From engine overloading to SPL

Marcus Börger

php|*Tropics*

From engine overloadig . . .
. . . to SPL

☑ Discuss overloadable engine features

☑ Learn about SPL aka Standard PHP Library

From engine overloading . . .

☑ Zend engine 2.0+ allows to overload the following

- ☑ by implementing interfaces
  - ☑ Foreach by implementing `Iterator`, `IteratorAggregate`
  - ☑ Array access by implementing `ArrayAccess`
  - ☑ Serializing by implementing `Serializable` (PHP 5.1)

- ☑ by providing magic functions
  - ☑ Function invocation by method `__call()`
  - ☑ Property access by methods `__get()` and `__set()`
  - ☑ Automatic loading of classes by function `__autoload()`

. . . to SPL

[illegible]

[illegible]

[illegible]

☑ A collection of standard interfaces and classes
  - Most of which based around engine overloading

☑ 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.

☑ Iterators allow to encapsulate algorithms

What are Iterators

- Iterators are a concept to iterate anything that contains other things. Examples:

Values and Keys in arrays	`ArrayObject`, `ArrayIterator`
Text lines in a file	`FileObject`
Files in a directory	`[Recursive]DirectoryIterator`
XML Elements or Attributes	ext: SimpleXML, DOM
Database query results	ext: PDO, SQLite, MySQLi
Dates in a calendar range	PECL/date
Bits in an image	?

- Iterators allow to encapsulate algorithms

What are Iterators

☑ Iterators are a concept to iterate anything that contains other things. Examples:

☑ Values and Keys in an array	`ArrayObject`, `ArrayIterator`
☑ Text lines in a file	`FileObject`
☑ Files in a directory	`DirectoryIterator`
☑ XML Elements or Attributes	ext: SimpleXML, DOM
☑ Database query results	ext: PDO, SQLite, MySQLi
☑ Dates in a calendar range	PECL/date
☑ Bits in an image	?

☑ Iterators allow to encapsulate algorithms

- ☑ Classes and Interfaces provided by SPL:

  `AppendIterator`, `CachingIterator`, `LimitIterator`, `FilterIterator`, `EmptyIterator`, `InfiniteIterator`, `NoRewindIterator`, `OuterIterator`, `ParentIterator`, `RecursiveIterator`, `RecursiveIteratorIterator`, `SeekableIterator`, . . .

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
- ☑ Iterators may iterate over other iterators

The big difference

☑ **Arrays**

- ☑ require memory for all elements
- ☑ allow to access any element directly

☑ **Iterators**

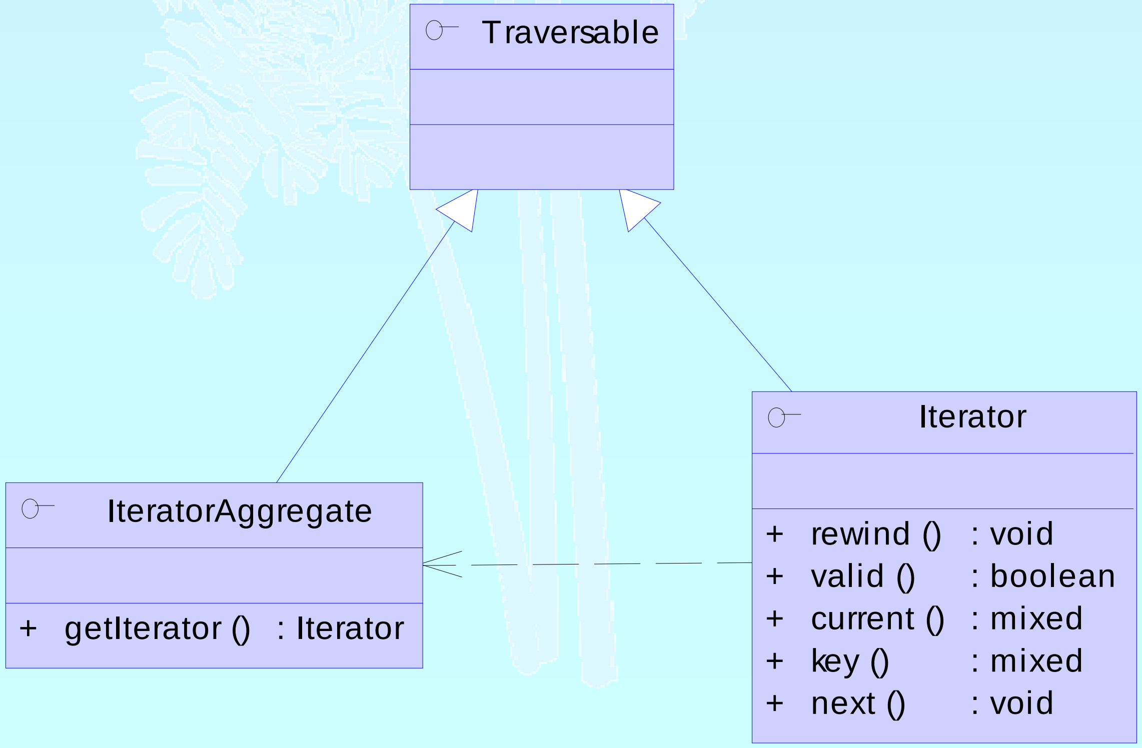
- ☑ only know one element at a time
- ☑ only require memory for the current element
- ☑ forward access only
- ☑ Access done by method calls

