Java y Herramientas de Desarrollo

Sesión 1: Lenguaje Java y Entorno de Desarrollo

Puntos a tratar

- Introducción a Java
- Entorno de desarrollo Eclipse
- Organización de las clases en paquetes
- Elementos de una clase Java
- Herencia e interfaces

Java

- *Java* es un lenguaje OO creado por *Sun Microsystems* para poder funcionar en distintos tipos de procesadores y máquinas.
- Similar a C o C++, pero con algunas características propias (gestión de hilos, ejecución remota, etc)
- Independiente de la plataforma, gracias a la JVM (*Java Virtual Machine*), que interpreta los ficheros objeto
- Se dispone de antemano de la API (*Application Programming Interface*) de clases de Java.

Ficheros JAR

- Permiten empaquetar varias clases en un solo fichero comprimido, parecido a un fichero TAR o ZIP.
- Se crean con la herramienta *jar* de Java
- Ventajas
  - *Seguridad*: mediante firmas digitales
  - *Descarga*: mejor descargar un archivo que varios
  - *Versiones*: podemos incluir información de la versión
  - *Portabilidad*: al ser un estándar de la plataforma Java

Variable CLASSPATH

- La variable CLASSPATH contiene los directorios y ficheros JAR con las clases externas a la API necesarias para compilar y/o ejecutar el programa

```
set CLASSPATH=%CLASSPATH%;C:\miDirectorio;.  (Windows)
export CLASSPATH=$CLASSPATH:/miDirectorio;.  (Linux)

set CLASSPATH=%CLASSPATH%;ruta\fichero.jar   (Windows)
export CLASSPATH=$CLASSPATH:ruta/fichero.jar (Linux)
```

- Se debe proporcionar la ruta hasta el directorio donde comiencen los paquetes (no los directorios de paquetes también)
- Se debe incluir el directorio actual '.' en el CLASSPATH

Compilar y ejecutar un programa

- Para *compilar un programa* utilizamos el comando *javac* y el nombre del fichero fuente:

```
javac NombreFichero.java
```

- Para *ejecutar un programa* utilizamos el comando *java* y el nombre de la clase. Podemos pasarle parámetros a continuación

```
java NombreFichero
java NombreFichero param1 param2
```

Compilar y ejecutar un programa

- Para *ejecutar un JAR ejecutable* utilizamos el comando *java* con el parámetro *–jar* y el nombre del fichero JAR

```
java -jar fichero.jar
```

- Al compilar una clase, se compilan automáticamente las que necesiten compilarse
- Las clases se compilan en ficheros con extensión *.class*
- Hay que asegurarse de que el CLASSPATH está bien definido antes de compilar o ejecutar
- Es importante respetar las mayúsculas y minúsculas

Extensiones

- Desde Java 1.2 se pueden añadir nuevas funcionalidades al núcleo de Java en forma de *extensiones*
- Son grupos de paquetes y clases que añaden nuevas funcionalidades
- Las extensiones son accesibles sin necesidad de incluirlas en el CLASSPATH
- Para crear una extensión, empaquetamos las clases y paquetes en un fichero JAR
- Después se copia el JAR en el directorio ***{java.home}/jre/lib/ext***

Algunas extensiones existentes

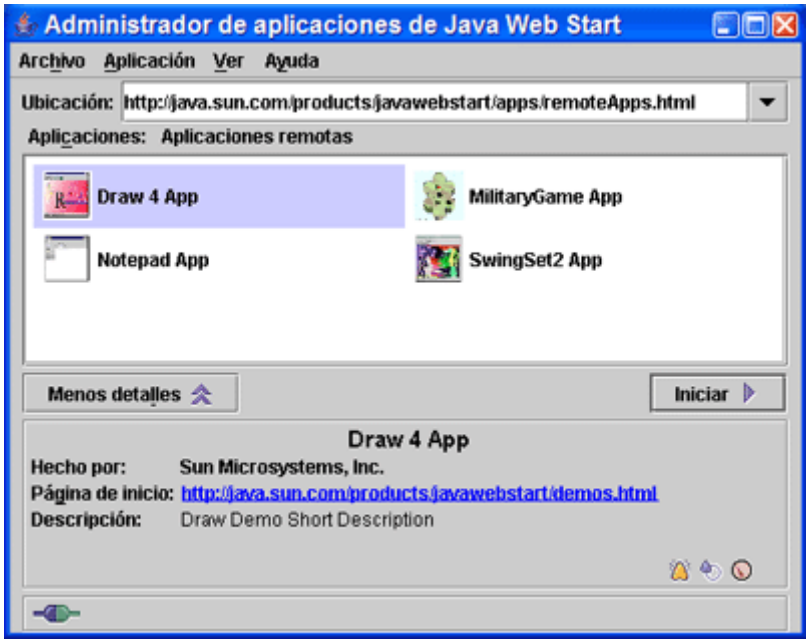
- *Java 3D*: para incluir mundos 3D en las aplicaciones
- *Java Media Framework*: para incorporar elementos multimedia(audio/video)
- *Java Advanced Imaging*: librería para procesamiento de imágenes
- *JavaHelp*: para incorporar ayuda en línea en nuestros programas
- *JavaMail*: para construir nuestro propio sistema de correo y mensajería

