SPL for the masses

Marcus Börger

What is SPL

- ☑ A collection of standard interfaces and classes
- ☑ A few helper functions

What is SPL about & what for

- ☑ Captures some common patterns
  - ☑ More to follow
- ☑ Advanced Iterators
- ☑ Functional programming
- ☑ Exception hierarchy with documented semantics
- ☑ Makes __autoload() useable

What are Iterators

- Iterators are a concept to iterate anything that contains other things. Examples:
  - Values and Keys in an array
  - Text lines in a file
  - Database query results
  - Files in a directory
  - Elements or Attributes in XML
  - Bits in an image
  - Dates in a calendar range
- Iterators allow to encapsulate algorithms

The basic concepts

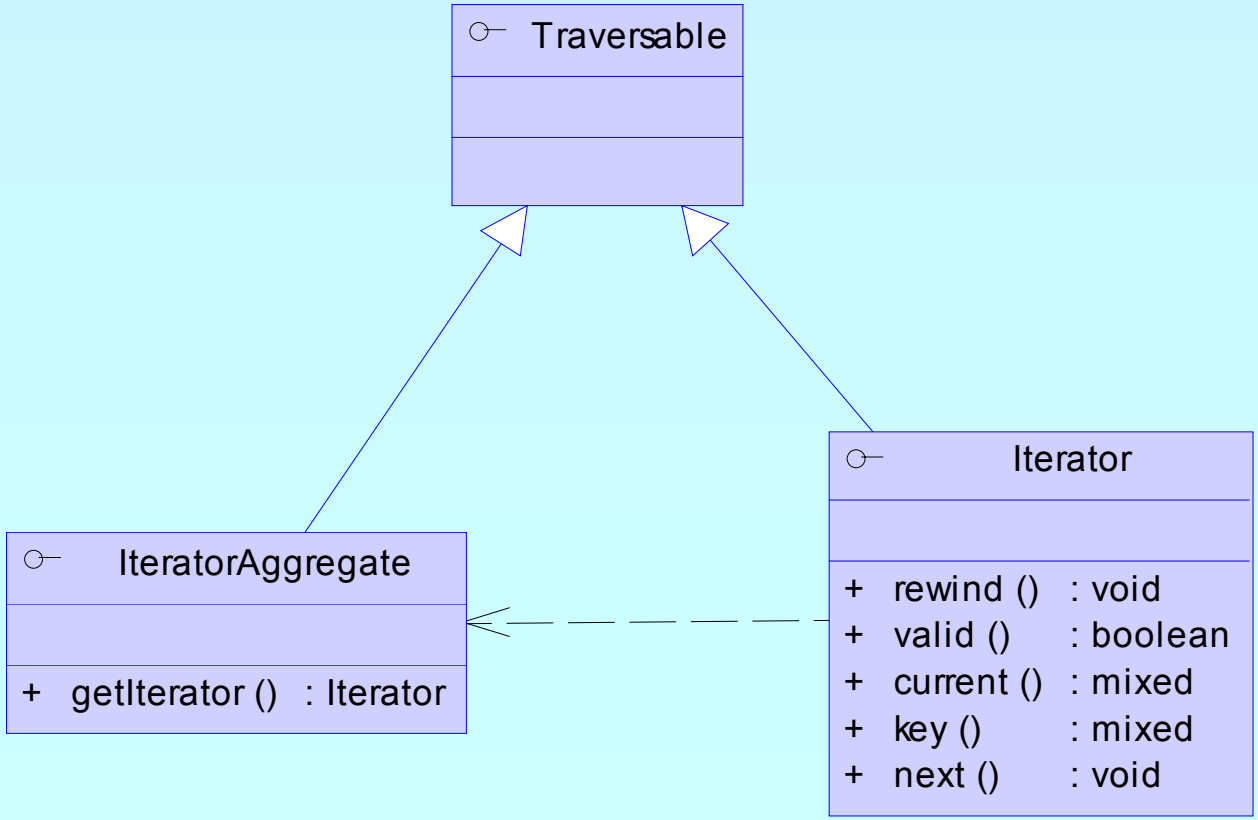
- ☑ Iterators can be internal or external
  also referred to as active or passive
- ☑ An internal iterator modifies the object itself
- ☑ An external iterator points to another object without modifying it
- ☑ PHP always uses external iterators at engine-level

