Web programozás projekt

*Inheritance*

Boga Áron, Gazdasági informatika

2022

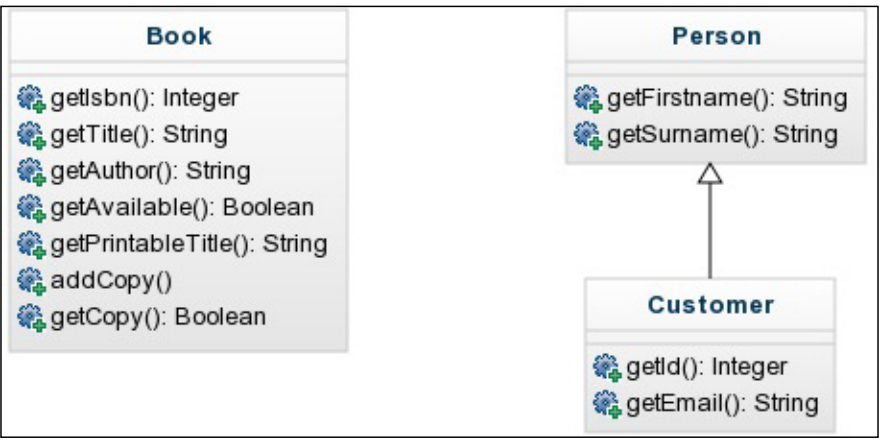
1. **Bevezető**

Az objektum-orientált paradigmákat úgy mutattuk be, mint az összetett adatstruktúrák csodaszere, és bár megmutattuk, hogy tudunk tulajdonságokkal rendelkező objektumokat definiálni, ami szépnek és elegánsnak tűnik, annak ellenére ez nem olyan dolog, amit ne tudnánk megoldani tömbökkel. A beágyazás volt az egyik olyan funkció, amely hasznosabbá tette az objektumokat, mint a tömböket, de valójában az igazi erejük az öröklődésben rejlik.

**Az öröklődés bemutatása**

Az öröklődés az OOP-nek az a képessége, hogy át tudjuk adni az osztály implementációját a szülő osztálytól a gyerekeknek. Igen, az osztályoknak lehetnek szülei, és ennek a technikai megvalósítása az, hogy egy osztály egy másik osztályból származik. Az osztályok bővítésekor minden megkapunk, vagyis a nem privátként definiált tulajdonságokat és metódusok, gyermekosztály pedig úgy használja őket, mintha a sajátjai lennének. A korlátozás az, hogy egy osztály csak egyetlen egy szülőosztályból származhat.

Példaként vegyük az Ügyfél osztályunkat. A ezeket a tulajdonságokat tartalmazza: keresztnév, vezetéknév, email és id. Az ügyfél valójában egy bizonyos típusú ember, olyan valaki, aki regisztrálva van a rendszerünkben, így hozzájuthat a könyvekhez. De lehet más típusú személy is a rendszerünkben, például könyvtáros vagy vendég. És mindegyiküknek van néhány közös tulajdonságuk, azaz a keresztnév és a vezetéknév. Így ennek akkor van értelme, ha létrehozunk egy Személy osztályt, és abból származtatjuk az Ügyfél osztályt. Az öröklődési fa a következőképpen nézne ki:



Nézzük meg, hogy az Ügyfél osztály hogyan kapcsolódik a Személy osztályhoz. A Személyben szereplő metódusok nincsenek meghatározva az Ügyfél osztályban, mivel ezek az öröklődésből erednek. Most mentsük az új osztályt src/Domain/Person.php:

<?php

namespace Bookstore\Domain;

class Person {

protected $firstname;

protected $surname;

public function \_\_construct(string $firstname, string $surname) {

$this->firstname = $firstname;

$this->surname = $surname;

}

public function getFirstname(): string {

return $this->firstname;

} public function getSurname(): string {

return $this->surname;

}

}

Az előző kódrészletben meghatározott osztály nem tűnik különlegesnek; mi csak két tulajdonságot definiáltunk, egy konstruktort és két getter metódust. Jegyezzük meg azonban, hogy meghatároztuk, hogy a tulajdonságoknak protected láthatósága legyen legyenek, mert ha privátnak határoztuk meg, akkor a gyerekek nem tudnak hozzájuk férni. Mostantól frissíthetjük Ügyfélosztályunkat úgy, hogy eltávolítjuk az ismétlődő tulajdonságokat és azok getter metódusait:

<?php

namespace Bookstore\Domain;

class Customer extends Person {

private static $lastId = 0;

private $id;

private $email;

public function \_\_construct( int $id,string $name, string $surname, string $email ) {

if (empty($id)) {

$this->id = ++self::$lastId;

} else {

$this->id = $id;

if ($id > self::$lastId) {

self::$lastId = $id;

}

}

$this->name = $name;

$this->surname = $surname;

$this->email = $email;

}

public static function getLastId(): int {

return self::$lastId;

}

public function getId(): int {

return $this->id;

}

public function getEmail(): string {

return $this->email;

}

public function setEmail($email): string {

$this->email = $email;

}

}

Vegyük figyelembe, hogy az új kulcsszó, az extends; azt mutatja meg a PHP-nek, hogy ez az osztály a Személy osztály gyermeke vagy más néven a leszármazott osztálya. Mivel a Személy és az Ügyfél ugyanabban a névtérben van, nem kell hozzáadni semmilyen használati utasítást, de ha nem, akkor tudatni kell vele, hogy hogyan találja meg a szülő osztályt. Ez a kód jól működik, de láthatjuk, hogy van egy kisebb hiba a kódban. A Megrendelő osztály konstruktora ugyanazt a munkát végzi, mint a Személy osztály konstruktora! Igyekszünk hamarosan megjavítani. Ezt hamarosan kiküszöböljük.

