MANEJO DE EXCEPCIONES

```
import java.awt.*;
import java.applet.Applet;

public class HolaIte extends Applet {
    private int i = 0;
    private String Saludos[] = {
        "Hola Mundo!",
        "HOLA Mundo!",
        "HOLA MUNDO!!"
        };

    public void paint( Graphics g ) {
        g.drawString( Saludos[i],25,25 );
        i++;
        }
    }
```

Normalmente, un programa termina con un mensaje de error cuando se lanza una excepción. Sin embargo, Java tiene mecanismos para excepciones que permiten ver qué excepción se ha producido e intentar recuperarse de ella.

Vamos a reescribir el método *paint()* de nuestra versión iterativa del saludo:

```
public void paint( Graphics g ) {
    try {
        g.drawString( Saludos[i],25,25 );
        } catch( ArrayIndexOutOfBoundsException e ) {
            g.drawString( "Saludos desbordado",25,25 );
        } catch( Exception e ) {
            // Cualquier otra excepción
            System.out.println( e.toString() );
        } finally {
            System.out.println( "Esto se imprime siempre!" );
        }
    i++;
    }
```

La palabra clave *finally* define un bloque de código que se quiere que sea ejecutado siempre, de acuerdo a si se capturó la excepción o no. En el ejemplo anterior, la salida en la consola, con `i=4` sería:

```
Saludos desbordado
¡Esto se imprime siempre!
```

GENERAR EXCEPCIONES EN JAVA

Cuando se produce un error se debería generar, o lanzar, una excepción. Para que un método en Java, pueda lanzar excepciones, hay que indicarlo expresamente.

```
void MetodoAsesino() throws NullPointerException,CaidaException
```

Se pueden definir excepciones propias, no hay por qué limitarse a las predefinidas; bastará con extender la clase **Exception** y proporcionar la funcionalidad extra que requiera el tratamiento de esa excepción.

También pueden producirse excepciones no de forma explícita como en el caso anterior, sino de forma implícita cuando se realiza alguna acción ilegal o no válida.

Las excepciones, pues, pueden originarse de dos modos: el programa hace algo ilegal (caso normal), o el programa explícitamente genera una excepción ejecutando la sentencia *throw* (caso menos normal). La sentencia throw tiene la siguiente forma:

```
    throw ObtejoExcepction;
```

El objeto ObjetoException es un objeto de una clase que extiende la clase **Exception**.

El siguiente código de ejemplo origina una excepción de división por cero:

```
class melon {
    public static void main( String[] a ) {
        int i=0, j=0, k;

        k = i/j;    // Origina un error de division-by-zero
        }
    }
```

