Desarrollo de contendido

IU Digital de Antioquia
INSTITUCIÓN UNIVERSITARIA DIGITAL DE ANTIOQUIA

Unidad 3

Programación Orientada a Objetos II

ORD No 74 de 2017 · VIGILADA MinEducación

Unidad 3. Gestión de errores y otras características.

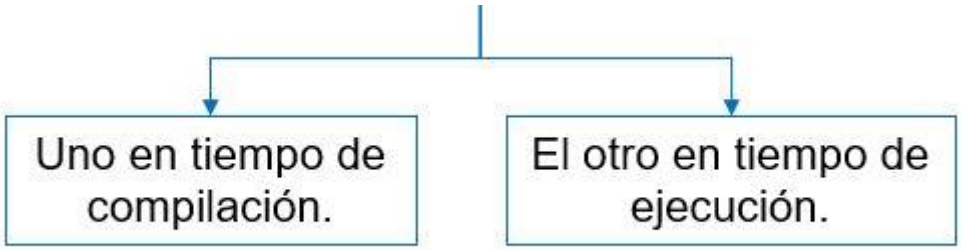
Java se considera un leguaje de compilación, y se pueden presentar errores cuando se está desarrollando o al momento de ejecutar los desarrollos. Los lenguajes de programación están creados de tal manera que puedan detectar estos errores; cuando se tienen errores en tiempo de ejecución, estos se presentan debido a situaciones inesperadas y no a errores de programación, y estos errores siempre se presentarán y su gestión está dada por medio de excepciones fundamentales en cualquier lenguaje de programación actual.

En la presente Unidad, estudiarás:

Gestión de errores.
Concurrencia.
Persistencia.
Recogiendo la basura.
Conexión Base de Datos JAVA.

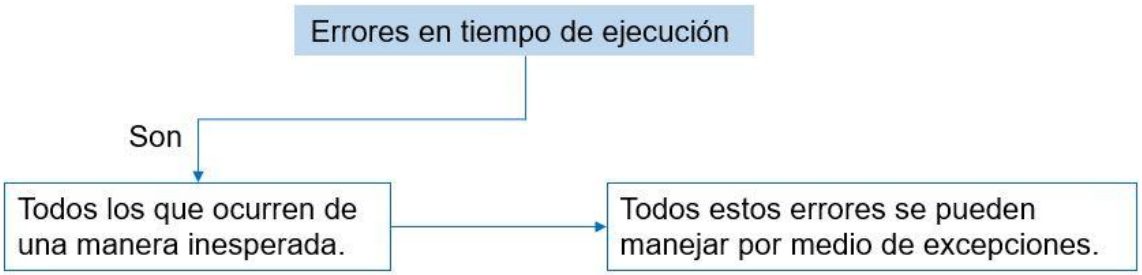
Tema 1. Gestión de errores

Java es considerado un lenguaje de compilación, y en el momento de estarlo desarrollando se pueden presentar dos tipos de errores:



NOTA: Los lenguajes de programación están diseñados de tal manera que puedan detectar la máxima cantidad posible de errores. Los errores de tiempo, en tiempo de ejecución, se presentan debido a situaciones inesperadas y no a errores de programación; en los errores en tiempo de ejecución, siempre se presentarán y su gestión está dada por medio de excepciones fundamentales en cualquier lenguaje de programación actual.

Errores en tiempo de ejecución: excepciones



Sabías que…

Existen también errores que se presentan debido a tareas multihilo y que se evidencian en tiempo de ejecución, y no todos pueden ser controlados. Un ejemplo claro es cuando se tiene un bloque entre hilos y se requiere controlar; para ello se debe añadir algún mecanismo que permita detectar estas situaciones y elimine los hilos que no correspondan con este proceso.

De seguro que te estarás preguntando:

¿Qué son las excepciones?
Se consideran eventos que se presentan durante la ejecución de un programa y lo que hacen es que se salga de su flujo normal de instrucciones. Con este proceso se busca que los errores sean tratados de una manera elegante, separando el código, para poder realizar el tratamiento de los errores del código normal del programa (Interpolados, 2017).

NOTA: Estas excepciones son lanzadas cuando se produce un error y pueden ser capturadas para poder realizar el tratamiento de estos errores.

Tipos de excepciones

Existen varios tipos de excepciones que dependen de la clase de error que fue generado; todas estas excepciones son heredadas de la clase Throwable, y esta tiene dos descendientes. Haz clic en cada una para conocerlas.

Error
Hace referencia a errores graves que se presentan en la máquina virtual de Java. Por lo general, estos errores no son tratados en los programas Java (Interpolados, 2017).

Exception
Aquí se representan errores que no son considerados críticos y, por lo tanto, no se pueden tratar, pero se permite continuar la ejecución de la aplicación. En la mayoría de los programas Java se hace uso de estas excepciones, permitiendo el tratamiento

