UTN.BA FACULTAD REGIONAL BUENOS AIRES
SECRETARÍA DE EXTENSIÓN UNIVERSITARIA FRBA UTN

Centro de e-Learning

EXPERTO UNIVERISTARIO EN MySQL Y PHP NIVEL AVANZADO

www.sceu.frba.utn.edu.ar/e-learning

Módulo I

PHP AVANZADO

Excepciones y manejo de errores.

Presentación de la Unidad:

En esta unidad aprenderán el manejo adecuado de errores en un ambiente de producción. Cual es la mejor manera de tratar los errores en este ambiente.

Aprenderán como se utilizan las excepciones y para que nos son útiles.

Verán un ejercicio práctico de utilidad para tratar los errores.

Objetivos:

- Aprender el manejo de excepciones.
- Aprender a definir nuestro propio handler de errores en lugar de utilizar el manejador de errores propio de PHP.

Temario:

Control de errores sin excepciones
Errores nativos de PHP
Controladores de error
Errores de usuario con trigger_error()
Depuración de errores
Crear un log de Errores
Manejador de errores
Manejo de excepciones
Uso del manejo de excepciones
Limpieza de pila
Aserciones
¿Por qué deberíamos usarlas?
¿Cuándo lanzo una excepción y cuando un error?
¿Cómo lanzo una excepción?
Métodos de una excepción.
Pila de ejecución
Gestión de excepciones
Creando tus propias excepciones
Casos especiales
Errores típicos

CONSIGNAS PARA EL APRENDIZAJE COLABORATIVO

En esta Unidad los participantes se encontrarán con diferentes tipos de consignas que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

1. Los foros asociados a cada una de las unidades.
2. La Web 2.0.
3. Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen las actividades sugeridas y compartan en los foros los resultados obtenidos.

Excepciones y manejo de errores

El PHP incluye en su distribución, como ya sabemos, el archivo php.ini, en donde podemos ubicar los parámetros a través de los cuales definimos cómo se comporta PHP.

Entre ellos como se comporta con respecto a los errores.

error_reporting = E_ALL Fija el nivel (detalle) con el que PHP te informa de errores. Esta directiva vuelca el informe de errores en la pantalla, y su uso está desaconsejado en páginas en producción, ya que el error puede revelar información sensible. Lo recomendado es permitir mostrar errores, con el máximo detalle posible, mientras desarrollas el script PHP; y cuando está terminado y en producción, deshabilitar el mostrado de errores en pantalla y activar en su lugar el archivo de errores.

Como cada nivel de informe de error está representado por un número, puedes designar el nivel deseado sumando valores:

1 errores normales
2 avisos normales
4 errores del parser (error de sintaxis)
8 avisos de estilo no críticos
El valor por defecto para esta directiva es 7 (se muestran los errores normales, avisos normales y errores de parser).

También podemos designar el nivel de error nominativamente:

Algunas combinaciones son:

error_reporting = E_ALL & ~E_NOTICE muestra todos los errores críticos, excluyendo advertencias que pueden indicar mal funcionamiento del código pero no impiden la ejecución del intérprete.

error_reporting = E_COMPILE_ERROR|E_ERROR|E_CORE_ERROR muestra solo errores.

error_reporting = E_ALL muestra todos los errores y advertencias.

display_errors = Off determina si los errores se visualizan en pantalla como parte de la salida en HTML o no.

Como queda dicho, es desaconsejado mostrar errores en pantalla en páginas

visibles al público.

display_startup_errors = Off Incluso con display_errors on, por defecto PHP no muestra los errores que detecta en la secuencia de encendido. Con esta directiva puedes mostrar estos errores. Desaconsejado activarla.

log_errors = On Guarda los mensajes de error en un archivo. Normalmente el registro del servidor. Esta opción, por tanto, es específica del mismo.

log_errors_max_len = 1024 Especifica el tamaño del archivo error_log. Si tiene un valor 0 significa que no hay restricción de tamaño

ignore_repeated_errors = Off Si está activado, no archiva mensajes repetidos. No se consideran mensajes repetidos aquellos que no provienen de la misma línea.

ignore_repeated_source = Off Si está activado, considera repetidos los mensajes de error iguales, aunque provengan de distinta línea / script

report_memleaks = On Mostrar o no. memory leak se refiere a cuando (por error) el script no libera la memoria usada cuando ya no la necesita, y en consecuencia usa cada vez mas hasta llegar a agotarla.

track_errors = Off Si lo activamos, tendremos el último mensaje de error/advertencia almacenado en la variable $php_errormsg

html_errors = Off Si activo, no incluye etiquetas HTML en los mensajes de error.

docref_root = /phpmanual/ y **docref_ext = .html** Si tienes html_errors activado, PHP automaticamente incluye enlaces en el mensaje de error que te dirigen a la página del manual que explica la función implicada. Puedes bajarte una copia del manual y indicar su ubicación (y extensión del archivo) usando estas directivas.

error_prepend_string = "" Cadena a añadir antes de cada mensaje de error.

error_append_string = "" cadena a añadir despues del mensaje de error.