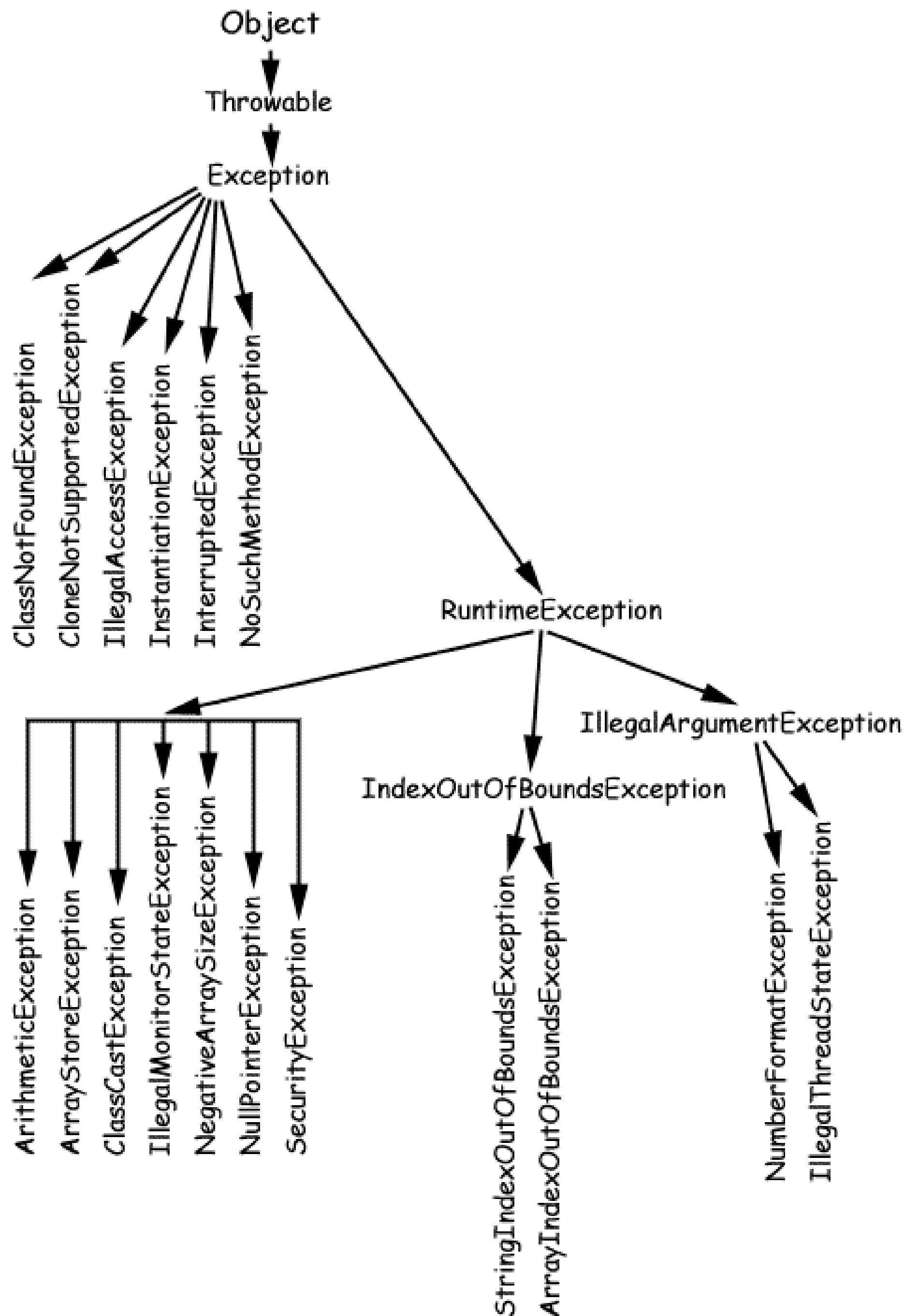
Si compilamos y ejecutamos esta aplicación Java, obtendremos la siguiente salida por pantalla:

```
> javac melon.java
> java melon
     java.lang.ArithmeticException: / by zero
            at melon.main(melon.java:5)
```

Las excepciones predefinidas, como *ArithmeticException*, se conocen como excepciones runtime. Actualmente, como todas las excepciones son eventos runtime, sería mejor llamarlas excepciones irrecuperables. Esto contrasta con las excepciones que generamos explícitamente, que suelen ser mucho menos severas y en la mayoría de los casos podemos recuperarnos de ellas. Por ejemplo, si un fichero no puede abrirse, preguntamos al usuario que nos indique otro fichero; o si una estructura de datos se encuentra completa, podremos sobreescribir algún elemento que ya no se necesite.

EXCEPCIONES PREDEFINIDAS

Las excepciones predefinidas y su jerarquía de clases es la que se muestra en la figura:



Los nombres de las excepciones indican la condición de error que representan. Las siguientes son las excepciones predefinidas más frecuentes que se pueden encontrar:

ArithmeticException

Las excepciones aritméticas son típicamente el resultado de una división por 0:

```
int i = 12 / 0;
```

NullPointerException

Se produce cuando se intenta acceder a una variable o método antes de ser definido:

```
class Hola extends Applet {
    Image img;

    paint( Graphics g ) {
        g.drawImage( img,25,25,this );
        }
    }
```

IncompatibleClassChangeException

El intento de cambiar una clase afectada por referencias en otros objetos, específicamente cuando esos objetos todavía no han sido recompilados.

