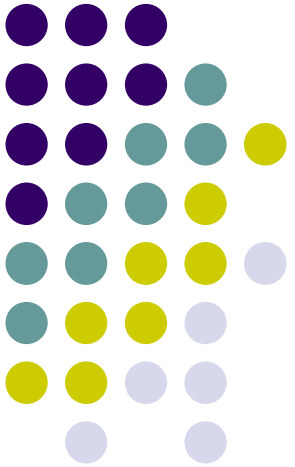
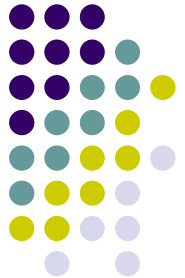
# Java

## IO y Excepciones



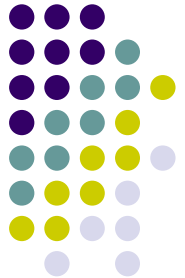
Lic. Claudio Zamoszczyk
claudio@honou.com.ar

# Contenidos



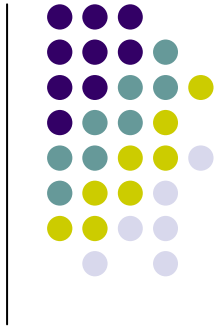
- Excepciones
- Entrada y Salida de datos en Java
- Manejo de Archivos

# Archivos .JAR



- **El formato de archivos Java permite empaquetar varios archivos en un sólo archivo. Típicamente un fichero JAR contendrá los archivos de clases y los recursos auxiliares asociados.**

# Archivos .JAR



- **El formato JAR proporciona muchos beneficios.**

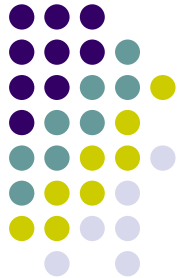
  **Seguridad: se puede firmar digitalmente el contenido de un JAR. Los usuarios que reconozcan la firma pueden permitir privilegios de seguridad que de otro modo no tendría.**

  **Compresión: El formato JAR permite comprimir el contenido para ahorrar espacio.**

  **Empaquetado sellado: Los paquetes almacenados en un archivo JAR pueden ser sellados opcionalmente para que el paquete puede reforzar su consistencia. El sellado de un paquete dentro de un JAR significa que todas las clases definidas en ese paquete deben encontrarse dentro del mismo fichero JAR para que funcionen.**

  **Portabilidad: El mecanismos para manejar los JAR son una parte estándar del corazón del API de la plataforma Java. Permitiendo crear componentes de software reutilizables en el desarrollo de otras aplicaciones. Los archivos JAR son lo mas cercano a una DLL.**

# Excepciones

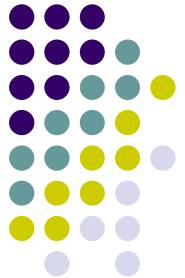


- **En el lenguaje Java, una Exception es un cierto tipo de error a una condición anormal que se produce durante la ejecución de un programa.**

- **Algunas exceptiones son fatales y provocan la finalización de la aplicación. En ese caso es conveniente salir ordenadamente e informar del inconveniente.**

- **Otras, como por ejemplo no encontrar un archivo que hay que leer son recuperables. En este caso el programa debe dar al usuario la oportunidad de corregir el error indicando un nuevo archivo para leer.**

