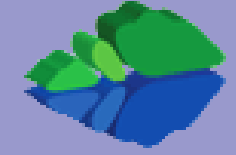
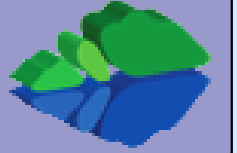


SPL
Standard PHP Library

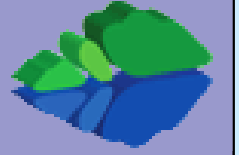
Marcus Börger

Vancouver PHP Conference 2007



SPL - Standard PHP Library

- ☑ Discuss overloadable engine features
- ☑ Learn about SPL aka Standard PHP Library

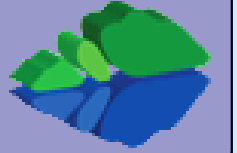


From engine overloading . . .

- ☑ 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`

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

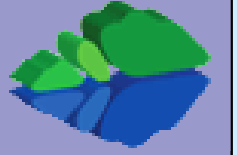


. . . to SPL

It is easy in a complex way

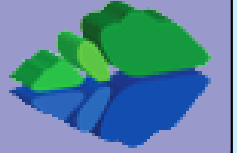
- Lukas Smith
php conference 2004

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



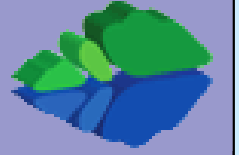
What is SPL about & what for

- ☑ Captures some common patterns
- ☑ Advanced Iterators
- ☑ Functional programming
- ☑ File and directory handling
- ☑ Makes __autoload() useable
- ☑ Exception hierarchy with documented semantics



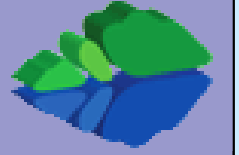
What are Iterators

- ☑ Iterators are a concept to iterate anything that contains other things.
- ☑ Iterators allow to encapsulate algorithms



What are Iterators

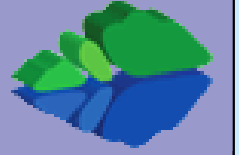
- Iterators are a concept to iterate anything that contains other things. Examples:
  - Values and Keys in an array — `ArrayObject`, `ArrayIterator`
  - Text lines in a file — `SplFileObject`
  - Files in a directory — [`Recursive`]`DirectoryIterator`
  - XML Elements or Attributes — ext: SimpleXML, 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 — `SplFileObject`
  - 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
  - Classes and Interfaces provided by SPL:

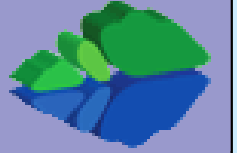
    `AppendIterator`, `CachingIterator`, `LimitIterator`, `FilterIterator`, `EmptyIterator`, `InfiniteIterator`, `NoRewindIterator`, `OuterIterator`, `ParentIterator`, `RecursiveIterator`, `RecursiveIteratorIterator`, `SeekableIterator`, `SplFileObject`, . . .



Array vs. Iterator

- ☑ An array in PHP — `$ar = array()`
  - ☑ can be rewound: `reset($ar)`
  - ☑ is valid unless it's key is NULL: `!is_null(key($ar))`
  - ☑ have current values: `current($ar)`
  - ☑ have keys: `key($ar)`
  - ☑ can be forwarded: `next($ar)`

- ☑ Something that is traversable — `$it = new Iterator;`
  - ☑ **may** know how to be rewound: `$it->rewind()`
    (does not return the element)
  - ☑ should know if there is a value: `$it->valid()`
  - ☑ **may** have a current value: `$it->current()`
  - ☑ **may** hava a key: `$it->key()`
    (may return NULL at any time)
  - ☑ can forward to its next element: `$it->next()`



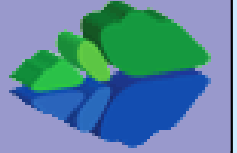
How Iterators work

- ☑ 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
}
?>
```



The big difference

☑ Arrays

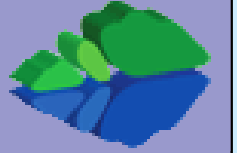
- ☑ require memory for all elements
- ☑ allow to access any element directly

☑ Iterators

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

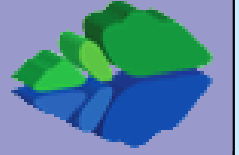
☑ Containers

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



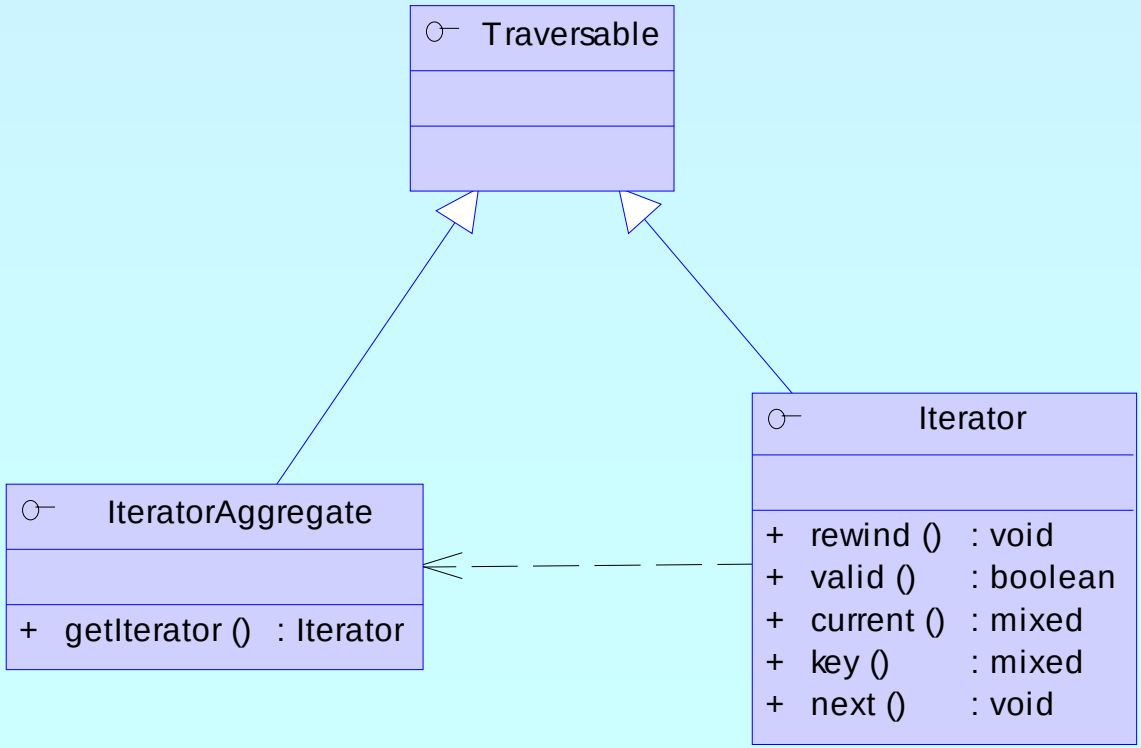
The basic concepts

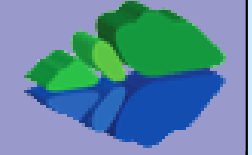
- ☑ Iterators can be internal or external
  also referred to as active or passive
- ☑ An internal iterator modifies the object itself
- ☑ An external iterator points to another object without modifying it
- ☑ PHP always uses external iterators at engine-level
- ☑ Iterators may iterate over other iterators



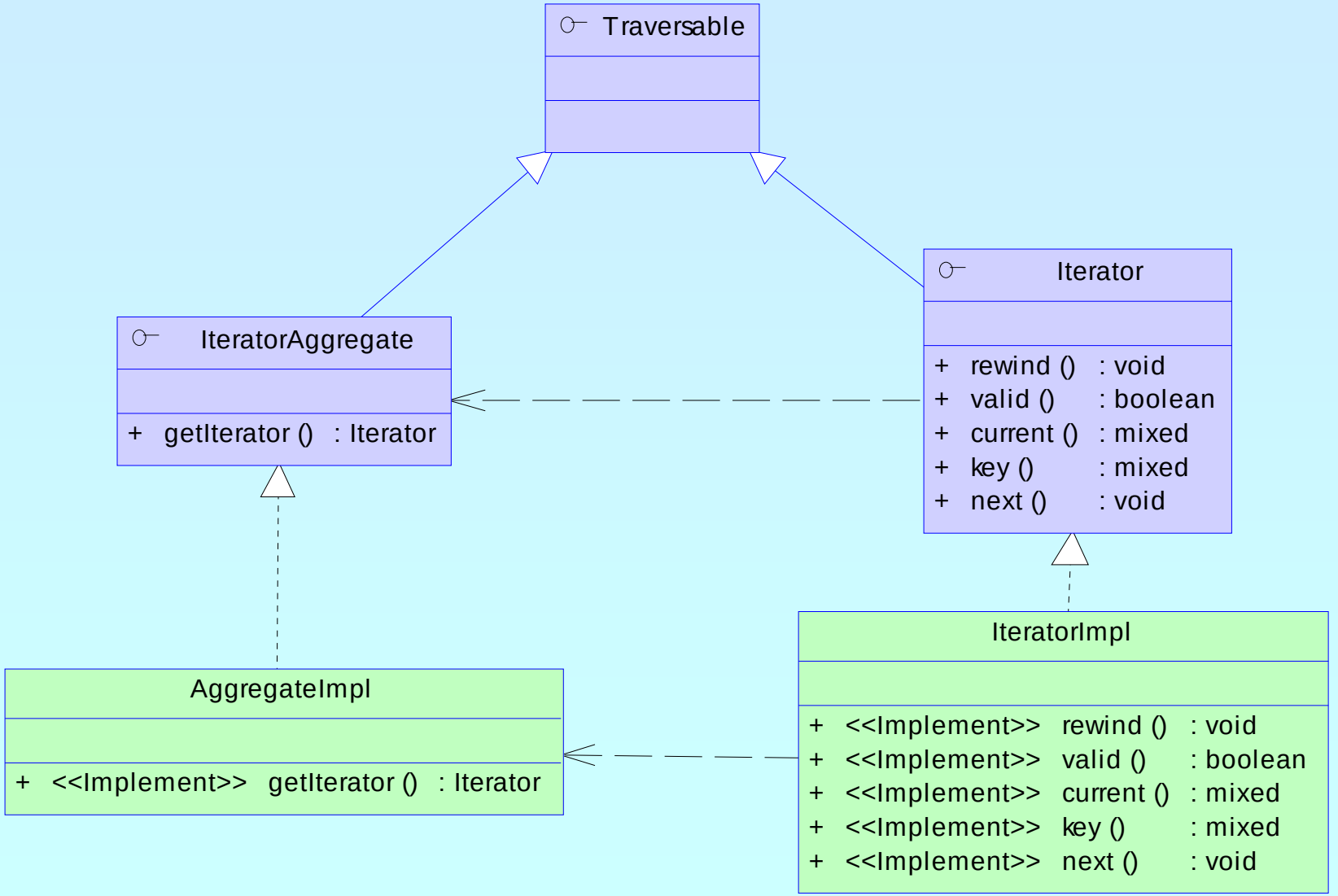
PHP Iterators

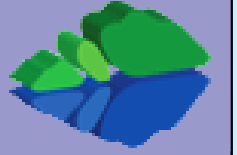
- ☑ Anything that can be iterated implements **`Traversable`**
- ☑ Objects implementing **`Traversable`** can be used in **`foreach`**
- ☑ User classes cannot implement **`Traversable`**
- ☑ **`IteratorAggregate`** is for objects that use external iterators
- ☑ **`Iterator`** is for internal traversal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





Implementing Iterators

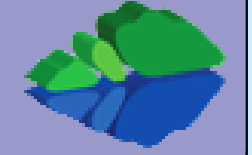




Overloading Array access

- 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)

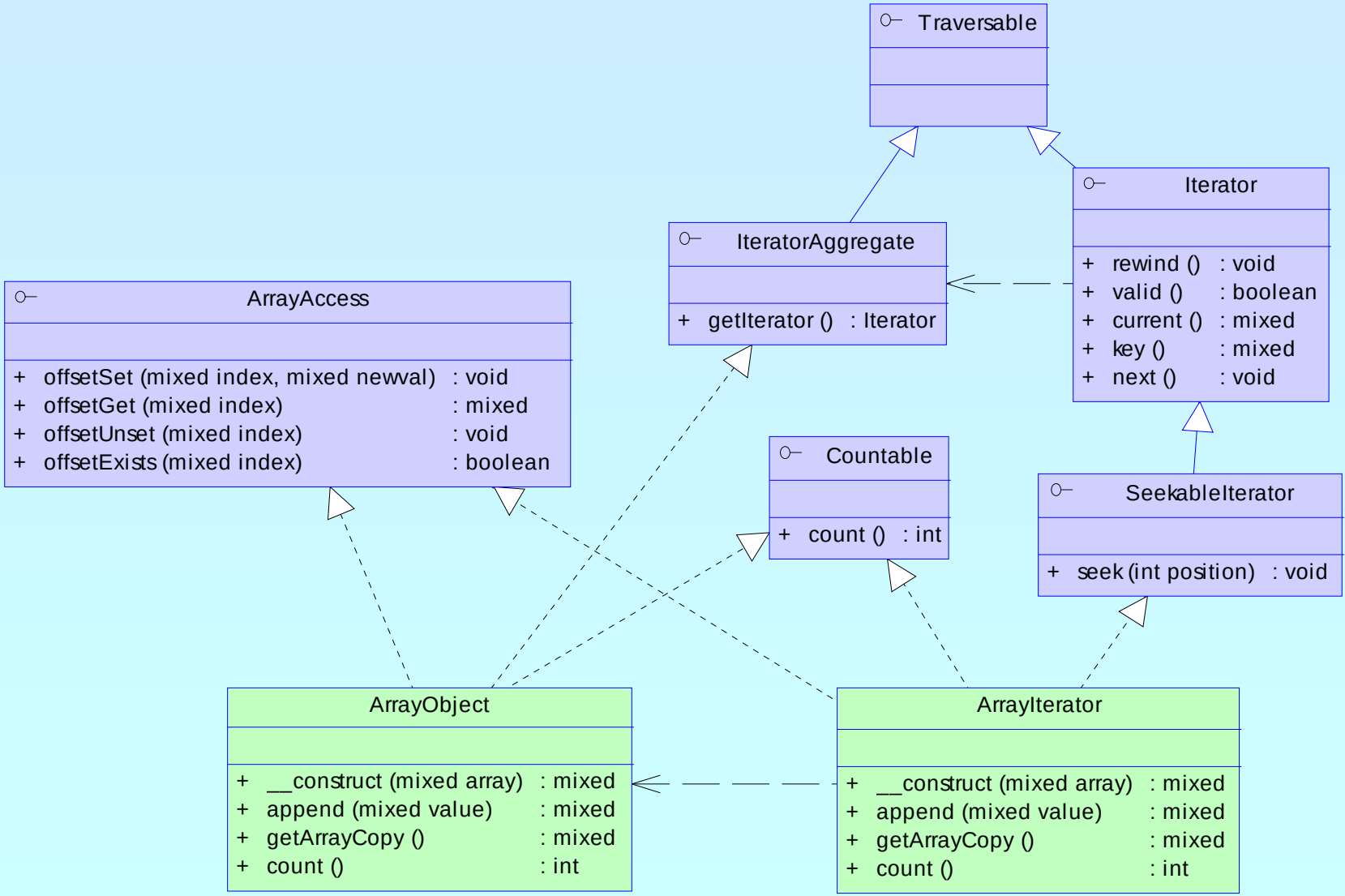
```
interface ArrayAccess {
   function &offsetGet($offset);
   function offsetSet($offset, &$value);
   function offsetExists($offset);
   function offsetUnset($offset);
}
```

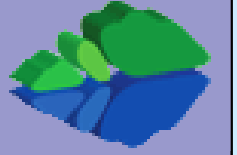


Array and property traversal

- ☑ **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





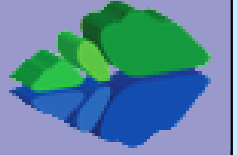
Functional programming?

- ☑ 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

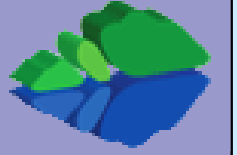


An example

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

Problem

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

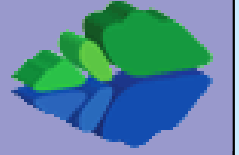


Recursion with arrays

☑ A typical solution is to directly call array functions

☠ No code reuse possible

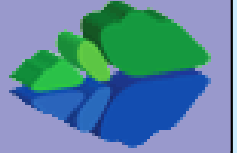
```php
<?php
function recurse_array($ar)
{
    // do something before recursion
    reset($ar);
    while (!is_null(key($ar))) {
       // probably do something with the current element
       if (is_array(current($ar))) {
          recurse_array(current($ar));
       }
       // probably do something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



