Algoritmos y Programación 3 - Excepciones y Persistencia

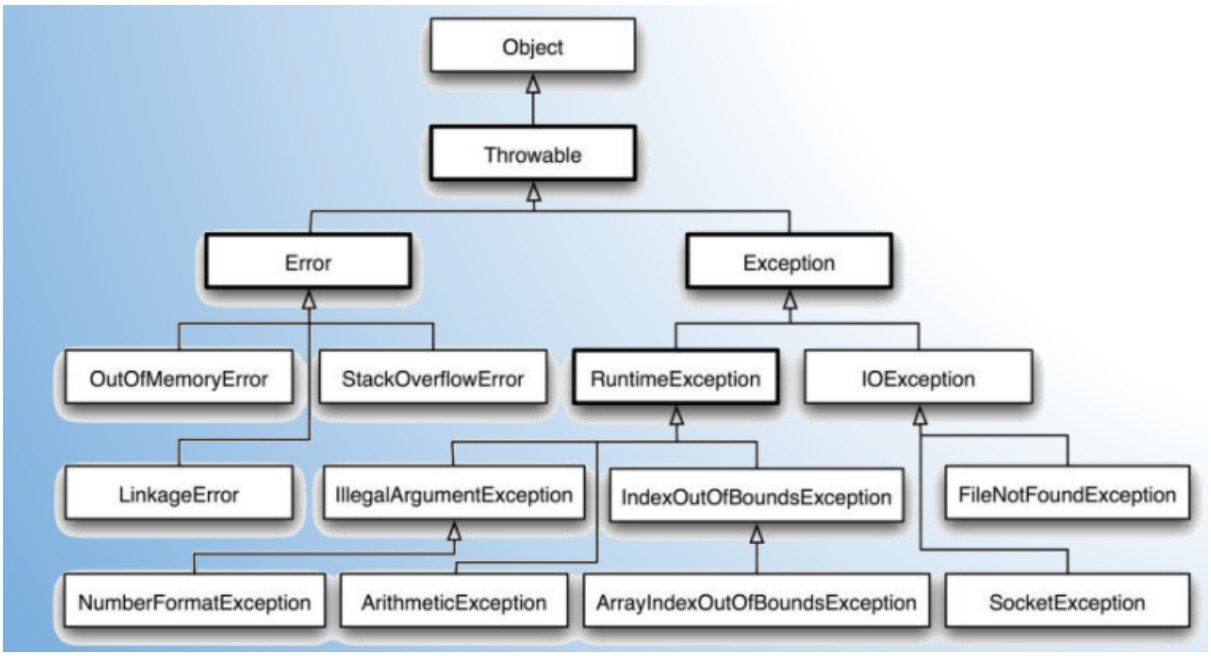
Excepción: objeto que se usa para comunicar una situación anómala desde un entorno que la detecta al ámbito desde el cual fue invocado un método.

Se “lanza” cuando en el contexto del método no hay suficiente información para resolver una anomalía, por lo que se debe salir del mismo hacia el módulo que lo invocó informando esta situación, pero *sin provocar la finalización del programa ni enviar mensajes a un presunto usuario del que no se sabe nada.*

Definición por partes:

1. **Una excepción es un objeto**: en efecto, las excepciones son objetos con identidad, comportamiento y estado.
2. **Se envía desde un entorno que la detecta**: habitualmente hay un método que crea y lanza un objeto de excepción porque no puede resolver el problema en su ámbito.
3. **Se envía al ámbito desde el cual fue invocado un método**: el objeto viaja desde el método que detectó el problema hasta el ámbito desde el cual ese método fue invocado. Este ámbito podrá recibir ese objeto y ver de qué manera tratar el problema.
4. **Se usa para comunicar una situación anómala:** su principal uso es la comunicación entre dos ámbitos, el que detecta el problema y el que debe lidiar con el mismo.

