

NIVEL 2 - POO con php

1. Introducción a la Programación Orientada a Objetos

- Conceptos básicos de la POO
- Ventajas de la POO sobre la programación estructurada
- Principios fundamentales: encapsulación, herencia, polimorfismo y abstracción

2. Clases y Objetos en PHP

- Definición de clases y objetos
- Propiedades (atributos) y métodos de clase
- Instanciación de objetos
- Acceso a propiedades y métodos

3. Encapsulación y Modificadores de Acceso

- Public, private , protected, default
- Métodos getters y setters
- Encapsulación de datos

4. Herencia

- Concepto de herencia
- Clases padres y clases hijas
- Sobrescritura de métodos
- Uso de la palabra clave `parent`

5. Polimorfismo

- Concepto de polimorfismo
- Implementación de interfaces
- Uso de la herencia para el polimorfismo
- Métodos y propiedades estáticas

6. Abstracción

- Interfaces y clases abstractas
- Diferencias entre interfaces y clases abstractas
- Implementación de interfaces

7. Manejo de Errores

- Excepciones en PHP
- Uso de try-catch para manejar errores
- Personalización de excepciones

8. Programación Orientada a Objetos Avanzada

- Traits en PHP
- Namespaces
- Clases finales y métodos finales en PHP:
 - Clases finales: concepto y uso.

- Métodos finales: definición y aplicación.
- Restricciones y ventajas de usar clases y métodos finales en PHP.

9. Relación entre Clases en PHP

- Asociación: relación entre objetos de diferentes clases.
- Composición: creación de objetos complejos mediante la inclusión de otros objetos.
- Agregación: relación entre un objeto y sus partes.

10. Dependencia entre Clases

- Concepto de dependencia entre clases.
- Uso de inyección de dependencias para desacoplar clases.

1. Introducción a la Programación Orientada a Objetos

La Programación Orientada a Objetos (POO) es un paradigma de programación que se basa en el concepto de "objetos", que son entidades que pueden contener datos (también conocidos como atributos) y métodos (funciones o procedimientos que operan sobre esos datos). Aquí tienes una explicación detallada de los puntos que mencionaste:

Conceptos básicos de la POO:

- **Objeto:** Es una instancia de una clase. Puede contener datos en forma de atributos y comportamientos en forma de métodos.
- **Clase:** Es una plantilla para crear objetos. Define los atributos y métodos que los objetos de esa clase tendrán.
- **Encapsulación:** Es el concepto de ocultar los detalles de implementación de un objeto y solo exponer una interfaz para interactuar con él. Se logra mediante el uso de modificadores de acceso como público, privado y protegido.
- **Herencia:** Es un mecanismo mediante el cual una clase puede heredar atributos y métodos de otra clase. Esto promueve la reutilización de código y facilita la organización de las clases en una jerarquía.

- **Polimorfismo:** Es la capacidad de objetos de distintas clases de responder al mismo mensaje de manera diferente. Puede manifestarse de dos formas: polimorfismo de subtipos (en tiempo de ejecución) y polimorfismo paramétrico (en tiempo de compilación).
- **Abstracción:** Es el proceso de identificar las características esenciales de un objeto, ignorando los detalles irrelevantes. Permite modelar entidades del mundo real de manera más sencilla y eficiente.


Ventajas de la POO sobre la programación estructurada:

- **Reutilización de código:** La POO promueve la reutilización de código a través de la herencia y la composición de objetos.
- **Modularidad:** Permite dividir un programa en módulos independientes (clases), lo que facilita el mantenimiento y la escalabilidad del código.
- **Facilidad de mantenimiento:** La encapsulación y la abstracción facilitan el mantenimiento del código, ya que los cambios en una parte del sistema no necesariamente afectan a otras partes.
- **Modelado del mundo real:** La POO se basa en el modelado de entidades del mundo real, lo que facilita la comprensión y el diseño de sistemas complejos

Principios fundamentales:

- **Encapsulación:** La encapsulación se logra definiendo propiedades y métodos como públicos, privados o protegidos dentro de una clase. Por ejemplo:

php

 Copy code

```
class Persona {
    private $nombre;
    private $edad;

    public function __construct($nombre, $edad) {
        $this->nombre = $nombre;
        $this->edad = $edad;
    }

    public function getNombre() {
        return $this->nombre;
    }


    public function setEdad($edad) {
        if ($edad > 0) {
            $this->edad = $edad;
        }
    }

    public function getInfo() {
        return "Nombre: " . $this->nombre . ", Edad: " . $this->edad;
    }
}

$persona = new Persona("Juan", 30);
echo $persona->getNombre(); // Imprime "Juan"
$persona->setEdad(-5); // No se cambia la edad porque es negativa
echo $persona->getInfo(); // Imprime "Nombre: Juan, Edad: 30"
```