Búsqueda de clases en Java

- Para buscar las clases necesarias al compilar o ejecutar, Java sigue este orden:
  - Clases principales (*bootstrap*) de Java (API)
  - Extensiones instaladas
  - CLASSPATH

Otras herramientas

- *Java Web Start*: para ejecutar aplicaciones desde internet, pinchando en su enlace simplemente



- Otros: *javadoc, appletviewer, rmiregistry, rmid*

Eclipse

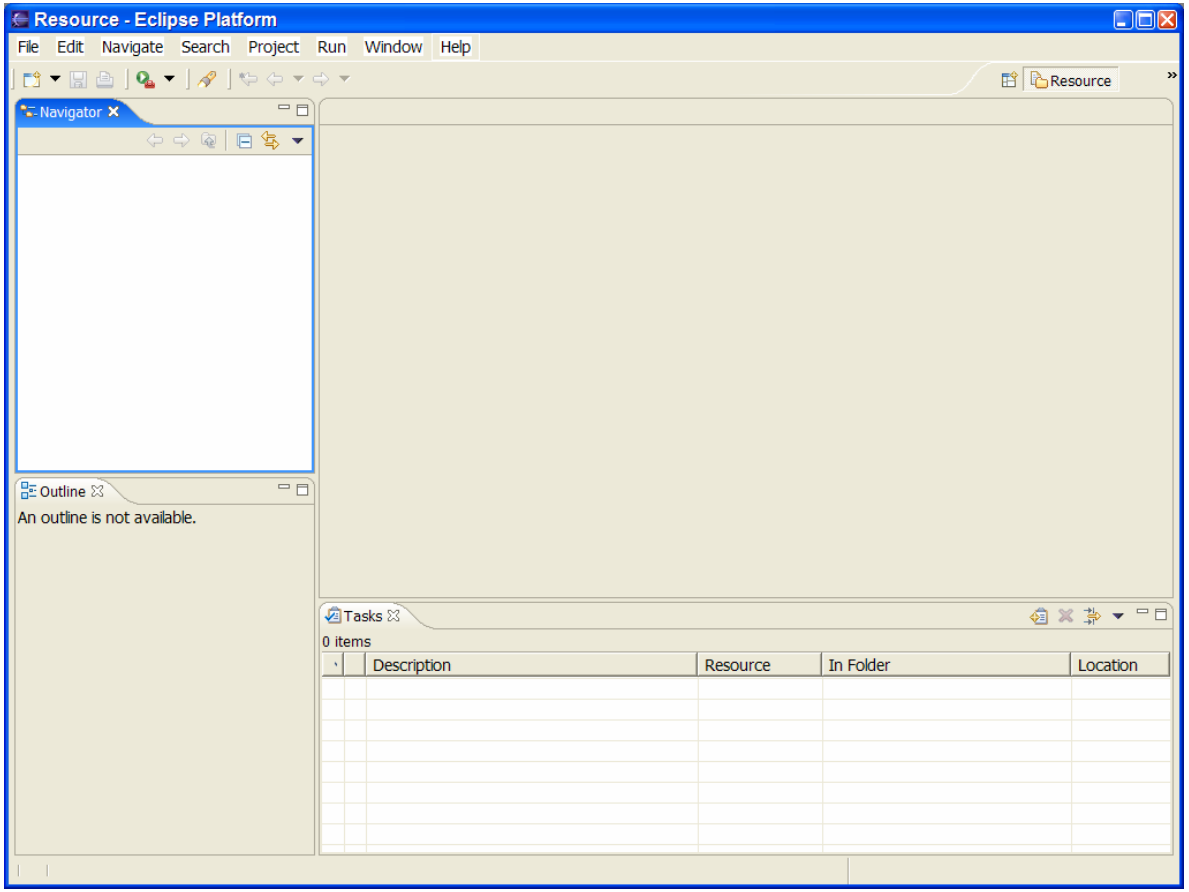
- Eclipse es una herramienta desarrollada por IBM que integra diferentes tipos de aplicaciones
- Su principal aplicación es el JDT (*Java Development Tooling*)
- Se pueden añadir nuevas funcionalidades mediante *plugins* fácilmente instalables
- Los *recursos* gestionados en Eclipse son igualmente visibles y actualizados por todas sus sub-aplicaciones

