

Tema 1 C – Revisión del API de Java

Unai Aguilera Irazabal

unai.aguilera@deusto.es

Grado en Ingeniería Informática

Facultad de Ingeniería – Universidad de Deusto

Reflexiona

- Piensa sobre el proceso de desarrollo de programas “reales”
 - Datos de entrada
 - Datos de salida
 - Dibujado en pantalla
 - Interacción con el usuario
 - Hilos
 - Comunicaciones
 - Etc.
- ¿Tiene sentido programar una y otra vez la misma funcionalidad?
- ¿Se te ocurre alguna forma de reducir el tiempo de desarrollo de programas?

Application Programming Interface - API

- Java no es solamente un lenguaje de programación, cuya especificación puede encontrarse aquí
 - <https://docs.oracle.com/javase/specs/>
- Además de dominar las características del lenguaje es necesario conocer las utilidades que proporciona en su librería, que se denomina API.
- La documentación de referencia para Java 8 se encuentra disponible en
 - <https://docs.oracle.com/javase/8/docs/api/>
- Y para Java 17 en
 - <https://docs.oracle.com/en/java/javase/17/docs/api/>
- El código de los ejemplos se encuentra disponible en
 - <https://github.com/unaguil/prog3-ejemplos-codigo>

The screenshot displays the Java Platform, Standard Edition 8 API Specification page. The left sidebar lists 'All Classes' and 'All Profiles'. The main content area shows the 'Overview' section, which includes a description of the API and a list of profiles (compact1, compact2, compact3). Below this, a table lists various packages and their descriptions.

Package	Description
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.
java.awt.print	Provides classes and interfaces for a general printing API.
java.beans	Contains classes related to developing beans -- components based on the JavaBeans™ architecture.
java.beans.beancontext	Provides classes and interfaces relating to bean context.

Paquetes de Java

- Todo la API de Java se encuentra organizada en paquetes.
- La tabla muestra aquellos más comúnmente utilizados.

Paquete	Descripción
<i>java.awt</i> <i>javax.swing</i>	Clases para crear interfaces de usuario y pintar gráficos e imágenes.
<i>java.io</i>	Clases para llevar a cabo E/S con <i>streams</i> , ficheros y serialización.
<i>java.lang</i>	Clases fundamentales utilizadas por el lenguaje Java.
<i>java.net</i>	Clases para comunicación a través de red.
<i>java.text</i>	Clases para manejar textos, números y mensajes.
<i>java.rmi</i>	Proporciona clases para RMI (Remote Method Invocation).
<i>java.security</i>	Clases para gestionar la seguridad de acceso a recursos.
<i>java.sql</i> <i>javax.sql</i>	Clases para acceder a bases de datos relacionales.
<i>java.util</i>	Contiene las colecciones, eventos, utilidades de fecha y tiempo, internacionalización.
<i>org.w3c</i> <i>org.xml</i>	Contiene clases para utilizar documentos DOM y XML, respectivamente.

Entrada/salida de datos

- La entrada/salida en Java se lleva a cabo mediante una abstracción en streams (flujos) de datos.
- En concreto hay dos clases abstractas en Java, en el paquete *java.io*, para representar la entrada/salida: ***InputStream***, ***OutputStream***, ***Writer***, ***Reader***
- Estas clases abstractas proporcionan la implementación básica de E/S de Java. Tienen funcionalidad para leer/escribir bytes, cerrar el flujo de datos, y hacer algunas otras comprobaciones.

```
java.io
Class InputStream

java.lang.Object
  java.io.InputStream

All Implemented Interfaces:
  Closeable, AutoCloseable

Direct Known Subclasses:
  AudioInputStream, ByteArrayInputStream, FileInputStream, FilterInputStream, InputStream, ObjectInputStream, PipedInputStream, SequenceInputStream, StringBufferInputStream
```

- El resto de clases del paquete, todas las clases *xxxInputStream* y *xxxOutputStream*, *xxxReader*, *xxxWriter*, heredan de estas clases y particularizan la funcionalidad para el flujo de datos concreto.
- Un aspecto importante de estas clases (y todas su clases derivadas) es que, a partir de Java 7, implementan la interfaz *Closeable*, que facilita el cierre automático de los recursos asociados.
 - Se debe hacer uso de esta nueva funcionalidad.