;error_log = filename Nombre del fichero para registrar los errores de un script. Si se utiliza el valor especial syslog, los errores se envían al registro de errores del sistema. Como verás, esta comentado (inhabilitado) por defecto.

Control de errores sin excepciones

En PHP, sabemos que hay multitud de funciones capaces de controlar errores, incluidos los errores nativos de PHP, controladores personalizados de error y los errores de usuario.

Errores nativos de PHP

PHP genera 3 tipos de error, dependiendo de su seriedad:

• Notice: No son errores severos y no crean problemas complejos. Por defecto, no se muestran en pantalla cuando suceden, a menos que digamos lo contrario en el archivo de configuración php.ini

• Warning: Alguna parte del código ha causado un error, pero la ejecución continúa.

• Fatal error: Un problema serio que hace abortar la ejecución del programa.

Cada tipo de error es representado por una constante diferente:

E_USER_NOTICE
E_USER_WARNING
E_USER_ERROR.

Podemos elegir el nivel de error que quiere mostrar en pantalla con la función eror_reporting( )

```
<?php
//Mostrar errores fatales
error_reporting(E_USER_ERROR);
//Mostrar advertencias y notificaciones
error_reporting(E_USER_WARNING | E_USER_NOTICE) ;
//Mostrar todos los errores
error_reporting (E_USER_ALL);
?>
```

El manejo de errores queda limitado la mayoría de veces a los del tipo warning. Los errores fatales se consideran tan críticos que no pueden controlarse y la ejecución del programa queda suspendida.

Controladores de error

Cuando suceda un error que no tenga contemplado en su aplicación, pueden aparecer mensajes en la pantalla que no interesa mostrar. La solución es crear su propia función para controlar y mostrar errores.

```
<?php
function errores($error,{$mensaje,$fichero,$linea) {
echo "<b>: : ERROR :: </b><br>" ;
echo "Sentimos comunicarle que se ha producido un error";
echo "Tipo de error: $error: $mensaje en $fichero en la línea
$linea";
}
?>
```

Con estas líneas hemos creado una función que muestra el error producido, pero con un diseño acorde a nuestra página Web. Puede mostrar tanta información como quiera. El siguiente paso es llamar a la función set_error_handler() para que PHP sepa que nuestra función errores() se va a encargar a partir de ahora de gestionar los posibles fallos en el flujo del programa.

```
<?php
set_error_handler("errores");
?>
```

Con el código anterior, todos los errores producidos del nivel que tenga permitido (fatal error , warning notice) serán enviados a la función errores() para mostrar su mensaje particular.

Errores de usuario con trigger_error()

Desde PHP 4, puede usarse una función para identificar los errores producidos por los usuarios, muy parecida a la función de PHP 5 throw() .
La función trigger_error() genera un error del tipo que quiera (fatal error , warning o

notice) y será gestionado por PHP o por su propio gestor de errores.

```
<?php
error_reporting(E_ALL)
;
function errores($error,$mensaje,$fichero,$linea) {
echo "<b>: :ERROR: :</b><br/>" ;

echo "Sentimos comunicarle que se ha producido un error";
echo "Tipo de error: $error: $mensaje en $fichero
en la línea $ linea";
}
set_error_handler("errores") ;
if (1 != 0 ) {
trigger_error(" Esto es falso ",E_USER_NOTICE);
}
?>
```

Es mejor utilizar siempre la función trigger_error( ) con set_error_handler (El error que creemos puede ser de cualquiera de los 3 tipos existentes en PHP. En este caso, generamos un error notice para indicar que la condición es falsa.

Depuración de errores

El manejo de errores puede ahorrar esfuerzos a la hora de depurar el funcionamiento óptimo de su código, pero también puede crear algunos dolores de cabeza si empieza a suprimir algunos fallos.
La función error_log() crea un archivo donde se irán almacenando las ocurrencias de su código. Con el siguiente ejemplo podrá capturar todos los errores producidos en el programa, para después hacer un estudio de lo que ha pasado.