ClassCastException

El intento de convertir un objeto a otra clase que no es válida.

```
y = (Prueba)x;      // donde
x no es de tipo Prueba
```

NegativeArraySizeException

Puede ocurrir si hay un error aritmético al intentar cambiar el tamaño de un array.

OutOfMemoryException

¡No debería producirse nunca! El intento de crear un objeto con el operador *new* ha fallado por falta de memoria. Y siempre tendría que haber memoria suficiente porque el *garbage collector* se encarga de proporcionarla al ir liberando objetos que no se usan y devolviendo memoria al sistema.

NoClassDefFoundException

Se referenció una clase que el sistema es incapaz de encontrar.

ArrayIndexOutOfBoundsException

Es la excepción que más frecuentemente se produce. Se genera al intentar acceder a un elemento de un array más allá de los límites definidos inicialmente para ese array.

UnsatisfiedLinkException

Se hizo el intento de acceder a un método nativo que no existe. Aquí no existe un método a.kk

```
class A {
    native void kk();
    }
```

y se llama a *a.kk()*, cuando debería llamar a *A.kk()*.

InternalException

Este error se reserva para eventos que no deberían ocurrir. Por definición, el usuario nunca debería ver este error y esta excepción no debería lanzarse.

CREAR EXCEPCIONES PROPIAS

También podemos lanzar nuestras propias excepciones, extendiendo la clase **System.exception**. Por ejemplo, consideremos un programa cliente/servidor. El código cliente se intenta conectar al servidor, y durante 5 segundos se espera a que conteste el servidor. Si el servidor no responde, el servidor lanzaría la excepción de time-out:

```
class ServerTimeOutException extends Exception {}

public void conectame( String nombreServidor ) throws Exception {
    int exito;
    int puerto = 80;

    exito = open( nombreServidor,puerto );
    if( exito == -1 )
        throw ServerTimeOutException;
    }
```

Si se quieren capturar las propias excepciones, se deberá utilizar la sentencia *try*:

```
public void encuentraServidor() {
   ...
   try {
        conectame( servidorDefecto );
        catch( ServerTimeOutException e ) {
```

```
            g.drawString(
                "Time-out del Servidor, intentando alternativa",
                5,5 );
            conectame( servidorAlterno );
            }
    ...
    }
```

Cualquier método que lance una excepción también debe capturarla, o declararla como parte de la interface del método. Cabe preguntarse entonces, el porqué de lanzar una excepción si hay que capturarla en el mismo método. La respuesta es que las excepciones no simplifican el trabajo del control de errores. Tienen la ventaja de que se puede tener muy localizado el control de errores y no tenemos que controlar millones de valores de retorno, pero no van más allá.

CAPTURAR EXCEPCIONES

Las excepciones lanzadas por un método que pueda hacerlo deben recoger en bloque *try/catch* o *try/finally*.

```
int valor;
try {
    for( x=0,valor = 100; x < 100; x ++ )
        valor /= x;
    }
catch( ArithmeticException e ) {
    System.out.println( "Matemáticas locas!" );
    }
catch( Exception e ) {
    System.out.println( "Se ha producido un error" );
    }
```

try

Es el bloque de código donde se prevé que se genere una excepción. Es como si dijésemos "intenta estas sentencias y mira a ver si se produce una excepción". El bloque try tiene que ir seguido, al menos, por una cláusula catch o una cláusula finally

catch

Es el código que se ejecuta cuando se produce la excepción. Es como si dijésemos "controlo cualquier excepción que coincida con mi argumento". En este bloque tendremos que asegurarnos de colocar código que no genere excepciones. Se pueden colocar sentencias catch sucesivas, cada una controlando una excepción diferente. No debería intentarse capturar todas las excepciones con una sola cláusula, como esta:

```
    catch( Excepcion e ) { ...
```

Esto representaría un uso demasiado general, podrían llegar muchas más excepciones de las esperadas. En este caso es mejor dejar que la excepción se propague hacia arriba y dar un mensaje de error al usuario.