- **Herencia:** Permite crear nuevas clases basadas en clases existentes, lo que fomenta la reutilización de código y la organización jerárquica de las clases.

php

 Copy code

```
class Empleado extends Persona {
    private $salario;


    public function __construct($nombre, $edad, $salario) {
        parent::__construct($nombre, $edad);
        $this->salario = $salario;
    }

    public function getSalario() {
        return $this->salario;
    }
}

$empleado = new Empleado("Ana", 25, 50000);
echo $empleado->getNombre(); // Imprime "Ana"
echo $empleado->getSalario(); // Imprime "50000"
```

- **Polimorfismo:** Permite que distintos objetos respondan de manera diferente a un mismo mensaje, lo que facilita el diseño de sistemas flexibles y extensibles.

php

 Copy code

```
interface Animal {  
    public function hacerSonido();  
}  
  
class Perro implements Animal {  
    public function hacerSonido() {  
        echo "Guau";  
    }  
}  
  
class Gato implements Animal {  
    public function hacerSonido() {  
        echo "Miau";  
    }  
}  
  
$perro = new Perro();  
$gato = new Gato();  
  
$perro->hacerSonido(); // Imprime "Guau"  
$gato->hacerSonido(); // Imprime "Miau"
```

- **Abstracción:** Permite modelar entidades del mundo real de manera más simple, identificando solo las características esenciales y ignorando los detalles irrelevantes.

```

abstract class Figura {
    abstract public function area();
}

class Circulo extends Figura {
    private $radio;

    public function __construct($radio) {
        $this->radio = $radio;
    }

    public function area() {
        return pi() * pow($this->radio, 2);
    }
}

class Cuadrado extends Figura {
    private $lado;

    public function __construct($lado) {
        $this->lado = $lado;
    }

    public function area() {
        return pow($this->lado, 2);
    }
}

$circulo = new Circulo(5);
echo "Área del círculo: " . $circulo->area(); // Imprime "Área del círculo: 78.539
$cuadrado = new Cuadrado(4);
echo "Área del cuadrado: " . $cuadrado->area(); // Imprime "Área del cuadrado: 16"

```

2. Clases y Objetos en PHP

Definición de Clases y Objetos en PHP

En PHP, una clase es un plano para crear objetos. Define las propiedades (atributos) y métodos (funciones) que los objetos de esa clase tendrán. Aquí tienes un ejemplo de cómo se define una clase en PHP:

```
php ✖ Save to grepper 📋 Copy code  
  
class Persona {  
    // Propiedades  
    public $nombre;  
    public $edad;  
  
    // Métodos  
    public function saludar() {  
        return "Hola, soy $this->nombre y tengo $this->edad años.";  
    }  
}
```

Propiedades y Métodos de Clase

Las propiedades son variables dentro de una clase que almacenan datos. Los métodos son funciones dentro de una clase que realizan acciones relacionadas con los objetos de esa clase. En el ejemplo anterior, `$nombre` y `$edad` son propiedades, y `saludar()` es un método.

Instanciación de Objetos

Una vez que has definido una clase, puedes crear objetos basados en esa clase. Esto se conoce como instanciación. Aquí tienes un ejemplo de cómo instanciar un objeto de la clase `Persona`:

```
php ✖ Save to grepper 📋 Copy code  
  
$persona1 = new Persona();  
$persona1->nombre = "Juan";  
$persona1->edad = 30;  
  
echo $persona1->saludar(); // Output: Hola, soy Juan y tengo 30 años.
```

Acceso a Propiedades y Métodos

Puedes acceder a las propiedades y métodos de un objeto utilizando el operador flecha `->`. En el ejemplo anterior, `$personal->nombre` y `$personal->edad` acceden a las propiedades, y `$personal->saludar()` accede al método `saludar()`.

3. Encapsulación y Modificadores de Acceso

La encapsulación y los modificadores de acceso son conceptos importantes en la programación orientada a objetos (POO). En PHP, estos conceptos nos permiten controlar el acceso a los atributos y métodos de una clase. Veamos cada uno de ellos con ejemplos en PHP:

Public, Private, Protected y Default:

- `public`: Los miembros públicos (atributos o métodos) son accesibles desde fuera de la clase.
- `private`: Los miembros privados solo son accesibles dentro de la clase misma.
- `protected`: Los miembros protegidos son accesibles dentro de la clase y las clases que heredan de ella.
- `default` (no es un modificador explícito en PHP): Si no se especifica ningún modificador, por defecto se considera público.

php



Save to grepper



Copy code

```
class Ejemplo {
    public $publico = 'