Kapitola 3

Objektově orientované programování

Do programovacího jazyka PHP přibylo velké množství objektově orientovaných funkcí už ve verzi PHP 4. Ve verzi PHP 5 se objektově orientované programování zásadně přepracovalo. Nyní máte k dispozici nové, přehlednější a rychlejší prostředí, které svou syntaxí nezapře podobnost s programovacím jazykem Java.

Pokud již nějaké zkušenosti s objektově orientovaným programováním máte, pak můžete bez obav přeskočit první kapitolu a začít studovat rovnou kapitolu 3.2, kde najdete několik příkladů pro využití objektově orientovaného prostředí v PHP 5.

Na konci této kapitoly pak naleznete vzorový příklad, na němž se ukáže objektově orientované programování (OOP) na základě jednoduchých struktur.

Úvod do objektově orientovaného programování

V polovině devadesátých let, kdy se stále většímu zájmu uživatelů těšil programovací jazyk Java, se stal v oboru informačních technologií pojem objektově orientované programování (OOP) doslova módou. Řešením všech problémů při programování mělo být právě objektově orientované programování.

Dnes stejně jako před lety potkáte řadu kritiků OOP. Jedním z bodů, v němž je OOP kritizováno, je fakt, že teorie se sice dobře poslouchá, ale praxe je často daleko složitější a řešení zdlouhavější.

Co je vlastně objektově orientované programování?

Zjednodušeně řečeno se u objektově orientovaného programování vše točí okolo objektů. Objekt je soubor dat a funkcí. Oproti strukturovanému programování je zde velký rozdíl v tom, že strukturované programování na rozdíl od objektově orientovaného od sebe přísně oddělovalo data a funkce.

Představme si například, že vyvíjíme program, který používá geometrické tvary: Vzhledem k tomu, že často používáme obdélník, vytvoříme třídu typu *Obdélník* (třída je zdrojový kód hotového objektu). Vaše třída *Obdélník* má vlastnosti *Šířka* a *Výška* (to jsou data objektu). Pro geometrické výpočty budete potřebovat ještě obsah a obvod

obdélníka. Proto se třída rozroste o dvě funkce – *getArea* a *getPerimeter*. Vaše třída tedy nyní obsahuje dvě proměnné, které představují data potřebná pro danou třídu, a dvě funkce, které z těchto dat počítají nové hodnoty. Každý nový objekt typu obdélníka, který vytvoříte, může vracet svůj obsah a obvod. Jedná se sice o silně zjednodušený příklad, ale základní myšlenku objektově orientované struktury snad ilustruje dobře.

OOP je natolik složité, že by si rozhodně zasloužilo více než dvě následující kapitoly. Proto když se budete chtít hlouběji poučit o OOP ve vztahu k PHP, pak vám doporučujeme najít si příslušnou literaturu zabývající se OOP ve vašem knihkupectví.

Pro koho je OOP?

Výhody OOP se projeví zejména při vývoji rozsáhlých projektů, tedy tam, kde víme, že bude nutno vytvořit velké množství zdrojového kódu. Na takových projektech většinou pracuje několik vývojářů. Vzhledem k tomu, že u OOP lze nastavit jasné rozhraní, je OOP ideálním prostředkem, jak danou aplikaci vyvíjet po částech.

Pokud použijete OOP, nebude s nejvyšší pravděpodobností váš kód ani kratší a ani méně složitý. Často je tomu právě naopak. Výhody spočívají spíše v lepší správě a údržbě a také ve snadné rozšiřitelnosti.

Přechod ze strukturovaného programování k OOP činí mnoha vývojářům značné potíže. Přece jen se jedná o úplně jinou záležitost. Pokud jste nikdy s OOP neměli co do činění, je možné, že ne všechno, o čem budeme v této kapitole psát, pochopíte hned napoprvé. To ale nevadí. Podívejte se na další příklady v této knize, neboť v nich neustále znovu používáme metody OOP. Věříme, že až budete tuto kapitolu číst podruhé, vše vám bude daleko jasnější.

Výhodou PHP je to, že můžete do OOP přejít velmi pohodlně. Začněte tím, že do své aplikace zabudujete několik funkcí z nějaké třídy. Možná si později všimnete, že tyto funkce potřebují ke své činnosti nějaká data – pak do zdrojového kódu třídy uložte také pro tato data příslušné proměnné.

OOP nebo strukturované programování?

PHP je ceněno také proto, že v něm jde vytvořit *všechno vcelku rychle*, a to ať se bavíme o rychlosti práce aplikace nebo o jejím vývoji. Rychlý vývoj aplikací často znamená malou strukturu a několik krátkých spolu nesouvisejících skriptů. To nemusí ale být vždy nevýhodou. Právě drobný skript v PHP, který splní vše, co od něj čekáme, je často tím nejideálnějším řešením.

Objektově orientované programování se naproti tomu pyšní jasnou strukturou, rozhraními a promyšleným konceptem. Jak tyto odlišné přístupy sjednotit? Na rozdíl od Javy se nemusíme v PHP do objektově orientovaného programování nijak nutit. Můžeme použít PHP 5 i starší verze a vyvíjet aplikace strukturované. Nejideálnější však je použít oba dva přístupy současně.

V PHP 5 byly nástroje pro OOP, které byly již v PHP 4, zcela předělány. Zatímco ve starších verzích PHP se používaly objekty jako primitivní typy (např. Integer nebo String), má Zend Engine II, který je součástí PHP 5, daleko promyšlenější implementaci. Nová verze umožňuje daleko jemnější správu přístupových práv uvnitř objektů a také daleko větší výkon.

UPOZORNĚNÍ: Vzorové příklady použité v kapitolách 11 a 14 používají jak strukturované programování, tak OOP. Pro připojení k databázi se většinou používá třída MyDb, která je orientovaná objektově. V kapitole 10.3 se dozvíte podrobnější informace o její struktuře.

Syntaxe pro objektově orientované programování v PHP 5

První třída

V úvodu vlastního programování by bylo hříchem, kdybychom zapomněli na vskutku již nepostradatelný příklad *Hello World*:

```
<?php    // Ukázkový soubor: helloworld.php
  class HelloWorld {
    public $who;
    function __construct($str="World") {
      $this->who = $str;   }
    function sayHello() {
      printf("Hello %s\n", $this->who);   }
  }
```

Tato velmi přehledně uspořádaná třída se může použít třeba takto:

```
<?php    // Ukázkový soubor: testhelloworld.php
  require 'helloworld.php';
  $hw = new HelloWorld();
  $hw->sayHello();        // Výsledek: Hello World
  $hu = new HelloWorld("Universe");
  $hu->sayHello();        // Výsledek: Hello Universe
```

Určitě jste si všimli, že v tomto případě ani žádnou třídu nepotřebujeme. Stejnou operaci totiž lze provést i pomocí jednoduché funkce.