# Excepciones



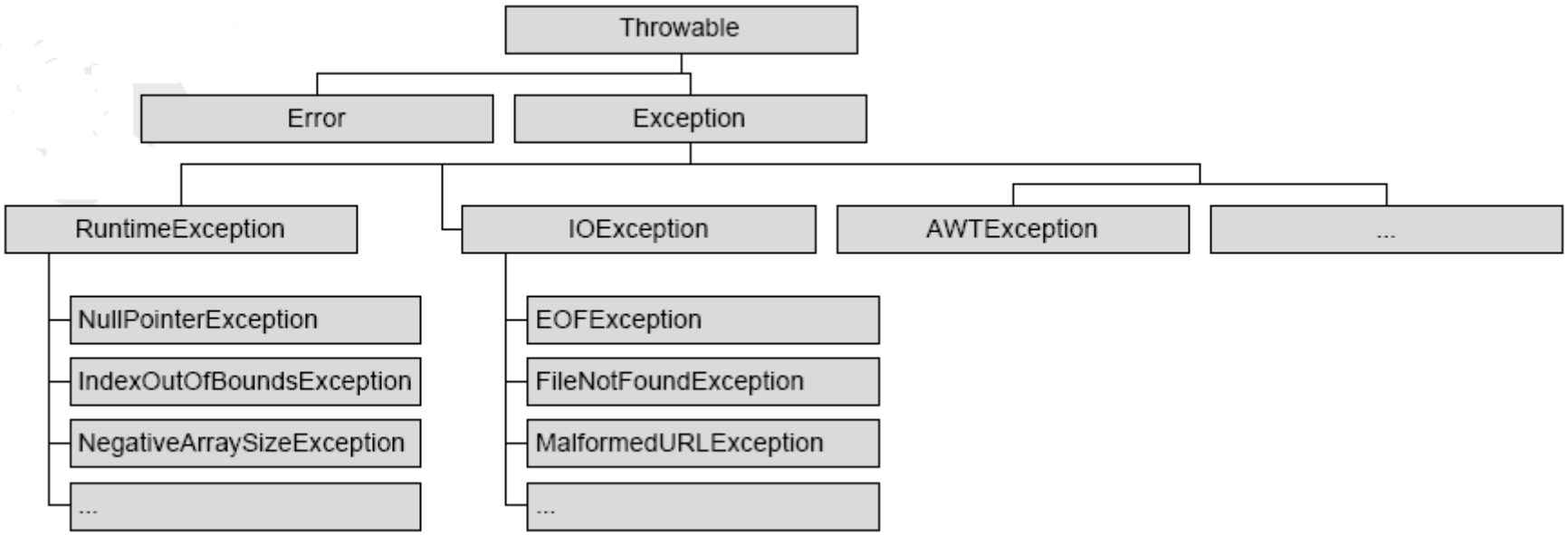
- **Un buen programa debe gestionar correctamente todas o la mayor parte de los errores que se pueden producir. Hay dos “estilos” de hacer esto:**

  **- A la “antigua usanza”: los métodos devuelve un código de error. Y con una serie de if else… se gestionan los diferentes tipos de errores. Este sistema resulta muy complicado cuando hay varios noveles de llamadas a los métodos.**

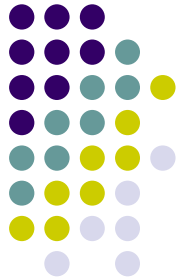
  **- Con soporte en el propio lenguaje: En este caso el propio lenguaje proporciona construcciones especiales para gestionar los errores.**

# Excepciones en Java

Los errores se representan mediante dos tipos de clases derivadas de la clase Throwable: **Error** y **Exception**



# Excepciones



- La clase Error esta relacionada con errores del sistema o de la JVM, y son irrecuperables.

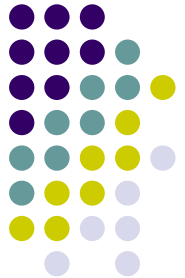
- La clase Exception tiene mas interés. Dentro de ella se puede distinguir

  -RuntimeException: Son excepciones muy frecuentes, ligadas a errores de programación.

  Ej: referencias a objetos no instanciados, infice fuera del alcance de un Array, etc etc. En estos casos el programador no tiene que utilizar un bloque try/catch para tratar los errores. (Implícitas)

  - Demás clases derivadas de Exception son excepciones explicitas. Java obliga a tenerlas en cuenta y chequear si se producen.

# Excepciones



- **En todos los casos, las excepciones poseen métodos que nos permiten recuperar información relacionada con las mismas.**

**- getMessage() : extrae el mensaje asociado con la excepción**

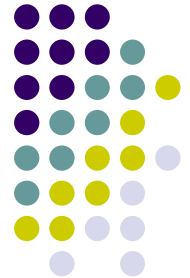
**- printStackTrace() : indica el método donde se lanzo la excepción.**

# Excepciones: try & catch

- En el caso de las excepciones que no pertenecen a RuntimeException y que por lo tanto Java obliga a tenerlas en cuenta habra que utilizar los bloques try, catch y finally.

```
try {
      …….
} catch (SQLException e) {
    System.out.println(“Error con la BD:” + e.getMessage());
    e.printStackTrace();
}
```

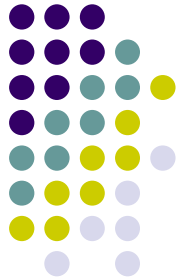
# Excepciones: try & catch



- **El código dentro del bloque try es el que se encuentra vigilado, a la espera de recibir un error. Si se produce una situación anormal y se lanza una excepcion el control sale del bloque try y pasa al bloque catch, que se hace cargo de la situación y decide que hay que hacer. Pueden existir tantos bloques catch como tipos de excepciones se quieran capturar.**