Crear un log de Errores

Como crear un archivo que almacena los errores que se han producido durante la ejecución de un programa, añadir un log de errores a nuestra página.
Un log de errores, nos permitirá controlar cuando se ha producido un error para corregirlo y evitar que se repita en el futuro.

Para crear un log, abriremos el archivo en modo 'a' (escritura al final) y escribiremos el error indicando la fecha, para simplificar el trabajo lo podemos incluir todo en una función:

```
<?php
function error($numero,$texto){
$ddf = fopen('error.log','a');
fwrite($ddf,"[".date("r")."] Error $numero: $texto\r\n");
fclose($ddf);
}
?>
```

Una vez declarada la función, tansolo tendremos que llamarla de la siguiente manera cuando se produzca un error para que se guarde en error.log:

```
<?php
// Si no existe la cookie
sesion
if(!isset($_COOKIE['sesion'])){
// Guardamos un error
error('001','No existe la cookie de sesion');
}
?>
```

De esta manera, cada vez que un usuario entra a esta página sin la cookie sesion, se almacena una nueva línea en el fichero indicando:

[fecha] Error 001: No existe la cookie de sesión

Vamos a ver ahora como podemos mejorar esto de manera que además de poder grabar los errores que nosotros definamos en nuestro sitio, nos almacene los errores producidos durante la ejecución del script php.

Esto lo conseguiremos indicando al intérprete que llame a la función error() cada vez que el código PHP contenga un error con la función set_error_handler:

```
<?php
set_error_handler('error');
?>
```

Entonces, el código completo nos queda de la siguiente manera:

```
<?php
function error($numero,$texto){
        $ddf = fopen('error.log','a');
        fwrite($ddf,"[".date("r")."] Error $numero:$texto\r\n");
        fclose($ddf);
}
set_error_handler('error');
?>
```

Y de esta manera damos por finalizado nuestro script para generar un log de errores personales y de PHP.

Los ficheros log *son muy utilizados en los Sistemas Operativos para mantener un registro de incidencias. Los ficheros* log *guardan desde el número de veces que ha entrado un usuario, hasta los errores que se han producido en el inicio del sistema.*

La función error_log() puede pasar como segundo parámetro un número entero que identifica el tipo de error y qué hacer con él.

0: Se utiliza el mecanismo del sistema operativo.
1: Envía el error a una dirección de correo especificada como cuarto parámetro. (Debe formar parte de una cabecera).
2: Envía el error a través de una sesión de depuración de PHP (Debe estar habilitado el modo depuración)
3: Envía el error a un fichero.

Manejador de errores

Como sabemos, un manejador, no es más que una función que recibe como parámetros los eventos que han provocado un fallo en la aplicación, podemos hacer como el ejemplo anterior y crearnos nuestro propio manejador de errores, sustituyendo al que viene por defecto en PHP. Un manejador de errores personalizado más depurado podría ser el siguiente:

```
<?php

function myErrorHandler($errno, $errstr, $errfile, $errline)
{
switch ($errno) {

        case E_USER_ERROR:
                echo "<b>ERROR</b> [$errno] $errstr<br />\n";
                echo "  Error en linea $errline in file $errfile";
                echo ", PHP " . PHP_VERSION . " (" . PHP_OS . ")<br />\n";
                echo "Abortando...<br />\n";
                exit(1);
                break;

        case E_USER_WARNING:
                echo "<b>Warning</b> [$errno] $errstr<br />\n";
                break;

        case E_USER_NOTICE:
                echo "<b>Notice</b> [$errno] $errstr<br />\n";
                break;

        case E_ERROR:
                echo "Error fatal...<br>";
                break;

        default:
                echo "Error desconocido, valor: [$errno] $errstr<br />\n";
                break;
}
return true;
}

?>
```

Con este manejador, podremos controlar los fallos que se producen en nuestra aplicación y obrar en consecuencia. Es obligado decir que no podremos capturar todos los fallos posibles y que si no somos un poco selectivos es posible que muestre demasiados errores.

Los errores más importantes que podemos encontrarnos en PHP versión 5 son:

- **E_ERROR** : Errores fatales en tiempo de ejecución
- **E_WARNING**: Advertencia en tiempo de ejecución (no son fatales)
- **E_PARSE**: Errores fatales en tiempo de compilación
- **E_NOTICE**: Avisos en tiempo de ejecución (menos importantes que las advertencias)

Si necesitásemos controlar alguno que no fuera ninguno de estos, utilizarías el default de nuestro manejador para obtener su identificador de error y lo controlaríamos añadiendo otro case al bucle.