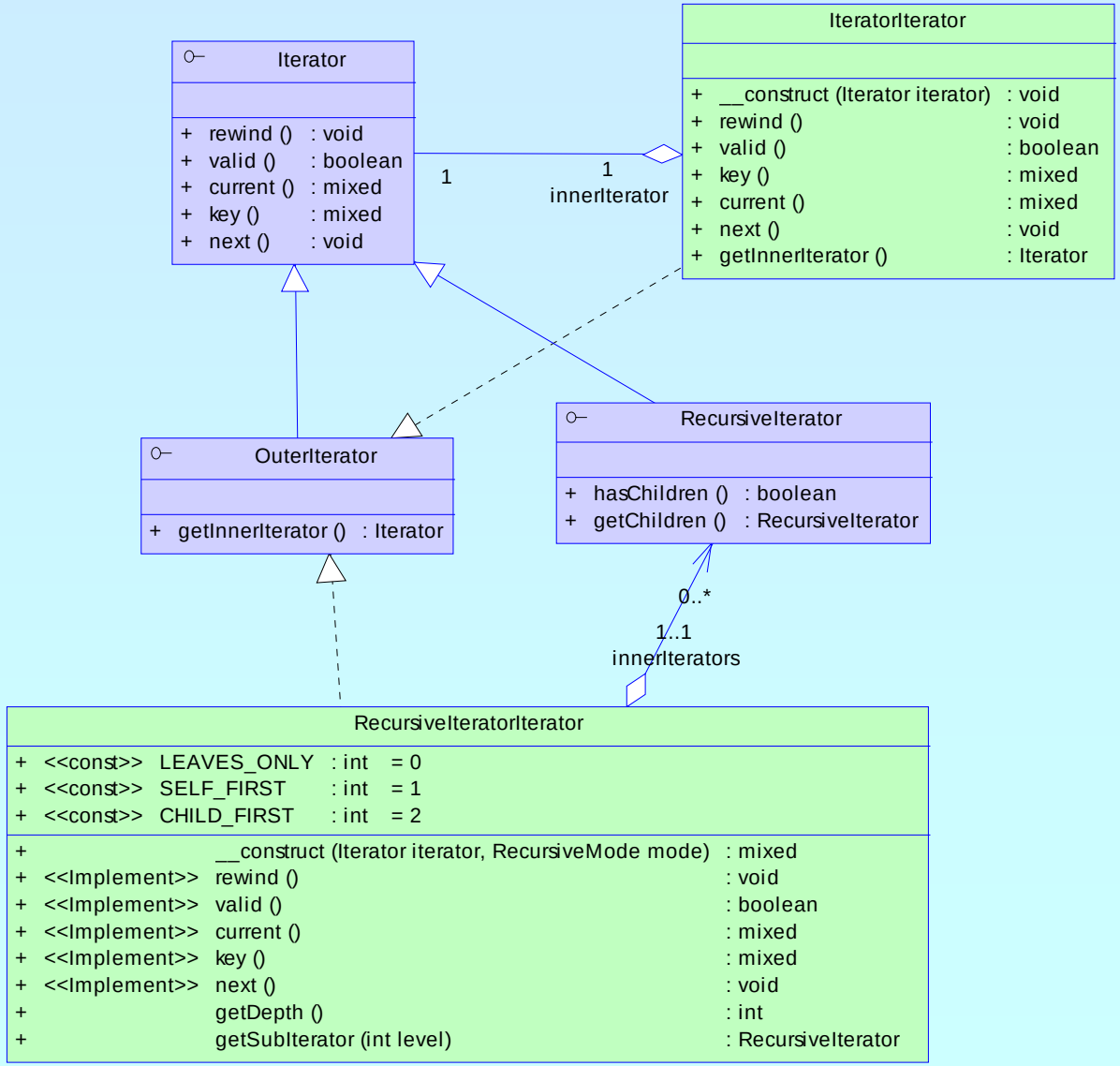
Debug Session

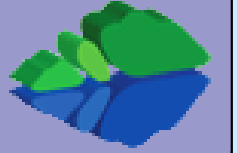
```php
<?php
$a = array('1','2',array('31','32'),'4');
$o = new RecursiveArrayIterator($a);
$i = new RecursiveIteratorIterator($o);
foreach($i as $key => $val) {
    echo "$key => $val\n";
}
?>
```

```
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()); }
}
?>
```

RecursiveIteratorIterator

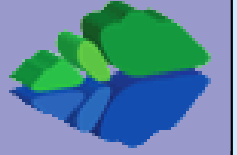




Making ArrayObject recursive

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

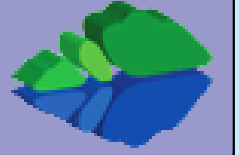
```
<?php
class RecursiveArrayObject extends ArrayObject
{
    function getIterator() {
       return new RecursiveArrayIterator($this);
    }
}
?>
```



Deriving RecursiveArrayObject

- How to generally enable the class to be derived
  - We simply need to change **getIterator**()
  - Return an instance of the instantiated class

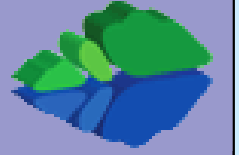
```
<?php
class RecursiveArrayObject extends ArrayObject
{
    function getIterator() {
       if (empty($this->ref)) {
          $this->ref = new ReflectionClass($this);
       }
       return $this->current() instanceof self
          ? $this->current()
          : $this->ref->newInstance($this->current());
    }
    protected $ref;
}
?>
```



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`**

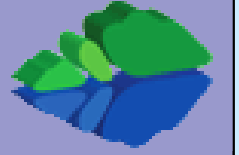
- ➔ **`Menu`** can store **`MenuEntry`** and **`SubMenu`**
- ➔ **`SubMenu`** can store in a **`MenuEntry`** or **`SubMenu`**
- ➔ **`MenuItem`** should know whether it has children
- ➔ **`Menu`** is a **`IteratorAggregate MenuIterator`**
- ➔ **`MenuIterator`** is a **`RecursiveIterator`**



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

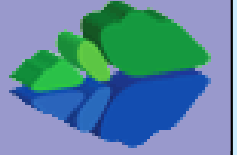
```
interface MenuItem
{
    /** @return string representation of item (e.g. name/link)*/
    function __toString();

    /** @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 and implement **IteratorAggregate**

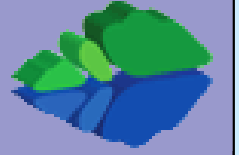
```
class Menu implements IteratorAggregate
{
    public $_ar = array(); // PHP does not support friend

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

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



How does our Menu look?

- ☑ Extend **RecursiveArrayIterator**
- ☑ Elements are non arrays

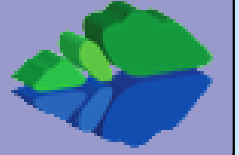
```
class RecursiveArrayIterator
   extends ArrayIterator implements RecursiveIterator
{
   function hasChildren() {
      return is_array($this->current());
   }
   function getChildren() {
      return new RecursiveArrayIterator($this->current());
   }
}
```



How does our Menu look?

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

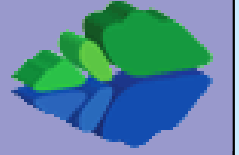
```
class RecursiveArrayIterator
   extends ArrayIterator implements RecursiveIterator
{
   function hasChildren() {
      return is_array($this->current());
   }
   function getChildren() {
      if (empty($this->ref))
         $this->ref = new ReflectionClass($this);
      return $this->ref->newInstance($this->current());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



How does our Menu look?

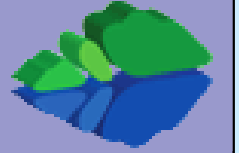
- ☑ Extend **`RecursiveArrayIterator`** but be typesafe
  - ☑ Ensure getChildren() returns the correct type
- ☑ Elements are non arrays
  - ☑ Recursion works slightly different
  - ☑ Override hasChildren() to not use is_array()
  - ☑ Keep existing getChildren() and other iterator methods

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



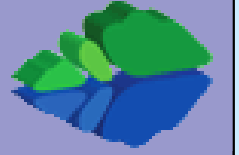
How does our Menu look?

```
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?

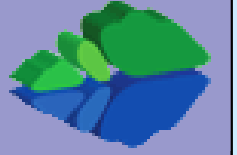
```
class SubMenu extends Menu implements MenuItem
{
    protected $name, $link, $active, $visible;

    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

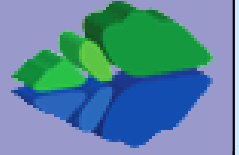
```php
<?php

$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

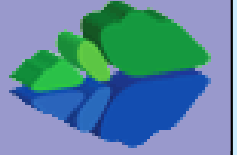
```
class RecursiveIteratorIterator
{
    /** @return $this->getInnerIterator()->hasChildren()*/
    function callHasChildren()

    /** @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, its sub menus and their sub menus.

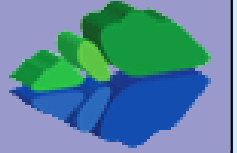


Reading a menu from array

```
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 recursive iteration...

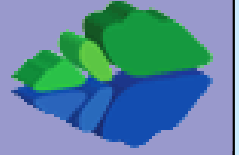
...a recursive structure.



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);
```

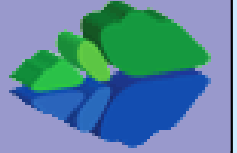
When recursing we create a new unnamed SubMenu and make it the new top level element of our 'level' storage.



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);
```

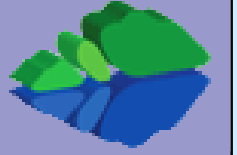
At the end of a sub array in our case representing a sub menu when pop that sub menu thus going to it's parent menu.



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);
```

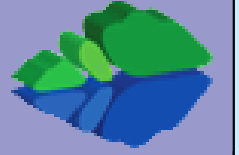
All elements in our definition that are not sub arrays are meant to end up as entries so we only want leaves as elements.



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);
```

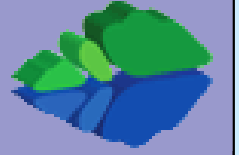
Now let us use the thing to fill in the menu from the definition in the array $def.



