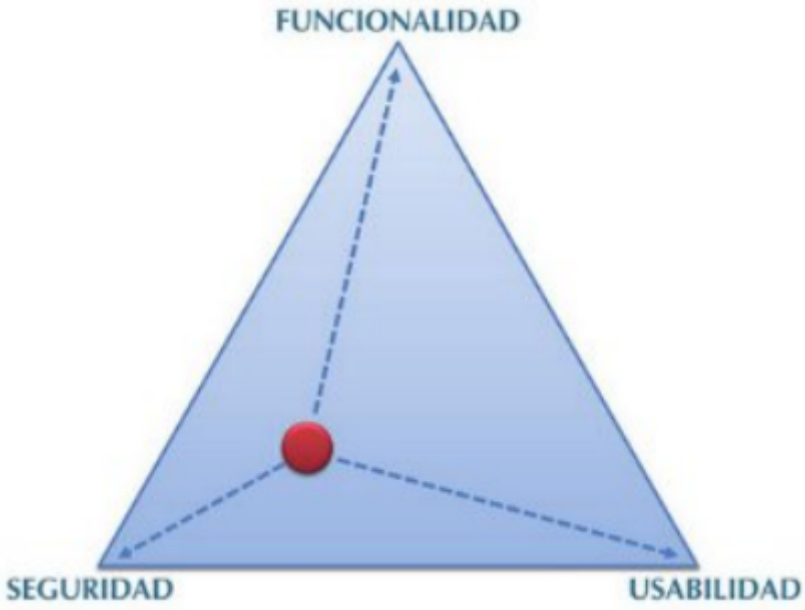
UD05: Programación segura

1. Introducción

Las Tecnologias de la Información y las Comunicaciones (TIC), los sistemas informáticos y la informática la encontramos prácticamente en todos los ámbitos de nuestro día a día. Todas estas tecnologas nos ayudan a realizar nuestro trabajo o disfrutar de nuestro tiempo de ocio de una manera más rápida y descentralizada. Al mismo tiempo eso supone que algo tan transversal como la seguridad informática nos debe acompañar en todo momento. Debemos tener siempre presente la seguridad informática en todas las acciones y decisiones que tomemos en nuestros sistemas informáticos. Este es el único modo de garantizar que esten el mayor tiempo posible disponibles, que realicen correctamente la tarea para los que fueron diseñados y que sean usados solamente por aquellos usuarios a los que se les ha autorizado.

2. Fundamentos de seguridad informática

La seguridad informática es una disciplina que requiere mucha dedicación. Esto se debe a la constante evolución de los sistemas de protección. Por su lado, los delincuentes, mejoran continuamente las técnicas y mecanismos para conseguir la información o acceso a sistemas para los que no deberían tenerlo.



Además en la seguridad informática se produce una situación muy contradictoria en la que confluyen tres características que son: La funcionalidad, la seguridad y la usabilidad. Como podemos ver en la figura están interrelacionadas, cuanto más seguro es un sistema, será a costa de su funcionalidad y su usabilidad que disminuirán. Lo mismo si lo que pretendemos es tener un sistema muy usable, penalizaremos de alguna manera su funcionalidad y seguridad.

Por tanto, se trata de llegar a un consenso entre todos los elementos involucrados en la seguridad de nuestro sistema informático.

>Des de el principio hay que entender que la seguridad absoluta y/o garantizada no existe, así que, nuestro objetivo debe ser conseguir unos niveles de seguridad tan altos como marquen nuestros objetivos. Según nuestras necesidades, requisitos y costes.

2.1. Conceptos de seguridad/fiabilidad

Existen tres principio básicos denominados la triada CIA, por sus siglas en inglés, respecto a la seguridad informática:

- Confidencialidad (en inglés Confidentiality): Es la característica que tiene un dato, información o mensaje que permite que solo sea entendible por el sistema o persona que está autorizado a comprender y utilizar su contenido.
- Integridad (en inglés Integrity): Es la propiedad que tiene un dato, información o mensaje que hace que se pueda asegurar que este no ha sufrido una modificación voluntaria o accidental del contenido original. Debe preservar su exactitud y completitud.
- Disponibilidad (en inglés Availability): Es la posibilidad que ofrecen los datos, información, mensajes o sistemas que permite que sean accesibles en todo momento

por cualquier usuario o sistema que esté autorizado para ello.

Además de estos, son importantes los servicios que nos proporciona:

- Autenticación (en inglés Authentication): Es el mecanismo que permite comprobar que un usuario o sistema es quien asegura ser. Se suele usar una combinación de al menos dos elementos de la siguientes factores:
    - Algo que sabes, que hemos elegido (por ejemplo: la contraseña de nuestro usuario, el pin de nuestra tarjeta de crédito, etcétera)
    - Algo que tienes, que nos han dado alguien (por ejemplo: una tarjeta de crédito, certificado digital, dispositivo móvil, etcétera)
    - Algo que eres, de nuestro cuerpo (por ejemplo: nuestras huellas dactilares, nuestro iris, nuestra cara, etcétera)
- Control de acceso (en inglés Access control): garantiza que el acceso a la información o sistemas está restringido en función de las necesidades de seguridad y de la empresa.
- No repudio (en inglés No repudiation): sirve para asegurar que un sistema/usuario ha participado en una comunicación, la elaboración de una información o acceso a la misma.
    - en origen: el emisor no puede negar que es él quien envió el mensaje.
    - en destino: el receptor no puede negar que recibió el mensaje.
- Trazabilidad (en inglés Traceability): registra la información sobre el uso o accesos que se han producido sobre los sistemas o la información objeto de observación. Almacena algo parecido a un historial con todo lo que ha ocurrido en el sistema y su información.

