

PHP Abstract Classes & Interfaces Handbook

1) Abstract Classes — Partially Implemented Blueprints

Definition

An abstract class:

- **Cannot be instantiated** directly.
 - May contain **fully implemented methods** and **abstract methods**.
 - May declare **properties**.
 - May define **constructors**.
 - Used to **share common code** while forcing subclasses to implement certain methods.
-

Syntax

```
abstract class ClassName {  
    abstract public function mustImplement(): void; // No body  
    public function alreadyDone(): void {  
        echo "This works right away!";  
    }  
}
```

Example

```
abstract class Shape {  
    public function __construct(  
        protected string $color = 'black'
```

```

    ) {}

    abstract public function area(): float; // Force child to
implement

    public function describe(): string {
        return "A {$this->color} shape";
    }
}

class Circle extends Shape {
    public function __construct(private float $radius, string $color =
'black') {
        parent::__construct($color);
    }
    public function area(): float {
        return pi() * $this->radius ** 2;
    }
}

$circle = new Circle(5, 'red');
echo $circle->describe(); // A red shape
echo $circle->area();     // 78.53...

```

Visibility with Abstract Methods

```

abstract class Example {
    abstract public function doSomething(): void;    // Must be public
in child
    abstract protected function helper(): void;     // Must be
protected in child
    public function normalMethod(): void {}         // Fully
implemented
}

```

Rules

- An abstract method **cannot** have a body.
 - A child class must **implement all abstract methods**, keeping the same visibility or making it less restrictive (protected → public allowed).
 - You can **extend only one abstract class** (single inheritance).
-

✓ Use Abstract Classes When:

- You have **shared logic + enforced methods**.
 - You need to store **state** in properties.
 - You want a **base template** with partial behavior.
-
-

2) Interfaces — Pure Contracts

Definition

An interface:

- **Cannot have properties.**
 - **Cannot have constructors.**
 - All methods are **public** by default.
 - No implementation, only signatures.
 - Classes can **implement multiple interfaces**.
-

Syntax

```
interface InterfaceName {  
    public function doSomething(): void; // Always public
```

```
}
```

Example

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

interface Drawable {
    public function draw(): void;
}

class Rectangle implements Resizable, Drawable {
    public function __construct(private float $width, private float
$height) {}

    public function resize(float $factor): void {
        $this->width *= $factor;
        $this->height *= $factor;
    }
    public function draw(): void {
        echo "Drawing rectangle of {$this->width} x
{$this->height}\n";
    }
}

$rect = new Rectangle(10, 5);
$rect->draw();
$rect->resize(2);
$rect->draw();
```

Rules

- No properties.
- All methods **must be implemented** in the concrete class.

- Supports **multiple inheritance** (many interfaces).
-

✔ Use Interfaces When:

- You want to **define capabilities**.
 - You want unrelated classes to share the **same API contract**.
 - You want **loose coupling** (for Dependency Injection).
-
-

3) Abstract Class vs Interface

Feature	Abstract Class	Interface
Properties allowed	✔ Yes	✗ No
Method bodies allowed	✔ Yes	✗ No
Constructors allowed	✔ Yes	✗ No
Visibility modifiers	✔ Yes	✗ No (all public)
Multiple inheritance	✗ No	✔ Yes
Purpose	Template + shared code	Contract only

4) Combining Them

You can combine them for powerful designs.

```
interface Storable {  
    public function save(): void;  
}
```

```
abstract class Model implements Storable {
    public function __construct(protected string $table) {}
    abstract public function save(): void;
}

class User extends Model {
    public function __construct(private string $name) {
        parent::__construct('users');
    }
    public function save(): void {
        echo "Saving {$this->name} to {$this->table}\n";
    }
}

$user = new User('Ada');
$user->save();
```

5) Design Decision Flow

- Do I need to **share code + enforce rules**? → Use **Abstract Class**.
 - Do I just want to **enforce a contract**? → Use **Interface**.
 - Do I want **multiple inheritance** of behavior? → Use **Interfaces**, or **Traits + Interfaces**.
-
-

6) Quick Reference

- Abstract method → no body, must be implemented in subclass.
- Interface method → no body, public only, must be implemented in concrete class.
- Abstract class can mix implemented & unimplemented methods.

- Interface = multiple inheritance, Abstract = single inheritance.
- You can have:
 - **Interface + Abstract Class**
 - **Interface + Trait**
 - **Multiple Interfaces**
 - **Abstract Class implementing Interface**