The big difference

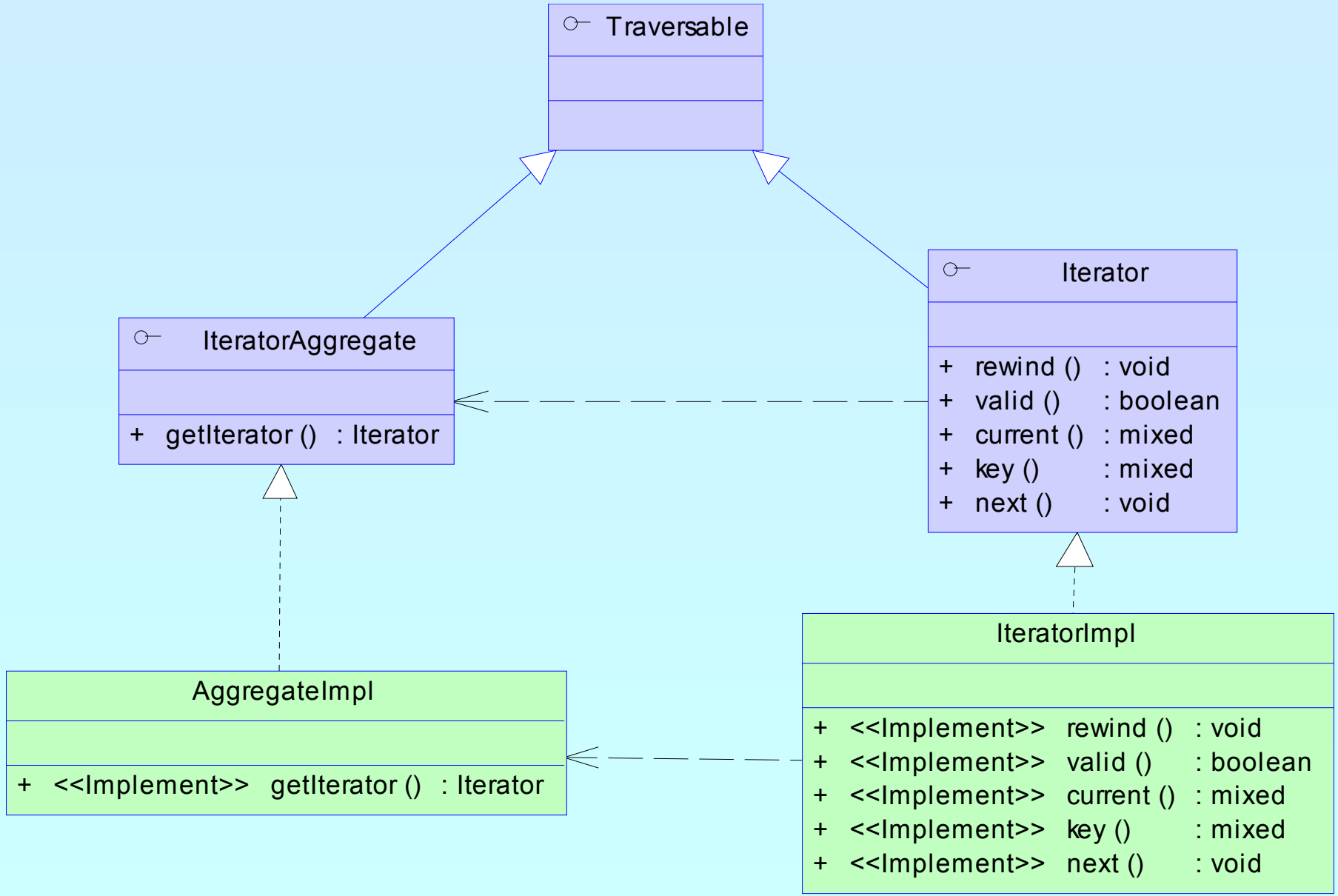
- ☑ **Arrays**
  - ☑ require memory for all elements
  - ☑ allow to access any element directly
- ☑ **Iterators**
  - ☑ only know one element at a time
  - ☑ only require memory for the current element
  - ☑ forward access only
  - ☑ Access done by method calls
- ☑ **Containers**
  - ☑ require memory for all elements
  - ☑ allow to access any element directly
  - ☑ can create external Iterators or are internal Iterators

PHP Iterators

- ☑ Anything that can be iterated implements `Traversable`
- ☑ Objects implementing `Traversable` can be used in `foreach`
- ☑ User classes cannot implement `Traversable`
- ☑ `Aggregate` is used for objects that use external iterators
- ☑ `Iterator` is used for internal traversal or external iterators



Implementing Iterators



How Iterators work

- ☑ Iterators can be used manually

```php
<?php
$o = new ArrayIterator(array(1, 2, 3));
$o->rewind();
while ($o->valid()) {
    $key = $o->key();
    $val = $o->current();
    // some code
    $o->next();
}
?>
```

