PHP OOP Handbook (Practical & Modern)

This is a concise, example-driven guide to PHP Object-Oriented Programming using modern PHP (8.1+). Copy/paste the snippets and you're good.

---

1) Class Basics

```
<?php
declare(strict_types=1);

class User {
    public function __construct(
        public string $name,          // promoted, typed
        private string $email,        // private = encapsulated
        public readonly int $id       // readonly after construction
(PHP 8.1+)
    ) {}

    public function email(): string {
        return $this->email;
    }
}

$u = new User('Ada', 'ada@example.com', id: 1);
echo $u->name;            // Ada
echo $u->email();         // ada@example.com
// $u->id = 10; // Error: readonly
```

Keywords: `public`, `protected`, `private`, `static`, `const`, `readonly`.

---

2) Properties, Methods, Constants, Static

```
class MathUtil {
    public const PI = 3.14159;
```

```
    public static function circleArea(float $r): float {
        return self::PI * $r ** 2;   // self:: for same class
    }
}

echo MathUtil::circleArea(2.0); // 12.565...
```

- `self::` = current class (compile-time).
- `static::` = late static binding (runtime)—useful with inheritance.

---

3) Constructors, Destructors, Cloning

```
class FileHandle {
    private $h;

    public function __construct(public string $path) {
        $this->h = fopen($path, 'r');
    }
    public function __destruct() {
        if (is_resource($this->h)) fclose($this->h);
    }
    public function __clone() {
        // deep copy if needed; here, reopen handle
        $this->h = fopen($this->path, 'r');
    }
}
```

---

4) Inheritance & Overriding

```
class Animal {
    public function speak(): string { return '...'; }
}
```

```
class Dog extends Animal {
    public function speak(): string { return 'woof'; }
}

$pet = new Dog();
echo $pet->speak(); // woof
```

- `final class X {}` prevents inheritance.
- `final public function f()` prevents overriding.
- Call parent: `parent::method()`.

---

5) Abstract Classes & Interfaces

```
abstract class Shape {
    abstract public function area(): float;
    public function describe(): string { return '2D shape'; }
}

interface Resizable {
    public function resize(float $factor): void;
}

class Rectangle extends Shape implements Resizable {
    public function __construct(public float $w, public float $h) {}
    public function area(): float { return $this->w * $this->h; }
    public function resize(float $f): void { $this->w *= $f; $this->h
*= $f; }
}
```

- Use **abstract** when you want base behavior + enforced methods.
- Use **interfaces** to define capabilities/roles.

---

6) Traits (Horizontal Reuse)

```
trait Timestamps {
    private \DateTimeImmutable $createdAt;
    private \DateTimeImmutable $updatedAt;

    public function initTimestamps(): void {
        $this->createdAt = $this->updatedAt = new
\DateTimeImmutable();
    }
    public function touch(): void { $this->updatedAt = new
\DateTimeImmutable(); }
    public function createdAt(): \DateTimeImmutable { return
$this->createdAt; }
}

class Post {
    use Timestamps;
    public function __construct(public string $title) {
$this->initTimestamps(); }
}
```

- Traits can contain properties + methods; resolve conflicts with `insteadof` / `as`.

---

7) Namespaces & Autoloading (Composer / PSR-4)

composer.json

```
{
  "autoload": { "psr-4": { "App\\": "src/" } }
}

project/
  src/
```

```
    Domain/
      User.php  // namespace App\Domain;
```

```php
<?php
// src/Domain/User.php
namespace App\Domain;

class User { /* ... */ }

// index.php
require __DIR__ . '/vendor/autoload.php';
use App\Domain\User;
$u = new User();
```

Run `composer dump-autoload` after changes.

