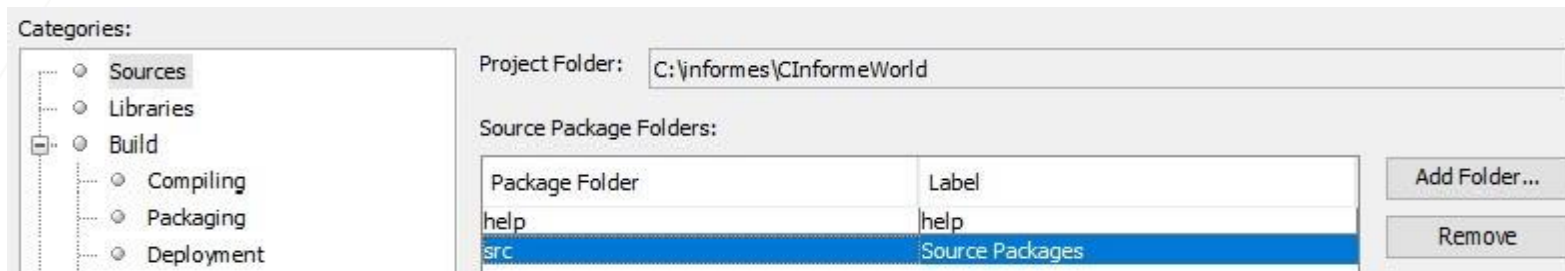
Hemos visto en unidades anteriores que además de recursos .class de nuestra aplicación, normalmente necesitamos agregar informes y ayuda para los usuarios en el funcionamiento de la aplicación. Esto o bien lo copiamos siguiendo rutas relativas con respecto a donde vayamos a tener ubicado el .jar (no muy profesional) o bien añadimos estos como recursos de nuestra aplicación y los cargamos desde esos recursos.

Si los informes ya los hemos guardado en la carpeta src, por defecto ya aparecerán dentro de nuestro .jar generado bajo el paquete principal. Si los hemos colocado en una carpeta, deberíamos añadir la carpeta como un source más del proyecto para que nos llévelos archivos correspondientes a los informes al .jar y podamos tratarlos como un recurso más. Eso si, podemos excluir los .jrxml (\*\*/\*.jrxml) y dejar únicamente los .jasper.

Lo mismo sucede con la ayuda. Si la hemos creado dentro de src, ya nos la llevará al .jar, si no, como es habitual, para tenerla separada del código de la aplicación, deberemos incluir la carpeta como un source más del proyecto para que nos lo incluya en la generación del .jar como un recurso más.

# Netbeans - Entorno de Desarrollo(V)

- En las propiedades del proyecto:



En la imagen se ve añadida la carpeta help que es donde están ubicados todos los recursos de la ayuda.

- Con esto al hacer clean and build, en el archivo .jar nos incluirá el contenido de help como un recurso más.



## Netbeans - Entorno de Desarrollo(VI)

- ▶ Para poder acceder a los recursos de la aplicación desde nuestro proyecto, basta con modificar algo el código que teníamos para hacer las pruebas de ejecución de la ayuda y los informes.
- ▶ En ambos casos debemos especificar que usamos un recurso en vez de un archivo físico con su ruta.
- ▶ Para los **informes**:

Ejemplo, si el paquete donde están los informes se llama Informes, para cargar el informe que se llame *EJ1\_PaisesInglesOficial.jasper*

```
JasperPrint print = JasperFillManager.fillReport(  
ClassLoader.getResourceAsStream("Informes/EJ1_PaisesInglesOficial.jasper"),  
params, conexion);
```

- Para la **ayuda**:

Ejemplo, si el archivo helpset de nuestra ayuda se llama help\_set.hs y está dentro de la carpeta help en la carpeta raíz del proyecto, para cargarlo sería:

```
ClassLoader loader = getClass().getClassLoader();  
URL helpSetURL = HelpSet.findHelpSet(loader, "help_set.hs");  
HelpSet helpset = new HelpSet(getClass().getClassLoader(), helpSetURL);
```

## Netbeans - Entorno de Desarrollo(VII)

- ▶ Realizado lo anterior ya podríamos realizar el clean and build del proyecto, y nos generaría un .jar totalmente operativo con las clases, los informes y la ayuda.
- ▶ Con este .jar de la carpeta dist generado, usando la creación del package como hemos visto, obtendríamos un .jar con todo lo necesario para funcionar correctamente o para generar el correspondiente .exe o instalador que veremos más adelante.



