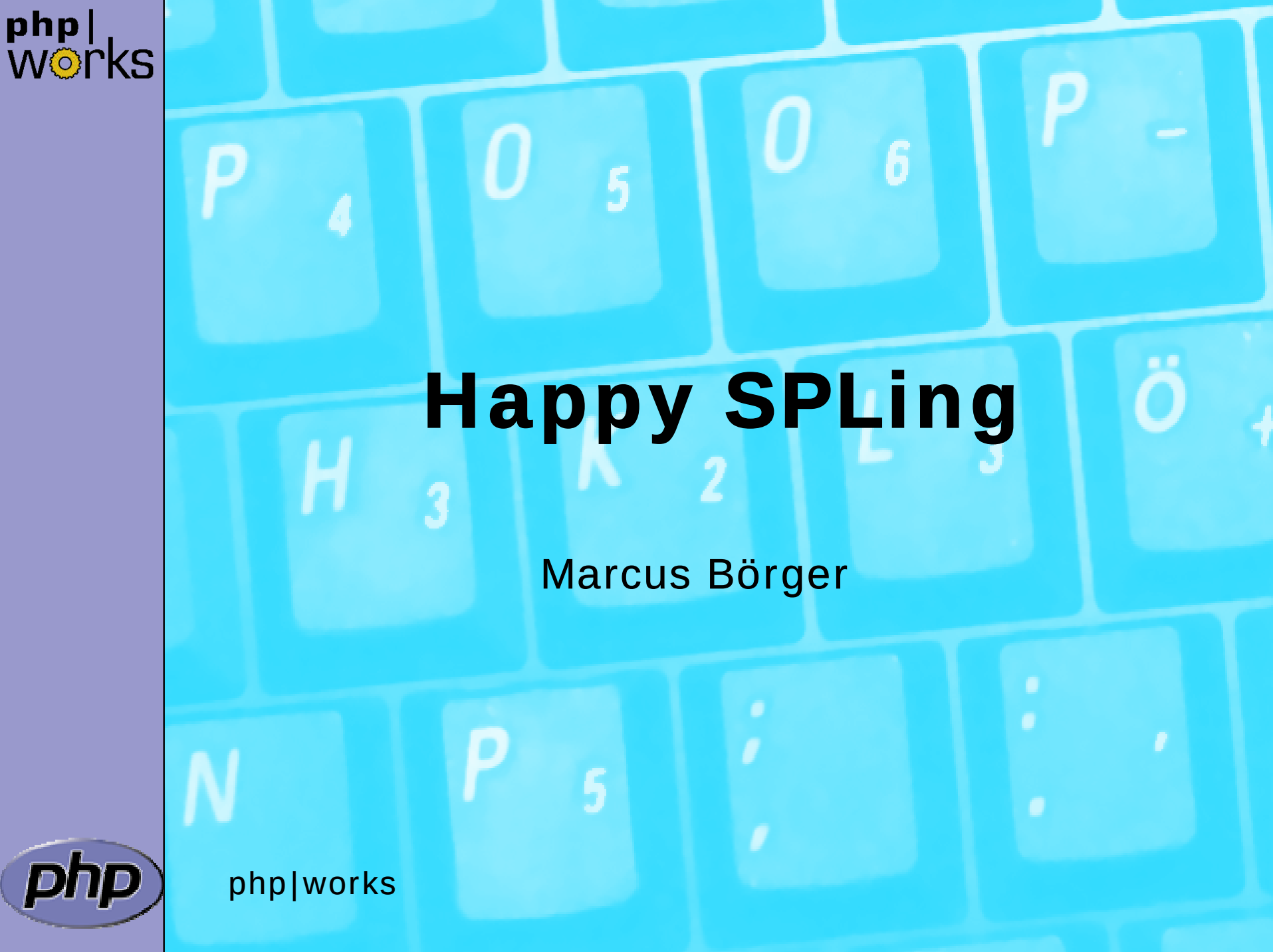


# Happy SPLing

Marcus Börger

# Happy SPLing

- ☑ 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

*It is easy in a complex way*

*- Lukas Smith*
*php conference 2004*

