Olio-ohjelmointi PHP:llä

- PHP:n nykyiset olio-ominaisuudet ovat muiden nykyaikaisten oliokielten tasolla.
- Tässä esitetään esimerkkien avulla keskeisimmät PHP:n olio-malliin liittyvät asiat. Oletuksena on, että osaat jo aiemmin perusteet olio-ohjelmoinnista jollakin kielellä.

Luokan määrittely ja olion käyttäminen

Esimerkki 1701

Määritellään luokka `Person` ja muodostetaan siitä olio `$ari` Ohjelma tulostaa "*Haloo, nimeni on Ari!*". PHP:ssä on normaalit näkyvyysmääreet *public*, *protected* ja *private*. Olion ominaisuuteen tai metodiin viitataan `->`-notaatiolla. Konstruktori merkitään avainsanalla `__construct`

```
1  <?php
2  / / person-olio.php
3
4  class Person {
5      private $name;
6
7      public function __construct($name) {
8          $this->name = $name;
9      }
10     public function print_name() {
11         echo "Terve, nimeni on {$this->name}! <br>";
12     }
13 }
14
15 $ari = new Person("Ari");
16 $ari->print_name();
17 ?>
```

Esimerkki 1702

Usein luokat kirjoitetaan omaan tiedostoonsa nimeämällä ne tyyliin `Luokka.class.php` Kirjoitetaan Player- ja Team-luokat pelaajien ja joukkueiden käsittelemistä varten. Huomaa: `protected`-määreitä tarvitaan jatkossa esitettävässä perintäesimerkissä.

Player.class.php

```php
<?php
// Player.class.php

class Player {
    protected $name;
    protected $nro;

    public function __construct($name, $nro) {
        $this->name = $name;
        $this->nro = $nro;
    }

    public function get_name() {
        return $this->name;
    }

    public function get_nro() {
        return $this->nro;
    }

}

?>
```

Team.class.php

```php
<?php
// Team.class.php

class Team {
    private $name;
    private $players = array();

    public function add($player) {
        $this->players[] = $player;
    }
```

```php

    public function playercount() {
        return count($this->players);
    }

    public function playerlist() {
        return $this->players;
    }
}

?>
```

teamplayers.php

Luodaan erikseen luokkia **Player** ja **Team** käyttävä ohjelma. Funktiolla `__autoload` voidaan tietyllä tavalla nimetyt luokat määritellä etukäteen automaattisesti ladattaviksi vasta sitten kun niitä tarvitaan.

```php
<?php
// teamplayers.php

// Tarvittavien luokkien automaattinen lataaminen tarvittaess
function __autoload($class_name) {
   require_once $class_name . '.class.php';
}

// Luokat voitaisiin ladata myös tyyliin:
// require_once("Player.class.php");


$joukkue = new Team();
$joukkue->add(new Player("Ari", 1));
$joukkue->add(new Player("Pasi", 2));

echo "Pelaajia: " . $joukkue->playercount() . " kpl<br>";

// Tulostetaan pelaajat pelinumeroineen
foreach ($joukkue->playerlist() as $pelaaja) {
    echo "Pelaaja: " . $pelaaja->get_name() .
              "[".  $pelaaja->get_nro() . "]<br>";
}

?>
```

Ohjelman tulostus on:

Pelaajia: 2 kpl
Pelaaja: Ari[1]
Pelaaja: Pasi[2]

Perintä

Peritään **Player**-luokan toiminnallisuus luotavalle **Golfplayer**-luokalle. **Golfplayer**-luokkaan lisätään pelaajan tasoitus eli HCP. Huomaa `protected`-määreet Player-luokassa, jotta ao. ominaisuudet voidaan periä. Lisäksi esitelty luokan erikoismetodi `__toString`, jota voidaan käyttää olion tulostamiseen merkkijonona viittaamatta tiettyyn metodiin.

Golfplayer.class.php

```php
<?php
// Golfplayer.class.php

require_once("Player.class.php");

class Golfplayer extends Player{
    private $hcp;

    public function __construct($name, $nro, $hcp) {
    parent::__construct($name, $nro);
        $this->hcp = $hcp;
    }

   function __toString() {
     return "<h4>$this->name [$this->nro]</h4>" .
            "<p>HCP: $this->hcp </p>";
   }
}
?>
```

golfplayers.php

Huomaa tässä `__toString()`-metodin käyttäminen riveillä 12 ja 13.

```php
<?php
// golfplayers.php

function __autoload($class_name) {
   require_once $class_name . '.class.php';
}

$golfplayer1 = new Golfplayer("Ari", 1, 54);
$golfplayer2 = new Golfplayer("Pasi", 2, 5);

// Käyttää luokan erikoismetodia __toString:
echo $golfplayer1;
echo $golfplayer2;

?>
```

Ohjelman tulostus on

Ari [1]
HCP: 54

Pasi [2]
HCP: 5

PHP:n olio-ominaisuuksia listattuna

Konstruktorit ja destruktorit

Määritellään seuraavasti.

```php
class Jokuluokka {
   function __construct() {
      echo "Konstruktoria suoritetaan";
   }
   function __destruct() {
      echo "Destruktoria suoritetetaan";
   }
}
```

Näkyvyysmääreet

- **public**: Käytettävissä kaikkialla
- **private**: Käytettävissä vain luokassa, jossa muuttuja on määritelty. Ei voi käyttää aliluokissa.
- **protected**: Käytettävissä siinä luokassa, jossa muuttuja on määritelty ja kaikissa aliluokissa

Luokkametodit ja luokkamuuttujat

Luokkametodi (eli staattinen metodi; static method, class method) tekee halutun toimenpiteen saamiensa parametrien perusteella ja palauttaa tuloksen. **Sen kutsua ei osoiteta millekään luokan oliolle**! Näin ollen luokkametodilla ei ole käytössä this-viittaustakaan.

Luokkamuuttuja (staattinen muuttuja; static variable, class variable) toimii luokkametodin tavoin. Luokkamuuttujalla ei voi olla erillisiä arvoja luokasta luotuja olioita kohden. Luokkamuuttujalla on vain yksi arvo.

```
1  class JokuLuokka {
2     static function tervehdys() {
3        echo "Terve vaan terve!";
4     }
5  }
6
7  JokuLuokka::tervehdys();
```

Abstraktit luokat ja metodit

- Abstrakteista luokista **ei voi** luoda sellaisenaan olioita.
- Ne kuvaavat vain aliluokkiensa yhteisiä muuttujia ja metodeja.
- Jos luokassa on yksikin abstrakti metodi, pitää myös asianomainen luokka määritellä abstraktiksi.

```
1  abstract class Astiat {
2     abstract function annaTilavuus();
3  }
```

Rajapinnat

- Rajapinta luettelee metodit, joille tulee löytyä toteutus asianomaisen rajapinnan toteuttavista luokista.
- Rajapinta ei sisällä ominaisuuksia eikä metodien toteutuksia.

- Erilaisille astioille tulee voida esimerkiksi laskea tilavuus, mutta tilavuuden laskeminen tapahtuu erimuotoisille astioille eri tavalla.
- PHP:ssa luokka voi periä vain yhden kantaluokan. Luokalta voidaan kuitenkin vaatia useiden rajapintojen toteutusta.

```
1  interface Astia {
2     function annaTilavuus();
3  }
4
5  class Tynnyri implements Astia {
6     function annaTilavuus($sade, $korkeus) {
7        return 3.14 * $sade * $sade * $korkeus;
8     }
```

Final-luokat ja metodit

Final-luokka on luokka, jota ei voi periä. Final-metodia ei voi määritellä uudelleen aliluokissa.

Esimerkki A

```
1  final class JokuFinalLuokka {
2     //...
3  }
4
5  // Tämä ei onnistu
6  class Peritty extends JokuFinalLuokka {
7  //...
8  }
```

Esimerkki B

```
1  class JokuLuokka {
2     final function annaTiedot() {
3        echo "Nimeni on $this->nimi";
4     }
5  }
6
7
8  class Peritty extends JokuLuokka {
9     //sama nimi, kun Final-metodilla, ei onnistu:
10    function annaTiedot() {
```

```
11          echo "Olen Tapani ja tapani on näyttää napani...";
12      }
13  }
```

Vakioarvot

Luokka voi sisältää vakioarvoja, joihin voidaan viitata luokan nimellä.

```
1  class JokuLuokka {
2      const VAKIO = "Vakiopaine";
3  }
4  echo JokuLuokka::VAKIO;
```

Ilmentymän toteaminen

instanceof-operaattorilla voidaan tarkistaa, onko joku olio tietyn luokan ilmentymä (instanssi)

```
1  if ($olio instanceof Tynnyri) {
2      echo '$olio on Tynnyri';
3  }
```

Olion kloonaaminen

Olio voidaan kloonata avainsanalla clone. Luokkaan voidaan määritellä myös `__clone()`-metodi, jota kutsutaan kloonauksen yhteydessä.

```
1  class Lammas {
2      function __clone() {
3          print "Lampaan kloonaus käynnissä";
4      }
5  }
6
7
8  $dolly = new Lammas();
9  dolly_copy = clone $dolly;
```

Tyyppivihjeet

Tyyppivihjeillä (type hints) voidaan esimerkiksi vaatia, että tietty metodi saa parametrinaan oikean tyyppisen luokan. Muussa tapauksessa kutsu aiheuttaa virheen.

```
1 function odotaJokuLuokka(JokuLuokka $olio) {
2  }
```

```
© #AriRantala
```