



Ministerio de Producción  
Presidencia de la Nación

Ministerio de Educación y Deportes

Subsecretaría de Servicios Tecnológicos y Productivos



**PROGRAMACIÓN ORIENTADA  
A OBJETOS**



# Programación Orientada a Objetos

## Temas:

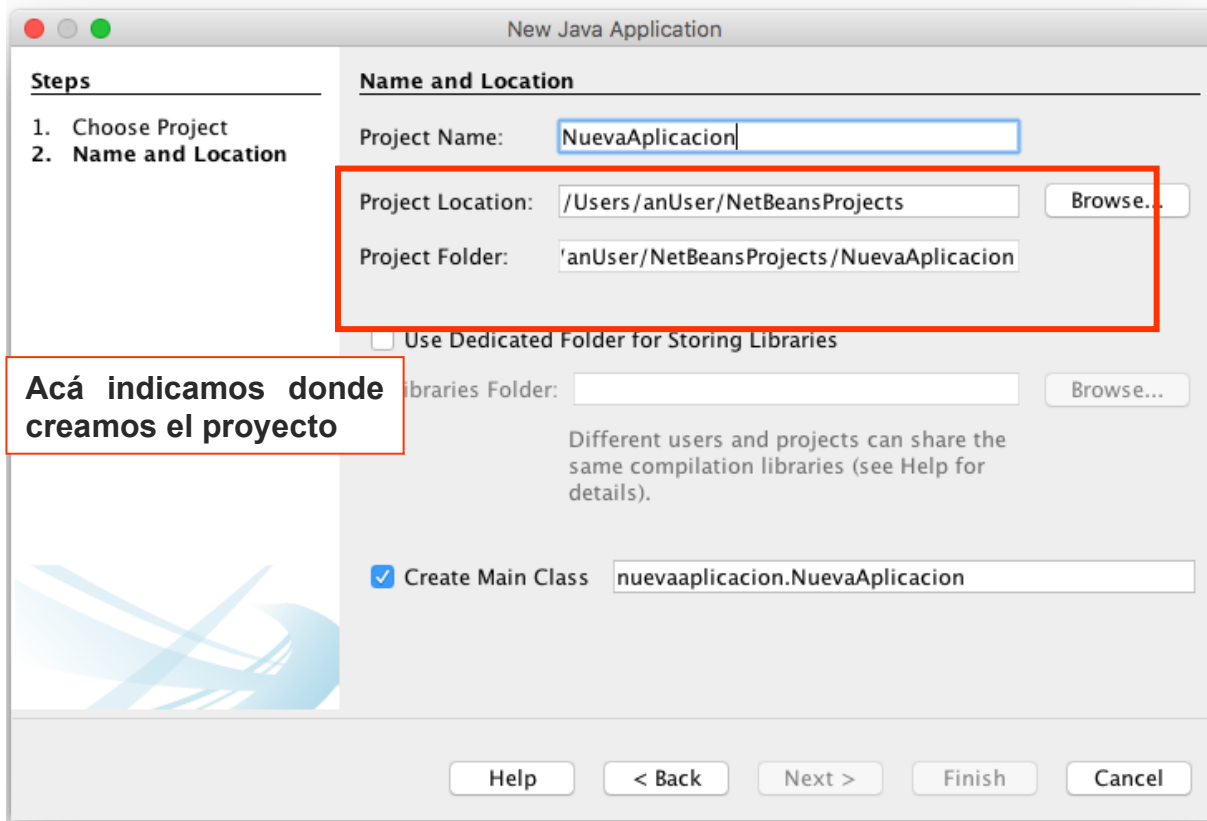
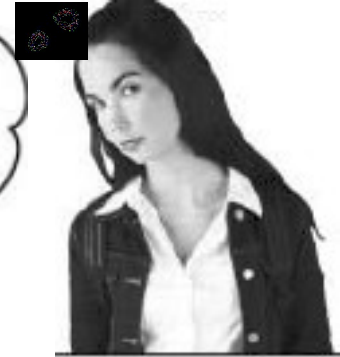
- Paquetes en Java
- El Classloader
- La variable de entorno CLASSPAH
- Archivos **JAR** (Java Archive)

# Paquetes – Organización

- Cuando se desarrolla una aplicación, es bueno organizar las distintas componentes que la conforman. Esto involucra dos partes fundamentales:
  1. **Organización a nivel de paquetes:** los paquetes clasifican a las clases y las agrupan de acuerdo a un criterio.
  2. **Separar los archivos fuentes (.java) de los archivos compilados, los bytecodes (.class):** esto facilita la preparación del software que se entregará a los clientes.
- El punto 1 lo hemos tratado en la clase anterior. Ahora nos concentraremos en cómo deberíamos hacer de forma práctica esta separación de los archivos fuentes de los bytecodes.

# Configuración de los fuentes

¿Cómo configuro la ubicación de los archivos fuentes y compilados (*bytecodes*) en *NeatBeans*?



**Steps**

1. Choose Project
2. **Name and Location**

**Name and Location**

Project Name: NuevaAplicacion

Project Location: /Users/anUser/NetBeansProjects **Browse...**

Project Folder: anUser/NetBeansProjects/NuevaAplicacion

☐ Use Dedicated Folder for Storing Libraries

Libraries Folder:  **Browse...**

Different users and projects can share the same compilation libraries (see Help for details).