Output HTML

- ☑ Problem how to format the output using
  - ☞ 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";
   }
}
```

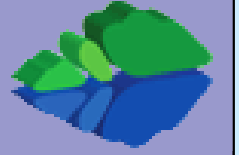


Output HTML

- ☑ 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
```

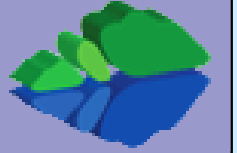


Output HTML

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

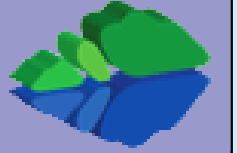
```
<ul>
<li>1</li>
<li>2</li>
  <ul>
  <li>31</li>
  <li>32</li>
  </ul>
<li>4</li>
</ul>
```

```php
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?

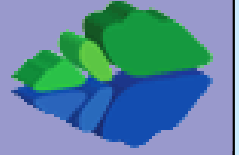
- ☑ Why did we use SPL here?
  - ☑ More reliability
    - ☑ Fix one time – no problem in finding all incarnations
  - ☑ Easier to change something without touching other stuff
    - ☑ Functional separation
    - ☑ Code ruse
    - ☑ Responsability control



OuterIterator

- ☑ OuterIterator is the interface for iterator wrapper
  - ☑ Allows read access to its inner iterator

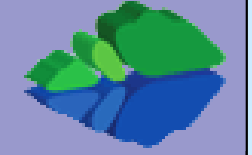
```
interface OuterIterator extends Iterator
{
   function getInnerIterator();
}
```



IteratorIterator

- ☑ IteratorIterator is an unspecified iterator wrapper

```
class IteratorIterator implements OuterIterator {
  function __construct(Traversable $iter, $classname)
  {
     $this->iterator = $iter;
  }
  function getInnerIterator() { return $this->iterator; }
  function valid()   {return $this->iterator->valid();  }
  function key()     {return $this->iterator->key();    }
  function current() {return $this->iterator->current();}
  function next()    {return $this->iterator->next();   }
  function rewind()  {return $this->iterator->rewind(); }
  function __call($func, $params) {
     return call_user_func_array(
       array($this->itator, $func), $params);
  }
  private $iterator;
}
```

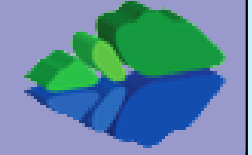


Filtering

Problem

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

```
<?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
}
?>
```

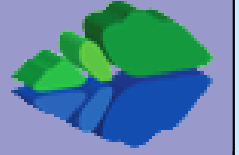


Filtering

Solution to filter the incoming data

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

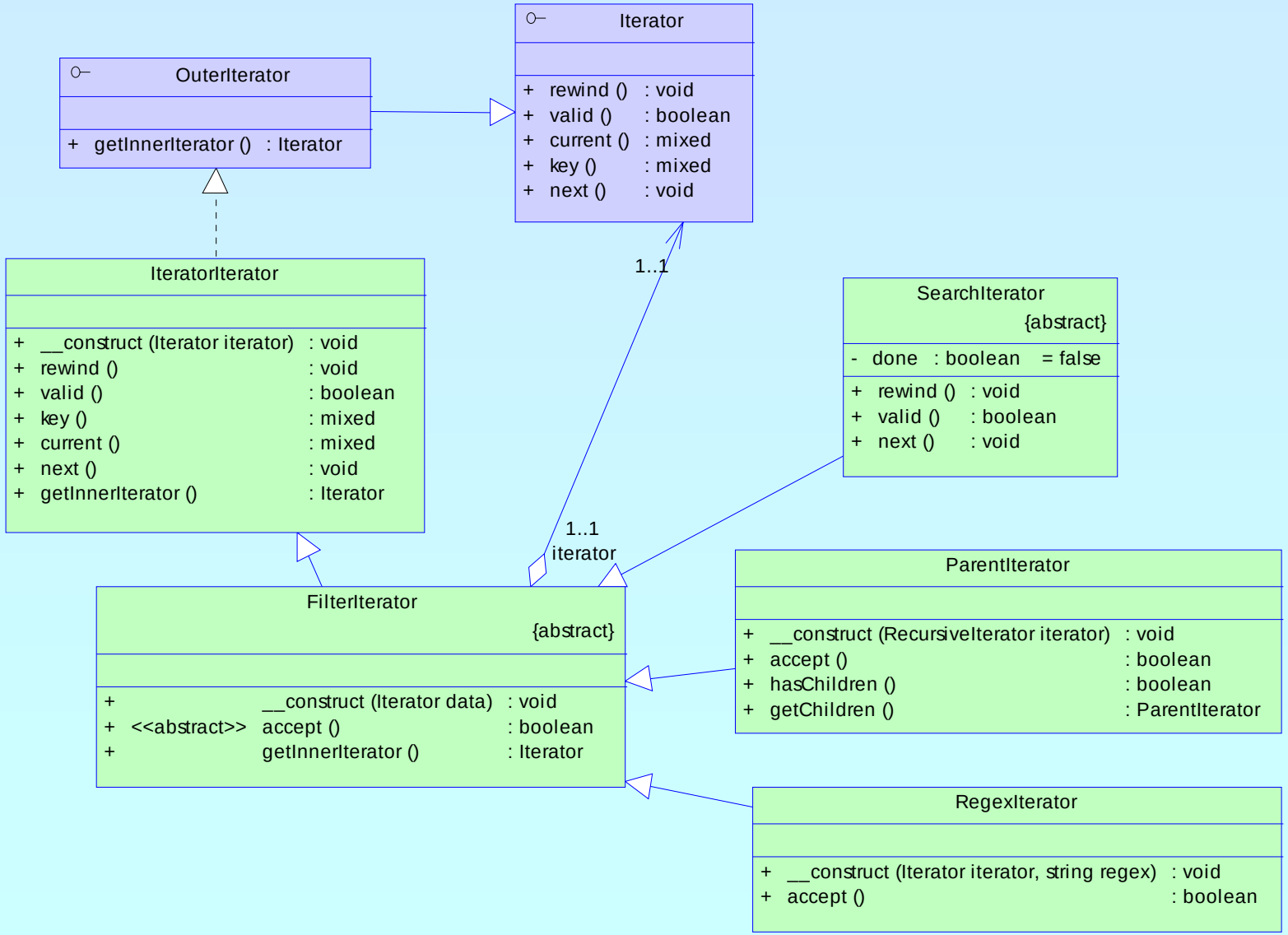
```
interface MenuItem
{
    // ...

    function isActive()  // return true if active
    function isVisible() // return true if visible
}
```

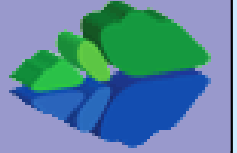


FilterIterator

- ☑ **`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

FilterIterator



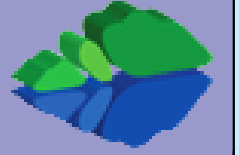


Debug Session

```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
```

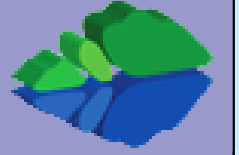
```php
<?php
class EvenFilter extends FilterIterator {
   function __construct(Iterator $it) {
      parent::__construct($it); }
   function accept() {
      return $this->current() % 2 == 0; }
}
class MyIterator implements Iterator {
   function __construct($ar) {
      $this->ar = $ar; }
   function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

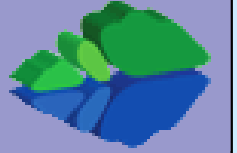


Putting it together

☑ 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";
    }
}
```

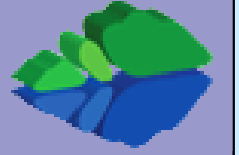


What now

- ☑ 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

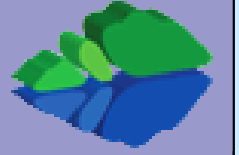


Using PDO

☑ Change **Menu** to read from database

- ☞ PDO supports Iterator based access
- ☞ PDO can create and read into objects
- ☞ PDO is integrated since PHP 5.1

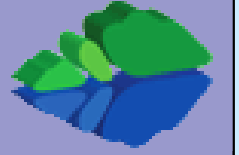
```
<?php
$db = new PDO("mysql://...");
$stmt= $db->prepare("SELECT ... FROM Menu ...", "Menu");
foreach($stmt->execute() as $m) {
      // fetch now returns Menu instances
      echo $m; // call $m->__toString()
}
?>
```



Using XML

- ☑ Change **Menu** to inherit from **SimpleXMLIterator**
  - ☑ Which is already a **RecursiveIterator**
  - ☑ We need to make it create **Menu** instances for children

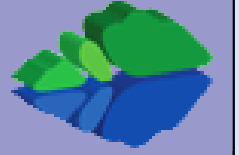
```
class Menu extends SimpleXMLIterator
{
    static function factory($xml)
    {
       return simplexml_load_string($xml, 'Menu');
    }
    function isActive() {
       return $this['active']; // access attribute
    }
    function isVisible() {
       return $this['visible']; // access attribute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



Speaking of XML

- ☑ SPL makes SimpleXML recursion aware
  - ☑ Use `simplexml_load_(file|string)` with 2nd param

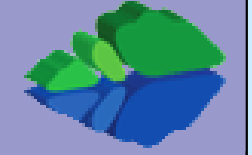
```php
<?php

$xml = simplexml_load_file($argv[1],'SimpleXmlIterator');

foreach(new RecursiveIteratorIterator($xml) as $e)
{
   if (isset($e['href']))
   {
      echo $e['href'] . "\n";
   }
}

?>
```



Speaking of XML

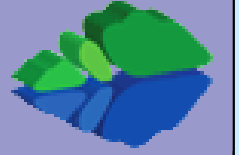
- ☑ SPL makes SimpleXML recursion aware
  - ☑ Use `simplexml_load_(file|string)` with 2nd param
  - ☑ Or `SimpleXmlIterator` direct by constructor

```php
<?php

$xml = new SimpleXmlIterator($argv[1], 0, true);

foreach(new RecursiveIteratorIterator($xml) as $e)
{
   if (isset($e['href']))
   {
      echo $e['href'] . "\n";
   }
}

?>
```



Another example

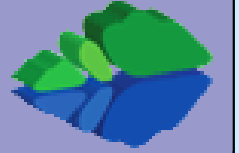
- ☑ An **`OuterIterator`** may not pass data from its **`InnerIterator`** directly
- ☑ Provide a 404 handler that looks for similar pages
  - ☑ Use **`RecursiveDirectoryIterator`** to test all files
  - ☑ Use **`FilterIterator`** to skip all files with low similarity
  - ☑ Sort by similarity -> convert iterated data into an array



Looking for files

- In PHP 4 you would use standard directory funcs

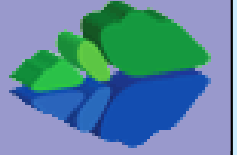
```
function search($path, $search, $limit, &$files) {
   if ($dir = @opendir($path)) {
      while (($found = readdir($dir) !== false) {
         switch(filetype("$path/$found")) {
         case 'file':
            if (($s = similariry($search, $found)) >= $limit) {
               $files["$path/$found"]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



Looking for files

- ☑ PHP 5 offers `RecursiveDirectoryIterator`

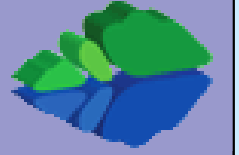
```
class FindSimilar extends FilterIterator {
   protected $search, $limit, $key;
   function __construct($root, $search, $limit) {
      parent::__construct(
         new RecursiveIteratorIterator(
            new RecursiveDirectoryIterator($root)));
      $this->search = $search;
      $this->limit = 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;
   }
}
```



Looking for files

- Filtering the `RecursiveDirectoryIterator`

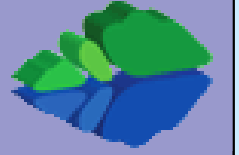
```
class FindSimilar extends FilterIterator {
   protected $search, $limit, $key;
   function __construct($root, $search, $limit) {
      parent::__construct(
         new RecursiveIteratorIterator(
            new RecursiveDirectoryIterator($root)));
      $this->search = $search;
      $this->limit = 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

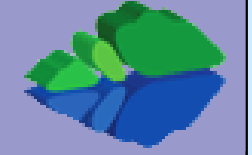
```
<html>
<head><title>File not found</title></head>
<body>
<?php
if (array_key_exists('missing', $_REQUEST)) {
   $missing = urldecode($_REQUEST['missing']);
   url_split($missing,$protocol,$host,$path,$ext,$query);
   $it = new FindSimilar($path);
   $files = iterator_to_array($it, $missing, 35);
   asort($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>
```



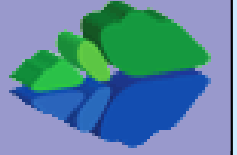
Error404.php

- ☑ Sorting requires iterator to array conversion

```
<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>
```



More Iterators pliezzze



Limiting iterators

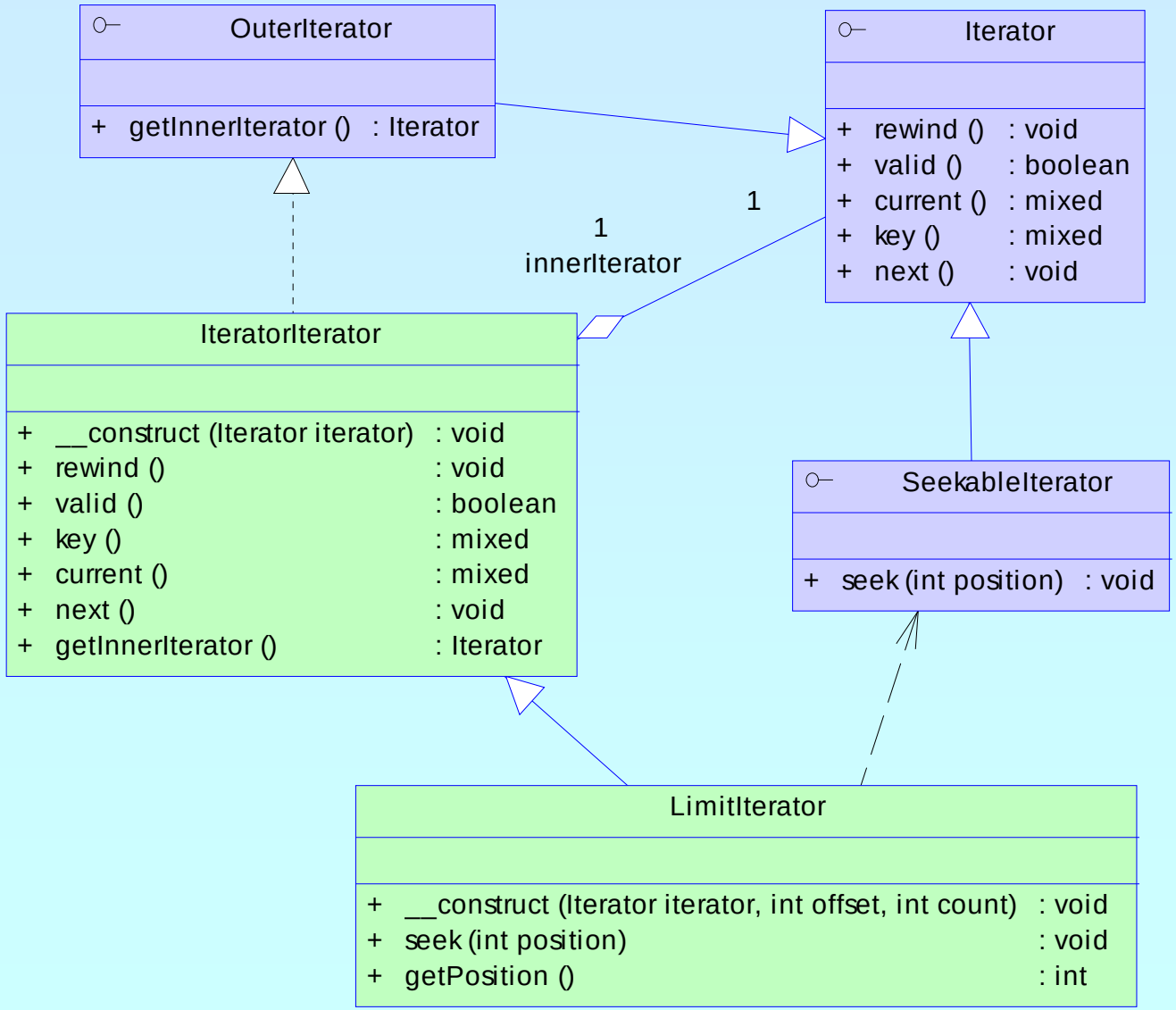
- **LimitIterator** allows to limit the returned values

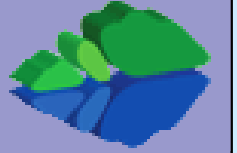
Compareable to **LIMIT** of some SQL dialects

- You can specify the start offset
- You can specify the number of returned values

- When the inner Iterator is a **SeekableIterator** then method seek will be used. Otherwise seek operation will be manually.

