Dlaczego programowanie obiektowe cieszy się dużym powodzeniem wśród programistów? Odpowiedź jest prosta: ponieważ programowanie obiektowe pozwala stworzyć model rzeczywistości podzielonej na poszczególne elementy (obiekty) oddziałujące wzajemnie na siebie. Brzmi to trochę dziwnie, ale jest w tym sporo prawdy. Może nie jest to na pierwszy rzut oka widoczne, ale czasem o wiele łatwiej jest patrzeć na dany problem, rozumiejąc go w sensie tychże oddziałujących na siebie obiektów, niż rozbić go na zmienne i funkcje.

Dlaczego zajmujemy się programowaniem obiektowym w PHP? Przecież wiedza, którą już zdobyliśmy, wystarczy, aby rozumieć zasadę generowania stron WWW w sposób dynamiczny po stronie serwera. A jednak… Gra nie byłaby warta świeczki, gdyby nie fakt, że PHP w wersji 5. wyposażony został w obiektowość z prawdziwego zdarzenia. Kolejny argument „za" to właśnie obiektowe podejście do budowy serwisu WWW. Dotychczas (i dla celów poznawczych będziemy się tego trzymali) wszystkie skrypty PHP, które pisaliśmy, miały budowę liniową, czyli wykonywane były od początku do końca, z ewentualnymi pętlami, rozgałęzieniami i wywołaniami funkcji. Można jednak — wzorując się na ASP.NET czy JSP — zbudować serwis całkowicie obiektowo. Niestety, PHP 5 nawet nie daje nam wyboru sposobu budowy serwisu — oznacza to, że jeżeli będziemy chcieli, możemy sobie zaimplementować serwis jako całkowicie obiektowy, jednak silnik PHP oraz większość rozszerzeń pozostaje proceduralna (API bazuje na zestawie funkcji).

I znów wstępniak, który może wywołać dreszcze — pojawiły się terminy takie jak: programowanie obiektowe, obiektowość, obiekt… Znaczenie tych terminów zostało wyjaśnione w tym właśnie rozdziale. Poza tym treść rozdziału ma być wprowadzeniem w programowanie obiektowe w języku PHP — dzięki niej czytelnik powinien nauczyć się składni obiektowej PHP oraz poznać podstawy programowania zorientowanego na definiowanie klas i współpracę między obiektami.

Należy pamiętać, że programowanie obiektowe w PHP 5 to opcja — nie trzeba tworzyć klas i obiektów, żeby serwis WWW był w pełni funkcjonalny — jednak warto poświęcić zagadnieniu obiektowości trochę czasu. Znajomość programowania obiektowego przyda się podczas lektury rozdziału poświęconego usługom sieciowym XML. Przyda się też osobom, które w przyszłości zechcą się zająć bardziej zaawansowanymi technologiami, takimi jak ASP.NET czy JSP.

Podstawowe pojęcia dotyczące programowania obiektowego

Podstawowym elementem programowania obiektowego jest klasa. Klasa to specyficzna konstrukcja języka skupiająca w sobie zmienne i funkcje (czyli pola i metody) opisujące pewien fragment rzeczywistości. Zwykle pola zawierają informację o stanie tego wycinka rzeczywistości, a metody pozwalają na zmianę i kontrolę stanu. Pojęciem ściśle związanym z klasą jest obiekt. Obiekt to nic innego, jak wystąpienie (instancja) klasy — o ile klasa zawiera opis pewnego fragmentu rzeczywistości, o tyle obiekt konkretyzuje klasę. W językach programowania o ścisłej kontroli typów można powiedzieć, że klasa jest typem złożonym, natomiast obiekt jest zmienną tego typu. Ponieważ PHP nie jest językiem ścisłej kontroli typów, skupimy się tylko na ostatniej części tego zdania: obiekt jest zmienną. Zmienna ta zostaje utworzona na podstawie klasy i stanowi rzeczywisty element, który może oddziaływać z innymi obiektami w programie.

Program stworzony w duchu idei programowania obiektowego powinien składać się wyłącznie z klas, dla których są zdefiniowane sposoby wzajemnego oddziaływania mające znaczenie dla poprawnej współpracy obiektów tworzonych na podstawie tych klas.