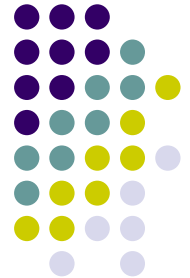
- **Por ultimo queda el bloque finally, que es opcional y nos permite realizar tareas de limpieza de variables o cualquier otro tipo de operación.**

# Relanzar Excepciones



- **Existen algunos casos en los cuales el código de un método puede generar una excepción y no se desea incluir en dicho método la gestión de errores.**
- **Java permite que este método pase o relance (throws) la excepción al método desde el que ha sido llamado, sin incluir el bloque try/catch correspondiente.**
- **Eso se consigue mediante la adición de throws mas el nombre de la Exception concreta.**

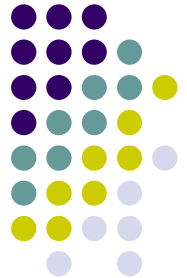
# Relanzar Excepciones



```
public void leerArchivo() throws IOException, MIException {
…..
}

public void recuperarConfig() {
    try {
            ….
            leerArchivo();
    } catch(IOException e) {
            …
    } catch(MIException e) {
            …
    }
}
```

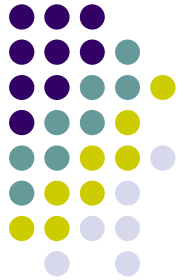
# Creando Excepciones



- **Nosotros mismos podemos crear nuestra excepciones con solo heredar de la clase Exception.**

Ver ejemplo PrincipalMiException.java

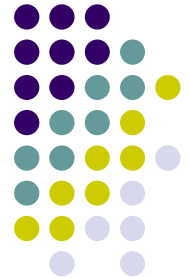
# ¿Preguntas?



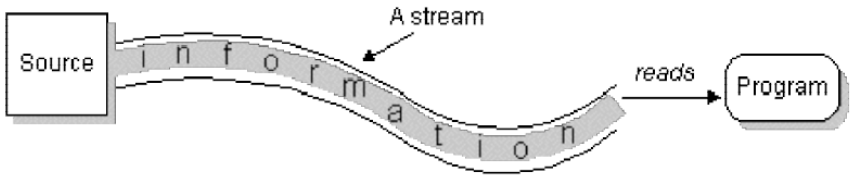
# IO – Entrada y Salida de datos

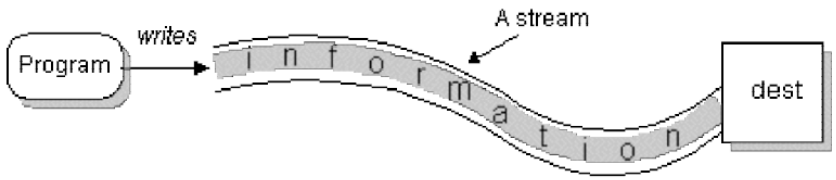
- **Los programas necesitan comunicarse con su entorno, tanto para recoger datos como para devolver un resultado.**
- **La manera de representar estas entradas y salidas en Java es a base de streams (flujos de datos). Un stream es una conexión entre nuestro programa y la fuente o destino de los datos. La información en trasladada en serie de caracteres a través de la conexión.**

# IO – Entrada y Salida de datos

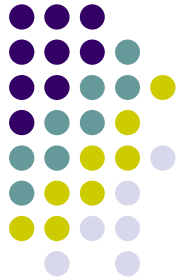


- **Esto permite un modelo más abstracto a la hora establecer operaciones de lectura y escritura, sin importar si es sobre archivos, puertos tcp, etc.**



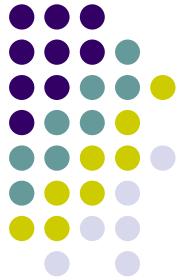



# IO – Java



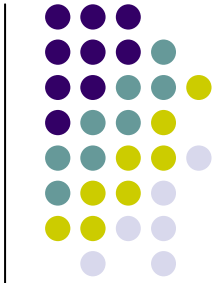
- **El package java.io contiene las clases necesarias para realizar las operaciones de lectura/escritura de datos. Dentro del package podemos encontrar clases para la lectura de bytes y caracteres sobre archivos, buffers de datos, transformadores de flujos, etc.**

# IO – Java



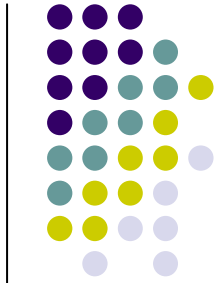
- **El package java.io contiene las clases necesarias para realizar las operaciones de lectura/escritura de datos. Dentro del package podemos encontrar clases para la lectura de bytes y caracteres sobre archivos, buffers de datos, transformadores de flujos, etc.**

# IO – Java



| Clases | Función que realizan |
|---|---|
| FileReader, FileWriter, FileInputStream y FileOutputStream | Son las clases que leen y escriben en **archivos** de disco. |
| StringReader, StringWriter, CharArrayReader, CharArrayWriter, ByteArrayInputStream, ByteArrayOutputStream, StringBufferInputStream | Estas clases tienen en común que se comunican con la **memoria** del ordenador. En vez de acceder del modo habitual al contenido de un String, por ejemplo, lo leen como si llegara carácter a carácter. Son útiles cuando se busca un modo general e idéntico de tratar con todos los dispositivos que maneja un programa. |
| PipedReader, PipedWriter, PipedInputStream, PipedOutputStream | Se utilizan como un "tubo" o conexión bilateral para transmisión de datos. Por ejemplo, en un programa con dos threads pueden permitir la comunicación entre ellos. Un thread tiene el objeto PipedReader y el otro el PipedWriter. Si los streams están conectados, lo que se escriba en el PipedWriter queda disponible para que se lea del PipedReader. Tambíen puede comunicar a dos programas distintos. |

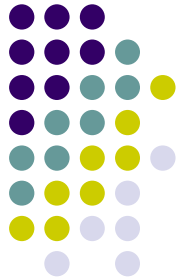
# IO – Java



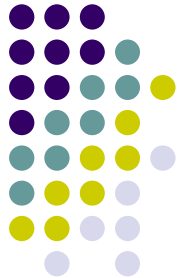
| Clases | Función que realizan |
|---|---|
| BufferedReader, BufferedWriter, BufferedInputStream, BufferedOutputStream | Como ya se ha dicho, añaden un **buffer** al manejo de los datos. Es decir, se reducen las operaciones directas sobre el dispositivo (lecturas de disco, comunicaciones por red), para hacer más eficiente su uso. ***BufferedReader*** por ejemplo tiene el método ***readLine()*** que lee una línea y la devuelve como un String. |
| InputStreamReader, OutputStreamWriter | Son clases puente que permiten **convertir** streams que utilizan bytes en otros que manejan caracteres. Son la única relación entre ambas jerarquías y no existen clases que realicen la transformación inversa. |
| ObjectInputStream, ObjectOutputStream | Pertenecen al mecanismo de la **serialización** y se explicarán más adelante. |
| FilterReader, FilterWriter, FilterInputStream, FilterOutputStream | Son clases base para aplicar diversos **filtros** o procesos al stream de datos. También se podrían extender para conseguir comportamientos a medida. |
| DataInputStream, DataOutputStream | Se utilizan para escribir y leer datos directamente en los formatos propios de Java. Los convierten en algo ilegible , pero independiente de plataforma y se usan por tanto para **almacenaje** o para **transmisiones** entre ordenadores de distinto funcionamiento. |
| PrintWriter, PrintStream | Tienen métodos adaptados para **imprimir** las variables de Java con la apariencia normal. A partir de un boolean escriben "true" o "false", colocan la coma de un número decimal, etc. |

# IO – Java

Ver ejemplos.



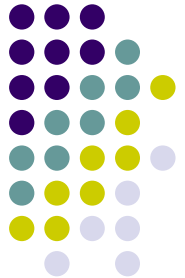
# IO – Java Serialización



- **La serialización es un proceso por el cual un objeto cualquiera se puede convertir en una secuencia de bytes con la que mas tarde se podrá reconstruir dicho objeto manteniendo el valor de sus variables. Esto permite guardar un objeto en un archivo o mandarlo por la red.**

- **Para que una clase pueda utilizar la serialización, debe implementar la interfase Serializable, que no define ningún método.**

# IO – Java Serialización

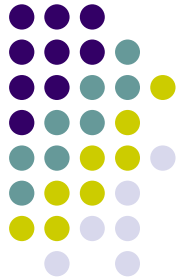


```
public MiClase implements Serializable {
        private String texto;
        ……

}
```

Ver ejemplo serialización.

# ¿Preguntas?



# ¿Preguntas?