Web programozás projekt

*Osztályok és objektumok*

Kelemen Boglárka, Gazdasági informatika

2022

Az objektumok valós elemek reprezentációi. Minden objektumnak van egy sor attribútuma amely megkülönbözteti az azonos osztályba tartozó többi objektumtól, és képes a

cselekvések halmazára. Az osztály annak meghatározása, hogy egy objektum hogyan néz ki és mire képes, mint egy minta az objektumok számára.

Vegyük a könyvesboltos példát, és gondoljunk arra, hogy milyen valós tárgyakat tartalmaz.

Könyveket tárolunk, és az emberek elvihetik őket, ha éppen szabadok. Gondolhatunk két

típusú tárgyakra: könyvekre és vásárlókra. Ezt a két osztályt a következőképpen definiálhatjuk:

<?php

class Book {

}

class Customer {

}

Egy osztályt a class kulcsszóval definiálhatunk, amelyet egy érvényes osztálynév követ - ez a kulcsszó ugyanazokat a szabályokat követi, mint bármely más PHP-címke, például a változók nevei, és egy kódrészlet. De ha egy konkrét könyvet, azaz egy Book objektumot - vagy példányt - szeretnénk. a Book osztályból -, akkor azt inicializálnunk kell. Egy objektum példányosításához használjuk a new kulcsszót használjuk, amelyet az osztály neve követ. A példányt egy változóhoz rendeljük, mintha egy primitív típus lenne:

$book = new Book();

$customer = new Customer();

Annyi példányt hozhatunk létre, amennyire szüksége van, amennyiben a következőhöz rendeli őket különböző változókhoz:

$book1 = new Book();

$book2 = new Book();

Az osztály tulajdonságai:

Gondoljunk először a könyvek tulajdonságaira: van egy címük, egy szerzőjük, és egy

ISBN. Lehetnek elérhetőek illetve nem elérhetőek is. Írjuk a következő kódot a könyv belsejébe:

**Book.php file:**

<?php

class Book {

public $isbn;

public $title;

public $author;

public $available;

}

Ez az előző részlet egy olyan osztályt definiál, amely egy könyv tulajdonságait reprezentálja.

Ne foglalkozzunk a public szóval; majd elmagyarázzuk, hogy mit jelent, amikor beszélünk róla. láthatóságról beszélünk a következő szakaszban. Egyelőre csak gondoljunk a tulajdonságokra úgy, mint változókra a következőn belül az osztályban. Ezeket a változókat objektumokban használhatjuk. Próbáljuk meg a következő kódot hozzáadni a

**Book.php file:**

$book = new Book();

$book->title = "1984";

$book->author = "George Orwell";

$book->available = true;

var\_dump($book);

Az objektum kiíratásakor az objektum minden egyes tulajdonságának értéke megjelenik, hasonlóan ahhoz, ahogyan a tömbök a kulcsokkal. Láthatjuk, hogy a tulajdonságoknak van egy típusuk a kiíratás pillanatában, de ezt a típust nem definiáltuk explicit módon; ehelyett a változó típusát a a hozzárendelt értékét. Ez pontosan ugyanúgy működik, mint a normál változók.

Amikor egy objektum több példányát hozza létre, és értékeket rendel a tulajdonságaikhoz, minden objektumnak sajátértékei lesznek, így nem írja felül őket. A következő kódrészlet bemutatja, hogyan működik ez:

$book1 = new Book();

$book1->title = "1984";

$book2 = new Book();

$book2->title = "To Kill a Mockingbird";

var\_dump($book1, $book2);

**Osztály metódusok:**

A metódusok egy osztályon belül definiált függvények. A függvényekhez hasonlóan a metódusok is kapnak néhány argumentumokat, és végrehajtanak bizonyos műveleteket, opcionálisan visszaadva egy értéket. A metódusok előnye ,hogy használhatják az őket meghívó objektum tulajdonságait. Így például, két különböző objektumban ugyanazt a metódust meghívva két különböző eredményt kaphatunk.

Még akkor is, ha általában rossz ötlet a HTML-t és a PHP-t keverni, a tanulás kedvéért,

adjunk hozzá egy metódust a Book osztályokhoz, amely visszaadja a könyvet, mint a már meglévő

függvényt:

**printableTitle:**

<?php

class Book {

public $isbn;

public $title;

public $author;

public $available;

public function getPrintableTitle(): string {

$result = '*' . $this->title*

*. '* - ' . $this->author;

if (!$this->available) {

$result .= ' **Not available**';

}

return $result;

}

}

A tulajdonságokhoz hasonlóan a public kulcsszót a függvény elejére tesszük,

de ettől eltekintve a többi pont úgy néz ki, mint egy normál függvény. A másik különleges bit

a $this használata: ez képviseli magát az objektumot, és lehetővé teszi, hogy hozzáférjünk a

objektum tulajdonságaihoz és metódusaihoz. Figyeljük meg, hogyan hivatkozunk a címre, szerzőre, és az elérhető tulajdonságokra.