Klasa jako element języka programowania ma znaczący wpływ na zwiększenie abstrakcji danych. Abstrakcja danych to sposób rozumienia obiektów — nie interesuje nas, programistów, w jaki sposób dany obiekt (w

zasadzie klasa, na podstawie której został utworzony) został zaimplementowany, jakich algorytmów użyto w metodach tego obiektu itp. Dla nas jest ważne, że obiekt danej klasy robi dokładnie to, czego od niego oczekujemy.

Kolejnym ważnym pojęciem związanym z programowaniem zorientowanym obiektowo jest hermetyzacja, nazywana również często enkapsulacją. Hermetyzacja polega na ukrywaniu przed użytkownikiem (tutaj rozumianym jako programista wykorzystujący w swoim projekcie daną klasę) cech obiektu, czyli pól, tak aby użytkownik nie mógł w bezpośredni sposób tych pól modyfikować. Zamiast w możliwość bezpośredniego wpływania na wartość pól klasa powinna być wyposażona w odpowiednie metody, za pomocą których użytkownik w sposób bezpieczny (dla obiektu), w ramach określonych ograniczeń i zastrzeżeń, będzie mógł odczytywać stan tych cech (pól) lub zmieniać ich wartość w sposób prosty (zastąpienie przypisania wywołaniem metody) albo przez wykonanie jakiegoś kodu, stanowiącego ciało metody. Hermetyzacja jest jedną z tych cech programowania obiektowego, które przyczyniły się do jego ciągle rosnącej popularności.

Najtrudniejszym do wyjaśnienia pojęciem w tematyce programowania zorientowanego obiektowo jest polimorfizm. Jest to cecha charakterystyczna dla programowania obiektowego, pozwalająca na używanie w taki sam sposób obiektów o podobnych cechach, lecz o różnych efektach działania. Polimorfizm został dobrze rozwinięty i zaimplementowany w potężnych, obiektowych językach programowania, takich jak C++.

Na koniec kolejna ważna cecha programowania obiektowego: dziedziczenie. Mechanizm dziedziczenia pozwala na tworzenie klas na podstawie innych klas. Powstaje wtedy hierarchia rodzic – dziecko, gdzie klasa pochodna (dziecko) dziedziczy cechy klasy nadrzędnej (rodzic) i posiada również swoje własne cechy. Dziedziczenie pozwala na tworzenie zaawansowanych, wielopoziomowych hierarchii klas.

Kolejne punkty tego rozdziału mają na celu rozwiać wszystkie wątpliwości, które z pewnością pojawiły się po tej, trzeba przyznać, dość ciężkiej dawce teorii. Na początek wyjaśnione zostanie tworzenie klas i obiektów, a następnie zajmiemy się bardziej zaawansowanymi zagadnieniami.

Klasy i obiekty

Klasę w PHP definiuje się za pomocą słowa kluczowego class, po którym następuje identyfikator klasy (tradycja nakazuje, żeby identyfikator zaczynał się od dużej litery) i ciało (wnętrze) klasy zamknięte w nawiasach { i }. Ogólna postać takiej konstrukcji jest następująca:

```
class nazwa_klasy

{

  // definicja  klasy

}
```

Najprostsza klasa to klasa pusta o postaci:

```
class Pusta

{

}
```

W ten sposób powstał nowy typ danych o nazwie Pusta — po takiej definicji można tworzyć zmienne (obiekty) tego typu. Taka klasa nie zawiera żadnych pól ani metod. W zasadzie nie nadaje się ona do niczego (nie ma praktycznego zastosowania), no, może oprócz dziedziczenia, ale o tym później.

Wiadomo, że obiekt to zmienna utworzona na podstawie klasy (czyli wystąpienie klasy). Aby utworzyć obiekt, należy użyć słowa new, po którym musi wystąpić nazwa klasy zakończona parą nawiasów okrągłych,

schematycznie:

new nazwa_klasy();

Aby jednak można było z tak utworzonego obiektu skorzystać, najlepiej go przypisać do jakiejś zmiennej:

$nazwa_zmiennej = new nazwa_klasy();

Po takim przypisaniu $nazwa_zmiennej pozwala na odwoływanie się do nowego obiektu typu nazwa_klasy. Utwórzmy zatem obiekt klasy Pusta. Posłuży do tego instrukcja:

$obiekt = new Pusta();

Jeśli jest to wymagane, w nawiasie występującym po nazwie klasy umieszcza się listę parametrów przekazywanych klasie (w zasadzie konstruktorowi klasy — wkrótce zostanie wyjaśnione, co to takiego). Z formalnego punktu widzenia, jeżeli parametrów nie ma, nie ma również konieczności wstawiania pustych nawiasów, ale dla tych wszystkich, którzy lubią porządek i jednoznaczność, nawiasy te są tutaj niezbędne. Zresztą, tak jak przy wywołaniu funkcji, warto umieszczać puste nawiasy — poprawia to znacznie czytelność kodu.