Vytváření a rozšiřování tříd

Obecně

Zdrojový kód, popřípadě plán pro vytvoření objektu, se označuje jako *třída*. Zdrojový kód obsahuje všechny znaky třídy, jako jsou vlastnosti, metody atd. Třídy mohou existovat samostatně nebo mohou rozšiřovat další třídy. Když vytvoříte podtřídu, pak tato

zdědí všechny metody nadřazené třídy. Tuto vlastnost označujeme jako *dědičnost* (*inheritance*) a jedná se o jeden z hlavních pilířů objektově orientovaného programování.

Novou třídu vytvoříte prostřednictvím klíčového slova *class*. Je zvykem každou třídu ukládat do zvláštního souboru – ostatně to přispívá i k přehlednosti zdrojového kódu. Zdrojový kód třídy se uzavírá do složených závorek. Uvnitř třídy pak můžete definovat proměnné, metody i konstanty.

```
class Hello {
```

Chcete-li rozšířit nějakou existující třídu, pak použijte klíčové slovo *extend* hned za jménem třídy:

```
class HelloWorld extends Hello {
```

Všimněte si, že v PHP jednou rozšířenou třídu už znovu rozšířit nemůžete.

Konstruktory a destruktory

Konstruktor představuje metodu, která se automaticky volá při vytvoření třídy. Parametry, které se třídě při jejím vytvoření předají, jsou pak k dispozici uvnitř konstruktoru.

Od verze PHP 5 má funkce konstruktoru jednotné jméno, a to *__construct*. Ve starších verzích PHP se jako konstruktor používala metoda, která měla stejný název jako vytvářená třída.

```
class ZipArchive {
  function __construct($filename = "") {
    if ($filename != "" && is_file($filename)) {
 ...
```

Když vytvoříte instanci této třídy, pak můžete přímo zadat název souboru, který se má ve třídě použít.

```
$zip = new ZipArchive("sqldump.zip");
```

V odvozených třídách se konstruktor základní třídy automaticky nevolá. Zde je nutno jej zavolat pomocí konstrukce *parent*. Konstrukce *parent* vám tak poskytuje přístup k základní třídě.

```
class MyZipArchive extends ZipArchive {
  function __construct($filename="") {
    parent::__construct($filename);
  }
...
```

Podobně jako se při vytváření třídy volá konstruktor, volá se při odstraňování objektu tzv. destruktor. Název této funkce je příznačný – jde o *__destruct*. Destruktor však není nutno ve většině případů vůbec používat, protože PHP provádí správu paměti automaticky a doba běhu skriptu je velmi krátká, takže není nutno objekty odstraňovat ručně. Destruktor se tedy používá jen zřídka, nicméně může mít svůj smysl, pokud chceme provést úklid paměti cíleně (například při uzavření otevřeného souboru).

Vytvoření nové instance

Nový objekt neboli novou instanci třídy vytvoříte pomocí operátoru *new*.

```
$zip = new ZipArchive();
```

Příkaz vytvoří nový objekt typu *ZipArchive* a přiřadí jej proměnné *$zip*. Přes tuto proměnnou pak máte přístup ke všem metodám a proměnným třídy. (Metody jsou funkce třídy.)

Můžete samozřejmě vytvářet i více objektů jedné třídy, například *zip1*, *zip2* atd. Připomeňme si ještě jednou rozdíl mezi třídou a objektem: *ZipArchive* je obecná třída (základní projekt) a *zip*, *zip1* či *zip2* jsou konkrétní objekty této třídy.

UPOZORNĚNÍ: V PHP 5 se změnilo chování při přiřazování objektů. Zatímco ve starších verzích se vytvářela vždy kopie objektu a té se přiřazovala proměnná, v PHP 5 se nové objekty předávají ve formě odkazů. Podrobnější informace si přečtete v kapitole 3.2.7.

Zapouzdření: Přístup k proměnným a funkcím členů

Dalším z pilířů OOP je *zapouzdření* (*encapsulation*). Zapouzdření znamená to, že se při vytvoření zdrojového kódu nové třídy můžete rozhodnout, kdo bude mít k datům a funkcím této třídy přístup. *Zend Engine II* v PHP 5 používá pro přístup k proměnným a metodám popisovače známé z jiných objektově orientovaných programovacích jazyků: *public*, *protected* a *private* – často se používá zkratka PPP. Význam popisovačů PPP osvětluje následující tabulka.

Popisovač	Význam
public	Libovolný přístup
protected	Přistupovat smí pouze třída, která ji vytvořila, nebo z ní odvozená třída
private	Přistupovat smí pouze třída, která ji vytvořila

Členské proměnné třídy uchovávají data tohoto objektu. Pokud jsou ve třídě definovány, pak musí být před nimi uveden popisovač.

```
class ZipArchive {
  private   $zip;     // Přístup k souboru Zip
  protected $items;   // Počítadlo položek v souboru Zip
  public    $verbose; // Přepínač pro stavová hlášení
...
```

Porušení pravidel pro přístup vede ke kritické chybě (*Fatal Error*) a činnost programu se ukončí.

```
$za = new ZipArchive("sqldump.zip");
$za->verbose = TRUE;  // Proměnnou lze použít, je typu public
echo $za->items;      // Kritická chyba: k proměnné $items nelze přistupovat
```

Odkaz na aktuální objekt: this

Možná vás již napadlo, k čemu asi bude sloužit proměnná *$this*, kterou jsme použili již v několika příkladech. Její význam je poněkud zvláštní, protože představuje odkaz na aktuální třídu. *$this* použijete vždy, když budete chtít uvnitř nějaké metody přistupovat k jiné metodě nebo členské proměnné třídy.

Statické proměnné a metody

Statické proměnné a metody lze volat i mimo kontext objektu. Tak můžete do třídy uložit nějakou metodu nebo proměnnou, kterou budete chtít později použít, aniž byste museli vytvářet instanci třídy. Chcete-li nějakou proměnnou nebo metodu označit jako statickou, pak použijte za popisovačem PPP klíčové slovo *static*.

```
class MyStatic {
  public static $version = "0.1.9";
}
echo MyStatic::$version;
```

U statických funkcí není k dispozici odkaz na aktuální objekt pomocí klíčového slova *$this*, protože se uvažuje, že je metoda od třídy oddělena. Pokud přesto chcete přistoupit k aktuálnímu objektu, použijte popisovač *self::*.

```
class MyStatic {
  public static $version = "0.1.9";
  public static function printVer() {
    echo self::$version."\n";
  }
}
```

Konečné metody a třídy

Klíčovým slovem *final* se označují ty metody, které se nesmí přepsat žádnou z podtříd. Pokus o přepsání metody označené jako *final* vede ke *kritické chybě*. Tuto chybu vyvoláme například v následujícím kódu:

```
class MyFinal {
  final function printVer() {
    echo "Version: 0.1.9\n";
  }
}
class MySubFinal extends MyFinal {
  function printVer() {    // FATAL ERROR!
    echo "Version: 0.1.4\n";
  }
}
```

Jako *finální* můžete označit i celou třídu. Pak vyvolá kritickou chybu každý pokus o rozšíření třídy (*extend*).