## Ej. Crear un .exe a partir de un .jar - Launch4j

1. Desde Netbeans, **limpiamos y construimos** nuestro proyecto java.
2. En la consola podemos ver como se ha creado un directorio llamado **dist** en nuestro proyecto que contiene un archivo jar y las librerías utilizadas.
3. Debemos tener una **clase main**, de lo contrario no podremos abrir el archivo jar y al probar el proyecto en NetBeans nos saldrá un mensaje indicando que el proyecto no tiene main.
4. Copiamos los archivos de la carpeta dist en una nueva carpeta.

## Ej. Crear un .exe a partir de un .jar - Launch4j

### 5. Abrir launch4j.

- Debemos ser cuidadosos al poner las versiones y demás datos. En Output File debemos poner la ruta del directorio donde queremos crear el archivo.
- No olvidéis la extensión .exe (en caso de Ubuntu sería .deb o .rpm).
- En Jar ponemos el archivo .jar de nuestra aplicación java.
- Podemos colocar un icono para la aplicación en Icon.
- Si seleccionamos **Don't wrap the jar, launch only** ocupará menos espacio ya que no contendrá el archivo .jar. Para poder ejecutarse correctamente en este caso, deberemos tener en el mismo directorio el archivo exe y el archivo jar.

Cuando acabamos de rellenar las pestañas con la configuración pulsaremos el engranaje:



## Ej. Crear un .exe a partir de un .jar - Launch4j

Este generará un archivo xml con los datos de configuración. Si quisiéramos hacer algún cambio para nuestro ejecutable podemos abrir este archivo con el programa y pulsando de nuevo el icono del engranaje sobrescribiríamos la configuración y se sustituiría también el archivo exe.

En caso de tener librerías en el proyecto NetBeans, para abrir correctamente el archivo jar, debemos tener la carpeta lib en el mismo directorio que el archivo jar. Al igual ocurre con el ejecutable .exe generado por el jar. Cuando hagamos el instalador veremos que podemos añadir las librerías para tener únicamente el archivo del instalador.

Supongamos que queremos tener un solo jar, es decir que las librerías jar del proyecto formen parte del jar del proyecto y no tengamos que estar pendiente de tenerlas en el mismo directorio que el archivo jar de la aplicación.

Para ello, después de generar el jar de la aplicación acudiremos en NetBeans a la pestaña Files y abriremos el archivo build.xml, que es el archivo que se ejecuta cuando se compila el proyecto.

## Ej. Crear un .exe a partir de un .jar - Launch4j

A continuación aparece en azul el código que habría que añadir si quisiera añadir librería.jar a mi proyecto, donde la clase main se llama ListImpresaExtended.class del paquete interfaz.extendidas:

```
<project name="Proyecto" default="default" basedir=". ">
```

```
  <target name="-post-jar">
```

```
    <jar jarfile="${dist.jar}" update="true">
```

```
      <zipfileset src= "C:\Proyecto\dist\lib\librería.jar" excludes="META-INF/*"/>
```

```
      <manifest>
```

```
        <attribute name= "Main-Class" value=
"interfaz.extendidas.ListImpresaExtended" />
```

```
      </manifest>
```

```
    </jar>
```

```
  </target>
```

```
<description>Builds, tests, and runs the project.</description>
```

```
<import file="nbproject/build-impl.xml"/>
```

El primer valor resaltado es la ruta del archive .jar que queremos incluir.

El segundo es la clase main del proyecto, cuya ruta sigue el esquema (partimos de src):

nombrePaquete.nombreClase



## Paquetes autoinstalables

Para crear paquetes autoinstalables existen herramientas como:

- **IzPack**

- Permite crear instaladores para **cualquier plataforma** y tipo de aplicación
- Necesita que Java esté instalado en la máquina del cliente (Java 1.5 o superior)
- Áltamente configurable
- Permite crear instaladores que funcionan directamente desde la web
- Tendremos que crear un archivo xml que define nuestro instalador
- Software libre bajo licencia Apache



## Paquetes autoinstalables (II)

- **NSIS** (Nullsoft Scriptable Install System)
  - Funciona a través de su propio lenguaje de scripts
  - Permite crear tanto instaladores como desinstaladores
  - Solo disponible para plataformas Windows
  - Una vez creados los scripts de instalación, NSIS los incorpora en un fichero ejecutable (.exe) que puede ser instalado en otro ordenador
  - Software libre (combinación de licencias)
  - Actualmente versión 3.05