☑ **Containers**

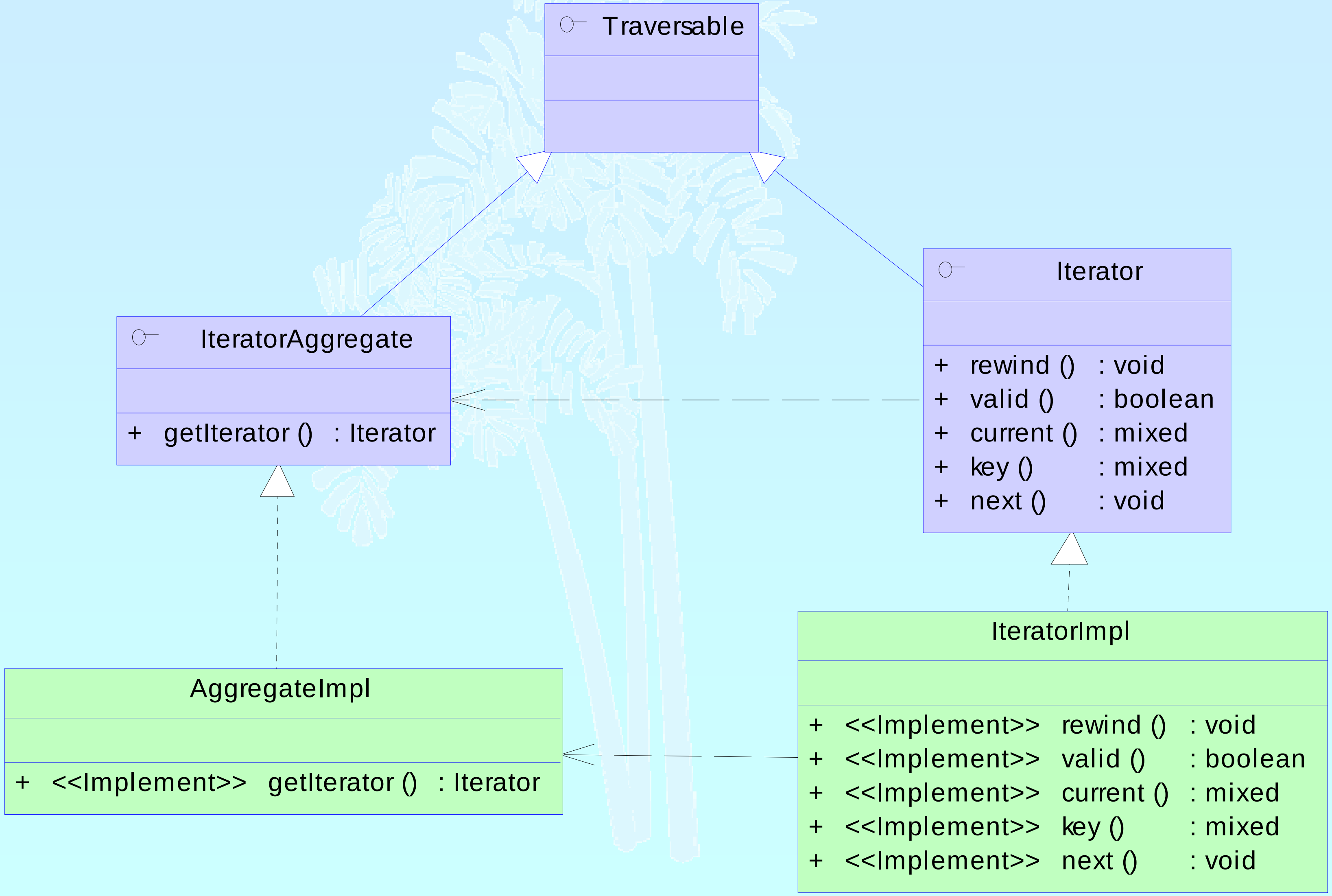
- ☑ require memory for all elements
- ☑ allow to access any element directly
- ☑ can create external Iterators or are internal Iterators