Salida de datos

- Si queremos escribir en un flujo de salida, por ejemplo, un fichero de texto, necesitamos crear un objeto *OutputStream* del tipo adecuado al tipo de salida.
 - Por ejemplo, si queremos escribir a un fichero, tenemos que crear un *FileOutputStream* y usarlo para escribir los datos.

```
// Creamos el objeto FileOutputStream usando el constructor
// que recibe la ruta al fichero de salida.

FileOutputStream fos = null;
try {
    fos = new FileOutputStream("output.txt");

    fos.write('H');
    fos.write('o');
    fos.write('l');
    fos.write('a');

} catch (FileNotFoundException e) {
    // Si no se encuentra el fichero al intentar abrirlo
    System.out.println("No se pudo encontrar el fichero");
} catch (IOException e) {
    // Si hay problemas al escribir en el fichero.
} finally {
    // En cualquier caso, tanto si hay error como si no hay,
    // se comprueba si el stream está abierto y se cierra
    if (fos != null) {
        try {
            fos.close();
        } catch (IOException e) {
            //
        }
    }
}
```

Este ejemplo escribe 'Hola'
en un fichero carácter a
carácter.

**¡Atención a la gestión
correcta de las
excepciones!**

EjemploSalidaNoRecomendado.java

Closeable

- La forma anterior para la gestión de los errores de E/S no es la recomendada a partir de Java 7 → Se debe hacer uso de la nueva interfaz *Closeable* y del *try-with-resources*.

```
// Creamos el objeto FileOutputStream usando el constructor
// que recibe la ruta al fichero de salida.

try (FileOutputStream fos = new FileOutputStream("output.txt")) {

    fos.write('H');
    fos.write('o');
    fos.write('l');
    fos.write('a');

} catch (FileNotFoundException e) {
    // Si no se encuentra el fichero al intentar abrirlo
    System.out.println("No se pudo encontrar el fichero");
} catch (IOException e) {
    // Si hay problemas al escribir en el fichero.
    System.out.println("Hay problemas al escribir");
}
```

Existe un nuevo try que permite poner el recurso (fichero, socket, etc) que se quiere usar entre paréntesis.

Java se encarga de cerrar el recurso cuando finaliza el try, y lo hace correctamente, tanto si ha habido una excepción como si no ha habido problemas.

- A partir de este momento, en todos los ejemplos que utilicen recursos, utilizaremos el *try-with-resources*, ya que simplifica enormemente la gestión de los errores.

BufferedWriter

- En el ejemplo anterior teníamos que escribir los datos uno a uno, ya que `FileOutputStream` es una clase sencilla para escribir a fichero.
- Si queremos escribir texto, tenemos que utilizar las clases `Writer`.

```
try (BufferedWriter buffWriter = new BufferedWriter(new FileWriter("output.txt"))) {  
    buffWriter.write("Hola");  
} catch (FileNotFoundException e) {  
    // Si no se encuentra el fichero al intentar abrirlo  
    System.out.println("No se pudo encontrar el fichero");  
} catch (IOException e) {  
    // Si hay problemas al escribir en el fichero.  
    System.out.println("Hay problemas al escribir");  
}
```

EjemploSalidaWriter.java

- Existen más clases relacionadas con *Writers*, *OutputStream* y las correspondientes para lectura: *Reader* e *InputStream*.
- Consultar las jerarquías de clases para ver todas las que hay.

Entrada desde consola

- En el paquete *java.util* existe una clase, *Scanner*, para facilitar la lectura de datos desde flujos de entrada y otras fuentes de datos.
- Contiene métodos para leer y convertir directamente datos desde la entrada.

```
// Creamos una instance de la clase y le pasamos  
// el stream de entrada de la consola  
try(Scanner sc = new Scanner(System.in)) {
```

```
    System.out.print("Nombre: ");  
    String name = sc.nextLine();
```

```
    System.out.print("Edad: ");  
    int age = sc.nextInt();
```

```
    System.out.println(  
        String.format("Te llamas %s y tienes %d años", name, age)  
    );  
}
```

EjemploScanner.java

Atención en el ejemplo al
uso del `String.format`

- Contiene métodos más avanzados para utilizar expresiones regulares.

