Pertemuan 10: Traits

Tujuan Pembelajaran

Setelah mengikuti pertemuan ini, mahasiswa diharapkan dapat:

1. Memahami konsep Traits dalam PHP
2. Menggunakan keyword `trait` dan `use` dengan benar
3. Mengatasi masalah multiple inheritance dengan Traits
4. Menangani method conflicts dalam Traits
5. Menggunakan `insteadof` dan `as` untuk conflict resolution
6. Memahami trait composition dan multiple trait usage
7. Menerapkan Traits untuk code reuse yang efektif

Konsep Traits

Definisi Traits

Traits adalah mekanisme untuk code reuse dalam single inheritance languages seperti PHP. Traits memungkinkan developer untuk reuse sets of methods secara bebas di beberapa independent classes.

Masalah yang Dipecahkan Traits

1. **Single Inheritance Limitation** - PHP hanya mendukung single inheritance
2. **Code Duplication** - Method yang sama di multiple classes
3. **Diamond Problem** - Konflik dalam multiple inheritance
4. **Horizontal Reuse** - Sharing functionality across unrelated classes

Karakteristik Traits

- **Not instantiable** - Tidak bisa dibuat object langsung dari trait
- **Code reuse mechanism** - Untuk berbagi method implementations
- **Composition over inheritance** - Menyusun functionality dari multiple sources
- **Conflict resolution** - Built-in mechanism untuk mengatasi conflicts
- **Priority rules** - Class methods override trait methods

Basic Trait Usage

Sintaks Dasar

```php
trait TraitName {
    public function method1() {
        return "Method from trait";
    }

    protected function method2($param) {
        // Implementation
    }
}

class MyClass {
    use TraitName;

    // Class dapat menggunakan methods dari trait
}

$obj = new MyClass();
echo $obj->method1(); // "Method from trait"
```

Trait Properties

```
trait PropertyTrait {
    protected $traitProperty = "default value";
    private $privateProperty;

    public function setTraitProperty($value) {
        $this->traitProperty = $value;
    }

    public function getTraitProperty() {
        return $this->traitProperty;
    }
}

class Example {
    use PropertyTrait;
}

$obj = new Example();
$obj->setTraitProperty("new value");
echo $obj->getTraitProperty(); // "new value"
```

Multiple Traits

Menggunakan Multiple Traits

```
trait TraitA {
    public function methodA() {
        return "Method A";
    }
}

trait TraitB {
    public function methodB() {
        return "Method B";
    }
}
```

```
trait TraitC {
    public function methodC() {
        return "Method C";
    }
}

class MultiTraitClass {
    use TraitA, TraitB, TraitC;
}

$obj = new MultiTraitClass();
echo $obj->methodA(); // "Method A"
echo $obj->methodB(); // "Method B"
echo $obj->methodC(); // "Method C"
```

Trait Composition

```
trait CompositeTrait {
    use TraitA, TraitB;

    public function compositeMethod() {
        return $this->methodA() . " + " . $this->methodB();
    }
}

class ComposedClass {
    use CompositeTrait;
}
```

Method Conflicts dan Resolution

Method Name Conflicts

```
trait TraitX {
    public function commonMethod() {
        return "From TraitX";
    }
}

trait TraitY {
    public function commonMethod() {
        return "From TraitY";
    }
}

// Error: Fatal error tanpa conflict resolution
class ConflictClass {
    use TraitX, TraitY {
        TraitX::commonMethod insteadof TraitY;  // Pilih TraitX
        TraitY::commonMethod as methodFromY;    // Alias untuk TraitY
    }
}

$obj = new ConflictClass();
echo $obj->commonMethod();  // "From TraitX"
echo $obj->methodFromY();   // "From TraitY"
```

Visibility Changes dengan `as`

```
trait VisibilityTrait {
    protected function protectedMethod() {
        return "Protected method";
    }

    private function privateMethod() {
        return "Private method";
    }

    public function publicMethod() {
        return "Public method";
    }
}

class VisibilityClass {
    use VisibilityTrait {
        protectedMethod as public;              // Ubah ke public
        privateMethod as protected altMethod;   // Ubah ke protected +
alias
        publicMethod as private;                // Ubah ke private
    }
}
```

Trait Inheritance

Trait Extending Trait

```
trait BaseTrait {
    public function baseMethod() {
        return "Base method";
    }

    protected function helper() {
        return "Helper from base";
    }
}

trait ExtendedTrait {
    use BaseTrait;

    public function extendedMethod() {
        return $this->baseMethod() . " + extended";
    }

    public function useHelper() {
        return $this->helper();
    }
}

class ExtendedClass {
    use ExtendedTrait;
}
```

Trait Method Precedence

```
class BaseClass {
    public function method() {
        return "From base class";
    }
}

trait OverrideTrait {
    public function method() {
        return "From trait";
    }
}

class DerivedClass extends BaseClass {
    use OverrideTrait;

    // Precedence: Current class > Trait > Parent class
    public function method() {
        return "From derived class";
    }
}

$obj = new DerivedClass();
echo $obj->method(); // "From derived class"
```