Para que entre en funcionamiento nuestro manejador tendremos que decírselo al servidor, para ello utilizaremos la función set_error_handler() a la cual le pasaremos como parámetro el nombre de nuestra función que hará de manejador de errores.

```
set_error_handler('myErrorHandler');
```

Si aparte de todo esto, queremos evitar que salga por pantalla cualquier error propio de php, bastará con colocar en los archivos correspondientes la instrucción

```
error_reporting(0);
```

Si queremos completarlo aún un poco más, definimos una función de gestión de errores que registra la información en un archivo (usado un foramto XML), y envía un

e-mail a los desarrolladores en caso de que suceda un error crítico en la lógica.

```
<?php
// haremos nuestra propia manipulación de
errores error_reporting(0);

// función de gestión de errores definida por el usuario
function gestorDeErroresDeUsuario($num_err, $mens_err, $nombre_archivo,
                        $num_linea, $vars)
{

   // marca de fecha/hora para el registro de error
   $dt = date("Y-m-d H:i:s (T)");

   // definir una matriz asociativa de cadenas de error
   // en realidad las únicas entradas que deberíamos
   // considerar son E_WARNING, E_NOTICE, E_USER_ERROR,

   // E_USER_WARNING y E_USER_NOTICE

   $tipo_error = array (
            E_ERROR              => 'Error', E_WARNING
                                    => 'Advertencia',
            E_PARSE              => 'Error de Intérprete',
            E_NOTICE             => 'Anotación',
            E_CORE_ERROR           => 'Error de
            Núcleo',
            E_CORE_WARNING           => 'Advertencia de Núcleo',
            E_COMPILE_ERROR          => 'Error de Compilación',
            E_COMPILE_WARNING  => 'Advertencia de Compilación',
            E_USER_ERROR           => 'Error de Usuario',
            E_USER_WARNING           => 'Advertencia de Usuario',
            E_USER_NOTICE            => 'Anotación de Usuario',
            E_STRICT             => 'Anotación de tiempo de
            ejecución', E_RECOVERABLE_ERROR  => 'Error Fatal
            Atrapable'
            );
   // conjunto de errores de los cuales se almacenará un rastreo
   $errores_de_usuario = array(E_USER_ERROR, E_USER_WARNING, E_USER_N
OTICE);

   $err = "<errorentry>\n";
   $err .= "\t<datetime>" . $dt . "</datetime>\n";
   $err .= "\t<errornum>" . $num_err . "</errornum>\n";
   $err .= "\t<errortype>" . $tipo_error[$num_err] . "</errortype>\n";
```

```
    $err .= "\t<errormsg>" . $mens_err . "</errormsg>\n";
    $err .= "\t<scriptname>" . $nombre_archivo . "</scriptname>\n";
    $err .= "\t<scriptlinenum>" . $num_linea . "</scriptlinenum>\n";

    if (in_array($num_err, $errores_de_usuario)) {
        $err .= "\t<vartrace>" . wddx_serialize_value($vars, "Variables") .
        "</vartrace>\n";
    }
    $err .= "</errorentry>\n\n";

    // para efectos de depuración
    // echo $err;

    // guardar en el registro de errores, y enviar un
    correo
    // electrónico si hay un error crítico de
    usuario error_log($err, 3,
    "/usr/local/php4/error.log"); if ($num_err
    == E_USER_ERROR) {
        mail("phpdev@example.com", "Error Crítico de Usuario", $err);
    }
}


function distancia($vect1, $vect2)
{
    if (!is_array($vect1) || !is_array($vect2)) {
        trigger_error("Parámetros incorrectos, se esperan matrices", E_USER_ERROR);

        return NULL;
    }

    if (count($vect1) != count($vect2)) {
        trigger_error("Los vectores deben ser del mismo tamaño", E_USER_ERROR);
        return NULL;
    }

    for ($i=0; $i<count($vect1); $i++) {
        $c1 = $vect1[$i]; $c2 = $vect2[$i];
        $d = 0.0;
        if (!is_numeric($c1)) {
            trigger_error("La coordenada $i en el vector 1 no es
                    un ". "número, se usará cero",
                      E_USER_WARNING);
            $c1 = 0.0;
        }
        if (!is_numeric($c2)) {
            trigger_error("La coordenada $i en el vector 2 no es
```

```
                    un". "número, se usará cero",
                    E_USER_WARNING);
            $c2 = 0.0;
        }
        $d += $c2*$c2 - $c1*$c1;
    }
    return sqrt($d);
}

$gestor_de_errores_anterior = set_error_handler("gestorDeErroresDeUsuario");

// constante indefinida, se genera una advertencia
$t = NO_ESTOY_DEFINIDA;

// definir algunos "vectores"
$a = array(2, 3, "foo");
$b = array(5.5, 4.3, -1.6);
$c = array(1, -3);

// generar un error de usuario
$t1 = distancia($c, $b) . "\n";

// generar otro error de usuario
$t2 = distancia($b, "no soy una matriz") . "\n";

// generar una advertencia
$t3 = distancia($a, $b) . "\n";

?>
```