Jerarquía de clases en Java



Hay 2 tipos de excepciones en Java:

Verificadas (checked)
No verificadas (unchecked)

Las excepciones verificadas extienden Exception (pero no RuntimeException) y, **para poder compilar el programa, deben ser declaradas (con throws) o atrapadas (con try..catch..finally).**

Las excepciones no verificadas extienden RuntimeException o Error, y no es obligatorio declararlas ni atraparlas.

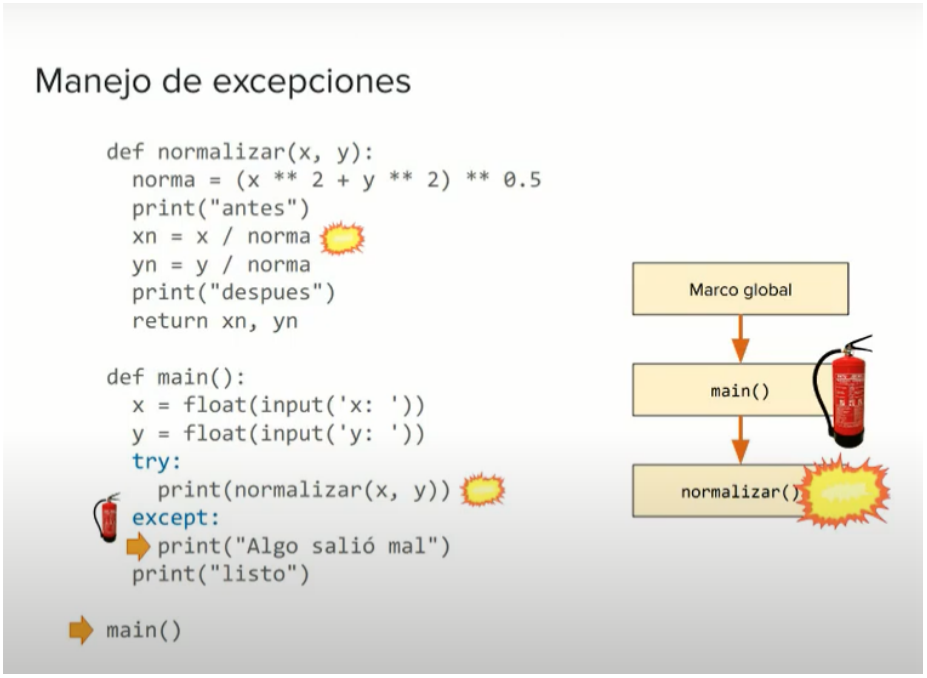
En el bloque *try* se encierra el código que podría provocar errores durante la ejecución y lanzar excepciones.

En el o los bloque(s) *catch* se encierra el código que se ejecuta cuando se atrapa una excepción del tipo indicado en el parámetro.

En el bloque *finally* (que es optativo cuando ya se definió, por lo menos, un bloque catch) se encierra el código que debe ejecutarse siempre, se haya o no lanzado y atrapado una excepción.

El bloque try debe preceder a, por lo menos, un bloque catch o al bloque finally. Si se definen dos o más bloques catch, los bloques que atrapan las excepciones más específicas deben aparecer primero y los que atrapan las más generales deben aparecer por último.

Las excepciones se propagan por todo el stack trace (pila de ejecución), la excepción ocurre en la función Normalizar() y va escalando hasta llegar al marco global que hace que el programa se detenga.



Atrapando la excepción en el main me permite que el programa no se detenga.

Try: detiene la propagación de la excepción (donde va la línea que podría o no tirar una excepción)
Catch: se ejecuta cuando atrapa la excepción

Finally: se ejecuta sí o sí, no importa que haya atrapado una excepción o no. Me sirve por ejemplo para asegurarme que un archivo se cerró.

Dato: si quiero crear una excepción propia no chequeada que sea del tipo RunTimeException.

(✿◠‿◠) (✿◠‿◠) (✿◠‿◠) (✿◠‿◠) (✿◠‿◠) (✿◠‿◠) (✿◠‿◠) (✿◠‿◠) (✿◠‿◠)

Persistencia: capacidad de un objeto de trascender el tiempo o el espacio. Permite que un objeto sea usado en diferentes momentos, por el mismo programa o por otros, así como en diferentes instalaciones de hardware.