Ahhoz, hogy a szülő osztály metódusára vagy tulajdonságára hivatkozzunk a gyermekosztályból, használhatjuk a $this kulcsszót, ez olyan, mintha a tulajdonság vagy metódus ugyanahhoz az osztályhoz tartozna. A PHP lehetővé teszi egy metódus újradefiniálását a gyermek osztályban, abban az esetben, ha az már jelen van a szülő osztályban. Ha implementálni szeretné a szülő osztályt , nem használhatja a $this kulcsszót, mivel a PHP a gyermekben lévőt fogja meghívni. Ahhoz, hogy rákényszerítsük a PHP-t, hogy használja a szülő metódusát, a parent:: kulcsszót kell használnunk a $this helyett. Frissítsük a Vevő osztály konstruktorát az alábbiak szerint:

public function \_\_construct( int $id, string $firstname, string $surname, string $email ) {

**parent::\_\_construct($firstname, $surname);**

if (empty($id)) {

$this->id = ++self::$lastId;

} else {

$this->id = $id;

if ($id > self::$lastId) {

self::$lastId = $id;

}

}

$this->email = $email;

}

Ez az új konstruktor nem duplikálja a kódot. Ehelyett a Személy osztály konstruktorát hívja meg. Kerüljük a kódduplikációt, és ráadásul ez megkönnyíti a személy konstruktorának jövőbeni módosításait. Ha meg kell váltotatnunk a Person konstruktorát, akkor azt megtehetjük egyetlen helyen, az összes gyerek helyett.

**Metódusok túlterhelése**

Ahogy korábban is mondtuk, az osztályból való öröklődéskor a szülő összes metódusát megkapja a gyerek osztály. Ez implicit, tehát valójában nem írjuk le őket a gyermek osztályában.

Mi történne, ha egy másik metódust implementálnánk ugyanazzal az elnevezéssel? Felülírjuk a metódust.

Mivel nincs szükségünk erre a funkcióra az osztályainkban, csak adjunk hozzá néhány kódot az init.php-hoz, hogy megjelenítse ezt a viselkedést, majd egyszerűen eltávolíthatjuk azt. Definiáljuk, a Pops osztályt, és egy Child osztályt, amely a szülőből származik, és egy sayHi metódus mindkettőben:

class Pops {

public function sayHi() {

echo "Hi, I am pops.";

}

}

class Child extends Pops{

**public function sayHi() {**

**echo "Hi, I am a child.";**

}

}

$pops = new Pops();

$child = new Child();

echo $pops->sayHi();

// Hi, I am pops. echo $child->sayHi();

// Hi, I am Child.

A kiemelt kód azt mutatja, hogy a metódus felül lett írva, tehát mikor gyermek szemszögéből hivatkozva inkább azt fogjuk használni, mintsem a szülőből örököltet. De mi történik, ha az örököltre akarunk hivatkozni? Mindig hivatkozhatunk rá a parent kulcsszóval. Lássuk, hogyan működik:

class Child extends Pops{

public function sayHi() {

echo "Hi, I am a child.";

**parent::sayHi();**

}

}

$child = new Child();

echo $child->sayHi(); // Hi, I am Child. Hi I am pops.

Most a gyerek köszön maga helyett és az szülő helyében is. Nagyon könnyűnek tűnik és

praktikusnak, igaz? Nos, van egy korlátozás. Képzeld el, hogy a gyerekolyan, mint a való életben nagyon félénk, és nem köszön mindenkinek. Megpróbálhatjuk beállítani a metódus láthatóságát, hogyha az protected de nézzük meg, mi történik:

class Child extends Pops{

**protected function sayHi() {**

echo "Hi, I am a child.";

}

}

Amikor megpróbálja ezt a kódot lefuttatni, akkor egy Fatal Error hibaüzenetet fogunk kapni, a metódus hozzáférési szintjével kapcsolatban. Az ok az, hogy mikor felülírjuk, akkor a metódusnak legalább akkora láthatósággal kell rendelkeznie, mint az öröklöttnek.

Ez azt jelenti, hogy ha egy védettet örököltetünk, felülírhatjuk egy másikkal védettel vagy nyilvánossal, de soha nem egy priváttal.

**Absztrakt osztályok**

Ne feledje, hogy minden alkalommal csak egy szülő osztályból örököltethetünk. Ez azt jelenti,

hogy az Ügyfél csak Személyből örökölhet. De ha ezt hierarchikussá akarjuk tenni bonyolultabb, gyermekosztályokat hozhatunk létre, amelyek az Ügyfélből származnak, és ezek az osztályok implicit módon a Személyből is örökölni fognak. Hozzunk létre két típusú ügyfélt: alap és prémium. Ez a két ügyfél ugyanazokkal a tulajdonságokkal rendelkezik valamint az Ügyféltől és a Személytől származnak.

Mentsük le az alábbi kódot ide: src/Domain/Customer/Basic.php

<?php

namespace Bookstore\Domain\Customer;

use Bookstore\Domain\Customer;

class Basic extends Customer {

public function getMonthlyFee(): float {

return 5.0;

}

public function getAmountToBorrow(): int {

return 3;

}

public function getType(): string {

return 'Basic';

}

}

És az alábbi kódot ide: src/Domain/Customer/Premium.php

<?php

namespace Bookstore\Domain\Customer;

use Bookstore\Domain\Customer;

class Premium extends Customer { public function getMonthlyFee(): float {

return 10.0;

}

public function getAmountToBorrow(): int {

return 10;

}

public function getType(): string {

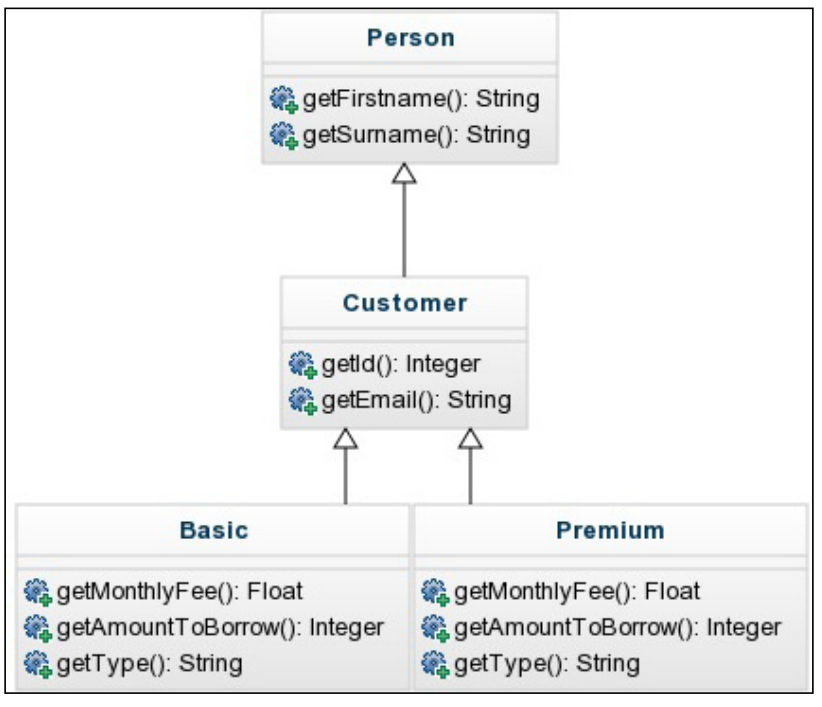
return 'Premium';

