

INTRODUCCIÓN A LA PROGRAMACIÓN

Semana 9

ING. MARTIN POLIOTTO

Docente a cargo del módulo

Septiembre 2020



DIPLOMATURA EN

**NUEVAS
TECNOLOGÍAS**

SECRETARÍA DE
EXTENSIÓN
UNIVERSITARIA
1974 - 1982

SEU

UTN
Facultad Regional Córdoba

Ministerio de
PROMOCIÓN DEL EMPLEO
Y DE LA ECONOMÍA FAMILIAR

Ministerio de
CIENCIA Y
TECNOLOGÍA

CBA
ENTRE TODOS

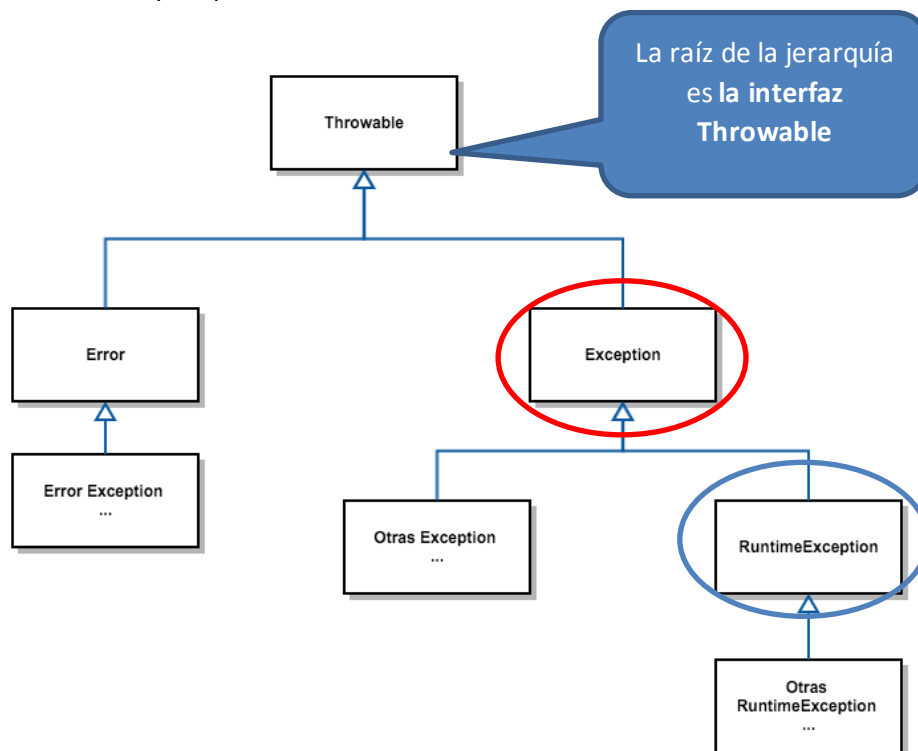
Oficina de Montevideo
Organización
de las Naciones Unidas
para la Educación,
la Ciencia y la Cultura

Semana 09

1. Control de errores

1.1 Excepciones

En Java, un error en tiempo de ejecución provocado por circunstancias anormales (puntero nulo invocando a un método, operaciones matemáticas no permitidas, etc.) se representa como un objeto. Java genera automáticamente objetos que representan a un error en tiempo de ejecución, permitiendo que el programador pueda controlar esa situación y eventualmente recuperarse de ella, incluso sin que la aplicación finalice de forma abrupta. Esos objetos reciben el nombre de **excepciones**. En la Api de Java las excepciones se modelan mediante una amplia jerarquía de clases que puede resumirse:



- Los errores representados por la clase **Error** son errores graves de hardware o de sistema frente a los cuales no se espera que el programador pueda hacer nada más que darse por notificado del hecho.
- Los errores representados por la clase **Exception** son errores comunes de programación que pueden requerir que el programador se vea obligado a escribir código de respuesta para esas excepciones, pues de lo contrario el programa no compilará. En ese sentido, las excepciones pueden clasificarse en **chequeadas** y en **no chequeadas**.
- En general, todas las clases de excepción que *derivan de la clase **Exception*** son chequeadas, salvo las que derivan a su vez de la clase **RuntimeException**.