Přetěžování proměnných a metod

Přetěžování proměnných

Klasickým znakem objektově orientovaných programů je využívání metod *set* a *get* pro všechna data třídy, které mají být k dispozici mimo třídu. Tyto metody můžete použít I v PHP, nicméně existuje daleko pohodlnější způsob, jak zmíněná data načíst.

Pokud potřebujete přistoupit na nějakou ještě nedefinovanou proměnnou, máte k dispozici dvě metody, které můžete vyvolat. Pro přiřazení je zde metoda *_set* a pro dotazování metoda *_get*. V objektově orientovaných jazycích nazýváme tuto schopnost přetěžování (*overloading*). Situaci nám osvětlí následující krátký příklad:

```
<?php    // Ukázkový soubor: MyGeometry.php
class MyGeometry {
  protected $my_rect = array("width" => 300, "height" => 200);
  function __get($key) {
    if (array_key_exists($key, $this->my_rect)) {
      return $this->my_rect[$key];
    } else {
      return FALSE;
    }
  }
}
$geom = new MyGeometry();
echo $geom->width."\n";
```

V něm vidíme, jak se řeší přístup na proměnnou *$geom->width*. Poté, co není ve třídě nalezena proměnná *$width*, se použije metoda *_get*. Tam se zkontroluje, zda požadovaná proměnná pochází jako prvek z pole *$my_rect*, a poté se vrátí příslušná hodnota. Pokud ve své třídě metodu *_get* nepoužijete a zavoláte nedeklarovanou proměnnou, zobrazí vám interpreter PHP varování typu *E_NOTICE*, nicméně program poběží dále.

Podobně se pro nastavení hodnoty použije metoda *_set*. Pokud metodu *_set* správně použijete, získáte plnou kontrolu nad tím, které proměnné se ve třídě vlastně nachází.

```
class SetGeom {
  protected $my_rect = array("width" => 300, "height" => 200);
  function __set($var, $val) {
    if (array_key_exists($var, $this->my_rect)) {
      $this->my_rect[$var] = $val;
    } else {
      exit("Tato proměná neexistuje v SetGeom\n");
    }
  }
}

$sg = new SetGeom();
$sg->width = 150;
$sg->widht = 99;
```

Pokud vaše třída nebude obsahovat žádnou metodu *_set* a přiřadíte hodnotu nějaké nedeklarované proměnné, pak se tato proměnná vytvoří. Takže pokud tak rádi děláte překlepy jako já, pak na posledním řádku výše uvedeného kódu bez metody *_set* vytvoříte novou proměnnou *$widht* (přehodili jsme znaky h a t). Pokud použijete příkaz *_set*, činnost programu se přeruší. Příkaz *exit* je možná poněkud drastický, ale vidíte, jak můžete reagovat na dotazy přicházející zvnějšku.

Přetěžování metod

Stejně jako pro nedeklarované proměnné existuje i metoda pro nedefinované metody. Pokud se zavolá metoda, která ve třídě neexistuje, použije se metoda *_call*.

```
  class Geometry {
    protected $my_rect = array("width" => 300, "height" => 100);
    function __call($name, $args){
      echo "Volali jste metodu $name \n";
    }
  }

$g = new Geometry();
echo $g->save();
echo "done\n";
```

Druhý parametr metody *_call* je pole, které obsahuje všechny parametry pro volanou funkci. Bez použití metody *_call* by příkaz *$g->save()* vedl ke *kritické chybě* a došlo by k předčasnému ukončení programu.

Pokud přetížení metod použijete správně, pak vám ušetří spoustu práce. Je zde však reálné nebezpečí, že když všechny požadavky na program vyřešíte přetěžováním metod, bude zdrojový kód aplikace nepřehledný.

Rozhraní

Rozhraní představují společné hranice mezi několika třídami. Když si chcete být jisti, že třída plní tu funkci, kterou jste jí přiřadili, pak jí musíte přiřadit i příslušné rozhraní. Toto rozhraní se definuje podobně jako třída, jenom se zde nachází pouze metody. Všechny metody jsou označeny jako public a nic neobsahují.

```
interface transformable {
  public function setX($x);
  public function setY($y);
}
```

Všechny třídy, které používají takové rozhraní, musí obsahovat metody *setX* a *setY*.

```
class Rectangle implements transformable {
  public $my_rect = array(
        "x" => 100, "y" => 200,
        "width" => 100, "height" => 150);
  function setX($x) {
    $this->my_rect["x"] = $x;
  }
```

```
  function setY($y) {
    $this->my_rect["y"] = $y;
  }
}
```

Nyní už jistě víte, že všechny třídy, které používají rozhraní *transformable*, mají k dispozici funkce *setX* a *setY*. Nikde není předepsáno, jak tyto metody mají vypadat. Výše uvedený příklad mění příslušným způsobem souřadnice členské proměnné *$my_rect*.

Ošetření chyb ve třídách

Jak jsme se již zmínili, používají se v objektově orientovaném programování *výjimky* k tomu, aby reagovaly na chyby při běhu programu. *Výjimky* umožňují předat chybu, která se objevila, volanému programu a současně zajistit, aby se při příštím spuštění programu tato chyba neobjevila.

```
<?php  // Ukázkový soubor: zipcreate.php
  class ZipCreate {
    public function createZipfile($filename) {
      if (is_file($filename)) {
        throw new Exception("CREATE: $filename již existuje\n");
      }
      // je možné případně provést několik testů:
      //  např. dostatek místa na disku, právo pro zápis atd.
      // další zdrojový kód pro vytvoření archivu ZIP ...
    }
    public function addFile($filename) {
      if (!is_file($filename)) {
        throw new Exception("ADD: $filename nenalezen\n");
      }
      // další zdrojový kód pro vložení souboru do archivu ZIP ...
    }
  }
```

Program, který výše uvedenou třídu používá, může vypadat třeba takto:

```
<?php    // Ukázkový soubor: testexception.php
  require "zipcreate.php";
  $z = new ZipCreate();
  try {
    $z->createZipfile("example.zip");
    $z->addFile("ex1.php");
  } catch (Exception $e) {
    echo "Failed in ZipCreate: ".$e->getMessage();
  }
```

Ještě lépe pak systém *výjimek* funguje tehdy, když vytvoříte podtřídy třídy PHP s názvem *Exception*. V již uvedeném příkladu by se mohlo jednat třeba o třídu *IoException*. Použití *výjimek* ukážeme v závěrečném příkladu v kapitole 3.3.

Tipy a triky

V této kapitole naleznete několik funkcí, které vám objektově orientované programování v PHP bezpochyby zpříjemní.

Automatické načítání souborů tříd

Pokud ve skriptech používáte vícero tříd a pokud tyto třídy ukládáte do zvláštního souboru, pak se vám ve zdrojovém kódu okamžitě objeví celá řada příkazů *require*. Příklady uvedené v této knize tento problém obchází tím, že používají jeden soubor *include*, který připojí všechny ostatní třídy. U velkých projektů tento princip nedoporučujeme, neboť ne vždy potřebujeme všechny třídy v jednom souboru.