Un objeto persistente es aquel que conserva su estado en un medio de almacenamiento permanente, pudiendo ser reconstruido por el mismo u otro proceso, de modo tal que al reconstruirlo se encuentre en el mismo estado en que se lo guardó. Al objeto no persistente se lo denomina efímero.

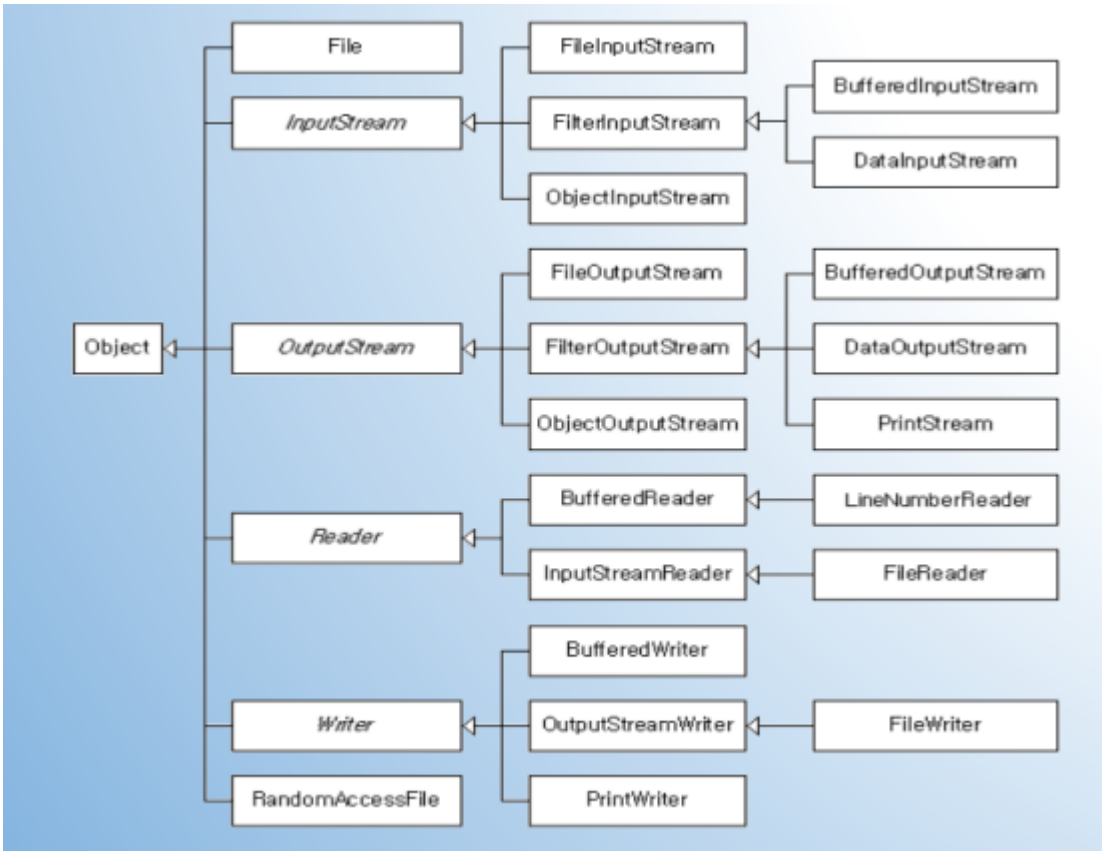
Manejo de archivos en Java

Archivo: unidades básicas de almacenamiento de datos.
Directorio: conjunto de archivos.

2 tipos de archivos: los binarios (InputStream y OutputStream) y los de texto (Reader y Writer).

La clase File representa una ruta en el sistema operativo. En los binarios InputStream representa un flujo de entrada de bytes, equivalente en el OutputStream para escribir.

Jerarquía de clases



Decorator en Java (voy agregándole ingredientes a mi pizza)

Los diseñadores del lenguaje pensaron que utilizar la herencia para componer las combinaciones de funcionalidades más comunes requeridas para el manejo de archivos habría resultado en una verdadera explosión de clases.