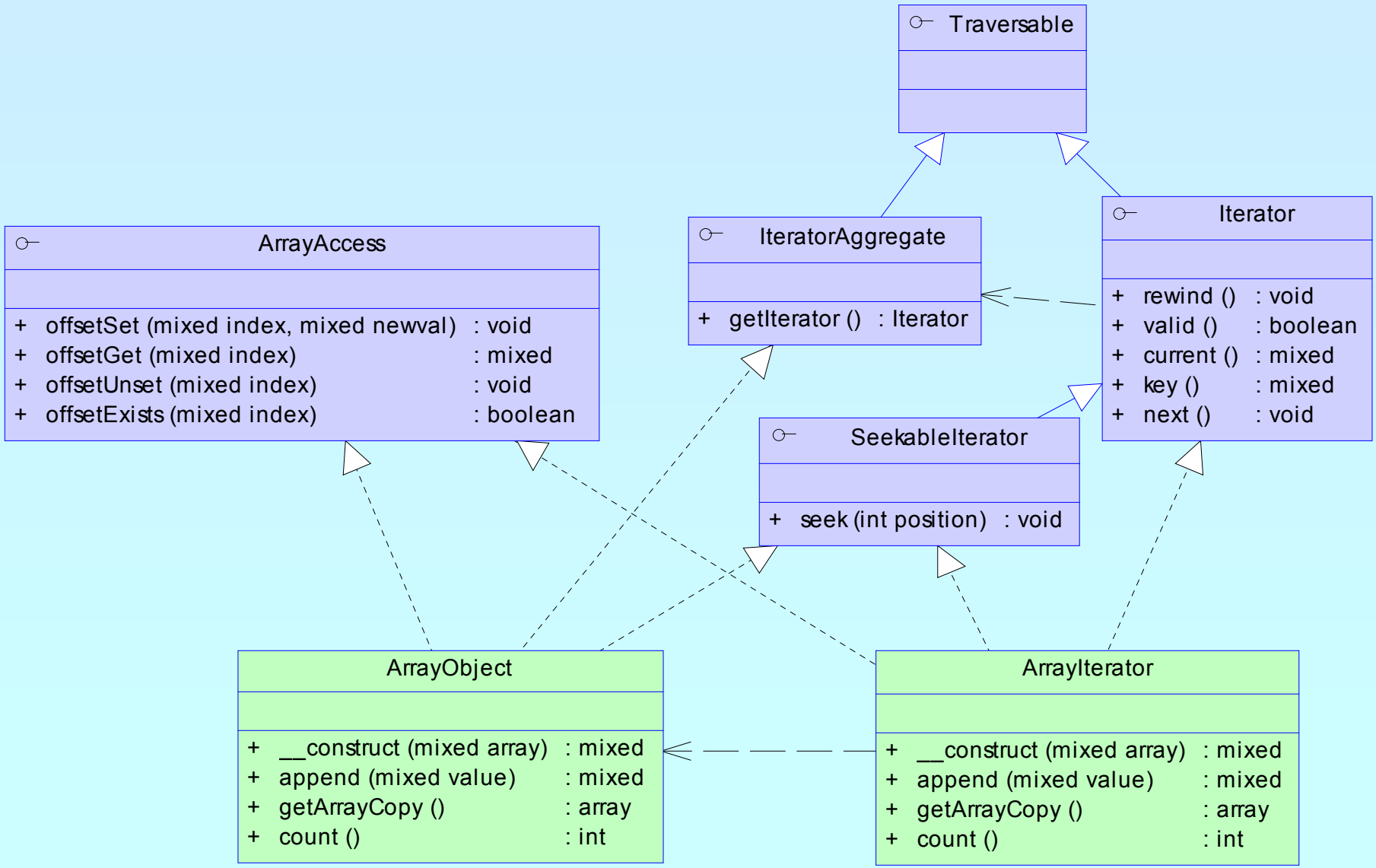
- ☑ Iterators can be used implicitly with **foreach**

```php
<?php
$o = new ArrayIterator(array(1, 2, 3));
foreach($o as $key => $val) {
    // some code
}
?>
```

Array and property traversal

- ☑ **ArrayObject** allows external traversal of arrays
- ☑ **ArrayObject** creates **ArrayIterator** instances
- ☑ Multiple **ArrayIterator** instances can reference the same target with different states
- ☑ Both implement **SeekableIterator** which allows to 'jump' to any position in the Array directly.

Array and property traversal



Functional programming?

- ☑ Abstract from the actual data (types)
- ☑ Implement algorithms without knowing the data

Example: Sorting

- ☞ Sorting requires a container for elements
- ☞ Sorting requires element comparison
- ☞ Containers provide access to elements

- ☞ Sorting and Containers must not know data

An example

- ☑ Reading a menu definition from an array
- ☑ Writing it to the output

Problem

- ☞ Handling of hierarchy
- ☞ Detecting recursion
- ☞ Formatting the output

Recursion with arrays

☑ A typical solution is to directly call array functions

☠ No code reuse possible

```php
<?php
function recurse_array($ar)
{
    // do something before recursion
    while (!is_null(key($ar))) {
        if (is_array(current($ar))) {
            recurse_array(current($ar));
        }
        // do something
        next($ar);
    }
    // do something after recursion
}
?>
```

Detecting Recursion

- ☑ An array is recursive
  - ☑ If the current element itself is an Array
  - ☑ In other words **current**() has children
  - ☑ This is detectable by **is_array**()
  - ☑ Recursing requires creating a new wrapper instance for the child array
  - ☑ **RecursiveIterator** is the interface to unify Recursion
  - ☑ **RecursiveIteratorIterator** handles the recursion

```
class RecursiveArrayIterator
    extends ArrayIterator implements RecursiveIterator
{
    function hasChildren() {
        return is_array($this->current());
    }
    function getChildren() {
        return new RecursiveArrayIterator($this->current());
    }
}
```

Debug Session

```php
<?php
$a = array('1','2',array('31','32'),'4');
$o = new RecursiveArrayIterator($a);
$i = new RecursiveIteratorIterator($o);
foreach($i as $key => $val) {
    echo "$key => $val\n";
}
?>
```

```
0 => 1
1 => 2
0 => 31
1 => 32
3 => 3
```

```php
<?php
class RecursiveArrayIterator implements RecursiveIterator {
   protected $ar;
   function __construct(Array $ar) {
      $this->ar = $ar; }
   function rewind() {
      reset($this->ar); }
   function valid() {
      return !is_null(key($this->ar)); }
   function key() {
      return key($this->ar); }
   function current() {
      return current($this->ar); }
   function next() {
      next($this->ar); }
   function hasChildren() {
      return is_array(current($this->ar)); }
   function getChildren() {
      return new RecursiveArrayIterator($this->current()); }
}
?>
```

Making ArrayObject recursive

- ☑ Change class type of `ArrayObjects` Iterator
  - ☞ We simply need to change `getIterator()`

```
<?php
class RecursiveArrayObject
    extends ArrayObject
{
    function getIterator() {
        return new RecursiveArrayIterator($this);
    }
}
?>
```

Output HTML

- ☑ Problem how to format the output using
  - ☞ Detecting recursion begin/end

```
<?php
class MenuOutput
   extends RecursiveIteratorIterator
{
   function __construct(Menu $m) {
      parent::__construct($m);
   }
   function beginChildren() {
      echo str_repeat(' ',$this->getDepth())."<ul>\n";
   }
   function endChildren() {
      echo str_repeat(' ',$this->getDepth())."</ul>\n";
   }
}
?>
```