PHP Iterators

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



Implementing Iterators


Traversable
IteratorAggregate
+ getIterator () : Iterator
Iterator
+ rewind () : void
+ valid () : boolean
+ current () : mixed
+ key () : mixed
+ next () : void
AggregateImpl
+ <<Implement>> getIterator () : Iterator
IteratorImpl
+ <<Implement>> rewind () : void
+ <<Implement>> valid () : boolean
+ <<Implement>> current () : mixed
+ <<Implement>> key () : mixed
+ <<Implement>> next () : void

How Iterators work

☑ Iterators can be used manually

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

Overloading Array access

☑ PHP provides interface ArrayAccess

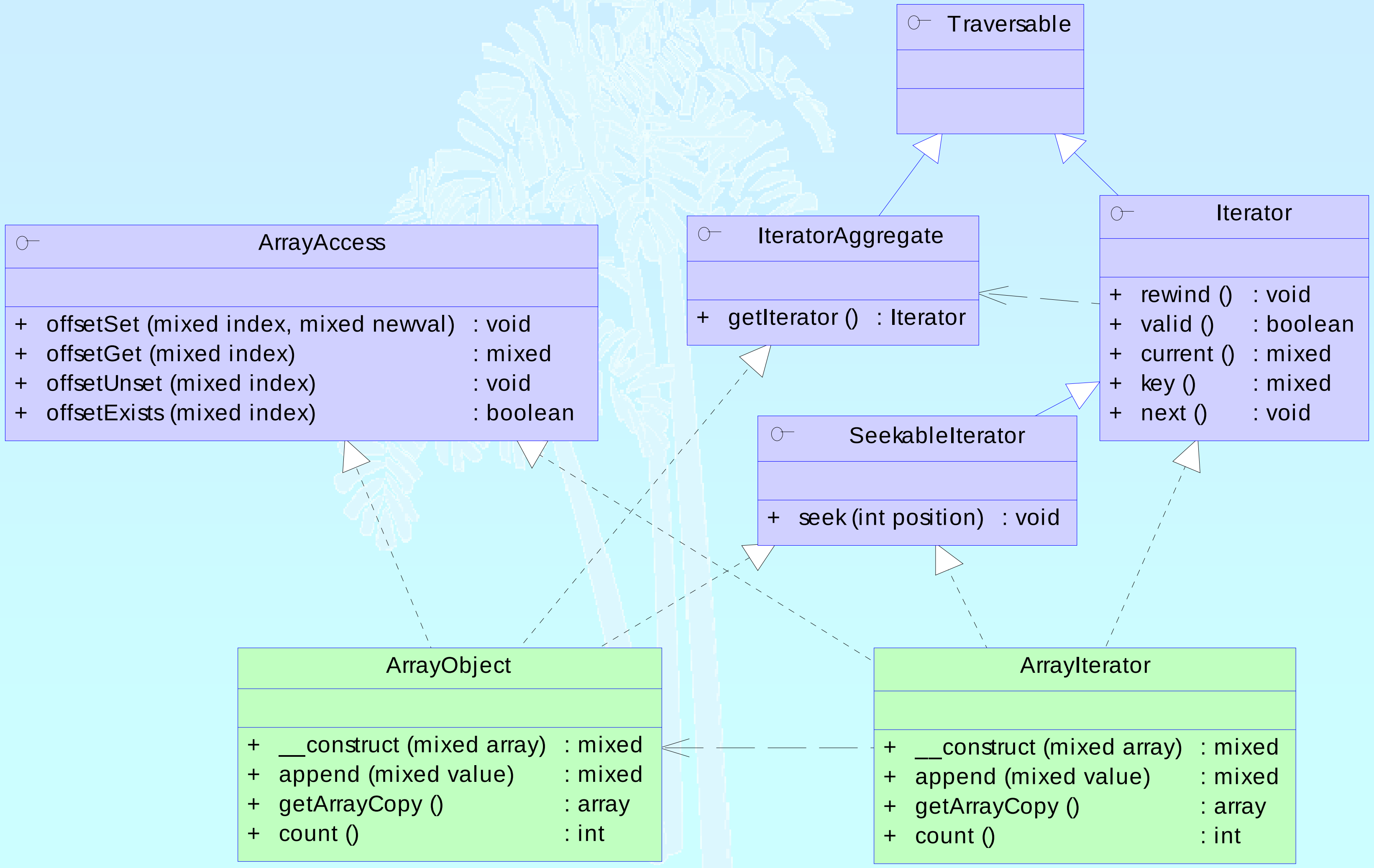
- ☑ Objects that implement it behave like normal arrays (only in terms of syntax though)
- ☑ Currently unstable API (not decided about references)

```
interface ArrayAccess {
   function &offsetGet($offset);
   function offsetSet($offset, &$value);
   function offsetExists($offset);
   function offsetUnset($offset);
}
```