☒ Create Main Class nuevaaplicacion.NuevaAplicacion

**Help** **< Back** **Next >** **Finish** **Cancel**

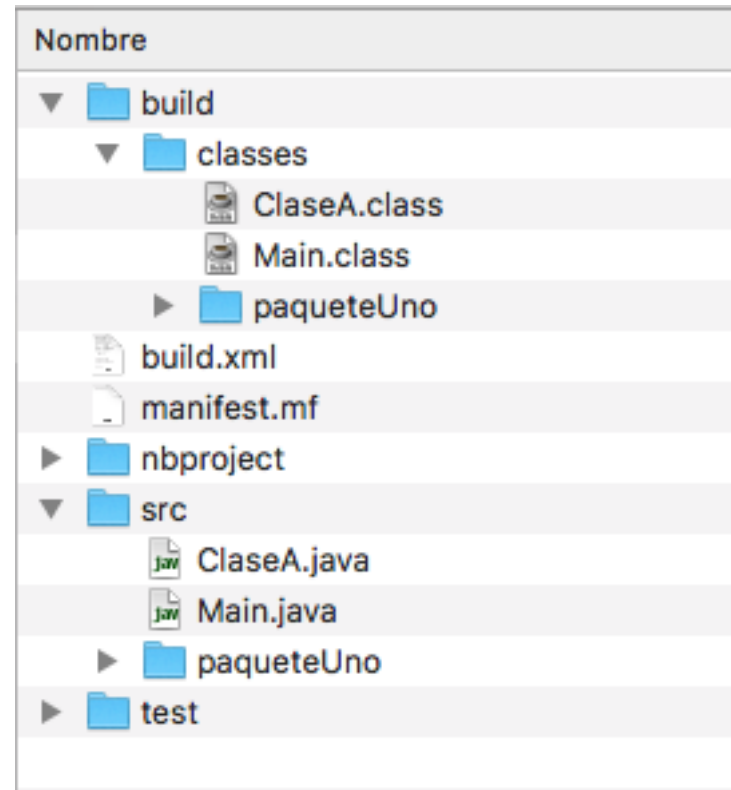
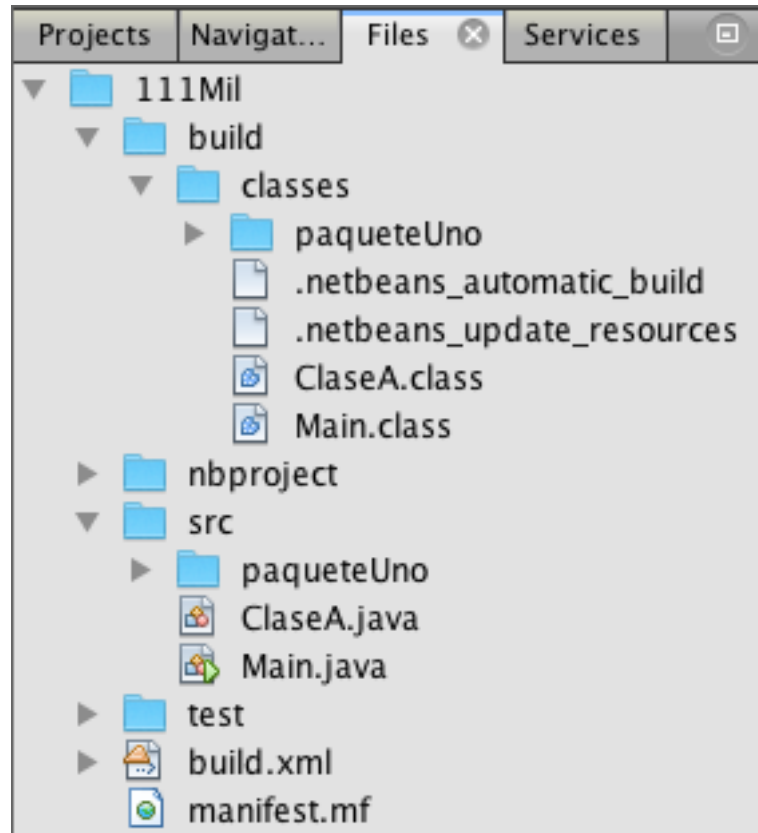
Acá indicamos donde creamos el proyecto

Los archivos fuentes se almacenan por default en la carpeta src y los compilados en la carpeta build

# Configuración de los fuentes

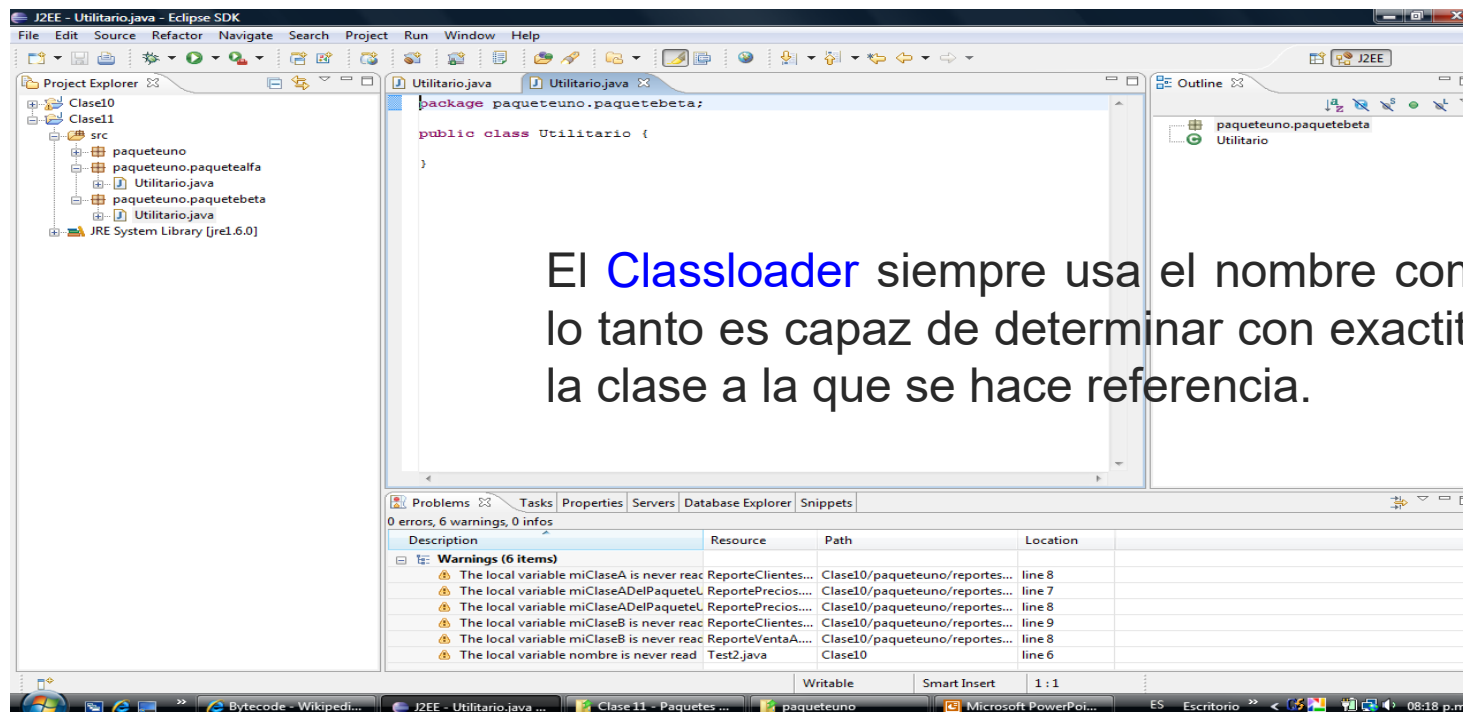
Utilizando la vista Navigator podemos observar como los fuentes (.java) y los *bytecodes* (.class) se encuentran en directorios diferentes de acuerdo a como lo configuramos.