Los tres principios básicos, junto con los cuatro servicios que acabamos de nombrar forman los 7 pilares de la seguridad informática:

2.2. Tipos de seguridad

Según el momento en que se apliquen las medidas de seguridad podemos encontrar:

- Seguridad activa: Son medidas que se aplican antes de que ocurra el problema de seguridad, y por lo tanto medidas preventivas.

- Seguridad pasiva: Son medidas correctivas o que se aplican después de que ocurra el problema y estan orientadas a reestablecer el servicio de manera que se reduzca el impacto del problema de seguridad informática.

2.3. Elementos vulnerables y tipos de amenazas

Llamamos vulnerabilidad a las posibles debilidades que presenta un sistema o información y que pueden ser explotadas por alguna amenaza. Es decir, son los puntos débiles de nuestro sistema.

Por tanto, debemos tenerlas identificadas, conocer su importancia y definir medidas de seguridad que nos ayuden a controlarlas o evitarlas. Las vulnerabilidades, con ejemplos, pueden ser:

- físicas: instalaciones no adecuadas (temperatura, humedad, ubicación, cableado, protección contra incendios, etcétera)
- naturales: terremotos, huracanes, lluvias, inundaciones, etcétera.
- hardware: incorrecto mantenimiento de los equipos, no disponer de respaldo, etcétera.
- software: configuración incorrecta, falta de actualizaciones, etcétera
- de comunicaciones: usar comunicaciones no cifradas, permitir el acceso desde el exterior sin control, etcétera.
- de almacenamiento: defecto de fabricación, lugar no adecuado para almacenar soportes digitales, etcétera.
- humanas: contraseñas débiles, compartir usuarios y contraseñas, falta de formación, etcétera.

Las amenazas son cualquier tipo de evento o acción que pueda producir daño sobre el sistema o información. Pueden llegar a ocasionar un incidente de seguridad, que a su vez provoque un daño y cuyo riesgo de producirse sea significativo.

- incidente de seguridad: evento inesperado y no deseado que puede comprometer la seguridad.
- daño: el perjuicio que se produce al ocurrir una amenaza.
- riesgo: el producto del daño y la probabilidad de que esta amenaza se produzca.
- En cuanto a los tipos de amenazas, vamos a diferenciar entre:
    - como se producen:
        - Naturales: Provocadas por la naturaleza.
        - Involuntarias: Acciones inconscientes, accidentales o errores.
        - Intencionadas: Acciones deliberadas como: vandalismo, sabotaje, robo, etcétera.
    - como se comportan:
        - Activas: El atacante tiene como objetivo modificar la información o crear una nueva que hará pasar como original.
        - Pasivas: El atacante consigue acceso a la información, pero no la modifica.

2.4. Protección y estándares relacionados con la seguridad informática

Un modo de asegurar el sistema informático a nuestro cargo es realizar una auditoria de seguridad, que analizará las amenazas y/o riesgos que podemos sufrir y calcular un grado de importancia o probabilidad de que estas situaciones se produzcan.

>Auditoría de la seguridad informática: Para realizar una auditoria de seguridad debemos obtener autorización expresa del propietario del sistema informático a analizar. Consta de varias fases: enumeración, detección y evaluación de vulnerabilidades, Medidas de corrección y implantación de medidas de prevención.

En cuanto a los estándares relacionados con la seguridad informática, debemos referenciar la familia de normas ISO/iec 270000 así como la UNE/ISO 17799.

También deberemos considerar estas leyes y reglamentos: LOPD-GDD, LSSICE, RGPD. Disponemos de servicios de ayuda como el INCIBE, el CCN-CERT y el CSIRT.

3. Vulnerabilidades en el software.

Debido al auge de Internet, las vulnerabilidades de software son uno de los mayores problemas de la informática actualmente. Las aplicaciones actuales cada vez tratan con más datos (personales, bancarios, etc.) y por eso, cada vez es más necesario que las aplicaciones sean seguras.

Una vulnerabilidad de software puede definirse como un fallo o hueco de seguridad detectado en algún programa o sistema informático que puede ser utilizado para entrar en los sistemas de forma no autorizada.De una forma sencilla, se trata de un fallo de diseño de algún programa, que puede tener instalado en su equipo, y que permite a los atacantes realizar una acción maliciosa. Si nos centramos en una aplicación, es posible que ese fallo de seguridad permita a un usuario que acceda o manipule a una información a la que no esté autorizado.

Pero el gran problema de los agujeros de seguridad reside en la forma en que son detectados. Cuando una aplicación tiene una cierta envergadura, se encarga el trabajo de detectar fallos de seguridad a usuarios externos para que utilicen la aplicación normalmente y/o a expertos de seguridad para que analicen a fondo la seguridad de la aplicación.