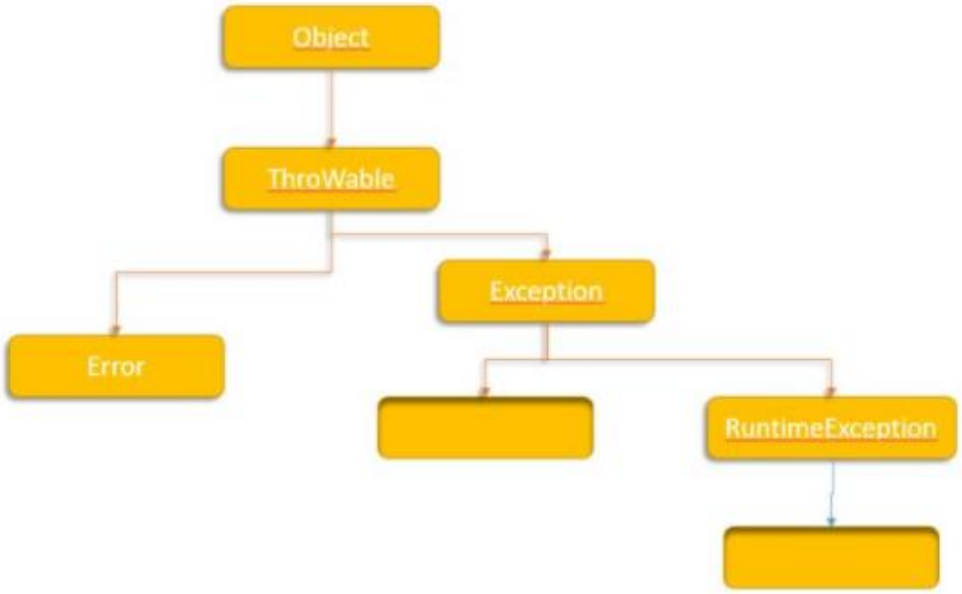
de los errores que pueden ocurrir en tiempo de ejecución del código (Interpolados, 2017).

Los tipos de excepciones guardan la información relacionada con el tipo de error al que hacen referencia y la información común a todas las excepciones.

NOTA: En Exception, encontramos una subclase especial de excepciones que se denomina RuntimeException, la cual hereda todas las excepciones que hacen referencia a errores comunes y que pueden producirse en cualquier fragmento de código. La clase RuntimeException es diferente a todas las demás excepciones en donde no se tiene el tipo Checked; estas excepciones son capturadas de manera obligatoria, y si no se llegasen a realizar, entonces se presenta un error en la compilación del código. Se deben añadir manejadores de excepciones y declarar a estas excepciones unchecked, es decir, no predecibles, para que nos indiquen los errores graves presentados en la lógica de la programación, lo que no debería ocurrir. Estos manejadores se utilizan para comprobar la consistencia interna que tienen los programas (Interpolados, 2017).

Se tiene una excepción checked que permite capturar errores que se presentan en el momento en que se ejecuta una aplicacion, debidos a factores externos.

La representación gráfica sería así:



Captura de excepciones

Para capturar una excepción, debemos hacer uso de la estructura que está conformada por 3 bloques de códigos.

Try: En el try se tiene el código regular del programa que puede generar una excepción en caso de error (Interpolados, 2017).

Catch: En el catch tenemos el código con el que vamos a tratar el error encontrado. Debemos especificar, para el bloque catch, el tipo o grupo de excepciones que vamos a trabajar; aquí se pueden incluir varios bloques al tiempo, cada uno para un grupo o tipo de excepción determinado (Interpolados, 2017).

Finally: Este bloque es el que tiene el código que se ejecutará una vez se finalice el proceso, sea porque se generó o no, una excepción; este bloque no es obligatorio ponerlo (Interpolados, 2017).

Ejemplo

```
Try {
//Código regular del programa
//Puede producir excepciones
}
 catch (TipoDeExcepcion1 el) {
//Código que trata las excepciones de tipo
//TipoDeExcepcion1 o subclases de ella
//Los datos sobre la excepción los encontraremos en el objeto él.
}
 catch (TipoDeEscepcion2 e2) {
//Código que trata las excepciones de tipo
//TipoDeEscepcion2 o subclases de ella
.//Los datos sobre la excepción los encontraremos en el objeto e2....
}
catch (TipoDeExcepcionN En) {
// Código que trata las excepciones de tipo
//TipoDeExcepcionN o subclases de ella.
//Los datos sobre la excepción los encontraremos en el objeto En.
}
```

```
Finally {
//Código de finalización (opcional)
}
```

Bibliografía

Interpolados. (2017, enero 27). Excepciones en Java. Recuperado de: https://interpolados.wordpress.com/2017/01/27/excepciones-en-java/

Cuando no se especifica un tipo o grupo de excepción, entonces se captura cualquier excepción, debido a que la subclase es común a todas las excepciones. En el bloque catch, podemos hacer uso de algunos métodos propios de la excepción.

Si tomamos como ejemplo estos dos métodos tenemos:

String getMessage()
En el método getMessage, se obtiene una cadena de texto que contiene la descripción del error, si se tiene error.

void printStackTrace()
El método printStackTrace nos muestra la salida estándar de todos los errores que se presentaron.

En el siguiente ejemplo podemos ver el llamado de estos métodos en la sentencia excepción.

```
try {

 ... // Aqui va el codigo que puede lanzar una excepcion

} catch (Exception e) {

 System.out.println ("El error es: " + e.getMessage());

 e.printStackTrace();

}
```

NOTA: El bloque catch, no debe dejarse vacío, porque si un error se produce, entonces no sabríamos qué error se presentó, sobre todo cuando tenemos excepciones no-checked.

Lanzar excepciones

En el método donde se va a lanzar la excepción, se deben seguir estos pasos, los cuales permiten que se ejecute una excepción cuando así se requiera.

Lo primero que debemos determinar en el método, es el tipo o grupo de excepciones que queremos lanzar, y esto se hace siguiendo el ejemplo a continuación:

```
public void lee_fichero()
 throws IOException, FileNotFoundException
{
 // Cuerpo de la función
}
```

Luego, indicaremos la cantidad de excepciones que queremos en la cláusula throws con la sentencia throw new IOException(mensaje_error); aquí se debe proporcionar el objeto correspondiente a la excepción que estamos lanzando, y esto lo hacemos en el ejemplo a continuación, donde unimos el paso anterior:

```
public void lee_fichero()
 throws IOException, FileNotFoundException
{
 ...
 throw new IOException(mensaje_error);
 ...
}
```

De esta manera lanzamos nuestras excepciones en nuestro código, indicando que no se tiene el comportamiento que se pretende que sea ejecutado.