- ☑ 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

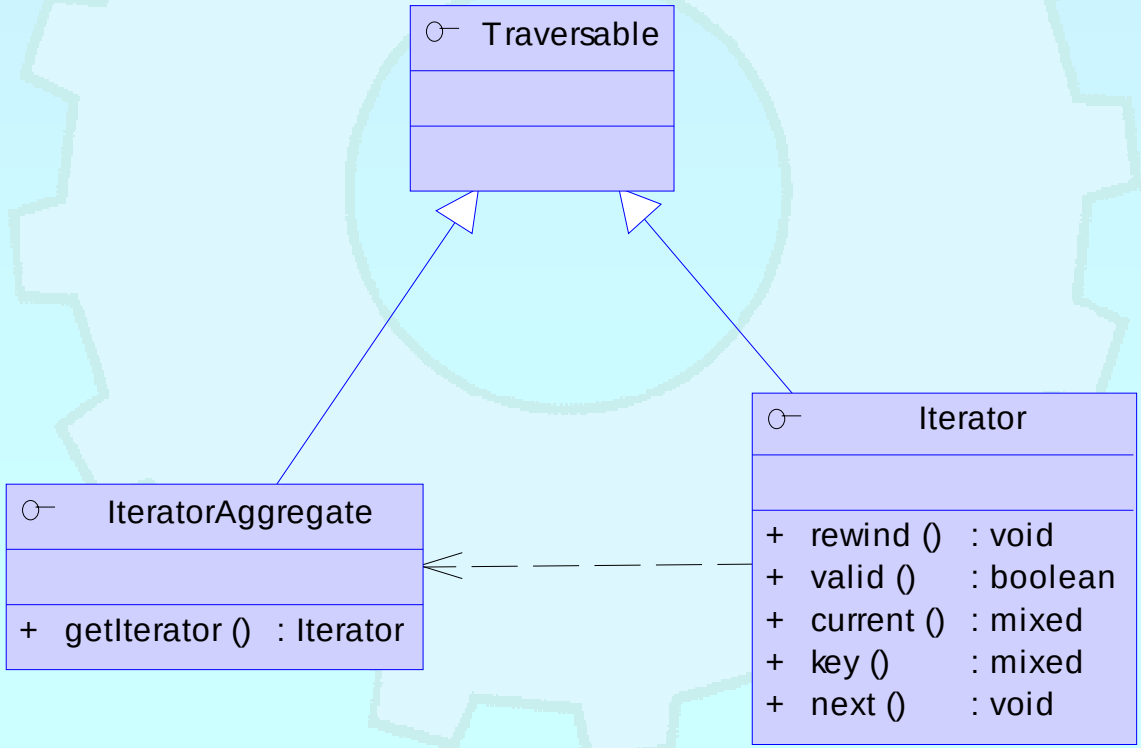
- ☑ only know one element at a time
- ☑ only require memory for the current element
- ☑ forward access only
- ☑ Access done by method calls