Más allá de las funciones que ya mencionamos, existen otras relacionadas al manejo de errores en PHP, ponemos a continuación la lista completa:

debug_backtrace - Genera un rastreo **debug_print_backtrace** - Muestra un rastreo
error_get_last - Obtener el último error que ocurrió
error_log - Enviar un mensaje de error a algún lugar
error_reporting - Establece qué errores de PHP son notificados
restore_error_handler - Recupera la función de gestión de errores previa
restore_exception_handler - Restaura la función de gestión de excepciones previamente definida
set_error_handler - Establecer una función de gestión de errores definida por el usuario
set_exception_handler - Establece una función de gestión de excepciones definida por el usuario
trigger_error - Generar un mensaje error/warning/notice de nivel de usuario
user_error - Alias de trigger_error

Manejo de excepciones

Una excepción en términos de lenguaje de programación es la indicación de un problema que ocurre durante la ejecución de un programa. Sin embargo la palabra excepción se refiere que este problema ocurre con poca frecuencia generalmente cuando existe algún dato o instrucción que no se apega al funcionamiento del programa por lo que se produce un error.

El manejo de excepciones permite al usuario crear aplicaciones tolerantes a fallas y robustos (resistentes a errores) para controlar estas excepciones y que pueda seguir ejecutando el programa sin verse afectado por el problema.

Uso del manejo de excepciones

El manejo de excepciones ayuda al programador a remover el código para manejo de errores de la línea principal de ejecución, además se pude elegir entre manejar todas las excepciones, las de cierto tipo o de las de grupos relacionados, esto hace que la probabilidad de pasar por alto los errores se reduzca y a la vez hace los programas más robustos.

Pero es importante utilizar un lenguaje de programación que soporte este manejo, de lo contrario el procesamiento de errores no estará incluido y hará el programa más vulnerable. Este manejo está diseñado para procesar errores que ocurren cuando se ejecuta una instrucción, algunos ejemplos son: desbordamiento aritmético, división

entre cero, parámetros inválidos de método y asignación fallida en la memoria. Sin embargo no está diseñado para procesar problemas con eventos independientes al programa como son pulsar una tecla o clic al mouse.

Las excepciones se dividen en verificadas y no verificadas. Es importante esta división porque el compilador implementa requerimientos de atrapar o declarar para las verificadas lo que hará que se detecten las excepciones automáticamente y de acuerdo al lenguaje de programación utilizado se utilizará un método para corregirlas. Sin embargo para las no verificadas se producirá un error indicando que deben atraparse y declararse. Por eso el programador debe pensar en los problemas que pueden ocurrir cuando se llama a un método y definir excepciones para verificarse cuando sean importantes.

Las clases de excepciones pueden derivarse de una superclase común, por lo que con un manejador para atrapar objetos de la superclase, también se pueden atrapar

todos los objetos de las subclases de esa clase. Pero también, se pueden atrapar a cada uno de los tipos de las subclases de manera individual si estas requieren ser procesadas diferente.

Limpieza de pila

En ocasiones cuando se hace lanza una excepción, pero no se atrapa en un enlace específico, la pila de llamadas se limpia y el programa intenta volverlo a atrapar en el siguiente bloque, esto se conoce como limpia de pila. Este proceso hace que el método en el que no se atrapó la excepción termina, todas sus variables quedan fuera del enlace y el control regresa a la instrucción que originalmente la invocó. La limpieza de pila de repetirá hasta que la excepción pueda ser atrapada porque de lo contrario se producirá un error a la hora de compilar.

Aserciones

Las aserciones ayudan a asegurar la validez del programa al atrapar los errores potenciales e identificar los posibles errores lógicos del desarrollo. Estas pueden escribirse como comentarios para apoyar a la persona que desarrolla el programa. Algunos ejemplos son: Precondiciones y pos condiciones.

