PHP `final`, `static` & Related Features Handbook

---

1) `final` Keyword

What it does

- **For classes**: prevents inheritance.
- **For methods**: prevents overriding in subclasses.

Final Class Example

```
final class Logger {
    public function log(string $msg): void {
        echo $msg;
    }
}

// class MyLogger extends Logger {} // ❌ Error: Cannot extend final
class
```

Final Method Example

```
class Base {
    final public function connect(): void {
        echo "Connected!";
    }
}

class Child extends Base {
```

```
    // public function connect(): void {} // ❌ Error: Cannot override
final method
}
```

✅ **When to use**:

- Lock down sensitive classes (security-related, framework internals).
- Prevent accidental override in libraries.
- Mark methods as **core behavior** that should not be modified.

---

2) `static` Keyword

Three contexts for `static` in PHP:

2.1 Static Properties & Methods

Belong to the **class**, not an instance.

```
class Counter {
    public static int $count = 0;

    public static function increment(): void {
        self::$count++;
    }
}

Counter::increment();
echo Counter::$count; // 1
```

- Access via `ClassName::property` or `ClassName::method`.

- Shared across all instances.
- Often used for utility functions and counters.

2.2 Late Static Binding

- `self::` → refers to **the class where it is written** (fixed at compile time).
- `static::` → refers to **the class that called it** (runtime).

```
class Base {
    public static function who(): string {
        return static::class; // late static binding
    }
}
class Child extends Base {}
echo Base::who();   // Base
echo Child::who();  // Child
```

2.3 Static Variables in Methods

Variable value persists across calls (per class).

```
function testStatic() {
    static $count = 0;
    $count++;
    echo $count . PHP_EOL;
}

testStatic(); // 1
testStatic(); // 2
```

---

3) Other "Similar" Modifiers in Modern PHP

3.1 `readonly` Properties (PHP 8.1+)

- Set **once** (in constructor or property promotion).
- Immutable afterwards.

```
class User {
    public function __construct(
        public readonly string $name
    ) {}
}

$u = new User("Ada");
// $u->name = "Bob"; // ❌ Error: Cannot modify readonly property
```

3.2 `const` (Class Constants)

- Immutable at compile time.
- Access via `ClassName::CONST_NAME`.

```
class Math {
    public const PI = 3.14159;
}
echo Math::PI;
```

3.3 `abstract` (Reminder)

- Opposite of `final` in spirit — *forces* subclasses to implement.
- Can’t be instantiated.

---

4) Rules & Gotchas

- `final` **class** → cannot be extended at all.
- `final` **method** → inherited but cannot be overridden.
- **Static methods** cannot use `$this`.
- Avoid **stateful static properties** in long-running scripts (memory leak risk).
- `readonly` is for **immutability**, `const` is for **compile-time constants**.

---

5) Quick Reference Table

Keyword	Applies To	Purpose
`final`	Class/Method	Lock class from inheritance / lock method from overriding
`static`	Property/Method	Belongs to class, shared across instances
`static`	Variable in func	Persists between calls
`static::`	Calls	Late static binding
`readonly`	Property	Can only be assigned once
`const`	Class Constant	Compile-time immutable value
`abstract`	Class/Method	Require subclass implementation

---

6) Example Putting It All Together

```
abstract class Shape {
    public const CATEGORY = "2D";

    public function __construct(
        public readonly string $color
    ) {}

    abstract public function area(): float;

    final public function category(): string {
        return self::CATEGORY;
    }
}

class Circle extends Shape {
    public static int $created = 0;

    public function __construct(private float $radius, string $color)
{
        parent::__construct($color);
        static::$created++;
    }

    public function area(): float {
        return pi() * $this->radius ** 2;
    }
}

$c1 = new Circle(5, "red");
$c2 = new Circle(3, "blue");

echo Circle::$created;      // 2
echo $c1->category();       // 2D
```

---

7) Best Practices

- Use `final` on classes/methods you **don’t want overridden** (library stability).
- Use `static` for **stateless utility functions** and **shared counters**.
- Prefer **dependency injection** over static methods for testability.
- Use `readonly` for **value objects**.
- Avoid mutable static state in long-running apps (workers, daemons).