Output HTML

- ☑ Problem how to write the output
  - ☞ Echo the output within **foreach**

```
<ul>
<li>1</li>
<li>2</li>
  <ul>
  <li>31</li>
  <li>32</li>
  </ul>
<li>4</li>
</ul>
```

```php
<?php
class MenuOutput
   extends RecursiveIteratorIterator
{
   function __construct(Menu $m) {
      parent::__construct($m);
   }
   function beginChildren() {
      echo str_repeat(' ',$this->getDepth())."<ul>\n";
   }
   function endChildren() {
      echo str_repeat(' ',$this->getDepth()+1)."</ul>\n";
   }
}
$ar = array('1','2',array('31','32'),'4');
$it = new MenuOutput(new RecursiveArrayIterator($ar));
echo "<ul>\n"; // for the intro
foreach($it as $m) {
   echo str_repeat(' ',$it->getDepth()+1)."<li>$m</li>\n";
}
echo "</ul>\n"; // for the outro
?>
```

Filtering

Problem

- ☞ Only recurse into active `Menu` elements
- ☞ Only show visible `Menu` elements
- ☠ Changes prevent `recurse_array` from reuse

```php
<?php
class Menu
{
    function isActive()  // return true if active
    function isVisible() // return true if visible
}
function recurse_array($ar)
{
    // do something before recursion
    while (!is_null(key($ar))) {
        if (is_array(current($ar)) && current($ar)->isActive()) {
            recurse_array(current($ar));
        }
        if (current($ar)->current()->isActive()) {
            // do something
        }
        next($ar);
    }
    // do something after recursion
}
?>
```

Filtering

Solution Filter the incoming data

- Unaccepted data simply needs to be skipped
- Do not accept inactive menu elements
- Using a `FilterIterator`

```php
<?php
class Menu extends RecursiveArrayIterator
{
    function isActive()  // return true if active
    function isVisible() // return true if visible
}
?>
```

FilterIterator

- ☑ `FilterIterator` is an abstract `OuterIterator`
  - ☑ Constructor takes an `Iterator` (called inner iterator)
  - ☑ Any iterator operation is executed on the inner iterator
  - ☑ For every element accept() is called after current/key
  - ➔ All you have to do is implementing accept()

Conférence PHP Québec

Debug Session

```php
<?php
$a = array(1,2,5,8);
$i = new EvenFilter(new MyIterator($a));
foreach($i as $key => $val) {
    echo "$key => $val\n";
}
?>
```

```
1 => 2
3 => 8
```

```php
<?php
class EvenFilter extends FilterIterator {
   function __construct(Iterator $it) {
      parent::__construct($it); }
   function accept() {
      return $this->current() % 2 == 0; }
}
class MyIterator implements Iterator {
   function __construct($ar) {
      $this->ar = $ar; }
   function rewind() {
      reset($this->ar); }
   function valid() {
      return !is_null(key($this->ar)); }
   function current() {
      return current($this->ar); }
   function key() {
      return key($this->ar); }
   function next() {
      next($this->ar); }
}
?>
```

Filtering

- ☑ Using a `FilterIterator`

```
<?php
class MenuFilter extends FilterIterator
    implements RecursiveIterator
{
    function __construct(Menu $m) {
       parent::__construct($m);
    }
    function accept() {
       return $this->current()->isVisible();
    }
    function hasChildren() {
       return $this->current()->hasChildren()
           && $this->current()->isActive();
    }
    function getChildren() {
       return new MenuFilter($this->current());
    }
}
?>
```

Putting it together

☑ Make `MenuOutput` operate on `MenuFilter`

- Pass a `Menu` to the constructor (guarded by type hint)
- Create a `MenuFilter` from the `Menu`
- `MenuFilter` implements `RecursiveIterator`

```
<?php
class MenuOutput extends RecursiveIteratorIterator
{
   function __construct(Menu $m) {
      parent::__construct(new MenuFilter($m));
   }
   function beginChildren() {
      echo "<ul>\n";
   }
   function endChildren() {
      echo "</ul>\n";
   }
}
?>
```

What now

- ☑ If your menu structure comes from a database
- ☑ If your menu structure comes from XML

  - ☞ You have to change `Menu`
  - ☞ Detection of recursion works differently

  - ☞ No single change in `MenuOutput` needed
  - ☞ No single change in `MenuFilter` needed

Using XML

- ☑ Change `Menu` to inherit from `SimpleXMLIterator`

```php
<?php
class Menu extends SimpleXMLIterator
{
    static function factory($xml)
    {
        return simplexml_load_string($xml, 'Menu');
    }
    function isActive() {
        return $this['active']; // access attribute
    }
    function isVisible() {
        return $this['visible']; // access attribute
    }
    // getChildren already returns Menu instances
}
?>
```

Using PDO

☑ Change `Menu` to read from database

- ☞ PDO supports Iterator based access
- ☞ PDO can create and read into objects
- ☞ PDO will be integrated into PHP 5.1
- ✋ PDO is under heavy development

```php
<?php
$db = new PDO("mysql://...");
$stmt= $db->prepare("SELECT ... FROM Menu ...", "Menu");
foreach($stmt->execute() as $m) {
      // fetch now returns Menu instances
      echo $m; // call $m->__toString()
}
?>
```

Filtering

- An `OuterIterator` may not pass data from its `InnerIterator` directly

Example:

Provide a 404 handler that looks for similar pages

- Use a RecursiveDirectoryIterator to test all files
- Use a FilterIterator to skip all files with low similarity