A pesar de que antes de lanzar una aplicación al mercado se intentan descubrir y solucionar todos sus fallos de seguridad, en muchas ocasiones se descubren después y por lo tanto, es necesario informar a los usuarios y permitir que se actualicen a la última versión que se encuentre libre de fallos.

Para poder garantizar la seguridad de una aplicación en Java nos vamos a centrar en dos pilares:

- Seguridad interna de aplicación. A la hora de realizar la aplicación se debe programar de una forma robusta para que comporte tal y como esperamos de ella. Algunas de las técnicas que podemos implementar, es la gestión de excepciones, validaciones de entradas de datos y la utilización de los ficheros de registro.
- Políticas de acceso. Una vez que tenemos una aplicación "segura" es importante definir las políticas de acceso para determinar las acciones que puede realizar la aplicación en nuestro equipo. Por ejemplo, podemos indicar que la aplicación pueda leer los ficheros de una determinada carpeta, enviar datos a través de Internet a un determinado equipo, etc. De esta forma, aunque un usuario malicioso utilice la aplicación de forma incorrecta, se limita el impacto de su ataque. Así, si hemos indicado que sólo puede leer los ficheros de la carpeta c:/datos, el atacante nunca podrá modificar esos datos o leer los ficheros de otro directorio.

3.1. Programación defensiva

Las situaciones de error se producen por diferentes causas, como una *implementación incorrecta*, cuando el código no se ajusta a las especificaciones, una *petición de objeto inapropiada*, como un índice no válido (en un vector) o una referencia nula, o que el estado de un objeto sea inapropiado o inexistente.

Pero no siempre es debido a un error de programación. Pueden producirse errores en el entorno, como la introduccion errónea de los datos o que se produzca una interrupción en las comunicaciones; también, durante el procesamiento de ficheros, como que no exista el archivo al que deseamos acceder o que los permisos de acceso sean inadecuados.

Estos acontecimientos provocan que, para garantizar el correcto funcionamiento de las aplicaciones, deba utilizarse la técnica de la programación defensiva, que conlleva la incorporación de un gran número de comprobaciones, con la consiguiente merma en rendimiento.

Pero, ¿qué cosas deben ser comprobadas? ¿Debe tratarse el fallo en el punto en que se produce o se informa de los errores al código que invocó a aquél en que se ha producido? ¿cómo debe tratarse el error? Si el error no se trata, ¿cómo se informa del mismo? ¿Puede el código anticiparse al fallo? Y si lo hace, ¿cuándo?

Veamos un ejemplo sencillo. Supongamos que tenemos una agenda; identificamos las entradas en la misma mediante algún tipo de clave que podemos utilizar para localizarlas. Queremos borrar un registro: localizamos el mismo a través de su clave y, a continuación, lo eliminamos:

```
public void eliminar(String clave) {
    Contacto contacto = agenda.buscar(clave);
    agenda.borrar(contacto);
}
```

Se produce un error: por ejemplo, que el registro no existe. ¿Quién tiene la culpa? En general, es preferible anticiparse a quejarse.

3.1.1. Comprobación de clave

Los argumentos son una vulnerabilidad:

- Inicializan el estado en un constructor
- Contribuyen al comportamiento en un método

La comprobación de los argumentos es una medida defensiva:

```
public void eliminar(String clave) {
    Contacto contacto;
    if (agenda.existe(clave)) {
        contacto = agenda.buscar(clave);
        agenda.borrar(contacto);
    }
}
```

En este caso, comprobamos si existe el registro que deseamos eliminar antes de proceder al borrado.

3.1.2. Información del error

El modelo clásico para informar de que se ha producido un error es la devolución de un diagnóstico. Por ejemplo, el método puede ser implementado como una función booleana que devuelva verdadero solo si la operación se ha realizado con éxito:

```java
public boolean eliminar(String clave) {
    Contacto contacto;
    if (agenda.existe(clave)) {
        contacto = agenda.buscar(clave);
        agenda.borrar(contacto);
        return true;
    } else {
        return false;
    }
}
```

La salida de la función deberá ser comprobada por el método que la invocó:

```java
if (!objeto.metodo(parametros)) {
    // tratar error
} else {
    // código normal
}
```

Este es un método sencillo pero que complica el código innecesariamente debido a la anidación. Además, pueden producirse otros tipos de errores de ejecución que no son capturables.

Los lenguajes modernos, especialmente los que soportan la programación orientada al objeto, incorporan un mecanismo de gestión del error a través de las denominadas excepciones.

La estructura del código en este caso, como antes, tiene dos partes, la del método invocado, en el que se produce el error, y el que invoca, en el que el error puede ser tratado. En el primero, cuando se produce el error, bien sea porque se incumple alguna condición o porque hay algún error en el entorno, se indica la excepción y se finaliza la ejecución del método. En el segundo, se codifican dos partes diferenciadas: la secuencia de ejecución normal, por un lado, y el tratamiento del error, por el otro.

3.2. Excepciones

3.2.1. Lanzamiento de excepciones

Cuando se produce un error y se genera una excepción, se dice que ésta se lanza: *throw*. Cualquier código es susceptible de lanzar una excepción, sea el entorno de trabajo de la JVM, una biblioteca o tu propio código. Ya hemos comentado que una excepción será una instancia de alguna subclase de Throwable.