Limiting iterators

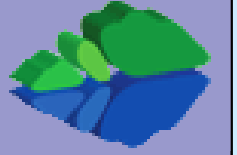




Limits of the LimitIterator

- ☑ Here using LimitIterator != limited use

```
<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);
   $it = new LimitIterator($it, 10);
   $files = iterator_to_array($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>
```



Appending Iterators

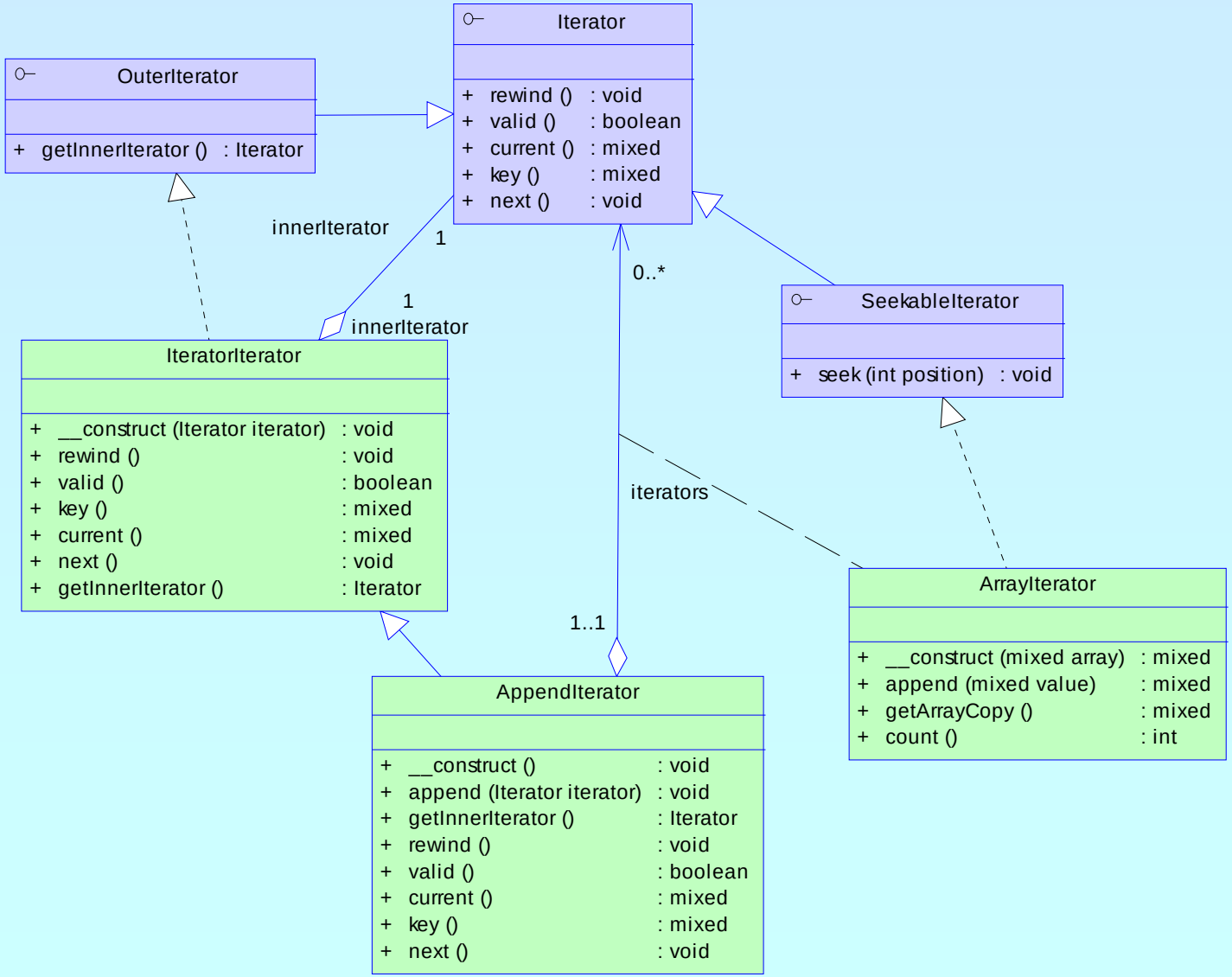
- **AppendIterator** allows to concatenate Iterators

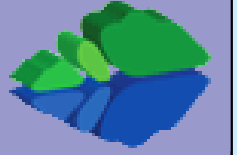
Compareable to SQL clause **UNION**

- Uses a private **ArrayIterator** to store Iterators

- **AppendIterator::append($it)**
  - allows to append iterators
  - does not call `rewind()`
  - if `$this` is invalid `$this` will move to appended iterator

Appending Iterators





Getting rid of rewind

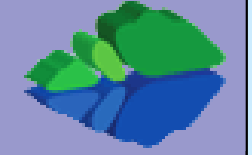
- ☑ **NoRewindIterator** allows to omit rewind calls

Especially helpful when appending with

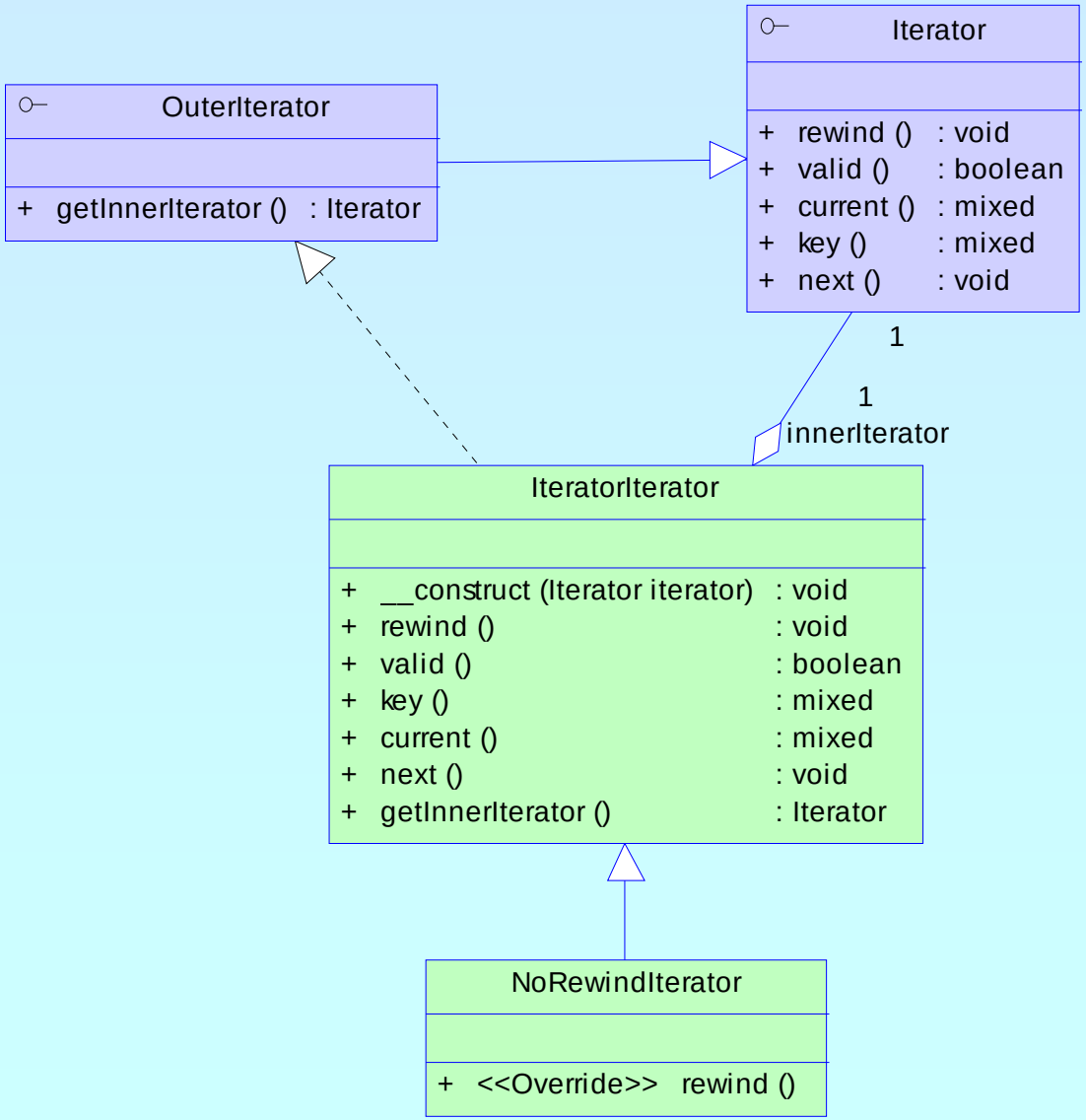
- ☑ **ArrayObject::append**()
- ☑ **ArrayIterator::append**()
- ☑ **AppendIterator::append**()

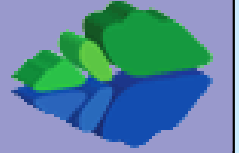
if your code would otherwise force a rewind()

Also helpfull when skipping a head part of iteration



Getting rid of rewind()

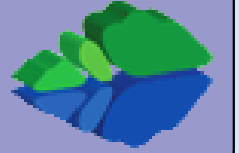




Limit and no rewind

- Example: Show the n-th set of filtered data

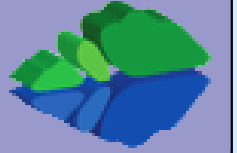
```
$input = array(0,1,2,3,4,5,6,7,8,9); $len = 2; $set = 1;

class EvenFilter extends FilterIterator
{
   function accept() {
      return $this->current() % 2 == 0;
   }
}

$ar = new EvenFilter(new ArrayIterator($input));
$ar->rewind();
$ar = new NoRewindIterator($ar);
while(--$set >= 0) {
   foreach(new LimitIterator($ar, 0, $len) as $v)  ;
}

foreach(new LimitIterator($ar, 0, $len) as $v) {
   echo "$v\n";
}
```



Limit and no rewind

- ☑ Provide Input data and a filter

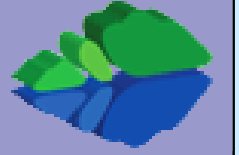
```
$input = array(0,1,2,3,4,5,6,7,8,9); $len = 2; $set = 1;

class EvenFilter extends FilterIterator
{
   function accept() {
      return $this->current() % 2 == 0;
   }
}

$ar = new EvenFilter(new ArrayIterator($input));
$ar->rewind();
$ar = new NoRewindIterator($ar);
while(--$set >= 0) {
   foreach(new LimitIterator($ar, 0, $len) as $v)  ;
}

foreach(new LimitIterator($ar, 0, $len) as $v) {
   echo "$v\n";
}
```



Limit and no rewind

- ☑ Must rewind before wrapping in NoRewindIterator

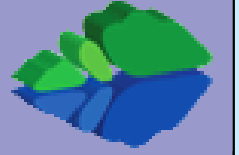
```
$input = array(0,1,2,3,4,5,6,7,8,9); $len = 2; $set = 1;

class EvenFilter extends FilterIterator
{
   function accept() {
      return $this->current() % 2 == 0;
   }
}

$ar = new EvenFilter(new ArrayIterator($input));
$ar->rewind();
$ar = new NoRewindIterator($ar);
while(--$set >= 0) {
   foreach(new LimitIterator($ar, 0, $len) as $v)  ;
}

foreach(new LimitIterator($ar, 0, $len) as $v) {
   echo "$v\n";
}
```



Limit and no rewind

- ☑ Skip top n-1 sets

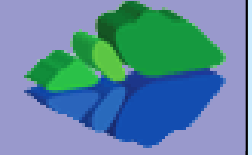
```
$input = array(0,1,2,3,4,5,6,7,8,9); $len = 2; $set = 1;

class EvenFilter extends FilterIterator
{
   function accept() {
      return $this->current() % 2 == 0;
   }
}

$ar = new EvenFilter(new ArrayIterator($input));
$ar->rewind();
$ar = new NoRewindIterator($ar);
while(--$set >= 0) {
   foreach(new LimitIterator($ar, 0, $len) as $v)  ;
}

foreach(new LimitIterator($ar, 0, $len) as $v) {
   echo "$v\n";
}
```



Limit and no rewind

- ☑ Showing/Using remaining data (n-th set)

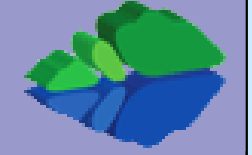
```
$input = array(0,1,2,3,4,5,6,7,8,9); $len = 2; $set = 1;

class EvenFilter extends FilterIterator
{
   function accept() {
      return $this->current() % 2 == 0;
   }
}

$ar = new EvenFilter(new ArrayIterator($input));
$ar->rewind();
$ar = new NoRewindIterator($ar);
while(--$set >= 0) {
   foreach(new LimitIterator($ar, 0, $len) as $v)  ;
}

foreach(new LimitIterator($ar, 0, $len) as $v) {
   echo "$v\n";
}
```