En el sistema de archivos  
también están almacenados así:



# ClassLoader

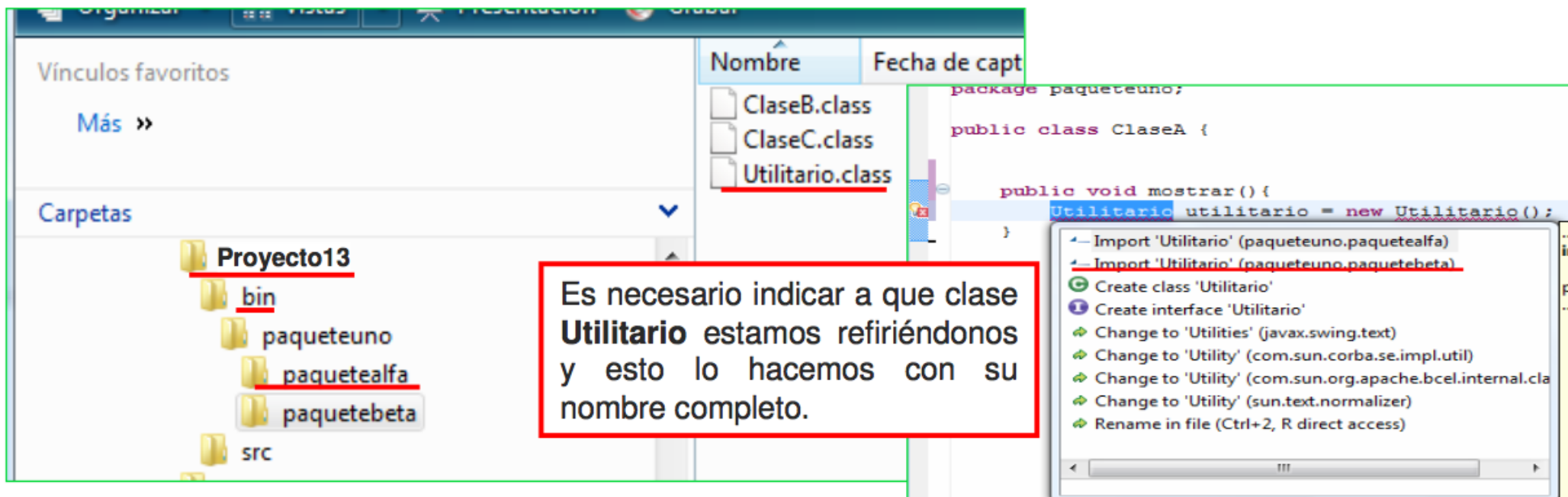
- Cuando instanciamos una clase usando **new()** el **ClassLoader** carga la clase (.class) correspondiente. Esta clase estará determinada por su **nombre completo** (nombre del paquete + nombre de la clase propiamente dicha).



# ClassLoader

## ¿qué pasa si tenemos dos clases con el mismo nombre Utilitario?

Como el **ClassLoader** trabaja con el nombre de la clase completa, no hay problema. Si desde alguna clase hacemos referencia a la clase **Utilitario**, siempre tenemos que indicar en qué paquete se encuentra, en este caso **paqueteuno.paquetebeta** o **paqueteuno.paquetealfa**



Vínculos favoritos  
Más »

Carpetas

- Proyecto13
  - bin
    - paqueteuno
      - paquetealfa
      - paquetebeta
    - src

Nombre	Fecha de capt
ClaseB.class	
ClaseC.class	
<u>Utilitario.class</u>	

```
package paqueteuno;  
  
public class ClaseA {  
  
    public void mostrar() {  
        Utilitario utilitario = new Utilitario();  
    }  
}
```

Es necesario indicar a que clase **Utilitario** estamos refiriéndonos y esto lo hacemos con su nombre completo.

- Import 'Utilitario' (paqueteuno.paquetealfa)
- Import 'Utilitario' (paqueteuno.paquetebeta)
- Create class 'Utilitario'
- Create interface 'Utilitario'
- Change to 'Utilities' (javax.swing.text)
- Change to 'Utility' (com.sun.corba.se.impl.util)
- Change to 'Utility' (com.sun.org.apache.bcel.internal.clas)
- Change to 'Utility' (sun.text.normalizer)
- Rename in file (Ctrl+2, R direct access)

# ClassLoader

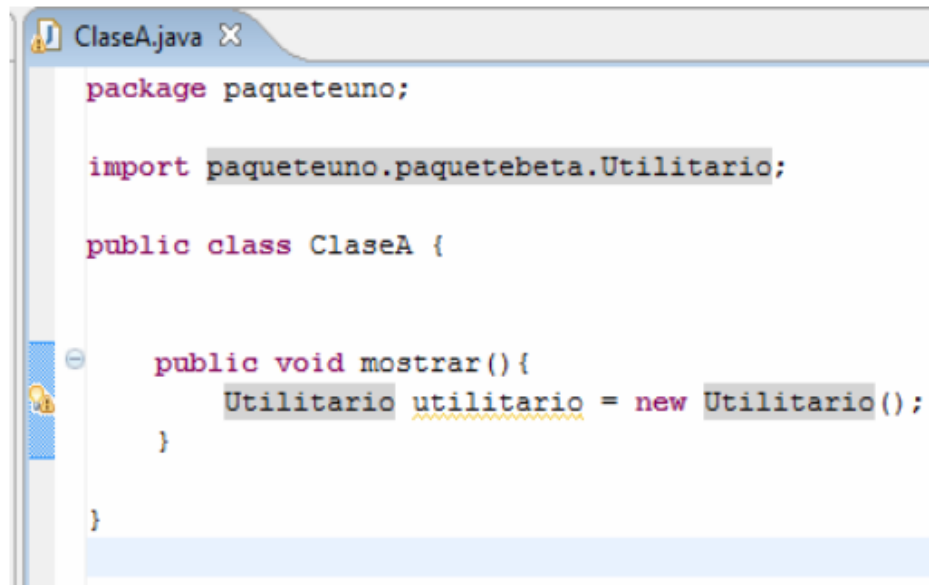
¿Qué es el Java **ClassLoader** o cargador de clases Java?

- Es una parte del **Java Runtime Environment (JRE)** que carga dinámicamente clases Java en la Máquina Virtual Java.
  1. Una librería de software es una colección de código compilado o ejecutable.
  2. En el lenguaje de programación Java, las librerías de software están típicamente empaquetadas en archivos **JAR** (Java ARchive) y contienen un conjunto de archivos `.class`.
  3. El **ClassLoader** es responsable de localizar las librerías, leer su contenido y, cargar en memoria los `.class` que están adentro de las librerías.
  4. La carga es normalmente hecha "bajo demanda", por lo que no ocurre hasta que la clase sea usada por el programa, por ejemplo cuando se crea la primera instancia.
  5. Los programas Java pueden hacer uso de librerías externas o de terceras partes (esto es, librerías escritas y suministradas por alguien distinto del autor del programa).



# ClassLoader - Classpath

¿Cómo encuentra el Classloader las clases que necesita cargar de acuerdo a lo que solicita el programa?



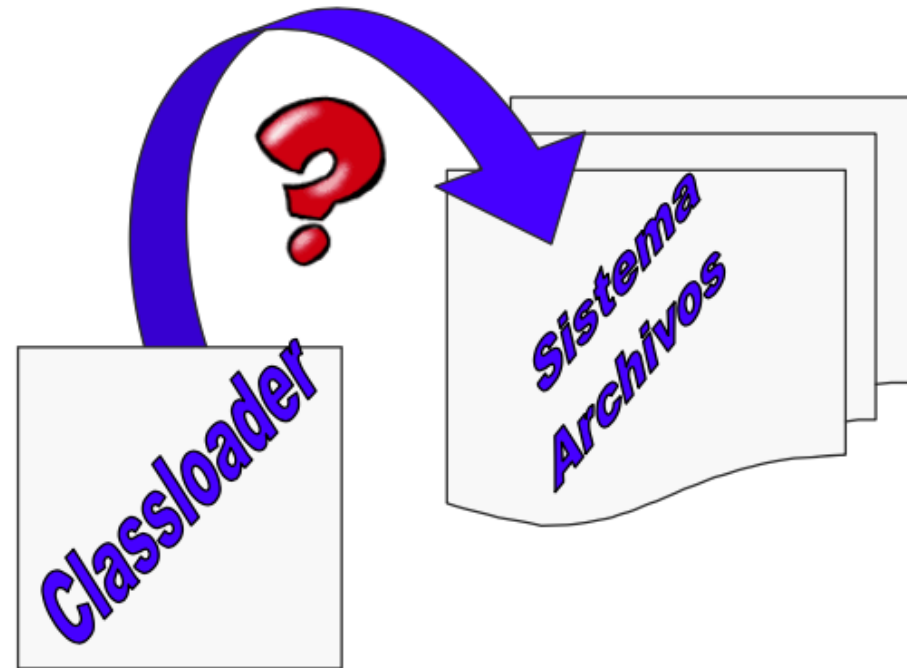
```
ClaseA.java X
package paqueteuno;

import paqueteuno.paquetebeta.Utilitario;

public class ClaseA {

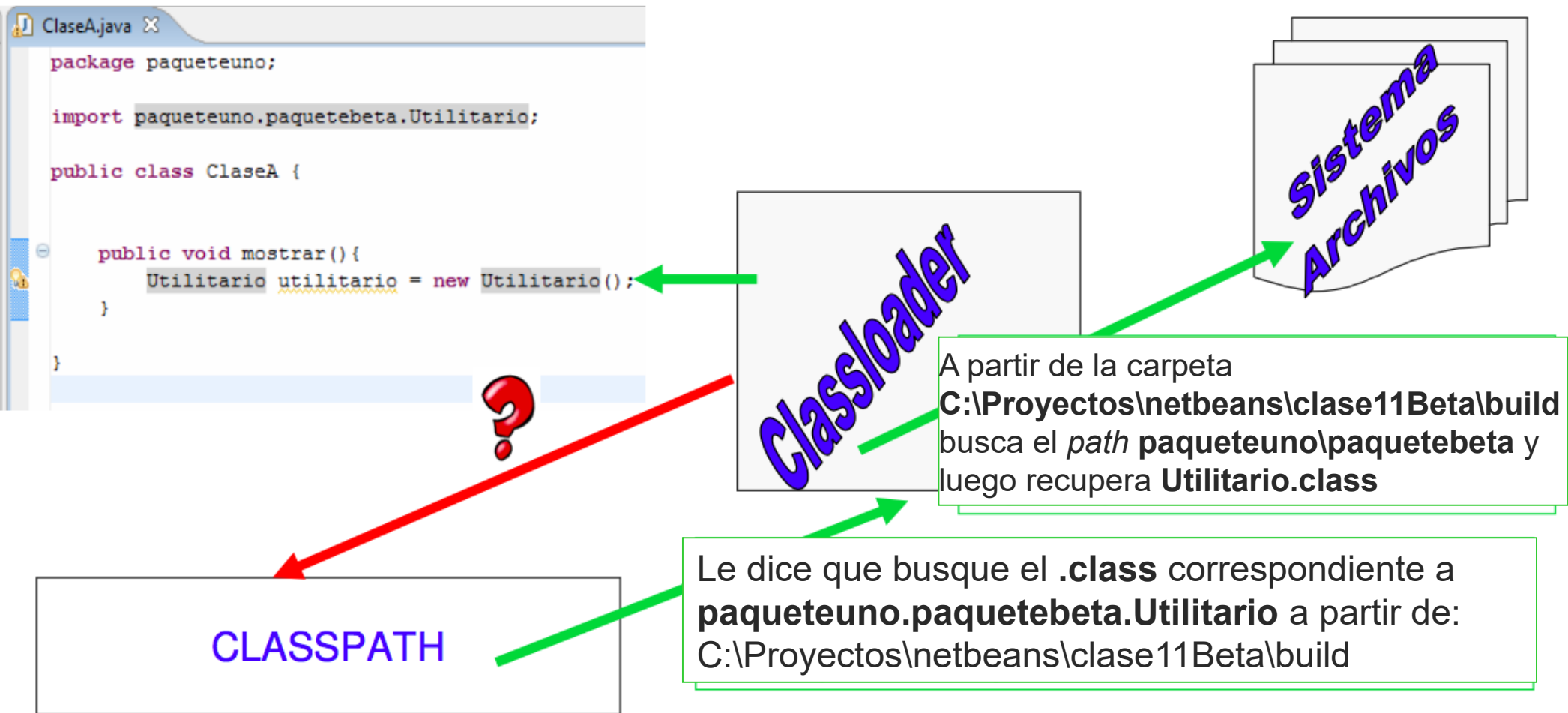
    public void mostrar(){
        Utilitario utilitario = new Utilitario();
    }

}
```



Para poder determinar de dónde tomar las clases, el Classloader inspecciona la variable de entorno **classpath** que tiene configuradas las carpetas a partir de dónde recuperar las clases solicitadas

# ClassLoader - Classpath



# Classpath - jar

## ¿Qué es el Classpath?

La variable de entorno **CLASSPATH** determina dónde buscar tanto los **.class** o librerías de Java (el API de Java) como otros **.class** programados por el usuario.