}

}

Az előző két kódból meg kell jegyezni, hogy az Ügyfélből örököltettönk két különböző osztályba

, és ez teljesen. Ezzel a kiegészítéssel a Személy hierarchikus fája a következőképpen nézne ki:



Ugyanazokat a metódusokat definiáljuk ebben a két osztályban, de az implementációjuk igen

különböző. Ennek a megközelítésnek az a célja, hogy mindkét típusú vevőt egyedileg használjuk, anélkül, hogy tudnánk, melyik az. Például átmenetileg beilleszthetnénk a következő kódot az init.php-ba. Ne felejtsük el hozzáadni a use utasítást az importáláshoz az Ügyfél osztályhoz, ha még nincs hozzáadva.

function checkIfValid(Customer $customer, array $books): bool {

return $customer->getAmountToBorrow() >= count($books);

}

Az előző függvény azt mondja meg, hogy egy adott ügyfél kölcsönözheti-e az összes könyvet

a tömbben. Ez elfogadja az objektumokat, amelyek az Ügyfél példányai vagy bármely más osztály példányai, amelyek az Ügyfélből származnak, azaz Basic vagy Premium osztályok. Jónak tűnik, igaz? Gyerünk akkor próbáljuk meg használni:

$customer1 = new Basic(5, 'John', 'Doe', 'johndoe@mail.com'); var\_dump(checkIfValid($customer1, [$book1])); // ok

$customer2 = new Customer(7, 'James', 'Bond', 'james@bond.com'); var\_dump(checkIfValid($customer2, [$book1])); // fails

Az első megszóllítás a vártnak megfelelően működik, de a második sikertelen, pedig mi

Ügyfél objektumot küldtünk. A probléma abból adódik, hogy a szülő nem ismer semmilyen getAmountToBorrow metódust! Az is veszélyesnek tűnik, hogy a gyerekek mindig használják ezt a metódust. A megoldás az absztrakt osztályok használatában rejlik.

Az absztrakt osztály olyan osztály, amely nem példányosítható. Egyedüli célja az, hogy meggyőzödjön arról, hogy a gyerek osztály helyesen van implementálva. Egy osztály absztraktnak nyilvánítása az abstract kulcsszóval történik, majd a normál osztály definíciójával. Mi is meg tudjuk határozni ezeket a metódusokat. Ezeket a metódusokat absztrakt metódusoknak nevezzük, és definiálva vannak az abstract kulcsszóval az elején. Természetesen a többi normál metódus is ott maradhat, és örökölhetik a gyermek osztályok:

<?php

**abstract class Customer extends Person {**

**//...**

**abstract public function getMonthlyFee();**

**abstract public function getAmountToBorrow();**

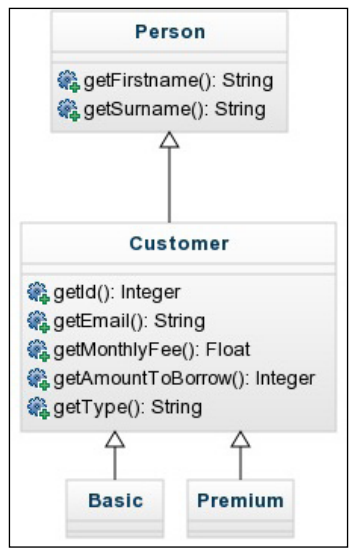
**abstract public function getType();**

**//...**

**}**

Az előző absztrakció mindkét problémát megoldja. Először is, nem fogunk tudni elküldeni az Ügyfél osztály bármely példányát, mert nem tudjuk példányosítani azt. Ez azt jelenti, hogy az összes objektum, amelyet a checkIfValid metódus elfogad, csak az Ügyfél osztály gyerekeitől származhat. Másrészt az absztrakt módszerek deklarálása minden gyereket kényszerít amelyek örökölnek az osztályok implementációjára. Ezzel biztosítjuk, hogy minden objektum megtörténjen implementálja a getAmountToBorrow-t, és a kódunk biztonságos.

Az új hierarchikus fa határozza meg a három absztrakt metódust az Ügyfél osztályban. Igaz, hogy ezeket implementálhatjuk a gyerekekben, de mivel ezeket az Ügyfél érvényesíti, és az absztrakciónak köszönhetően biztosak vagyunk abban, hogy minden az ebből kiinduló osztálynak végre kell hajtaniuk azokat, és ez biztonságos. Lássuk, hogyan történik ez:



Az utolsó hozzáadással az init.php fájl hibás lesz. Az ok az, hogy az megpróbálja példányosítani az Ügyfél osztályt, de az absztrakt, így nem lehet. Példányosítsunk egy konkrét osztályt, vagyis olyat, ami nem absztrakt a probléma megoldásához.

**Interfészek**

Az interfész egy OOP elem, amely függvénydeklarációkészletet csoportosít anélkül, hogy implementálná ezeket, vagyis megadja a nevet, a visszatérési típust és az argumentumokat, de a kódblokkot nem . Az interfészek különböznek az absztrakt osztályoktól, mivel egyáltalán nem tartalmazhatnak bármilyen implementációt, míg az absztrakt osztályok keverhetik a metódust definíciókat és az implementáltakat. Az interfészek célja annak meghatározása, hogy milyen osztály csinálhatja, de azt nem határozza meg hogy miként tehetik meg.