Crear nuevas excepciones

También podemos crear nuestras propias excepciones si así es requerido. Para crearlas, debemos generar una clase que hereda de Exception o cualquier otra clase de subtipos de excepciones que ya existen; aquí agregamos nuestros atributos, métodos y propiedades que nos permitan guardar la información que hace relación al error que se presenta. Miremos este ejemplo de cómo podemos crear una nueva excepción.

```
public class NuevaExcepcion extends Exception
{
 public NuevaExcepcion (String mensajeE)
 {
 super(mensajeE);
 }
}
```

NOTA: **También se pueden crear subclases del nuevo tipo de excepción creado, y así podemos generar grupos de excepciones, si queremos hacer uso de estas excepciones, y entonces hacemos el mismo procedimiento descrito anteriormente.**

Nested Exceptions

Por lo general, si en un método de una librería se produce una excepción, esta se reproduce al llamador en lugar que se gestione este método en la librería, para que de esta manera el llamador tenga claro que se produjo un error y pueda tomar las medidas necesarias en el momento justo.

Cuando en esta excepción se requiere pasar al nivel súper, se debe propagar la excepción que llegó o puede crear y lanzar una nueva excepción; cuando hablamos

de una nueva excepción, entonces debemos contener la excepción anterior, ya que necesitamos tener la información de la causa original del error que se produjo en la librería; este nos servirá para depurar la aplicación.

La excepción que se produjo se debe enviar como parámetro del constructor de la nueva clase.

```
public class MiExcepcion extends Exception
{
 public MiExcepcion (String mensaje, Throwable causa)
 {
 super(mensaje, causa);
 }
}
```

En el método de la librería que se esté trabajando y en el que se produce el error, se deberá capturar la excepción generada por el error y lanzar la excepción al llamador, y para eso usamos el siguiente código:

```
try {
 ...
} catch(IOException e) {
 throw new MiExcepcion("Mensaje de error", e);
}
```

Una vez capturamos la excepción, se puede consultar por medio de las siguientes líneas de código:

```
Exception causaE = (Exception)e.getCause();
```

Las nested exceptions son muy útiles para:

1. Poder encadenar errores que son producidos en las secuencias de métodos que han sido llamados.
2. Para facilitar la depuración de las aplicaciones, permitiendo conocer de dónde proviene el error y por cuál método ha pasado.
3. Se lanza una excepción propia por cada método que permite ofrecer información mucho más detallada, en vez de utilizar una única excepción genérica.
4. Aísla el llamador de la implementación específica de una librería.

Errores en tiempo de compilación

En el lenguaje JAVA se tienen los errores de compilación y las advertencias; cuando se habla de advertencia decimos que no son obligatorias, mientras que los errores sí, porque no permiten que el codificador pueda compilar el código.

Es recomendable que no se dejen advertencias en nuestro código, porque estas pueden indicarnos que se está presentando algún tipo de incorrección.

Cuando trabajamos con Eclipse, podemos revisar todos los errores y advertencias conforme vamos escribiendo nuestro código. Es necesario que se configure en el menú Preferences / Java / Compiler / Errors.

Los errores en tiempo de compilación se clasifican en: (haz clic sobre los nombres)

Errores de sintaxis
Cuando se está programando y el código tecleado no cumple con las reglas sintácticas propias del lenguaje Java, decimos que tenemos errores de sintaxis.

Errores semánticos

Cuando el código sintácticamente no cumple con las reglas de más alto nivel, decimos que tenemos errores semánticos.

Con el siguiente ejemplo podemos entender este tipo de errores:

public ***void funcion()***

{

int ***a;***

Console.println(a);

}

Prueba.java:12: variable a might not have been initialized

Console.println(a);

1 error

Errores en cascada

No son considerados tipos de error pero sí confunden al compilador, y devuelven un mensaje que puede indicar la causa del error, diferente al origen real del mismo.

Para conocer un ejemplo donde puedes observar que está mal escrita la sentencia ***for,*** haz clic en el botón.

Ejemplo

```
fo ( inti = 0; i < 4; i++ )
{
}
Prueba.java:24: '.class' expectedfo ( int i = 0; i < 4; i++ )
^
Prueba.java:24: ')' expectedfo ( int i = 0; i < 4; i++ )
^
Prueba.java:24: not a statementfo ( int i = 0; i < 4; i++ )
^
Pueba.java:24: ';' expectedfo ( int i = 0; i < 4; i++ )
^
Prueba.java:24: unexpected typerequired: valuefound : classfo ( int i = 0; i < 4; i++ )
^
```

Prueba.java:24: cannot resolve symbolsymbol :variable ilocation: class Pruebafo (int i = 0; i < 4; i++)6 errors

Comprobación de tipos: tipos genéricos

En java existen muchas estructuras de datos que están preparadas para almacenar cualquier objeto; por ejemplo, en vez de tener un ArrayList que recibe y devuelve un número entero, este recibe y devuelve un objeto. Cuando se devuelve un objeto, lo que se realiza es un reparto explícito, es decir:

```
Integer i =(Integer)v.get(0);
```

En este caso, el programador sabe que este conjunto solo puede tener un entero, pero el problema está en que el elenco no es correcto, entonces se produce un error en el tiempo de ejecución.

Con este ejemplo podemos entender mejor:

```
List v = new ArrayList();

v.add("test");

Integer i = (Integer)v.get(0); // Error en tiempo de ejecución
```

Para poder evitar todo esto, a partir de versión 1.5 de Java se incorporaron los tipos genéricos, cuya función es forzar a devolver el tipo devuelto, pero solo en la declaración de la clase de la instancia. Los tipos int y float, no deben ser utilizados en los tipos genéricos.