Abstract Methods dalam Traits

Abstract Methods

```
trait AbstractTrait {
    abstract public function requiredMethod();

    public function concreteMethod() {
        return "Concrete: " . $this->requiredMethod();
    }
}

class ConcreteClass {
    use AbstractTrait;

    // Harus implement abstract method
    public function requiredMethod() {
        return "Implementation provided";
    }
}
```

Static Methods dalam Traits

```
trait StaticTrait {
    public static function staticMethod() {
        return "Static method from trait";
    }

    public function instanceMethod() {
        return self::staticMethod();
    }
}

class StaticClass {
    use StaticTrait;
}

echo StaticClass::staticMethod(); // "Static method from trait"
```

Practical Examples

1. Timestampable Trait

```
trait Timestampable {
    protected $created_at;
    protected $updated_at;

    public function touch() {
        $now = new DateTime();
        if ($this->created_at === null) {
            $this->created_at = $now;
        }
        $this->updated_at = $now;
    }

    public function getCreatedAt() {
        return $this->created_at;
    }

    public function getUpdatedAt() {
        return $this->updated_at;
    }

    public function getAge() {
        if ($this->created_at === null) {
            return null;
        }
        return $this->created_at->diff(new DateTime());
    }
}

class User {
    use Timestampable;

    private $name;

    public function __construct($name) {
        $this->name = $name;
        $this->touch();
    }
}
```

2. Singleton Trait

```
trait Singleton {
    private static $instance;

    protected function __construct() {
        // Protected constructor
    }

    public static function getInstance() {
        if (static::$instance === null) {
            static::$instance = new static();
        }
        return static::$instance;
    }

    private function __clone() {
        // Prevent cloning
    }

    private function __wakeup() {
        // Prevent unserialization
    }
}

class Logger {
    use Singleton;

    private $logs = [];

    public function log($message) {
        $this->logs[] = date('Y-m-d H:i:s') . ': ' . $message;
    }

    public function getLogs() {
        return $this->logs;
    }
}

$logger = Logger::getInstance();
```

3. Cacheable Trait

```
trait Cacheable {
    private $cache = [];
    private $cacheEnabled = true;

    protected function getCacheKey($method, $args = []) {
        return $method . ':' . md5(serialize($args));
    }

    protected function cache($method, $args, $callback) {
        if (!$this->cacheEnabled) {
            return $callback();
        }

        $key = $this->getCacheKey($method, $args);

        if (isset($this->cache[$key])) {
            return $this->cache[$key];
        }

        $result = $callback();
        $this->cache[$key] = $result;

        return $result;
    }

    public function enableCache() {
        $this->cacheEnabled = true;
    }

    public function disableCache() {
        $this->cacheEnabled = false;
    }

    public function clearCache() {
        $this->cache = [];
    }
}

class ExpensiveCalculator {
    use Cacheable;

    public function fibonacci($n) {
        return $this->cache(__METHOD__, [$n], function() use ($n) {
            if ($n <= 1) return $n;
            return $this->fibonacci($n - 1) + $this->fibonacci($n - 2);
        });
    }
}
```

Advanced Trait Patterns

1. Mixin Pattern

```
trait JsonSerializable {
    public function toJson() {
        return json_encode($this->toArray());
    }

    abstract public function toArray();
}

trait Arrayable {
    public function toArray() {
        $reflection = new ReflectionClass($this);
        $properties = $reflection-
>getProperties(ReflectionProperty::IS_PUBLIC);

        $array = [];
        foreach ($properties as $property) {
            $array[$property->getName()] = $property->getValue($this);
        }

        return $array;
    }
}

class Product {
    use JsonSerializable, Arrayable;

    public $name;
    public $price;

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

2. Observer Pattern dengan Traits

```
trait Observable {
    private $observers = [];

    public function addObserver($observer) {
        $this->observers[] = $observer;
    }

    public function removeObserver($observer) {
        $key = array_search($observer, $this->observers, true);
        if ($key !== false) {
            unset($this->observers[$key]);
        }
    }

    protected function notify($event, $data = null) {
        foreach ($this->observers as $observer) {
            $observer->update($this, $event, $data);
        }
    }
}

trait Observer {
    abstract public function update($subject, $event, $data);
}

class User {
    use Observable;

    private $name;

    public function setName($name) {
        $oldName = $this->name;
        $this->name = $name;
        $this->notify('name_changed', ['old' => $oldName, 'new' =>
$name]);
    }
}
```

3. Validation Trait

```
trait Validatable {
    private $errors = [];
    private $rules = [];
    public function addRule($field, $rule, $message = null) {
        if (!isset($this->rules[$field])) {
            $this->rules[$field] = [];
        }
        $this->rules[$field][] = ['rule' => $rule, 'message' => $message];
    }

    public function validate() {
        $this->errors = [];

        foreach ($this->rules as $field => $fieldRules) {
            $value = $this->$field ?? null;

            foreach ($fieldRules as $ruleData) {
                $rule = $ruleData['rule'];
                $message = $ruleData['message'] ?? "Validation failed for
{$field}";

                if (is_callable($rule)) {
                    if (!$rule($value)) {
                        $this->errors[$field][] = $message;
                    }
                }
            }
        }

        return empty($this->errors);
    }

    public function getErrors() {
        return $this->errors;
    }

    public function hasErrors() {
        return !empty($this->errors);
    }
}

class Form {
    use Validatable;

    public $email;
    public $password;

    public function __construct() {
        $this->addRule('email', function($value) {
            return filter_var($value, FILTER_VALIDATE_EMAIL);
        }, 'Invalid email format');

