

Java y Herramientas de Desarrollo

Sesión 8: Depuración y logging



Puntos a tratar

- El depurador de Eclipse
- Gestión de logs con Log4Java
- La librería commons-logging



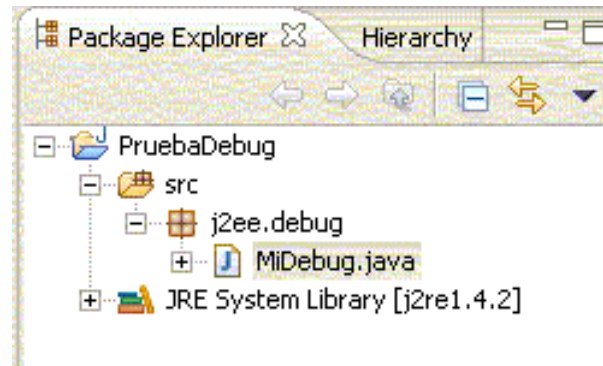
El depurador de Eclipse

- Eclipse incorpora un depurador que permite inspeccionar cómo funciona nuestro código
- Incorpora varias funcionalidades:
 - Establecimiento de *breakpoints*
 - Consulta de valores de variables en cualquier momento
 - Consulta de valores de expresiones complejas
 - Parada/Reanudación de hilos de ejecución
- Existe también la posibilidad de depurar otros lenguajes (C/C++), instalando los plugins adecuados
- Desde Java 1.4 permite cambiar código “en caliente” y seguir con la depuración



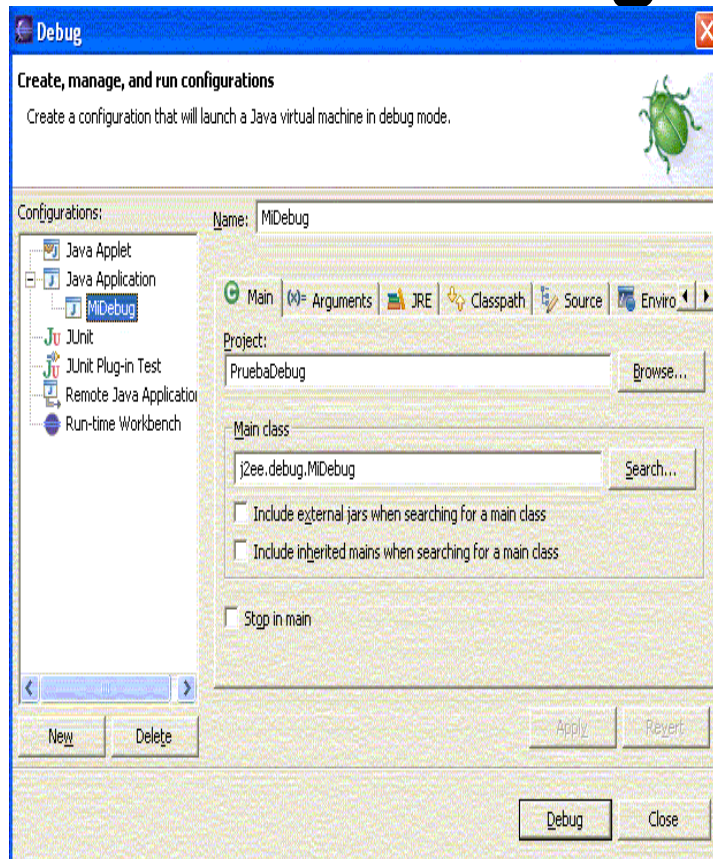
Paso 1: un proyecto compilado

- Para poder probar el depurador, deberemos tener ya nuestro proyecto hecho y correctamente compilado