Instalación

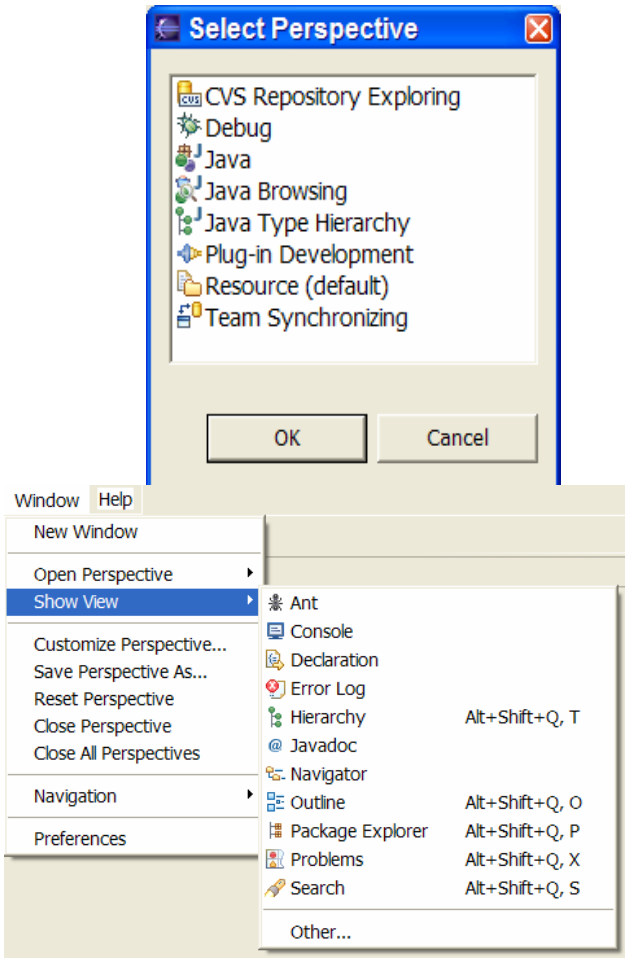
- Para instalar Eclipse se requiere:
  - Windows, Linux, Solaris, QNX o Mac OS/X con 256 MB de RAM
  - JDK o JRE 1.3 o superior (1.4.1 recomendado)
  - El fichero ZIP con los archivos de Eclipse para instalar
- La instalación se compone de los pasos:
  - Instalar JDK o JRE
  - Descomprimir el ZIP de Eclipse en el lugar deseado

Ejecución

- Para ejecutar Eclipse, se tiene un ejecutable *eclipse.exe* o *eclipse.sh*



Perspectivas, vistas y editores

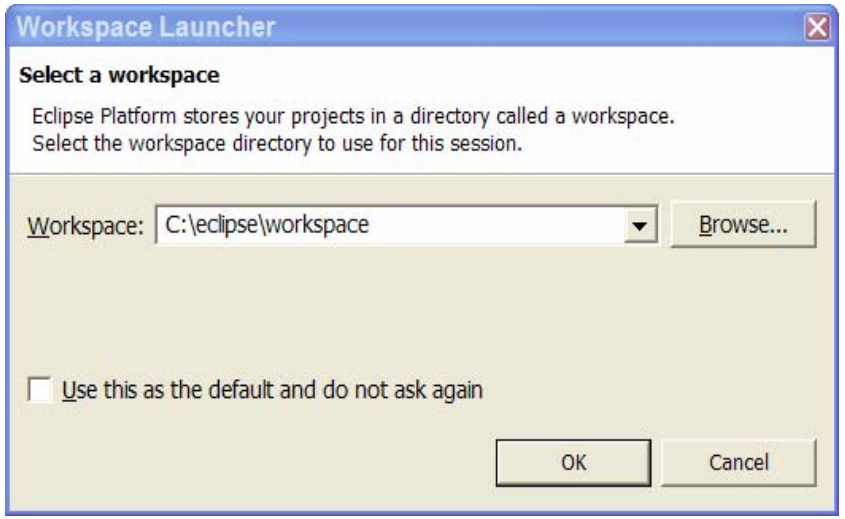


- Una *perspectiva* es un conjunto de *vistas* y *editores*
- Podemos elegir diferentes tipos de perspectivas (como la perspectiva Java, la más usual), o la de depuración, desde el menú *Window* (*Open Perspective*)
- Dentro de una perspectiva, podemos elegir qué vistas o editores queremos tener presentes, todo desde el menú *Window* (*Show View*)

Espacio de trabajo

- Por defecto el espacio de trabajo es *ECLIPSE_HOME/workspace*. Podemos elegir uno diferente al lanzar el programa:

