**EXCEPCIONES**

**Qué es una excepción**

**Definicion**

Una excepción que no fue capturada correctamente hace que el sistema se caiga. Una excepción es un evento que ocurre durante la ejecución de un programa, que interrumpe el flujo normal de ejecución.

Java utiliza excepciones para el manejo de errores, error handling. Es un sinónimo de eventos excepcionales. Un buen manejo de Excepciones hace a un sistema robusto y estable, confiable.

**Bloques try, catch y finally**

Dentro del bloque try se pondrá todo el código que puede arrojar alguna excepción. Los bloques catch son los encargados de atrapar las excepciones arrojadas por alguna sentencia del bloque try. El bloque finally se ejecuta siempre después del try y del catch.

Ejemplo:

try

{

...

}

catch (ClassNotFoundException excep1)

{

...

}

catch (Exception excep2)

{

...

}

finally

{

... // limpieza de código

}

T**ipos de Excepciones**

**Unchecked Exceptions**

Son las excepciones que tienen como superclase a la clase RuntimeException. No hay necesidad de capturarlas, es decir que no se necesita utilizar el bloque try/catch/finally, pero al saltar una excepción de este tipo, como todas las excepciones corta el flujo de ejecución.

Pueden ser difíciles de detectar, pero finalmente hacen que el sistema se caiga si no fueron tomadas en cuenta.

Ejemplo: Cuando se realiza una división por cero se lanza automáticamente una ArithmeticExcepcion. Cuando se quiere acceder a un objeto que apunta a null se lanza automáticamente una NullPointerException. Cuando se quiere acceder a un array con un índice que es mayor al tamaño del array, se lanza automáticamente una ArrayIndexOutOfBoundExcepcion.

  int unValor = 0; // Este valor se suponia que no podia ser Cero,

                                  //pero en algun momento se convirtio en Cero

  try{

    int resultado = 10/unValor

  }catch(ArithmeticExcepcion ae){

    // Aqui tengo el control del flujo por el error de dividir por Cero

    // Asi que informo al usuario que hubo un error.

  }finally{

    // Y finalmente continuo con el sistema, reestableciendo el control.

  }

**Checked Exceptions**

Son las excepciones que tienen como superclase a la clase Exception. Necesitan ser capturadas, caso contrario no se podrá compilar el código.

En el caso del FileReader se está accediendo a lo que se llama un recurso externo al sistema en sí. El sistema consta de las variables, las clases, los packages y los métodos que definimos, pero un archivo es un recurso externo, es decir que puede no existir al momento de ejecutar nuestro programa, o ser borrado mientras nuestro programa está funcionando.

También llamamos recurso externo a una conexión con la base de datos, porque depende de una conexión de red. Todos estos artefactos que no son parte del programa, sino que se interactúa con ellos, pueden traer algunas complicaciones como, no existir, que la red no esté conectada, que no haya más espacio en disco, que la red se caiga en medio de una corrida del programa.

Todos estos motivos llevan a la necesidad de evaluar si se pueden realizar correcta y completamente las instrucciones del programa. Por lo tanto, si algo sale mal, lo que no puede ocurrir es que dejemos una conexión tomada, o un archivo sin que otro sistema pueda acceder a él.

Aquí es donde tenemos que prestar mucha atención para que nuestro sistema no traiga problemas, simplemente verificar en el finally que se han liberado los recursos.

Ejemplo: La clase FileReader se utiliza para el acceso a disco. Cuando utilizo un acceso a disco con métodos de la clase FileReader, debo capturar la excepcion IOException.

FileReader archivo = null;

String nombreDeArchivoSinAcceso = "archivito.txt";

try{

archivo = new FileReader( new File( nombreDeArchivoSinAcceso ) );

  // Aqui intento de hacer algo con el archivo

  // puede ser que tenga tomado el archivo o

    // puede ser que no se llegue porque salta la excecion

}catch(IOException ioe){

    // Aqui informo al usuario que Ocurrio un error de Acceso al Recurso

Externo.

}finally{

    // Finalmente libero el recurso externo, si fue tomado.

    if( archivo!=null ) {

           try{

             archivo.close();

           }catch(Exception e){

      }

    }

  }

**La sentencia “throw”**

**Qué es**

La sentencia throw se utiliza para arrojar excepciones. Requiere un único argumento, una instancia de la clase java.lang.Throwable, que es implementada por la clase Exception y errores.

**Utilización**

Ejemplo:

boolean hayError = true;

if(hayError){

throw new Exception();

}

Esto significa que luego de evaluar la cláusula hayError y el hecho que, de verdadero, significa que debe cortarse el flujo de corrida del programa, y desde el método que llama a este método debe capturarse este error, informar al usuario que ocurrió el error y luego restablecer el sistema.