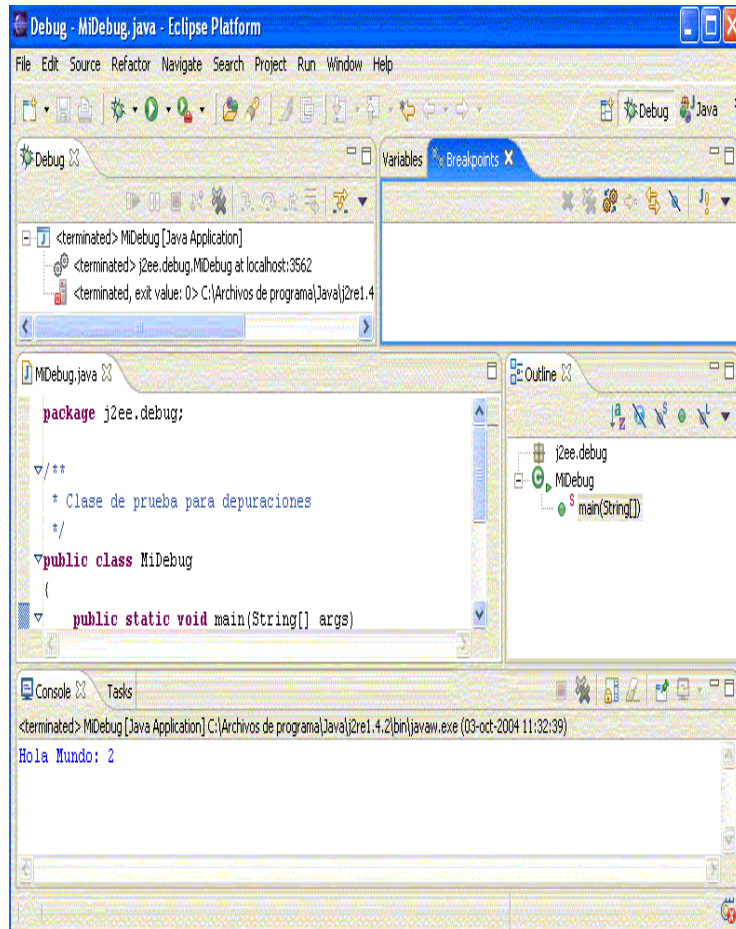
Paso 2: configurar la depuración



- Vamos a *Run – Debug* y creamos (*New*) una nueva configuración de depuración, estableciendo la clase principal a probar
- Podremos tener tantas configuraciones sobre un proyecto como necesitemos (tantas como clases principales, normalmente)
- Pulsando en *Debug* pasaremos a depurar el código. Pulsando en *Close* cerramos la configuración



Paso 3: ir a perspectiva de depuración



- Normalmente al depurar pasamos a la perspectiva de depuración
- Si no es así, vamos a *Window – Open Perspective – Debug*
- Vemos los hilos que se ejecutan, los breakpoints establecidos, las variables que entran en juego... etc



Establecer *breakpoints*

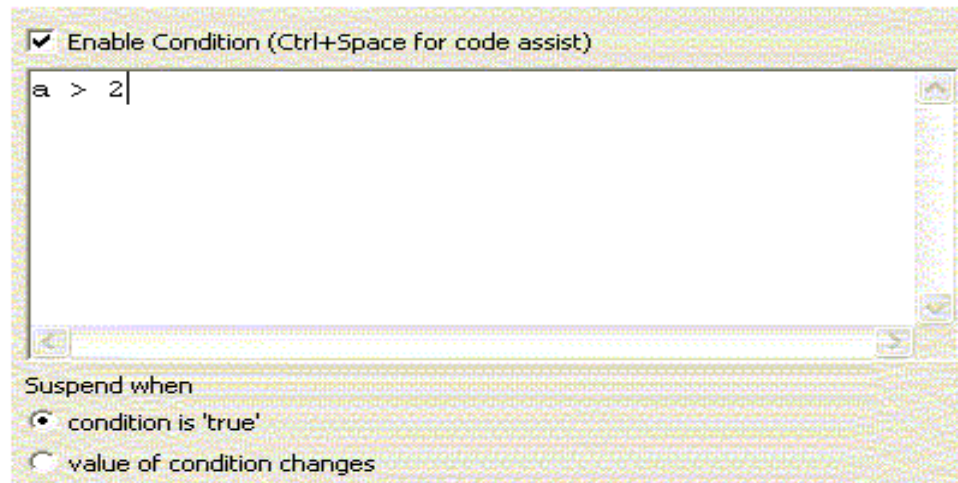
- Un *breakpoint* es un punto donde la ejecución del programa se detiene para examinar su estado
- Para establecerlos, hacemos doble click en el margen izquierdo de la línea donde queremos ponerlo
- Luego arrancamos el programa desde *Run - Debug*

```
{  
    public static void main(String[] args)  
    {  
        int a = 2;  
        for (int i = 1; i < 5; i++)  
            a *= i;  
        System.out.println("Hola Mundo: " + a);  
    }  
}
```