Por tanto, tenemos:

```
List<String> v = new ArrayList<String>();

v.add("test");

String s = v.get(0); // Correcto (sin necesidad de cast explícito)
```

Integer i = v.get(0); // Error en tiempo ce compilación

Para poder definir que una clase trabaje con un tipo genérico, se debe añadir un identificador que estará entre los símbolos menor y mayor <>, al final de la clase.

Con este ejemplo puedes ver cómo se representan estos tipos genéricos, y se utilizan las interfaces List e Iterator:

```
public interface List<E> {

 void add(E x);

 Iterator<E> iterator();

}

public interface Iterator<E> {

 E next();

 boolean hasNext();

}
```

La clases también pueden tener tipos genéricos y se sigue la misma sintaxis

Si queremos declarar objetos de la clase genérica que creamos anteriormente, debemos indicar el tipo concreto que se trabajará por cada caso:

- Entry<String, String> grade440 = new Entry<String, String>("mike", "A");
- Entry<String, Integer> marks440 = new Entry<String, Integer>("mike", 100);
- System.out.println("grade: " + grade440);

- System.out.println("marks: " + marks440);

También podemos definir los métodos con tipos genéricos:

- public static <T> Entry<T,T> twice(T value) {
- return new SimpleImmutableEntry<T,T>(value, value);

En este método se utiliza el tipo genérico que nos indica qué genéricos tiene la clase que el método devuelve y si se puede utilizar este mismo tipo genérico para indicar de qué tipo es el argumento del método. Cuando se usa el método, este tipo puede ser indicado o inferido por el compilador, en vez de ser declarado.

Entry<String, String> pair = this.<String>twice("Hola"); // Declarado

Entry<String, String> pair = twice("Hola"); // Inferido

Subtipos y comodines

Si tenemos una clase hija denominada subtipo (subclase o subinterfaz) de una clase padre, por ejemplo ArrayList que es una clase genérica, entonces ArrayList<Hija> no es subtipo de ArrayList<Padre>.

Se pueden flexibilizar estos tipos genéricos por medio de "wildcards" o comodines. Para hacer que funcionen para tipos y subtipos, entonces se hace uso de comodín <?> unidos a la clave extends, y de esta manera se indica cual es la clase/interfaz de la que hereda:

ArrayList<? extends Padre>

Es así como se dice que es válido Padre como su clase derivada Hija.

Es posible que se indique a un método o a una clase, el tipo de excepción que debe lanzar, por medio de un genérico. Como ejemplo tenemos:

```
public <T extends Throwable> void metodo() throws T {
 throw new T();
}
```

O también:

```
public class Clase<T extends Throwable>
 public void metodo() throws T {
 throw new T();
 }
}
```

Lo que no se puede hacer es crear excepciones a partir de tipos genéricos, es decir, crear nuestra propia excepción para poder incluir distintos tipos:

```
public class MiExcepcion<T extends Object> extends Exception {
 private T someObject;


 public MiExcepcion(T someObject) {
 this.someObject = someObject;
 }
```

```
 public T getSomeObject() {

 return someObject;

 }

}
```

Se puede presentar un problema con las cláusulas catch, ya que cada una corresponde a un tipo determinado:

```
try {

 //Código que lanza o bien MiExcepcion<String>, o bien MiExcepcion<Integer>

}

catch(MiExcepcion<String> ex) {

 // A

}

catch(MiExcepcion<Integer> ex) {

 // B

}
```

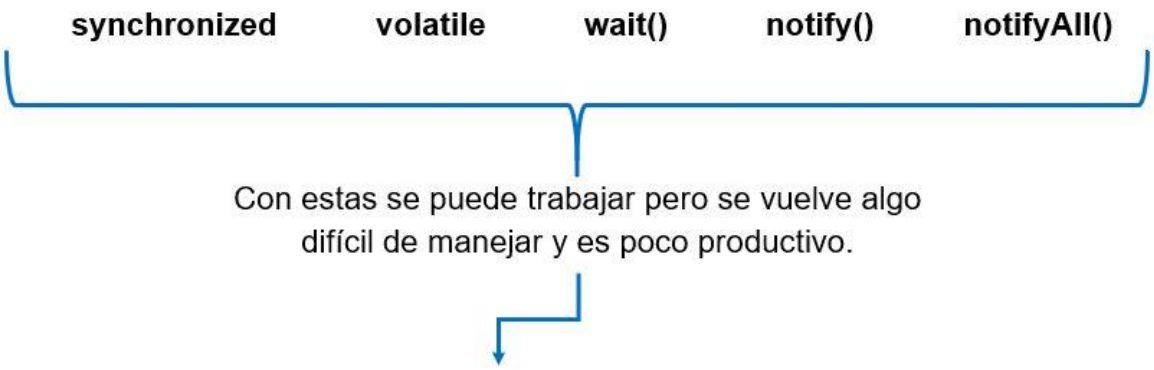
Se observa en este código que no sería posible identificar en qué bloque catch se debe entrar, debido a que después de la compilación, estos serían idénticos debido al borrado de tipos, o "type erasure":

```
try {
```

```
 //Código que lanza o bien MiExcepcion<String>, o bien MiExcepcion<Integer>
}
catch(MiExcepcion ex) {
 // A
}
catch(MiExcepcion ex) {
 // B
}
```

Concurrencia

Java tiene incluido el manejo de concurrencia, como:



Fue así como se creó el Java Specification Request (JSR) 166; cuyo objetivo principal es proporcionar un conjunto de bloques de construcción de alto nivel que permite manejar la concurrencia; estos pueden ser:/p>

NOTA: El paquete java.util.concurrent es uno de los aportes fundamentales y principales de JSR; con este paquete se puede manejar concurrencia, y con estos elementos estándar aquellos programas JAVA que manejan concurrencias son mucho más confiables y escalables pero ante todo son mucho más fáciles de entender.