Cuando la excepción se produce, el método que está ejecutándose se detiene prematuramente y no devuelve valor alguno. Además, el control no vuelve al punto desde el que el método fue invocado: se dirige al código que gestiona la excepción o se re-lanza hacia un método anterior, que pueda gestionar el error. Por supuesto, resulta posible *capturarla* (tener un punto en el código para el tratamiento de la excepción) y no tratarla (gestionar el error); sería conveniente, en cualquier caso, proporcionar algún tipo de mensaje o notificación.

Pero, ¿cómo indicamos en nuestro código que se ha producido una excepción?

1. Construir un objeto de excepción. Una excepción, como hemos indicado es una clase; por tanto, usaremos el operador new.

2. El lanzamiento de la excepción se lleva a cabo con el operador throw. Lo habitual es realizar ambas operaciones en una única instrucción

```
throw new ExceptionType();
```

3. Documentar que puede producirse la excepción y por qué. Se incluye en la documentación Javadoc del método:

```
@throws ExceptionType causa_que_provoca_el_lanzamiento
```

Una excepción es un evento que ocurre durante de la ejecución de un programa e interrumpe el flujo normal de las instrucciones. Por ejemplo, si desea crear una aplicación que lea un archivo y lo envíe por la red,algunas de las posibles excepciones son: que el dispositivo donde se almacena el fichero no esté disponible, que no tenga permisos de lectura sobre el fichero, que el fichero no exista, que la red no esté disponible, etc.

3.2.2. Tratamiento de las excepciones

Es importante gestionar las posibles excepciones que puedan ocurrir en el programa para evitar cualquier tipo de fallos en su ejecución. De forma general, para incluir el manejo de excepciones en un programa hay que realizar los siguientes pasos:

1. Dentro de un bloque try inserte el flujo normal de las instrucciones.
2. Estudie los errores que pueden producirse durante la ejecución de las instrucciones y compruebe que cada uno de ellos provoca una excepción.
3. Capture y gestione las excepciones en bloques catch.

Cuando el programador ejecuta un código que puede provocar una excepción (por ejemplo: una lectura o escritura de un fichero), debe incluir este fragmento de código dentro de un bloque try:

```
try {
    // Código posiblemente problemático
}
```

Pero lo importante es cómo controlar qué hacer con la posible excepción que se genera. Para ello se utilizan las clausulas `catch` que permiten especificar la acción a realizar cuando se produce una determinada excepción.

```
try {
    // Código posiblemente problemático
} catch ( tipo_de_excepcion e) {
    // Código para solucionar la excepción e
} catch ( tipo_de_excepcion_mas_general e) {
    // Código para solucionar la excepción e
}
```

Como puede ver en el ejemplo, se pueden anidar sentencias catch para ir gestionando, de forma diferente, diferentes errores. Es conveniente hacerlo indicando en último lugar las excepciones más generales (es decir, que se encuentren más arriba en el árbol de herencia de excepciones), porque el intérprete Java ejecuta el bloque de código catch cuyo parámetro es el tipo de una excepción lanzada.

Si desea realizar una acción común a todas las excepciones se utiliza la sentencia finally. Este código se ejecuta tanto si se trata una excepción (catch) como sino.

```
try {
    // Código posiblemente problemático
} catch ( Exception e ) {
    // Código para solucionar la excepción e
} finally {
    // Se ejecuta tras try o catch
}
```

Ejemplo:

```
try {
    int a[] = new int[5];
    a[5] = 30 / 0;
} catch (ArithmeticException e) {
    System.out.println("Ha ocurrido un error aritmético");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Ha ocurrido un error de índice fuera de rango");
} finally {
    System.out.println("Este mensaje siempre lo veremos");
}
System.out.println("Esto se verá solo si el bloque completo sale sin excepción.");
```

3.2.3. Crear nuestras propias excepciones

Aunque Java proporciona una gran cantidad de excepciones, en algunas ocasiones necesitaremos crear excepciones propias.

Las excepciones propias deben ser **subclases de la clase Exception**.

Normalmente crearemos excepciones propias cuando queramos manejar excepciones no contempladas por la librería estándar de Java. Por ejemplo, vamos a crear un tipo de excepción llamado ValorNoValido que se lanzará cuando el valor utilizado en una determinada operación no sea correcto.

La clase podría ser la siguiente:

```
public class ValorNoValido extends Exception{

    public ValorNoValido(){
    }

    public ValorNoValido(String cadena){
        super(cadena); //Llama al constructor de Exception y le pasa el contenido de cadena
    }
}
```

Un programa para probar la excepción creada podría ser este:

```
public static void main(String[] args) {
```

```
    try {
        double x = leerValor();
        System.out.println("Raiz cuadrada de " + x + " = " + Math.sqrt(x));
    } catch (ValorNoValido e) {
        System.out.println(e.getMessage());
    }
}

public static double leerValor() throws ValorNoValido {
    Scanner sc = new Scanner(System.in);
    System.out.print("Introduce número > 0 ");
    double n = sc.nextDouble();
    if (n <= 0) {
        throw new ValorNoValido("El número debe ser positivo");
    }
    return n;
}
```

3.2.4. Consejos

- No usar excepciones para el flujo de control

  La técnica intenta separar claramente la ejecución del código normal del control del error.