**Creación de excepciones propias**

L**a clase Exception como superclase**

Por convención, su nombre debería terminar con la palabra Exception. Consiste en crear una clase que hereda de java.lang.Exception. Lo conveniente es ir identificando qué tipo de excepciones, en el flujo de información de la vida real, pueden ser representadas por una excepción creada a medida, de forma tal de poder entender de forma inmediata un bloque de código.

El manejo de excepciones hace que el código sea legible y fácilmente comprensible.

Ejemplo:

public class MiExcepcionException extends Exception{

// Esto es un comentario. Aqui va la definicion de mi clase propia de excepcion.

}

**La keyword “throws”**

La palabra clave throws se utiliza en la firma de los métodos que pueden lanzar excepciones. Una buena combinación de Creación de Excepciones propias del negocio que estamos modelando con el sistema, y la declaración en los métodos indicados que pueden lanzar este tipo de excepciones, hace que sea fácil programar y compartir clases entre distintos equipos de desarrollo.

Estas prácticas facilitan mucho el compartir código y queda uno siempre obligado a capturar excepciones que permiten un buen control del flujo de datos dentro del sistema, y lo hacen robusto.

Ejemplo:

 public class Hombre{

        public void comer() throws MiExcepcion{

          // Esto es un comentario. Aqui va algo de codigo

      //Esto es otro comentario, Aqui evaluamos una condicion que hace que arrojemos una excepcion

      throw new MiException();

    }

  }

**ENUMS**

Enum es una palabra reservada en Java, una funcionalidad que es utilizada para representar un número fijo de valores conocidos en Java, por ejemplo, el número de días en una semana, número de planetas en el sistema solar, etc. Este concepto fue introducido en la versión 1.5 de la JDK dentro de la versión 5.0 de J2SE.

A continuación, observaremos un ejemplo de cómo se resolvía la representación de valores enumerables antes de la versión 1.5 de la JDK. En este ejemplo utilizaremos los días de la semana, DOMINGO(0), LUNES(1), MARTES(2), MIERCOLES(3), JUEVES(4), VIERNES(5) Y SABADO(6)

public class DiasSemana {

   public static final int DOMINGO = 0;

   public static final int LUNES = 1;

   public static final int MARTES = 2;

   public static final int MIERCOLES = 3;

   public static final int JUEVES = 4;

public static final int VIERNES = 5;

public static final int SABADO = 6;

}

public class Dia {

   private int dia; // Ej.: DiasSemana.LUNES,

}

Aunque este código puede resolver su propósito tiene serias limitaciones:

1) No tiene Type-Safety: no es type-safe, se puede asignar cualquier valor válido de int a la variable día por ejemplo 99 aunque no existe un día para representar ese valor.

 2) Representación: al imprimir el valor de alguna de estas constantes imprimirá el valor numérico en lugar de un valor representativo del día, ej.: al imprimir LUNES se imprimirá un 1 en lugar de "LUNES".

3) Sin namespace: para acceder a la constante DiaSemana es necesario prefijar el nombre de la clase “DiaSemana.LUNES” en lugar de usar solo LUNES.

Java Enum es la respuesta a estas limitaciones. Las Enums en Java son type-safe, proveen nombres con significado del tipo String y tienen su propio namespace. A continuación, veremos un ejemplo utilizando Enum en Java:

public enum Dias {DOMINGO, LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO};

Dia dia = Dias.LUNES;

dia = 1; //compilation error

En el ejemplo, Dias es un enum y DOMINGO, LUNES, MARTES, MIERCOLES, JUEVES, VIERNES y SABADO son constantes enum. Existen llaves rodeando las constantes enum, esto es debido a que las Enum son tipos como las clases y las interfaces en Java.  De la misma forma existen convenciones para nombrar las enums al igual que en las clases y las interfaces (primera letra en mayúscula) así mismo las constantes enum son implícitamente finales y estáticas por tal motivo se utilizan nombres con todas las letras en mayúscula al igual que las constantes en Java.

**Beneficios de las Enums en Java**

1) Enum es lo que se conoce como type-safe, no se puede asignar otro tipo de valor diferente al que se ha predefinido en las constantes Enum a una variable Enum.

2) Enum tiene su propio name-space.

3) Se pueden utilizar dentro de una estructura condicional Switch al igual que los tipos primitivos int o char.

4) Sumar nuevas constantes en un Enum es fácil y tiene menos impacto en el código existente.