LPS: Ficheros y excepciones

Federico Peinado
www.federicopeinado.es

Depto. de Ingeniería del Software e Inteligencia Artificial
disia.fdi.ucm.es

Facultad de Informática
www.fdi.ucm.es

Universidad Complutense de Madrid
www.ucm.es

Ficheros

Ficheros

- Los representa la clase `java.io.File`
- Atributos y métodos para manipular ficheros:
  - pathSeparator
    - Cadena que representa la separación entre directorios dentro de una ruta (*dependiente del S.O. subyacente*)
  - canRead
  - canWrite
  - createNewFile
  - delete
  - isDirectory
  - isFile
  - mkdir
  - …

Funcionamiento

- Su tratamiento se hace con los *streams* que ya hemos estudiado anteriormente
  - Pueden usarse para crear flujos de entrada/salida (FileInputStream / FileOutputStream)
  - Pueden usarse para crear lectores/escritores (FileReader / FileWriter)
- Van a sufrir un rediseño total en Java 7
  - Una correspondencia aproximada entre métodos: http://download.oracle.com/javase/tutorial/essential/io/legacy.html#mapping

Ficheros de propiedades

- Se pueden cargar de (o guardar desde) un objeto de la clase `java.util.Properties`
  - Es una subclase de Hashtable donde tanto la clave como el valor son cadenas de texto
- Métodos para trabajar con propiedades:
  - getProperty
  - list
  - load
  - loadFromXML
  - setProperty
  - store
  - storeToXML
  - ...

Ejemplos de estos ficheros

```
Profesor = Federico Peinado
Asignatura = LPS
Dificultad = Sin comentarios
...
```

```
# Sample Messages
MESSAGE.INTRO: Game engine v.1
MESSAGE.HELP: You will not get help from me
  MESSAGE.QUIT: Coward...
MESSAGE.GAMEOVER :  You lose. Bye bye!
...
```

```
! Comentario tonto
Cantidad_de_oro  14
   ! Más comentarios...
 Cantidad_de_madera   : 2300

Vidas_extra    = No quedan
```

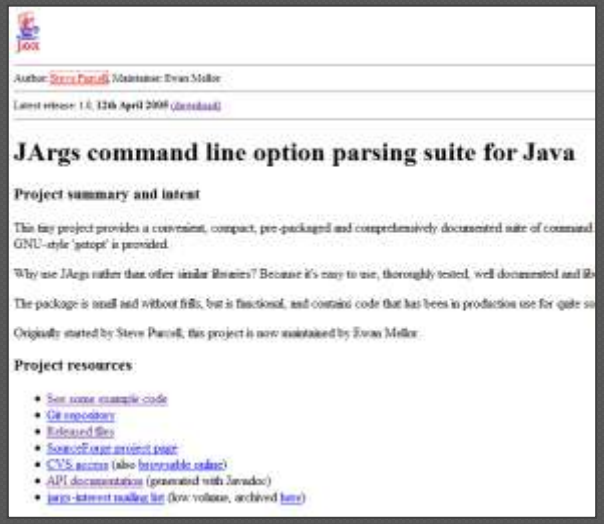
StreamTokenizer

- Clase **`java.io.StreamTokenizer`** que actúa como analizador simple para extraer los lexemas (*tokens*) de un flujo de entrada
  - Puede servir para analizar el flujo de un fichero de texto en un cierto "lenguaje formal" de sintaxis básica
  - Utiliza una tabla de categorías léxicas y una serie de centinelas (*flags*) para variar sus funciones
  - Reconoce identificadores "tipo Java", números, cadenas de texto entrecomilladas y comentarios
  - Para lenguajes más complejos, hay librerías Java muy avanzadas con las que hacer procesadores de lenguajes
    - ANTLR http://www.antlr.org

Funcionamiento

- Atributos interesantes
  - nval y sval
    - Variables donde se almacena el valor (numérico o textual) del último lexema leído
  - TT_EOF, TT_EOL, TT_NUMBER y TT_WORD
    - Constantes enteras que dicen si el último lexema ha sido fin de flujo, de línea, número u otra cosa
    - Curiosidad: La clase *no usa enumerados* por ser muy vieja
- Constructor (recibe un Reader)
  - Por eficiencia conviene envolver el flujo de entrada así:
    `new BufferedReader(new InputStreamReader(inStream))`
- Métodos interesantes
  - nextToken
  - lineno
    - Número de línea por el que se encuentra el análisis
  - Varios para configurar las definiciones léxicas de cada *token* antes de empezar a analizar, como commentChar, ordinaryChar, quoteChar, etc.