- Las excepciones son excepcionales

  Se refiere a que sucede con poca frecuencia, cuando algún dato se encuentra fuera de su dominio, por ejemplo, y se produce el error.

- No lanzar ni capturar excepciones genéricas

  Cuanto más concretamente podamos determinar la causa del error, más apropiada será el control del mismo.

- No usar cláusulas catch vacías

  Si no es posible solucionarlo, es mejor propagarlo.

- Usar diferentes bloques try para sentencias que lanzan las mismas excepciones

  Ello permite definir mejor cuál es el origen del error.

- Utilizar excepciones estándar

  La bilbioteca de Java define excepciones para la mayoría de los errores que pueden producirse

- Utilizar excepciones cuando el constructor falla

  Si el objeto no puede crearse es imposible trabajar con el mismo

- Documentar las excepciones

  Siempre añadir la información pertinente en la documentación del método que la lance. Y si es una excepción de usuario, explicar claramente su cometido.

3.3. Registros o Logs (`Log4j2`)

El software con suficiente registro y supervisión le permitirá detectar posibles incidentes cuando su código se implemente en un entorno de producción. `Log4j` se utiliza para iniciar sesión. El registro es el proceso de escribir un mensaje de registro en cualquier archivo, base de datos, consola, etc.

Si usamos la instrucción SOP `System.out.print()` para imprimir el mensaje de registro, podemos tener algunas desventajas:

1. Solo podemos imprimir el mensaje de registro en la consola. Entonces, cuando la consola esté cerrada, perderemos todos los registros.
2. No podemos almacenar mensajes de registro en ningún lugar permanente. Estos mensajes se imprimirán uno por uno en la consola porque es un entorno de un solo subproceso.

Para superar estos problemas, entró en escena el marco `Log4j`. `Log4j` es un marco de código abierto proporcionado por Apache solo para proyectos Java.

Para usar Log4j2 en su marco, solo necesita agregar las siguientes [bibliotecas] (https://logging.apache.org/log4j/2.x/download.html):

```
log4j-api-<version>.jar
log4j-core-<version>.jar
```

3.3.1. Configuración automática

Podemos configurar Log4j2 con nuestra aplicación mediante un archivo de configuración escrito en formato XML, JSON, YAML o de propiedades. También podemos hacerlo mediante programación, pero centrémonos en configurar usando archivos de configuración a partir de ahora. Log4j tiene la capacidad de configurarse automáticamente durante la inicialización. Tiene una orden para buscar el archivo de configuración en la aplicación. Log4j proporcionará una configuración predeterminada si no puede localizar un archivo de configuración.

3.3.2. Niveles de registro

Los niveles de registro son un mecanismo para categorizar los registros. Niveles utilizados para identificar la gravedad de un evento. Podemos configurar niveles fácilmente para especificar qué detalles de registro queremos ver. Log4j proporciona los siguientes [niveles] (https://logging.apache.org/log4j/2.x/log4j-api/apidocs/org/apache/logging/log4j/Level.html):

1. TODO: para registrar todos los eventos.
2. DEBUG: un evento de depuración general.
3. ERROR – Un error en la aplicación, posiblemente recuperable.
4. FATAL: un error grave que impedirá que la aplicación continúe.
5. INFO – Un evento con fines informativos.
6. SEGUIMIENTO: un mensaje de depuración detallado, que normalmente captura el flujo a través de la aplicación.
7. WARN: un evento que posiblemente podría conducir a un error.
8. APAGADO: no se registrarán eventos.

Log4j sigue el orden de la siguiente manera:

TODO < TRAZA < DEPURACIÓN < INFORMACIÓN < ADVERTENCIA < ERROR < FATAL

Si mencionamos el nivel de registro como INFO, se registrarán todos los eventos INFO, WARN, ERROR y FATAL. Si mencionamos el nivel de registro como WARN, se registrarán todos los eventos WARN, ERROR y FATAL. En términos simples, se considerarán todos los niveles por debajo del nivel especificado, incluido el nivel especificado.

3.3.3. Anexadores

Podemos especificar destinos para mantener registros de eventos. Es posible que queramos imprimir esos registros en la consola o en cualquier archivo externo. Los agregadores generalmente solo son responsables de escribir los datos del evento en el destino objetivo. Podemos usar múltiples appenders.

3.3.4. Código de ejemplo

```
package UD05.Log4j2;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class ModuleA {

    private static final Logger logger = LogManager.getLogger();

    // Log messages
    public static void main(String[] args) {
        logger.debug("It is a debug logger.");
        logger.error("It is an error logger.");
        logger.fatal("It is a fatal logger.");
        logger.info("It is a info logger.");
        logger.trace("It is a trace logger.");
        logger.warn("It is a warn logger.");
    }
}
```

La salida será:

```
11:14:47.469 [main] ERROR UD05.Log4j2.ModuleA - It is an error logger.
11:14:47.471 [main] FATAL UD05.Log4j2.ModuleA - It is a fatal logger.
```

Hemos usado todos los niveles, pero en la consola solo vemos dos niveles. En realidad, cuando no proporcionamos ningún archivo de configuración, por defecto `Log4j` usa una configuración predeterminada.