Por eso, diseñaron las clases para el manejo de archivos siguiendo el patrón de diseño Decorator. En lugar de usar la herencia para agregar funcionalidad a las clases, este patrón permite agregar funcionalidad a los objetos individuales (en tiempo de ejecución).

Clase File

```
public class Main {
  public static void main(String[] args) {
    String nomElegido = "";
    JFileChooser fc = new JFileChooser("src/clasefile");
    fc.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
    if (fc.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
      nomElegido = fc.getSelectedFile().getPath();
    }
    File f = new File(nomElegido);
    System.out.println("Nombre: " + f.getName());
    System.out.println("Ubicado en: " + f.getParent());
    System.out.println("Última modificación: " +
                       DateFormat.getInstance().format(new Date(f.lastModified())));
    if (f.isFile()) {
      System.out.println("ES UN ARCHIVO");
      System.out.println("Tamaño: " + f.length() + " bytes");
    } else if (f.isDirectory()) {
      System.out.println("ES UN DIRECTORIO");
      System.out.println("Contenido:");
      int cantArchivos = 0;
      long tamArchivos = 0;
      int cantDirectorios = 0;
      for (File elemento : f.listFiles()) {
        System.out.print(DateFormat.getInstance().format(new Date(elemento.lastModified())));
        if (elemento.isFile()) {
          cantArchivos++;
          tamArchivos += elemento.length();
          System.out.printf("     %,14d  ", elemento.length());
        } else if (elemento.isDirectory()) {
                cantDirectorios++;
                System.out.printf("     <DIR>           ");
        }
        System.out.println(elemento.getName());
      }
      System.out.printf(" %,7d archivos  %,14d  bytes\n", cantArchivos, tamArchivos);
      System.out.printf(" %,7d dirs      %,14d  bytes libres\n",
                        cantDirectorios, f.getFreeSpace());
    }
  }
}
```

Archivos de Texto Plano (Unicode)

Secuencia de decoración:

FileReader (fr) es un lector de archivos lento y no tiene cómo leer renglones.
Se lo decora con BufferReader (br) para que sea más rápido, donde además puedo leer renglón por renglón.
Vuelvo a decorar con LineNumberReader (lr) donde voy a poder contar los renglones.

```
public class Main {

    public static void main(String[] args) throws FileNotFoundException, IOException {
        String nomArch = "";
        JFileChooser fc = new JFileChooser("src/archivosDeTexto");
        if (fc.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
            nomArch = fc.getSelectedFile().getPath();
        }

        FileReader fr = new FileReader(nomArch);
        BufferedReader br = new BufferedReader(fr);
        LineNumberReader lr = new LineNumberReader(br);

        FileWriter fw = new FileWriter(nomArch + ".bak");
        BufferedWriter bw = new BufferedWriter(fw);
        PrintWriter pw = new PrintWriter(bw);

        String renglon;
        while ((renglon = lr.readLine()) != null) {
            pw.println(lr.getLineNumber() + ") " + renglon);
        }

        pw.close();
    }
}
```

Archivos Binarios Secuenciales (misma lógica que el anterior)

```
public class Main {
    public static void main(String[] args) throws FileNotFoundException, IOException {
        String nomArch = "";
        JFileChooser fc = new JFileChooser("src/archivosBinariosSecuenciales");
        if (fc.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
            nomArch = fc.getSelectedFile().getPath();
        }
        FileInputStream fis = new FileInputStream(nomArch);
        BufferedInputStream bis = new BufferedInputStream(fis);
        DataInputStream dis = new DataInputStream(bis);
        FileOutputStream fos = new FileOutputStream(nomArch + ".bak");
        BufferedOutputStream bos = new BufferedOutputStream(fos);
        DataOutputStream dos = new DataOutputStream(bos);
        int byteLeido;    // byte no permite contener 128..255. Por eso: int
        boolean finDeArchivo = false;
        while (!finDeArchivo) {
            try {
                byteLeido = dis.readByte();
                dos.writeByte(byteLeido);
            } catch (EOFException ex) {
                dos.close();
                finDeArchivo = true;
            }
        }
    }
}
```