        $this->addRule('password', function($value) {
            return strlen($value) >= 8;
        }, 'Password must be at least 8 characters');
    }
}
```

Testing Traits

Unit Testing Traits

```php
// Test trait in isolation
class TraitTestHelper {
    use TraitToTest;
}

class TraitTest extends PHPUnit\Framework\TestCase {
    public function testTraitMethod() {
        $helper = new TraitTestHelper();
        $result = $helper->traitMethod();
        $this->assertEquals('expected', $result);
    }
}

// Mock abstract methods
trait TestableTrait {
    abstract protected function getDependency();

    public function processData($data) {
        $dependency = $this->getDependency();
        return $dependency->process($data);
    }
}

class TestableTraitTest extends PHPUnit\Framework\TestCase {
    public function testProcessData() {
        $mock = $this->getMockForTrait(TestableTrait::class);
        $dependency = $this->createMock(Processor::class);

        $mock->expects($this->once())
             ->method('getDependency')
             ->willReturn($dependency);

        $dependency->expects($this->once())
                   ->method('process')
                   ->with('test')
                   ->willReturn('processed');

        $result = $mock->processData('test');
        $this->assertEquals('processed', $result);
    }
}
```

Best Practices

1. Trait Naming

```
// ✅ Good - Adjective ending in -able
trait Cacheable { }
trait Timestampable { }
trait Validatable { }

// ✅ Good - Capability or behavior
trait CanCache { }
trait HasTimestamps { }
trait ValidatesInput { }

// ❌ Avoid - Noun-like names
trait Cache { }         // Confusing with class
trait Timestamp { }     // Not clear it's a behavior
```

2. Single Responsibility

```
// ✅ Good - Single, focused responsibility
trait Timestampable {
    // Only timestamp-related methods
}

trait Cacheable {
    // Only cache-related methods
}

// ❌ Bad - Multiple responsibilities
trait UtilityTrait {
    // Timestamps, caching, validation, etc.
}
```

3. Documentation

```
trait Documentable {
    /**
     * Convert object to array representation
     *
     * Classes using this trait must implement getAttributes() method
     * to define which attributes should be included in the array.
     *
     * @return array
     */
    public function toArray(): array {
        // Implementation
    }

    /**
     * Get attributes that should be included in array representation
     *
     * @return array
     */
    abstract protected function getAttributes(): array;
}
```

Common Pitfalls

1. Trait Conflicts

```
// Problem: Method name conflicts
trait A {
    public function method() { return "A"; }
}

trait B {
    public function method() { return "B"; }
}

// Solution: Explicit conflict resolution
class Resolved {
    use A, B {
        A::method insteadof B;
        B::method as methodB;
    }
}
```

2. Property Conflicts

```
// Problem: Property name conflicts
trait PropertyA {
    protected $property = "A";
}

trait PropertyB {
    protected $property = "B";  // Conflict!
}

// Solution: Use different property names
trait PropertyA {
    protected $propertyA = "A";
}

trait PropertyB {
    protected $propertyB = "B";
}
```

3. Testing Complexity

```php
// Problem: Hard to test classes with many traits
class ComplexClass {
    use TraitA, TraitB, TraitC, TraitD;
}

// Solution: Test traits separately and integration tests
class TraitATest extends PHPUnit\Framework\TestCase {
    // Test TraitA in isolation
}

class ComplexClassTest extends PHPUnit\Framework\TestCase {
    // Test integration and class-specific behavior
}
```

Traits vs Other Patterns

Traits vs Inheritance

Aspect	Traits	Inheritance
Relationship	"Can do"	"Is a"
Multiple	Yes	No (single)
Flexibility	High	Medium
Coupling	Low	Higher
Override	Method level	Class level

Traits vs Composition

Aspect	Traits	Composition
Code reuse	Compile-time	Runtime
Dependencies	Embedded	Injected
Testing	Harder	Easier
Flexibility	Lower	Higher

Contoh Implementasi

Lihat file `example.php` untuk berbagai contoh implementasi Traits di PHP.

Latihan

1. Buat trait `Loggable` untuk logging functionality
2. Implementasikan trait `Serializable` untuk object serialization
3. Buat trait `Comparable` untuk object comparison
4. Implementasikan multiple traits dengan conflict resolution

Tugas Rumah

Buat sistem blog sederhana dengan traits:

- `Timestampable` - untuk created_at, updated_at
- `Sluggable` - untuk generate URL-friendly slug
- `Taggable` - untuk tag management
- `Searchable` - untuk search functionality
- `Cacheable` - untuk caching hasil query
- Terapkan traits pada class `Post`, `User`, `Category`
- Handle conflicts yang muncul
- Buat unit tests untuk setiap trait