Las características para el manejo de concurrencia se agrupan en:

1. Nuevo marco de trabajo.
2. Colecciones concurrentes.
3. Candados de alto desempeño
4. Sincronizadores
5. Variables atómicas
6. Mejoras en el JVM

Dentro de las concurrencias encontramos los Executors y las colecciones concurrentes.

Executor

El nuevo marco de trabajo estábasado en la interfaz Executor; quees un objeto que ejecuta las tareas Runnable. De acuerdo con el Executorque fue implementado, estas tareas se ejecutan en un hilo que fue recientemente creado o en alguno que ya existía, y esto se ejecuta de manera secuencial o concurrente. Una de las diferencias que más se resaltan a la hora de utilizar este marco, es que sesepara el lanzamiento de la tarea y las políticas de ejecución. Por ejemplo, en vez de usar la sentencia:new Thread(algunRunnable).start();para lanzar una tarea,ahora utilizamos:miExecutor.execute(algunRunnable);La interfaz ExecutorServiceextiende a Executory proporciona un marco de ejecución con tareas asíncronas más completo; este maneja colas, programación de tareas, y tiene la terminación

controlada de estas (controlled shutdown), permitiendo manejar el ciclo de vida completo de cada hilo de ejecución. La interfaz ScheduledExecutorServicepermite agregar un soporte en la ejecución de las tareas periódicas y/o con retraso.La clase Executorsprovee los métodos de fábrica (Factory Methods) a través de los cuales se pueden crear los tipos y configuraciones más comunes de Executors. Otros dos elementos de este marco de trabajo son las interfaces Callabley Future. Una interfazCallablees la analogía de unaRunnable,la interfaz tradicionalmente utilizada en Java para clases ejecutadas porhilos,pero mejorado, ya que su ejecución puede regresar un resultado, o arrojar una excepción. Un Futurerepresenta una referencia a una tarea asíncrona, sin importar si esta ya ha sido ejecutada, se encuentre en ejecución, o apenas esté programada para su próxima ejecución. Es útil cuando uno o más hilos deben esperar a que cierto resultado se produzca antes de continuar con su trabajo. (Java 5. Manejo de concurrencia.

Colecciones concurrentes

El paquete java.util.concurrentpermite definir colecciones que facilitan el acceso de manera concurrente, tales como ConcurrentHashMap. Cuando utilizamos HashTableo Collections.synchronizedMap, observamos que estos elementos son seguro antehilos ya que todos sus métodos son synchronized, dando como resultado que múltiples hilos operen al mismo tiempo sobre ellos.

Este acceso se da de manera secuencial, por lo que parte de este código puede convertirse en un cuello de botella.Pero si usamos las clases como ConcurrentHashMapno solo tenemos hilos seguros, sino que también tenemos un alto acceso concurrente, lo que nos indica que diferentes hilos pueden traslapar sus operaciones al mismo tiempo, sin que se necesite esperar a un candado.

También tenemos la cola (queue); todas las implementaciones de cola que encontramos en la librería java.util.concurrentestán diseñadas para que sus operaciones seanconcurrentes. Supongamos que tenemos la clase ConcurrentLinkedQueue, y esta tiene implementada una cola tipo FIFO (first in, first out) por medio de un algoritmo libre de espera, lo que permite que se traslapen operaciones de inserción y eliminación.

Ahora, también existe una interfaz BlockingQueue, que permite definir una cola bloqueante, muy utilizada para evitar que las colas crezcan demasiado y consuman demasiados recursos.La mayoría de los elementos de java.util.concurrentno hacen uso de synchronized. De ahí la pregunta sobre como podemos tener hilos seguros; aquí ya tenemos otro nuevo concepto y es la utilización de una nueva interfaz que se llama Lock; este tiene una semántica muy similar con synchronized, con la diferencia que tiene un mayor desempeño y contiene características adicionales como interrumpir un hilo que opera por bloqueos; esperar a un bloqueo por un tiempo determinado, preguntar si hay disponibilidad de un bloqueo entre muchas otras características.

A pesar que el uso de los mecanismos generales de los métodos y las sentencias synchronized hace mucho más fácil utilizar bloqueos y evita muchos errores comunes, hay ocasiones en que necesitamos trabajar con los bloqueos de una forma más flexible. Por ejemplo, algunos algoritmos para el recorrido de estructuras de datos accedidas de manera concurrente, requierenel uso de bloqueos en cadena o mano sobre mano (hand-over-hand): se bloquea el nodo A, luego el nodo B, luego libera A, entonces adquiere C, entonces libera B y adquiere D, y así sucesivamente.

Las implementaciones de la interfaz Lockhabilitan el uso de tales técnicas permitiendo a un bloqueo ser adquirido y liberado en diferentes niveles, así como en cualquier orden. De la misma manera que esto incrementa la flexibilidad, también trae responsabilidades adicionales. La ausencia de estructuras de bloqueo elimina la liberación automática de estos, lo cual provoca que las estructuras para trabajar con este tipo de bloqueos tengan que ser manipuladas a mano. (Java 5. Manejo de concurrencia.)La siguiente sintaxis debe ser utilizada para estos fines:Lock l =...;l.lock();try { // accedemos al recurso protegido por este bloqueo }Finally{ l.unlock(); }

Cuando el bloqueo y desbloqueo ocurren en diferentes niveles, debemos tener cuidado para asegurarnos que todo el código que es ejecutado mientras el bloqueo está activo, sea protegido por un bloque try-finallyo try-catchy así asegurarnos que el bloqueo se libere cuando sea necesario. Las implementaciones de Lockproveen funcionalidad adicional, tal como un intento no bloqueante de adquirir un bloqueo (tryLock()),un intento de adquirir un bloqueo que pueda ser interrumpido

(lockInterruptibly())y el intento de adquirir un bloqueo que puede caducar (tryLock(long, TimeUnit)).(Java 5. Manejo de concurrencia.)