Listing 8.1 przedstawia skrypt, który zawiera pokazaną wcześniej definicję klasy Pusta, tworzący obiekt tej klasy oraz prezentujący sposób uzyskania informacji o identyfikatorze klasy, na podstawie której powstał obiekt (funkcja get_class()).

Listing 8.1. Definicja klasy

```
<?php

 class Pusta {

 }



 $obiekt = new Pusta();

 print(get_class($obiekt));

?>
```

Spróbujmy teraz utworzyć klasę zawierającą jedno pole — niech to będzie pole o nazwie $imie. Ponieważ język PHP nie wymaga deklaracji zmiennych przed ich użyciem, nie musimy definiować pola wewnątrz klasy, wystarczy zrobić coś takiego (jako rozwinięcie skryptu z listingu 8.1):

$obiekt->imie = "Rafał";

Wygląda dziwnie, ale działa! Do pól obiektu, niezależnie od tego, czy wykonujemy operację odczytu, czy zapisu, odwołujemy się, podając nazwę obiektu, znak strzałki -> i nazwę pola niepoprzedzoną tym razem symbolem $.

Spróbujmy wyświetlić zawartość pola $imie:

print($obiekt->imie);

W efekcie zobaczymy w oknie przeglądarki zawartość pola $imie. Na listingu 8.2 przedstawiono zmodyfikowany skrypt dodający do naszego obiektu pole $imie.

Listing 8.2. Dodawanie pola do obiektu

```
<?php
 class Pusta
 {
 }

 $obiekt = new Pusta();
 print(get_class($obiekt));
 print(&amp;quot;<br />&amp;quot;);
 $obiekt->imie = &amp;quot;Rafał&amp;quot;;
 print($obiekt->imie);
?>
```

Efekt działania tego skryptu przedstawiony został na rysunku 8.1.

Rysunek 8.1. Efekt działania przykładowego skryptu (listing 8.2)

Pola dla danego obiektu można też tworzyć za pomocą metod — w końcu metody służą do operowania na polach obiektu. Listing 8.3 przedstawia skrypt zawierający dwie metody służące do odczytu i zapisu pola $imie. Pole to nie jest bezpośrednio zdefiniowane w klasie, lecz tworzone w momencie pierwszego użycia jednej z metod.

Listing 8.3. Klasa z metodami

```
<?php
 class Osoba
 {
   function podajImie() {
     return $this->imie;
   }
   function ustawImie($imie) {
     $this->imie = $imie;
   }
```

```
}


$obiekt = new Osoba();

$obiekt->ustawImie(&amp;quot;Rafał&amp;quot;);


print($obiekt->podajImie());

?>
```

Jak wynika z listingu 8.3, metody to nic innego, jak tylko funkcje definiowane wewnątrz klasy. Funkcja (metoda) podajImie() zwraca zawartość pola $imie obiektu, natomiast funkcja (metoda) ustawImie() przypisuje do pola $imie obiektu wartość parametru $imie. Uwagę zwraca zmienna o nazwie $this — jest to zmienna, która wewnątrz danego obiektu (czyli używana w metodach tego obiektu) jest synonimem jego samego. W przypadku obu metod użycie tej zmiennej jest konieczne w celu jednoznacznego określenia, o jaki element $imie chodzi. Gdyby nie zastosować $this, funkcja działałaby nieprawidłowo (funkcja ustawImie() musiałaby mieć nawet inną konstrukcję) — tworzona by była lokalna zmienna $imie widoczna tylko wewnątrz funkcji.