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


Traversable
IteratorAggregate
+ getIterator () : Iterator
Iterator
+ rewind () : void
+ valid () : boolean
+ current () : mixed
+ key () : mixed
+ next () : void
ArrayAccess
+ offsetSet (mixed index, mixed newval) : void
+ offsetGet (mixed index) : mixed
+ offsetUnset (mixed index) : void
+ offsetExists (mixed index) : boolean
SeekableIterator
+ seek (int position) : void
ArrayObject
+ __construct (mixed array) : mixed
+ append (mixed value) : mixed
+ getArrayCopy () : array
+ count () : int
ArrayIterator
+ __construct (mixed array) : mixed
+ append (mixed value) : mixed
+ getArrayCopy () : array
+ count () : int

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
function recurse_array($ar)
{
   // do something before recursion
   reset($ar);
   while (!is_null(key($ar))) {
      // probably do something with the current element
      if (is_array(current($ar))) {
         recurse_array(current($ar));
      }
      // probably do something with the current element
      // probably only if not recursive
      next($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 => 4
```

```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 </ul>

  ☞ Detecting recursion begin/end

```
<?php
class MenuOutput
   extends RecursiveIteratorIterator
{
   function __construct(Menu $m) {
      parent::__construct($m);
   }
   function beginChildren() {
      // called after childs rewind() is called
      echo str_repeat('&nbsp;',$this->getDepth())."<ul>\n";
   }
   function endChildren() {
      // right before child gets destructed
      echo str_repeat('&nbsp;',$this->getDepth())."</ul>\n";
   }
}
?>
```

Output HTML

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

```
<?php
class MenuOutput
   extends RecursiveIteratorIterator
{
   function __construct(/* Array (or Menu) */ $m)
      parent::__construct($m);
   }
   function beginChildren() {
      echo str_repeat('&nbsp;',$this->getDepth())."<ul>\n";
   }
   function endChildren() {
      echo str_repeat('&nbsp;',$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 $m) {
   echo str_repeat('&nbsp;',$it->getDepth()+1)."<li>$m</li>\n";
}
echo "</ul>\n"; // for the outro
?>
```

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

Filtering

Problem

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

```
<?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
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()`

↺ Debug Session