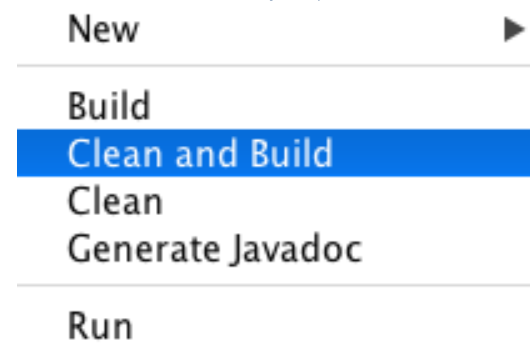
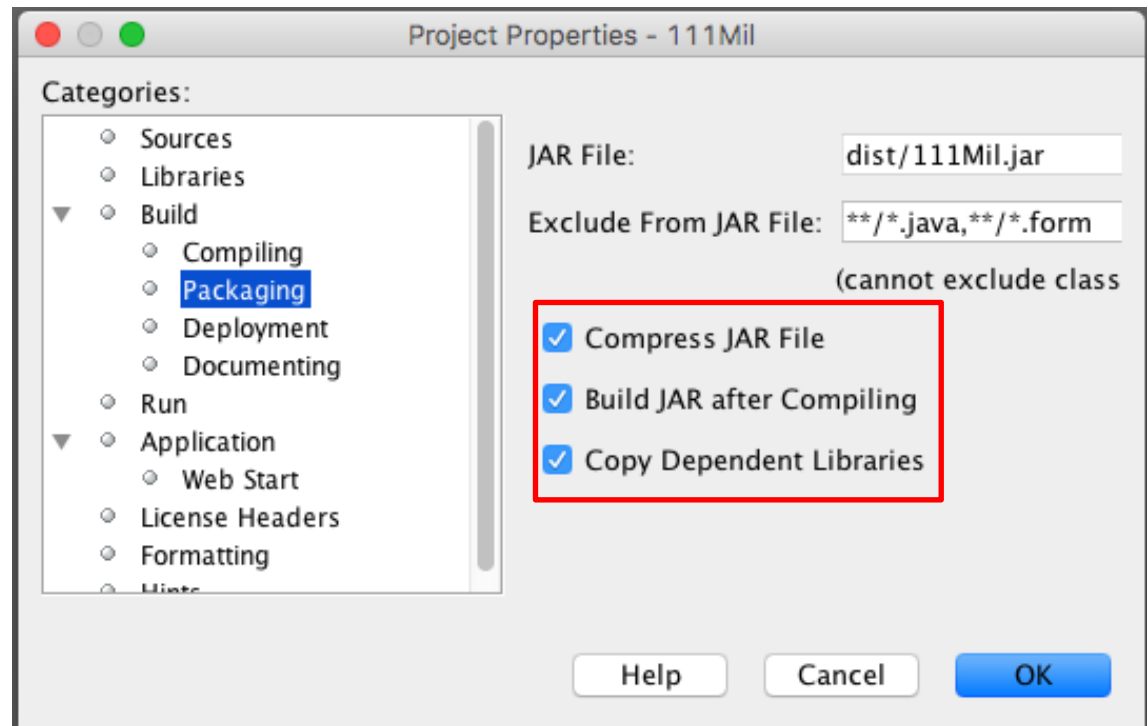
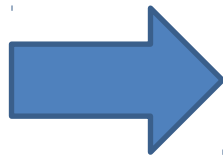
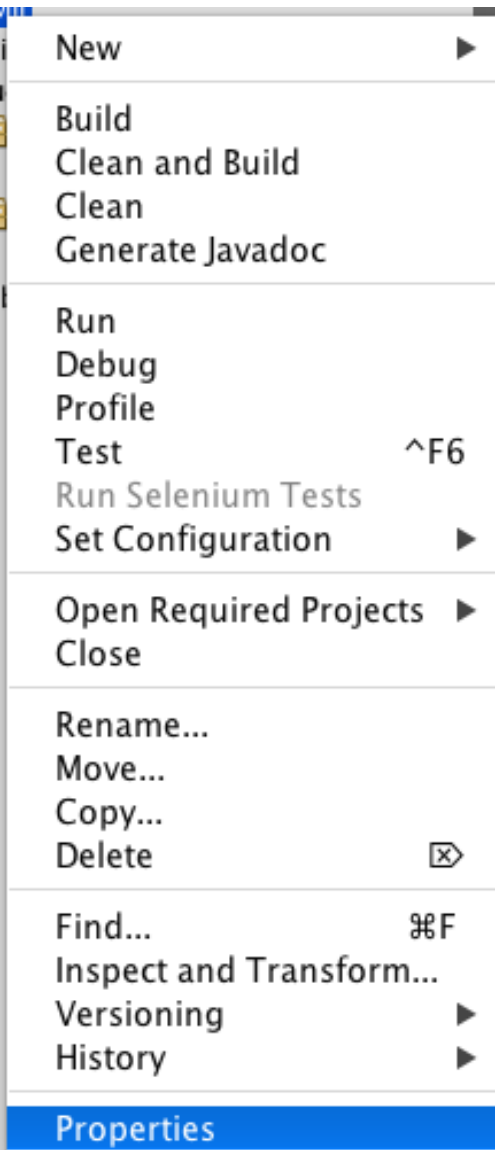
La variable **CLASSPATH** puede incluir la ruta de directorios o archivos \*.zip o \*.jar en los que se encuentren los archivo \*.class. Por ejemplo:

```
set CLASSPATH=c:\proyectos\libreriaMonedas.jar;  
c:\proyectos\motores.jar;c:\proyectos\;
```

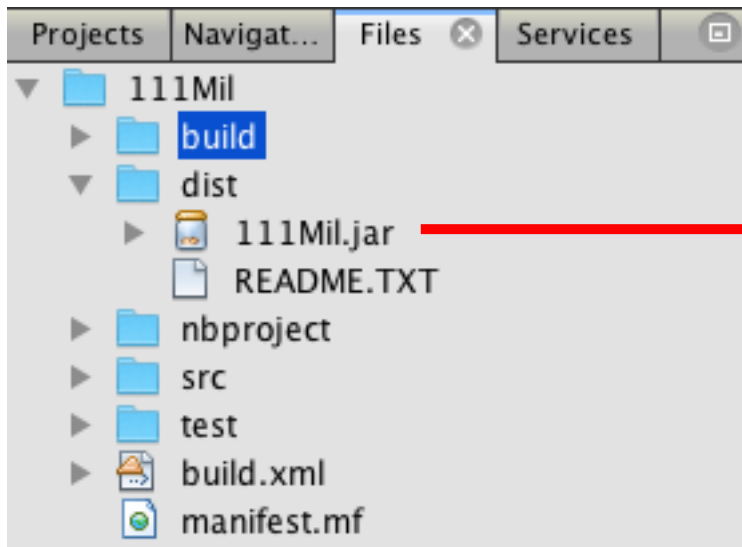
*Todas las clases compiladas que se encuentran adentro de libreriaMonedas.jar y motores.jar, más las clases de la carpeta proyectos **estarán accesibles!!**.*

Si bien es posible que al crear las clases, la podríamos dejar desempaquetadas e indicar dónde ubicarlas, lo más razonable sería ponerlas todas juntas en un paquete, y para esto Java nos provee una forma estándar de empaquetar, que es el formato JAR.