PHP 5 díky funkci *_autoload* umožňuje v případě potřeby načíst potřebné třídy. Tato funkce se volá automaticky vždy, když se pokusíte použít nějakou funkci, jejíž kód se ještě nenačetl. Do funkce můžete také zadat všemožné cesty k jiným vašim knihovnám.

Následující příklad ukazuje použití funkce *_autoload*. Všeho všudy se použijí tři soubory, přičemž obě třídy (`rectangle.php` a `circle.php`) jsou uloženy v podsložce `inc`.

```
testgeom.php
inc/rectangle.php
inc/circle.php
```

Třída *Circle* je naprogramována následovně:

```
<?php    // Ukázkový soubor: inc/circle.php
  class Circle {
    public $radius = 0;
    public function __construct($r) {
      $this->radius = $r;
    }
    public function getArea() {
      return $this->radius*$this->radius * pi();
      return pow($this->radius,2) * pi();
    }
    public function getPerimeter() {
      return 2 * pi() * $this->radius;
    }
  }
```

Třída *Rectangle* je naprogramována podobně, a proto ji zde ani neuvádíme celou.

```
<?php    // Ukázkový soubor: inc/rectangle.php
  class Rectangle {
    protected $width = 0;
    protected $height = 0;
    public function __construct($w,$h) {
      $this->width = $w;
      $this->height = $h;
    }
    public function getArea() {
...
```

Obě třídy používá program `testgeom.php`. Používáme zde funkci *_autoload*.

```
<?php    // Ukázkový soubor: testgeom.php
  function __autoload($class_name) {
    require strtolower("inc/$class_name.php");
  }

  $circ = new Circle(300);
  $rect = new Rectangle(300,400);
  echo $rect->getPerimeter()."\n";
  echo $circ->getArea()."\n";
```

Třídy *Circle* i *Rectangle* lze použít, aniž by se musely ručně načítat příkazy *require* nebo *include*. Funkce *_autoload* tuto operaci zařídí pro všechny třídy ve složce inc. Další příklad použití funkce *_autoload* naleznete v kapitole 3.3.

UPOZORNĚNÍ: V Linuxu/Unixu hraje velkou roli používání malých a velkých písmen. Název třídy je nutno zapisovat přesně tak, jak vypadá název souboru, jinak funkce *require* soubor nenajde. Když všechny názvy souborů píšete malými písmeny, pak můžete podobně jako v minulém příkladu parametr pro funkci *require* převést na malá písmena příkazem *strtolower*.

Cyklus přes všechny členské proměnné objektu

Jistým způsobem je možné používat objekty jako pole. Pomocí konstrukce *foreach* můžete naprogramovat cyklus přes všechny členské proměnné daného objektu.

```
<?php    // Ukázkový soubor: loop.php
  class Loop {
    public $pu = "public";
    protected $po = "protected";
    function printMembers() {
      foreach ($this as $member) {
        print "V třídě: $member\n";
      }
    }
  }
  $loop = new Loop();
  $loop->printMembers();    // zobrazí obě proměnné
  foreach ($loop as $l) {   // zobrazí pouze proměnnou $pu
    print "Outside: $l\n";
  }
```

Metoda *printMembers* uvnitř třídy umožňuje přistupovat ke všem členským proměnným. Cyklu mimo třídu vidí pouze členy označené jako *public*. Všimněte si také použití proměnné *$this* uvnitř funkce *printMembers*.

Příkaz print a objekty

Pokud na objekt (ne však na nějakou členskou proměnnou) použijete příkaz *print*, pak se vám jako výsledek zobrazí řetězec *Object* a vnitřní ID objektu.

```
<?php    //  Ukázkový soubor: tostring.php
  class StringTest {
    public $text = "stringtest";
  }
  $st = new StringTest;
  print $st;           // Výsledek: Object id #1
```

Většinou to ale není to, co byste se chtěli o objektu dozvědět. Zvláště při vývoji aplikací se složitými třídami by se hodilo zobrazit krátkou informaci o třídě. Metoda *_toString* to zařídí. Pokud zavoláte objekt příkazem print nebo *echo* a pokud má objekt metodu *_toString*, pak se zobrazí návratová hodnota této metody.

```
  class StringTest {
    public $text = "stringtest";
    function __tostring() {
      return $this->text."\n";
    }
  }
  $st = new StringTest;
  print $st;           // Výsledek: stringtest
```

Klonování objektů

Poněkud citlivé téma klonování v této knize můžeme probrat bez problémů, protože se naštěstí zabýváme pouze nulami a jedničkami. Jak jsme se již zmínili v kapitole 3.2.2, změnilo se v PHP 5 chování ohledně přístupu k objektům. Přiřazení se nyní provádí pouze odkazem na objekt, a nikoliv kopírováním hodnoty.

Následující příklad nám objasní rozdíl mezi kopírováním hodnoty a odkazováním. PHP se standardně chová tak, že při obvyklém přiřazování proměnné vždy vytvoří kopii.

```
$a = 5;
$b = $a;
$b++;
printf("a:%d | b:%d\n", $a,$b);  // Výsledek: a:5 | b:6
```

Při přiřazení *$b=$a* se vytvoří kopie *$a*. Obsah proměnných *$a* a *$b* se pak nachází v různých oblastech operační paměti.

Co se týče objektů, tam je tomu v PHP úplně jinak. Předá se totiž pouze odkaz (programátoři v jazyku C používají výraz ukazatel):

```
class MyRef {
  public $member = 0;
}
$mr = new MyRef();
printf("mr->member: %d\n", $mr->member);
```

```
$mr2 = $mr;    // Předání odkazem
$mr2->member++;
printf("mr->member: %d, mr2->member: %d\n",
    $mr->member, $mr2->member);
// Výsledek: mr->member: 1, mr2->member: 1
```

Příklad ukazuje, že se zvýšení členské proměnné v *$mr2* dotkne obou proměnných – *$mr* i *$mr2*. Jde o to, že obě objektové proměnné odkazují na jeden a ten samý objekt.

Pomocí metody *clone* můžete vytvořit kopii objektu. Pokud je ve třídě, kterou klonujete, k dispozici metoda *_clone*, pak se příkazy této metody po klonování použijí i na nový objekt.

Výše uvedený příklad tak můžete rozšířit o funkci *clone*:

```
$mr3 = clone($mr);
$mr3->member++;
printf("mr: %d, mr2: %d, mr3: %d\n",
    $mr->member, $mr2->member, $mr3->member);
// Výsledek: mr: 1, mr2: 1, mr3: 2
```

Klonovaný objekt pak už původní třídy nijak neovlivní.

Příklad: Systém pro rezervaci pokojů a zařízení

Problém s rezervacemi se objevuje už u středně velkých firem: je zasedací místnost dnes od 15 hodin volná, nebo se tam koná nějaká schůze? Když firma umožňuje půjčování firemního zařízení svým zaměstnancům, záhy se vyskytne požadavek na systém pro správu výpůjček. U velkých podniků nebo třeba univerzit jsou logistické problémy bez promyšleného plánování a správy takřka neřešitelné.