## ☑ Containers

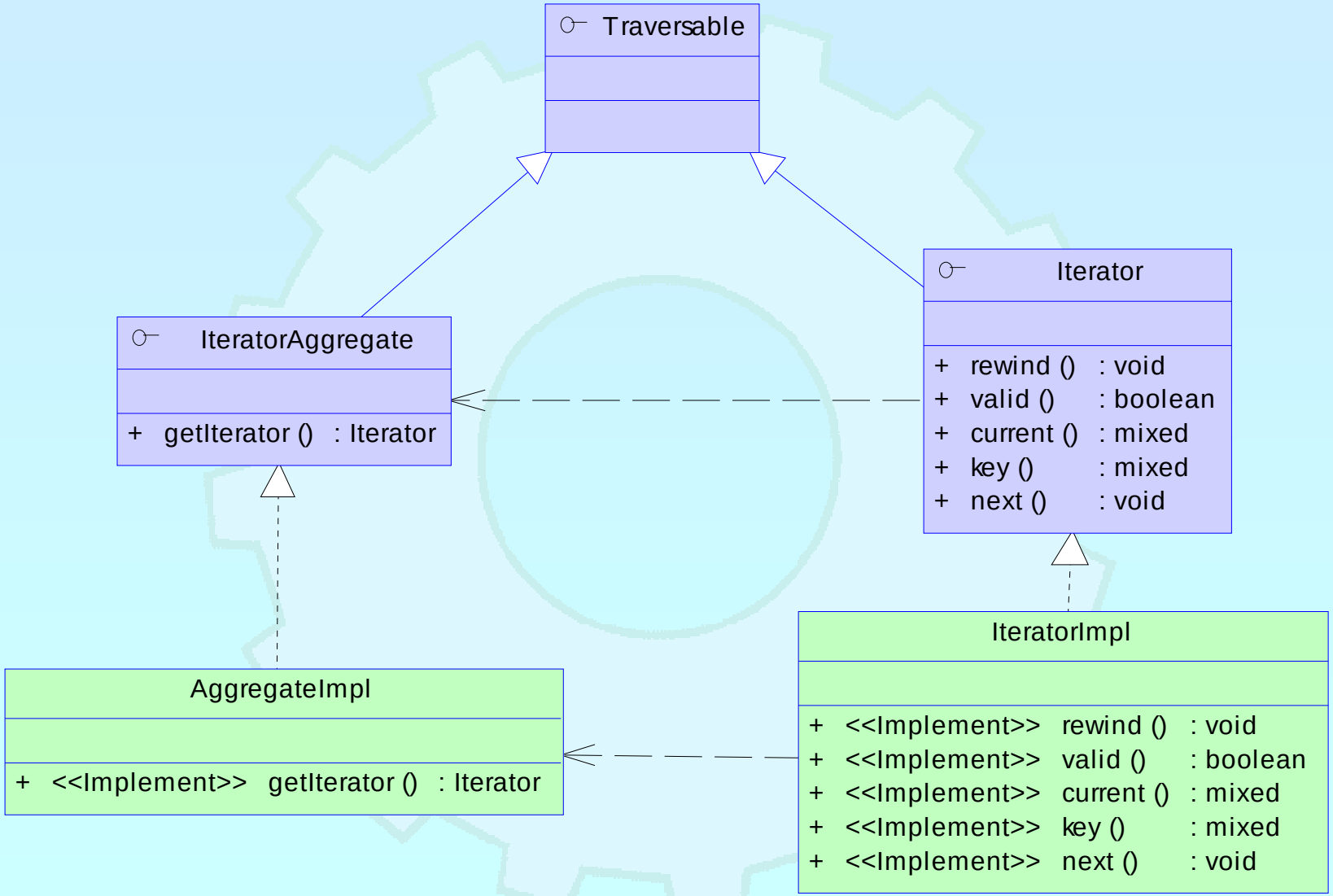
- ☑ require memory for all elements
- ☑ allow to access any element directly
- ☑ can create external Iterators or are internal Iterators