---

8) Magic Methods (Use Sparingly)

- `__construct()`, `__destruct()`, `__clone()`
- `__get()`, `__set()`, `__isset()`, `__unset()` – property overloading
- `__call()`, `__callStatic()` – method overloading
- `__toString()` – string cast
- `__invoke()` – callable object
- `__serialize()`, `__unserialize()` – custom serialization (replace `Serializable`)

```php
class Bag {
    private array $data = [];

    public function __get(string $k)   { return $this->data[$k] ??
null; }
    public function __set(string $k, $v) { $this->data[$k] = $v; }
```

```
    public function __toString(): string { return
json_encode($this->data); }
}
```

---

9) Type System Essentials

- Scalar types + class types + `mixed` + `object`
- Union types: `int|float`
- Nullable: `?string`
- `false` as a literal type (e.g., `string|false`)
- Return `never` for non-returning functions (throw/exit)
- **Variance**:
    - Return types can be **more specific** (covariant)
    - Parameter types can be **less specific** (contravariant)

```
function f((A&B)|null $x): void {} // intersection types (A&B)
```

---

10) Enums (PHP 8.1+)

```
enum Status: string {
    case Draft = 'draft';
    case Published = 'published';

    public function canEdit(): bool {
        return $this === self::Draft;
    }
}
```

```
$status = Status::Draft;
if ($status->canEdit()) { /* ... */ }
```

- Backed enums (`: string|int`) or pure enums.
- Methods and constants allowed.

---

11) Attributes (Native Annotations)

```
#[Attribute(Attribute::TARGET_CLASS | Attribute::TARGET_METHOD)]
class Route {
    public function __construct(public string $path) {}
}

#[Route('/hello')]
class HelloController {
    #[Route('/hello/index')]
    public function index() {}
}
```

Frameworks read attributes via `Reflection`.

---

12) Exceptions & Error Handling

```
class DomainException extends \RuntimeException {}

function transfer(Account $a, Account $b, int $amount): void {
    if ($amount <= 0) throw new DomainException('Amount must be
positive');
}

try {
    transfer($a, $b, -10);
} catch (DomainException $e) {
```

```
    // handle domain error
} finally {
    // cleanup
}
```

- Always throw exceptions for exceptional cases, not for flow control.
- Create domain-specific exception types.

---

13) SPL Interfaces You'll Actually Use

```
class Items implements \IteratorAggregate, \Countable, \ArrayAccess {
    private array $items = [];

    public function getIterator(): Traversable { return new
ArrayIterator($this->items); }
    public function count(): int { return count($this->items); }

    public function offsetExists($o): bool { return
isset($this->items[$o]); }
    public function offsetGet($o): mixed { return $this->items[$o]; }
    public function offsetSet($o, $v): void {
        $o === null ? $this->items[] = $v : $this->items[$o] = $v;
    }
    public function offsetUnset($o): void { unset($this->items[$o]); }
}
```

Also: `SplStack`, `SplQueue`, `SplObjectStorage`.

---

14) Dependency Injection (DI) & SOLID

- **Single Responsibility:** one reason to change.

- **Open/Closed:** open for extension, closed for modification.
- **Liskov Substitution:** subtype must honor base type contracts.
- **Interface Segregation:** small interfaces.
- **Dependency Inversion:** depend on abstractions, not concretions.

```
interface Mailer { public function send(string $to, string $body):
void; }

class SmtpMailer implements Mailer { /* ... */ }

class RegistrationService {
    public function __construct(private Mailer $mailer) {}
    public function register(User $user): void {
        // ...
        $this->mailer->send($user->email(), "Welcome!");
    }
}
```

Use a container (Symfony/PSR-11) to wire dependencies.

---

15) Common Patterns (Minimal, Practical)

- **Factory**: encapsulate complex object creation.
- **Strategy**: interchangeable algorithms (e.g., different pricing).
- **Adapter**: wrap external API to your interface.
- **Decorator**: add behavior dynamically.
- **Repository**: persistence façade, return domain objects.
- **Value Object**: immutable domain data with invariants.

```
final class Money {
    public function __construct(
        public readonly int $amount,           // in minor units
        public readonly string $currency
    ) {
        if ($amount < 0) throw new \InvalidArgumentException();
    }
    public function add(Money $other): self {
        $this->guardSameCurrency($other);
        return new self($this->amount + $other->amount,
$this->currency);
    }
    private function guardSameCurrency(Money $other): void {
        if ($this->currency !== $other->currency) throw new
\LogicException();
    }
}
```

---

16) Immutability Tips

- Use `readonly` props + no setters.
- Return **new** instances for changes.
- Safer for concurrency, easier to reason about.