Se pueden controlar grupos de excepciones, es decir, que se pueden controlar, a través del argumento, excepciones semejantes. Por ejemplo:

```
class Limites extends Exception {}
class demasiadoCalor extends Limites {}
class demasiadoFrio extends Limites {}
class demasiadoRapido extends Limites {}
class demasiadoCansado extends Limites {}
    .
    .
    .
    try {
        if( temp > 40 )
            throw( new demasiadoCalor() );
        if( dormir < 8 )
            throw( new demasiado Cansado() );
    } catch( Limites lim ) {
        if( lim instanceof demasiadoCalor )
            {
            System.out.println( "Capturada excesivo calor!" );
            return;
            }
        if( lim instanceof demasiadoCansado )
            {
            System.out.println( "Capturada excesivo cansancio!" );
            return;
            }
    } finally
        System.out.println( "En la clausula finally" );
```

La cláusula catch comprueba los argumentos en el mismo orden en que aparezcan en el programa. Si hay alguno que coincida, se ejecuta el bloque. El operador *instanceof* se utiliza para identificar exactamente cual ha sido la identidad de la excepción.

finally

Es el bloque de código que se ejecuta siempre, haya o no excepción. Hay una cierta controversia entre su utilidad, pero, por ejemplo, podría servir para hacer un log o un seguimiento de lo que está pasando, porque como se ejecuta siempre puede dejarnos grabado si se producen excepciones y nos hemos recuperado de ellas o no.

Este bloque finally puede ser útil cuando no hay ninguna excepción. Es un trozo de código que se ejecuta independientemente de lo que se haga en el bloque try.

Cuando vamos a tratar una excepción, se nos plantea el problema de qué acciones vamos a tomar. En la mayoría de los casos, bastará con presentar una indicación de error al usuario y

un mensaje avisándolo de que se ha producido un error y que decida si quiere o no continuar con la ejecución del programa.

Por ejemplo, podríamos disponer de un diálogo como el que se presenta en el código siguiente:

```
public class DialogoError extends Dialog {
    DialogoError( Frame padre ) {
        super( padre,true );
        setLayout( new BorderLayout() );

        // Presentamos un panel con continuar o salir
        Panel p = new Panel();
        p.add( new Button( "¿Continuar?" ) );
        p.add( new Button( "Salir" ) );

        add( "Center",new Label(
            "Se ha producido un error. ¿Continuar?" ) )
        add( "South",p );
        }

    public boolean action( Event evt,Object obj ) {
        if( "Salir".equals( obj ) )
            {
            dispose();
            System.exit( 1 );
            }
        return false;
        }
    }
```

Y la invocación, desde algún lugar en que se suponga que se generarán errores, podría ser como sigue:

```
try {
    // Código peligroso
    }
catch( AlgunaExcepcion e ) {
    VentanaError = new DialogoError( this );
    VentanaError.show();
    }
```

PROPAGACION DE EXCEPCIONES

La cláusula catch comprueba los argumentos en el mismo orden en que aparezcan en el programa. Si hay alguno que coincida, se ejecuta el bloque y sigue el flujo de control por el bloque finally (si lo hay) y concluye el control de la excepción.

Si ninguna de las cláusulas catch coincide con la excepción que se ha producido, entonces se ejecutará el código de la cláusula finally (en caso de que la haya). Lo que ocurre en este

caso, es exactamente lo mismo que si la sentencia que lanza la excepción no se encontrase encerrada en el bloque try.

El flujo de control abandona este método y retorna prematuramente al método que lo llamó. Si la llamada estaba dentro del ámbito de una sentencia try, entonces se vuelve a intentar el control de la excepción, y así continuamente.