1.2 Bloque try/catch/finally

El **try/catch** en programación se utiliza para manejar fragmentos de código que son propensos a fallar como es el caso de acceder a un archivo en disco o parsear una cadena a número. Por ejemplo al intentar convertir una cadena en un número podemos obtener un error de parseo como se muestra a continuación:

```
try{  
    // Intenta convertir un valor de tipo String a int  
    int numero = Integer.parseInt("xx909");  
} catch(Exception e){  
    System.out.println("ERROR: número incorrecto!");  
}
```

Consideraciones:

- Si el código que está dentro del try falla, se ejecuta el catch y el programa se sigue ejecutando. Dentro del try se debe de colocar el código que es propenso a fallar y dentro del catch el código a ejecutarse si el try falla, como puede ser un mensaje de error.
- Puede haber más de una sentencia que genere excepciones, en cuyo caso se puede definir más de un **catch**, dependiendo del tipo de excepción a manejar.
- Los tipos de excepciones se evalúan en cada bloque catch dependiendo de los tipos de excepciones definidos en cada uno de estos bloques.
- Siempre se evalúa desde el caso más particular hasta el más general, es decir, hasta capturar objetos **Exception**.
- El bloque que capture objetos de tipo Exception debería colocarse siempre como último caso
- Por último el bloque **finally**, es el bloque de código que se ejecuta siempre, haya o no excepción. Por ejemplo, podría servir para hacer un log o un seguimiento de lo que está pasando, porque como se ejecuta siempre puede dejar grabado si se producen excepciones y si el programa se ha recuperado de ellas o no.

1.3 Palabras clave **throw/throws**

Siempre que un bloque de código dentro de un método puede fallar, entonces tenemos las siguientes alternativas:

- Si la excepciones no es chequeada simplemente no hacer nada (apropiado aunque muy poco recomendado)
- Si la excepción es chequeada entonces para poder compilar:

- a) Tratar localmente el error dentro del método mediante un bloque try/catch/finally
- b) Declarar en la firma del método que es posible obtener una excepción de un tipo específico. La palabra clave **throws** se utiliza para identificar la lista posible de excepciones que un método puede lanzar. Si un método es capaz de provocar una excepción que no maneja él mismo, debería especificar este comportamiento, para que todos los métodos que lo llamen puedan colocar protecciones frente a esa excepción.

Java también permite crear y lanzar manualmente una excepción mediante la palabra reservada **throw**.

En primer lugar se debe obtener un descriptor de un objeto Throwable, bien mediante un parámetro en una cláusula catch o, se puede crear utilizando el operador new. La forma general de la sentencia throw es:

throw ObjetoThrowable;

El flujo de la ejecución se detiene inmediatamente después de la sentencia throw, y nunca se llega a la sentencia siguiente. Se inspecciona el bloque try que la engloba más cercano, para ver si tiene la cláusula catch cuyo tipo coincide con el del objeto creado. Si lo encuentra se transfiere el control a dicho código, caso contrario se lanza hacia el método de llamada.

2. Acceso a Bases de Datos con Java

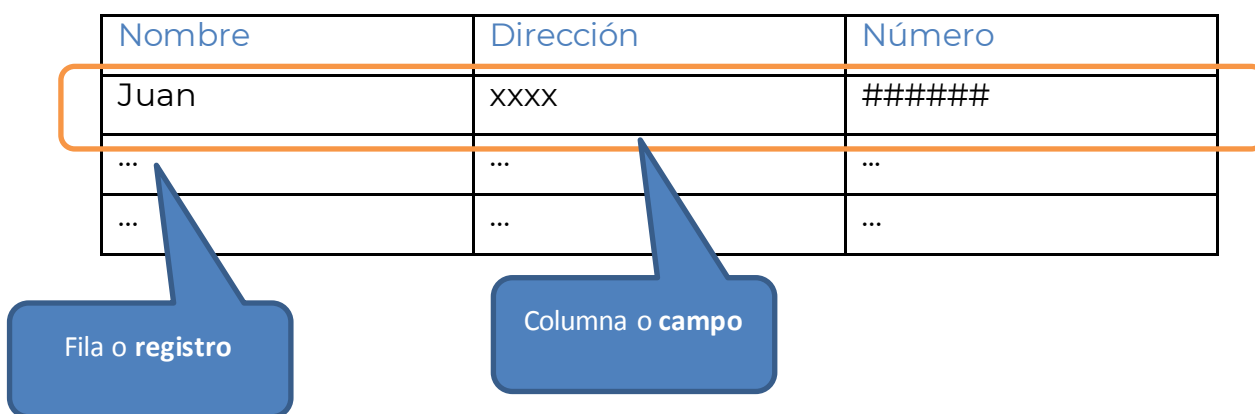
2.1 Concepto de Base de datos

Una base de datos es **una colección de información organizada** de forma que un programa de ordenador pueda seleccionar rápidamente los fragmentos de datos que necesite. Una base de datos es un sistema de archivos electrónico.