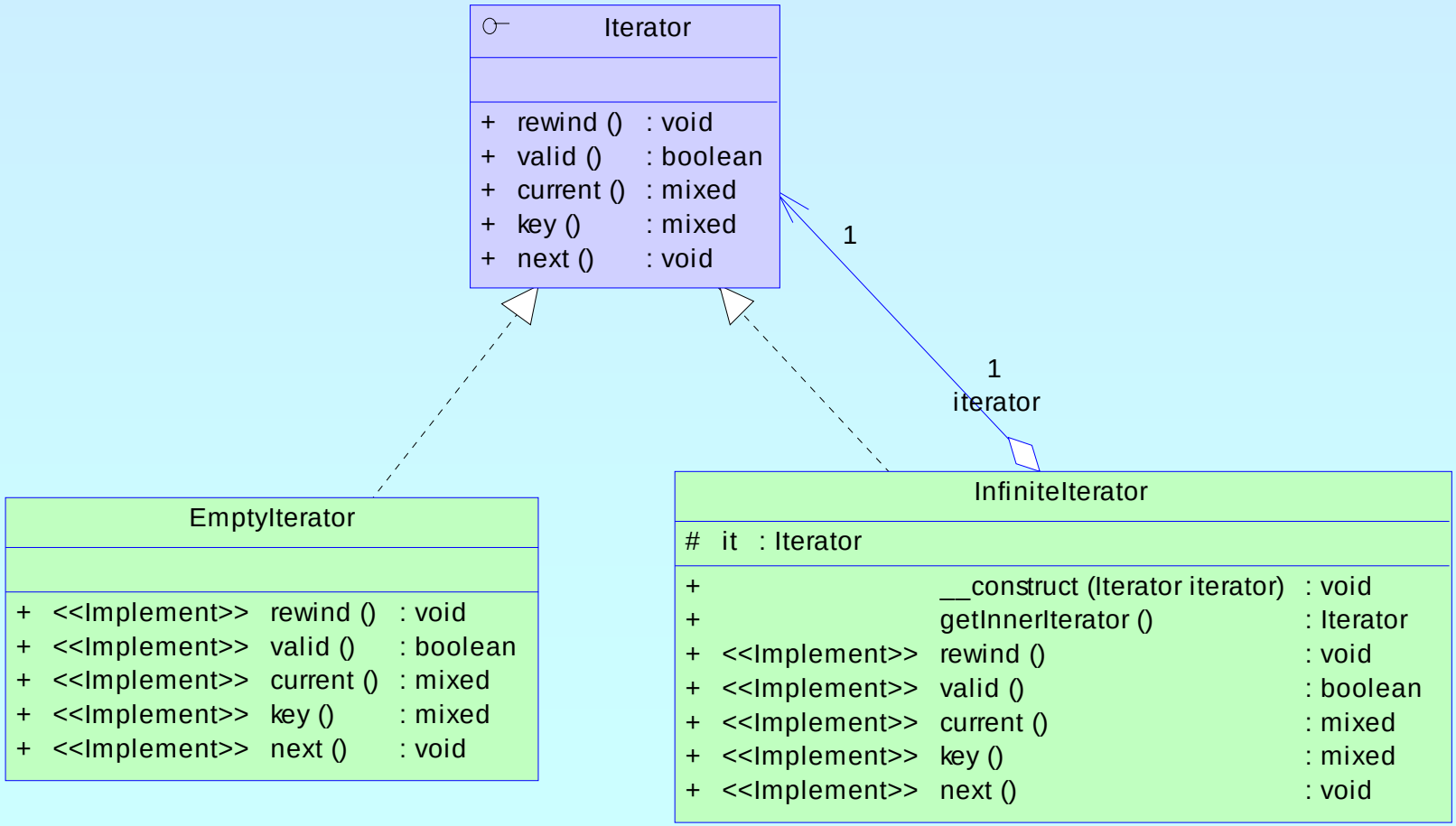
Vacuity & Infinity

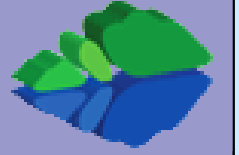
Sometimes it is helpful to have

- ☑ **EmptyIterator** as a placeholder for no data
- ☑ **InfiniteIterator** to endlessly repeat data in an iterator



Vacuity & Infinity



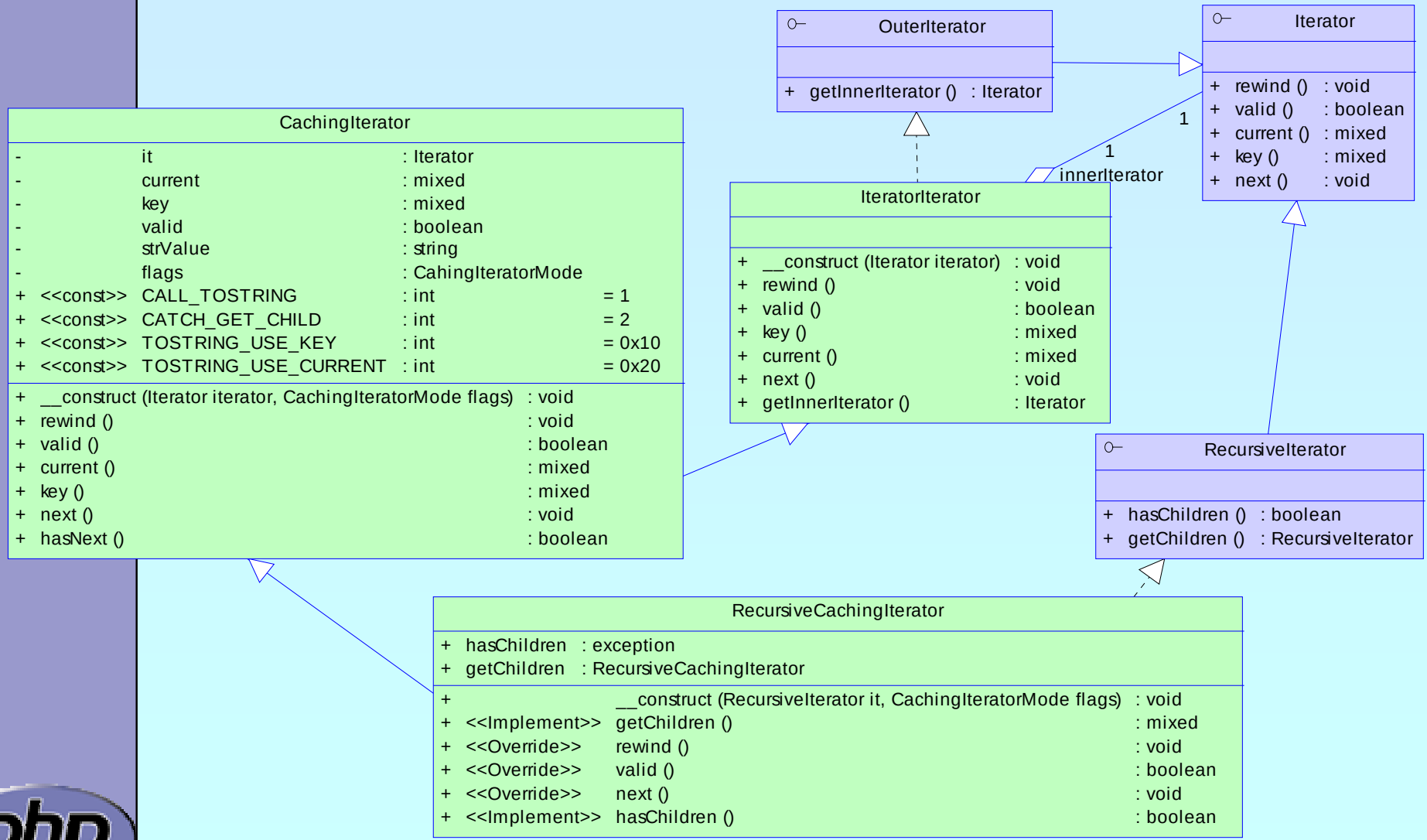


hasNext ?

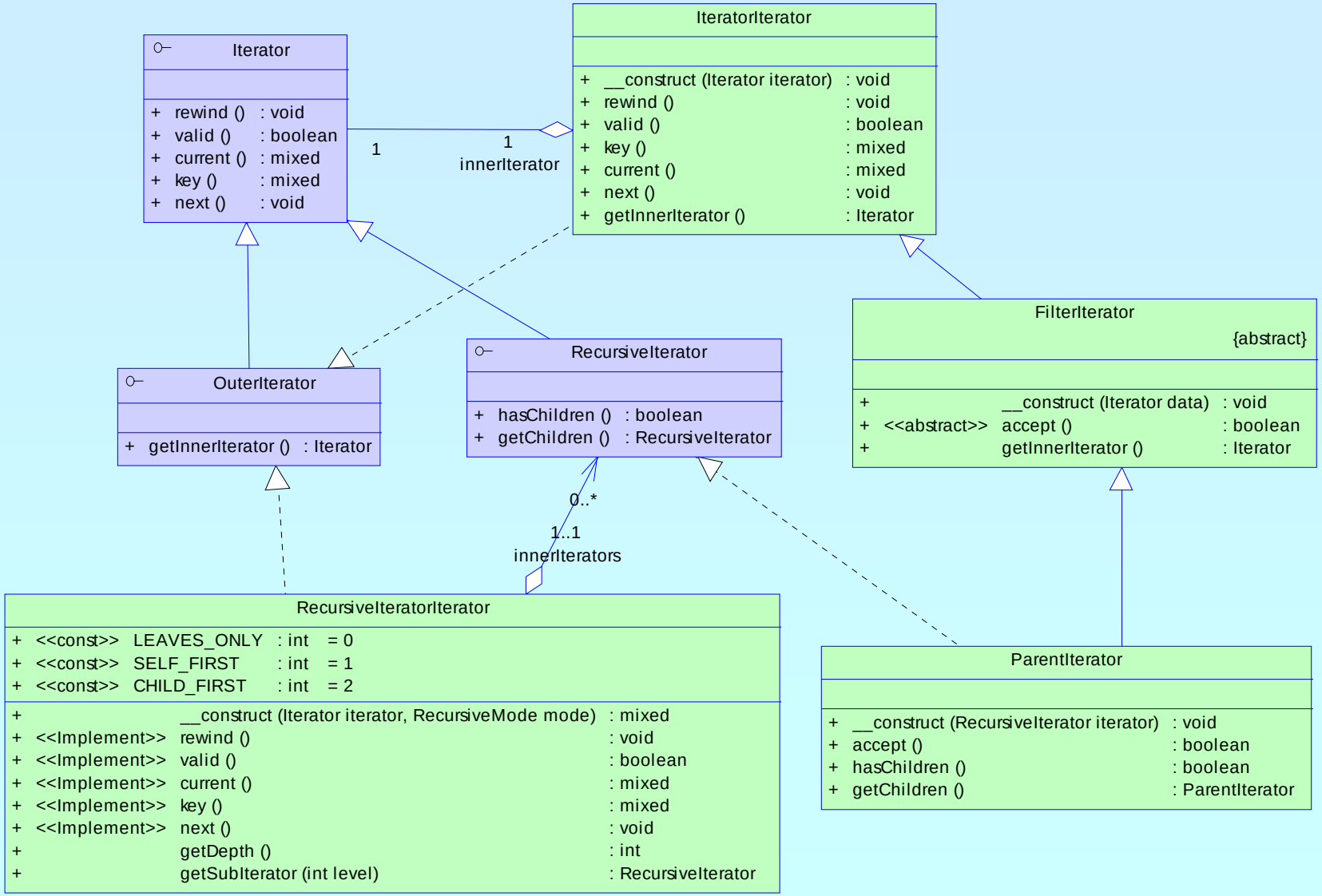
- **CachingIterator** caches the current element
  - This allows to know whether one more value exists
- **RecursiveCachingIterator** does this recursively
  - This allows to draw tree graphics

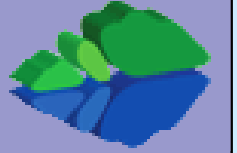
```
marcus@frodo /usr/src/php-cvs $ php ext/spl/examples/tree.php ext/spl
ext/spl
|-CVS
|-examples
| |-CVS
| \-tests
|   \-CVS
\-tests
  \-CVS
```

hasNext ?