Veamos lo que sucede cuando una excepción no es tratada en la rutina en donde se produce. El sistema Java busca un bloque try..catch más allá de la llamada, pero dentro del método que lo trajo aquí. Si la excepción se propaga de todas formas hasta lo alto de la pila de llamadas sin encontrar un controlador específico para la excepción, entonces la ejecución se detendrá dando un mensaje. Es decir, podemos suponer que Java nos está proporcionando un bloque catch por defecto, que imprime un mensaje de error y sale.

No hay ninguna sobrecarga en el sistema por incorporar sentencias try al código. La sobrecarga se produce cuando se genera la excepción.

Hemos dicho ya que un método debe capturar las excepciones que genera, o en todo caso, declararlas como parte de su llamada, indicando a todo el mundo que es capaz de generar excepciones. Esto debe ser así para que cualquiera que escriba una llamada a ese método esté avisado de que le puede llegar una excepción, en lugar del valor de retorno normal. Esto permite al programador que llama a ese método, elegir entre controlar la excepción o propagarla hacia arriba en la pila de llamadas. La siguiente línea de código muestra la forma general en que un método declara excepciones que se pueden propagar fuera de él:

```
tipo_de_retorno( parametros ) throws e1,e2,e3 { }
```

Los nombres *e1,e2,...* deben ser nombres de excepciones, es decir, cualquier tipo que sea asignable al tipo predefinido *Throwable*. Observar que, como en la llamada al método se especifica el tipo de retorno, se está especificando el tipo de excepción que puede generar (en lugar de un objeto exception).

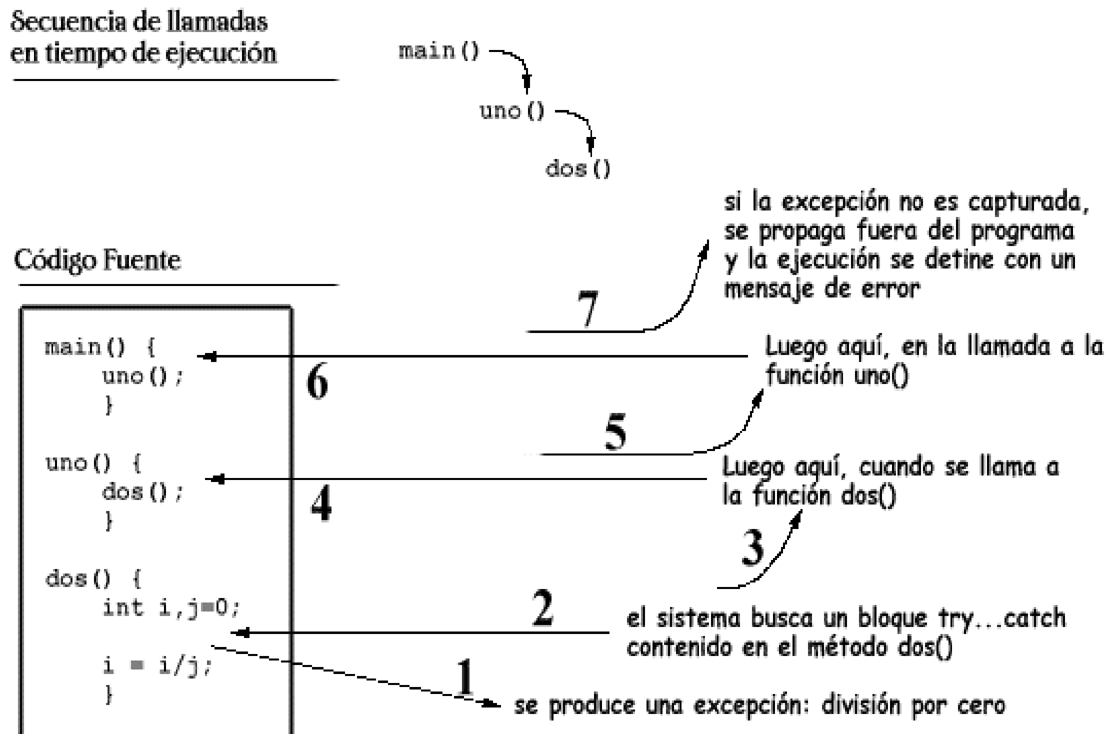
He aquí un ejemplo, tomado del sistema Java de entrada/salida:

```
byte readByte() throws IOException;
short readShort() throws IOException;
char readChar() throws IOException;

void writeByte( int v ) throws IOException;
void writeShort( int v ) throws IOException;
void writeChar( int v ) throws IOException;
```