Az aktuális objektum értékeit az egyik függvényből is frissítheti. Használjuk a rendelkezésre álló tulajdonságot egész számként, amely az egységek számát mutatja rendelkezésre álló egységet mutatja ahelyett, hogy csak egy Boolean lenne. Ezzel lehetővé tehetjük, hogy több ügyfél ugyanannak a könyvnek különböző példányait kölcsönözni. Adjunk hozzá egy metódust, amellyel egy példányt adhatunk egy könyv egy példányát egy ügyfélnek, frissítve a rendelkezésre álló példányok számát:

public function getCopy(): bool {

if ($this->available < 1) {

return false;

} else {

$this->available--;

return true;

}

}

Ebben az előző módszerben először azt ellenőrizzük, hogy van-e legalább egy elérhető egységünk. Ha nincs, false-t adunk vissza, hogy tudassuk velük, hogy a művelet nem volt sikeres. Ha van egy egység az ügyfél számára, akkor csökkentjük a rendelkezésre álló egységek számát, majd true-t adunk vissza, tudtára adva, hogy a művelet sikeres volt. Lássuk, hogyan

használhatjuk ezt az osztályt:

<?php

$book = new Book();

$book->title = "1984";

$book->author = "George Orwell";

$book->isbn = 9785267006323;

$book->available = 12;

if ($book->getCopy()) {

echo 'Here, your copy.';

} else {

echo 'I am afraid that book is not available.';

}

Mit írna ki ez az utolsó kódrészlet? Pontosan, itt, a te példányod. De mi lenne az?

a rendelkezésre álló tulajdonság értéke? 11 lenne, ami az eredménye a

a getCopy meghívásának eredménye.

**Az osztály konstruktorai**

Talán észrevetted, hogy a Book osztály példányosítása hosszúnak tűnik, és a

minden egyes alkalommal be kell állítani az összes értékét. Mi lenne, ha az osztályunknak négy helyett 30 tulajdonsága lenne? Nos, remélhetőleg ezt soha nem fogod megtenni, mivel ez nagyon rossz gyakorlat. Mégis, van egy módja annak, hogy lerövidítsük: a konstruktorok.

A konstruktorok olyan függvények, amelyeket akkor hívunk meg, amikor valaki egy új példányt hoz létre az osztályból. Úgy néznek ki, mint a normál metódusok, azzal a különbséggel, hogy a nevük mindig \_\_construct, és hogy nincs return utasításuk, mivel mindig az új példányt kell visszaadniuk. Lássunk egy példát:

public function \_\_construct(int $isbn, string $title, string $author, int $available) {

$this->isbn = $isbn;

$this->title = $title;

$this->author = $author;

$this->available = $available;

}

A konstruktor négy argumentumot fogad el, majd hozzárendeli az egyik argumentum értékét a példány minden egyes tulajdonságához. A Book osztály példányosítása, a következőket használjuk:

$book = new Book("1984", "George Orwell", 9785267006323, 12);

Ez az objektum pontosan megegyezik azzal az objektummal, amelyiknek minden egyes tulajdonságainak értékét kézzel állítjuk be.De ez tisztábbnak tűnik, nem igaz? Ez nem jelenti azt, hogy nem lehet manuálisan új értékeket beállítani ehhez az objektumhoz, ez csak segít a felépítésben, új objektumok létrehozásában.

Mivel a konstruktor még mindig függvény, használhat alapértelmezett argumentumokat. Képzeljük el, hogy az egységek száma általában 0 lesz az objektum létrehozásakor, és később a könyvtáros egységeket fog hozzáadni, amikor azok rendelkezésre állnak. Beállíthatnánk egy alapértelmezett értéket a $available argumentumot a konstruktorban, így ha nem küldjük el az egységek számát a létrehozáskor, akkor a az objektumot, az alapértelmezett értékkel lesz példányosítva:

public function \_\_construct(

int $isbn,

string $title,

string $author,

int $available = 0

) {

$this->isbn = $isbn;

$this->title = $title;

$this->author = $author;

$this->available = $available;

}

Az előző konstruktort kétféleképpen használhatjuk:

$book1 = new Book("1984", "George Orwell", 9785267006323, 12);

$book2 = new Book("1984", "George Orwell", 9785267006323);

$book1 a rendelkezésre álló egységek számát 12-re állítja, míg $book2 a következő értéket állítja be alapértelmezett értéke 0. De ne bízz bennem; próbáld ki magad!

**Mágikus metódusok:**

Van a metódusoknak egy speciális csoportja, amelyek a normál metódusoktól eltérő viselkedéssel rendelkeznek. mint a többi. Ezeket a metódusokat magic metódusoknak nevezzük, és általában az alábbi parancsok váltják ki őket az osztály vagy az objektum interakciója, és nem a meghívás. Már láttad a az egyiket, az osztály konstruktorát, a \_\_construct-ot. Ez a metódus nem hívódik meg közvetlenül, hanem egy új példány létrehozásakor használjuk a new segítségével. Könnyedén azonosítani a mágikus metódusokat, mert \_\_-vel kezdődnek. Az alábbiakban néhányat mutatunk be a leggyakrabban használt mágikus módszerek:

- \_\_toString: Ezt a metódust akkor hívjuk meg, amikor egy objektumot egy

stringre. Nem fogad el paramétereket, és várhatóan egy karakterláncot ad vissza.