Estas características son utilizadas por los programadores para hacer un análisis de lo esperado del programa antes y después de su ejecución. Son importantes porque gracias a ellas se pueden detectar posibles fallas en el programa y corregirlas. Las precondiciones son verdaderas cuando se invoca a un método, estas describen las características del método y las expectativas que se tienen en el estado actual del programa.

Si no se cumplen las precondiciones el comportamiento del método es indefinido por lo que se lanza una excepción que esté preparada o continuar con el programa esperando el error. Las pos condiciones describen las restricciones en el entorno y cualquier efecto secundario del método. Es recomendable escribirlas para saber que esperar en un futuro si es que se hacen modificaciones.

Resumamos entonces ¿Que es una excepción?

Una excepción es un evento que ocurre durante la ejecución de un programa y requiere de la ejecución controlada de un bloque de código fuera del flujo de normal de ejecución

El manejo de excepciones es una herramienta muy potente a la hora de realizar una

gestión de una situación. Lo primero que hay que comprender es que una excepción no es un error. Es una situación que se experimenta un bloque de código y que este no es capaz de manejar.

¿Por qué deberíamos usarlas?

Un correcto manejo de excepciones, pueden hacer su código mucho más fiable ahorrándonos dolores de cabeza.

¿Cuándo lanzo una excepción y cuando un error?

Depende un poco de la política del proyecto, en el que estemos trabajando, y los gustos de cada programador. Como con los nombres de las variables, lo mejor es llegar a una especie de estándar dentro de cada equipo.

En general si estamos trabajando en una clase de alto nivel de una aplicación y muy acoplada con la misma, la mejor opción puede ser informar del error, mediante la gestión de errores de la propia aplicación, y tratar de controlarlo en este mismo lugar. Si por el contrario estamos trabajando en una clase de bajo nivel, desacoplada de la aplicación en la que estás trabajando y susceptible de ser reutilizada, lo mejor es lanzar una excepción y dejar que las clases de arriba traten de capturarlo, para informar al usuario.

¿Cómo lanzo una excepción?

Una excepción se lanza a través de la palabra reservada **thrown**, veamos un ejemplo.

```
?php
function foo(){

    throw new Exception (' Upps !!');
 }
 foo();


 // Salida:  PHP Fatal error:  Uncaught exception 'Exception' with message ' Upps !!'
```

Cuando una excepción es lanzada PHP tratará de encontrar un bloque **catch{}** capaz de capturarla, si no encuentra ninguno, entonces, se interrumpirá la ejecución y se emitirá un error fatal de PHP con el mensaje "Uncaught Exception …".

Para "capturar" una excepción el código debe encontrarse dentro de un bloque **try{}** y debe tener al menos un bloque **catch{}**. Veamos un ejemplo.

```
<?php
function foo() {
    try {
        throw new Exception(' Upps !!');
    } catch (Exception $e) {
        echo "<h1>El programa no se detiene y muestra este texto...</h1>";
    }
}

foo();
?>
```

En este segundo ejemplo la ejecución del programa no se interrumpe a pesar de que se ha lanzado una excepción.

Veamos otro ejemplo más con varios bloques **catch:**

```
<?php
function foo() {
    try {

        throw new Exception(' Upps !!');
    } catch (ErrorException $e) {
        // este bloque no se ejecuta, no coincide el tipo de
        excepción echo 'Excepción capturada: ',$e-
        >getMessage();
    } catch (Exception $e) {
        // este bloque captura la excepción
        echo 'Excepción capturada: ',$e->getMessage();
```

```
    }
}
 foo();

?>
```

Métodos de una excepción.

Una excepción es un objeto al fin y al cabo, por ende posee propiedades y métodos, a continuación vemos los métodos que tiene disponibles:

- getMessage() devuelve el mensaje de la excepción.
- getCode() devuelve el código del error.
- getFile() devuelve el nombre del fichero que lanzó la excepción.
- getLine() devuelve el la línea del fichero que lanzó la excepción.
- getTrace() devuelve la pila de ejecución en formato de array.
- getTraceAsString() devuelve la pila de ejecución en formato de string.