Persistencia

Es la propiedad que posee un objeto por medio de la cual su existencia trasciende en el tiempo y el espacio.

Nos indica que un objeto persistente continúa existiendo aún después que el programa del cual es originario ya finalizó, y que este se movió de localización de memoria en la que se creó.
Está presente en aquellos objetos en los que la aplicación así lo requiera, puesto que de otra manera se tendrían muchos objetos innecesarios; la persistencia es posible cuando se almacena en un dispositivo secundario de almacenamiento, la información que se requiere de un objeto para luego ser consultada.

NOTA.
Java permite la serialización de objetos por medio de un flujo de bytes, que luego pueden ser escritos a un archivo en disco, leídos y deserializados para reconstruir el objeto original; de aquí obtenemos lo que se conoce como "persistencia ligera" (lightweight persistence en inglés).

Para lograr escribir un objeto a un archivo, es necesario realizar esta serie de pasos:

Se debe crear una instancia de la clase ***FileOutputStream,*** y enviar al constructor una cadena de caracteres como argumento. Esta cadena debe contener el nombre del archivo que se quiere crear; si encontramos que el archivo ya existe, entonces debemos borrar el archivo original y luego crear uno nuevo con el mismo nombre. El constructor envía una instancia de la clase ***IOException*** si por alguna razón el archivo no fue creado.

Luego se crear una instancia de la clase ***ObjectOutputStream***, en donde se debe enviar al constructor un argumento, esto se da por medio de la instancia de la clase ***FileOutputStream*** que fue creada en anterior punto. El constructor debe lanzar una instancia de la clase ***IOException*** si se presenta algún error.

Se debe invocar el método ***writeObject()*** en la instancia de la clase ***ObjectOutputStream*** del anterior punto; se debe enviar como argumento el objeto que se quiere escribir en el archivo. Este objeto es instanciado de alguna clase que implemente la interfaz ***Serializable***. El método ***writeObject()*** arroja las siguientes excepciones:

- ***InvalidClassException***. Existe algún error con la clase que se desea serializar.
- ***NotSerializableException***. El objeto a ser serializado no implementa la interfaz Serializable.
- I***OException***. Se produjo algún error al escribir al archivo.

Luego, se debe cerrar el objeto de la clase *ObjectOutputStream*; de esta manera se tiene garantizada que la información sea almacenada completamente en el disco; el proceso se puede ejecutar de manera implícita si hacemos uso de la sentencia ***try-con-recursos,*** y si se realiza de manera explícita, entonces se debe hacer invocando el método ***close()***; este método lanza una instancia de la clase ***IOException*** si se encuentra algún error en el momento de cerrar el archivo.

El siguiente código ejemplifica todos los puntos anteriores.

```
try (FileOutputStream fos = new FileOutputStream("datos.bin");

 ObjectOutputStream oos = new ObjectOutputStream(fos)) {

 // Objeto que se desea hacer persistente.

 Integer x = new Integer(5);

 oos.writeObject(x);

} catch (Exception e) {

 e.printStackTrace();

}
```

Para la mayoría de las aplicaciones, almacenar y recuperar información implica alguna forma de interacción con una base de datos relacional. Esto ha representado un problema fundamental para los desarrolladores, porque algunas veces el diseño de datos relacionales y los ejemplares orientados a objetos, comparten estructuras de relaciones muy diferentes dentro de sus respectivos entornos.

Recogiendo la basura

Cuando se termina de eyectar un programa, no solo se deben inicializar las variables, también se debe limpiar la memoria, ya que los objetos podrían no ser eliminados por el recolector de basuras.

Algunas veces tendemos a pensar que un objeto es liberado una vez que terminamos de ejecutar el programa, pero siempre la biblioteca permite que el objeto sea liberado cuando se culmine con este. Hay muchas cosas que el recolector de basura no realiza o no tiene a cargo ejecutar.

Un ejemplo, es el caso de aquellos objetos a los que se les asignó memoria sin la utilización de la sentencia new; en este caso, se debe hacer uso de la función finalize() la cual se puede utilizar en cada clase, realizando un trabajo en conjunto, pues el recolector de basura llama a este y en el momento en que se finaliza, este también se encarga de limpiar el espacio relacionado con algo importante; el recolector limpiará el resto a la vez.

Finalize () y su utilidad

Este método se relaciona con la asignación en memoria; este método es requerido de manera explícita en aquellos casos en donde se desea reservar memoria de almacenamiento en un objeto que es creado sin utilizar la sentencia new(). En estos casos especiales, son aquellos que se dan por medio de métodos nativos, es decir código no-Java.

Funciones de limpieza

Como se observa, finalize() está diseñado para limpiezas de memoria no comunes, que de hecho casi no se usarán, por lo que es necesario crear funciones de limpieza. Sin embargo, finalize() brinda una funcionalidad aparte, denominada verificación de la condición de muerte de un objeto. Una vez que un objeto se desee eliminar (sea innecesario), este debe estar en un estado que permita liberar su memoria de manera segura; el caso de un fichero abierto, debe cerrarse antes de ser eliminado por el corrector de basura y, si no se cierra, puede acarrear un fallo luego que no se elimina completamente el objeto, lo cual no será sencillo de supervisar. No basta con inicializar una variable, no hay que olvidar limpiar la memoria muchas veces. Los objetos podrían no ser eliminados por el recolector de basura. Recogiendo la basura 3 Finalize() ayuda a encontrar el fallo y descubrir el problema. (Caldas, 2013)