Breakpoints condicionales

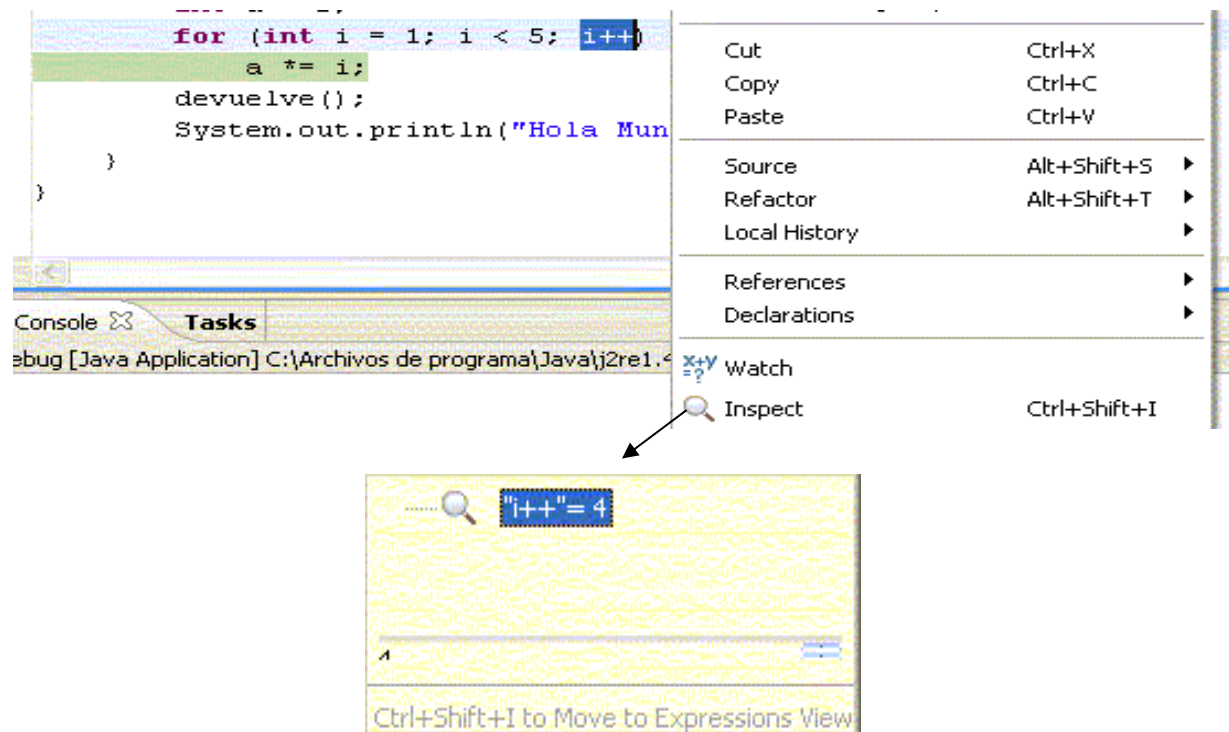
- Se disparan sólo cuando se cumple una determinada condición
- Se establecen con el botón derecho sobre el *breakpoint*, eligiendo *Breakpoint Properties*
- Colocamos la condición en *Enable Condition*