Ejemplo:

```
<?php

function foo() {

    try {

        throw new Exception(' Upps !!');

    } catch (ErrorException $e) {

        // este bloque no se ejecuta, no coincide el tipo de excepción, Exception
!= ErrorException

        echo 'Excepción capturada: ',$e->getMessage();

    } catch (Exception $e) {

        // este bloque captura la excepción

                                                                    echo
```

```
        'Mensaje de Excepción: ',$e->getMessage();
                echo "<br/>";
                echo 'Código de Excepción: ',$e->getCode();
                echo "<br/>";
                echo 'Archivo donde surge la Excepción: ',$e->getFile();
                echo "<br/>";
                echo 'Línea de  la Excepción: ',$e->getLine();
                echo "<br/>";
                echo 'Pila de ejecución en formato array: ',$e->getTrace();
                echo "<br/>";
                echo 'Pila de ejecución en formato string: ',$e->getTraceAsString();
    }
}
 foo();
?>
```

Pila de ejecución

Hagamos un breve paréntesis para definir el concepto de pila de ejecución mencionado en el ejemplo anterior.

Una pila de llamadas (en inglés call stack) es una estructura dinámica de datos LIFO, (una pila), que almacena la información sobre las subrutinas activas de un programa de computadora. Esta clase de pila también es conocido como una pila de ejecución, pila de control, pila de función, o pila de tiempo de ejecución, y a menudo se describe en forma corta como "la pila".

Una pila de llamadas es de uso frecuente para varios propósitos relacionados, pero la principal razón de su uso, es seguir el curso del punto al cual cada subrutina activa debe

retornar el control cuando termine de ejecutar. (Las subrutinas activas son las que se

han llamado pero todavía no han completado su ejecución ni retornando al lugar siguiente desde donde han sido llamadas).

Si, por ejemplo, una subrutina DibujaCuadrado llama a una subrutina DibujaLinea desde cuatro lugares diferentes, el código de DibujaLinea debe tener una manera de saber a dónde retornar. Esto es típicamente hecho por un código que, para cada llamada dentro de DibujaCuadrado, pone la dirección de la instrucción después de la sentencia de llamada particular (la "dirección de retorno") en la pila de llamadas.

Puesto que la pila de llamadas se organiza como un pila, el programa que llama (a la subrutina) empuja (push) la dirección de retorno en la pila (la dirección en la que el programa continuará después de volver de la subrutina), y la subrutina llamada, cuando finaliza, retira (pop) la dirección de retorno de la pila de llamadas y transfiere el control a esa dirección.

Si una subrutina llamada, llama a otra subrutina, la primera empuja (push) su dirección de retorno en la pila de llamadas, y así sucesivamente, con la información de direcciones de retorno apilándose y desapilándose como el programa dicte. Si el empujar (push) consume todo el espacio asignado para la pila de llamadas, ocurre un error llamado desbordamiento de pila. Agregando una entrada de subrutina a la pila de llamadas es a veces llamado bobinando o enrollando (winding); inversamente, la eliminación de entradas es llamado desenbobinando o desenrollando (unwinding).

Usualmente hay exactamente una pila de llamadas asociado a un programa en ejecución (o más precisamente, con cada tarea o hilo de un proceso), aunque pilas adicionales pueden ser creados para el manejo de señales o la multitarea cooperativa (como con setcontext). Puesto que hay solamente una pila en este importante contexto, puede ser referido como la pila (implícito, "de la tarea").

En lenguajes de programación de alto nivel, las características específicas de la pila de llamadas usualmente son ocultadas al programador. Ellos solo tienen acceso a la lista de funciones, y no a la memoria de la pila en sí misma. La mayoría de los lenguajes ensambladores, por una parte, requieren que los programas sean invocados con manipulación explícita de la pila. En un lenguaje de programación, los detalles reales de la pila dependen del compilador, del sistema operativo, y del conjunto de instrucciones disponible.

Veamos otro ejemplo

```
<?php
try {
   if (!$handle = fopen('archivo.txt', "w+"))
      throw new Exception('no se puede acceder al archivo');
   if (fwrite($handle2,'contenido ..') === false)
      throw new Exception('no se puede escribir en el archivo');
   fclose($handle);
   echo 'archivo creado satifactoriamente ..';
}
catch (Exception $e){
   echo 'Excepción capturada: ',$e->getMessage();
}
echo ' - independientemente del error, imprimirá esta linea..';
?>
```

Devuelve en este caso:

"Excepción capturada: no se puede escribir en el archivo ...."

debido a que no se le pasa un valor correcto al primer parámetro de fwrite.