# 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



# 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
}
?>
```

# Overloading Array access

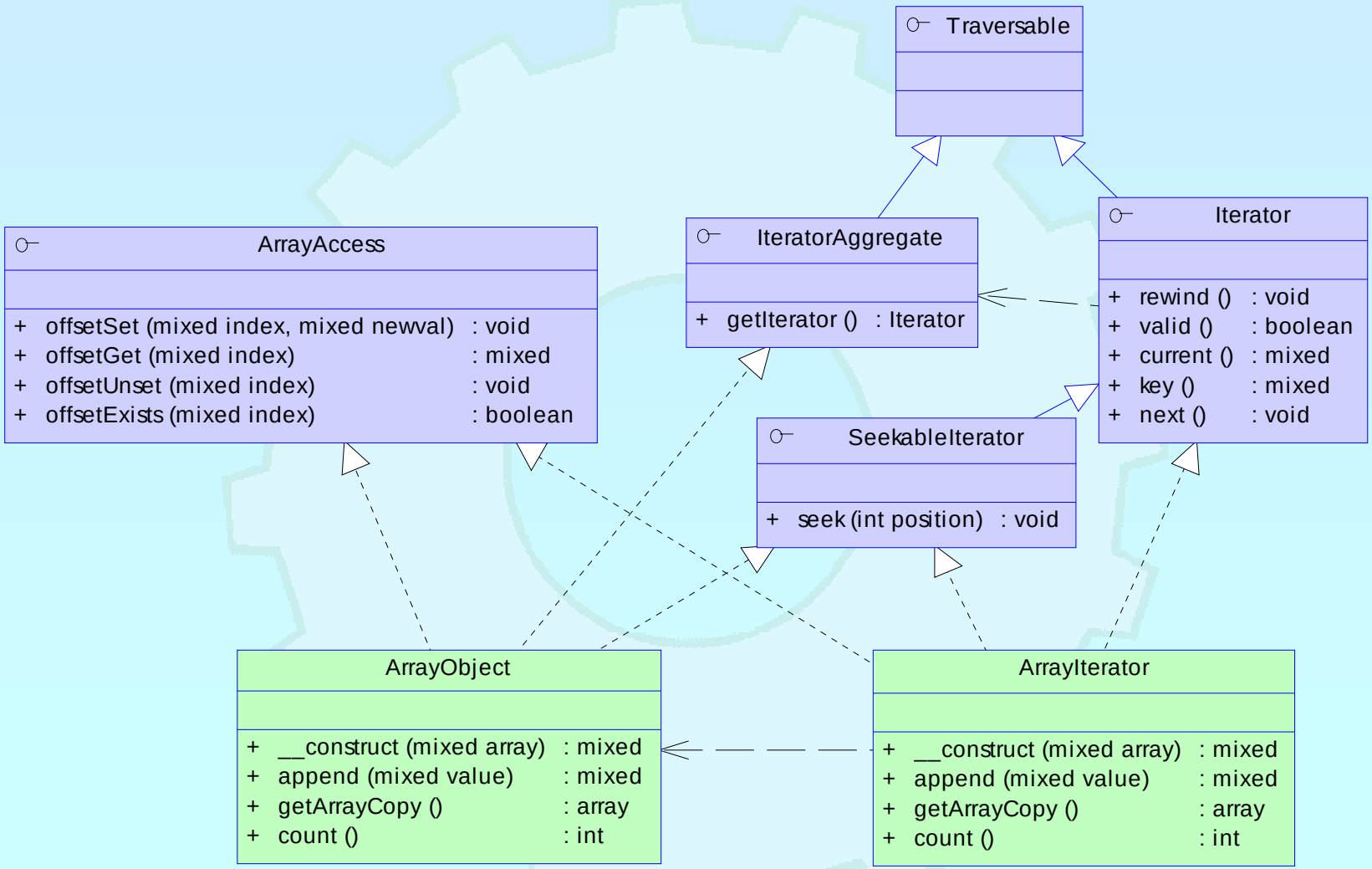
- PHP provides interface `ArrayAccess`
  - Objects that implement it behave like normal arrays (only in terms of syntax though)
  - ArrayAccess does not allow references (the following is an error)

```
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



# 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($ar)
{
    // do something before recursion
    reset($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);
    }
}
?>
```

# How does our Menu look?

- ☑ The basic interface is **MenuItem**
- ☑ A **MenuEntry** is the basic element of class **Menu**
- ☑ A **Menu** stores one or more **MenuItem** objects
- ☑ A **SubMenu** stores one or more **MenuItem** objects
- ☑ A **SubMenu** is a **Menu** and a **MenuItem**
- ☑ A **MenuIterator** shall iterate **Menu** and **SubMenu**

- ➔ **Menu** can store **MenuEntry** and **SubMenu**
- ➔ **SubMenu** can store in a **MenuEntry** or **SubMenu**
- ➔ **MenuItem** should know whether it has children
- ➔ **Menu** is a **IteratorAggregate** **MenuIterator**
- ➔ **MenuIterator** is a **RecursiveIterator**

# How does our Menu look?

- The general interface for menu entries
  - Only talking to entries through this interface ensures the code works no matter what we later add or change