- Sort by similarity -> convert iterated data to array

Looking for files

- ☑ In PHP 4 you would use standard dir funcs

```
function search($path, $search, $limit, &$files) {
   if ($dir = @opendir($path)) {
      while (($found = readdir($dir) !== false) {
         switch(filetype("$path/$found")) {
         case 'file':
            if (($s=similariry($search, $found)) >= $limit) {
               $files["$path/$found"] = $s;
            }
            break;
         case 'dir':
            if ($found != '.' && $found != '..') {
               search("$path/$found", $search, $limit, $files);
            }
            break;
         }
      }
      closedir($dir);
   }
}
```

Looking for files

- ☑ PHP 5 offers RecursiveDirectoryIterator

```
class FindSimilar extends FilterIterator {
   protected $search, $limit, $key;
   function __construct($root, $search, $limit) {
      parent::__construct(
         new RecursiveIteratorIterator(
            new RecursiveDirectoryIterator($root)));
      $this->search = $search;
      $this->limit = min(max(0, $limit), 100);
   }
   function current() {
      return similarity($this->search, $this->current());
   }
   function key() {
      return $this->getSubPathname();
   }
   function accept() {
      return $this->isFile() && this->current()>=$this->limit;
   }
}
```

Error404.php

- ☑ Displaying alternatives in an error 404 handler

```
<html>
<head><title>File not found</title></head>
<body>
<?php
if (array_key_exists('missing', $_REQUEST)) {
   $missing = urldecode($_REQUEST['missing']);
   url_split($missing, $protocol, $host, $path, $ext, $query);
   $files = iterator_to_array($path, $missing, 35);
   asort($files);
   foreach($files as $file => $similarity) {
      echo "<a href='" . $file . "'>";
      echo $file . " [" . $similarity . "%]</a><br/>";
   }
} else {
   echo "No alternatives were found\n";
}
?>
</body>
</html>
```

Conclusion so far

- ☑ Iterators require a new way of programming
- ☑ Iterators allow to implement algorithms abstracted from data
- ☑ Iterators promote code reuse
- ☑ Some things are already in SPL
  - ☑ Filtering
  - ☑ Handling recursion
  - ☑ Limiting

Dynamic class loading

☑ `__autoload()` is good when you're alone

- ☑ Requires a single file for each class
- ☑ Only load class files when necessary
  - ☑ No need to parse/compile unneeded classes
  - ☑ No need to check which class files to load

☒ Additional user space code

☠ Only one single loader model is possible

__autoload & require_once

- Store the class loader in an include file
  - In each script:
    require_once('<path>/autoload.inc')
  - Use INI option:
    auto_prepend_file=<path>/autoload.inc

```php
<?php
function __autoload($class_name)
{
   require_once(dirname(__FILE__) . '/' .
   $class_name . '.p5c');
}
?>
```

SPL's class loading

- ☑ Supports fast default implementation
  - ☑ Look into path's specified by INI option include_path
  - ☑ Look for specified file extensions (.inc, .inc.php)
- ☑ Ability to register multiple user defined loaders
- ☑ Overwrites ZEND engine's __autoload() cache
  - ☑ You need to register __autoload if using spl's autoload

```
<?php
   spl_autoload_register('spl_autoload');
   spl_autoload_register('__autoload');
?>
```

SPL's class loading

- ☑ `spl_autoload($class_name)`
  Load a class though registered class loaders
  Fast c cod eimplementation
- ☑ `spl_autoload_extensions([$extensions])`
  Get or set files extensions
- ☑ `spl_autoload_register($loader_function)`
  Registers a single loader function
- ☑ `spl_autoload_unregister($loader_function)`
  Unregister a single loader function
- ☑ `spl_autoload_functions()`
  List all registered loader functions
- ☑ `spl_autoload_call($class_name)`
  Load a class though registered class loaders
  Use `spl_autoload()` as fallback

Exceptions

☑ Respect these rules

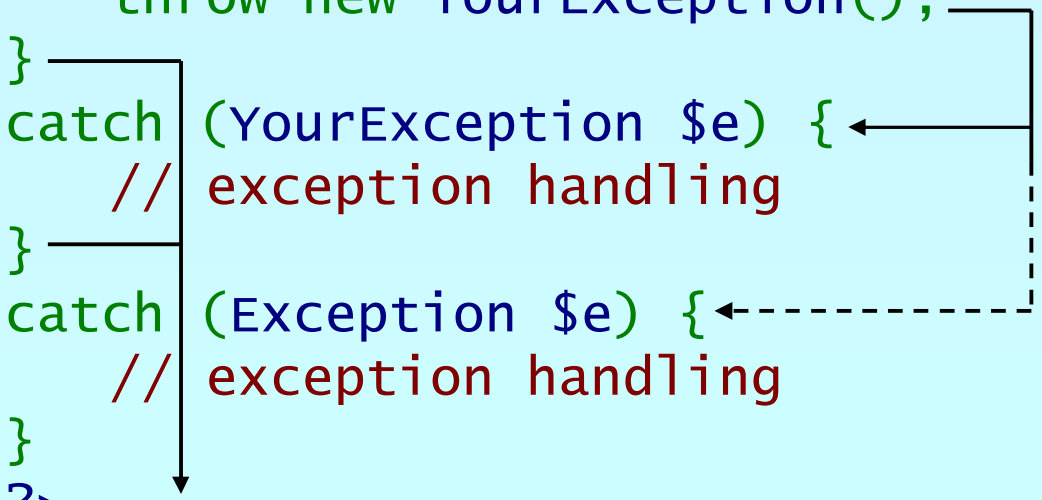
1. Exceptions are exceptions
2. Never use exceptions for control flow
3. Never ever use exceptions for parameter passing

```php
<?php
try {
    // your code
    throw new Exception();
}
catch (Exception $e) {
    // exception handling
}
?>
```

Exception specialization

- ☑ Exceptions should be specialized
- ☑ Exceptions should inherit built in class exception

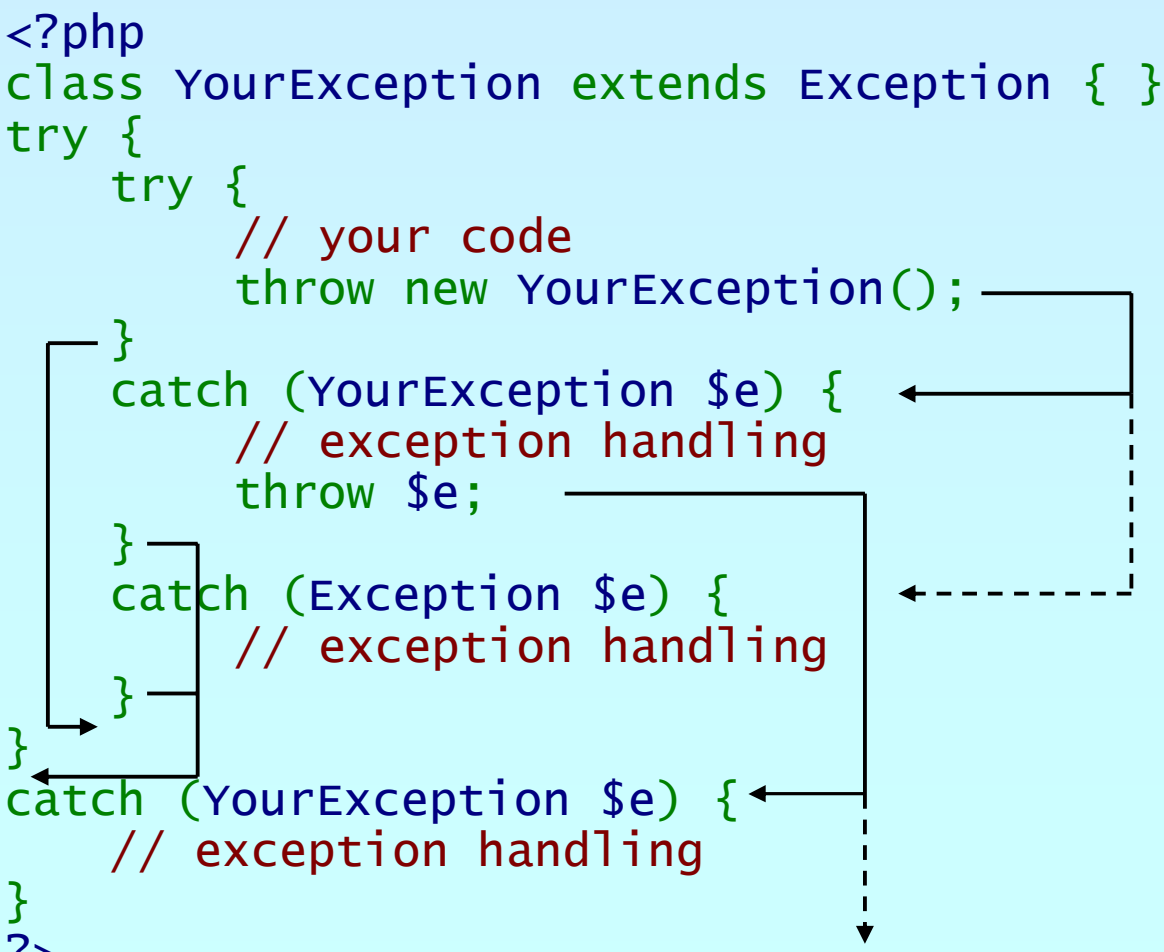
```php
<?php
class YourException extends Exception {
}
try {
    // your code
    throw new YourException();
}
catch (YourException $e) {
    // exception handling
}
catch (Exception $e) {
    // exception handling
}
?>
```



Exception specialization

- ☑ Exception blocks can be nested
- ☑ Exceptions can be re thrown

```php
<?php
class YourException extends Exception { }
try {
    try {
        // your code
        throw new YourException();
    }
    catch (YourException $e) {
        // exception handling
        throw $e;
    }
    catch (Exception $e) {
        // exception handling
    }
}
catch (YourException $e) {
    // exception handling
}
?>
```



Constructor failure

- ☑ Constructors do not return the created object
- ☑ Exceptions allow to handle failed constructors

```php
<?php
class Object {
    function __construct() {
        throw new Exception;
    }
}
try {
    $o = new Object;
}
catch (Exception $e) {
    echo "Object could not be instantiated\n";
}
?>
```