# Creación de un jar



# Creación de un jar

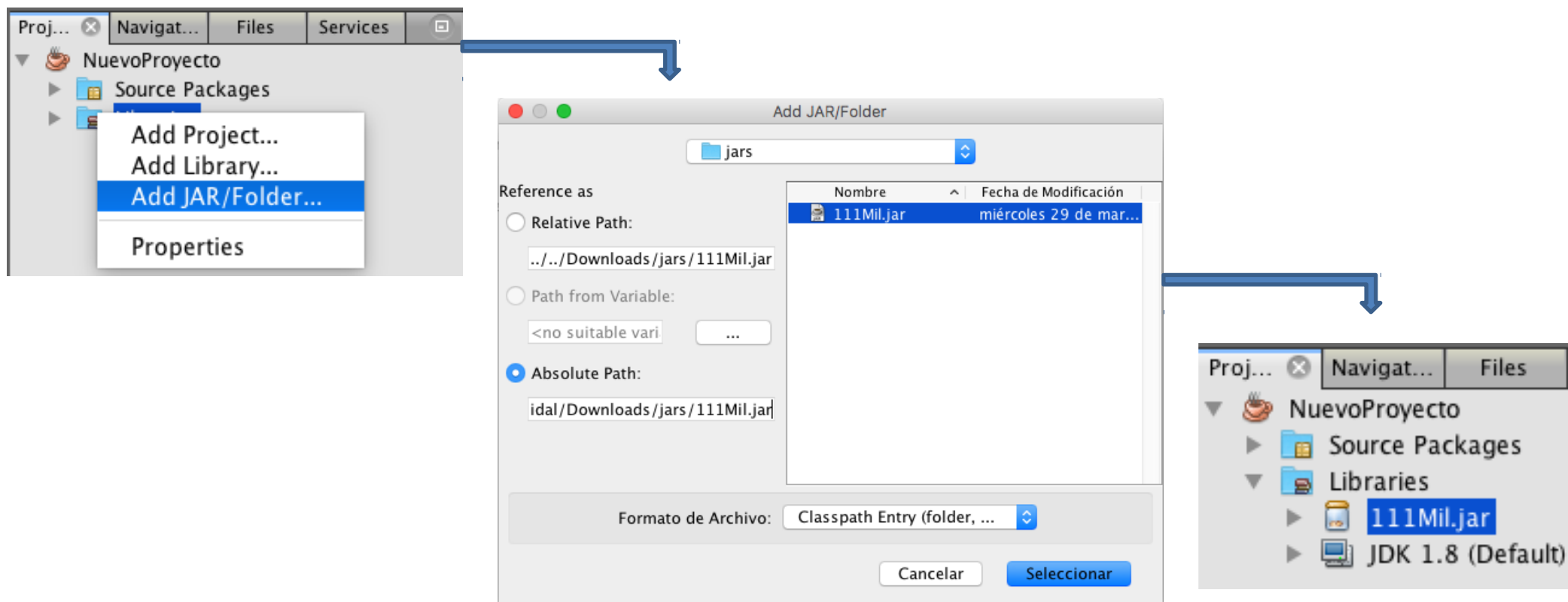


El archivo .jar se genera en un folder **dist** dentro del proyecto

# Uso de un jar

Veamos ahora como tenemos que hacer para utilizar el archivo **sistema.jar** que hemos creado.

- Durante la creación de un proyecto, aparecerá la ventana “1”
- Después de creado, se puede seleccionar el menú contextual del proyecto -> **properties-> Java Build Path.**

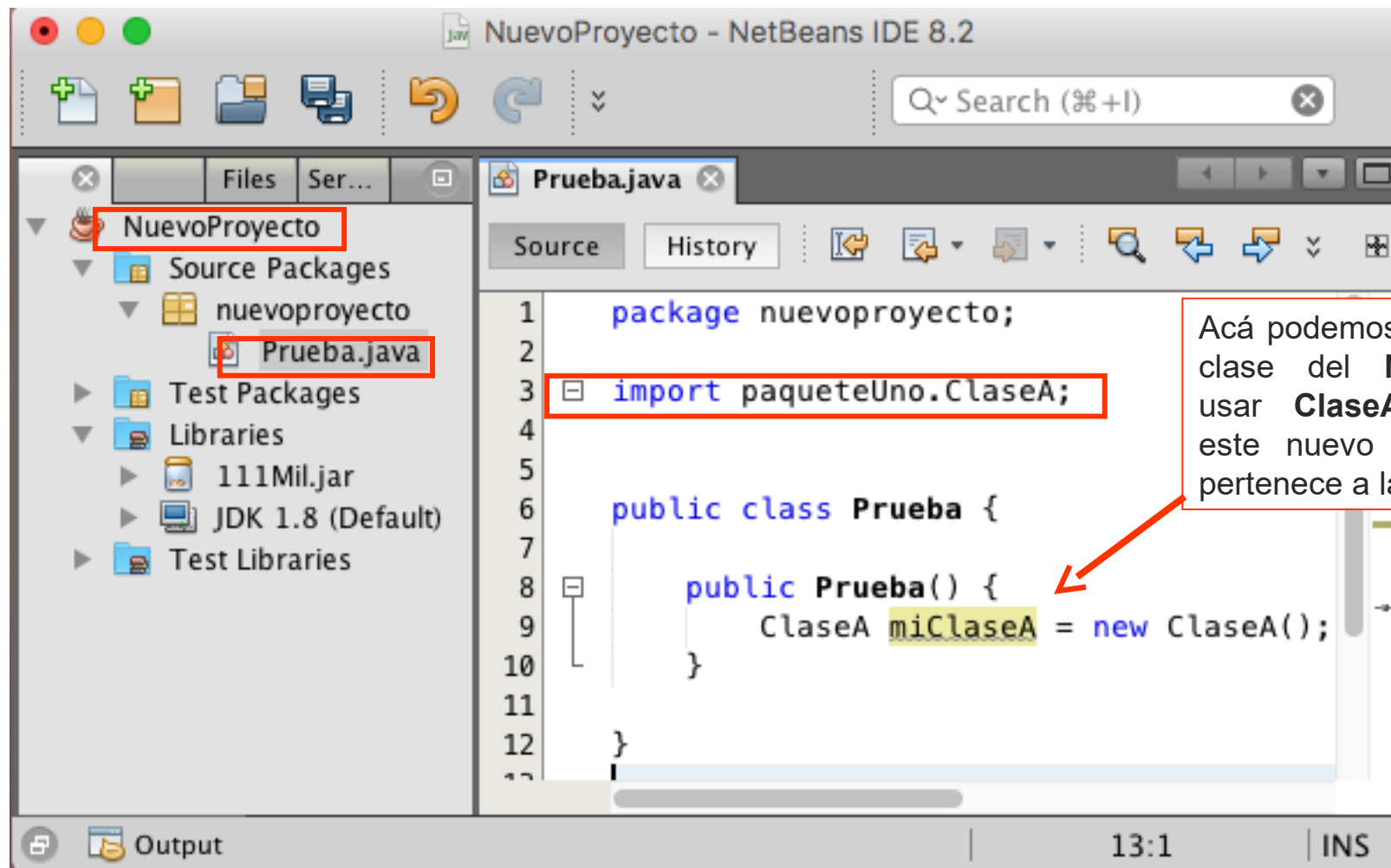


The screenshot illustrates the process of adding a custom JAR file to an Eclipse project:

- Project Explorer:** A right-click context menu is shown for the project 'NuevoProyecto'. The option 'Add JAR/Folder...' is selected.
- Add JAR/Folder Dialog:** The dialog box shows the file '111Mil.jar' selected in the file list. The 'Absolute Path' radio button is chosen, and the path 'idal/Downloads/jars/111Mil.jar' is entered. The 'Formato de Archivo' is set to 'Classpath Entry (folder, ...)'. The 'Seleccionar' button is clicked.
- Project Explorer:** The '111Mil.jar' file is now listed under the 'Libraries' folder of the project.