La configuración predeterminada, provista en la clase `DefaultConfiguration`, establecerá:

- Un `ConsoleAppender` adjunto al registrador raíz, es decir, los registros se imprimirán en la consola.
- Un `PatternLayout` establecido en el patrón: "`%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n`" adjunto a `ConsoleAppender`

Tenga en cuenta que, por defecto, `Log4j` asigna el registrador raíz a `Level.ERROR` y esos registros se imprimirán en la consola estándar.

Comprendamos el formato de patrón en el que se imprimen los registros. Como no hemos pasado ningún archivo de configuración, utiliza el formato predeterminado como se muestra arriba. Visualicemos la salida con el patrón predeterminado.

%d{HH:mm:ss.SSS} es la marca de tiempo de ejecución, es decir, `18:07:15.984`. **[%t]** es el nombre del hilo, es decir, `[principal]`. **%-5level** es el nombre del nivel, es decir, `ERROR`. **%logger{36}** es el nombre del registrador que estamos creando como primer paso, es decir, `UD05.Log4j2.ModuleA`. **%msg%n** es un mensaje, es decir, "`Es un registrador de errores`" seguido de un carácter de nueva línea.

3.3.5. Configuración manual

Log4j tiene la capacidad de configurarse automáticamente durante la inicialización. Cuando se inicie Log4j, localizará todos los complementos de ConfigurationFactory y los organizará en orden ponderado de mayor a menor. Tal como se entrega, Log4j contiene cuatro implementaciones de ConfigurationFactory: una para JSON, una para YAML, una para propiedades y otra para XML.

1. Log4j inspeccionará la propiedad del sistema "log4j2.configurationFile" y, si está configurada, intentará cargar la configuración utilizando ConfigurationFactory que coincida con la extensión del archivo. Tenga en cuenta que esto no está restringido a una ubicación en el sistema de archivos local y puede contener una URL.
2. Si no se establece ninguna propiedad del sistema, las propiedades ConfigurationFactory buscarán log4j2-test.properties en el classpath.
3. Si no se encuentra dicho archivo, YAML ConfigurationFactory buscará log4j2-test.yaml o log4j2-test.yml en el classpath.
4. Si no se encuentra dicho archivo, JSON ConfigurationFactory buscará log4j2-test.json o log4j2-test.jsn en el classpath.
5. Si no se encuentra dicho archivo, XML ConfigurationFactory buscará log4j2-test.xml en el classpath.
6. Si no se puede ubicar un archivo de prueba, ConfigurationFactory de propiedades buscará log4j2.properties en el classpath.
7. Si no se puede ubicar un archivo de propiedades, YAML ConfigurationFactory buscará log4j2.yaml o log4j2.yml en el classpath.
8. Si no se puede ubicar un archivo YAML, JSON ConfigurationFactory buscará log4j2.json o log4j2.jsn en el classpath.
9. Si no se puede ubicar un archivo JSON, XML ConfigurationFactory intentará ubicar log4j2.xml en el classpath.
10. Si no se pudo localizar ningún archivo de configuración, se utilizará la configuración predeterminada. Esto hará que la salida de registro vaya a la consola.

Esta es nuestra muestra de archivo de configuración log4j2.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} -
%msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="trace">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

Ahora sabemos que Log4j2 buscará un archivo de configuración en el classpath de los proyectos. Para esto, debemos mantener un archivo de configuración en la carpeta de recursos. Realizaremos cambios en el archivo XML anterior para imprimir todos los niveles de registros en la consola. Debajo de la etiqueta "Loggers" tenemos otra etiqueta llamada "Root". Root Logger es el elemento superior en cada jerarquía de registradores. Esta etiqueta "Raíz" contiene una propiedad llamada "nivel". Hemos cambiado el nivel a "traza" para que se impriman todos los niveles.

Si añadimos este archivo a nuestro "src"/"resources" carpeta obtendremos esta salida:

```
11:29:06.786 [main] DEBUG UD05.Log4j2.ModuleA - It is a debug logger.
11:29:06.787 [main] ERROR UD05.Log4j2.ModuleA - It is an error logger.
11:29:06.787 [main] FATAL UD05.Log4j2.ModuleA - It is a fatal logger.
11:29:06.787 [main] INFO  UD05.Log4j2.ModuleA - It is a info logger.
11:29:06.787 [main] TRACE UD05.Log4j2.ModuleA - It is a trace logger.
11:29:06.787 [main] WARN  UD05.Log4j2.ModuleA - It is a warn logger.
```

También podemos establecer la ruta del archivo de configuración con la opción VM:

```
-Dlog4j.configurationFille=src/UD05/Log4j2/log4j2.xml
```

Hemos seguido esta guía:http://makeseleniumeasy.com/2021/03/11/log4j2-tutorial-1-introduction-to-apache-log4j2/ adn https://logging.apache.org/log4j/2.x/

3.4. Validación de entradas.

Una vía importante de errores de seguridad y de inconsistencia de datos dentro de una aplicación se produce a través de los datos que introducen los usuarios. Un fallo de seguridad muy frecuente, consiste en los errores basados en buffer overflow, se producen cuando se desborda el tamaño de una determinada variable, vector, matriz, etc. y se consigue acceder a zonas de memoria reservadas. Por ejemplo, si reservamos memoria para el nombre de usuario que ocupa 20 caracteres y, de alguna forma, el
usuario consigue que la aplicación almacene más datos, se está produciendo un buffer overflow ya que los primeros 20 valores se almacenan en un lugar correcto, pero los restantes valores se almacenan en zonas de memoria destinadas a otros fines.