Archivos Binarios de Acceso Aleatorio (No se pueden decorar)

```
public class Main {

    public static void main(String[] args) throws FileNotFoundException, IOException {
        String nomArch = "";
        JFileChooser fc = new JFileChooser("src/archivosBinariosDeAccesoAleatorio");
        if (fc.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
            nomArch = fc.getSelectedFile().getPath();
        }

        RandomAccessFile raf = new RandomAccessFile(nomArch, "rw");

        long tam = raf.length();
        for (long i = 0; i < tam / 2; i++) {
            raf.seek(i);
            int b1 = raf.read();
            raf.seek(tam - i - 1);
            int b2 = raf.read();
            raf.seek(i);
            raf.write(b2);
            raf.seek(tam - i - 1);
            raf.write(b1);
        }

        raf.close();
    }
}
```

Lo importante de acá es **ENTENDER CONCEPTUALMENTE EL DECORATOR QUE SE REALIZA.**

Serialización, con ella logramos la Persistencia

Con serializar() grabamos un objeto, y al de salida le paso lo que quiera serializar.

IMPORTANTE: ¿Qué es el objeto al que le mando lo que quiero guardar?

Tenemos un FileInputStream primitivo, que guarda solo bytes.
Lo decoro con ObjectInputStream para que me guarde objetos y luego lo decoro con BufferedInputStream para que tenga velocidad.

Con eso obtengo: un objeto que es capaz de leer objetos rápidos en un archivo.

Los objetos que quiero que persistan en el tiempo deben implementar Serializable.

```
package serializacion;
import java.io.*;
import java.util.ArrayList;
public class Lista implements Serializable {
  private ArrayList<String> valores;
  public Lista() {
    valores = new ArrayList<>();
  }
  public void agregar(String s) {
    valores.add(s);
  }
  public void mostrar() {
    String cadena = "La lista contiene:\n";
    for (String s : valores) {
        cadena += s + "\n";
    }
    EntradaSalida.mostrarString(cadena);
  }
  public Lista deSerializar(String nomArch) throws IOException,ClassNotFoundException {
    ObjectInputStream o =
        new ObjectInputStream(new BufferedInputStream(new FileInputStream(nomArch)));
    Lista j = (Lista) o.readObject();
    o.close();
    return j;
  }
  public void serializar(String nomArch) throws IOException {
    ObjectOutputStream o =
       new ObjectOutputStream(new BufferedOutputStream(new FileOutputStream(nomArch)));
    o.writeObject(this);
    o.close();
  }
}
```

```
package serializacion;

import javax.swing.JOptionPane;

public class EntradaSalida {

  public static String leerString(String texto) {
    String st = JOptionPane.showInputDialog(texto);
    return (st == null ? "" : st);
  }

  public static boolean leerBoolean(String texto) {
    int i=JOptionPane.showConfirmDialog(null,texto,"Consulta",JOptionPane.YES_NO_OPTION);
    return i == JOptionPane.YES_OPTION;
  }

  public static void mostrarString(String s) {
    JOptionPane.showMessageDialog(null, s);
  }
}
```

```java
package serializacion;
import java.io.IOException;
public class Main {
  public static void main(String[] args) {
      Lista li = new Lista();
      try {
          li = li.deSerializar("lista.txt");
          if (!EntradaSalida.leerBoolean("Ya hay una lista. ¿Desea reutilizarla?")) {
              li = new Lista();
          }
      } catch (IOException | ClassNotFoundException e) {
          EntradaSalida.mostrarString("Lista nueva!");
      }
      do {
          String dato = EntradaSalida.leerString("Ingrese una cadena");
          li.agregar(dato);
      } while (EntradaSalida.leerBoolean("Desea seguir agregando valores?"));
      li.mostrar();
      try {
          li.serializar("lista.txt");
      } catch (Exception e) {
          EntradaSalida.mostrarString(e.getMessage() + "\nERROR AL GRABAR!");
      }
  }
}
```