Fechas y tiempo

- En la versión 8 de Java se añadió nuevas clases en el paquete *java.time* que mejoran la gestión de fechas y tiempos.

```
// Obtenemos la fecha/tiempo actual
ZonedDateTime now = ZonedDateTime.now();

// Escribimos por consola la fecha con la localización
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss ZZ");
System.out.println("Fecha/hora/zona: " + formatter.format(now));

// Sumar 10 a la fecha actual
ZonedDateTime futureTime = now.plusYears(10);
System.out.println("Fecha/hora/zona: " + formatter.format(futureTime));

// Convertir un string en formato dd-MM-YY a una fecha interna
DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern("dd-MM-yyyy");
LocalDate newDate = LocalDate.parse("22-09-2020", dateFormatter);
System.out.println("Fecha parseada: " + newDate);
```

EjemploTime.java

Localización

- El paquete *java.util.locale* contiene clases para ayudar con la localización de los programas.

```
// Obtenemos la fecha/tiempo actual
ZonedDateTime now = ZonedDateTime.now();

// Escribimos por consola la fecha con formato de texto largo en el locale actual
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL);
System.out.println(formatter.format(now));

// Escribimos por consola la fecha con formato de texto largo en el locale japonés
DateTimeFormatter japaneseFormatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL).withLocale(Locale.JAPAN);
System.out.println(japaneseFormatter.format(now));

// Escribimos por consola la fecha con formato de texto largo en el locale italiano
DateTimeFormatter italianFormatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL).withLocale(Locale.ITALY);
System.out.println(italianFormatter.format(now));
```

EjemploLocale.java

Números y localización

- Además de fechas podemos formatear con la localización adecuada otros aspectos como los separadores de números.
 - En los países ingleses se usa el punto en vez de la coma.
 - Java proporciona utilidades para facilitar esta tarea.

```
double d = 1_234_567.89; // Los _ se pueden usar para leer mejor los números largos

DecimalFormat dfLocale = new DecimalFormat();
NumberFormat nfUS = DecimalFormat.getNumberInstance(Locale.US);
DecimalFormat dfManual = new DecimalFormat("0");
DecimalFormat dfManualConDecimales = new DecimalFormat("0.000");

System.out.println( "Formato local: " + dfLocale.format( d ));
System.out.println( "Formato US: " + nfUS.format( d ));
System.out.println( "Formato adhoc entero: " + dfManual.format( d ));
System.out.println( "Formato adhoc real: " + dfManualConDecimales.format( d ));

// Ver documentación de java.util.Formatter
System.out.println( "Formato a través de " + String.format( "String.format(): #%%1$5d# vs. #%%2$,12.1f#", 123, d ));
```

EjemploDecimalFormat.java

StringTokenizer

- Una clase de *java.util.* para procesar Strings en elementos es StringTokenizer.
 - Nos permite partir un String en partes usando un separador.

```
try (Scanner sc = new Scanner(System.in)) {  
    System.out.print("Escribe algo: ");  
    String input = sc.nextLine();  
  
    // Se crea una instancia del tokenizer y se pasa el string  
    // a partir y los caracteres por los que partir"  
    StringTokenizer st = new StringTokenizer(input, " ,.");  
  
    // se itera sobre las partes que nos da el tokenizer  
    while (st.hasMoreTokens()) {  
        // se obtiene el token siguiente y se imprime  
        String token = st.nextToken();  
        System.out.println(token);  
    }  
}
```