Evaluar expresiones

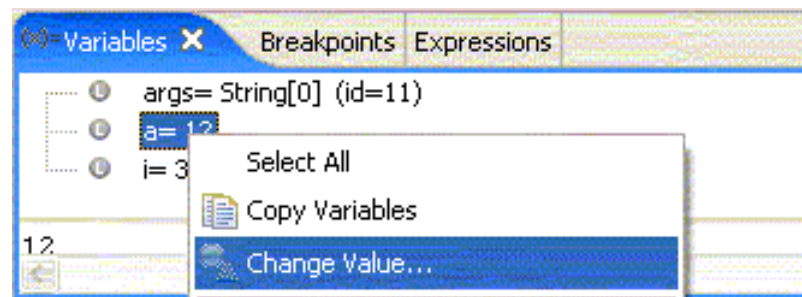
- Podemos ver el valor de una expresión compleja seleccionándola (durante una parada por *breakpoint*) y eligiendo con el botón derecho *Inspect*





Explorar variables

- Si queremos ver qué valores va tomando una variable paso a paso, una vez alcanzado un *breakpoint* vamos a *Run* y vamos dándole a *Step Over* o *F6*
- También podemos, en el cuadro de variables, pinchar sobre una y cambiar su valor





Introducción a Log4J

- Log4Java (Log4J) es una librería *open source* que permite controlar la salida de los mensajes que generen nuestros programas
- Tiene diferentes niveles de mensajes, que se permiten monitorizar con cierta granularidad
- Es configurable en tiempo de ejecución
- Más información en:
 - <http://www.jakarta.apache.org/log4j>



Estructura de Log4J

- El funcionamiento de Log4J se basa en 3 elementos:
 - *Loggers*: entidades asociadas a paquetes o clases, que recogen los mensajes que dichos paquetes o clases generan
 - Se estructuran en forma de árbol, partiendo de un *logger* raíz que existe siempre
 - *Appenders*: indican la salida por la que se muestran los mensajes de los *loggers* (por pantalla, a un fichero... etc)
 - *Layouts*: indican el formato que va a tener el mensaje al mostrarse (fecha, tipo de mensaje, prioridad... etc)



Loggers

- Se tienen 5 tipos (niveles) de mensajes de log:
 - *DEBUG*: mensajes de depuración
 - *INFO*: información acerca del programa (versión, etc)
 - *WARN*: alerta sobre situaciones que no afectan al correcto funcionamiento
 - *ERROR*: errores que afectan al funcionamiento correcto del programa, pero que le permiten continuar
 - *FATAL*: para mensajes críticos que terminan el programa
- Los loggers recogen los mensajes de algunos o todos estos niveles sobre una clase o conjunto de clases
- Hay dos niveles más, *ALL* y *OFF*, para recoger todos los niveles o ninguno



Asociar loggers con clases

- Colocamos un objeto *org.apache.log4j.Logger* en la clase de la que queremos captar sus mensajes
- Luego llamamos a los métodos del logger para generar mensajes de uno u otro nivel:
 - *debug (String mensaje)*
 - *info (String mensaje)*
 - *warn (String mensaje)*
 - *error (String mensaje)*
 - *fatal (String mensaje)*
- Se puede configurar qué niveles recoger



Asociar loggers con clases

- Un ejemplo:

```
import org.apache.log4j.*;
public class MiClase
{
    static Logger logger = Logger.getLogger(MiClase.class);
    ...
    public static void main (String[] args)
    {
        logger.info ("Entrando en la aplicación");
        ...
        logger.warn ("Esto es una advertencia");
        ...
        logger.fatal ("Error fatal");
    }
}
```