Extra: JArgs



JArgs command line option parsing suite for Java

Project summary and intent

Project resources

- JArgs http://jargs.sourceforge.net
  - Releases (última de 2005)
    - Librería JAR que debemos usar
  - Documentación del API
  - CVS (repositorio colaborativo, al ser un proyecto de código libre)
  - Lista de correo, información y contacto del autor
- Pequeño analizador para los argumentos que recibe la aplicación por línea de comandos
  - Ej: `run -c engine.cfg ExampleGame.game`

Extra: JArgs

```
import jargs.gnu.CmdLineParser; // Teniendo el JAR en el Build Path
...
public static void main(String[] args) {
   CmdLineParser parser = new CmdLineParser();
   CmdLineParser.Option debug = parser.addBooleanOption('d', "debug");
   CmdLineParser.Option size = parser.addIntegerOption("size");
   CmdLineParser.Option name = parser.addStringOption('n', "name");
   ...
   try { parser.parse(args); }
   catch (CmdLineParser.OptionException e) {
         System.err.println(e.getMessage());
         // Mostrar opciones las disponibles al usuario (debug, size...)
         System.exit(-1);
   }
   Boolean debugValue = (Boolean)parser.getOptionValue(debug);
   Integer sizeValue = (Integer)parser.getOptionValue(size);
   String nameValue = (String)parser.getOptionValue(name);
   ...
   String[] otherArgs = parser.getRemainingArgs();
   // Lanzar el programa con las correspondientes opciones activas
   System.exit(0);
}
...
```

Excepciones

Excepciones

- Mecanismo de gestión de errores de ejecución
  - Fragmentos de *código fuente* específicos para esta gestión
  - *Palabras reservadas* como **`throw`**, **`throws`**, **`try`**, **`catch`**…
- Las clases que representan excepciones heredan de **`java.lang.Exception`**
- No deben usarse para implementar *la lógica del programa*, pero sí deben documentarse en los Javadoc porque son relevantes para el cliente
- Java Tutorials
  http://java.sun.com/docs/books/tutorial/essential/exceptions

Ejemplo de excepción

```
public class RaizNoExiste extends Exception {
...
};


/**
 * ...
 * @throws NoHayRaiz
 */
public class Ecuacion {
   public float raiz(int i) throws NoHayRaiz{
       ...
         throw new NoHayRaiz();
       ...
  }
}
```