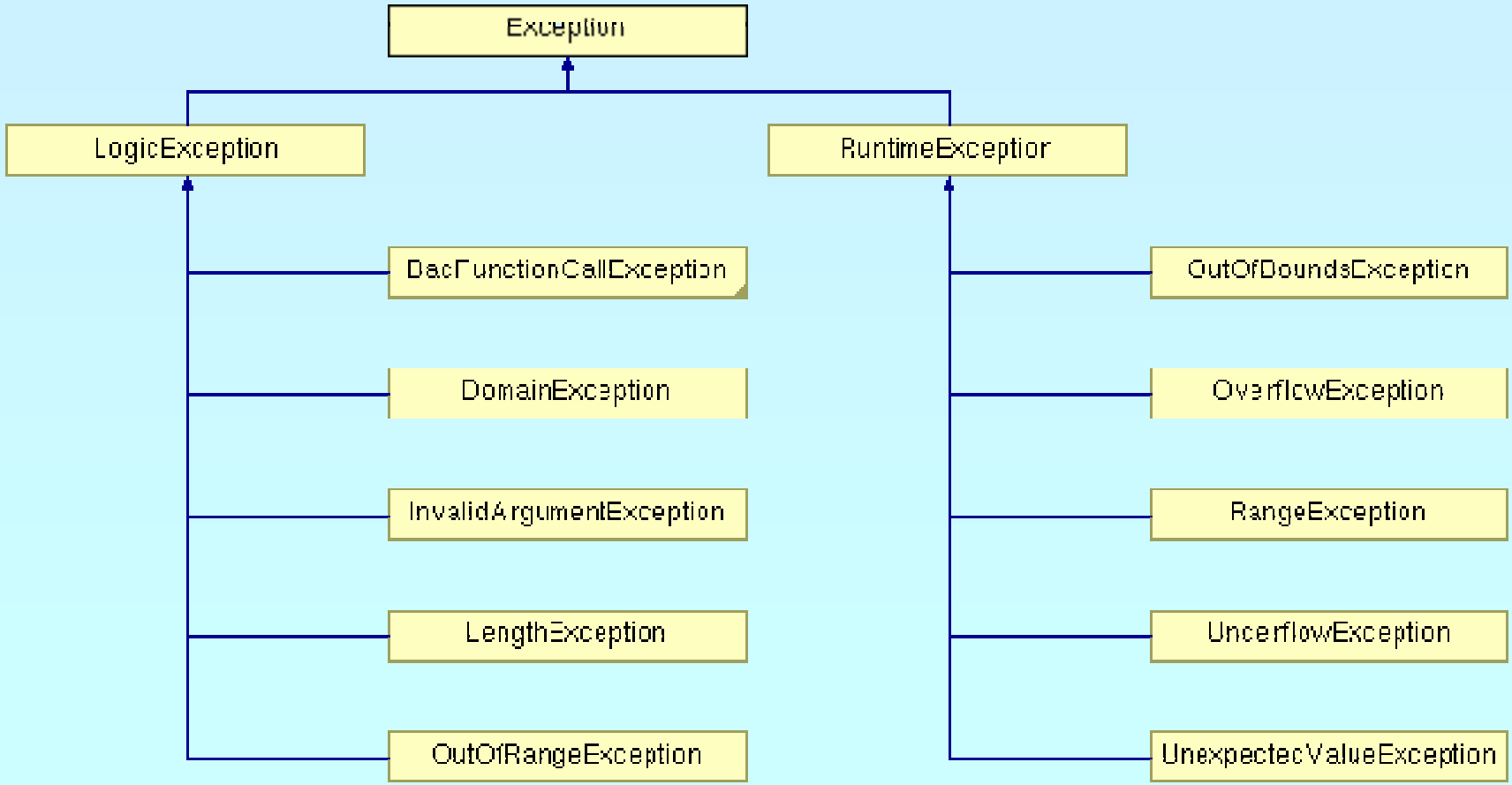
Convert Errors to Exceptions

- ☑ Implementing PHP 5.1 class ErrorException

```
<?php
class ErrorException extends Exception {
  protected $severity;
  function __construct($message, $code, $severity){
     parent::__construct($message, $code);
     $this->severity = $severity;
  }
  function getSeverity() {
     return $this->severity;
  }
}
function ErrorsToExceptions($severity, $message) {
  throw new ErrorException($message, 0, $severity);
}
set_error_handler('ErrorsToExceptions');
?>
```

SPL Exceptions

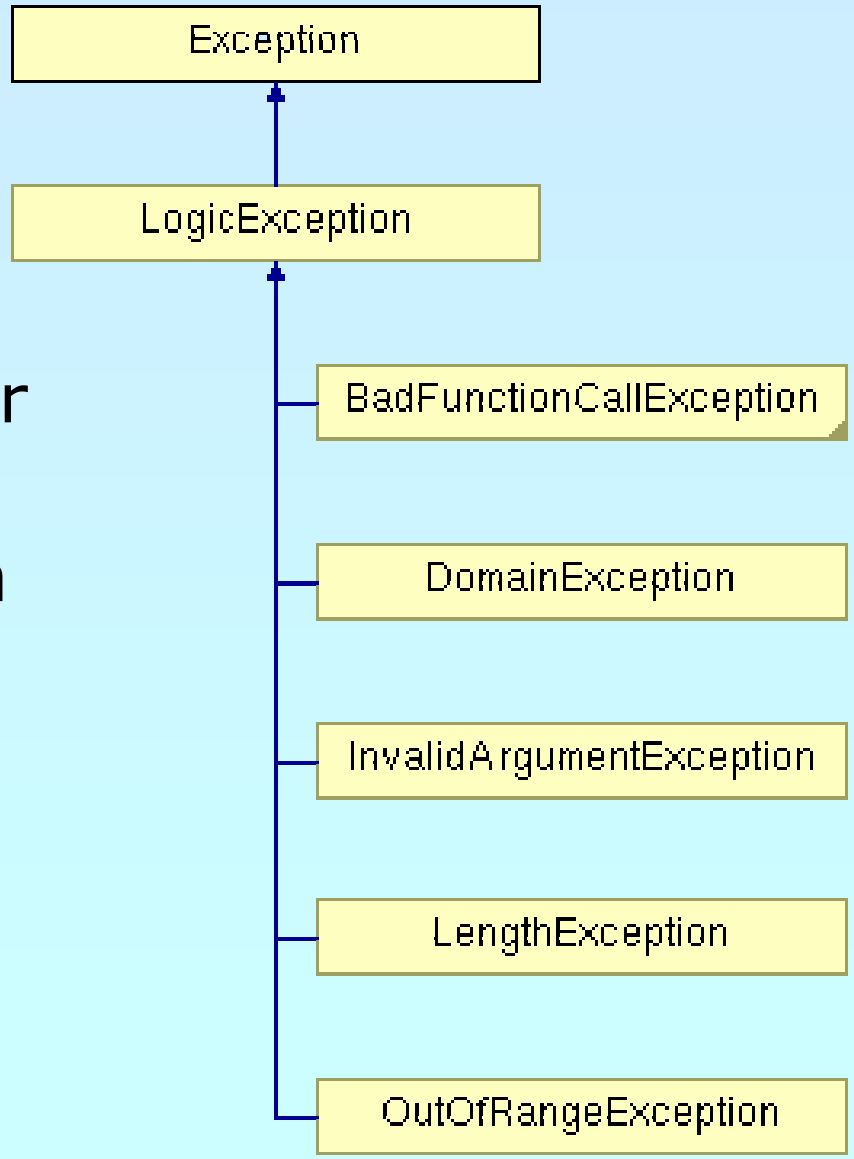
- ☑ SPL provides a standard set of exceptions
- ☑ Class Exception must be the root of all exceptions