Appenders

- Permiten indicar dónde van a ir los mensajes de log
- Todos son clases del paquete *org.apache.log4j*
 - *ConsoleAppender*: para dirigir los mensajes a pantalla
 - *Threshold=WARN* // Nivel mínimo
 - *ImmediateFlush=true* // No buffering
 - *Target=System.err* // Tipo de salida
 - *FileAppender*: para dirigir los logs a un fichero
 - *Threshold=WARN*
 - *ImmediateFlush=true*
 - *File=logs.txt* // Nombre del fichero
 - *Append=false* // Sobreescritura



Appenders

- *RollingFileAppender*: para dirigir los logs a un fichero, que se permite rotar
 - *Threshold=WARN*
 - *ImmediateFlush=true*
 - *File=logs.txt*
 - *Append=false*
 - *MaxFileSize=1MB* // Tamaño máximo (KB|MB|GB)
 - *MaxBackupIndex=2* // Ficheros antiguos a guardar



Otros appenders

- *JDBCAppender*: para dirigir los logs a una base de datos JDBC
- *SocketAppender*: redirecciona los mensajes a un servidor remoto
- *SMTPAppender*: envía un e-mail con los mensajes de log que se le indiquen
- *SyslogAppender*: envía los logs al demonio *syslog* de Unix
- ... etc
- Más adelante veremos cómo establecerlos y configurarlos



Layouts

- Indican el formato de salida de los mensajes de log
- Todos son clases del paquete *org.apache.log4j*
 - *SimpleLayout*: prioridad del mensaje, y texto del mismo
 - DEBUG – Hola, esto es un mensaje
 - *PatternLayout*: admite marcas %x para mostrar las partes que nos interesen:
 - %d: fecha del mensaje (%d{dd/MM/yy HH:mm:ss})
 - %m: texto del mensaje
 - %n: salto de línea en la salida
 - %p: prioridad del evento de log
 - ... etc



Otros layouts

- *HTMLLayout*: la salida la vuelca a una tabla HTML
- *XMLLayout*: saca la salida a un fichero XML compatible con la DTD de Log4J (*log4j.dtd*)
- *TTCCLayout*: saca algunos elementos predefinidos, como fecha, hilo de ejecución, categoría del log y NDC (*Nested Diagnostic Context*) del evento de log.
- ... etc



Configurar Log4J

- Una vez hemos incluido el *Logger* en nuestras clases Java que queremos gestionar, basta con configurar la clase principal
- La configuración por defecto la tenemos con el método *BasicConfigurator.configure()*

```
import org.apache.log4j.*;
public class MiClase {
    static Logger logger = ...;
    ...
    public static void main(String[] args) {
        BasicConfigurator.configure();
        ...
        logger.warn("...");
        ...
    }
}
```



Otras configuraciones

- Podemos configurar Log4J con otras opciones, desde ficheros de propiedades (con *PropertyConfigurator*), o desde ficheros XML (con *DOMConfigurator*)

```
import org.apache.log4j.*;
public class MiClase {
    static Logger logger = ...;
    ...
    public static void main(String[] args) {
        PropertyConfigurator.configure(String fichero);
        DOMConfigurator.configure(String fichero);
        ...
    }
}
```

- Se tienen los métodos *configureAndWatch(...)* en uno y otro, para revisar si la configuración cambia en tiempo de ejecución



Ejemplo de fichero de propiedades

Coloca el nivel root del logger en DEBUG (muestra mensajes de DEBUG hacia arriba)