Przedstawiony dotychczas sposób definiowania pól i metod nie jest elegancki, nie oddaje istoty obiektowości. O prawdziwej definicji klasy możemy mówić, jeśli zastosujemy hermetyzację, określając, do których elementów klasy jest dostęp z zewnątrz, a które elementy należą wyłącznie do klasy i nie są dostępne poza nią. Elementy klasy (pola i metody) mogą występować w trzech „trybach" dostępu: jako prywatne, chronione i publiczne. Pola i metody prywatne są wyłączną własnością danej klasy i nie są widoczne na zewnątrz, jednak mogą być wykorzystywane wewnątrz metod tej klasy. Pola i metody chronione zachowują się jak prywatne, jednak mogą być dziedziczone przez klasy pochodne (w klasach pochodnych również nie są widoczne na zewnątrz). Pola i metody publiczne są dostępne zarówno wewnątrz klasy, jak i na zewnątrz. Oczywiście należy pamiętać, że odwołania do pól i metod mogą występować tylko dla wcześniej utworzonego obiektu danej klasy (nie dotyczy to tzw. składowych statycznych). Tryby dostępu do pól i metod określa się przez umieszczenie przed identyfikatorem pola lub metody jednego z trzech specyfikatorów dostępu: private (prywatne), protected (chronione) lub public (publiczne). Domyślnie, jeśli nie użyjemy żadnego ze specyfikatorów dostępu, pole lub metoda są traktowane jako publiczne.

Na listingu 8.4 przedstawiono zmodyfikowaną klasę Osoba z zastosowaniem hermetyzacji — podstawowa cecha obiektu (pole $imie) jest prywatna, ukryta dla użytkownika, dostęp do niej możliwy jest tylko przez metody. Dodatkowo metoda ustawImie() sprawdza, czy parametr do niej przekazany zawiera wartość typu ciąg znaków (string).

Listing 8.4. Klasa Osoba — prosta hermetyzacja

```
<?php

class Osoba

{

  private $imie;


  public function podajImie() {
```

```
    return $this->imie;
  }
  public function ustawImie($imie) {
    if (is_string($imie))
      $this->imie = $imie;
    else
      print(&quot;Błędna wartość parametru!&quot;);
  }
}

$obiekt1 = new Osoba();
$obiekt1->ustawImie(&quot;Rafał&quot;);

// Nowy  obiekt klasy Osoba
$obiekt2 = new Osoba();
// Błędna  wartość parametru!
$obiekt2->ustawImie(123);
// Błąd!  Odwołanie do pola prywatnego:
$obiekt2->imie = &quot;Rafał&quot;;

print(&quot;<br />&quot;);
print($obiekt1->podajImie());
?>
```

Jeśli nie oznaczymy jako komentarz (lub nie usuniemy) linii:

```
$obiekt2->imie = &quot;Rafał&quot;;
```

skrypt w momencie uruchomienia wygeneruje dla tej linii komunikat o błędzie krytycznym:

```
Fatal error: Cannot access private property  Osoba::$imie in /home/rafal/public_html/klasa3.php on line 25
```

i zakończy działanie (podana w komunikacie ścieżka dostępu do pliku będzie oczywiście różna w różnych

systemach). Będzie to spowodowane tym, że zmienna $imie jest polem prywatnym klasy Osoba (o czym zresztą informuje komunikat).

Do pól i metod można się odwoływać tylko i wyłącznie dla obiektu danej klasy — mówimy, że są to pola i metody wystąpienia. Istnieje jednak sposób, aby do pól i metod można się było odwoływać bez potrzeby tworzenia instancji danej klasy — mamy wtedy do czynienia z polami i metodami klasy, czyli polami i metodami zdefiniowanymi jako statyczne (z wykorzystaniem słowa kluczowego static). Listing 8.5 przedstawia klasę Osoba, której wszystkie elementy zdefiniowano jako statyczne.