Las bases de datos tradicionales se organizan por **campos**, **registros** y **tablas**. Un campo es una pieza única de información; un registro es un sistema completo de campos; y una tabla es una colección de registros.

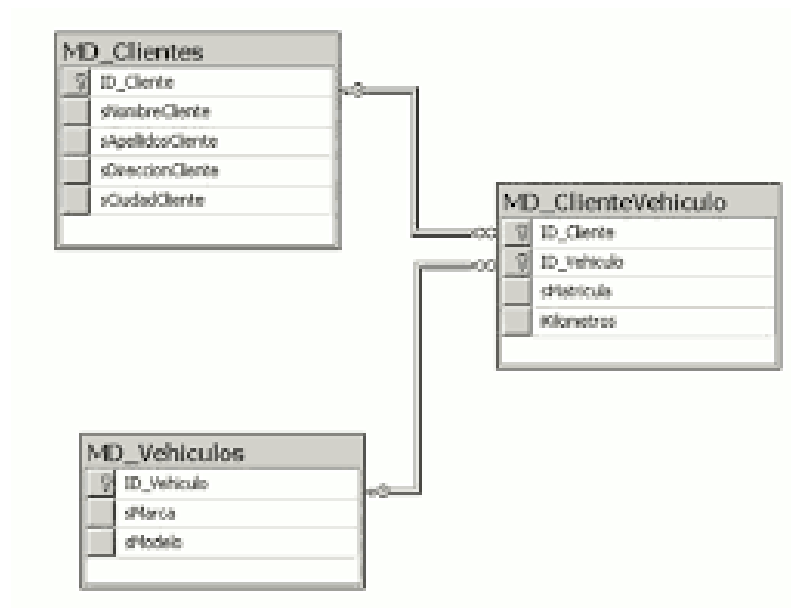
Por ejemplo, una guía de teléfono es análoga a una tabla. Contiene una lista de registros, cada uno de los cuales consiste en tres campos: nombre, dirección, y número de teléfono.

Nombre	Dirección	Número
Juan	xxxx	#####
...
...



Note que desde este punto de vista una base de datos puede resultar simplemente una planilla de cálculo o un archivo de Access. Existen otros programas más específicos que dan soporte a bases de datos, tales como: MySql, Oracle, DB2, SQL Server o Informix. Estos productos representan la información como un

conjunto de tablas y relaciones entre ellas, tal como muestra el siguiente gráfico:



2.2 SQL – Lenguaje Estructurado de Consulta

La programación SQL se puede usar para compartir y administrar datos, en particular la información organizada en tablas que se encuentra bases de datos relacionales.

El nombre **SQL** significa **Lenguaje de Consulta Estructurado (Structured Query Language)**. SQL es un lenguaje de bases de datos global: cuenta con sentencias para definir datos, consultas y actualizaciones. Además, dispone de características para definir vistas en la base de datos, especificar temas de seguridad y autorización, definir restricciones de integridad, y especificar controles de transacciones (control sobre los datos).

Existen dos tipos de comandos SQL:

- **DDL:** permiten crear y definir nuevas bases de datos, campos e índices.
 - ✓ **CREATE:** Crea nuevas tablas, campos e índices.
 - ✓ **DROP:** Elimina tablas e índices.
 - ✓ **ALTER:** Modifica las tablas agregando campos o cambiando la definición de los campos.
- **DML:** permiten generar consultas para ordenar, filtrar y extraer datos de la base de datos.
 - ✓ **SELECT:** Consulta registros de la base de datos que satisfagan un criterio determinado.
 - ✓ **INSERT:** Carga lotes de datos en la base de datos en una única operación.
 - ✓ **UPDATE:** Modifica los valores de los campos y registros especificados.
 - ✓ **DELETE:** Elimina registros de una tabla de una base de datos.