OuterIterator
+ getInnerIterator () : Iterator
Iterator
+ rewind () : void
+ valid () : boolean
+ current () : mixed
+ key () : mixed
+ next () : void
1
1
innerIterator
CachingIterator
- it : Iterator
- current : mixed
- key : mixed
- valid : boolean
- strValue : string
- flags : CahingIteratorMode
+ <<const>> CALL_TOSTRING : int = 1
+ <<const>> CATCH_GET_CHILD : int = 2
+ <<const>> TOSTRING_USE_KEY : int = 0x10
+ <<const>> TOSTRING_USE_CURRENT : int = 0x20
+ __construct (Iterator iterator, CachingIteratorMode flags) : void
+ rewind () : void
+ valid () : boolean
+ current () : mixed
+ key () : mixed
+ next () : void
+ hasNext () : boolean
IteratorIterator
+ __construct (Iterator iterator) : void
+ rewind () : void
+ valid () : boolean
+ key () : mixed
+ current () : mixed
+ next () : void
+ getInnerIterator () : Iterator
RecursiveIterator
+ hasChildren () : boolean
+ getChildren () : RecursiveIterator
RecursiveCachingIterator
+ hasChildren : exception
+ getChildren : RecursiveCachingIterator
+ __construct (RecursiveIterator it, CachingIteratorMode flags) : void
+ <<Implement>> getChildren () : mixed
+ <<Override>> rewind () : void
+ <<Override>> valid () : boolean
+ <<Override>> next () : void
+ <<Implement>> hasChildren () : boolean

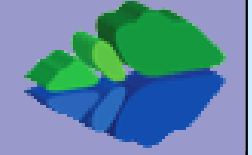
Parents only



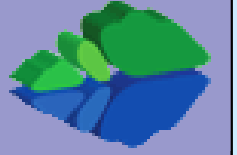


Conclusion so far

- ☑ 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

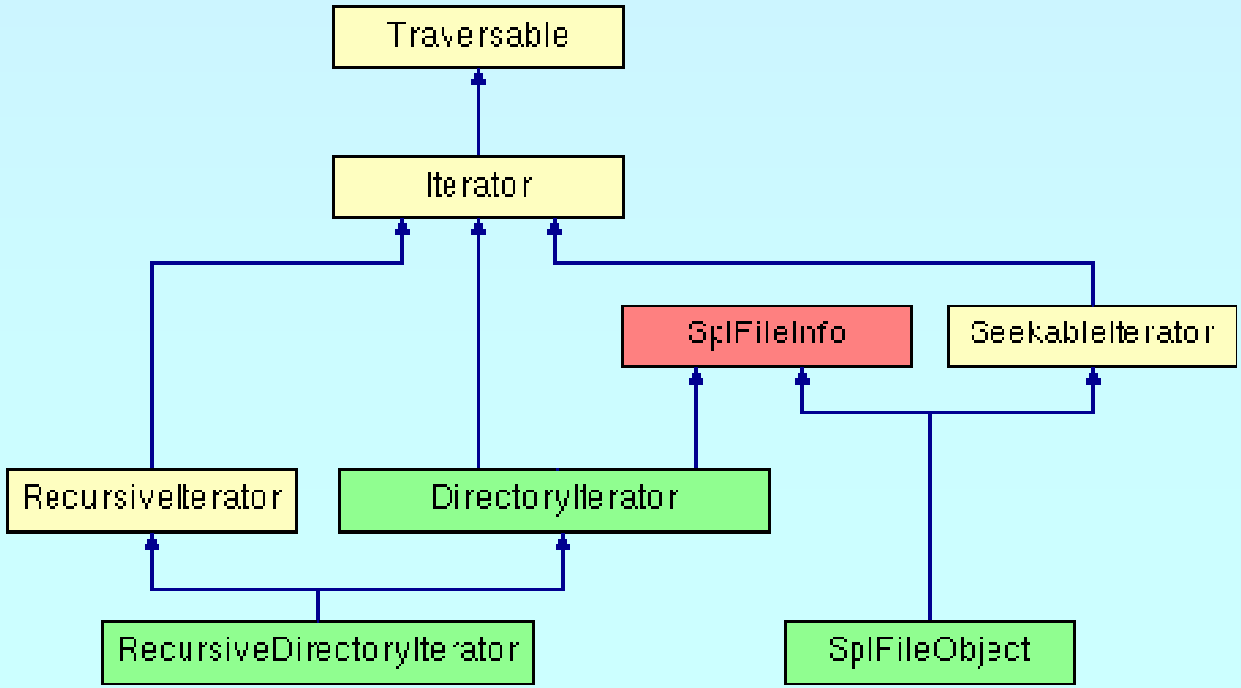


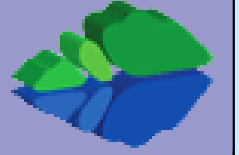
Files & Directories



File and directory handling

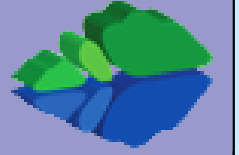
- ☑ SplFileInfo is the *filesystem information* base class
  - ☑ getATime, getCTime, getMTime, isDir, isFile, isLink
  - ☑ getFilename, getPath, getPathname
  - ☑ getPerms, getOwner, getINode, getType
  - ☑ getFileInfo, getPathInfo
  - ☑ openFile





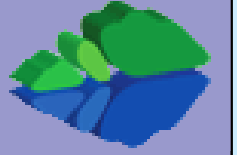
File and directory handling

```
class SplFileInfo {
   private $fname;

   function __construct($file_name) {
      $this->fname = $file_name;
   }

   function getFilename() {return basename($this->fname); }
   function getPath()     {return dirname($this->fname);  }
   function getPathname() {return $this->fname; }
   function __toString()  {return $this->getPathname(); }

   function isDir()       {return is_dir($this->fname); }
   function isFile()      {return is_file($this->fname); }
   function isLink()      {return is_link($this->fname); }
   function getATime()    {return fileATime($this->fname); }
   function getCTime()    {return fileCTime($this->fname); }
   function getMTime()    {return fileMTime($this->fname); }
   function getSize()     {return fsize($this->fname); }
   // more file functions
```



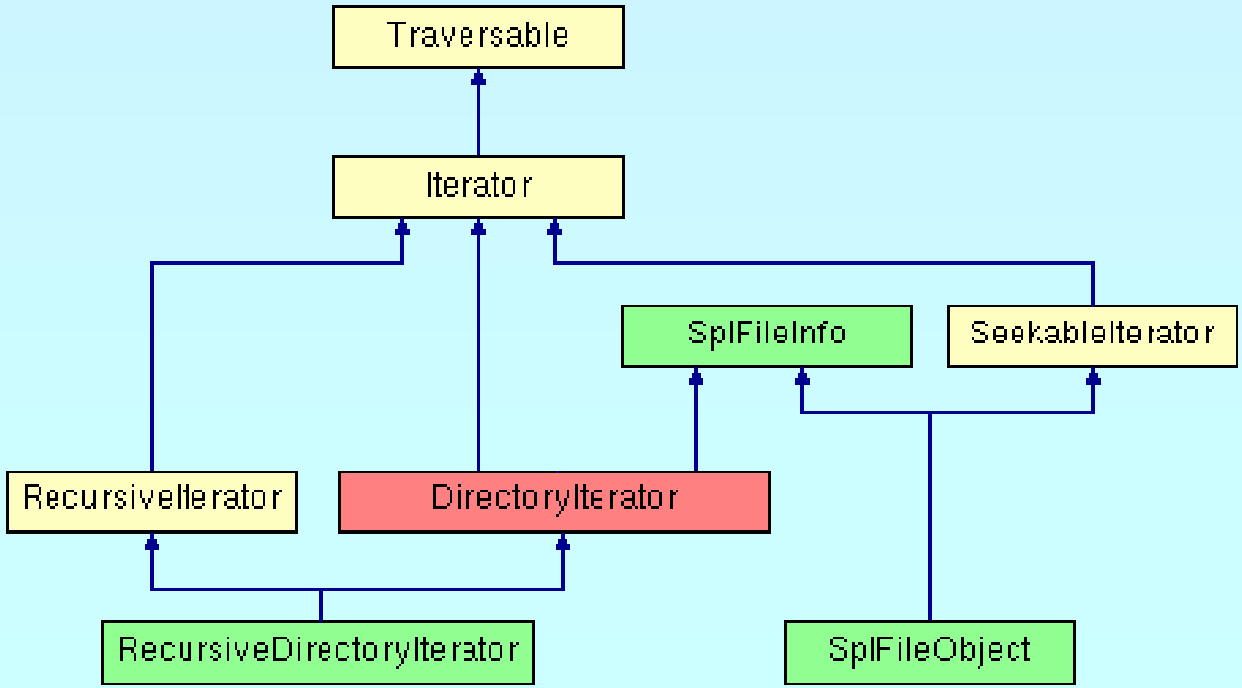
File and directory handling

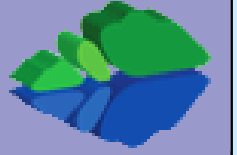
```
class SplFileInfo {
   // continued
   private $info_class = 'SplfileInfo';
   private $file_class = 'SplFileObject';

   function getFileInfo($class_name = NULL) {
      if (!isset($class)) $class = $this->info_class;
      $r = new ReflectionClass($class);
      return $r->newInstance($this->getFilename());
   }

   function openFile($mode = 'r') {
      $r = new ReflectionClass($this->file_class);
      return $r->newInstance($this->getFilename(),$mode);
   }

   function setFileClass($class_name) {
      if ($class_name instanceof SplFileInfo)
         $this->file_class = $class_name;
   }
}
```



File and directory handling

- DirectoryIterator for non recursive dir handling
  - `current()` returns `$this`
  - `key()` returns numeric index
  - `isDot()` returns whether current entry is '.' or '..'

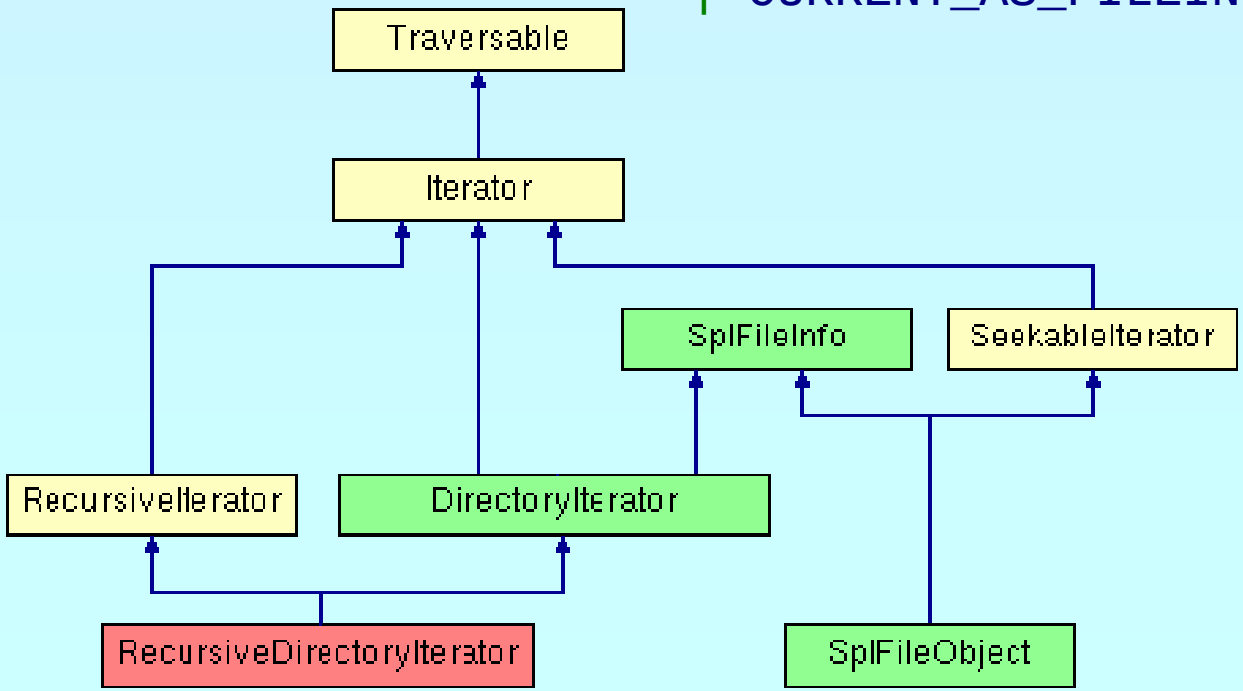


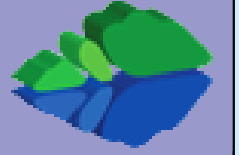


File and directory handling

- ☑ RecursiveDirectoryIterator goes into subdirs
  - ☑ Supports different modes for key() and cirrent()

```
CURRENT_AS_SELF = 0          KEY_AS_PATHNAME = 0
CURRENT_AS_PATHNAME          KEY_AS_FILENAME
CURRENT_AS_FILEINFO
NEW_CURRENT_AND_KEY = KEY_AS_FILENAME
                    | CURRENT_AS_FILEINFO
```



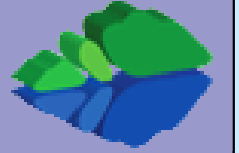


Putting it to the tree?

- Example: Retrieving the hierarchy of a filesystem

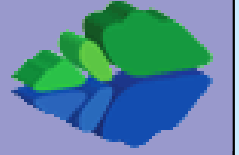
```
marcus@frodo /usr/src/php-cvs $ php ext/spl/examples/tree.php ext/spl
ext/spl
|-CVS
|-examples
| |-CVS
| \-tests
|   \-CVS
\-tests
  \-CVS
```

- Need to recursively iterate over the filesystem
  - ➔ `RecursiveDirectoryIterator`
- Efficiently ignore files
  - ➔ `ParentIterator`
- On each level check whether more elements exist
  - ➔ `CachingIterator`



Providing structure

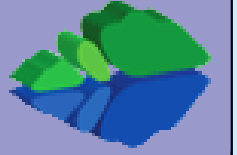
```
class DirectoryTreeIterator
   extends RecursiveIteratorIterator
{
   function __construct($path) {
      parent::__construct(new RecursiveCachingIterator(
         new RecursiveDirectoryIterator($path,
            RecursiveDirectoryIterator::KEY_AS_FILENAME),
         CachingIterator::CALL_TOSTRING),
      parent::SELF_FIRST);
   }

   function current() {
      $cur = "";
      for ($l = 0; $l < $this->getDepth(); $l++) {
         $cur .= $this->getSubIterator($l)->hasNext()
                ? "| " : "  ";
      }
      $l = $this->getSubIterator($l);
      return $cur .($l->hasNext() ? "|-" : "\-") .(string)$l;
   }
}
```