La validación de datos permite:

- Mantener la consistencia de los datos. Por ejemplo, si a un usuario le indicamos que debe introducir su DNI éste debe tener el mismo formato siempre.
- Evitar desbordamientos de memoria buffer overflow. Al comprobar el formato y la longitud del campo evitamos que se produzcan los desbordamientos de memoria.

Para llevar un riguroso control, sobre los datos de entrada de los usuarios hay que tener en cuenta la validación del formato y la validación del tamaño de la entrada.

Java incorpora una potente y útil librería (`java.util.regex`) para utilizar la clase `Pattern` que nos permite definir expresiones regulares. Las expresiones regulares permiten definir exactamente el formato de la entrada de datos.

Ejemplo:

```
1  public class Validacion {
2
3      public static void main(String[] args) {
4          Scanner teclado = new Scanner (System.in);
5
6          Pattern pat = null;
7          Matcher mat = null;
8
9          pat=Pattern.compile("[0-9]{8}-[a-zA-Z]");
10         System.out.print("Introduce un DNI con formato 00000000-X: ");
11         mat=pat.matcher(teclado.nextLine());
12
13         if (mat.find()){
14             System.out.println("El DNI cumple el formato");
15         } else {
16             System.out.println("El DNI NO cumple el formato");
17         }
18
19         //Otros patterns de ejemplo:
20         //pat=Pattern.compile("Almería","Granada","Jaén","Málaga",
21         //          "Sevilla","Cádiz","Córdoba","Huelva");
22         //pat=Pattern.compile("Verdadero","Falso","V","F",
23         //          "True","False","Córdoba","Huelva");
24
25     }
26 }
```