# Uso de un jar

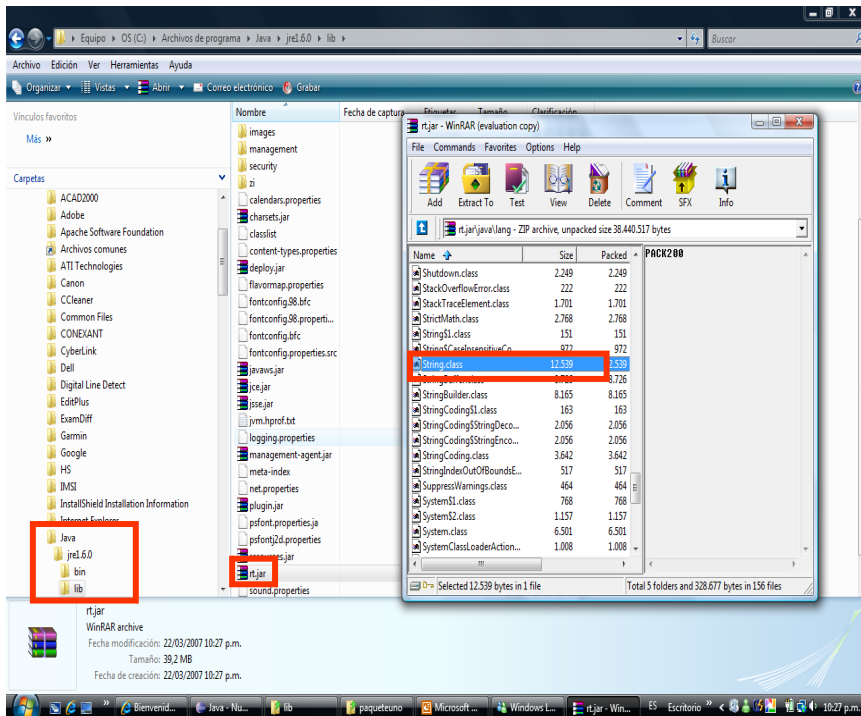
Sólo incorporando **sistema.jar**, disponemos de todas las clases públicas que contiene el mismo



Acá podemos observar cómo desde una clase del **NuevoProyecto**, podemos usar **ClaseA** que NO está definida en este nuevo proyecto y que tampoco pertenece a la API JAVA.

# Uso de un jar

De esta manera hemos incorporado un JAR que creamos nosotros mismos a un proyecto configurando el *classpath*. A su vez, existen muchas clases más que ya vienen definidas en la API Java y que están presente siempre y que están empaquetadas en el *rt.jar*. La configuración por defecto del *classpath* nos pone disponible esta librería de clases.



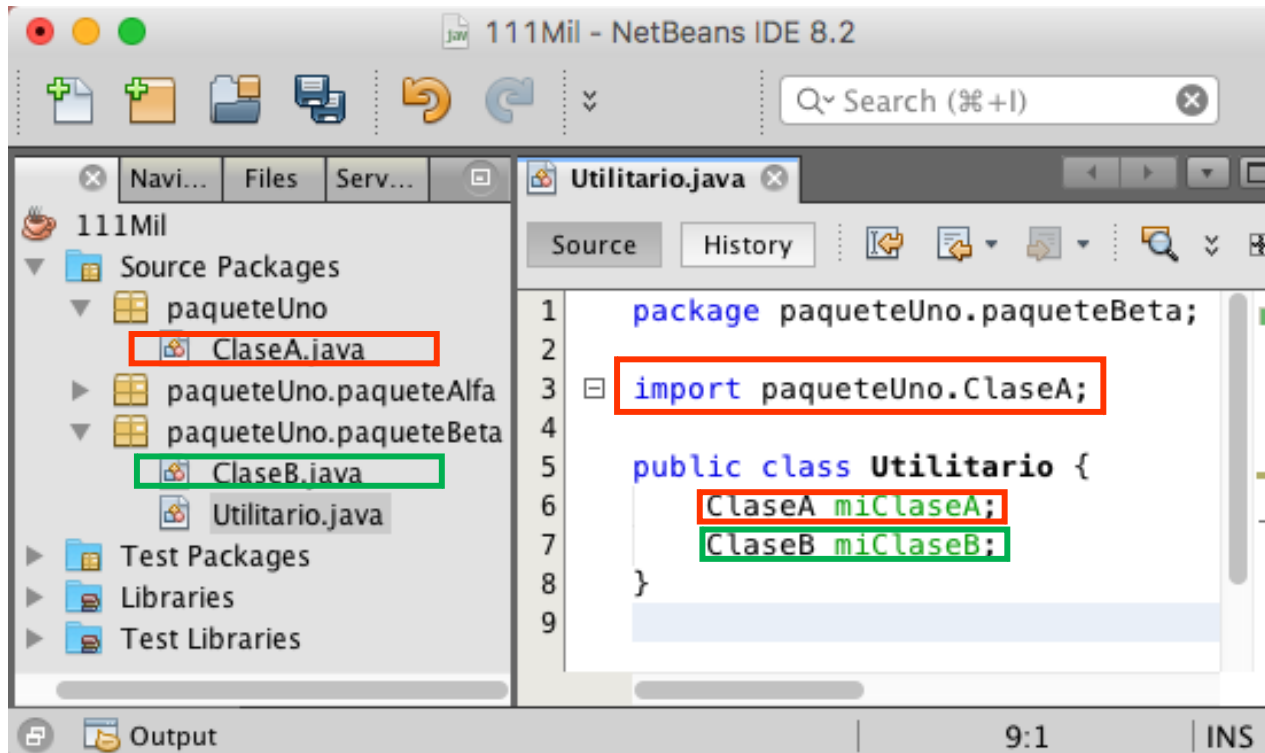




# Imports implícitos

- Todas aquellas clases que pertenecen al paquete **java.lang** no es necesario importarlas.
- Lo mismo ocurre cuando definimos clases en un paquete. Si una clase quiere utilizar otra clase que pertenece al mismo paquete no hace falta importar NADA.
- Los únicos paquetes que no es necesario importar son el paquete **java.lang** y el paquete sobre el que se está definiendo la clase o paquete "actual".