Listing 8.5. Pola i metody statyczne

```
<?php

 class Osoba

 {

   private static $imie;



   static function podajImie() {

     return self::$imie;

   }

   static function ustawImie($imie) {

     if (is_string($imie))

       self::$imie = $imie;

     else

       print(&amp;quot;Błędna wartość parametru!&amp;quot;);

   }

 }



 Osoba::ustawImie(&amp;quot;Rafał&amp;quot;);

 print(Osoba::podajImie());

?>
```

W kodzie z listingu 8.5 zauważyć można dwie zasadnicze zmiany: przede wszystkim nie jest tworzony obiekt klasy Osoba. Zamiast odwoływać się do metod obiektu klasy Osoba, odwołujemy się bezpośrednio do metod klasy Osoba, podając nazwę klasy, po której następuje operator zasięgu :: i nazwa metody. Druga rzecz to użycie słowa self z operatorem zasięgu zamiast zmiennej $this — ta zmiana jest konieczna, ponieważ zmienna $this może być używana tylko w kontekście obiektu, a w naszym przypadku obiekt nie jest tworzony. Efektem działania tego skryptu będzie wyświetlenie imienia „Rafał” w oknie przeglądarki. Pola i metody statyczne stosuje się tylko wtedy,

kiedy jest to konieczne. Definiowanie wszystkich składników klasy jako statycznych jest nieoptymalne ze względu na gospodarkę zasobami — jeżeli używamy obiektów, po ich wykorzystaniu silnik PHP (a dokładniej mechanizm zbierania nieużytków) usuwa obiekty z pamięci.

Na koniec tego punktu parę słów na temat pól i metod finalnych. Zdarza się czasem, że kod, który utworzyliśmy (np. metoda danej klasy), jest ze wszech miar doskonały i nie chcielibyśmy, żeby ta np. doskonała metoda naszej klasy była przedefiniowywana przez klasy pochodne. Pomocnym okazuje się wtedy słowo final, które umieszczone przed definicją pola lub metody czyni to pole lub metodę końcową, bez możliwości jej przedefiniowywania. Przedefiniowywanie metod w klasach pochodnych zostało opisane w punkcie dotyczącym dziedziczenia.

Konstruktor i destruktor

Konstruktor i destruktor to dwie specyficzne metody każdej klasy, wywoływane w ściśle określonych momentach „życia" obiektu. Definiowanie tych metod nie jest konieczne (jak pokazały poprzednie przykłady w tym rozdziale) — w takim przypadku interpreter PHP wygeneruje dla klasy konstruktor i destruktor pusty.

Konstruktor wywoływany jest automatycznie po utworzeniu obiektu danej klasy. Głównym jego przeznaczeniem jest inicjalizacja pól i wykonanie czynności koniecznych do poprawnego działania tworzonego obiektu.

Destruktor wywoływany jest automatycznie w momencie niszczenia obiektu (usuwania obiektu z pamięci). Przeznaczeniem destruktora jest wykonanie czynności końcowych dla obiektu, np. zwalnianie zasobów, zamykanie plików, zamykanie połączeń z bazami danych itp.

Definicja konstruktora polega na umieszczeniu w kodzie klasy metody o nazwie __construct, schematycznie:

```
class nazwa_klasy

{

  function __construct()

  {

    //treść konstruktora

  }

  //pozostałe składowe klasy

}
```

Jest to konstrukcja charakterystyczna dla PHP5 i taki kod nie zadziała w poprzedniej, 4. wersji tego języka. W przypadku gdyby istniała konieczność zachowania przenośności kodu między tymi wersjami, należy zdefiniować konstruktor jako metodę o nazwie zgodnej z nazwą klasy, schematycznie:

```
class nazwa_klasy

{

  function nazwa_klasy()

  {

    //treść konstruktora
```

```
  }

  //pozostałe  składowe klasy

}
```

Destruktor, podobnie jak w przypadku konstruktora, jest to specjalna funkcja o nazwie __destruct, którą należy umieścić w kodzie klasy, schematycznie taka konstrukcja ma postać:

```
class nazwa_klasy

{

  function __destruct()

  {

    //treść  destruktora

  }

  //pozostałe  składowe klasy

}
```

Na listingu 8.6 przedstawiono definicję klasy zawierającej konstruktor i destruktor. Obie te metody wyświetlają w oknie przeglądarki informację wskazującą na ich użycie.

Listing 8.6. Definicje konstruktora i destruktora

```
<?php

 class Testowa

 {

   function __construct() {

     print(&amp;quot;Konstruktor<br  />&amp;quot;);

   }

   function __destruct() {

     print(&amp;quot;Destruktor<br  />&amp;quot;);

   }

 }



 $obiekt = new Testowa();

?>
```

Efekt działania tego skryptu przedstawiony został na rysunku 8.2.