- \_\_hívás: Ez az a metódus, amelyet a PHP akkor hív meg, amikor megpróbálsz meghívni egy metódust egy nem létező osztályon. Megkapja a metódus nevét egy karakterláncként.

és a meghíváshoz használt paraméterek listáját tömbként, a argumentumon keresztül.

- \_\_get: Ez a \_\_call egy változata a tulajdonságok számára. Megkapja a

tulajdonságot, amelyet a felhasználó a paramétereken keresztül próbált elérni, és ez lehet

bármit visszaadhat

Használhatja a \_\_toString metódust a jelenlegi getPrintableTitle helyettesítésére

metódust a Book osztályunkban. Ehhez csak a metódus nevét kell megváltoztatnunk a következőképpen:

public function \_\_toString() {

$result = '*' . $this->title . '* - ' . $this->author;

if (!$this->available) {

$result .= ' **Not available**';

}

return $result;

}

Az előző kód kipróbálásához egyszerűen hozzáadhatja a következő részletet, amely létrehoz egy book objektumot, majd egy karakterlánccá alakítja azt, meghívva a \_\_toString metódust:

$book = new Book(1234, 'title', 'author');

$string = (string) $book; // title - author Not available

Ahogy a neve is mutatja, ezek mágikus módszerek, így a legtöbbször a jellemzőik

varázslatosnak tűnnek. Nyilvánvaló okokból, mi személy szerint arra bátorítjuk a fejlesztőket, hogy használják a konstruktorokat és esetleg a \_\_toStringet, de legyünk óvatosak, hogy mikor használjuk a többit, mivel a a kódodat eléggé kiszámíthatatlanná teheted az azt nem ismerők számára.

**Tulajdonságok és metódusok láthatósága**

Eddig a Book osztályunkban definiált összes tulajdonság és metódus a következő címkével volt ellátva public. Ez azt jelenti, hogy bárki számára elérhetőek, pontosabban a

bárhonnan. Ezt nevezzük a tulajdonság vagy metódus láthatóságának, és háromféleképpen lehet

láthatóságnak három típusa van. A szigorúbbtól a kevésbé szigorúig terjedő sorrendben ezek a következők:

- privát: Ez a típus csak az azonos osztály tagjai számára engedélyezi a hozzáférést. Ha A és B a C osztály példányai, A hozzáférhet B tulajdonságaihoz és metódusaihoz.

- védett: Ez a típus lehetővé teszi az azonos osztály tagjainak elérését, és

példányaihoz, amelyek csak az adott osztályból öröklődnek. Láthatja az örökléssel a következő szakaszban.

- nyilvános: Ez a típus olyan tulajdonságra vagy metódusra utal, amely elérhető a

bárhol. Bármely osztály vagy általában az osztályon kívüli kód hozzáférhet hozzá.

Néhány példa bemutatásához először hozzunk létre egy második osztályt az alkalmazásunkban.

Ezt mentsük el a Customer.php fájlba:

<?php

class Customer {

private $id;

private $firstname;

private $surname;

private $email;

public function \_\_construct(

int $id,

string $firstname,

string $surname,

string $email

) {

$this->id = $id;

$this->firstname = $firstname;

$this->surname = $surname;

$this->email = $email;

}

}

Ez az osztály egy ügyfelet reprezentál, és tulajdonságai az alábbi általános információkból állnak amelyeket a könyvesboltok általában ismernek a vásárlóikról. Biztonsági okokból azonban nem adhatjuk ki mindenkinek a vásárlóink személyes adatait, ezért minden egyes tulajdonságot privátnak állítjuk be.

Eddig az objektumok létrehozásához szükséges kódot ugyanabban a Book.php fájlban adtuk hozzá, de mivel most már két osztályunk van, természetesnek tűnik, hogy az osztályokat meghagyjuk a saját fájlokban hagyjuk, és az objektumokat egy külön fájlban hozzuk létre és játszunk velük. Nevezzük el ezt init.php fájlnak. Egy adott osztály objektumainak példányosításához a PHP-nek szüksége van a következőkre tudnia kell, hogy hol van az osztály. Ehhez csak be kell csatolnunk a fájlt require\_once kapcsolóval.

<?php

require\_once \_\_DIR\_\_ . '/Book.php';

require\_once \_\_DIR\_\_ . '/Customer.php';

$book1 = new Book("1984", "George Orwell", 9785267006323, 12);

$book2 = new Book("To Kill a Mockingbird", "Harper Lee", 9780061120084, 2);

$customer1 = new Customer(1, 'John', 'Doe', 'johndoe@mail.com');

$customer2 = new Customer(2, 'Mary', 'Poppins', '[mp@mail.com](mailto:mp@mail.com)');

A fájlokat nem kell minden egyes alkalommal beillesztenie. Ha egyszer már beillesztette őket, a PHP tudja, hol találja meg az osztályokat, még akkor is, ha a kódod egy másik fájlban van.