En el alcance de este módulo solo se abordarán comandos SELECT que permitan recuperar datos mediante una librería estándar de acceso a base de datos desde Java (JDBC).

2.3 JDBC

JDBC o Java DataBase Connectivity es la API (librería) estándar de acceso a base de datos desde Java. Está incluida en Java SE (*Standard Edition*). En Java SE 6 se incluye JDBC 4.0, pero actualmente la mayoría de bases de datos soportan JDBC 3.0 y JDBC 4.0.

Para conectarse a una base de datos concreta, es necesario su **driver JDBC**. El driver es un fichero JAR que se añade a la aplicación como cualquier otra librería (no necesita instalación adicional). La mayoría de las bases de datos incorporan un driver JDBC

2.3.1 Derby

Apache Derby es un sistema de base de datos relacional escrito en Java que puede ser empotrado en aplicaciones Java y utilizado para procesos de transacciones en línea. Tiene un tamaño de 2 MB de espacio en disco. Apache Derby es un proyecto open source disponible en virtud de la *Apache 2.0 License*. Actualmente se distribuye como Sun Java DB.

Este producto viene totalmente integrado con Netbeans por lo que no es necesario instalar ningún software adicional.

2.3.1 Primera conexión con Derby

A continuación se muestran los pasos necesarios para poder conectarse con el motor Derby y ejecutar un primer comando SQL para obtener todos los registros de usuarios de una base de prueba llamada Test.

```
public class Ejecutable {  
  
    public static void main(String[] args) {  
  
        try {  
  
            //1. Registrar el driver:  
  
            Class.forName("org.apache.derby.jdbc.ClientDriver");  
  
            //2. Obtener una conexión a Derby:  
  
            Connection                                cnn                                =  
            DriverManager.getConnection("jdbc:derby://localhost:1527/DBTest", "test", "test");  
  
            //3. Crear una sentencia:  
  
            Statement stmt = cnn.createStatement();  
  
            //ejecutar la sentencia y obtener un conjunto de resultados:  
  
            ResultSet rs = stmt.executeQuery("SELECT * FROM Usuarios");  
  
            //4. Recorrer filas:  
  
            while(rs.next()){  
  
                //5. Acceder a cada campo de la fila:  
  
                int id = rs.getInt("id_usuario");  
  
                String usuario = rs.getString("nombre");  
  
                String pass = rs.getString("password");  

```

```
System.out.println("ID:" + id + " |Usuario: " + usuario + " "  
+ "|Password: " + pass);
```

```
}
```

//6. Liberar recursos:

```
rs.close();
```

```
stmt.close();
```

//7. cerrar conexión

```
cnn.close();
```

```
}catch(ClassNotFoundException cnf){
```

```
System.out.println("Driver no encontrado!");
```

```
}
```

```
catch (SQLException e) {
```

```
System.out.println("No se pudo conectar con DERBY");
```

```
}
```

```
}
```

```
}
```

Pasos:

1. Carga del driver

- ✓ Antes de poder conectarse a la base de datos es necesario cargar el driver JDBC
- ✓ Sólo hay que hacerlo una única vez al comienzo de la aplicación

- ✓ Se puede elevar la excepción **ClassNotFoundException** si hay un error en el nombre del driver o si el fichero .jar no está correctamente en el **CLASSPATH** o en el proyecto

2. Establecer la conexión

- ✓ Las bases de datos actúan como servidores y las aplicaciones como clientes que se comunican a través de la red
- ✓ Un objeto **Connection** representa una conexión física entre el cliente y el servidor
- ✓ Para crear una conexión se usa la clase **DriverManager**
- ✓ Se especifica la URL, el nombre y la contraseña

3. Ejecutar sentencia SQL

- ✓ Una vez que tienes una conexión puedes ejecutar sentencias SQL
- ✓ Primero se crea el objeto **Statement** desde la conexión
- ✓ Posteriormente se ejecuta la consulta y su resultado se devuelve como un **ResultSet**

4. Acceso al conjunto de resultados

- ✓ El **ResultSet** es el objeto que representa el resultado
- ✓ No carga toda la información en memoria
- ✓ Internamente tiene un cursor que apunta a una fila concreta del resultado en la base de datos
- ✓ Hay que posicionar el cursor en cada fila y obtener la información de la misma