V následujícím příkladu bude chybět jedna důležitá součást, pro kterou nasazení v praxi víceméně nebude ani možné, a to současná práce více uživatelů. Nepředpokládáme, že by více uživatelů z různých pracovišť mohlo upravovat stejná data. Tento nedostatek vychází z toho, že vámi zadaná data se budou ukládat pouze do vaší *relace*. Pokud by mělo aplikaci používat více uživatelů, bylo by nutno data vložit do databáze – to však předpokládá znalosti, kterým se budeme věnovat až v dalších kapitolách knihy. Přes to všechno příklad určitě svoji roli splní, protože vám objasní představu objektově orientované aplikace.

Výhodou ukládání dat do jedné *relace* je to, že metody pro ukládání a dotazování objektů budou velmi jednoduché. V prostředí *relace* se vytvoří nové pole, v němž vytvoříme nové objekty. Nevýhodou je, že se všechna data ztratí, jakmile zavřeme okno prohlížeče.

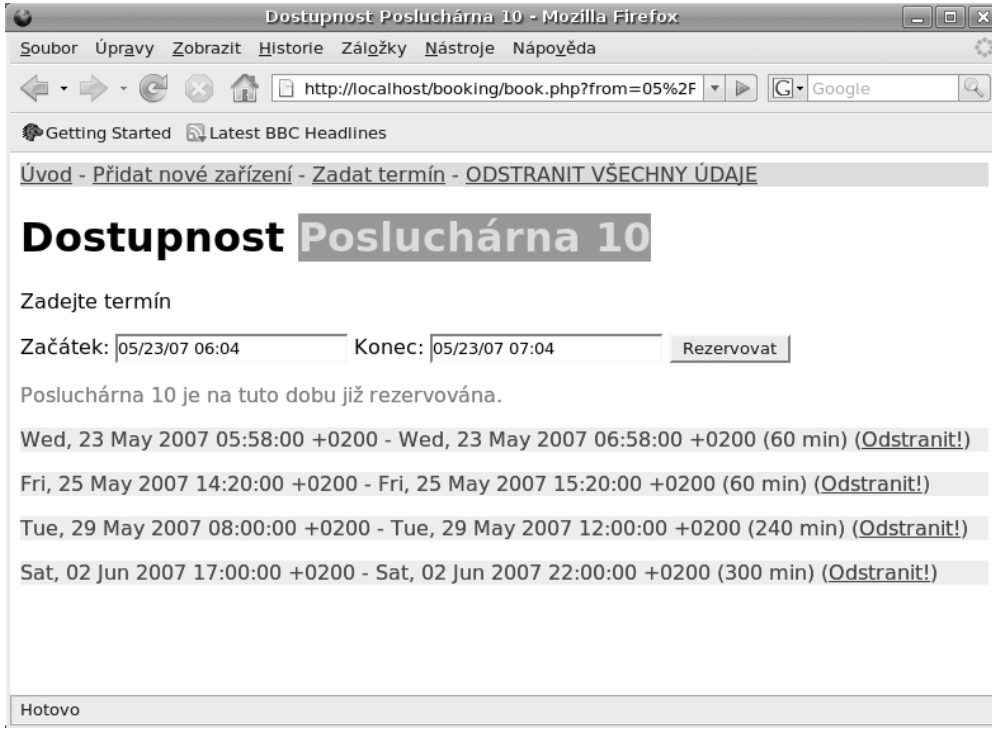
UPOZORNĚNÍ: Podrobnější informace o *relacích* naleznete v kapitole 5.4.

Pro rezervační systémy podobného druhu existují již hotová řešení vytvořená pro PHP a MySQL. Přehledná a stabilní je například aplikace na adrese http://mrbs.sourceforge.net.

Návod k použití

Nejprve musíte vytvořit zařízení, která se budou moci rezervovat. V naší ukázkové aplikaci se bude k rezervaci nabízet pět různých zařízení: posluchárna, učebna, beamer, digitální fotoaparát a flipchart. Tyto objekty jsou každý úplně jiný, tak například u posluchárny můžete zadat počet posluchačů, zatímco u beameru se bude hodit zadávat například rozlišení.

Po vložení jednoho nebo více zařízení můžete začít s rezervováním termínů. Stačí vybrat zobrazení dostupnosti požadovaného zařízení nebo místnosti a zadat čas, po který si chcete zařízení rezervovat. Pokud se při rezervaci bude váš termín překrývat s termínem někoho jiného, pak vás na to aplikace upozorní. Na obrázku 3.1 vidíte dostupnost posluchárny a také, co se stane, pokud se pokusíte zarezervovat již obsazený termín.

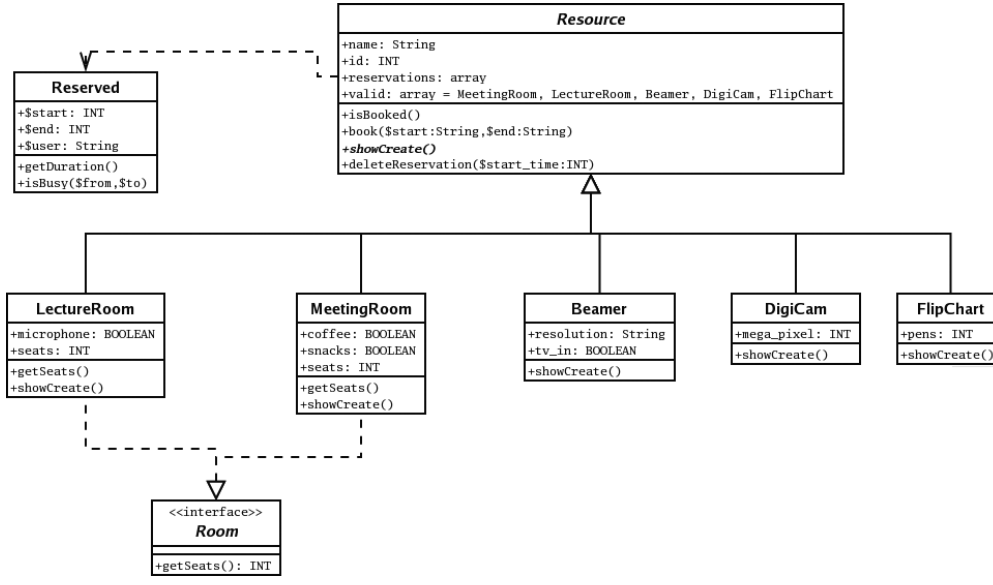


Obrázek 3.1: Dostupnost posluchárny v rezervačním systému

Po uzavření internetového prohlížeče se relace včetně všech zadaných dat ztratí. Data však můžete rovněž sami odstranit, a to klepnutím na odkaz ODSTRANIT VŠECHNY ÚDAJE. Tím se všechny proměnné *relace* odstraní a zároveň dojde k přesměrování na úvodní stránku.