Otro ejemplo más mundano

```
<?php
function division($dividendo, $divisor) {
   if ($divisor == 0) {
      throw new Exception('Imposible dividir por 0');
   }
   else return $dividendo/$divisor;
}

try {
   echo  division(10,  5)  .  "<br
   />";  echo  division(25,  3)  .
   "<br />"; echo division(8, 0) .
   "<br />";
} catch (Exception $e) {
   echo  'Excepción  capturada:  ',   $e->getMessage(),
   "\n";
}

?>
```

Resultado:

```
2
8.33333333333
Excepción capturada: Imposible dividir por 0
```

Otro ejemplo:

```
<?php

try {

if(!@include('fichero.php')) {

throw new Exception('Error al cargar el fichero');

}

}
```

```
catch(Exception $e) {

print $e->getMessage();

}

?>
```

Gestión de excepciones

Gracias a **set_exception_handler()** podemos controlar las excepciones no capturadas por bloques try/catch.

Gracias a esta característica de PHP, podemos configurar que hacer cuando ocurre una excepción, como por ejemplo:

- Generar un log.
- Mostrar un mensaje de error de forma controlada.
- Enviar información completa a los administradores del sistema, para que puedan depurarlo.

Veamos un ejemplo genérico para set_exception_handler:

```
<?php

/*
 * recupera información de la
 * maquina en la que se está efecutando
 *
 * @return string $info
 */

function getInfoMachine() { /* ... */
}

/*
 * recupera información de la
 * variables, $_POST, $_GET, $_SESSION
```

```
 *
 * @return string $info
 */

function getInfoEnviroment() { /* ... */
}

/*
 * recupera información del usuario que
 * está ejecutando el proceso, en una aplicación
 * con gestión de usuarios.
 *
 * @return string $info
 */

function getInfoUser() { /* ... */
}

/*
 * recupera información de la excepción
 *
 * @param Exception $e
 * @return string $info
 */

function getInfoException(Exception $e) { /* ... */
}

/*
 * envia informacion a los administradores del sistema
 *
 * @param Exception $e
 * @return true or false
 */

function exception_handle(Excepion $e) {
   $text = getInfoMachine() .
         getInfoEnviroment() . getInfoUser() .
         getInfoException($e);
   mail($admins, $e-&gt;getMessage(), $message);
}

set_exception_handler('exception_handler');
```

Creando tus propias excepciones

¿Por que querríamos crear un nuevo tipo de excepción? Muy fácil, para contemplar situaciones concretas, si por ejemplo estoy haciendo una librería que realiza tratamiento de imágenes, puedo querer añadir información extra a la excepción, relativa al tratamiento de imágenes.

```
<?php

class MyException extends Exception

{/*

 ...

*/}


?>
```

De esta manera estamos indicando que nuestra clase MyException, heredará propiedades y métodos de la clase Exception.

Casos expeciales

A continuación veremos algunos casos especiales que debemos conocer.

Errores típicos

Un error típico cuando se trabaja con excepciones, y no se tiene experiencia, es omitir bloques de código que con necesarios al capturar excepciones, veamos un ejemplo más.

```
<?php
function foo() {
    DB::query('LOCK TABLES `a`, `b` WRITE');
    try {
        /* operaciones SQL */
    } catch (Exception $e) {
        /* operaciones SQL */
    }
    DB::query('UNLOCK TABLES');
```

```
}
?>
```

En este ejemplo vemos como el código que desbloquea las tablas, se encuentra fuera del bloque catch, por lo que si ocurre una excepción las tablas se quedarán bloqueadas. Por lo tanto el código catch, debe ocuparse de liberar las tablas.

```
<?php
function foo() {
    DB::query('LOCK TABLES `a`, `b` WRITE');
    try {
        /* operaciones SQL */
    } catch (Exception $e) {
        /* operaciones SQL */
            DB::query('UNLOCK TABLES');
    }
    DB::query('UNLOCK TABLES');
}
?>
```

Aclaración

Como ya dijimos, desde PHP 5 podemos hacer el uso de excepciones como casi en cualquier lenguaje de programación, pero hay que tener en cuenta que aunque las excepciones sean una gran herramienta, también podrían ser peligrosas cuando son usadas indebidamente.

Veamos un ejemplo del mal uso de excepciones:

```
<?php
function foo() {
    if ($usuario !== "Hola") {
        throw new Exception("Usuario incorrecto");
    }
}
?>
```

Esto es totalmente incorrecto, ya que este es un error que podríamos manejarlo nosotros mismos, es un error de usuario y no de la aplicación. **Las excepciones deben ser usadas sólo cuando nuestra aplicación termine debido a un estado excepcional**.