General distinguishing

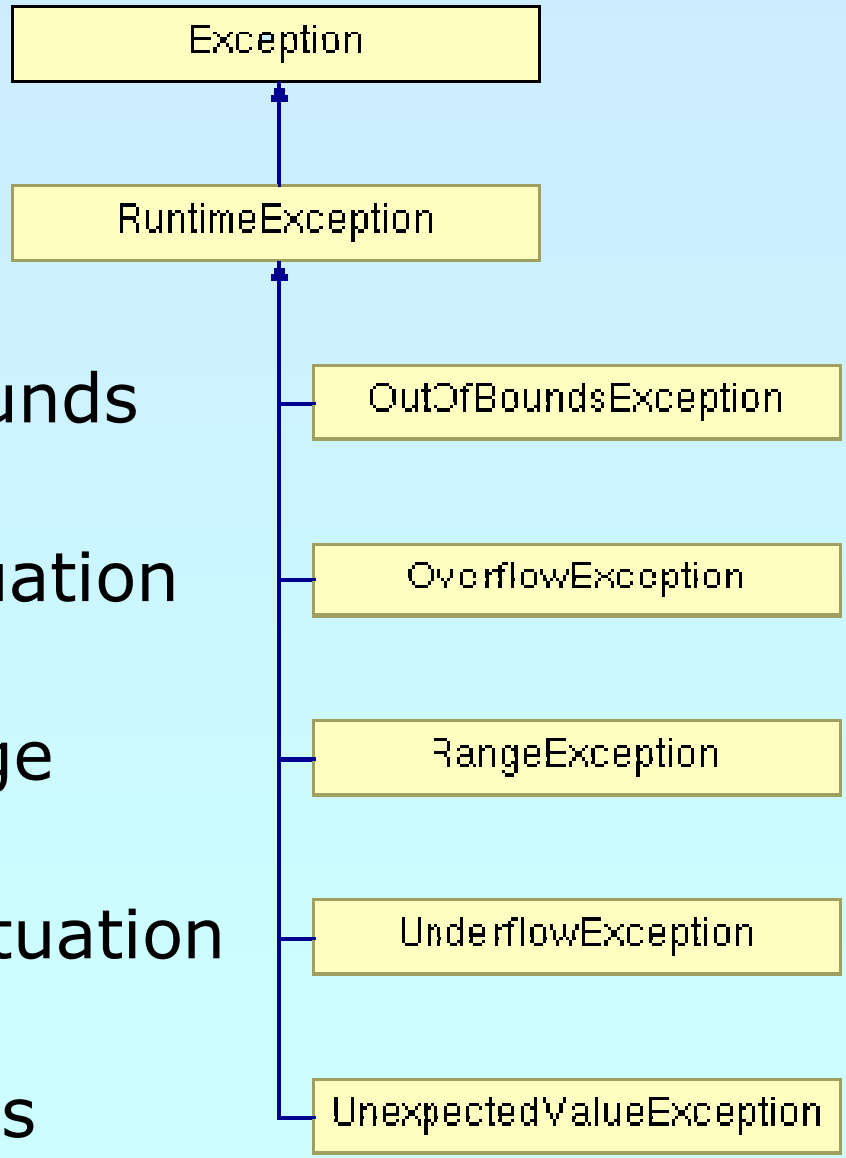
- ☑ `LogicException`
  - ➔ Anything that could have been detected at compile time or during application design
- ☑ `RuntimeException`
  - ➔ Anything that is unexpected during runtime
  - ➔ Base Exception for all database extensions

LogicException



- ☑ Function not found or similar
- ☑ Value not in allowed domain
- ☑ Argument not valid
- ☑ Length exceeded
- ☑ Some index is out of range

RunTimeException



- ☑ An actual value is out of bounds
- ☑ Buffer or other overflow situation
- ☑ Value outside expected range
- ☑ Buffer or other underflow situation
- ☑ Any other unexpected values

THANK YOU

http://somabo.de/talks/

http://php.net/~helly