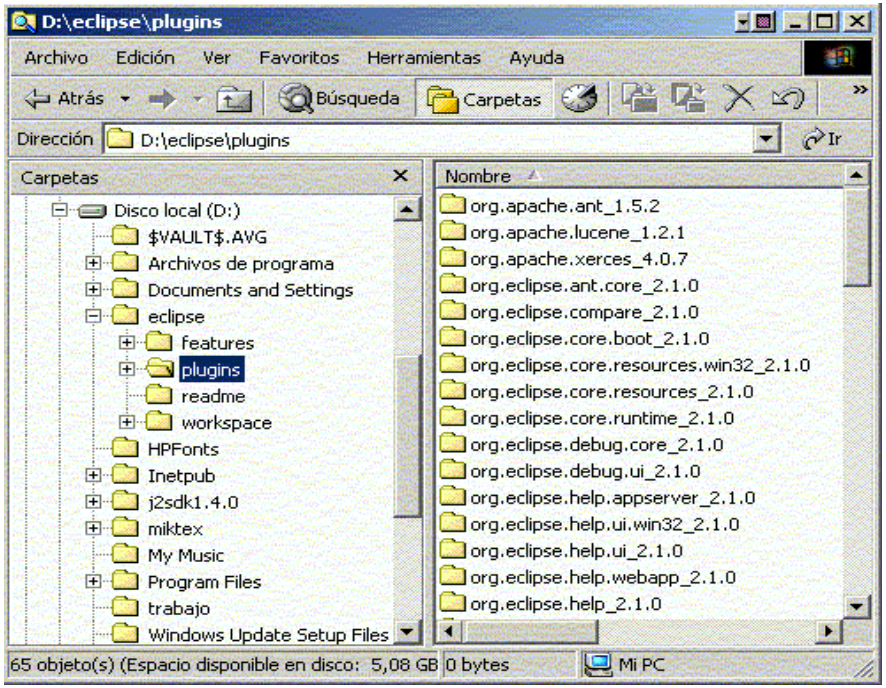
```
eclipse –data c:\misTrabajos
```



- También podemos crear proyectos y trabajos fuera del espacio de trabajo

Plugins

- Para instalar nuevos plugins se copian en la carpeta *ECLIPSE_HOME/plugins*
- Después hay que reiniciar Eclipse para tomar los cambios

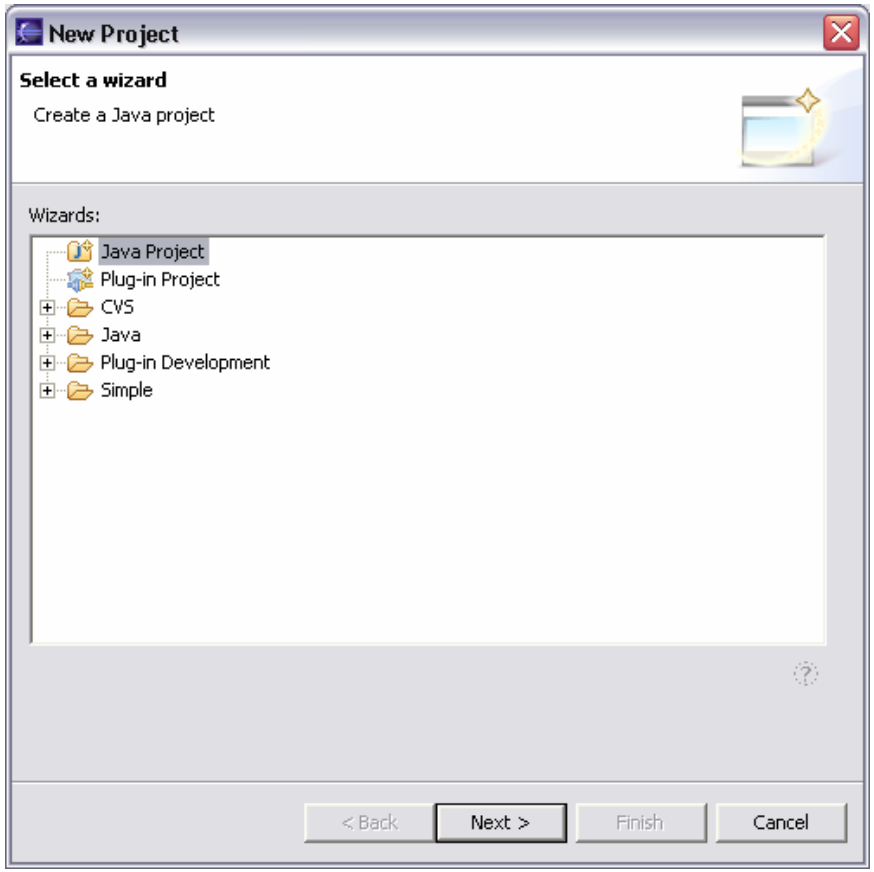


Algunos plugins conocidos

- *EclipseUML*: para realizar diseños UML (diagramas de clases, de paquetes, etc)
- *Lomboz*: para desarrollo de aplicaciones J2EE (servlets, JSP, EJBs, servicios Web, etc)
- *V4ALL*: para desarrollo de aplicaciones gráficas
- Otros plugins: para gestión de WebDAV, desarrollo con SWT, etc.