**Konvenciók az osztályokhoz**

Amikor osztályokkal dolgozol, tudnod kell, hogy van néhány konvenciók, amelyeket mindenki igyekszik követni a tiszta kód biztosítása érdekében, amely könnyen karbantartható. A legfontosabbak a következők:

- Minden osztálynak egy olyan fájlban kell lennie, amelynek a neve megegyezik az osztály nevével .php kiterjesztéssel

- Az osztályok nevének CamelCase-ben kell lennie, azaz minden szónak

nagybetűvel kezdődik, majd a szó többi része kisbetűs

- Egy fájl csak egy osztály kódját tartalmazhatja.

- Egy osztályon belül először a tulajdonságokat kell elhelyezni, majd a

konstruktor, végül pedig a többi metódus.

Hogy megmutassuk, hogyan működik a láthatóság, próbáljuk ki a következő kódot:

$book1->available = 2; // OK

$customer1->id = 3; // Error!

Azt már tudjuk, hogy a Book osztály objektumainak tulajdonságai nyilvánosak, és a

ezért kívülről szerkeszthetők. De amikor megpróbáljuk megváltoztatni egy értéket a Customer,

a PHP panaszkodik, mivel a tulajdonságai privátak.

**Egységbezárás**

Amikor objektumokkal dolgozol, az egyik legfontosabb fogalom, amit ismernünk

és alkalmaznunk kell, az a egységbezárás. A egységbezárás az objektum adatait igyekszik csoportosítani metódusaival, hogy megpróbálja elrejteni az objektum belső struktúráját a világ többi része elől. Egyszerű szavakkal azt mondhatjuk, hogy akkor használunk egységbezárást, ha a objektum tulajdonságai magánjellegűek, és azok frissítésének egyetlen módja az, hogy a

nyilvános metódusokon keresztül.

A egységbezárás használatának oka az, hogy a fejlesztő számára könnyebbé tegye a

az osztály belső struktúráján változtatásokat anélkül, hogy közvetlenül befolyásolná az osztály külső kódját, amely az osztályt használja. Képzeljük el például, hogy a Customer osztályunk, amely most már rendelkezik két tulajdonsággal, és ez határozza meg a nevét - keresztnév és vezetéknév -, meg kell változtatnunk. A oldalról mostantól csak egy tulajdonság nevünk van, amely mindkettőt tartalmazza. Ha elérnénk a tulajdonságaihoz, azonnal meg kellene változtatnunk az összes ilyen elérést!

Ehelyett, ha a tulajdonságokat privátnak állítjuk be, és két nyilvános metódust engedélyezünk, getFirstname és getSurname, még ha meg is kell változtatnunk a belső szerkezetét a az osztályt, akkor csak a két metódus implementációját változtathatjuk meg - ami

csak egy helyen történik, és az osztályunkat használó kód többi részét ez nem fogja érinteni.

egyáltalán nem érinti. Ezt a koncepciót információelrejtésnek is nevezik.

Ezt az ötletet a legegyszerűbben úgy valósíthatjuk meg, hogy az osztály összes tulajdonságát úgy be állítjuk privátnak, és két metódust engedélyezünk minden egyes tulajdonsághoz: az egyik az aktuális értéket kapja meg (más néven getter), a másik pedig lehetővé teszi egy új érték beállítását (más néven setter). Ez legalábbis a legáltalánosabb és legegyszerűbb módja az adatok egységbezárásának.

De menjünk egy lépéssel tovább: amikor egy osztályt definiálunk, gondoljunk azokra az adatokra, amelyeket szeretnénk a felhasználó képes legyen megváltoztatni és lekérdezni, és csak a következő adatokhoz adjunk hozzá állítót és kapót ezekhez. Például az ügyfelek megváltoztathatják az e-mail címüket, de a nevüket nem, vezetéknevük és azonosítójuk ugyanaz marad, ha egyszer már létrehoztuk őket. Az új definíció a osztály definíciója a következőképpen nézne ki:

<?php

class Customer {

private $id;

private $name;

private $surname;

private $email;

public function \_\_construct(

int $id,

string $firstname,

string $surname,

string $email

) {

$this->id = $id;

$this->firstname = $firstname;

$this->surname = $surname;

$this->email = $email;

}

public function getId(): id {

return $this->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;

}

}

Másrészt a könyveink is szinte változatlanok maradnak. Az egyetlen lehetséges változás

a rendelkezésre álló egységek száma. De általában egyszerre csak egy könyvet veszünk vagy adunk hozzá, ahelyett, hogy beállítanánk a rendelkezésre álló egységek konkrét számát, így a setter itt nem igazán hasznos. Már van a getCopy metódusunk, amely lehetőség szerint egy példányt vesz át; legyen adjunk hozzá egy addCopy metódust, valamint a többi gettert:

<?php

class Book {

private $isbn;

private $title;

private $author;

private $available;

public function \_\_construct(

int $isbn,

string $title,

string $author,

int $available = 0

) {

$this->isbn = $isbn;

$this->title = $title;

$this->author = $author;

$this->available = $available;

}

public function getIsbn(): int {

return $this->isbn;

}

public function getTitle(): string {

return $this->title;

}

public function getAuthor(): string {

return $this->author;

}

public function isAvailable(): bool {

return $this->available;

}

public function getPrintableTitle(): string {

$result = '*' . $this->title . '* - ' . $this->author;

if (!$this->available) {

$result .= ' **Not available**';

}

return $result;

}

public function getCopy(): bool {

if ($this->available < 1) {

return false;

} else {

$this->available--;

return true;

}

}

public function addCopy() {

$this->available++;

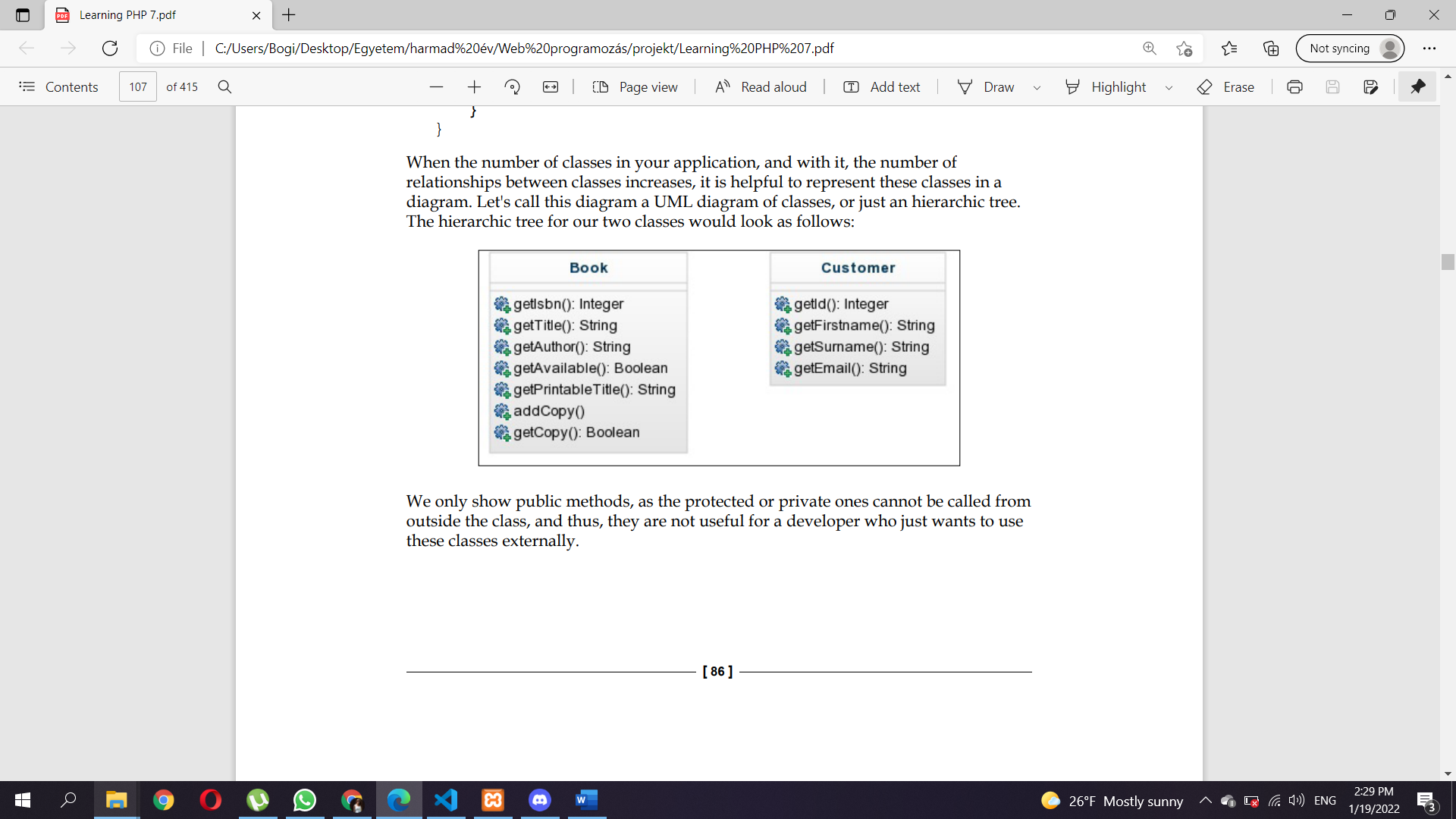
}

}

Amikor az alkalmazásban lévő osztályok száma, és ezzel együtt a

osztályok közötti kapcsolatok száma növekszik, hasznos ezeket az osztályokat egy

diagramon. Nevezzük ezt a diagramot az osztályok UML-diagramjának, vagy egyszerűen csak hierarchikus fának.A két osztályunk hierarchikus fája a következőképpen nézne ki:



Csak a nyilvános metódusokat mutatjuk meg, mivel a védett vagy privát metódusok nem hívhatók meg a az osztályon kívülről, és így nem hasznosak egy olyan fejlesztő számára, aki csak külsőleg szeretné használni ezeket az osztályokat.