Lo más interesante aquí es que la rutina que lee un char, puede devolver un char; no el entero que se requiere en C. C necesita que se devuelva un int, para poder pasar cualquier valor a un char, y además un valor extra (-1) para indicar que se ha alcanzado el final del fichero. Algunas de las rutinas Java lanzan una excepción cuando se alcanza el fin del fichero.

En el siguiente diagrama se muestra gráficamente cómo se propaga la excepción que se genera en el código, a través de la pila de llamadas durante la ejecución del código:



Cuando se crea una nueva excepción, derivando de una clase **Exception** ya existente, se puede cambiar el mensaje que lleva asociado. La cadena de texto puede ser recuperada a través de un método. Normalmente, el texto del mensaje proporcionará información para resolver el problema o sugerirá una acción alternativa. Por ejemplo:

```
class SinGasolina extends Exception {
    SinGasolina( String s ) {    // constructor
        super( s );
        }
    ....

// Cuando se use, aparecerá algo como esto
    try {
        if( j < 1 )
            throw new SinGasolina( "Usando deposito de reserva" );
    } catch( SinGasolina e ) {
        System.out.println( o.getMessage() );
        }
```

Esto, en tiempo de ejecución originaría la siguiente salida por pantalla:

```
> Usando deposito de reserva
```

Otro método que es heredado de la superclase **Throwable** es *printStackTrace()*. Invocando a este método sobre una excepción se volcará a pantalla todas las llamadas hasta el momento en donde se generó la excepción (no donde se maneje la excepción). Por ejemplo:

```
// Capturando una excepción en un método
class testcap {
    static int slice0[] = { 0,1,2,3,4 };

    public static void main( String a[] ) {
        try {
            uno();
        } catch( Exception e ) {
            System.out.println( "Captura de la excepcion en main()" );
            e.printStackTrace();
            }
        }

    static void uno() {
        try {
            slice0[-1] = 4;
        } catch( NullPointerException e ) {
            System.out.println( "Captura una excepcion diferente" );
            }
        }
    }
```

Cuando se ejecute ese código, en pantalla observaremos la siguiente salida:

```
> Captura de la excepcion en main()
> java.lang.ArrayIndexOutOfBoundsException: -1
        at testcap.uno(test5p.java:19)
        at testcap.main(test5p.java:9)
```

Con todo el manejo de excepciones podemos concluir que proporciona un método más seguro para el control de errores, además de representar una excelente herramienta para organizar en sitios concretos todo el manejo de los errores y, además, que podemos proporcionar mensajes de error más decentes al usuario indicando qué es lo que ha fallado y por qué, e incluso podemos, a veces, recuperarnos de los errores.

La degradación que se produce en la ejecución de programas con manejo de excepciones está ampliamente compensada por las ventajas que representa en cuanto a seguridad de funcionamiento de esos mismos programas.

Las excepciones en Java están destinadas, al igual que en el resto de los lenguajes que las soportan, para la detección y corrección de errores. Si hay un error, la aplicación no debería morirse y generar un *core* (o un *crash* en caso del DOS). Se debería lanzar (*throw*) una excepción que nosotros deberíamos capturar (*catch*) y resolver la situación de error. Java sigue el mismo modelo de excepciones que se utiliza en C++. Utilizadas en forma adecuada, las excepciones aumentan en gran medida la robustez de las aplicaciones.