Like pieces of a puzzle

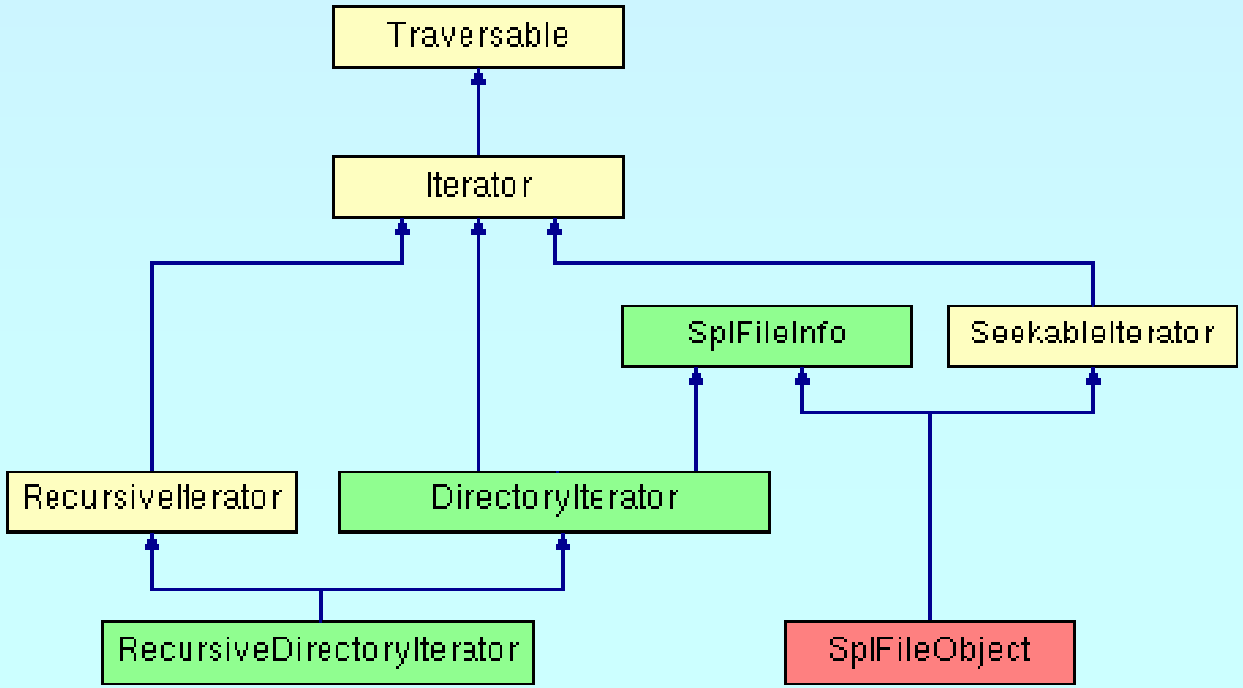
- ☑ Apply ParentIterator as filter

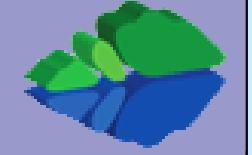
```
class DirectoryGraphIterator
   extends DirectoryTreeIterator
{
   function __construct($path)
   {
      parent::__construct(new RecursiveCachingIterator(
         new ParentIterator(
            new RecursiveDirectoryIterator($path,
               RecursiveDirectoryIterator::KEY_AS_FILENAME)),
         CachingIterator::CALL_TOSTRING),
      parent::SELF_FIRST);
   }
}

foreach(new DirectoryGraphIterator($argv[1]) as $file) {
   echo $file . "\n";
}
```



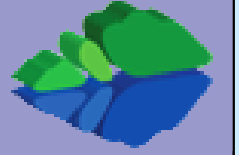
File and directory handling

- ☑ SplFileObject allows to access files as an iterator
  - ☑ Allows to skip empty lines
  - ☑ Allows to retrieve lines as csv (PHP 5.2)





Making __autoload usable



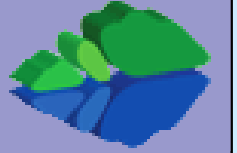
Dynamic class loading

☑ `__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

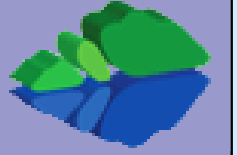
☠ Only one single loader model is possible



__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

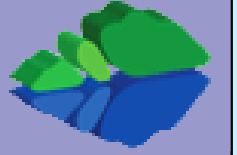
```php
<?php
function __autoload($class_name)
{
   require_once(
      dirname(__FILE__) . '/' . $class_name . '.p5c');
}
?>
```



SPL's class loading

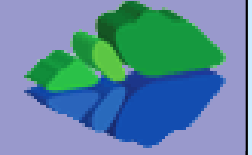
- ☑ Supports fast default implementation
  - ☑ Look into path's specified by INI option include_path
  - ☑ Look for specified file extensions (.inc, .inc.php)
- ☑ Ability to register multiple user defined loaders
- ☑ Overwrites ZEND engine's __autoload() cache
  - ☑ You need to register __autoload if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

```
<?php
    spl_autoload_register('spl_autoload');
    if (function_exists('__autoload')) {
        spl_autoload_register('__autoload');
    }
?>
```

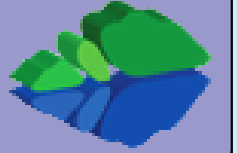


SPL's class loading

- ☑ `spl_autoload($class_name,$extensions=NULL)`
  - Load a class from in include path
  - Fast c code implementation
- ☑ `spl_autoload_extensions($extensions=NULL)`
  - Get or set filename extensions
- ☑ `spl_autoload_register($loader_function)`
  - Register 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 through registered class loaders
  - Uses `spl_autoload()` as fallback



Exceptions

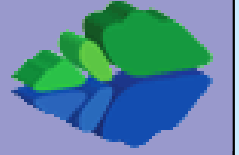


Exceptions

☑ Respect these rules

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

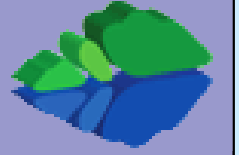
```php
<?php
try {
    // your code
    throw new Exception();
}
catch (Exception $e) {
    // exception handling
}
?>
```



Exception specialization

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

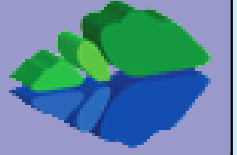
```php
<?php
class YourException extends Exception {
}
try {
    // your code
    throw new YourException();
}
catch (YourException $e) {
    // exception handling
}
catch (Exception $e) {
    // exception handling
}
?>
```



Exception specialization

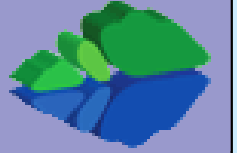
- ☑ Exception blocks can be nested
- ☑ Exceptions can be re thrown

```
<?php
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
}
?>
```



Practial use of exceptions

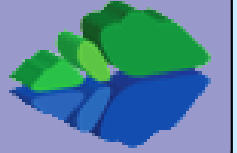
- ☑ Constructor failure
- ☑ Converting errors/warnings to exceptions
- ☑ Simplify error handling
- ☑ Provide additional error information by tagging



Constructor failure

- ☑ In PHP 4.4 you would simply `unset($this)`
- ☑ Provide a param that receives the error condition

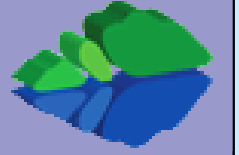
```
<?php
class Object
{
   function __construct( &$failure)
   {
      $failure = true;
   }
}
$error = false;
$o = new Object($error);
if (!$error) {
   // error handling, NOTE: the object was constructed
   unset($o);
}
?>
```



Constructor failure

- ☑ In 5 constructors do not return the created object
- ☑ Exceptions allow to handle failed constructors

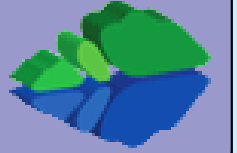
```
<?php
class Object
{
   function __construct()
   {
      throw new Exception;
   }
}
try {
   $o = new Object;
}
catch (Exception $e) {
   echo "Object could not be instantiated\n";
}
?>
```



Convert Errors to Exceptions

- ☑ Implementing PHP 5.1 class ErrorException

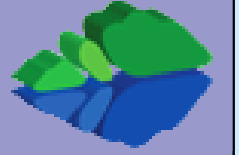
```
<?php

class ErrorException extends Exception
{
   protected $severity;
   function __construct($msg, $code, $errno, $file,$line)
   {
      parent::__construct($message, $code);
      $this->severity = $severity;
      $this->file = $file;
      $this->line = $line;
   }
   function getSeverity() {
      return $this->severity;
   }
}

?>
```



Convert Errors to Exceptions

- ☑ Implementing the error handler

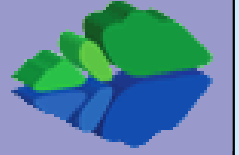
```
<?php

function ErrorsToExceptions($errno, $msg, $file, $line)
{
   throw new ErrorException($msg, 0, $errno,$file,$line);
}

set_error_handler('ErrorsToExceptions');

?>
```



Simplify error handling

- ☑ Typical database access code contains lots of if's

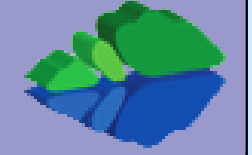
```
<html><body>
<?php
$ok = false;
$db = new PDO('CONNECTION');
if ($db) {
   $res = $db->query('SELECT data');
   if ($res) {
      $res2 = $db->query('SELECT other');
      if ($res2) {
         // handle data
         $ok = true; // only if all went ok
      }
   }
}
if ($ok) echo '<h1>Service currently unabvailable</h1>';
?>
</body></html>
```



Simplify error handling

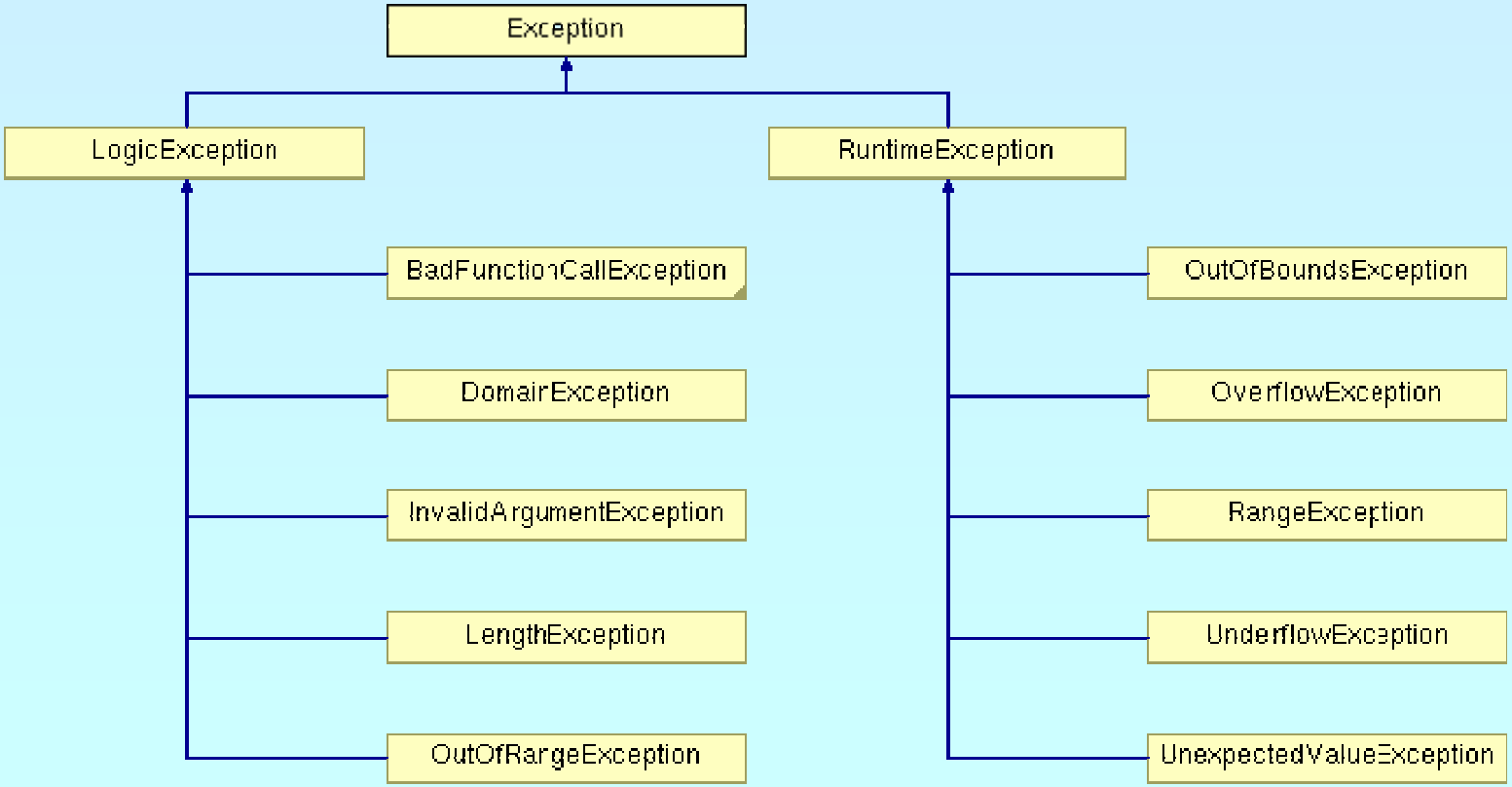
- ☑ Trade code simplicity with a new complexity

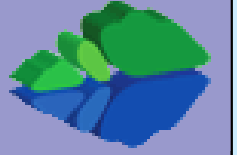
```
<html><body>
<?php
try {
   $db = new PDO('CONNECTION');
   $db->setAttribute(PDO::ATTR_ERRMODE,
                                 PDO::ERRMODE_EXCEPTION);
   $res = $db->query('SELECT data');
   $res2 = $db->query('SELECT other');
   // handle data
}
catch (Exception $e) {
   echo '<h1>Service currently unabvailable</h1>';
   error_log($e->getMessage());
}
?>
</body></html>
```



SPL Exceptions

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





General distinguishing

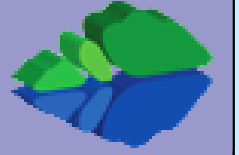
☑ `LogicException`

➔ Anything that could have been detected at compile time, during application design or by the good old technology:
"look precisely"