```
interface MenuItem
{
   /** @return string representation of item (e.g. name/link)*/
   function __toString();

   /** @return whether item has children */
   function getChildren();

   /** @return children of the item if any available */
   function hasChildren();

   /** @return whether item is active or grayed */
   function isActive();

   /** @return whether item is visible or should be hidden */
   function isVisible();

   /** @return the name of the entry if any */
   function getName();
}
```

# How does our Menu look?

- ☑ We need a storage for the items
  - ☑ Either extend **RecursiveArrayIterator**
  - ☑ Or use an array an implement **IteratorAggregate**

```
class Menu implements IteratorAggregate
{
   public $_ar = array(); // PHP does not support friend

   function addItem(MenuItem $item) {
      $this->_ar[$item->getName()] = $item;
      return $item;
   }

   function getIterator() {
      return new MenuIterator($this);
   }
}
```

# How does our Menu look?

- ☑ Extend **`RecursiveArrayIterator`** but be typesafe
- ☑ Elements are non arrays

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

# How does our Menu look?

- ☑ Extend **`RecursiveArrayIterator`** but be typesafe
  - ☑ Ensure getChildren() returns the correct type
- ☑ Elements are non arrays

```
class RecursiveArrayIterator
   extends ArrayIterator implements RecursiveIterator
{
   function hasChildren() {
      return is_array($this->current());
   }
   function getChildren() {
      if (empty($ref)) $this->ref = new ReflectionClass($this);
      return $ref->newInstance($this->current());
   }
   protected $ref;
}
```

# How does our Menu look?

- ☑ Extend **`RecursiveArrayIterator`** but be typesafe
  - ☑ Ensure getChildren() returns the correct type
- ☑ Elements are non arrays
  - ☑ Recursion works slightly different
  - ☑ Override hasChildren() to not use is_array()
  - ☑ Keep existing getChildren() and other iterator methods

```
class MenuIterator extends RecursiveArrayIterator
{
   function __construct(Menu $menu) {
      parent::__construct($menu->_ar);
   }
   function hasChildren() {
      return $this->current()->hasChildren();
   }
}
```

# How does our Menu look?

```
class MenuEntry implements MenuItem
{
   protected $name, $link, $active, $visible;

   function __construct($name, $link = NULL) {
      $this->name = $name;
      $this->link = is_numeric($link) ? NULL : $link;
      $this->active = true;
      $this->visible = true;
   }
   function __toString() {
      if (strlen($this->link)) {
         return '<a href="'.$this->link.'">'.$this->name.'</a>';
      } else {
         return $this->name;
      }
   }
   function hasChildren() { return false; }
   function getChildren() { return NULL; }
   function isActive()    { return $this->active; }
   function isVisible()   { return $this->visible; }
   function getName()     { return $this->name; }
}
```

# How does our Menu look?

```
class SubMenu extends Menu implements MenuItem
{
   protected $name, $link, $active, $visible;

   function __construct($name = NULL, $link = NULL) {
      $this->name = $name;
      $this->link = is_numeric($link) ? NULL : $link;
      $this->active = true;
      $this->visible = true;
   }
   function __toString() {
      if (strlen($this->link)) {
         return '<a href="'.$this->link.'">'.$this->name.'</a>';
      } else if (strlen($this->name)) {
         return $this->name;
      } else return '';
   }
   function hasChildren() { return true; }
   function getChildren() { return new MenuIterator($this); }
   function isActive()    { return $this->active; }
   function isVisible()   { return $this->visible; }
   function getName()     { return $this->name; }
}
```

# How to create a menu

- To create a Menu we manually call **`addItem()`**
  - We need to keep track of the level in local temp vars

```php
<?php

$menu = new Menu();

$menu->addItem(new MenuEntry('Home'));

$sub = new SubMenu('Downloads');

$sub->addItem(new MenuEntry(''));

$menu->addItem($sub);

?>
```

# Reading a menu from an array

- ☑ We'd need to **foreach** the array and do recursion
- ☑ **RecursiveIteratorIterator** helps with events

```
class RecursiveIteratorIterator
{
   /** @return $this->getInnerIterator()->hasChildren()*/
   function callHasChildren()