Struktura tříd

Obrázek 3.2 ukazuje strukturu tříd ve formě schématu.



Obrázek 3.2: Třídy použité v ukázkovém příkladu

Základní třída: Resource

Všechny třídy, které představují jednotlivá zařízení, jsou odvozeny od základní třídy *Resource*. Tato třída disponuje všemi funkcemi, které se využijí u jednotlivých zařízení. Kromě toho tato třída obsahuje abstraktní funkci *showCreate*. Tuto funkci musí každá podtřída přepsat. Třídy, které obsahují jednu nebo více abstraktních funkcí, se deklarují klíčovými slovy *abstract class*. Abstraktní třídy si můžete představit, jako by to byly *Interfaces*. Nelze od nich vytvořit žádnou instanci a předem určují ty funkce, které se musí přepsat. Velkým rozdílem oproti *Interfaces* je to, že abstraktní třídy mohou velmi dobře obsahovat členské proměnné, ale nikoliv abstraktní funkce.

Ve zmiňované základní třídě jsou ve formě pole dvě zajímavé členské proměnné – *$reservations*, kde se ukládají všechny zadané termíny, a *$valid*, která obsahuje seznam platných podtříd.

```
abstract class Resource {

    public $id;
    public $name;
    public $reservations = array();
    // existující podtřídy
    public static $valid = array(
      "MeetingRoom" => "Posluchárna",
      "LectureRoom" => "Učebna",
      "Beamer"      => "Beamer",
      "DigiCam"     => "Digitální fotoaparát",
      "FlipChart"   => "Flipchart"
    );

    // zobrazí formulář HTML pro vytvoření nové třídy
    // každá podtřída musí tuto funkci přepsat
    public abstract function showCreate();
```

Po výše uvedených deklaracích následuje funkce pro rezervování časového úseku (book). Tato funkce vyžaduje zadání začátku a konce rezervace zařízení. Zadané časy se pomocí funkce *strtotime* převedou do *časového razítka Unixu* a odpovídají tak velmi flexibilním, i když bohužel anglickým formátům pro datum a čas. Pro jednoduché výpočty dat a časových razítek je velmi rozumné, že se jedná o celá čísla, pro něž se snadno dají použít běžné matematické funkce (>, <, = =, ...).

Pokud jeden z časových údajů nezadáte, způsobíte chybu při běhu programu, nebo spíše vyvoláte *výjimku*. Typ chyby je *DateFormatException*, což je podtřída třídy *Exception*, o níž budeme hovořit později. Běh funkce se po vzniku výjimky přeruší.

Když nelze formát proměnné rozpoznat jako platné datum, pak funkce převodu řetězcové proměnné na časové razítko pomocí příkazu *strtotime* vrátí -1. Opět se vyvolá výjimka *DateFormatException*. Pokud jsou k dispozici dva platné časové údaje, pak se tyto předají novému objektu typu *Reserved* společně s IP-adresou počítače, z něhož se program spouští (*$_SERVER[“REMOTE_ADDR“]*). Tato třída v konstruktoru testuje, zda je časový interval platný. Pokud ne, pak se ve třídě *Reserved* vyvolá *výjimka*, která se předá volané funkci a rovněž přeruší činnost aplikace.

```
    public function book($s, $e) {
      if ($s == '' || $e == '')
        throw new DateFormatException("Prázdné datum.");
      $start = strtotime($s);
      $end = strtotime($e);
      $user = $_SERVER["REMOTE_ADDR"];
      if ($start == -1 || $end == -1)
        throw new DateFormatException("Chyba při převodu časového údaje");

      $res = new Reserved($start, $end, $user);
      if ($this->isBooked($start,$end))
        throw new Exception("{$this->name je v tuto chvíli".
```

```
        "již rezervováno.");
    $this->reservations[$start] = $res;
  }
```

Před přidáním nového objektu typu *Reserved* k existujícímu objektu se samozřejmě ještě musí prověřit, zda není daný časový úsek již rezervován. Použije se k tomu funkce *isBooked*.

```
public function isBooked($start, $end=NULL) {
  // pokud se spuštění koná pouze na základě zadaného času,
  // bude čas ukončení nastaven na jednu sekundu po zahájení akce
  if ($end == NULL) {
    $end = $start + 1;
  }
  if (count($this->reservations) > 0) {
    foreach ($this->reservations as $r) {
      if ($r->isBusy($start, $end)) {
        return TRUE;
      }
    }
  }
  return FALSE;   // nebyla nalezena žádná rezervovaná zařízení
}
```

Tato metoda podporuje *přetěžování*, tj. v tomto případě to znamená, že ji lze zavolat prostřednictvím jednoho nebo dvou parametrů (časem nebo časovým intervalem). Vlastní práci vykoná cyklus, který běží přes všechna existující zařízení (*$this->reservations*). Pro každé zarezervované zařízení se prověřuje, zda není náhodou termín již zarezervovaný (*$r->busy*). Pokud se časy překrývají, vrátí funkce hodnotu *TRUE*. Pokud je zadaný časový interval volný, vrátí funkce hodnotu *FALSE* (funkci třídy *Reserved* objasníme hned za touto kapitolkou).

Kromě pomocných funkcí k zobrazování HTML obsahuje třída *Resource* ještě funkce pro ukládání a volání objektů. Již jsme zmiňovali v úvodu, že se objekty ukládají do prostředí *relace*. (Pokud chcete data uložit trvale, musíte to nastavit u těchto funkcí. Namísto do *relace* můžete data ukládat do souboru nebo do databáze. Přístup k souboru je většinou problematický, protože musíte neustále dávat pozor na to, aby nedocházelo zároveň ke čtení a k zápisu do souboru. Databáze jsou v tomto případě daleko lepším řešením, protože tuto kontrolu provedou za vás.)

Aktuální objekt (*$this*) v poli *Session $_SESSION['resource']*. Pro nové vyhledání objektu se použije jako index prvku pole ID daného objektu.

```
public function save() {
  $_SESSION['resource'][$this->id] = $this;
  return TRUE;
}
public static function clearAll() {
  unset($_SESSION['resource']);
  return TRUE;
}
public static function getAll() {
```

```
    if (array_key_exists('resource', $_SESSION)) {
      return $_SESSION['resource'];
    } else
      return FALSE;
  }
  // Funkce relace: vrací objekt přesně zpět
  public static function getById($id) {
    if (array_key_exists($id, $_SESSION['resource'])) {
      return $_SESSION['resource'][$id];
    } else {
      throw new Exception("Id not found");
    }
  }
```

Funkce *clearAll*, *getAll* a *getById* jsou deklarovány jako statické. Znamená to, že pro jejich zavolání nepotřebujete žádnou instanci třídy, což je velká výhoda těchto funkcí. Funkce *getById* testuje, zda existuje požadované ID proměnné relace. Pokud tomu tak není, vyvolá se ve spuštěné funkci událost výjimky.

Třída pro jednotlivé časové údaje: Reserved.php

Třída *Reserved* potřebuje v konstruktoru čas začátku a konce rezervace a ještě uživatelské jméno. Každý objekt představuje jednu rezervovanou položku.

```
class Reserved {
  public $start;
  public $end;
  public $user;
  // požaduje zadání dvou časových údajů a řetězce obsahujícího jméno
 // uživatele
  public function __construct($start, $end, $user) {
    if ($start > $end) {
      throw new Exception('Čas začátku rezervace');
    }
    $this->start = $start;
    $this->end   = $end;
    $this->user  = $user;

   if ($this->getDuration() > 3600*24*14) {
      throw new Exception('žádné rezervace během 14 dní');
    }
  }
```

Konstruktor prověřuje dvě věci: za prvé, zda čas zahájení rezervace nenastane později než čas ukončení rezervace (pak by se jednalo o nesmyslný časový interval), a za druhé, zda není doba rezervace delší než 14 dní (3 600*24*14 sekund). Naposledy zmíněná podmínka má zabránit příliš dlouhým rezervacím jednotlivých zařízení. V obou případech se vyvolá výjimka a objekt se nevytvoří.

```
  public function getDuration() {
    return $this->end - $this->start;
  }
```

```
// vrací hodnotu TRUE, pokud je již objekt
// zarezervován mezi proměnnými $from a $to
    public function isBusy($from, $to) {
      if (($this->start <= $from && $this->end >= $from) ||
          ($this->start <= $to   && $this->end >= $to)   ||
          ($from <= $this->start && $to >= $this->end)) {
        return TRUE;
      } else {
        return FALSE;
      }
    }
```

Funkce *isBusy* se volá se dvěma časovými údaji a testuje, zda dotazované časy nekolidují s časem objektu. Dále se testuje, zda čas zahájení a konce rezervace leží v intervalu objektu. Třetí řádek v podmínce *if* nakonec kontroluje, zda požadovaný časový interval začíná před začátkem rezervace objektu a poté trvá až do konce.

Podtřídy

Všechny podtřídy třídy *Resource* musí definovat metodu *showCreate*. Do této metody se zadávají všechny vlastnosti daného zařízení. Ve třídě Beamer se například ukládá rozlišení zařízení a také to, zda jej lze připojit k televizi. Do základní třídy *Resource* se implementují také funkce *showTextFeature* a *showBoolFeature* a v podstatě odpovídají za formátování HTML. Aby bylo zařízení i tam k dispozici, předává konstruktor základní třídě i ID zařízení.

```
class Beamer extends Resource {
  public $resolution = '';
  public $tv_in = true;
  public function __construct($id) {
    parent::__construct($id);
  }
  public function showCreate() {
    echo "<table border='1'>\n";
    $this->showTextFeature('Název', 'name');
    $this->showTextFeature('Rozlišení', 'resolution');
    $this->showBoolFeature('Připojení k TV', 'tv_in');
    echo "</table>\n";
    echo "<input type='hidden' name='object' ".
      "value='Beamer'>\n";
  }
}
```

Třídy pro digitální fotoaparát a flipchart pracují v podstatě stejně jako výše uvedená třída Beamer. Rozdíl spočívá pouze ve správě místností, tedy u tříd *LectureRoom* a *MeetingRoom*. Ty mají další společnou vlastnost, a to počet míst. **Rozhraní** *Room* zajišťuje, aby všechny podtřídy pro místnosti měly k dispozici funkci pro počet míst. Všechny třídy, které používají rozhraní *Room*, musí obsahovat funkci *getSeats*. Třída *LectureRoom* pak má kupříkladu následující kód:

```
class LectureRoom extends Resource implements Room {
  public $microphone = FALSE;
  public $seats = 0;
  ...
  public function getSeats() {
    return $this->seats;
  }
```

Ošetření chyb: DateFormatException

Pokud se chybně zadá nějaký časový údaj, pak se vyvolá výjimka typu *DateFormatException*. Ta je odvozena od systémové třídy *Exception* a provede naformátování chybového hlášení pomocí příkazů pro šablony stylů a mimo jiné zobrazí odkaz na dokumentaci obsahující práci s formáty datumu.

```
class DateFormatException extends Exception {
  public function __construct($msg) {
    parent::__construct($msg);
  }
  // zobrazí odkaz k dokumentaci obsahující práci s formáty data
public function showHelp() {
    echo "<p class='hint'><a ".
      "href='http://www.gnu.org/software/tar/manual/".
      "html_chapter/tar_7.html'>Seznam</a> ".
      "podporovaných formátů data.</p>\n";
  }
  public function __toString() {
    return "<p class='error'>".$this->getMessage().
      " (".__CLASS__.").</p>\n";
  }
```

Díky funkci *_toString* lze poté naformátované chybové hlášení zobrazit příkazem *echo $e*.

Použití tříd

Běh aplikace je řízen třemi skripty: `index.php`, `book.php` a `add.php`, přičemž skript `add.php` slouží pouze k přidávání nového zařízení. Všechny skripty na prvním řádku načítají pomocný soubor `inc/main.php`. Kromě funkcí pro vytvoření hlavičky HTML (*html_start*) a konce souboru HTML (*html_end*) se zde spouští vlastní relace. Funkce PHP 5 s názvem *_autoload* zajišťuje příkazem *require_once* načtení všech požadovaných tříd. Když do systému přidáte novou třídu, kterou příslušným způsobem pojmenujete (název třídy obsahující malá písmena), pak ji funkce *_autoload* nalezne automaticky.

```
<?php    // Ukázkový soubor:  kap03/booking/inc/main.php
  session_start();
  function __autoload($class_name) {
    require_once(strtolower(dirname(__FILE__)."/$class_name.php"));
  }
```

Vzhledem k tomu, že soubory všech tříd jsou psány malými písmeny, použije se při zavolání příkazu *require_once* řetězec s parametrem *strtolower*. Díky příkazu *dirname(_FILE_)* se zajistí nastavení správné cesty k jednotlivým třídám.

UPOZORNĚNÍ: Funkce strtolower pracuje v závislosti na použitých nastavení *Locale*. Pokud použijete v názvu souboru znaky s diakritikou (což mimochodem vůbec není dobrý nápad) a zapomenete odpovídajícím způsobem nastavit Locale, pak funkce *strtolower* tyto znaky nepřevede.