# NSIS

- El primer paso será crear un fichero con extension .nsi. Puedes usar como editor Notepad++.
- La primera línea, en el caso de querer usar el UI moderno será la siguiente:

```
!include "MUI.nsh"
```

- En los scripts NSIS cada línea es tratada como un comando. Si nuestra línea es demasiado larga podemos usar el back-slash ' \ ' al final de la línea para continuar en la siguiente, como por ejemplo:

```
MessageBox MB_YESNO|MB_ICONQUESTION \  
"¿Quiere borrar todos los ficheros de este directorio? \  
(Si creó algo que quiera mantener, click No  
IDNO NoRemoveLabel
```

- Un script NSIS puede contener atributos del instalador, páginas, secciones y funciones

## NSIS (II)

- **Atributos del instalador**

Determinan el comportamiento y el look and feel de nuestro instalador. Con estos atributos podemos cambiar los mensajes que se mostrarán durante la instalación.

Name "Mi nombre"

InstallDir \$DESKTOP

OutFile "instalador.exe"

ShowInstDetails show

SetOverwrite on

LicenseForceSelection radiobuttons "Acepto" "Rechazo"

- **Donde:**

- Name es el atributo correspondiente al nombre de nuestra aplicación
- InstallDir será el directorio elegido para instalar la aplicación
- Outfile establece el nombre del fichero generado
- ShowInstDetails muestra los detalles de la instalación al usuario
- SetOverwrite en caso de encontrarse los ficheros se sobrescriben, etc.
- LicenseForceSelection fuerza a aceptar la licencia antes de proceder



## NSIS (III)

- Variables y Constantes

Se declaran mediante Var y se pueden copiar con StrCpy.

Var BLA

;Declaramos la variable

Section bla

    StrCpy \$BLA "123" ;Ahora la variable BLA vale "123"

SectionEnd

Existen variables predefinidas como:

\$INSTDIR: directorio de instalación

\$SMPROGRAMAS: menú de inicio

\$DESKTOP: path al escritorio

Para valores constantes usaremos la directiva #define:

!define APP\_NAME "Ejercicio1"

Name "\${APP\_NAME}"

OutFile "\${APP\_NAME}.exe"

## NSIS (IV)

- Páginas

Un instalador puede mostrar diferentes páginas al usuario, como por ejemplo la página bienvenida, la de aceptación de licencia , la de selección del directorio de instalación etc..:

;Mostramos la página de bienvenida

!insertmacro MUI\_PAGE\_WELCOME

;Página donde mostramos el contrato de licencia

!insertmacro MUI\_PAGE\_LICENSE "licencia.txt"

;página donde se muestran las distintas secciones definidas

!insertmacro MUI\_PAGE\_COMPONENTS

;página donde se selecciona el directorio donde instalar nuestra aplicación

!insertmacro MUI\_PAGE\_INSTFILES

;página final !insertmacro MUI\_PAGE\_FINISH

## NSIS (IV)

- Icono e imagen de cabecera

Es posible personalizar muchos aspectos del instalador como por ejemplo el icono o la imagen de cabecera del instalador:

```
!define MUI_ICON "calimero.ico"
```

```
!define MUI_HEADERIMAGE
```

```
!define MUI_HEADERIMAGE_BITMAP "calimero.bmp"
```

```
!define MUI_HEADERIMAGE_RIGHT
```

- Secciones

En un instalador pueden hacerse categorías de instalación. Y así separar la instalación en varios componentes, dando a elegir al usuario cuales instalar y cuales no.

```
Section "My Program"
```

```
    SetOutPath $INSTDIR
```

```
    File "My Program.exe"
```

```
    File "Readme.txt"
```

```
SectionEnd
```

En el código anterior, File extrae los ficheros y SetOutPath le dice dónde. Con ficheros podemos hacer las operaciones típicas: FileOpen, FileWrite, FileClose, Delete, RMDir /r

# NSIS (V)

## ► Funciones

Contienen código semejante a las secciones , pero se diferencian de éstas en el modo en que se llaman. Hay dos tipos de funciones:

- Las definidas por el usuario, que se llaman con la instrucción Call
- Las que se activan cuando ocurren determinados eventos en la instalación

Function .onInit

```
    MessageBox MB_YESNO "Esto instalará mi programa ¿Quiere  
continuar<u>?"</u>
```

```
IDYES gogogo
```

```
    Abort
```

```
gogogo:
```

FunctionEnd

Abort es una función especial que hace que el instalador termine inmediatamente.