**Statikus tulajdonságok és metódusok**

Eddig az összes tulajdonság és metódus egy adott példányhoz volt kötve; így két

különböző példány két különböző értéket kaphattak ugyanarra a tulajdonságra. A PHP lehetővé teszi

hogy a tulajdonságok és metódusok ne az objektumhoz, hanem magához az osztályhoz kapcsolódjanak. Ezeket a tulajdonságokat és metódusokat a static kulcsszóval definiáljuk.

private static $lastId = 0;

Adja hozzá az előző tulajdonságot a Customer osztályhoz. Ez a tulajdonság az utolsó azonosítót mutatja felhasználóhoz rendelt azonosítót, és hasznos ahhoz, hogy tudjuk, hogy milyen azonosítót kell rendelni egy felhasználóhoz, új felhasználóhoz kell rendelni. Módosítsuk az osztályunk konstruktorát a következőképpen:

public function \_\_construct(

int $id,

string $name,

string $surname,

string $email

) {

if ($id == null) {

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

} else {

$this->id = $id;

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

self::$lastId = $id;

}

}

$this->name = $name;

$this->surname = $surname;

$this->email = $email;

}

Vegyük észre, hogy amikor egy statikus tulajdonságra hivatkozunk, nem használjuk a $this változót. Ehelyett a self::-t használjuk, amely nem egy példányhoz, hanem magához az osztályhoz kötődik. A oldalon ez utóbbi konstruktorban két lehetőségünk van. Vagy kapunk egy azonosító értéket ami nem null, vagy egy nullát küldünk helyette. Ha a kapott azonosító nulla, akkor a $lastId statikus tulajdonságot, hogy megismerjük az utoljára használt azonosítót, növeljük eggyel, és hozzárendeljük a a $id tulajdonsághoz. Ha az utoljára beillesztett azonosító 5 volt, akkor ez frissíti a statikus tulajdonságot 6-ra, majd hozzárendeljük a példány tulajdonsághoz. Legközelebb, amikor létrehozunk egy új ügyfelet, a $lastId statikus tulajdonság 6 lesz. Ehelyett, ha egy érvényes azonosítót kapunk részeként az argumentumok között, hozzárendeljük, és ellenőrizzük, hogy a hozzárendelt $id nagyobb-e, mint a statikus $lastId. Ha igen, akkor frissítjük. Lássuk, hogyan használnánk ezt:

$customer1 = new Customer(3, 'John', 'Doe', 'johndoe@mail.com');

$customer2 = new Customer(null, 'Mary', 'Poppins', 'mp@mail.com');

$customer3 = new Customer(7, 'James', 'Bond', '007@mail.com');

Az előző példában $customer1 azt adja meg, hogy az azonosítója 3, valószínűleg azért, mert már meglévő ügyfél, és szeretné megtartani ugyanazt az azonosítót. Ez beállítja mind az azonosítóját, mind a az utolsó statikus azonosítót 3-ra. A második ügyfél létrehozásakor nem adjuk meg az azonosítót, így a konstruktor az utolsó azonosítót veszi, megnöveli 1-gyel, és hozzárendeli az ügyfélhez. Így $customer2 azonosítója 4 lesz, és a legutóbbi azonosító is 4 lesz. Végül a titkos ügynökünk tudja, hogy mit akar, ezért rákényszeríti a rendszert, hogy az ID 7 legyen. A legutóbbi ID szintén 7-re frissül.

A statikus tulajdonságok és metódusok másik előnye, hogy nincs szükségünk objektumra, hogy használatukhoz. Egy statikus tulajdonságra vagy metódusra úgy hivatkozhatunk, hogy megadjuk a nevét a osztály, majd a :: és a tulajdonság/módszer nevének megadása után. Ez természetesen akkor van így, ha a láthatósági szabályok ezt lehetővé teszik, ami ebben az esetben nem lehetséges, mivel a tulajdonság a private. Adjunk hozzá egy nyilvános statikus metódust az utolsó azonosító lekérdezésére:

public static function getLastId(): int {

return self::$lastId;

}

Hivatkozhatsz rá az osztály nevével vagy egy létező példányával, bárhol a kódban:

Customer::getLastId();

$customer1::getLastId();

**Névterek**

Tudja, hogy nem lehet két azonos nevű osztálya, mivel a PHP

nem tudná, hogy melyikre hivatkozik egy új objektum létrehozásakor. Ennek megoldására a PHP lehetővé teszi a névterek használatát, amelyek úgy működnek, mint a fájlrendszerben az elérési utak. Ebben a esetben annyi azonos nevű osztályunk lehet, amennyire csak szükségünk van, amíg a mind különböző névterekben vannak definiálva. Érdemes megjegyezni, hogy bár a

névterek és a fájl elérési útvonala általában megegyezik, ezt a fejlesztő, nem pedig a nyelv; valójában bármilyen névteret használhat, amelyik nincs köze a fájlrendszerhez.