A kódunkból azonosítani tudjuk az interfészek lehetséges felhasználását. Az ügyfeleknek van egy elvárt viselkedésük, de megvalósítása az ügyfél típusától függően változik. Tehát az Ügyfél lehet interfész, absztrakt osztály helyett. De interfészként nem tud semmilyen függvényt megvalósítani, és nem tartalmazhat tulajdonságokat, át kell helyeznünk a konkrét kódot az Ügyfél osztályból valahova máshová. Egyelőre tegyük fel a Személy osztályba. Szerkesszük a Személy osztályt a képen látható módon:

<?php

namespace Bookstore\Domain;

class Person {

**private static $lastId = 0;**

**protected $id;**

protected $firstname;

protected $surname;

**protected $email;**

**public function \_\_construct( int $id, string $firstname, string $surname, string $email ) {**

**$this->firstname = $firstname;**

**$this->surname = $surname;**

**$this->email = $email;**

**if (empty($id)) {**

**$this->id = ++self::$lastId;**

**} else {**

**$this->id = $id;**

**if ($id > self::$lastId) {**

**self::$lastId = $id;**

}

}

}

public function getFirstname(): string {

return $this->firstname;

}

public function getSurname(): string {

return $this->surname;

}

**public static function getLastId(): int {**

**return self::$lastId;**

**}**

**public function getId(): int {**

**return $this->id;**

**}**

**public function getEmail(): string {**

**return $this->email;**

**}**

}

**A dolgok bonyolítása több mint fontos**

Az interfészek nagyon hasznosak, de mindig megvan mindennek az ideje és a helye. Mivel alkalmazásunk didaktikai jellege miatt nagyon egyszerű, nincs igazi szükség rájuk. Az interfészek működésének bemutatásához a kódunkat hozzájuk fogjuk igazítani. Ne aggódjon azonban, mivel a legtöbb kódot, amelyet bemutattunk most jobb gyakorlatok váltják fel, miután bevezetjük az adatbázisokat és az MVC-mintát az 5. fejezetben (Adatbázisok használata).

Amikor saját applikációkat ír, ne próbálja bonyolítani a dolgokat jobban a kelleténél. Gyakran előfordul, hogy nagyon összetett kódot látunk fejlesztőktől, akik igyekeznek megmutatni minden képességüket, amivel rendelkeznek. Csak a szükséges eszközöket használja a tiszta kódért.

Módosítsa a Customer.php tartalmát az alábbiakkal:

<?php

namespace Bookstore\Domain;

interface Customer {

public function getMonthlyFee(): float;

public function getAmountToBorrow(): int;

public function getType(): string;

}

Vegyük figyelembe, hogy egy interfész nagyon hasonlít egy absztrakt osztályhoz. A különbségek az a interface kulcsszóval van definiálva, és metódusaiban nem szerepel az absztrakt szó. Az interfészek nem példányosíthatók, mivel metódusaik nincsenek implementálva mint az mint absztrakt osztályoknál. Az egyetlen dolog, amit tehetsz velük, hogy osztályt készítesz, hogy implementáld őket.

Egy interfész implementálása az összes benne meghatározott metódus implementálását jelenti, például, amikor kiterjesztettünk egy absztrakt osztályt. Az absztrakt osztály kiterjesztésének minden előnyével rendelkezik, például az adott típushoz tartozó osztályok.

A fejlesztő szemszögéből egy felületet megvalósító osztály használata olyan, mint az írás

szerződés: biztosítod, hogy az osztályodban mindig a deklarált metódusok legyenek az interfészek, a megvalósítástól függetlenül. Emiatt az interfészek csak a nyilvános metódusokkal törődnek, amelyeket más fejlesztők használhatnak. Csak annyit kell módosítania a kódban, hogy kicseréled az extends kulcsszavat implements-re:

class Basic implements Customer {

Tehát miért használna valaki egy interfészt, ha használhatna egy absztrakt osztályt ami nemcsak a metódusok implementációját kényszeríti ki, hanem lehetővé teszi az öröklődést is? Ennek az az oka, hogy csak egy osztályból tudsz származtatni, de megvalósíthatsz több példányt egyszerre. Képzeld el, hogy van egy másik felületed egy meghatározott fizetővel. Ez azonosíthatja valakit, aki képes fizetni valamit, függetlenül attól, hogy mi az.

Mentsd el a következő kódot az src/Domain/Payer.php fájlba:

<?php

namespace Bookstore\Domain;

interface Payer {

public function pay(float $amount);

public function isExtentOfTaxes(): bool;

}

Mostantól alap és prémium ügyfeleink is implementálhatja mindkét interfészeket. Az alap ügyfél a következőképpen fog kinézni:

//...

use Bookstore\Domain\Customer;

use Bookstore\Domain\Person;

class Basic extends Person implements Customer {

public function getMonthlyFee(): float {

//...

És a prémium ügyfél így:

//...

use Bookstore\Domain\Customer;

use Bookstore\Domain\Person;

class Premium extends Person implements Customer {

public function getMonthlyFee(): float {

//...

Látnia kell, hogy ez a kód már nem működik. Az ok az, hogy bár egy második interfészt valósítunk meg, a metódusok nincsenek implementálva. Add hozzá ezt a két metódust az ügyfélosztályhoz:

public function pay(float $amount) {

echo "Paying $amount.";

}

public function isExtentOfTaxes(): bool {

return false;

}

Add ezt a két metódust a prémium ügyfélosztályhoz:

public function pay(float $amount) {

echo "Paying $amount.";

}

public function isExtentOfTaxes(): bool {

return true;

}

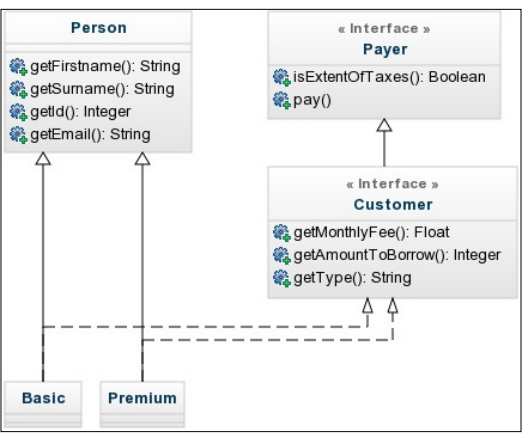
Ha tudja, hogy minden ügyfélnek fizetőnek kell lennie, akkor akár az Ügyfél interfész is örökölheti a Fizető interfészből:

interface Customer extends Payer {

Ez a változás egyáltalán nem érinti osztályaink használatát. A többi fejlesztő meglátja hogy alap- és prémium ügyfeleink örökölnek Fizetőtől és Ügyféltől és, minden szükséges metódust tartalmaznak. Hogy ezek az interfészek függetlenek legyenek, illetve hogy egymásból örököljenek, aminek nem lesz túlságosan nagy hatása.

Az interfészek csak más interfészekből, az osztályok pedig csak más osztályokból örökölhetnek. Az egyetlen módja annak, hogy összekeverjük őket, ha egy osztály egy interfészt implementál, de sem egy osztály nem örököl interfészből, sem interfész osztályból. De a típusos szempontjából felváltva használhatók.

A rész összefoglalásához és a dolgok egyértelművé tételéhez mutassuk meg, hogy hogyan néz ki a hierarchikus fa. Mint az absztrakt osztályokban, a metódusok deklaráltak egy interfészben is.