Přidávání nových zařízení: add.php

Soubor `add.php` pracuje ve třech krocích: nejprve zobrazí seznam všech typů zařízení neboli všechny podtřídy třídy *Resource*. Po výběru typu se v dalším kroku zobrazí formulář pro zadávání dat pro nový objekt (funkce *showCreate* příslušné třídy). Nakonec se vytvoří nový objekt.

```
<select name="resource">
<?
  // Krok 1: Zobrazení všech dostupných tříd
  foreach (Resource::$valid as $k=>$v) {
    echo "<option value='$k'>$v</option>\n";
  }
?>
```

Ze seznamu resource se do prvního formuláře přiřadí název třídy (jedná se o prvek pole *$valid* ve třídě *Resource*, např. ve třídě *LectureRoom*). V dalším kroku se z toho názvu třídy vytvoří nový objekt (*$r*). Díky tomuto drobnému triku se může použít pro vyvolání funkce *showCreate* fiktivní objekt s ID 0.

```
  // Krok 2: Formulář pro vytvoření nové třídy
  if ($send != '') {   // testování, ze kterého formuláře
                       // požadavek přichází
    $r = new $res(0);
    html_start("Add a new resource");
    echo "<h2>Přidání nového zařízení: $res</h2>\n";
    printf("<form method='GET' action='%s'>\n",$_SERVER["PHP_SELF"]);
    $r->showCreate();
    echo "<input type='submit' name='add' value='Přidat'>\n";
    html_end();
    exit();
  }
```

Po novém odeslání stránky se vytvoří *skutečný* objekt (*$o*). Jeho ID bude odpovídat aktuálnímu časovému razítku z Unixu (nelze tedy v jedné sekundě současně vytvořit dva objekty). Všechny údaje předané z formuláře (*$_GET*) se do nového objektu uloží jako členské proměnné (*$o->$k=$v)*.

```
// Krok 3: vytvoření nového objektu
if ($add != '') {
  $o = new $object(time());
  foreach($_GET as $k=>$v) {
    $o->$k = $v;
  }
```

Pokud název nového zařízení nezadáte (*$o->name = = ''*), pak se jako název použije ID objektu. Vzhledem k tomu, že se při ukládání (*$o->save*) vyvolá výjimka, je tento příkaz ošetřen blokem *try-catch*. V odstavci *catch* se v případě potřeby zajistí zobrazení příslušného varovného hlášení.

```
  try {
    if ($o->name == '')
      $o->name = $o->id;
    $o->save();
    html_start("Přidání nového zařízení");
    echo "<h2>Přidání nového zařízení: ID: ".
      "{$o->id}</h2>\n";
    echo "<a href='book.php?id={$o->id}'>Rezervovat ".
      "termín</a>.\n";
    html_end();
    exit();
  } catch (Exception $e) {
    html_start("Chyba při přidání zařízení");
    echo "<h2>Vyskytla se chyba: ".
      $e->getMessage()."</h2>\n";
  }
}
```

Po úspěšném vytvoření nového objektu se zobrazí odkaz pro rezervování termínu.

Úvodní stránka: index.php

Úvodní stránku našeho příkladu představuje soubor `index.php` (viz obrázek 3.3).

```
<?php    // Ukázkový soubor: kap02/booking/index.php
  require_once(dirname(__FILE__)."/inc/main.php");
  $elements = Resource::getAll();
  if ($elements == FALSE) {
    header("Location: add.php");
    exit();
  }
  html_start("Rezervační systém");
```

Pokud funkce *Resource::getAll()* vrátí hodnotu *False*, pak nejsou k dispozici žádná zařízení a pomocí funkce *header* se poté provede přesměrování na stránku `add.php` pro zadání nového zařízení. V opačném případě se pomocí cyklu *foreach* zobrazí seznam všech dostupných zařízení (porovnejte s obrázkem 3.3).

Obrázek 3.3: Úvodní stránka aplikace s možností výběru různých zařízení

```
<?php
  foreach($elements as $r) {
    echo "<option value='{$r->id}'>{$r->name}";
    if ($r instanceof Room) {
      printf(" (%s míst)", $r->getSeats());
    }
    echo "</option>\n";
  }
?>
```

Na tomto cyklu je zajímavý operátor *instanceof*. Ten provádí implementaci třídy rozhraní *Room* a je tedy automaticky instancí třídy *Room*. Tak může u každé místnosti zobrazit, kolik míst je k dispozici.

Rezervace zařízení: book.php

Nejzajímavější pro uživatele zřejmě bude rezervace zařízení. Skript `book.php` je nutno volat pomocí ID (*$id*), protože jinak by došlo k přesměrování na úvodní stránku.

```
if ($id == '' || $id == FALSE) {
   header("Location: index.php");
 }
 try {
   $r = Resource::getById($id);
 } catch (Exception $e) {
   html_start();
   echo "<p class='error'>Nelze najít ".
     "požadované zařízení\n";
   html_end();
   exit();
 }
```

Požadovaný objekt zajistí statická metoda *getById* ze třídy *Resource*. Pokud se ID nepodaří najít, vyvolá se výjimka, na kterou program zareaguje zobrazením příslušného chybového hlášení. Poté se ze třídy Resource zavolá pomocná funkce, která zobrazí zadávací políčka pro začátek a konec termínu rezervace (*$r->showAddReservation*). Po odeslání formuláře pro rezervaci (proměnná *$reserve* obsahuje nějakou hodnotu), pak se systém pokusí pro zadaný časový interval zařízení rezervovat. V opačném případě se celý blok podmínky *if* přeskočí a funkce *$r->showBooked* zobrazí seznam všech rezervovaných termínů.

```
$r->showAddReservation();
if ($reserve != '') {
  $from = array_item($_GET, 'from');
  $to = array_item($_GET, 'to');
  try {
    $r->book($from, $to);
    $r->save();
  } catch (DateFormatException $e) {
    echo $e;
    $e->showHelp();
  } catch (Exception $e) {
    printf("<p class='error'>%s</p>\n", $e->getMessage());
  }
}
$r->showBooked();
```

Vlastní rezervaci provádí funkce *$r->book* ze třídy *Resource*. Ta dokáže, jak jsme již zmínili, vyvolat různé výjimky v závislosti na chybě, která nastala (problém s datem či problém s již rezervovaným zařízením). V souboru `book.php` se chyby vzniklé při převodu data (*DateFormatException*) ošetřují zvlášť, neboť se u nich navíc zobrazuje odkaz přinášející informace o správném zápisu formátu data (*$e->showHelp*). Všechny ostatní výjimky se zachycují v dalším cyklu.

V aplikaci nalezneme ještě dva drobné skripty: `del.php` a `reset.php`. Skript reset je velmi jednoduchý: volá totiž statickou funkci *clearAll* ze třídy *Resource* a provádí přesměrování uživatele zpět na úvodní stránku. Skript `del.php` odstraňuje položku ze seznamu rezervovaných termínů. I k tomu používá příslušnou funkci ve třídě *Resource* (*deleteReservation*).

UPOZORNĚNÍ: Tento příklad slouží skutečně pouze k tomu, aby vám osvětlil některé základní principy objektově orientovaného programování. V této podobě však není schopen prakticky žádného v praxi použitelného uplatnění. I proto jsme se například vůbec nezabývali kontrolou zadaných údajů, aby je nějaký útočník nemohl svévolně změnit. Zadávání údajů do textových políček a proměnné GET používané v otevřeném formuláři je velmi citlivou záležitostí a musí být dobře zabezpečeno. Těmto tématům se však budeme dostatečně věnovat v následujících příkladech a kapitolách (například v kapitole 5.5).