Rysunek 8.2. Użycie konstruktora i destruktora (wykonanie kodu z listingu 8.6)

Mimo iż jedyną operacją wykonywaną przez skrypt jest utworzenie obiektu klasy Testowa, w oknie przeglądarki pojawiły się napisy Konstruktor i Destruktor. Napisy te pokazują, kiedy konstruktor i destruktor są wywoływane. Konstruktor wywołany został podczas tworzenia obiektu, a destruktor w momencie kończenia wykonywania skryptu.

Destruktor obiektu można także wywołać, używając funkcji unset() do usunięcia zmiennej.

Aby umożliwić utworzenie obiektu, konstruktor powinien być zdefiniowany jako metoda publiczna. Jeżeli konstruktor określimy jako private, PHP wygeneruje komunikat o błędzie, w którym pojawi się informacja o braku możliwości utworzenia obiektu. Destruktor powinien być definiowany jako publiczny. Choć zdefiniowanie destruktora jako metody prywatnej nie spowoduje błędu, tylko ostrzeżenie, to jednak destruktor nie zostanie poprawnie wykonany, dlatego też wszystkie operacje, które miał wykonać, zostaną pominięte.

Konstruktor może przyjmować parametry — będą one wykorzystane do nadania polom obiektu odpowiednich wartości. Destruktor musi mieć pustą listę parametrów. Argumenty przekazuje się dokładnie w taki sam sposób jak w przypadku zwykłych metod, czyli budowa takiego konstruktora jest następująca:

```
class nazwa_klasy

{

  __construct(argument1,  argument2,..., argumentN)

  {

    //kod  konstruktora

  }

}
```

Oczywiście jeśli konstruktor przyjmuje argumenty, przy tworzeniu obiektu należy je podać, czyli zamiast stosowanej do tej pory konstrukcji:

```
$zmienna = new nazwa_klasy()
```

trzeba zastosować wywołanie:

```
$zmienna = new nazwa_klasy(argument1, argument2,..., argumentN)
```

Na listingu 8.7 przedstawiono skrypt zawierający definicję klasy Osoba. Klasa ta ma dwie metody, znane z poprzednich przykładów, dwa pola prywatne ($imie i $data) oraz dodatkową metodę umożliwiającą wyświetlenie daty utworzenia obiektu (zapamiętanej dzięki konstruktorowi). Destruktor nie był w przypadku tej klasy potrzebny, dlatego też nie został zdefiniowany.

Listing 8.7. Klasa Osoba z zastosowaniem konstruktora

```
<?php

 class Osoba

 {

   private $imie;

   private $data;


   function __construct($imie) {

     $this->imie = $imie;

     $this->data = date(&quot;Y-m-d,  H:i:s&quot;);

   }

   function podajImie() {

     return $this->imie;

   }

   function ustawImie($imie) {

     if (is_string($imie))

       $this->imie = $imie;

     else

       echo &quot;Błędna wartość parametru!&quot;;

   }

   function dataUtworzenia() {

     return $this->data;

   }

 }


 $obiekt = new Osoba(&quot;Rafał&quot;);

 print($obiekt->podajImie());

 print(&quot;<br />&quot;);

 print($obiekt->dataUtworzenia());
```

```
?>
```

Konstruktor klasy Osoba przyjmuje jeden parametr — jest to imię osoby, czyli ciąg znaków, który następnie przypisuje polu prywatnemu $imie. Jak widać w kodzie z listingu 8.7, parametr konstruktora podaje się w momencie tworzenia obiektu danej klasy, w nawiasie następującym po identyfikatorze klasy. Do odczytu dokładnej daty i godziny utworzenia obiektu wykorzystana została funkcja date(), zwracająca ciąg znaków określający bieżący czas w formie zależnej od ciągu formatującego przekazanego jako parametr. Dokładny opis tej funkcji oraz elementów, które mogą tworzyć ciąg formatujący, znaleźć można w dokumentacji PHP: http://www.php.net/manual/pl/function.date.php.

Wynik działania skryptu z listingu 8.7 pokazuje rysunek 8.3.