**Polimorfizmus**

A polimorfizmus egy OOP újítás, amely lehetővé teszi, hogy különböző osztályokkal dolgozzunk, amelyek ugyanazt az interfészt valósítják meg. Ez az objektum-orientált programozás egyik szépsége. Lehetővé teszi a fejlesztő számára, hogy összetett osztályrendszert és hierarchikus fákat hozzon létre, de egyszerű módot kínál a velük való munkavégzésre.

Képzeljük el, hogy van egy függvényünk, amely adott fizető esetében ellenőrzi, hogy mentes-e az adó alól vagy sem, és bizonyos összeget fizet. Ez a kódrészlet nem nagyon kell figyeld, ha a fizető az ügyfél, könyvtáros vagy valaki, akinek nincs semmi dolga a könyvesbolttal. Az egyetlen dolog, ami érdekel, az az, hogy a fizetőnek megvan a lehetősége fizetni. A függvény a következő lehet:

function processPayment(Payer $payer, float $amount) {

if ($payer->isExtentOfTaxes()) {

echo "What a lucky one...";

} else {

$amount \*= 1.16;

}

$payer->pay($amount);

}

Ebbe a funkcióba alap- vagy prémium ügyfeleket küldhetsz, és a viselkedés más lesz. De mivel mindkettő megvalósítja a Fizető interfészt, mindkét megadott objektum érvényes típusok, és mindkettő képes a szükséges műveletek végrehajtására.

A checkIfValid függvény egy vásárlót és egy könyvlistát vesz fel. Már láttuk hogy bármilyen ügyfél küldésével a funkció az elvárt módon működik. De mit történik, ha elküldjük a Librarian osztály objektumát, amely a Fizetőből származik? Mivel a Fizető nem tud az Ügyfélről (vagy fordítva), a funkció „panaszkodni” fog, mivel a típus-hivatkozás nem valósul meg.

A PHP egyik hasznos funkciója az, hogy ellenőrizni tudja, hogy egy objektum valóban egy adott osztály vagy interfész példánya-e. A használat módja: a változó megadása ezt követi az instanceof kulcsszó és az osztály vagy felület neve. Logikai értéket ad vissza, ami igaz, ha az objektum olyan osztályból származik, amely örökli vagy végrehajtja, vagy egyébként false. Lássunk néhány példát:

$basic = new Basic(1, "name", "surname", "email");

$premium = new Premium(2, "name", "surname", "email");

var\_dump($basic instanceof Basic);

// true var\_dump($basic instanceof Premium);

// false var\_dump($premium instanceof Basic);

// false var\_dump($premium instanceof Premium);

// true var\_dump($basic instanceof Customer);

// true var\_dump($basic instanceof Person);

// true var\_dump($basic instanceof Payer); // true

Ne felejtsük el hozzáadni az összes use utasítást minden osztályhoz vagy interfészhez, így a PHP megérti, hogy a megadott osztálynév a fájl névterében található.

**Tulajdonságok**

Eddig megtanultuk, hogy az osztályokból való öröklődés lehetővé teszi, hogy kódot is örökölj. (tulajdonságok és metódusmegvalósítások), de ez azzal a korlátozással jár, hogy csak egy osztályból lehet örökölni. Másrészt, interfészek segítségével több viselkedést is megvalósíthatsz ugyanabból az osztályból, de így nem örökölhetsz kódot. Ennek a hiánynak a kitöltéséhez, vagyis ahhoz, hogy több helyről tudjon örökölni kódot, vannak a traitek.

A tulajdonságok olyan mechanizmusok, amelyek lehetővé teszik a kód újrafelhasználását, vagy inkább "örökléstét", kód másolását-beillesztését, egyszerre több forrásból. A tulajdonságok, úgy mint az absztrakt osztályok vagy interfészek, nem példányosíthatók; ezek csak olyan funkciók tömbjei, amelyek más osztályokból is használhatók.

Ha emlékszel, van néhány kódunk a Személy osztályban, amely kezeli a id-k kiosztását. Ez a kód valójában nem egy személy része, hanem inkább egy Azonosító rendszeré, amelyet más, id-vel rendelkező dolog is használhat. Az egyik módja ennek a funkciónak a személyből való kinyeréséhez –nem ez a legjobb módja ennek, de azért, hogy a tulajdonságokat működés közben lássuk, ezt választjuk– az, hogy áthelyezzük egy tulajdonságba.

Egy tulajdonság meghatározásához tegyeünk úgy, mintha egy osztályt határoznánk meg, csak használjuk a trait kulcsszót a class helyett. Adjuk meg a névterét, adjuk hozzá a szükséges use utasításokat, deklaráljuk tulajdonságait és implementáljik a metódusait, és mindent berakunk egy fájlba, amely követi a ugyanazokat a konvenciókat.

Adjuk hozzá a következő kódot az src/Utils/Unique.php fájlhoz:

<?php

namespace Bookstore\Utils;

**trait Unique** {

private static $lastId = 0;

protected $id;

**public function setId(int $id) {**

if (empty($id)) {

$this->id = ++self::$lastId;

} else {

$this->id = $id;

if ($id > self::$lastId) {

self::$lastId = $id;

}

}

} public static function getLastId(): int {

return self::$lastId;

} public function getId(): int {

return $this->id;

}

}

Ügyeljen arra, hogy a névtér nem ugyanaz, mint általában, mivel ezt a kódot egy másik fájlban tároljuk. Ez konvenciók kérdése, de ön teljesen szabadon használhatja azt a fájlt struktúrát, amelyet minden jobbnak tart. Ebben az esetben nem gondoljuk, hogy ez a tulajdonság az „üzleti logikát” képviselné, mint ahogy az ügyfelek és a könyvek teszik.

A Személyből származó id-hoz kapcsolódó összes kódot megadjuk. Ez magában foglalja a tulajdonságokat, a gettereket és a konstruktoron belüli kódot. Mivel a tulajdonság nem példányosítható, nem adhatunk hozzá konstruktort. Ehelyett hozzáadtunk egy setId metódust, amely tartalmazza azt a kódot. Amikor létrehozunk egy új példányt, amely ezt a tulajdonságot használja, meghívhatjuk ezt a setId metódust az azonosító beállításához a felhasználó által argumentumként küldött adatok alapján.

Az Személy osztályt is meg kell változnia. El kell távolítanunk minden id-re való hivatkozást és valahogyan meg kell határoznunk, hogy az osztály használja a tulajdonságot. Ehhez használjuk a use kulcsszót, mint a névterekben, de az osztályon belül.