   /** @return $this->getInnerIterator()->getChildren()*/
   function callGetChildren()

   /** Called if recursing into children */
   function beginChildren()

   /** called after last children */
   function endChildren()

   /** called if a new element is available */
   function nextElement()

   // ...
}
```

# Reading a menu from array

```
class MenuLoadArray extends RecursiveIteratorIterator {
   protected $sub = array();
   function __construct(Menu $menu, Array $def) {
      $this->sub[0] = $menu;
      parent::__construct(
         new RecursiveArrayIterator($def, self::LEAVES_ONLY));
   }
   function callGetChildren() {
      $child = parent::callGetChildren();
      $this->sub[] = end($this->sub)->addItem(new SubMenu());
      return $child;
   }
   function endChildren() {
      array_pop($this->sub);
   }
   function nextElement() {
      end($this->sub)->addItem(
         new MenuEntry($this->current(), $this->key()));
   }
}

$def = array('1', '2', array('31', '32'), '4');
$menu = new Menu();
foreach(new MenuLoadArray($menu, $def) as $v);
```

Provide some storage for the menu and ist sub menus and their sub menus.

# Reading a menu from array

```
class MenuLoadArray extends RecursiveIteratorIterator {
   protected $sub = array();
   function __construct(Menu $menu, Array $def) {
      $this->sub[0] = $menu;
      parent::__construct(
         new RecursiveArrayIterator($def, self::LEAVES_ONLY));
   }
   function callGetChildren() {
      $child = parent::callGetChildren();
      $this->sub[] = end($this->sub)->addItem(new SubMenu());
      return $child;
   }
   function endChildren() {
      array_pop($this->sub);
   }
   function nextElement() {
      end($this->sub)->addItem(
         new MenuEntry($this->current(), $this->key()));
   }
}

$def = array('1', '2', array('31', '32'), '4');
$menu = new Menu();
foreach(new MenuLoadArray($menu, $def) as $v);
```

MenuLoadArray controls the recursice iteration...

...a recursive structure.

# Reading a menu from array

```
class MenuLoadArray extends RecursiveIteratorIterator {
   protected $sub = array();
   function __construct(Menu $menu, Array $def) {
      $this->sub[0] = $menu;
      parent::__construct(
         new RecursiveArrayIterator($def, self::LEAVES_ONLY));
   }
   function callGetChildren() {
      $child = parent::callGetChildren();
      $this->sub[] = end($this->sub)->addItem(new SubMenu());
      return $child;
   }
   function endChildren() {
      array_pop($this->sub);
   }
   function nextElement() {
      end($this->sub)->addItem(
         new MenuEntry($this->current(), $this->key()));
   }
}

$def = array('1', '2', array('31', '32'), '4');
$menu = new Menu();
foreach(new MenuLoadArray($menu, $def) as $v);
```

When recursing we create a new unnamed SubMenu and make it the new top level element of our 'level' storage.

# Reading a menu from array

```
class MenuLoadArray extends RecursiveIteratorIterator {
   protected $sub = array();
   function __construct(Menu $menu, Array $def) {
      $this->sub[0] = $menu;
      parent::__construct(
         new RecursiveArrayIterator($def, self::LEAVES_ONLY));
   }
   function callGetChildren() {
      $child = parent::callGetChildren();
      $this->sub[] = end($this->sub)->addItem(new SubMenu());
      return $child;
   }
   function endChildren() {
      array_pop($this->sub);
   }
   function nextElement() {
      end($this->sub)->addItem(
         new MenuEntry($this->current(), $this->key()));
   }
}

$def = array('1', '2', array('31', '32'), '4');
$menu = new Menu();
foreach(new MenuLoadArray($menu, $def) as $v);
```

At the end of a sub array in our case representing a sub menu when pop that sub menu thus going to it's parent menu.

# Reading a menu from array

```
class MenuLoadArray extends RecursiveIteratorIterator {
   protected $sub = array();
   function __construct(Menu $menu, Array $def) {
      $this->sub[0] = $menu;
      parent::__construct(
         new RecursiveArrayIterator($def, self::LEAVES_ONLY));
   }
   function callGetChildren() {
      $child = parent::callGetChildren();
      $this->sub[] = end($this->sub)->addItem(new SubMenu());
      return $child;
   }
   function endChildren() {
      array_pop($this->sub);
   }
   function nextElement() {
      end($this->sub)->addItem(
         new MenuEntry($this->current(), $this->key()));
   }
}

$def = array('1', '2', array('31', '32'), '4');
$menu = new Menu();
foreach(new MenuLoadArray($menu, $def) as $v);
```

All elements in our definition that are not sub arrays are meant to end up as entries so we only want leaves as elements.

# Reading a menu from array

```
class MenuLoadArray extends RecursiveIteratorIterator {
   protected $sub = array();
   function __construct(Menu $menu, Array $def) {
      $this->sub[0] = $menu;
      parent::__construct(
         new RecursiveArrayIterator($def, self::LEAVES_ONLY));
   }
   function callGetChildren() {
      $child = parent::callGetChildren();
      $this->sub[] = end($this->sub)->addItem(new SubMenu());
      return $child;
   }
   function endChildren() {
      array_pop($this->sub);
   }
   function nextElement() {
      end($this->sub)->addItem(
         new MenuEntry($this->current(), $this->key()));
   }
}

$def = array('1', '2', array('31', '32'), '4');
$menu = new Menu();
foreach(new MenuLoadArray($menu, $def) as $v);
```

Now lets use the thing to fill in the menu from the definition in the array.

# Output HTML

- Problem how to format the output using </ul>
  - Detecting recursion begin/end

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

# Output HTML

- ☑ Problem how to write the output
  - ☞ Echo the output within **foreach**
- ☑ The following works for our Array def

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

```php
class MenuOutput
   extends RecursiveIteratorIterator
{
   function __construct(RecursiveIterator $ar) {
      parent::__construct($ar);
   }
   function beginChildren() {
      echo str_repeat(' ',$this->getDepth())."<ul>\n";
   }
   function endChildren() {
      echo str_repeat(' ',$this->getDepth())."</ul>\n";
   }
}
$def = array('1', '2', array('31', '32'), '4');
$menu = new RecursiveArrayIterator($def);

$it = new MenuOutput($menu);
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
```

# Output HTML

- ☑ Problem how to write the output
  - ☞ Echo the output within **foreach**
- ☑ The following works for our **Menu**

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

```
class MenuOutput
   extends RecursiveIteratorIterator
{
   function __construct(Menu $ar) {
      parent::__construct($ar);
   }
   function beginChildren() {
      echo str_repeat(' ',$this->getDepth())."<ul>\n";
   }
   function endChildren() {
      echo str_repeat(' ',$this->getDepth())."</ul>\n";
   }
}
$def = array('1', '2', array('31', '32'), '4');
$menu = new Menu();
foreach(new MenuLoadArray($menu, $def) as $v);
$it = new MenuOutput($menu);
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
```

# Wow - but why?

- ☑ Why did we used SPL here?
  - ☑ More reliability
    - ☑ Fix one time – no problem in finnding all incarnations
  - ☑ Easier to change soemthing without touching other stuff
    - ☑ Functional separation
    - ☑ Code ruse
    - ☑ Responsability control

# Filtering

Problem

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

```php
<?php
class MenuItem
{
   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 to filter the incoming data

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

```
interface MenuItem
{
    // ...

    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
    Inside the call `current()`/`key()` are valid
  - ➔ All you have to do is implement `accept()`
- ☑ `RecursiveFilterIterator` is also available

# 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 RecursiveFilterIterator
{
    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()->getChildren());
    }
}
?>
```

# 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**
- ☞ We could also use a special **MenuFilter/Menu** proxy
- ☞ We could also have **Menu** as an interface of **MenuFilter**

```
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` or provide an alternative to `MenuLoadArray`
  - ☞ 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**
  - ☑ Which is already a **RecursiveIterator**
  - ☑ We need to make it create **Menu** instances for children

```
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

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

# 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

# Other magic

# 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