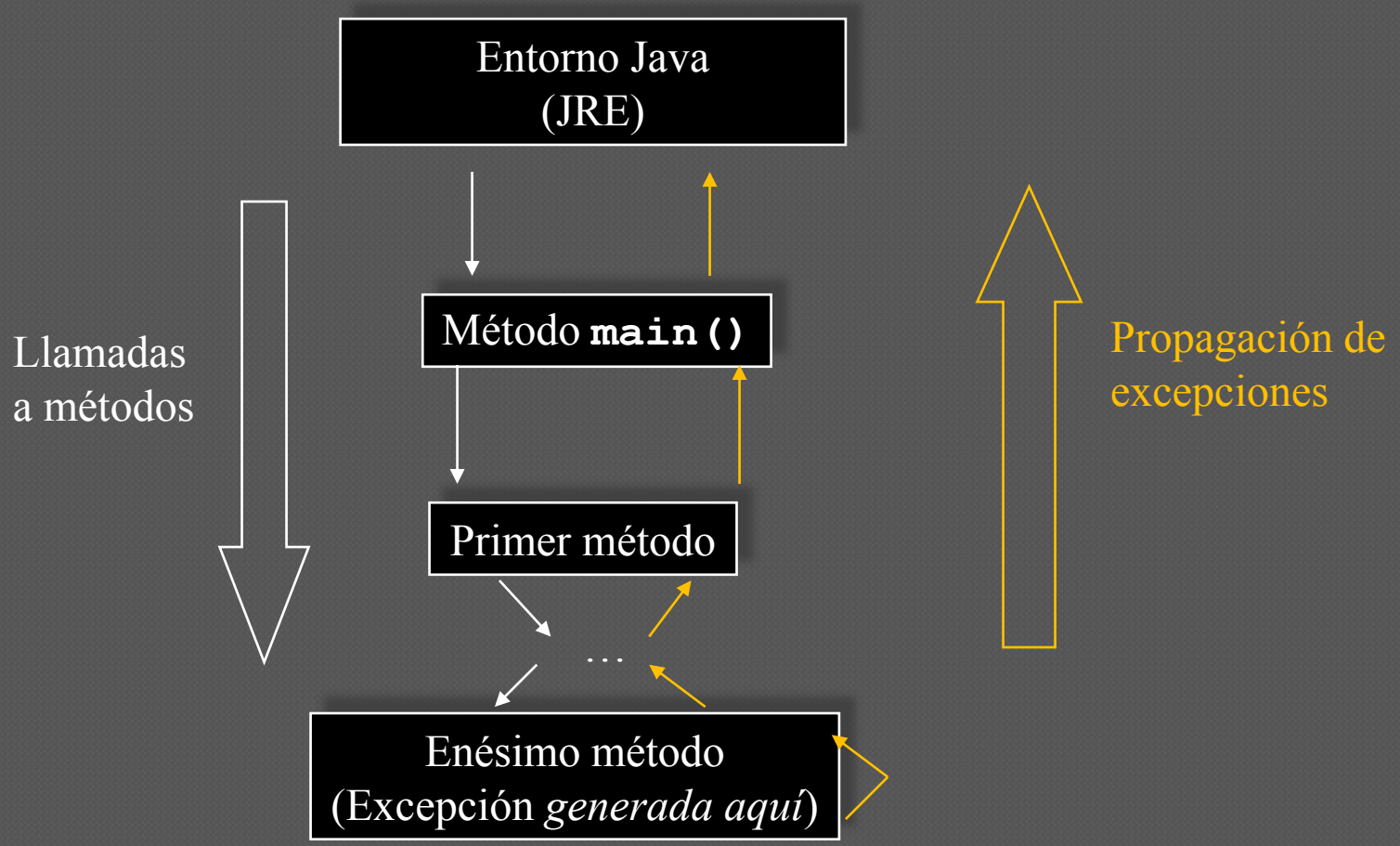
Tipos de excepciones

- Excepciones comprobadas (*checked*)
  - Representan situaciones excepcionales que se producen en un método y deben ser propagadas al que lo ha llamado
  - El compilador *obliga* al método que llama a tratar dichas excepciones
- Excepciones no comprobadas *(unchecked)*
  - Representan situaciones excepcionales que es producen en un método y que normalmente impedirán continuar con la ejecución del programa
  - *Opcionalmente* podemos tratar y recuperarnos de dichas excepciones desde el método que llama
  - Heredan de `RuntimeException`
    - `ArrayIndexOutOfBoundsException`
    - `ArithmeticException`
    - `NullPointerException`
    - `ClassCastException`
    - `IllegalArgumentException`
    - …

Propagación de excepciones

- Las excepciones se propagan de método en método en sentido contrario a las llamadas
  - Hasta que algún catch las captura y entonces la ejecución continúa por ahí
  - Si la excepción alcanza el método `main()` sin ser capturada el programa y todo el JRE se detiene, mostrando el estado de la pila de llamadas por pantalla
- Los métodos *obligatoriamente* deben declarar las excepciones comprobadas que lanzan en su cabecera
  - Aunque conviene documentarlas todas, comprobadas y no comprobadas

Esquema de propagación



Tratamiento de excepciones

```
try {
   // Bloque de código en el que se puede producir una excepción
}

catch (TipoExcepcion1 ex1) {
   // Gestor de excepciones de tipo TipoExcepcion1
   // Se puede seguir propagando esta u otras excepciones con
   throw(ex1);
}

catch (TipoExcepcion2 ex2) {
   // Gestor de excepciones de tipo TipoExcepcion2
   ex2.printStackTrace(); // Imprimir por pantalla la pila de llamadas
}

finally {
   // Bloque de código que se ejecuta siempre,
   // se haya producido o no una excepción
}
```

(Una posible) política de uso

- Las excepciones son para situaciones realmente "extraordinarias" (aunque posibles, claro). Todas las "ordinarias" deberían poder resolverse sin ellas
  - Documentando bien aquellos casos extremos o peculiares
  - Incluyendo toda información relevante en los objetos devueltos
  - …
- Las excepciones comprobadas se usan *comprometiendo al cliente* a que "haga algo" con respecto a la situación extraordinaria ocurrida
  - Cuanto antes detectemos la excepción y la lancemos, mejor
- Las excepciones no comprobadas se usan *asumiendo que la culpa es del código cliente* (o que ha habido *un error irremediable del sistema*) y por lo tanto, es posible que se produzca un fallo total de la aplicación

Críticas, dudas, sugerencias…

Federico Peinado

www.federicopeinado.es