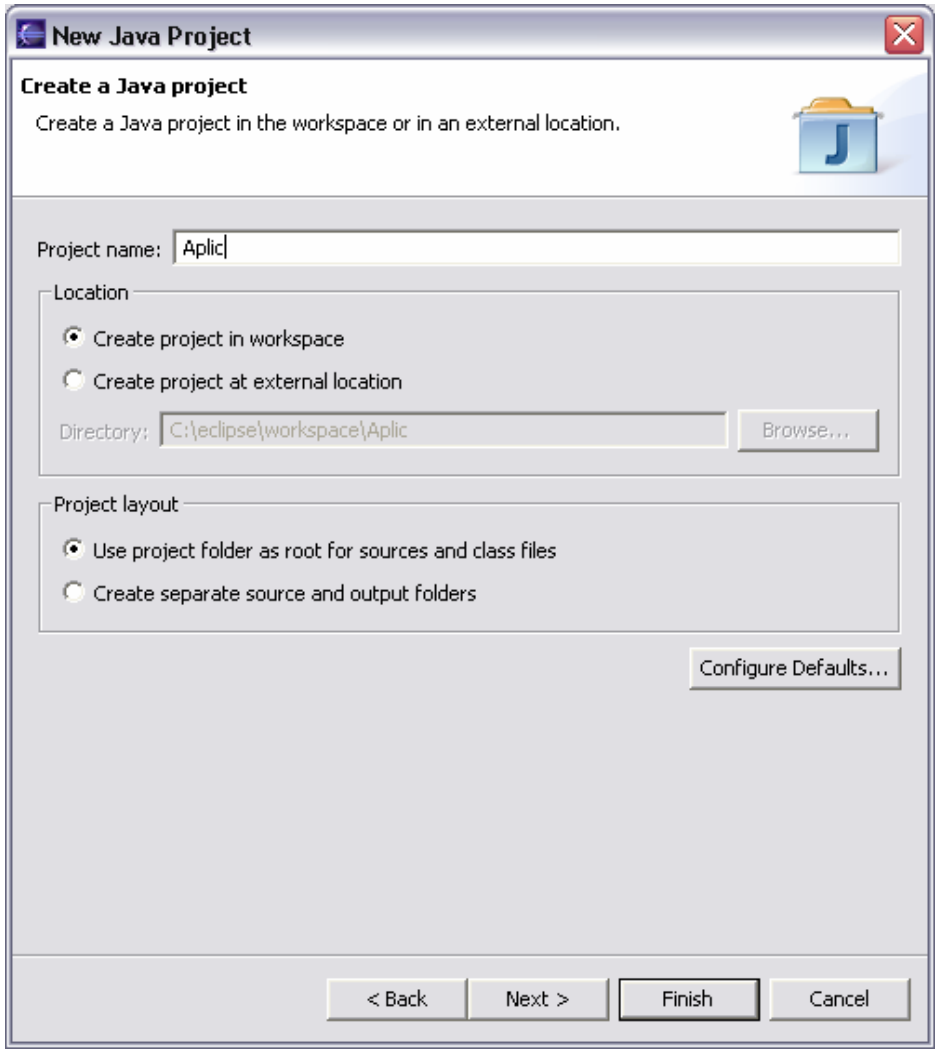
Nuevo proyecto

- Se crean desde *File – New – Project*, eligiendo luego el tipo de proyecto (*Java Project*, normalmente)



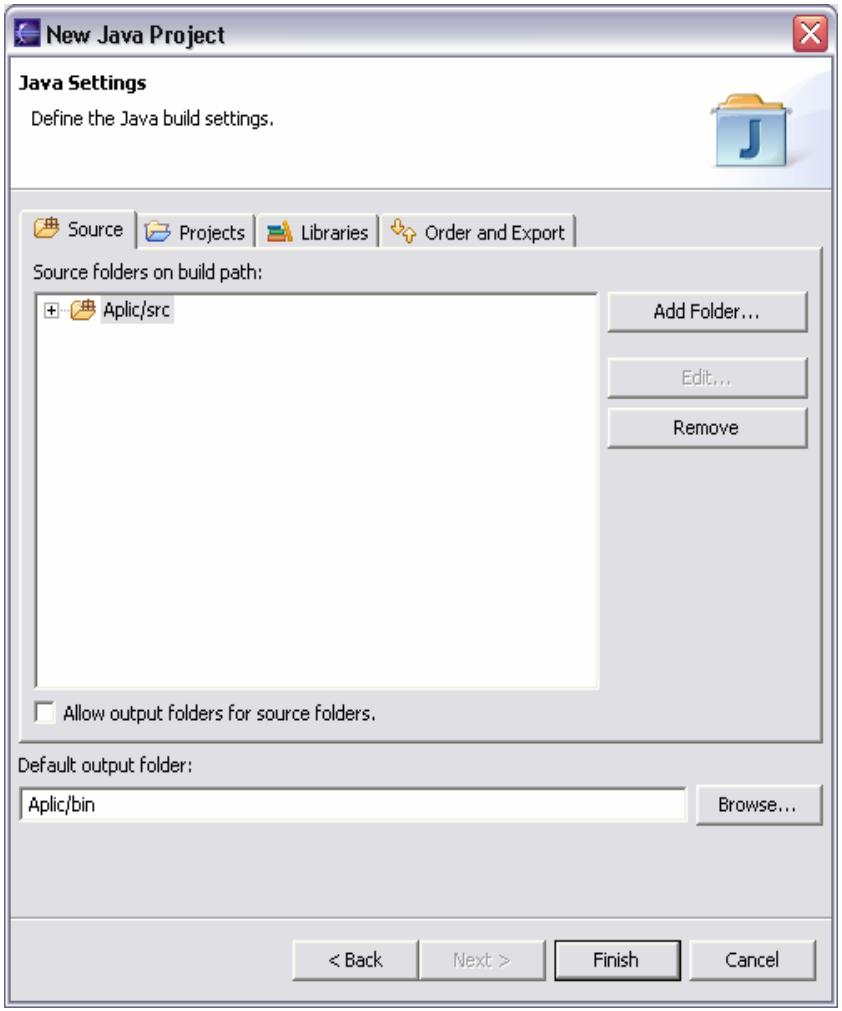
Nuevo proyecto

- Después se nos pregunta dónde está el proyecto, o dónde guardarlo si es nuevo, y con qué nombre