A névtér megadása az első dolog, amit egy fájlban meg kell tennie. Ahhoz, hogy a

a namespace kulcsszót használja, amelyet a névtér követ. Minden egyes szakasza a

névteret \-vel választjuk el, mintha egy másik könyvtár lenne. Ha nem adja meg a

a névteret, az osztály az alap névtérhez, vagy a gyökérhez fog tartozni. Az elején

mindkét fájl - a Book.php és a Customer.php - elejére a következőket írjuk:

<?php

namespace Bookstore\Domain;

Az előző kódsor az osztályaink névterét Bookstore\Domain néven állítja be.

Az osztályaink teljes neve így Bookstore\Domain\Book és Bookstore\ lesz.Domain\Customer. Ha megpróbáljuk elérni az init.php fájlt a böngészőnkből, akkor a következőket kapjuk

hibaüzenetet fog kapni, miszerint a Book vagy a Customer osztály nem található.

De ugye beillesztettük a fájlokat? Ez azért történik, mert a PHP azt hiszi, hogy

hogy a \Book és a \Customer fájlokat a gyökérből próbálja elérni. Ne aggódjon, többféle

mód van ennek módosítására.

Az egyik megoldás az lenne, ha az osztályok teljes nevét megadnánk a hivatkozáskor.

azaz a $customer = new Bookstore\Domain\Book(); helyett a $customer = new Bookstore\Domain\Book(); használatával.

helyett $book = new Book();. De ez nem hangzik praktikusan, ugye?

Egy másik lehetőség az lenne, ha azt mondanánk, hogy az init.php fájl a BookStore \

Domain névtérhez tartozik. Ez azt jelenti, hogy az init.php fájlban található összes osztályra való hivatkozás a BookStore\Domain előtaggal lesz ellátva, és használhatja a Book és Customer. Ennek a megoldásnak az a hátránya, hogy nem tudunk egyszerűen hivatkozni a más osztályok más névterekből, mivel az osztályokra történő hivatkozásoknak előtagja lesz az adott névtérrel.

A legjobb megoldás a use kulcsszó használata. Ez a kulcsszó lehetővé teszi, hogy megadja egy teljes osztálynevet a fájl elején, majd az egyszerű nevet használhatja a osztály nevét használja a fájl többi részében. Lássunk egy példát:

<?php

use Bookstore\Domain\Book;

use Bookstore\Domain\Customer;

require\_once \_\_DIR\_\_ . '/Book.php';

require\_once \_\_DIR\_\_ . '/Customer.php';

//…

Az előző fájlban minden egyes alkalommal, amikor a Book vagy a Customer hivatkozásra hivatkozunk, a PHP tudni fogja, hogy hogy valójában a teljes osztálynevet akarjuk használni, azaz Bookstore\Domain\előtaggal. Ez a megoldás lehetővé teszi, hogy tiszta kódot kapjunk, amikor hivatkozunk ezekre a, ugyanakkor más névterek osztályaira is tudunk hivatkozni ha szükséges.

De mi van akkor, ha két különböző, azonos nevű osztályt akarunk felvenni a

fájlba? Ha két use utasítást állít be, a PHP nem fogja tudni, hogy melyiket válassza,

így továbbra is ugyanazzal a problémával állunk szemben, mint korábban! Ezt úgy oldhatjuk meg, hogy vagy a teljes osztályt használjuk

nevet - névtérrel együtt - minden alkalommal, amikor bármelyik osztályra hivatkozni akarunk, vagy pedig aliasokat használunk.

Képzeljük el, hogy van két Book osztályunk, az első a Bookstore\ névtérben.

Domain és a második a Library\Domainben. A konfliktus feloldásához a következőt tehetjük

a következőképpen:

use Bookstore\Domain\Book;

use Library\Domain\Book as LibraryBook;

Az as kulcsszó az adott osztály aliasát állítja be. Ebben a fájlban minden alkalommal, amikor hivatkozik a LibraryBook osztályra hivatkozol, valójában a Library\Domain\Book osztályra fogsz hivatkozni. Amikor pedig a Book osztályra hivatkozik, a PHP a Bookstore osztályra fog hivatkozni. Probléma megoldva!

**Autoloading osztályok**

Mint már tudod, ahhoz, hogy egy osztályt használhass, be kell építened a fájlt, amely definiálja. Eddig manuálisan építettük be a fájlokat, mivel csak néhány osztályunk volt

és egy fájlban használtuk őket. De mi történik akkor, ha több osztályt használunk több

fájlokban? Kell lennie egy okosabb módszernek, nem? Valóban van. Az automatikus betöltés a megmentő!

Az automatikus betöltés egy PHP funkció, amely lehetővé teszi a program számára, hogy a fájlokat keresse és betöltse automatikusan, előre meghatározott szabályok alapján. Minden alkalommal, amikor hivatkozik egy osztályra amelyről a PHP nem tud, megkérdezi az automatikus betöltőt. Ha az autoloader ki tudja találni hogy melyik fájlban van az osztály, betölti azt, és a program végrehajtása

a szokásos módon folytatódik. Ha nem, akkor a PHP leállítja a végrehajtást.

Tehát, mi az az autoloader? Ez nem más, mint egy PHP függvény, amely megkapja az osztály nevét. paraméterként kap, és elvárható, hogy betöltsön egy fájlt. A megvalósításnak két módja van autoloader: vagy az \_\_autoload függvény vagy a spl\_autoload\_register segítségével.