# Imports implícitos

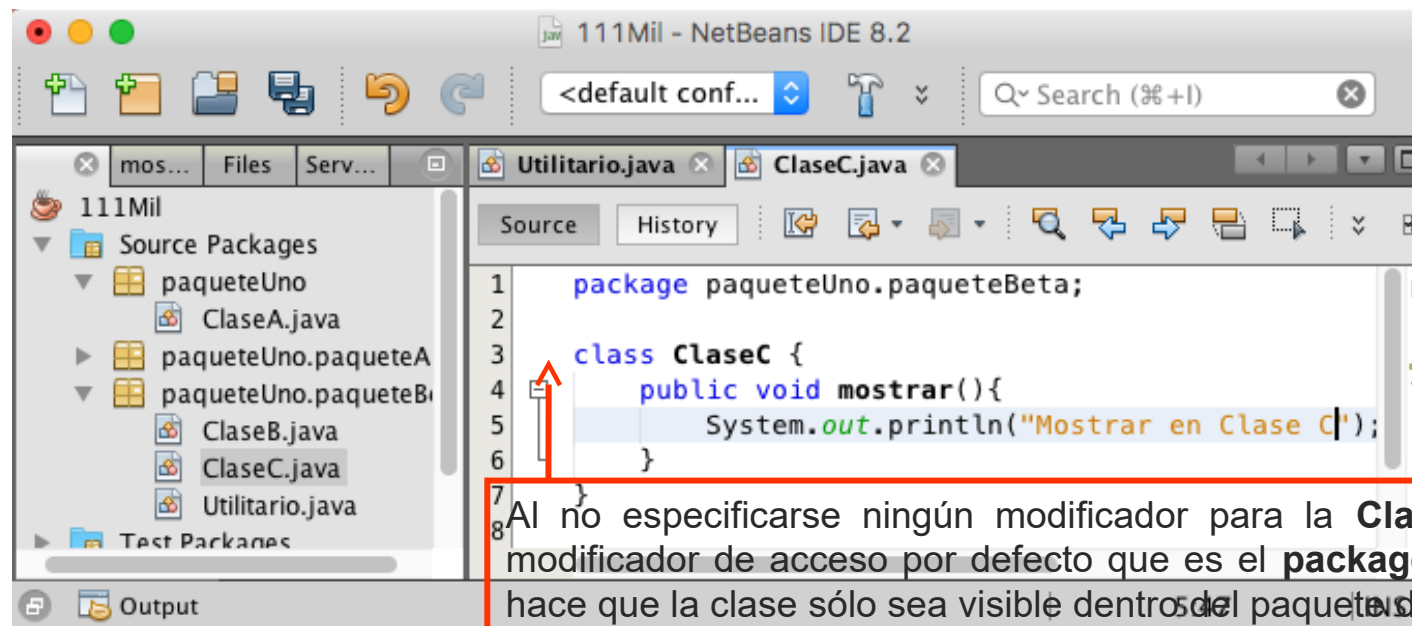


Observando el proyecto 111Mil, vemos que la clase Utilitario para usar la **ClaseA** de otro **paquete** debe importarla, mientras que para usar la **ClaseB** que está en el mismo **paquete** que Utilitario no es necesario.

# Modificadores de acceso

Los modificadores de acceso, nos determinan la **visibilidad** que tiene el elemento al cual se aplica. En este caso veremos dos modificadores de accesos aplicados a clases.

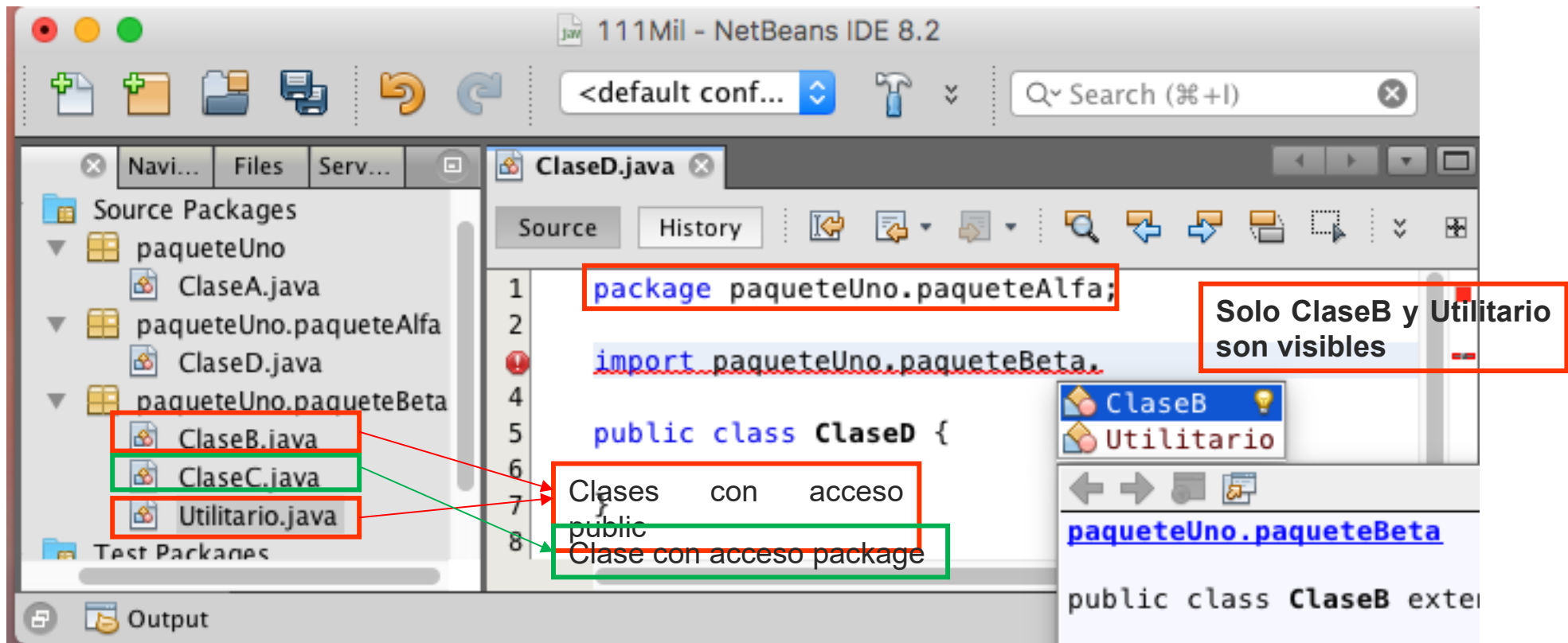
Modificador de acceso **public** y **package** (o defecto o predeterminado)



Las clases definidas sin modificador de acceso, adquieren el de defecto que es package. Estas clases son **DESCONOCIDAS** fuera del paquete donde se declararon.

# Modificadores de acceso

La **ClaseC** tiene modificador de acceso package, por lo tanto **NO** está **VISIBLE** fuera del paquete **paqueteuno.paqueteBeta** y NO es posible usarla en otro paquete. Esto es un ejemplo claro de **Ocultamiento de Información**, que es uno de los conceptos fundamentales de la Programación Orientada a Objetos.



111Mil - NetBeans IDE 8.2

<default conf... Search (⌘+I)

Source Packages

- paqueteUno
  - ClaseA.java
  - paqueteUno.paqueteAlfa
    - ClaseD.java
  - paqueteUno.paqueteBeta
    - ClaseB.java
    - ClaseC.java
    - Utilitario.java
- Test Packages

Source History

```
1 package paqueteUno.paqueteAlfa;  
2  
3 import paqueteUno.paqueteBeta;  
4  
5 public class ClaseD {  
6  
7  
8
```

Solo ClaseB y Utilitario son visibles

ClaseB  
Utilitario

paqueteUno.paqueteBeta

public class ClaseB exte

Clases con acceso public  
Clase con acceso package