Ejemplo:

```
class Comic{ boolean comprobado = false;
Comic (boolean comprobar){comprobado = comprobar;
}
void correcto (){ comprobado = false;
 }
public void finalize()
{ if (comprobado) System.out.println("Error: comprobado");
}
}
public class CondicionMuerte {public static void main (String[] args)
 {Comic aventura = new Comic(true) ;
// Eliminación correcta: aventura. correcto () ;
// Cargarse la referencia, olvidando la limpieza:new Comic (true) ;
// Forzar la recolección de basura y finalización: System.gc ();
}
}
```

En Java, el espacio se asigna de manera contigua; como los recursos no son infinitos, entonces la memoria puede acabarse y se presenta un error de paginación excesiva. Aquí aparece el recolector, que constantemente moverá el puntero del montículo, y reorganiza los elementos. En la siguiente tabla encontramos algunos conceptos que nos ayudan a entenderlo mejor.

Técnica	Descripción	Rendimiento
Contar referencias.	Se tiene un contador de referencias que incrementa cuando se adjunta una referencia de un objeto. Si una referencia se pone en *null*, se decrementa el contador.	Es lento y es una carga constante aunque pequeña, que se produce durante la vida del programa. No se implementa en ninguna máquina virtual de Java.
Adaptativo	Los objetos vivos se pueden recorrer realizando una traza y encontrando los objetos vivos. Cuando los encuentra puede hacer diversas cosas, entre ellas: **parar y copiar**. Copia el objeto de un montículo a otro (requiere cambiar todas las referencias que apuntan a este objeto), dejando detrás toda la basura. Otro esquema es **marcar y barrer** donde cada vez que se encuentra un objeto vivo, lo marca con un indicador, luego de acabar de marcar todo lo que puede, barre los objetos muertos.	Más rápido. Al requerir dos montículos y manejar memoria del traslado entre estos, consumen demasiada memoria.

Conexión base de datos Java

Si queremos realizar una conexión a la base de datos por medio de un servicio Java, tenemos que hacer uso de la API JDBC (Java DataBase Connectivity) de Java.

Para poder hacer el llamado de una base de datos en JAVA,hacemos uso de este código:

```
Class.forName("org.postgresql.Driver");

Connection connection = DriverManager.getConnection("jdbc:postgresql:

//hostname:port/dbname","username", "password");

...connection.close();
```

Lo primero para hacer es instanciar el driver para que se auto registre en el Driver Managery luego se invoca el método get Connectioncon el que se obtiene la conexión, y luego finalizamos con el cierre de la conexión. Con un objeto de tipo Connection instanciamos por medio de la sentencia Statement, y con esta podemos enviar sentencias SQL. Si queremos hacer una conexión Web, entonces procedemos a ejecutar la siguiente sentencia. Partimos desde el punto que el servicio está configurado de esta manera en el descriptor de despliegue:

```
<!--database example -->

<servlet>

<servlet-name>

example-db

</servlet-name>

<servlet-class>

org.fao.unredd.portal.ExampleDBServlet

</servlet-class>

</servlet>

<servlet-mapping>
```

<servlet-name>

example-db</servlet-name>

<url-pattern>

/example-db

</url-pattern>

</servlet-mapping>

y que luego se implementará su funcionalidad en el método GET:

package org.

Fao.unredd.portal;

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public class ExampleDBServlet extends HttpServlet {private static final long serialVersionUID = 1L;

@Overrideprotected void doGet(HttpServletRequest req, HttpServletResponse resp)throws ServletException, IOException

{

//

...

}

}

Para evitar crear una conexión cada vez que hacemos una ejecución, debemos configurar el contenedor de aplicaciones; un ejemplo de esto es usar Tomcat, que permite gestionar las conexiones por nosotros. Para ello, se le debe pasar a Tomcatla información necesaria que permita conectarse modificando dos ficheros. El primer fichero context.xmlque existe en el directorio de configuración del servidor conf.en el que se declara un recurso que se ha llamado “jdbc/mis-conexiones” el cual incluye todos los datos necesarios para poder hacer la conexión.

<Resource name="jdbc/mis-conexiones" auth="Container"

type="javax.sql.DataSource"driverClassName="org.postgresql.Driver"

url="jdbc:postgresql://192.168.0.18:5432/geoserverdata"username="nfms"

password="unr3dd" maxActive="20" maxIdle="10"maxWait="-1" />

El otro fichero a modificar es el descriptor de despliegue web-fragment.xml del plugin que estamos desarrollando, donde añadiremos una referencia al recurso anterior,

“jdbc/mis-conexiones”:<resource-ref><description>Application database</description>

<res-ref-name>

jdbc/mis-conexiones

</res-ref-name><restype>javax.sql.DataSource</res-type><res-

auth>Container</res-auth></resource-ref>

Una vez estos ficheros han sido modificados ya no tenemos que preocuparnos de realizar la conexión porque Tomcatlas gestiona por nosotros.