MessageBox es una función de E/S que muestra un mensaje.

## NSIS (VI)

### ► Compilación y prueba de ejecutables

Es posible compilar un fichero nsi escogiendo la opción “Compile NSIS Script” del menu contextual, o bien abrir el NSIS y escoger la opción “Compile NSI Script” de la sección Compiler:



- Si el proceso ha ido bien se habrá creado un ejecutable con el instalador. Podéis probarlo desde el propio sistema operativo, o desde el botón “Test Installer” de la ventana de salida de NSIS:

## Paquetes autoinstalables (III)

- **INNS** (Inno Script/Setup Studio)
  - Funciona a través de su propio lenguaje de scripts. Permite crear tanto instaladores como desinstaladores y poder configurar la instalación con bastante detalle. Sólo disponible para plataformas Windows.
  - Una vez creados los scripts de instalación, los incorpora en un fichero ejecutable (.exe) que puede ser instalado en otro ordenador.
  - Licencia BSD modificada
  - En la carpeta software de la unidad está la guía de usuario completa junto a un ejemplo de fichero iss.

# Versiones

Los métodos y herramientas disponibles para controlar todo lo referente a cambios en el tiempo de un archivo se le conoce como **Control de Versiones**.

Difícilmente se termina un archivo o document en el primer intento, normalmente encontramos algún bug o algún error ortográfico o cualquier otra cosa que implique un cambio al archivo original.

También es necesario para los propios desarrolladores del software, para que en todo momento sepan los cambios que se están produciendo.

## **Control de versiones con el método X.Y.Z**

El método más común para numerar las versiones de un sistema es basándose en dos o tres cifras decimales, dependiendo de la importancia de los cambios es el número que se debe cambiar. La primera cifra siempre se cambia cuando se hizo una modificación crítica o muy importante (major), siendo la segunda cifra de menor importancia (minor).

- Se debe iniciar desde **0.Y.Z**, con esto estamos diciendo que el documento no está listo aún o que no cumple con los requerimientos mínimos.
- Cada cambio en esta cifra denota una reescritura o la incompatibilidad con versiones anteriores.

## Versiones (II)

- La segunda cifra X.**0**.Z se cambia cuando hay modificaciones en el **contenido o la funcionalidad** del documento, pero no lo suficientemente importantes como para decir que ya no es el mismo documento.

Cuando se hace un cambio mayor (en la primer cifra), el segundo número se reinicia a **0**

- La tercer cifra se cambia cuando se hacen **correcciones al documento pero no se ha añadido ni eliminado nada relevante**.

Si se hace un cambio en la segunda cifra se debe reiniciar el número de la tercera a **0**

Ejemplo: Supongamos que actualmente estamos en la 1.7.2 y hacemos un cambio que aumenta el menor (el 7). En número de versión queda por tanto 1.8.0. En cambio, si hiciéramos un cambio en mayor (el 1) quedaría 2.0.0.



## Versiones (III)

- Además de tener las versiones por números podemos agregar una clasificación **por estabilidad** del proyecto.
- **Alpha** es una versión inestable que es muy probable que tenga muchas opciones que mejorar, pero queremos que sea probada para encontrar errores y poder poner a prueba funcionalidades, en la mayoría de los casos podemos decir que está casi listo el producto. Ejemplo: 1.0Alpha, 1.0a1, 1.0a2.
- **Beta** una versión más estable que Alpha en la que contamos con el producto en su totalidad, y se desea realizar pruebas de rendimiento, usabilidad y funcionamiento de algunos módulos para ver cómo funciona bajo un ambiente no tan controlado. Aquí aparece el nombre de Beta Tester que escuchamos mucho en el mundo del software. Ejemplo: 2.0Beta, 2.0b, 2.0b1
- El siguiente paso es **RC (Release Candidate)** es una versión candidata a versión final o también llamada candidata para el lanzamiento.
  - El Producto está preparado para ser publicado como versión definitiva. Se encuentra en una fase de pruebas y será publicado a menos que haya errores que se lo impidan.
- Fase estable versión beta 100% estable. Esta última versión del software aporta ajustes y correcciones gráficas para concretar todas las funcionalidades previstas en el proyecto, que ya estaban activas y funcionando en la beta y que se han probado y testeado, tanto por los desarrolladores como por los usuarios o clientes.