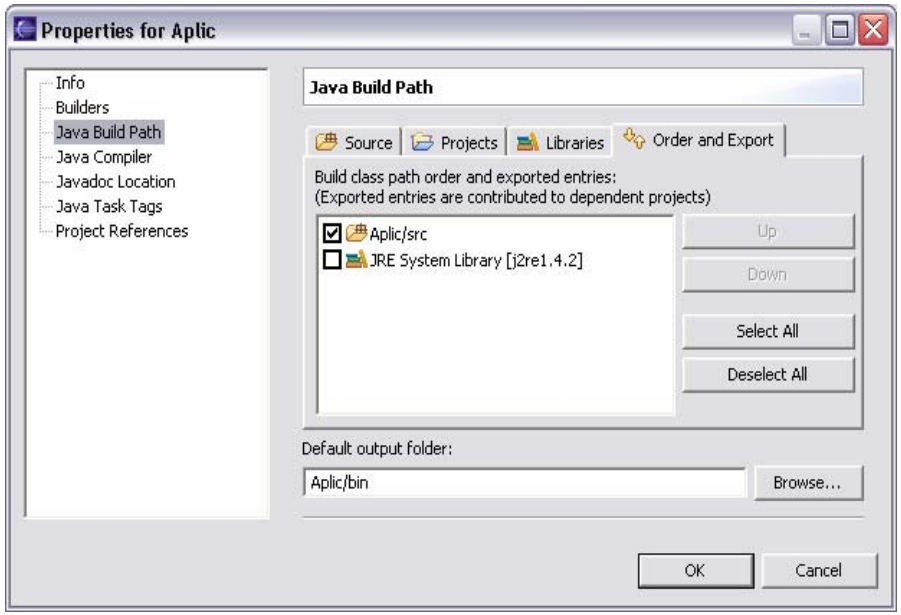


Nuevo proyecto

- En la última pantalla indicamos
  - Qué carpetas tienen el código (*Source*)
  - El directorio donde sacar las clases compiladas *(Default Output Folder*)
  - El CLASSPATH (*Libraries*)
  - etc



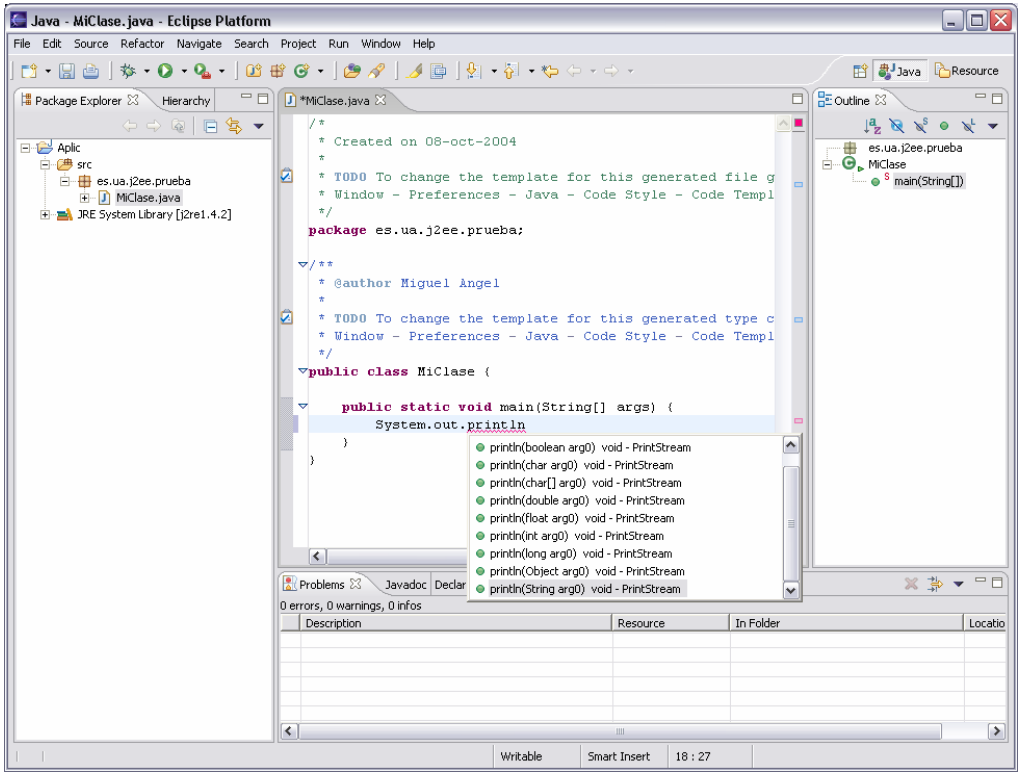
Buildpath de un proyecto



- Pulsando el botón derecho sobre el proyecto y yendo a *Properties* accedemos a su *Java Build Path*
- En él se establecen las clases a compilar, recursos (ficheros JAR, directorios, etc) que debe tener en cuenta, etc.