Rysunek 8.3. Wynik działania skryptu z listingu 8.7

Dziedziczenie

Dziedziczenie polega na tworzeniu hierarchii obiektów przez definiowanie klas rozszerzających cechy i możliwości swoich klas bazowych. Dziedziczenie często stosowane jest wtedy, gdy np. istniejąca klasa niekoniecznie nadaje się do użycia w naszym programie, ale jeśli dodamy do niej parę elementów, to otrzymamy taką klasę, jakiej oczekiwaliśmy. Ponieważ łatwiej jest zastosować dziedziczenie (przejęcie cech i metod publicznych klasy nadrzędnej, zwanej również bazową), tworzymy klasę rozszerzającą klasę istniejącą.

Dziedziczenie w PHP jest realizowane, podobnie jak w niektórych innych językach programowania, za pomocą słowa extends. Schematycznie taka konstrukcja ma następującą postać:

```
class klasa_bazowa

{

  //kod  klasy bazowej

}



class klasa_potomna  extends klasa_bazowa

{

  //kod  klasy potomnej

}
```

Na listingu 8.8 przedstawiono skrypt zawierający definicję klasy bazowej i pochodnej.

Listing 8.8. Proste dziedziczenie

```
<?php

  class Bazowa

  {
```

```
  public $napis;

  function __construct() {

    $this->napis = &amp;quot;Napis dodany w klasie  bazowej&amp;quot;;;

  }

}



class Pochodna extends Bazowa

{

  function podajNapis() {

    return $this->napis;

  }

}



$obiekt = new Pochodna();

print($obiekt->podajNapis());

?>
```

Na listingu 8.8 widzimy definicję klasy Bazowa, która zawiera pole publiczne $napis oraz konstruktor przypisujący temu polu pewien ciąg znaków. Klasa Pochodna, zdefiniowana poniżej klasy Bazowa, rozszerza klasę Bazowa (czyli dziedziczy jej cechy i metody publiczne) i wprowadza metodę podajNapis(), która odwołuje się do pola $napis, odziedziczonego po klasie Bazowa. Dalej tworzony jest obiekt klasy Pochodna i wywoływana jest metoda podajNapis().

W momencie tworzenia obiektu klasy Pochodna utworzony zostaje również obiekt klasy nadrzędnej w stosunku do Pochodna, czyli klasy Bazowa, stąd po wywołaniu dla obiektu klasy Pochodna metody podajNapis() uzyskujemy efekt, jak pokazano na rysunku 8.4.

Rysunek 8.4. Dziedziczenie (wynik działania skryptu z listingu 8.8)

Niech teraz klasa Bazowa zawiera pewną metodę (listing 8.9), a klasa Pochodna przedefiniuje tę metodę, czyli będzie zawierała metodę o tej samej nazwie, lecz innym, specyficznym działaniu.

Listing 8.9. Przedefiniowanie metody w klasie Pochodna

```
<?php
```

```
class Bazowa

{

  private $napis;

  function __construct() {

    $this->napis = &amp;quot;Napis dodany w klasie  bazowej&amp;quot;;

  }

  function podajNapis() {

    return $this->napis;

  }

}


class Pochodna extends Bazowa

{

  function podajNapis() {

    $str = parent::podajNapis();

    print(&amp;quot;Dostęp z klasy Pochodna: '$str'&amp;quot;);

  }

}


$obiekt = new Pochodna();

$obiekt->podajNapis();

?>
```

Teraz klasa Bazowa zawiera publiczną metodę podajNapis(), ponieważ pole $napis jest polem prywatnym i nie jest dostępne na zewnątrz ani dziedziczone (klasa Pochodna nie ma do niego bezpośredniego dostępu). Klasa Pochodna również zawiera metodę podajNapis(), ale wykonującą co innego niż metoda, którą dziedziczy po klasie Bazowa. W metodzie podajNapis() klasy Pochodna najpierw wywoływana jest metoda o tej samej nazwie, ale zdefiniowana w klasie Bazowa (słowo parent z operatorem zasięgu). Wynik działania tej metody zapisywany jest w zmiennej $str, a następnie wyświetlany w oknie przeglądarki (rysunek 8.5).

Rysunek 8.5. Dziedziczenie (wynik działania skryptu z listingu 8.9)

Jeśli chcemy, żeby niektóre pola klasy bazowej (nadrzędnej) zachowały swój prywatny charakter, a jednocześnie były dziedziczone w klasach pochodnych, powinniśmy je oznaczyć jako chronione (protected). W takim przypadku metody klasy pochodnej będą widziały dziedziczone pola chronione tak, jak własne pola prywatne. Podobnie jest z metodami oznaczonymi jako chronione.