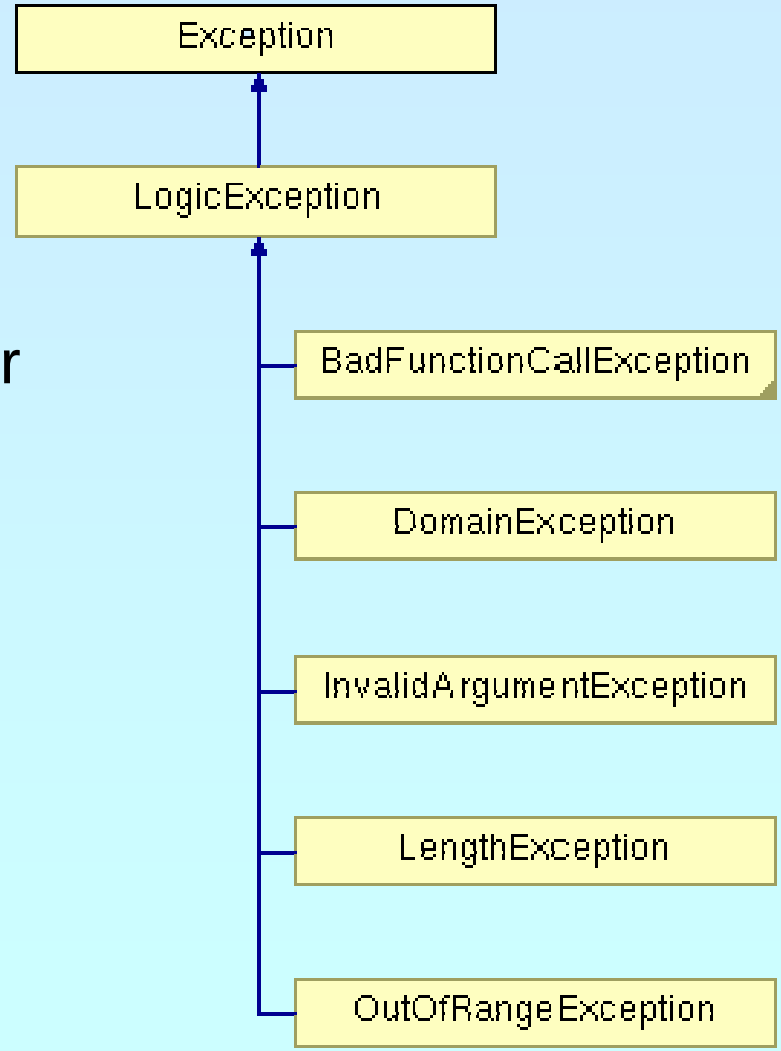
☑ `RuntimeException`

➔ Anything that is unexpected during runtime

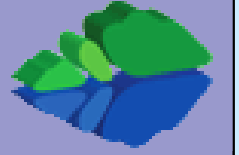
➔ Base Exception for all database extensions



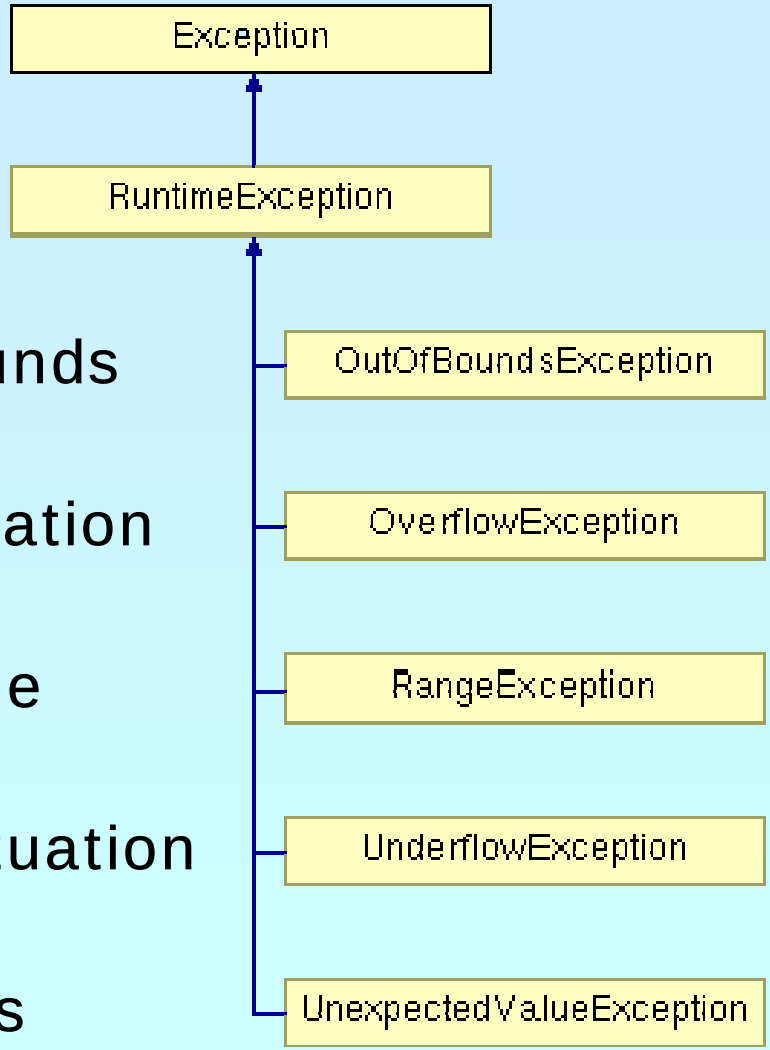
LogicException



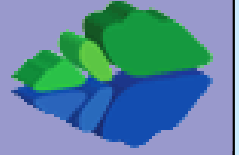
- ☑ Function not found or similar
  `BadMethodCallException`
- ☑ Value not in allowed domain
- ☑ Argument not valid
- ☑ Length exceeded
- ☑ Some index is out of range



RunTimeException



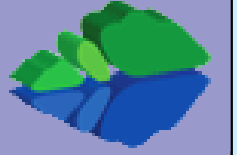
- ☑ An actual value is out of bounds
- ☑ Buffer or other overflow situation
- ☑ Value outside expected range
- ☑ Buffer or other underflow situation
- ☑ Any other unexpected values



Overloading __call

- ☑ If using __call, ensure only valid calls are made

```
abstract class MyIteratorWrapper implements Iterator
{
   function __construct(Iterator $it)
   {
      $this->it = $it;
   }
   function __call($func, $args)
   {
      $callee = array($this->it, $func);
      if (!is_callable($callee)) {
        throw new BadMethodCallException();
      }
      return call_user_func_array($callee, $args);
   }
}
```



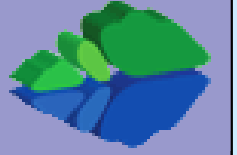
Expecting formatted data

☑ Opening a file for reading

Run-Time:

Filemight not be accessible or exist

```
$fo = new SplFileObject($file);
$fo->setFlags(SplFileObject::DROP_NEWLINE);
$data = array();
```



Expecting formatted data

- ☑ Reading a formatted file line by line

Run-Time:

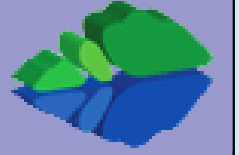
File might not be accessible or exist

```
$fo = new SplFileObject($file);
$fo->setFlags(SplFileObject::DROP_NEWLINE);
$data = array();
foreach($fo as $l) {
   if (/*** CHECK DATA ***/) {
      throw new Exception();
   }
   $data[] = $l;
}
```

Run-Time:

data is different for every execution

☑ `!preg_match($regex, $l)`	`UnexpectValueException`
☑ `count($l=split(',', $l)) != 3`	`RangeException`
☑ `count($data) > 100`	`OverflowException`



Expecting formatted data

- ☑ Cehcking data after pre-processing

Run-Time:

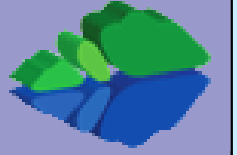
Filemight not be accessible or exist

```
$fo = new SplFileObject($file);
$fo->setFlags(SplFileObject::DROP_NEWLINE);
$data = array();
foreach($fo as $l) {
   if (!preg_match('/\d,\d/', $l)) {
      throw new UnexpectedValueException();
   }
   $data[] = $l;
}
// Checks after the file was read entirely
if (count($data) < 10) throw new UnderflowException();
if (count($data) > 99) throw new OverflowException();
if (count($data) < 10 || count($data) < 99)
                        throw new OutOfBoundsException();
```

Run-Time:

data is different for every execution

- ☑
- ☑
- ☑



Expecting formatted data

☑ Processing pre-checked data

```
$fo = new SplFileObject($file);
$fo->setFlags(SplFileObject::DROP_NEWLINE);
$data = array();
foreach($fo as $l) {
   if (!preg_match('/\d,\d/', $l)) {
      throw new UnexpectedValueException();
   }
   $data[] = $l;
}
if (count($data) < 10) throw new UnderflowException();
// maybe more precessing code
foreach($data as &$v) {
   if (count($v) == 2) {
      throw new DomainException();
   }
   $v = $v[0] * $v[1];
}
```

Run-Time:

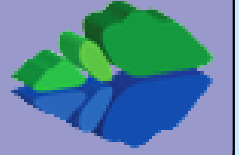
Filemight not be accessible or exist

Run-Time:

data is different for every execution

Compile-Time:

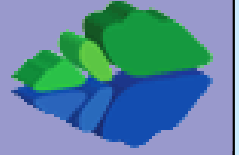
exception signals failed precondition



Iterator meets regex

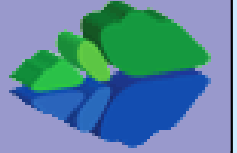
- ☑ Use a regular expression as accept function
- ☑ The regex gets compiled only once
- ☑ Example: Updating .cvsignore files

```
$dr = new RecursiveDirectoryIterator($path,
      RecursiveDirectoryIterator::CURRENT_AS_PATHNAME);

$it = new RecursiveRegExIterator($dr, '/.*\.cvsignore/',
      RecursiveRegExIterator::USE_KEY);

foreach(new RecursiveIteratorIterator($it) as $f) {
   $c = file($f);
   if (!in_array($c, '.libs')) {
      $c[] = '.libs';
      file_put_contents($f, $c);
   }
}
```



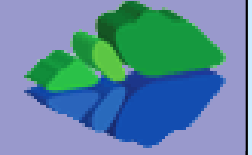
Reading CSV data

- ☑ `SplFileObject` will get more flags in 5.2 (5.1.4?)
  - ☑ `SplFileObject::DROP_NEW_LINE`
    (unchanged)
  - ☑ `SplFileObject::READ_AHEAD`
    Read in `rewind()`, `next()`
  - ☑ `SplFileObject::SKIP_EMPTY`
    Skip empty lines, includes `READ_AHEAD`
  - ☑ `SplFileObject::READ_CSV`
    Read CSV data instead of pure text

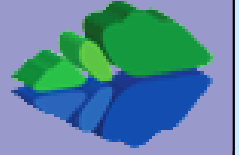


Cache as Cache Can

- ☑ New flag `CachingIterator::FULL_CACHE`
  - ☑ cache all read data — PHP 5.2
  - ☑ random access to read data using `ArrayAccess` — PHP 5.2
- ☑ More to come on that
  - ☑ Counting elements in cache using `Countable` — PHP 6
  - ☑ seek by implementing `Seekable` — not yet



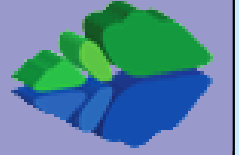
Upcoming stuff



SPL_Types

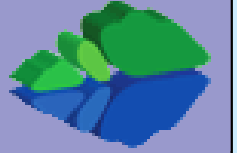
- Adds enumeration support to PHP
  - Overloaded objects that can represent only class consts

```
class Weekday extends SplEnum
{
   const Sunday = 0, Monday = 1, Tuesday = 2;
   const Wedensday = 3, Thursday = 4, Friday = 5;
   const Saturday =  6;
   const __default = Weekday::Monday;
}

$day = new Weekday(Weekday::Sunday);

foreach(Weekday::getConstList() as $name => $wday)
{
   echo $name . ": " . ($day == $wday ? "yes\n" :"no\n");
}
```



SPL_Types

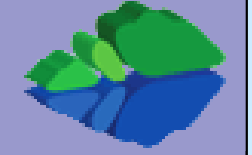
- ☑ `SplType` is the root class in extension Spl_Types
  - ☑ `SplType` and `SplEnum` are abstract classes
  - ☑ `SplType` can be made type strict (2nd ctor param)
  - ☑ `SplType` throws `UnxepectedValueException`
- ☑ `SplBool` can only represent `true` or `false`
- ☑ `SplInt` can only represent numeric values



Phar

- ☑ Archive format like Zip
- ☑ Extension is not really required
- ☑ Normal PHP scripts that can be executed with PHP

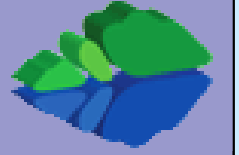
```php
<?php
$phar = new Phar($argv[1], 0, 'newphar');
$dir = new RecursiveDirectoryIterator($argv[2]);
$dir = new RecursiveIteratorIterator($dir);
$dir = new RegexIterator($dir, '/' . $argv[3] . '/');
$phar->begin();
foreach($dir as $file) {
   echo $file . "\n";
   copy($file, 'phar://newphar/' . $file);
}
$phar->compressAllFilesBZIP2();
$phar->commit();
?>
```



Phar

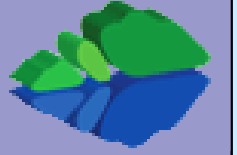
- ☑ Archive entries are accessible as streams
- ☑ Archive stub can contain PHP_Archive
- ☑ Archive can be given an alias name
- ☑ Entries can referenced by archive alias
- ☑ Entries can be compressed

```
<?php
Phar::mapPhar('myphar');
include 'phar://myphar/main.php';
__HALT_COMPILER();
?>

$> php myphar.phar
```



Solving return by reference

- ☑ Right noe ArrayAccess cannot return by reference
  1. Implement ArrayAccessByRef
     - ☑ Reference on return of `offsetGet` or in arg of `setOffset`
     - ☑ Actually the engine should be able to distinguish
       - ☑ &offsetGetRef()
       - ☑ offsetSetRef($offset, &$value)
  2. Allowing transparent return
     - ☑ Done for internal functions in PHP 6
     - ☑ Code only needs to be activated in 5
  3. Another solution, add proxies
     - ☑ `spl_member_proxy($object, $member)`
     - ☑ `spl_index_proxy($object, $index)`
     - ☑ Problems to solve: `unset` and `isset`/`empty`



At Last some Hints

- ☑ List of all SPL classes — PHP 5.0.0

```
php –r 'print_r(array_keys(spl_classes()));'
```

- ☑ Reflection of a built-in class — PHP 5.1.2

```
php --rc <Class>
```

- ☑ Reflection of a function or method — PHP 5.1.2

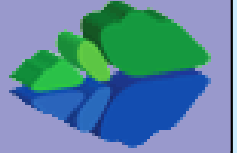
```
php --rf <Function>
```

- ☑ Reflection of a loaded extension — PHP 5.1.2

```
php --re <Extension>
```

- ☑ Extension information/configuration — PHP 5.2.2

```
php --ri <Extension>
```



THANK YOU

- ☑ This Presentation
  http://somabo.de/talks/
- ☑ SPL Documentation
  http://php.net/~helly
- ☑ SPL_Types
  http://pecl.php.net/package/spl_types
- ☑ Phar
  http://pecl.php.net/packages/phar
  http://php.net/phar