Az \_\_autoload függvény használata

Egy \_\_autoload nevű függvény definiálása azt mondja a PHP-nek, hogy ez a függvény az automatikus betöltő amelyet használnia kell. Egy egyszerű megoldást is megvalósíthatsz:

function \_\_autoload($classname) {

$lastSlash = strpos($classname, '\\') + 1;

$classname = substr($classname, $lastSlash);

$directory = str\_replace('\\', '/', $classname);

$filename = \_\_DIR\_\_ . '/' . $directory . '.php';

require\_once($filename);

}

Az a szándékunk, hogy minden PHP fájlt az src-ben, azaz a forrásban tartsunk. Ebben a könyvtárban, a könyvtárfa az osztályok névtérfáját fogja utánozni, kivéve az első BookStore szakasz kivételével, amely névtérként hasznos, de könyvtárként nem szükséges. Ez azt jelenti, hogy a Book osztályunk, a teljes BookStore\Domain\Book osztálynévvel, a következő lesz

az src/Domain/Book.php állományban lesz.

Ennek érdekében a \_\_autoload függvényünk megpróbálja megtalálni az első előfordulású

a backslash \ strpos segítségével, majd ettől a pozíciótól kezdve a végéig a substr. Ez a gyakorlatban csak a névtér első szakaszát, a BookStore-t távolítja el. Ezután minden \-t /-re cserélünk, hogy a fájlrendszer megértse az elérési utat. Végül az aktuális könyvtárat, az osztály nevét mint könyvtárat és a

.php kiterjesztést

Mielőtt ezt kipróbálnád, ne felejtsd el létrehozni az src/Domain könyvtárat és áthelyezni a

két osztályt ebbe a könyvtárba. Továbbá, hogy megbizonyosodjunk arról, hogy az autoloader-t teszteljük, mentsük el a a következőt init.php fájlként, és menjünk a http://localhost:8000/init.php oldalra:

<?php

use Bookstore\Domain\Book;

use Bookstore\Domain\Customer;

function \_\_autoload($classname) {

$lastSlash = strpos($classname, '\\') + 1;

$classname = substr($classname, $lastSlash);

$directory = str\_replace('\\', '/', $classname);

$filename = \_\_DIR\_\_ . '/src/' . $directory . '.php' require\_once($filename);

}

$book1 = new Book("1984", "George Orwell", 9785267006323, 12);

$customer1 = new Customer(5, 'John', 'Doe', '[johndoe@mail.com](mailto:johndoe@mail.com)');

A böngésző most nem panaszkodik, és nincs explicit require\_once. Továbbá

ne feledjük, hogy a \_\_autoload függvényt csak egyszer kell definiálni, nem pedig minden egyes

fájlban. Tehát mostantól kezdve, amikor használni akarod az osztályaidat, amint az osztály a

egy olyan névtérben és fájlban van, amely követi a konvenciót, csak a use

utasítást. Sokkal tisztább, mint korábban, nem igaz?

A spl\_autoload\_register függvény használata

A \_\_autoload megoldás elég jónak tűnik, de van egy kis problémája: mi van, ha a mi

kódunk annyira összetett, hogy nem csak egy konvenciónk van, és több konvencióra van szükségünk egynél több implementációra van szükségünk a \_\_autoload függvényhez? Mivel nem definiálhatunk két függvényt ugyanazzal a névvel, szükségünk van arra, hogy a PHP-nek megmondjuk, hogy tartson egy listát a lehetséges megvalósításainak listáját, hogy mindegyiket kipróbálhassa, amíg az egyik nem működik.

Ez a spl\_autoload\_register feladata. Meghatározza az autoloader függvényt

egy érvényes névvel, majd meghívja a spl\_autoload\_register függvényt, és elküldi a

az autoloader nevét argumentumként. Ezt a függvényt annyiszor hívhatja meg, ahány

annyiszor, ahány különböző autoloadere van a kódjában. Valójában még akkor is, ha van

csak egy autoloader, ennek a rendszernek a használata még mindig jobb megoldás, mint a \_\_autoload mint a másik, mivel megkönnyíted a dolgát annak, akinek később új autoloadert kell hozzáadnia:

function autoloader($classname) {

$lastSlash = strpos($classname, '\\') + 1;

$classname = substr($classname, $lastSlash);

$directory = str\_replace('\\', '/', $classname);

$filename = \_\_DIR\_\_ . '/' . $directory . '.php';

require\_once($filename);

}

spl\_autoload\_register('autoloader');

Öröklődés

Az objektumorientált paradigmát úgy mutattuk be, mint a komplex adatok csodaszerét.

struktúrákat, és bár megmutattuk, hogy definiálhatunk objektumokat tulajdonságokkal

és metódusokkal, és ez szépnek és díszesnek tűnik, ez nem olyasmi, amit nem tudnánk megoldani tömbökkel. A egységbezárásvolt az egyik olyan tulajdonság, amely az objektumokat hasznosabbá tette a tömböknél, de az igazi erejük az öröklésben rejlik.