Listing 8.10 zawiera kod skryptu, w którym pole $napis klasy Bazowa zostało oznaczone jako chronione.

Listing 8.10. Dziedziczenie pola chronionego

```
<?php
  class Bazowa
  {
    protected $napis;
    function __construct() {
      $this->napis = &amp;quot;Napis dodany w klasie bazowej&amp;quot;;
    }
  }

  class Pochodna extends Bazowa
  {
    function podajNapis() {
      print($this->napis);
    }
  }

  $obiekt = new Pochodna();
  // To jest błąd:
  $obiekt->napis = &amp;quot;Czy tak można?&amp;quot;;
  $obiekt->podajNapis();
?>
```

Skrypt z listingu 8.10 wygeneruje błąd:

Fatal error: Cannot access protected property Pochodna::$napis in /home/rafal/public_html/klasa5.php on line 17

Stanie się tak, ponieważ pole $napis jest polem chronionym w klasie Bazowa, a ponieważ jest dziedziczone przez klasę Pochodna, również dla obiektu tej klasy jest polem, do którego nie ma dostępu z zewnątrz. Jeśli oznaczymy jako komentarz wiersz 17., w którym jest nieprawidłowe odwołanie do tego pola, skrypt wykona się prawidłowo (wyświetlony zostanie napis Napis dodany w klasie bazowej).

Automatyczne ładowanie klas

Ciekawą funkcjonalnością PHP jest automatyczne ładowanie klas. Mechanizm ten działa w oparciu o funkcję __autoload(), która jako parametr przyjmuje nazwę klasy. Funkcja ta, jeżeli jest zdefiniowana w skrypcie, jest wywoływana, gdy skrypt usiłuje stworzyć obiekt danej klasy, a definicja tej klasy nie jest osiągalna w tym skrypcie. Definicja klasy — żeby automatyczne ładowanie klas działało prawidłowo — powinna być umieszczona w osobnym pliku, najlepiej o nazwie będącej dokładnym odpowiednikiem nazwy klasy i rozszerzeniu .php. Poniższy przykład (listing 8.11) zawiera definicję klasy Testowa, która ma tylko jedną metodę publiczną, wypisującą ciąg znaków w oknie przeglądarki. Klasa ta jest ładowana automatycznie przez skrypt z listingu 8.12 w momencie tworzenia obiektu tej klasy.

Listing 8.11. Klasa Testowa

```
<?php

 class Testowa

 {

   function pisz() {

     print(&amp;quot;Klasa Testowa!&amp;quot;);

   }

 }

?>
```

Listing 8.12. Automatyczne ładowanie klas

```
<?php

  function __autoload($nazwa_klasy) {

    require_once(&amp;quot;$nazwa_klasy.php&amp;quot;);

  }



  $obiekt = new Testowa();

  $obiekt->pisz();

?>
```

Instrukcja require_once() służy do dołączania i wykonywania kodu PHP umieszczonego w innym pliku.

Podsumowanie

Jak to zwykle bywa, w tak krótkim opracowaniu nie sposób zawrzeć wszystkiego, nie można przedstawić wszystkich aspektów programowania zorientowanego obiektowo — przecież na ten temat powstały czasem nawet wielotomowe książki. Jednak rozdział ten miał przybliżyć zagadnienia obiektowości i zachęcić do dalszego drążenia tematu. Charakter PHP do dziś nie narzuca obiektowego stylu programowania, ale czasem warto spróbować zaprojektować serwis WWW w sposób obiektowy — może to znacznie ułatwić pielęgnację i rozwój tego serwisu.

W dokumentacji PHP znajdują się rozdziały poświęcone obiektowości w PHP 4 (http://www.php.net/manual/pl/language.oop.php) oraz w PHP 5 (http://www.php.net/manual/pl/language.oop5.php), jednak w tym rozdziale przedstawiona została obiektowość w stylu PHP 5. Lektura dokumentacji PHP będzie niewątpliwie doskonałym uzupełnieniem tego rozdziału. Opis programowania w stylu obiektowym znajduje się również w publikacji PHP5. Praktyczny kurs (http://helion.pl/ksiazki/php5pk.htm).