Posicionamiento del cursor

- El cursor puede estar en una fila concreta
- También puede estar en dos filas especiales
 - Antes de la primera fila (*Before the First Row*, BFR)
 - Después de la última fila (*After the Last Row*, ALR)
- Inicialmente el **ResultSet** está en BFR
- **next()** mueve el cursor hacia delante
 - Devuelve **true** si se encuentra en una fila concreta y **false** si alcanza el ALR

5. Obtención de los datos de la fila

- ✓ Cuando el ResultSet se encuentra en una fila concreta se pueden usar los métodos de acceso a las columnas:
 - String getString(String columnLabel)
 - String getString(int columnIndex)

Importante: los índices de las columnas empiezan en 1 y no en cero como habría de esperarse.

6. Liberar recursos

- Cuando se termina de usar una Connection, un Statement o un ResultSet es necesario liberar los recursos que necesitan
- Puesto que la información de un ResultSet no se carga en memoria, existen conexiones de red abiertas
- Métodos close():

- ✓ `ResultSet.close()` – Libera los recursos del `ResultSet`. Se cierran automáticamente al cerrar el `Statement` que lo creó o al reejecutar el `Statement`.
- ✓ `Statement.close()` – Libera los recursos del `Statement`.
- ✓ `Connection.close()` – Finaliza la conexión con la base de datos

3. Apéndice Comando SELECT

SELECT [ALL / DISTINCT] [*] / [ListaColumnas_Expresiones] AS
[Expresión]

- ✓ **SELECT** Permite seleccionar las columnas que se van a mostrar y en el orden en que lo van a hacer. Simplemente es la instrucción que la base de datos interpreta como que vamos a solicitar información. •
- ✓ **ALL:** es el valor predeterminado, especifica que el conjunto de resultados puede incluir filas duplicadas. Por regla general nunca se utiliza.
- ✓ **DISTINCT:** especifica que el conjunto de resultados sólo puede incluir filas únicas. Es decir, si al realizar una consulta hay registros exactamente iguales que aparecen más de una vez, éstos se eliminan. Muy útil en muchas ocasiones.
- ✓ **ListaColumnas_Expresiones** : nombres de campos de la tabla que nos interesan

WHERE codiciones

- ✓ **Where** Especifica la condición de filtro de las filas devueltas. Se utiliza cuando no se desea que se devuelvan todas las filas de una tabla, sino sólo las que cumplen ciertas condiciones
- ✓ **Condiciones:** Son expresiones lógicas a comprobar para la condición de filtro, que tras su resolución devuelven para cada fila TRUE o FALSE, en función de que se cumplan o no > (Mayor), >= (Mayor o igual), < (Menor), <= (Menor o igual), = (Igual), <> o !=

(Distinto), IS [NOT] NULL (para comprobar si el valor de una columna es o no es nula, es decir, si contiene o no contiene algún valor)

- ✓ **LIKE** para la comparación de un modelo. Para ello utiliza los caracteres comodín especiales: “%” y “_”. Con el primero indicamos que en su lugar puede ir cualquier cadena de caracteres, y con el segundo que puede ir cualquier carácter individual (un solo carácter). Por ejemplo:
 - El nombre empieza por A: Nombre LIKE „A%”
 - El nombre acaba por A: Nombre LIKE „%A”
 - El nombre contiene la letra A: Nombre LIKE „%A%”
 - El nombre empieza por A y después contiene un solo carácter cualquiera: Nombre LIKE „A_”
 - El nombre empieza una A, después cualquier carácter, luego una E y al final cualquier cadena de caracteres: Nombre LIKE „A_E%”
- ✓ **BETWEEN**: para un intervalo de valores. Por ejemplo: Clientes entre el 30 y el 100: CodCliente **BETWEEN 30 AND 100**
- ✓ **IN()**: para especificar una relación de valores concretos. Por ejemplo: Ventas de los Clientes 10, 15, 30 y 75: CodCliente **IN(10, 15, 30, 75)**

ORDER BY columnas [ASC/DESC]

Order By Define el orden de las filas del conjunto de resultados. Se especifica el campo o campos (separados por comas) por los cuales queremos ordenar los resultados.

- **ASC / DESC**

ASC es el valor predeterminado, especifica que la columna indicada en la cláusula ORDER BY se ordenará de forma ascendente, o sea, de menor a mayor. Si por el contrario se especifica **DESC** se ordenará de forma descendente (de mayor a menor).