---

17) Testing

- Use **PHPUnit** or **Pest**.
- Mock via **Mockery** or PHPUnit mocks.
- Test public API; avoid testing internals.

```
final class RectangleTest extends \PHPUnit\Framework\TestCase {
```

```
    public function testArea(): void {
        $r = new Rectangle(3, 4);
        $this->assertSame(12.0, $r->area());
    }
}
```

---

18) Coding Standards & Static Analysis

- **PSR-12** coding style; run **PHP-CS-Fixer/PHPCS**.
- **PHPStan/Psalm** for static analysis (set level high).
- Use **phpdoc** when types are not expressible (e.g., generics):

```
/**
 * @template T
 * @implements IteratorAggregate<T>
 */
class Collection implements IteratorAggregate {
    /** @var list<T> */
    private array $items = [];
    /** @param iterable<T> $items */
    public function __construct(iterable $items = []) { foreach
($items as $x) $this->items[] = $x; }
    /** @return Traversable<T> */
    public function getIterator(): Traversable { return new
ArrayIterator($this->items); }
}
```

---

19) Performance & Memory

- Prefer typed properties and scalar types.
- Avoid magic methods in hot paths.

22) Mini Reference (Cheats)

```
// Late Static Binding
class Base { public static function who(): string { return
static::class; } }
class Child extends Base {}
echo Base::who();   // Base
echo Child::who();  // Child

// Anonymous class
$i = new class implements \IteratorAggregate {
    public function getIterator(): Traversable { return new
ArrayIterator([1,2,3]); }
};

// Callable object
class Greeter { public function __invoke(string $name): string {
return "Hi $name"; } }
echo (new Greeter)('Ada'); // Hi Ada

// Match expression in methods
function levelName(int $lvl): string {
    return match (true) {
        $lvl >= 90 => 'S',
        $lvl >= 75 => 'A',
        $lvl >= 60 => 'B',
        default    => 'C',
    };
}
```

---

23) Putting It All Together (Tiny App Sketch)

```
declare(strict_types=1);

namespace App\Domain;

enum Role: string { case Admin='admin'; case User='user'; }
```

- Use `readonly` for smaller zvals; prefer value objects but be mindful of copies.
- For heavy collections, consider generators:

```
function streamLines(string $path): \Generator {
    $h = fopen($path, 'rb');
    try {
        while (!feof($h)) yield rtrim(fgets($h), "\r\n");
    } finally {
        fclose($h);
    }
}
```

---

20) Practical Architecture Tips

- Keep domain pure (no HTTP/DB code).
- Use DTOs for I/O boundaries.
- Map frameworks at the edges (controllers, CLI commands).
- Validate at boundaries; assert inside domain.

---

21) Security Quick Hits

- Never interpolate into SQL; use prepared statements.
- Validate and encode output (XSS).
- Don't `unserialize()` untrusted data; use JSON or `__serialize()`.
- Hide internal exceptions; map to safe messages.

```
final class Email {
    public function __construct(public readonly string $value) {
        if (!filter_var($value, FILTER_VALIDATE_EMAIL)) {
            throw new \InvalidArgumentException("Invalid email");
        }
    }
}

interface Mailer { public function send(Email $to, string $body):
void; }

final class User {
    public function __construct(
        public readonly int $id,
        public string $name,
        public Email $email,
        public Role $role = Role::User,
    ) {}
    public function promote(): void { $this->role = Role::Admin; }
}

final class RegistrationService {
    public function __construct(private Mailer $mailer) {}
    public function register(User $user): void {
        // persist via repository (omitted)
        $this->mailer->send($user->email, "Welcome, {$user->name}!");
    }
}
```