Pero, ¿cómo podemos obtener una de estas conexiones gestionadas por Tomcat? El código Java cambia ligeramente, ya que ahora se obtiene un objeto de tipo java.sql.DataSourceque es el que nos proporciona las conexiones:

InitialContext

```
context;

DataSource dataSource;

try {context = new InitialContext();

dataSource = (DataSource) context.lookup("java:/comp/env/jdbc/mis-conexiones");

}

 catch (NamingException e) {throw new ServletException("Problema en la configuración");

}

try {Connection connection = dataSource.getConnection();

// ...connection.close();

}

 catch (SQLException e) {throw new ServletException("No se pudo obtener una conexión");

}

try {context.close();

}

 catch (NamingException e) {// ignore}
```

Al remplazar la línea que contiene los puntos suspensivos por el código que realizará la conexión, podemos devolver un JSON con el arrayde nombres que contiene en una tabla:

```
package org.fao.unredd.portal;import java.io.IOException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;
```

```
import java.util.ArrayList;

import javax.naming.InitialContext;

import javax.naming.NamingException;

<res-ref-name>

jdbc/mis-conexiones

</res-ref-name>

<res-type>

javax.sql.DataSource

</res-type><res-auth>

Container

</res-auth>

</resource-ref>
```

Una vez estos ficheros han sido modificados ya no tenemos que preocuparnos de realizar la conexión porque Tomcatlas gestiona por nosotros. Pero, ¿cómo podemos obtener una de estas conexiones gestionadas por Tomcat? El código Java cambia ligeramente, ya que ahora se obtiene un objeto de tipo java.sql.DataSourceque es el que nos proporciona las conexiones :

```
InitialContext context;

DataSource dataSource;

try

{

context = new InitialContext();

dataSource = (DataSource) context.lookup("java:/comp/env/jdbc/mis-conexiones");

}

 catch (NamingException e)
```

```
 {
throw new ServletException("Problema en la configuración");
}
try
{
Connection connection = dataSource.getConnection()
;
// ...connection.close();
}
 catch (SQLException e) {throw new ServletException("No se pudo obtener una conexión");
}
try {context.close();
}
 catch (NamingException e)
 {
// ignore
}
```

Al remplazar la línea que contiene los puntos suspensivos por el código que realizará la conexión, podemos devolver un JSON con el arrayde nombres que contiene en una tabla:

```
package org.fao.unredd.portal;
import java.io.IOException;
import java.sql.Connection;
import java.sql.ResultSet;
```

```
import java.sql.SQLException;

import java.sql.Statement;

import java.util.ArrayList;

import javax.naming.InitialContext;

import javax.naming.NamingException;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.sql.DataSource;import net.sf.json.JSONSerializer;

public class ExampleDBServlet extends HttpServlet {private static final long serialVersionUID = 1L;

@Overrideprotected void doGet(HttpServletRequest req, HttpServletResponse resp)throws ServletException, IOException {InitialContext context;DataSource dataSource;

Try

 {

context = new InitialContext();dataSource = (DataSource) context.lookup("java:/comp/env/jdbc/mis-conexiones");

}

catch (NamingException e) {throw new ServletException("Problema en la configuración");

}

ArrayList<String> provincias = new ArrayList<String>();

try
```

```
{
Connection connection = dataSource.getConnection();
Statement statement = connection.createStatement();
ResultSet result = statement.executeQuery("SELECT name_1 FROM gis.arg_adm1");
while (result.next()) {provincias.add(result.getString("name_1"));
}
resp.setContentType("application/json");
JSONSerializer.toJSON(provincias).write(resp.getWriter());
connection.close();
}
 catch (SQLException e) {throw new ServletException("No se pudo obtener una conexión", e);
}
Try
 {
context.close();
}
 catch (NamingException e) {throw new ServletException("No se pudo liberar el recurso");
}
        }
                }
```

Bibliografía

Curso desarrollo. (s.f.). Conexión a base de datos en Java. Recuperado de: https://snmb-desarrollo.readthedocs.io/en/4.0.1/howtos/database-connection.html

La clase DBUtils
La referencia a esa conexión está configurada en el web-fragment.xml de core, que todo plugindebe incluir como dependencia y,por tanto, todo pluginpuede utilizar:

```
<resource-ref>
<description>
Application database
</description>
<res-ref-name>
jdbc/unredd-portal
</res-ref-name>
<res-type>javax.sql.DataSource</res-type><res-auth>Container</res-auth></resource-ref>
```

Como se puede observar, el nombre es “jdbc/unredd-portal” por lo que,con esta información, y usando la clase DBUtilsvista anteriormente, sería posible reescribir el servletanterior de la siguiente manera y sin tocar ningún fichero de configuración:

```
package org.fao.unredd.portal;
import java.io.IOException;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;
```

```
import net.sf.json.JSONSerializer;
public class ExampleDBServlet extends HttpServlet {private static final long serialVersionUID = 1L;
@Overrideprotected void doGet(HttpServletRequest req, HttpServletResponse resp)throws ServletException, IOException {final ArrayList<String> provincias = new ArrayList<String>();try {DBUtils.processConnection("unredd-portal", new DBUtils.DBProcessor() {@Overridepublic void process(Connection connection) throws SQLException
 {
Statement statement = connection.createStatement();
ResultSet result = statement.executeQuery("SELECT name_1 FROM gis.arg_adm1");
while (result.next()) {provincias.add(result.getString("name_1"));
}
}
}
);
}
 catch (PersistenceException e)
{
throw new ServletException("No se pudo obtener una conexión", e);
}
resp.setContentType("application/json");
JSONSerializer.toJSON(provincias).write(resp.getWriter());
}
}
htmlStatement statement = connection.createStatement();ResultSet result = statement.executeQuery("SELECT name_1 FROM gis.arg_adm1");
while (result.next()) {provincias.add(result.getString("name_1"));
}
}
}
);
}
 catch (PersistenceException e) {throw new ServletException("No se pudo obtener una conexión", e);
```

```
}
resp.setContentType("application/json");
JSONSerializer.toJSON(provincias).write(resp.getWriter());
}
}
```

Bibliografía

Curso desarrollo. (s.f.). Conexión a base de datos en Java. Recuperado de: https://snmb-desarrollo.readthedocs.io/en/4.0.1/howtos/database-connection.html

IU Digital de Antioquia

INSTITUCIÓN UNIVERSITARIA DIGITAL DE ANTIOQUIA

ORD No 74 de 2017 · VIGILADA MinEducación

www.iudigital.edu.co