Editor de código

- Dispone de realce de sintaxis, y ayuda contextual que permite autocompletar sentencias de código



Paquetes

- *Paquetes*: organizan las clases en una jerarquía de paquetes y subpaquetes
- Para indicar que una clase pertenece a un paquete o subpaquete se utiliza la palabra *package* al principio de la clase

```
package paquete1.subpaquete1;
class MiClase {
```

- Para utilizar clases de un paquete en otro, se colocan al principio sentencias *import* con los paquetes necesarios:

```
package otropaquete;
import paquete1.subpaquete1.MiClase;
import java.util.*;
class MiOtraClase {
```

Paquetes

- Si no utilizamos sentencias *import*, deberemos escribir el nombre completo de cada clase del paquete no importado (incluyendo subpaquetes)

```
class MiOtraClase {
   paquete1.subpaquete1.MiClase a = ...;     // Sin import
   MiClase a = ...;                          // Con import
```

- Los paquetes se estructuran en directorios en el disco duro, siguiendo la misma jerarquía de paquetes y subpaquetes

```
./paquete1/subpaquete1/MiClase.java
```

Paquetes

- Siempre se deben incluir las clases creadas en un paquete
  - Si no se especifica un nombre de paquete la clase pertenecerá a un paquete “sin nombre”
  - No podemos importar clases de paquetes “sin nombre”, las clases creadas de esta forma no serán accesibles desde otros paquetes
  - Sólo utilizaremos paquetes “sin nombre” para hacer una prueba rápida, nunca en otro caso

Convenciones de paquetes

- El nombre de un paquete deberá constar de una serie de palabras simples siempre en minúsculas
  - Se recomienda usar el nombre de nuestra DNS al revés
    `jtech.ua.es → es.ua.jtech.prueba`
- Colocar las clases interdependientes, o que suelan usarse juntas, en un mismo paquete
- Separar clases volátiles y estables en paquetes diferentes
- Hacer que un paquete sólo dependa de paquetes más estables que él
- Si creamos una nueva versión de un paquete, daremos el mismo nombre a la nueva versión sólo si es compatible con la anterior

Clases

- *Clases*: con la palabra *class* y el nombre de la clase

```
class MiClase
{
   ...
}
```

- Como nombre utilizaremos un sustantivo
- Puede estar formado por varias palabras
- Cada palabra comenzará con mayúscula, el resto se dejará en minúscula
  - Por ejemplo: `DataInputStream`
- Si la clase contiene un conjunto de métodos estáticos o constantes relacionadas pondremos el nombre en plural
  - Por ejemplo: `Resources`

Campos y variables

- *Campos y variables*: simples o complejos
- Utilizaremos sustantivos como nombres

```
Properties propiedades;
File ficheroEntrada;
int numVidas;
```

- Puede estar formado por varias palabras, con la primera en minúsculas y el resto comenzando por mayúsculas y el resto en minúsculas
  - Por ejemplo: `numVidas`
- En caso de tratarse de una colección de elementos, utilizaremos plural
  - Por ejemplo: `clientes`
- Para variables temporales podemos utilizar nombres cortos, como las iniciales de la clase a la que pertenezca, o un carácter correspondiente al tipo de dato

```
int i;
Vector v;
DataInputStream dis;
```

Constantes

- *Constantes*: Se declarán como *final* y *static*

```
final static String TITULO_MENU = “Menu”;
final static int ANCHO_VENTANA = 640;
final static double PI = 3.1416;
```

- El nombre puede contener varias palabras
- Las palabras se separan con ‘_’
- Todo el nombre estará en mayúsculas
  - Por ejemplo: `MAX_MENSAJES`

Métodos

- *Métodos*: con el tipo devuelto, nombre y parámetros

```
void imprimir(String mensaje)
{
   ...// Codigo del método
}
Vector insertarVector(Object elemento, int posicion)
{
   ...// Codigo del método
}
```

- Los nombres de los métodos serán verbo
- Puede estar formados por varias palabras, con la primera en minúsculas y el resto comenzando por mayúsculas y el resto en minúsculas
  - Por ejemplo: `imprimirDatos`

Constructores

- *Constructores*: se llaman igual que la clase, y se ejecutan con el operador *new* para reservar memoria

```
MiClase()
{
   ...// Codigo del constructor
}
MiClase(int valorA, Vector valorV)
{
   ...// Codigo del otro constructor
}
```

- No hace falta destructor, de eso se encarga el *garbage collector*

Modificadores de acceso

- Las clases y sus elementos admiten unos modificadores de acceso:
  - *privado*: el elemento es accesible sólo desde la clase en que se encuentra
  - *protegido*: el elemento es accesible desde la propia clase, desde sus subclases, y desde clases del mismo paquete
  - *público*: el elemento es accesible desde cualquier clase
  - *paquete*: el elemento es accesible desde la propia clase, o desde clases del mismo paquete.