Nézzük, hogyan nézne ki:

<?php

namespace Bookstore\Domain;

use Bookstore\Utils\Unique;

class Person {

use Unique;

protected $firstname;

protected $surname;

protected $email;

public function \_\_construct(

int $id,

string $firstname,

string $surname,

string $email

) {

$this->firstname = $firstname;

$this->surname = $surname;

$this->email = $email;

$this->setId($id);

}

public function getFirstname(): string {

return $this->firstname;

}

public function getSurname(): string {

return $this->surname;

}

public function getEmail(): string {

return $this->email;

}

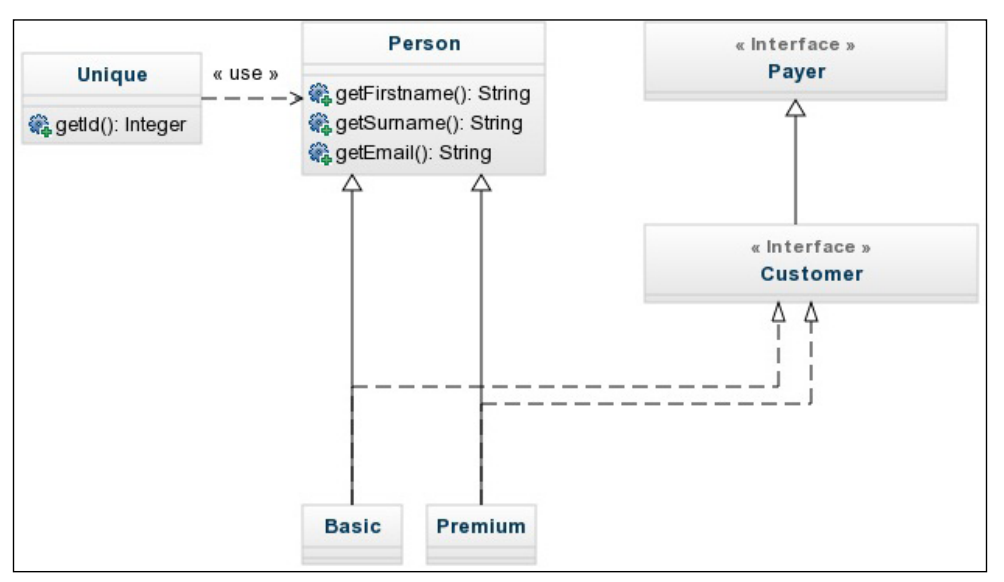
public function setEmail(string $email) {

$this->email = $email;

}

}

Hozzáadjuk az use Unique; utasítást, hogy az osztály tudja, hogy ez használja a tulajdonságot. Mindent eltávolítunk, ami az azonosítókkal kapcsolatos, még a konstruktoron belül is. A konstruktor első argumentumaként továbbra is egy ID-t kapunk, de megkérdezzük a tulajdonságból a setId metódust, hogy tegyen meg mindent számunkra. Ne feledje, hogy erre a metódusra a $this kulcsszóval hivatkozunk, mintha a metódus az osztályon belül lenne. A frissített hierarchikus fa a következőképpen néz ki (vegye figyelembe, hogy nem adunk hozzá minden metódust minden olyan osztályhoz vagy interfészhez, amely nem érintett a legutóbbi változtatásokban, hogy a diagram a lehető legkisebb és legolvashatóbb olvashatóbb legyen):



Lássuk, hogyan működik, még akkor is, ha úgy működik, ahogyan azt valószínűleg elvárnánk. Adja hozzá ezt a kódot az init.php fájlhoz, használja a use utasításokat, és futtassa a böngészőjében:

$basic1 = new Basic(1, "name", "surname", "email");

$basic2 = new Basic(null, "name", "surname", "email");

var\_dump($basic1->getId()); // 1

var\_dump($basic2->getId()); // 2

Az előző kód két ügyfelet példányosít. Ezek közül az első egy meghatározott azonosítóval rendelkezik, míg a második lehetővé teszi, hogy a rendszer azonosítót válasszon hozzá. Az eredmény az, hogy a második alapügyfél 2-es azonosítóval rendelkezik. Ez várható is, mivel mindkét ügyfél alapügyfél. De mi történne, ha az ügyfelek különböző típusúak lennének?

$basic = new Basic(1, "name", "surname", "email");

$premium = new Premium(null, "name", "surname", "email");

var\_dump($basic->getId()); // 1

var\_dump($premium->getId()); // 2

Az azonosítók továbbra is ugyanazok. Ez várható is, mivel a tulajdonság a Person osztályba tartozik, így a $lastId statikus tulajdonság a Person osztály összes példánya között meg lesz osztva, beleértve az alap és prémium ügyfeleket is. Ha a Személy helyett az Alap és Prémium ügyfél tulajdonságát használná (de nem szabadna), akkor a következő eredményt kapná:

var\_dump($basic->getId()); // 1

var\_dump($premium->getId()); // 1

Minden osztálynak saját statikus tulajdonsága lesz. Az összes alapszintű példánynak ugyanaz a $lastId-ja lesz, ami különbözik a prémium példányok $lastId-jétől. Ezért világossá kell tennie, hogy a tulajdonságok statikus tagjai ahhoz az osztályhoz kapcsolódnak, amelyik használja őket, nem pedig magához a tulajdonsághoz. Ez tükröződhet a következő kód tesztelésében is, amely az eredeti forgatókönyvünket használja, ahol a tulajdonságot a Személyből használják:

$basic = new Basic(1, "name", "surname", "email");

$premium = new Premium(null, "name", "surname", "email");

var\_dump(Person::getLastId()); // 2

var\_dump(Unique::getLastId()); // 0

var\_dump(Basic::getLastId()); // 2

var\_dump(Premium::getLastId()); // 2

Ha jól látja a problémákat, akkor elkezdhet gondolkodni a tulajdonságok használatával kapcsolatos lehetséges problémákon. Mi történik, ha két olyan tulajdonságot használunk, amelyek ugyanazt a módszert tartalmazzák? Vagy mi történik, ha olyan tulajdonságot használ, amely egy olyan metódust tartalmaz, amely már implementálva van az adott osztályban?

Ideális esetben kerülnie kell az ilyen helyzetekbe való belefutást; ezek figyelmeztető lámpák az esetleges rossz kialakításra. De mivel mindig lesznek rendkívüli esetek, lássunk néhány elszigetelt példát, hogyan viselkednének.