Añade dos appenders, llamados A1 y A2

log4j.rootLogger=DEBUG, A1, A2

A1 se dirige a la consola

log4j.appender.A1=org.apache.log4j.ConsoleAppender

A1 utiliza PatternLayout

log4j.appender.A1.layout=org.apache.log4j.PatternLayout

log4j.appender.A1.layout.ConversionPattern=%r [%t] %-5p %c %x %m

A2 se dirige a un fichero

log4j.appender.A2=org.apache.log4j.RollingFileAppender

A2 solo muestra mensajes de tipo WARN o superior, en el fichero logs.txt, hasta 1 MB

log4j.appender.A2.Threshold=WARN

log4j.appender.A2.File=logs.txt

log4j.appender.A2.MaxFileSize=1MB



Ejemplo de fichero XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration>
  <!-- Definimos appender A1 de tipo ConsoleAppender -->
  <appender name="A1" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%r [%t] %-5p %c %x %m"/>
    </layout>
  </appender>
  <!-- Definimos appender A2 de tipo RollingFileAppender -->
  <appender name="A2" class="org.apache.log4j.RollingFileAppender">
    <param name="File" value="logs.txt"/>
    <param name="Threshold" value="warn"/>
    <param name="MaxFileSize" value="1MB"/>
    <layout class="org.apache.log4j.SimpleLayout">
  </appender>
  <!-- Configuramos el root logger -->
  <root>
    <priority value ="debug" />
    <appender-ref ref="A1"/>
    <appender-ref ref="A2"/>
  </root>
</log4j:configuration>
```




La librería commons-logging

- Es una librería de Jakarta que permite encapsular la librería de log que queramos (actúa como *wrapper*).
- De esta forma, no hace falta tocar el código fuente si queremos cambiar de librería de logging, basta con cambiar unos ficheros de configuración.
- Internamente, permite trabajar, entre otras, con Log4J, con la librería básica de log de JDK (*SimpleLog*), etc.

<http://jakarta.apache.org/commons/logging/>



Definir mensajes en nuestras clases

```
import org.apache.commons.logging.*;
public class MiClase
{
    static Log logger = LogFactory.getLog(MiClase.class);
    ...
    public static void main (String[] args)
    {
        logger.info ("Entrando en la aplicación");
        ...
        logger.warn ("Esto es una advertencia");
        ...
        logger.fatal ("Error fatal");
    }
}
```



Configurar el logging: Log4J

- Necesitamos 2 ficheros de *properties* en el CLASSPATH:
- Uno llamado *commons-logging.properties* que indique que se va a usar Log4J como librería interna de logging:

```
org.apache.commons.logging.Log=org.apache.commons.logging.impl.Log4JLogger
```

- Otro llamado *log4j.properties* como los de configuración ya vista de Log4J, con la configuración que queramos utilizar:

```
# Coloca el nivel root del logger en DEBUG (muestra mensajes de DEBUG hacia arriba)
# Añade appender A1
log4j.rootLogger=DEBUG, A1
# A1 se dirige a la consola
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.Threshold=INFO
# A1 utiliza PatternLayout
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d{dd/MM/yyyy HH:mm:ss} - %p - %m %n
```



Configurar el logging: SimpleLog

- Podemos utilizar otras librerías, como la de logging de JDK (*SimpleLog*). Necesitamos 2 ficheros de *properties* en el CLASSPATH:
- Uno llamado *commons-logging.properties* como antes, pero que ahora indique que se va a utilizar el *SimpleLog* de JDK:

```
org.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog
```

- Otro llamado *simplelog.properties* con la configuración específica para *SimpleLog*. Un ejemplo de configuración sería:

```
# Nivel de log general ("trace", "debug", "info", "warn", "error", o "fatal"). Defecto="info"  
org.apache.commons.logging.simplelog.defaultlog=warn  
# A true si queremos que el nombre del Log aparezca en cada mensaje en la salida  
org.apache.commons.logging.simplelog.showlogname=false  
# A true si queremos que el nombre corto del Log aparezca en cada mensaje en la salida  
org.apache.commons.logging.simplelog.showShortLogname=true  
# A true si queremos poner fecha y hora actuales en cada mensaje en la salida  
org.apache.commons.logging.simplelog.showdatetime=true
```