Modificadores de acceso

- *private* se utiliza para elementos PRIVADOS
- *protected* se utiliza para elementos PROTEGIDOS
- *public* se utiliza para elementos PUBLICOS
- No se especifica nada para elementos PAQUETE

```
public class MiClase {
   private int n;
   protected void metodo() { ... }
```

- Todo fichero Java debe tener una y solo una clase pública, llamada igual que el fichero (más otras clases internas que pueda tener)

Otros modificadores

- *abstract*: para definir clases y métodos abstractos
- *static*: para definir elementos compartidos por todos los objetos que se creen de la misma clase
  - NOTA: dentro de un método estático sólo podemos utilizar elementos estáticos, o elementos que hayamos creado dentro del propio método
- *final*: para definir elementos no modificables ni heredables
- *synchronized*: para elementos a los que no se puede acceder al mismo tiempo desde distintos hilos de ejecución

```
public abstract class MiClase {
   public static final int n = 20;
   public abstract void metodo();
   ...
```

Getters y Setters

- Es buena práctica de programación declarar todos los campos de las clases privados

```
public class MiClase {
   private int login;
   private boolean administrador;
}
```

- Para acceder a ellos utilizaremos métodos
  - *Getters* para obtener el valor del campo
  - *Setters* para modificar el valor del campo
- Estos métodos tendrán prefijo *get* y *set* respectivamente, seguido del nombre del campo al que acceden, pero comenzando por mayúscula
  - Por ejemplo: `getLogin()`, `setLogin(String login)`
- El *getter* para campos booleanos tendrá prefijo *is* en lugar de *get*
  - Por ejemplo: `isAdministrador()`

Ejecución de clases: método *main(...)*

- Las clases que queramos ejecutar en una aplicación deben tener un método *main(...)* con la siguiente estructura:

```
public static void main(String[] args)
{
   ... // Codigo del método
}
```

Ejemplo completo

```
package paquete1.subpaquete1;
import otropaquete.MiOtraClase;
import java.util.Vector;
public class MiClase {
   Vector v;
   MiOtraClase mc;

   public MiClase() {
        ... // Codigo del constructor
   }

   void imprimir(String mensaje) {
        ... // Codigo del método
   }
}
```

Ejemplo completo

- Podemos utilizar una instancia de la clase en otra clase, y utilizar sus campos o métodos:

```
import paquete1.subpaquete1.*;
public class OtraClase {
   void metodo() {
        MiClase mc = new MiClase();
        mc.imprimir(“Hola”);
   }
}
```

Convenciones de formato

- Indentar el código uniformemente
- Limitar la anchura de las líneas de código (para impresión)
- Utilizar líneas en blanco para separar bloques de código
- Utilizar espacios para separar ciertos elementos en una línea

Convenciones de documentación

- Escribir documentación de uso y mantenimiento
- Eliminar palabras innecesarias
- Utilizar javadoc para describir la interfaz de programación
- Documentar precondiciones, postcondiciones y condiciones invariantes
- Utilizar `/* . . . */` para esconder código sin borrarlo
- Utilizar `// . . .` para detalles de la implementación

Clases abstractas e interfaces

- Una *clase abstracta* es una clase que deja algunos métodos sin código, para que los rellenen las subclases que hereden de ella

```
public abstract class MiClase {
   public abstract void metodo1();
   public void metodo2() {
        ...
   }
}
```

- Un *interfaz* es un elemento que sólo define la cabecera de sus métodos, para que las clases que implementen dicha interfaz rellenen el código según sus necesidades.

```
public interface Runnable {
   public void run();
}
```

- Asignaremos un nombre a los interfaces de forma similar a las clases, pudiendo ser en este caso adjetivos o sustantivos.

Herencia e interfaces

- Herencia
  - Definimos una clase a partir de otra que ya existe
  - Utilizamos la palabra *extends* para decir que una clase hereda de otra (Pato *hereda de* Animal):

```
class Pato extends Animal
```

  - Relación “es”: Un pato *ES* un animal
- Interfaces
  - Utilizamos la palabra *implements* para decir que una clase implementa los métodos de una interfaz

```
class MiHilo implements Runnable {
   public void run() {
        ... // Codigo del método
   }
```

  - Relación “actúa como”: MiHilo *ACTÚA COMO* ejecutable

Herencia e interfaces

- Si una variable es del tipo de la superclase, podemos asignarle también un objeto de la clase hija

```
Animal a = new Pato();
```

- Si una variable es del tipo de una interfaz implementada por nuestra clase, podemos asignarle también un objeto de esta clase

```
Runnable r = new MiHilo();
```

- Sólo se puede heredar de una clase, pero se pueden implementar múltiples interfaces:

```
class Pato extends Animal implements Runnable, ActionListener
```

Punteros *this* y *super*

- *this* se utiliza para hacer referencia a los elementos de la propia clase:

```
class MiClase {
   int i;
   MiClase(int i) {
        this.i = i;    // i de la clase = i del parámetro
```

- *super* se utiliza para llamar al mismo método en la superclase:

```
class MiClase extends OtraClase{
   MiClase(int i) {
        super(i);      // Constructor de OtraClase(...)
```

¿Preguntas...?