```
<?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
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
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()
}
?>
```

Another example

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

- ☑ Provide a 404 handler that looks for similar pages
  - ☑ Use `RecursiveDirectoryIterator` to test all files
  - ☑ Use `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 directory 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, $limit), 100); // percentage
   }
   function current() {
      return similarity($this->search, $this->current());
   }
   function key() {
      return $this->getSubPathname(); // $root stripped out
   }
   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);
   $it = new FindSimilar($path);
   $files = iterator_to_array($it, $missing, 35);
   asort($files);
   foreach($files as $file => $similarity) {
      echo "<a href='" . $file . "'>";
      echo $file . " [" . $similarity . "%]</a><br/>";
   }
   if (!count($files)) {
      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

Old fashioned serializing

☑ Serializing by `__sleep()` and `__wakeup()`

- ☒ Not suitable for private member of base classes
- ☒ Shutdown code must be executed before serializing
- ☒ Wakeup code gets called after object construction
- ☒ Wakeup does not call any constructor

```
class Test {
   function __sleep() {
      return array(/* property names */);
   }
   function __wakeup() {
      /* some wakeup code */
   }
}
```

New PHP 5.1 serializing

☑ By implementing interface **Serializable**

```
Class Test implements Serializable {
   function __construct() {
   }
   function serialize() {
      /* some shutdown code */
      /* you may also call inherited serialize */
      return serialize(get_object_vars($this));
   }
   function unserialize($data) {
      $r = unserialize($data);
      $this->__construct(/* possibly refer to $r */);
      foreach($r as $prop => $val) {
           $this->$prop = $val;
      }
   }
}
```

Other magic

Overloading property access

- ☑ PHP allows to overload access to object properties
  - ☑ Enables lazy initialization
  - ☑ Enables virtual properties
  - ☑ Enables read or write only properties

  - ☑ Reading is achieved by simply providing:
    `mixed __get(mixed $name)`

  - ☑ Writing is achieved by simply providing:
    `void __set(mixed $name, mixed $value)`

  - ☒ Does not allow exists() checks
  - ☒ Does not allow dedicated visibility
  - ☒ Does not handle static properties or constsnts

Overloading property access

- ☑ Function `property_exists()`
  - ☑ Returns `true` even if the property value is NULL
  - ☑ Returns `false` for overloaded properties if value is NULL
  - ☑ Respects visibilty

- ☑ Method `ReflectionClass::hasProperty()`
  - ☑ Returns `true` even if the property value is NULL
  - ☑ Returns `false` for overloaded properties if value is NULL
  - ☑ Does not respect visibility

Overloading property access

☑ Typically virtual property handling is done with a protected array

```
class VirtualProperties
{
    protected $virtual = array();

    function __get($name) {
         return isset($this->virtual[$name])
           ? $this->virtual[$name] : NULL;
    }

    functoin __set($name, $value) {
         $this->virtual[$name] = $value;
    }
}
```

Lazy property initialization

☑ Property overloading and late read from database

```
class LazyProperties
{
   protected $id;
   protected $lazy = array();

   function __construct($id) {
      $this->id = $id;
   }
   function __get($name) {
      if (!array_key_exists($name, $this->lazy)) {
         $this->lazy[$name] = read_from_db($this, $name);
      }
      return $this->lazy[$name];
   }
   functoin __set($name, $value) {
      store_into_db($this, $name, $value);
      $this->lazy[$name] = $value;
   }
}
```

Overloading method invocation

- ☑ Allows aggregation
- ☑ Allows to simulate multiple inheritance
- ☑ Allows to simulate polymorphic methods

- ☒ Does not support inheritance

Overloading method invocation

☒ Does not support inheritance

Even if you expose all functions of an interface that are implemented in a property the object itself still does not implement that interface.

And if the object implements the interface itself you cannot use method overloading for the methods of that interface at the same time.

Overloading method invocation vs. inheritance

```
interface TestIf {
   function TestFunc();
}
class TestImpl implements TestIf {
   function TestFunc() {}
}
class Proxy implements TestIf {
   protected $v = array();

   function __construct() {
      $this->v['TestIf'] = new TestImpl;
   }

   function __call($name, $args) {
      return call_user_func_array(
         array($this->v['TestIf'], $name), $args);
   }
   function TestFunc() {
      return call_user_func_array(
         array($this->v['TestIf'], 'TestFunc'), $args);
   }
}
```

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');
    if (function_exists('__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 filename 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
  - Uses `spl_autoload()` as fallback

THANK YOU

☑ This Presentation

http://somabo.de/talks/

☑ SPL Documentation

http://php.net/~helly