EjemploTokenizer.java

Random

- El paquete `java.util` contiene también la clase `Random` para generar números pseudo-aleatorios.
- Un computador normal no puede generar números completamente aleatorios.

```
Random r = new Random();
System.out.println("Tres enteros aleatorios de 0 a 99:" );
System.out.println(r.nextInt(100) + ", " + r.nextInt(100) + ", " + r.nextInt(100));
System.out.println("Tres reales aleatorios de 0 a 1:" );
System.out.println(r.nextDouble() + ", " + r.nextDouble() + ", " + r.nextDouble());

r = new Random(15); // ponemos una semilla para forzar la secuencia
System.out.println("Tres enteros aleatorios con semilla 15, de 0 a 99:" );
System.out.println(r.nextInt(100) + ", " + r.nextInt(100) + ", " + r.nextInt(100));

r = new Random(15); // si elegimos la misma semilla obtenemos la misma secuencia.
System.out.println("Tres enteros aleatorios con nueva semilla 15, de 0 a 99:" );
System.out.println(r.nextInt(100) + ", " + r.nextInt(100) + ", " + r.nextInt(100));
```

Expresiones regulares

- Las expresiones regulares son una secuencia de caracteres que definen un patrón de búsqueda.
- El significado de cada símbolo se encuentra en <https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html>

```
// Significado de la expresión regular:  
// 1. Empieza por: p  
// 2. Le siguen cero o más caracteres: .*  
// 3. Después viene un punto: \. ¡Cuidado! La barra hay que escaparla por eso se pone \\  
// 4. Después vienen uno o más caracteres: .+  
String regex = "p.*\\.+";  
  
System.out.println("Patrón: " + regex);  
Pattern pattern = Pattern.compile(regex); // Se compila el patrón de búsqueda  
for (String s : testNombres) {  
    if (pattern.matcher(s).matches()) // Se comprueba si pasa el filtro  
        System.out.println("Cumple: " + s);  
    else  
        System.out.println("No cumple: " + s);  
}
```

EjemploRegex.java

Logger

- Cuando se tiene un programa complejo pueden ocurrir muchas cosas durante su ejecución.
 - A veces, es complicado saber qué ha ocurrido hasta cierto punto y/o por qué se ha producido cierto fallo.
- Para tener un registro de lo ocurrido durante la ejecución es usual sacar mensajes por la consola o a un fichero de las acciones que va realizando el programa.
 - Se puede utilizar `System.out` para sacar mensajes pero no se recomienda.
- La opción recomendada es utilizar las librerías de logging para Java.
 - Se basan en sacar mensajes con distintos niveles de importancia.
 - Estos mensajes pueden ser filtrados por su nivel o desactivados completamente.
 - Pueden enviarse a consola, fichero, enviar a un servidor etc.
 - Dependerá de la configuración que hayamos indicado al *logger*.

```
logger.info("Programa comenzado");
```

```
...
```

```
logger.log(Level.SEVERE, "No se ha encontrado el fichero 'noexiste.txt')"
```


Creación del objeto de logger

- Cada clase que quiera sacar información a través del *logging* debe comenzar creando una instancia del objeto de *logger*.
 - Se crea una instancia estática en los atributos de clase.
 - Esta instancia es la que se utilizará en esta clase para sacar mensajes de log.

```
private static Logger = Logger.getLogger(EjemploLogger.class.getName());
```

- Cada logger se identifica con un nombre, que se utilizará en el mensaje de salida.
 - Lo usual, como vemos en el ejemplo anterior es utilizar el nombre de la clase desde donde se está *loggeando*.

```
private static Logger = Logger.getLogger(EjemploLogger.class.getName());
```

- En cada clase que se cree un logger se utilizará como nombre el de esa propia clase.

Usando el *logger* en la clase

- Ahora que hemos instanciado el logger en la clase donde se va a utilizar podemos empezar a sacar mensajes a través del mismo.
- Utilizamos para ello los métodos disponibles del objeto *logger*.
 - *logger.fine(mensaje)*: saca el mensaje con nivel FINE.
 - *logger.config(mensaje)*: saca el mensaje con nivel CONFIG.
 - *logger.info(mensaje)*: saca el mensaje con nivel INFO.
 - *logger.warning(mensaje)*: saca el mensaje con nivel WARNING.
 - *logger.severe(mensaje)*: saca el mensaje con nivel SEVERE.
- Existe también un método genérico de salida en el que el nivel de log para el mensaje se indica como un parámetro.
 - *logger.log(level, mensaje)*: donde *level* es un de los niveles disponibles en la clase Level. Por ejemplo, *Level.FINE*, *Level.WARNING*, *Level.INFO*, etc.