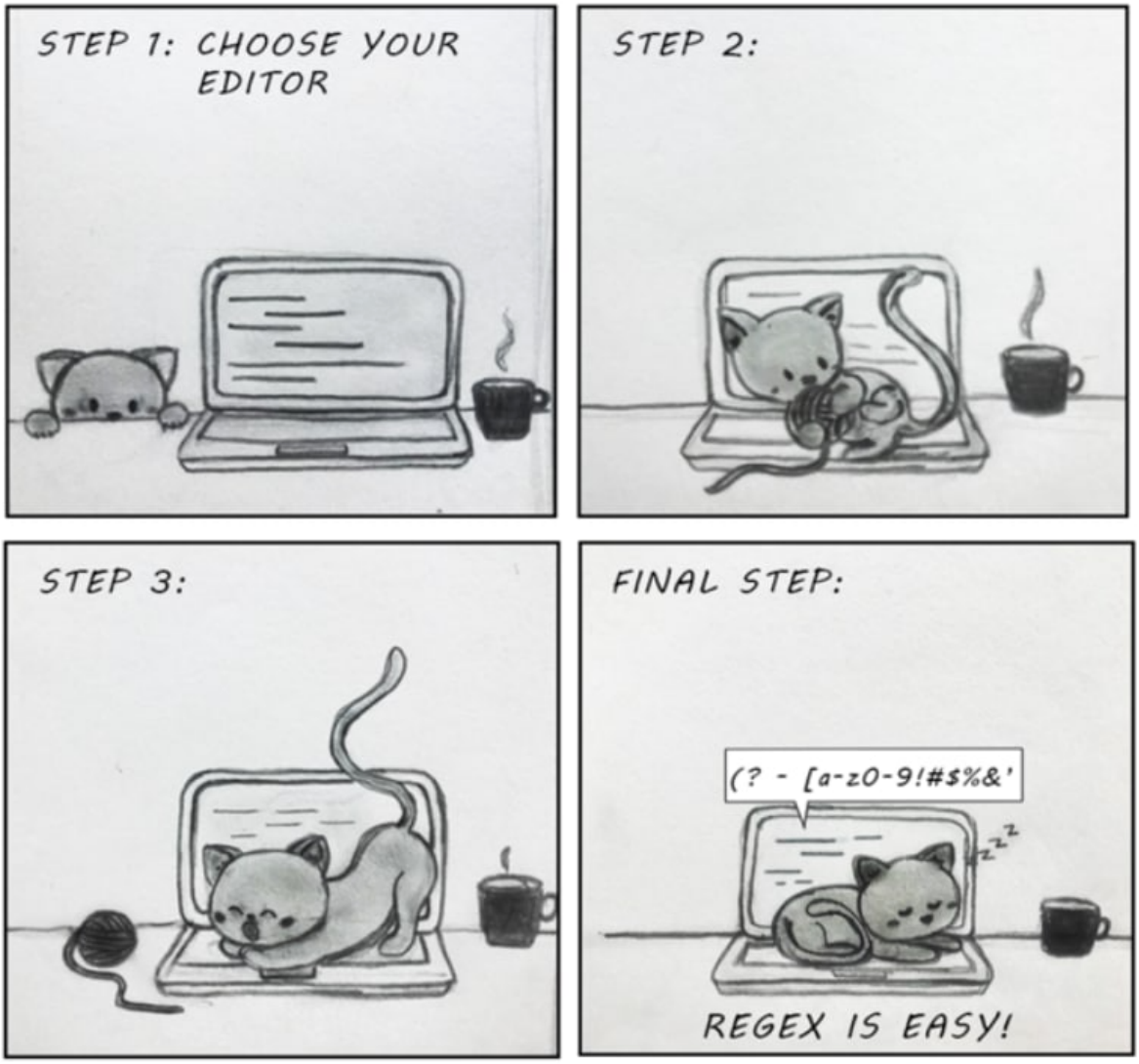


Figure 8.1 Letting your cat play over the keyboard is not the best solution for generating regular expressions (regex). To learn how to generate regexes you can use an online generator like https://regexr.com/.

3.5. Políticas de seguridad.

Se incrementa la seguridad del sistema permitiendo establecer las políticas de acceso tanto a las aplicaciones locales o remotas. Una vez que la aplicación es autorizada, se envían las peticiones al Security Manager para poder acceder a los recursos del sistema.

Las políticas de seguridad se pueden establecer utilizando los siguientes elementos:

- Origen (usuario o ruta de acceso de la aplicación).
- Permisos (por ejemplo, se puede indicar si tiene permisos de lectura/escritura sobre un fichero, si puede enviar o no información).
- Destino. Destino al que afecta el permiso. Por ejemplo si trabamos con ficheros el destino es su ubicación.
- Acción. Las acciones que se pueden realizar sobre el fichero. Por ejemplo, en un fichero los permisos son lectura o escritura.

Cabe destacar que **todo lo que se menciona aquí no funciona si el usuario tiene acceso al sistema y puede ejecutar el intérprete de Java sin restricciones**. Es decir, se necesita el trabajo de un administrador de sistemas para restringir la manera en la que el usuario ejecuta el código.

Supongamos entonces que estamos en un entorno seguro donde los programas Java se ejecutan utilizando un gestor de seguridad, es decir, se lanzan ejecutando `java -Djava.security.manager Clase`. En principio, los programas no podrán hacer muchas cosas, como por ejemplo, conectarse a Internet o leer un fichero que no esté en el mismo directorio de la clase.

3.6. Prohibido

Las cosas que no se deben hacer:

- Escribir código que usa nombres de ficheros relativos, deben usarse referencias completas
- Referir dos veces en un programa un mismo fichero por su nombre
- Invocar programas o servicios no confiables y/o obsoletos o sin mantenimiento
- Asumir que los usuarios no son maliciosos o no pueden actuar mal
- No realizar gestión de excepciones y asumir siempre el éxito
- Invocar un Shell o una línea de comandos desde la aplicación
- Autenticarse con criterios que no sean de confianza
- Usar áreas de almacenamiento con permisos de escritura (cuidado con esto)
- Guardar datos confidenciales en una base de datos sin contraseña o cifrado alguno
- Hacer eco de las contraseñas o mostrarlas en las vistas de usuario
- Emitir contraseñas o credenciales por correo electrónico (y olvidando el doble factor, etc.)
- Distribuir mediante programación alguna información confidencial por medio del email
- Guardar las contraseñas (o cualquier otra información sensible) sin cifrar
- Distribuir contraseñas sin encriptar entre los sistemas (o cualquier información sensible)
- Tomar decisiones de acceso según variables o argumentos en tiempo de ejecución
- Confiar en exceso en software de terceros para las operaciones críticas

4. Fuentes de información

- Wikipedia
- Programación de servicios y procesos - FERNANDO PANIAGUA MARTÍN [Paraninfo]
- Programación de Servicios y Procesos - ALBERTO SÁNCHEZ CAMPOS [Ra-ma]
- Programación de Servicios y Procesos - Mª JESÚS RAMOS MARTÍN - [Garceta] (1ª y 2ª Edición)
- Programación de servicios y procesos - CARLOS ALBERTO CORTIJO BON [Síntesis]
- Programació de serveis i processos - JOAR ARNEDO MORENO, JOSEP CAÑELLAS BORNAS i JOSÉ ANTONIO LEO MEGÍAS [IOC]
- GitHub repositories:
    - https://github.com/ajcpro/psp
    - https://oscarmaestre.github.io/servicios/index.html
    - https://github.com/juanro49/DAM/tree/master/DAM2/PSP
    - https://github.com/pablohs1986/dam_psp2021
    - https://github.com/Perju/DAM
    - https://github.com/eldiegoch/DAM
    - https://github.com/eldiegoch/2dam-psp-public
    - https://github.com/franlu/DAM-PSP
    - https://github.com/ProgProcesosYServicios
    - https://github.com/joseluisgs
    - https://github.com/oscarnovillo/dam2_2122
    - https://github.com/PacoPortillo/DAM_PSP_Tarea02_La-Cena-de-los-Filosofos