A forgatókönyv, amikor a tulajdonság és az osztály ugyanazt a metódust valósítja meg, egyszerű. Az osztályba explicit módon implementált metódus a nagyobb prioritású, ezt követi a tulajdonságban megvalósított metódus, végül pedig a szülőosztálytól örökölt metódus. Lássuk, hogyan működik. Vegyük például a következő tulajdonság- és osztálydefiníciókat:

<?php

trait Contract {

public function sign() {

echo "Signing the contract.";

}

}

class Manager {

use Contract;

public function sign() {

echo "Signing a new player.";

}

}

Mindkettő a sign metódust alkalmazza, ami azt jelenti, hogy alkalmaznunk kell a korábban meghatározott elsőbbségi szabályokat. Az osztályban definiált metódus elsőbbséget élvez a tulajdonságból származóval szemben, tehát ebben az esetben a végrehajtott metódus lesz az osztályból származó metódus:

$manager = new Manager();

$manager->sign(); // Signing a new player.

A legbonyolultabb forgatókönyv az lenne, amikor egy osztály két tulajdonságot használ ugyanazzal a módszerrel. Nincsenek olyan szabályok, amelyek automatikusan megoldják a konfliktust, ezért ezt kifejezetten meg kell oldania. Nézze meg a következő kódot:

<?php

trait Contract {

public function sign() {

echo "Signing the contract.";

}

}

trait Communicator {

public function sign() {

echo "Signing to the waitress.";

}

class Manager {

use Contract, Communicator;

}

$manager = new Manager();

$manager->sign();

Az előző kód végzetes hibát dob, mivel mindkét tulajdonság ugyanazt a módszert valósítja meg. A használni kívánt típus kiválasztásához ehelyett az operátort kell használnia. Használatához adja meg a tulajdonság nevét és a használni kívánt metódust, utána az insteadof kulcsszavat és a tulajdonságot, amelynek használatát elutasítja. Opcionálisan használja az as kulcsszót név hozzáadásához, ahogyan azt a névtereknél tettük, így mindkét módszert használhatja:

class Manager {

use Contract, Communicator {

Contract::sign insteadof Communicator;

Communicator::sign as makeASign;

}

}

$manager = new Manager();

$manager->sign(); // Signing the contract.

$manager->makeASign(); // Signing to the waitress.

Láthatja, hogy úgy döntöttünk, hogy a Contract metódust használjuk a Communicator helyett, de hozzáadtuk az álnevet, hogy mindkét módszer elérhető legyen. Remélhetőleg belátható, hogy még a konfliktusokat is meg lehet oldani, és vannak konkrét esetek, amikor nem lehet mást tenni, mint kezelni őket; általában rossz jelnek tűnnek .

**Kivételek kezelése**

Nem számít, mennyire egyszerű és intuitív az alkalmazás, előfordulhat, hogy a felhasználó rosszul használja, vagy csak véletlenszerű csatlakozási hibákat fog okozni, és a kódnak készen kell állnia ezekre a forgatókönyvekre, hogy a felhasználói élmény megfelelő legyen. Ezeket a forgatókönyveket kivételeknek nevezzük: a nyelv olyan eleme, amely azonosít egy olyan esetet, amely nem úgy történik, ahogyan azt vártuk.

**A try…catch blokk**

A kódunk manuálisan adhat kivételeket, amikor szükségesnek tartjuk. Vegyük például a setId metódust az Unique tulajdonságból. A típusjelzésnek köszönhetően az id-t numerikusra állítjuk, de ez a lényeg. Mi történne, ha valaki olyan azonosítót próbálna beállítani, amely negatív szám? A kód jelenleg lehetővé teszi, hogy átmenjen, de preferenciáitól függően szeretnénk elkerülni. Ez egy jó hely lenne a kivételkezelésre. Nézzük meg, hogyan adjuk hozzá ezt az ellenörzést és az ebből következő kivételt:

public function setId($id) {

if ($id < 0) {

**throw new \Exception('Id cannot be negative.');**

}

if (empty($id)) {

$this->id = ++self::$lastId;

} else {

$this->id = $id;

if ($id > self::$lastId) {

self::$lastId = $id;

}

}

}

Mint látható, a kivételek az osztálykivétel objektumai. Ne felejtse el hozzáírni a fordított perjelet az osztály nevéhez. Az Exception osztály konstruktora felvesz néhány opcionális argumentumot, ezek közül az első a kivétel üzenete. A Kivétel osztály példányai önmagukban semmit sem tesznek; kezelni kell, hogy a program észrevegye.

Próbáljuk rákényszeríteni a programunkat, hogy dobja ezt a kivételt. Ennek érdekében próbáljunk meg negatív azonosítóval rendelkező ügyfelet létrehozni. Az init.php fájlban adjuk hozzá a következőket:

$basic = new Basic(-1, "name", "surname", "email");

Ha most kipróbálja a böngészőjében, a PHP végzetes hibát fog jelezni, mondván, hogy egy nem elkapott kivétel történt, ami a várt viselkedés. A PHP esetében kivétel az, amit nem tud helyreállítani, ezért leállítja a végrehajtást. Ez távolról sem ideális, mivel csak egy hibaüzenetet szeretne megjeleníteni a felhasználónak, és hagyja, hogy újra próbálkozzon.

A try…catch blokkokkal rögzítheti – és kell is rögzíteni–a kivételeket. Beillesztjük a kódot, amely kivételt dobhat a try blokkba, és ha kivétel történik, a PHP a catch blokkra ugrik. Lássuk, hogyan működik:

public function setId(int $id) {

**try {**

if ($id < 0) {

throw new Exception('Id cannot be negative.');

}

if (empty($id)) {

$this->id = ++self::$lastId;

} else {

$this->id = $id;

if ($id > self::$lastId) {

self::$lastId = $id;

}

**}**

**} catch (Exception $e) {**

**echo $e->getMessage();**

**}**

}

Ha a böngészőnkben teszteljük az utolsó kódrészletet, akkor a catch blokkból kiirt üzenetet fogjuk látni. A getMessage metódus kivételes példányon történő meghívása megkapja az üzenetet – az első argumentumot az objektum létrehozásakor. De ne feledje, hogy a konstruktor argumentuma nem kötelező; ezért ne hagyatkozzon túlságosan a kivétel üzenetére, ha nem biztos benne, hogyan jön létre, mert lehet, hogy üres.

Vegye figyelembe, hogy a kivétel kidobása után semmi más nem kerül végrehajtásra a try blokkon belül; A PHP egyenesen a catch blokkhoz megy. Ezenkívül a blokk argumentumot kap, ami a kivétel. Itt kötelező a típusjelzés – hamarosan meglátod, miért. Az argumentum $e-nek való elnevezése egy széles körben használt konvenció, bár nem jó gyakorlat, a rossz leíró nevek használata a változókhoz.

Kicsit kritikus lévén, ebben a példában a kivételek alkalmazásából eddig semmiféle valódi előny nem látható. Egy egyszerű, if…else pontosan ugyanazt a munkát végezné, igaz? A kivételek valódi ereje azonban a módszerek közötti terjesztés képességében rejlik. Ez azt jelenti, hogy a setId metódusra dobott kivétel, ha nem kerül rögzítésre, a metódus meghívásának helyére terjed, lehetővé téve számunkra, hogy ott rögzítsük. Ez nagyon hasznos, mivel a kódunk különböző helyei eltérő módon kívánhatják kezelni a kivételt. Hogy megtudjuk, hogyan történik ez, távolítsuk el a setId-be beszúrt try…catch-et, és helyette helyezzük el a következő kódrészletet az init.php fájlba:

try {

$basic = new Basic(-1, "name", "surname", "email");

} catch (Exception $e) {

echo 'Something happened when creating the basic customer: '

. $e->getMessage();

}

Az előző példa azt mutatja, hogy mennyire hasznos a továbbított kivételek elkapása: pontosabban határozhatjuk meg, hogy mi történik, hiszen tudjuk, hogy a felhasználó mit akart tenni a kivétel kidobásakor. Ebben az esetben tudjuk, hogy megpróbáltuk létrehozni az ügyfelet, de előfordulhat, hogy ez a kivétel egy meglévő ügyfél azonosítójának frissítésekor történt, amihez más hibaüzenetre lenne szükség.

**A finaly elem**

Van egy harmadik elem, amelyet a kivételek kezelésére használunk: az finaly blokk. Ez az elem a try…catch után kerül hozzáadásra, és nem kötelező. Valójában a catch is opcionális; a korlátozás az, hogy egy próbálkozást legalább az egyiknek követnie kell. Tehát a következő három forgatókönyv lehet:

// scenario 1: the whole try-catch-finally

try {

// code that might throw an exception

} catch (Exception $e) {

// code that deals with the exception

} finally {

// finally block

}

// scenario 2: try-finally without catch

try {

// code that might throw an exception

} finally {

// finally block

}

// scenario 3: try-catch without finally

try {

// code that might throw an exception

} catch (Exception $e) {

// code that deals with the exception

}

A végleges blokkon belüli kód akkor kerül végrehajtásra, amikor vagy a try vagy a catch blokkok teljes végrehajtása megtörténik. Tehát, ha van egy forgatókönyvünk, ahol nincs kivétel, a try blokkon belüli összes kód végrehajtása után a PHP végrehajtja finally-ben lévő kódot. Másrészt, ha a try blokkon belül van kivétel, akkor a PHP a catch blokkra ugrik, és miután mindent végrehajtott ott, végrehajtja a finally blokkot is.

Ennek a funkciónak a teszteléséhez hajtsunk végre egy függvényt, amely egy try…catch…finly block-ot tartalmaz, megpróbál egy adott azonosítóval (argumentumon keresztül) ügyfelet létrehozni, és naplózza az összes végrehajtott műveletet. A következő kódrészletet hozzáadhatja az init.php fájlhoz:

function createBasicCustomer($id)

{

try {

echo "\nTrying to create a new customer.\n";

return new Basic($id, "name", "surname", "email");

} catch (Exception $e) {

echo "Something happened when creating the basic customer: "

. $e->getMessage() . "\n";

} finally {

echo "End of function.\n";

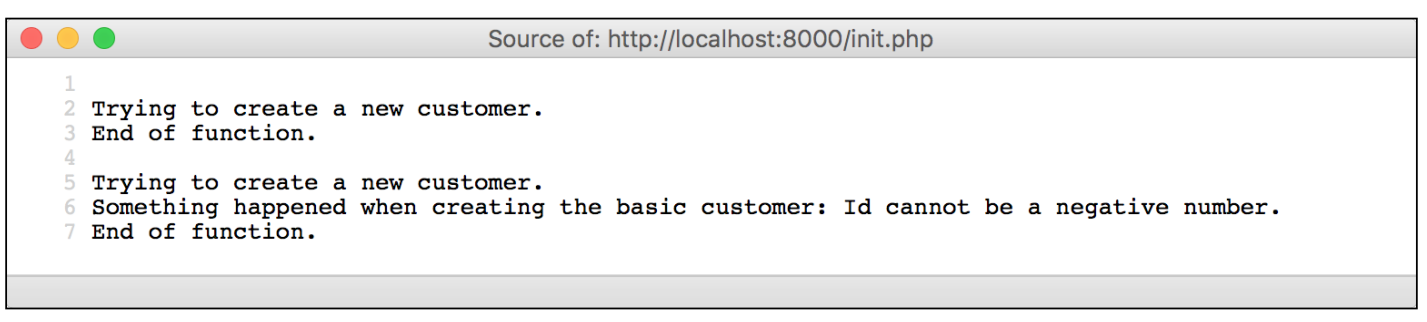
}

}

createBasicCustomer(1);

createBasicCustomer(-1);

Ha ezt kipróbálja, a böngészője a következő kimenetet mutatja – ne felejtse el megjeleníteni az oldal forráskódját, hogy szépen formázva lássa:



Lehet, hogy az eredmény nem az lesz, amire számítottunk. Amikor először hívjuk meg a függvényt, probléma nélkül létrehozhatjuk az objektumot, ami azt jelenti, hogy végrehajtjuk a return utasítást. Normál függvényben ennek a végén kell lennie, de mivel a try…catch…finally blokkon belül vagyunk, még mindig végre kell hajtanunk a finally kódot! A második példa intuitívabbnak tűnik, a try-tól a catch-ig , majd a végső blokkra ugrik.

Az finally blokk nagyon hasznos erőforrások, például adatbázis-kapcsolatok kezelésekor. Az 5. fejezetben (Adatbázisok használata) megtudhatja, hogyan kell használni őket. A kapcsolat típusától függően használat után be kell zárnia, hogy más felhasználók is csatlakozhassanak. Az utolsó blokk a kapcsolatok lezárására szolgál, függetlenül attól, hogy a függvény kivételt dob-e vagy sem.

**Könyvészet**

1. Antonio Lopez, Learning PHP 7, PACKT PUBLISHING