Niveles del logger

- Hemos visto que cada mensaje que se saca a través del logger tiene un nivel de salida asociado.
- Existen 5 niveles principales de *logging*
 - FINE, CONFIG, INFO, WARNING, SEVERE
 - Sus nombres indican para que tipo de mensajes se utiliza cada uno.
- El uso de niveles permite
 - Clasificar los mensajes en la salida.
 - Realizar un filtrado de los mensajes que se sacan. Es decir, en una ejecución determinada del programa podríamos indicar que únicamente estamos interesados en los mensajes de INFO o superiores.
 - Es decir, saldrían por el log los mensajes de INFO, WARNING y SEVERE, pero no aquellos de nivel inferior FINE y CONFIG.
 - De esta forma, podemos reducir la cantidad de mensajes que vemos en el log en una ejecución determinada.
 - El nivel de salida del *logger principal* se establece de la siguiente manera

```
Logger.getLogger("").setLevel(Level.FINE);  
Logger.getLogger("").getHandlers()[0].setLevel(Level.FINE);
```

Práctica I C – Parte I

- Crea un programa principal para probar la funcionalidad del logger.
 - Instancia en la clase un logger como se ha explicado anteriormente.
 - Ahora en el código prueba los métodos de salida de información, escribiendo distintos mensajes con diferentes niveles de salida.
- Prueba el programa que has creado para ver si los mensajes indicados salen correctamente.
 - Verás que, por defecto, no salen los mensajes de nivel inferior a INFO.
- Cambia la configuración, utilizando el método de configuración del nivel de salida visto anteriormente, para que salgan los mensajes de nivel FINE.
 - Comprueba que ahora sí salen estos mensajes por la consola.

Fichero de configuración

- Es común establecer la configuración del logger en un fichero externo llamado, normalmente, *logger.properties*.
 - Es un fichero de propiedades *clave=valor*.
- Son comunes las claves que se explican a continuación
 - *handlers*: indica dónde se escribe la salida del log. Puede ser a varios destinos simultáneamente:

```
handlers = java.util.logging.FileHandler, java.util.logging.ConsoleHandler
```
 - *.level*: Nivel de filtrado de alto nivel. Puede ser uno de los niveles explicados anteriormente o *ALL* para sacar todos los mensajes.
- Incluso se pueden especificar configuraciones para cada *handler* que se utiliza en el *logger*.

Ejemplo de configuración del logger

- Incluye comentarios explicativos

```
# Esta es la configuración general del logger
handlers = java.util.logging.FileHandler, java.util.logging.ConsoleHandler
# Configuramos el logger principal para sacar todos los mensajes
.level = ALL

# Configuración de la salida a consola
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

# Configuración de la salida a fichero
java.util.logging.FileHandler.pattern = ejemplo.log
java.util.logging.FileHandler.level = FINE
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
```

- Para cargar una configuración de *logger* desde fichero

```
try (FileInputStream fis = new FileInputStream("logger.properties")) {
    LogManager.getLogManager().readConfiguration(fis);
} catch (IOException e) {
    logger.log(Level.SEVERE, "No se pudo leer el fichero de configuración del logger");
}
```

Práctica I C – Parte II

- Se pide ahora que crees, en el directorio donde este el programa principal, un fichero de configuración del logger.
- Lee este fichero para configurar el logger.
- De tal forma que el programa principal
 - Escriba el log por consola
 - Escriba a un fichero de texto que se llame “salida.log”.
 - Filtra para que a consola salgan únicamente mensajes INFO o superior.
 - Filtra para que a fichero salgan todos los mensajes FINE o superior.
- Prueba el programa para ver que el logger funciona como se espera.
 - Haz diferentes pruebas cambiando opciones en el fichero de configuración del logger.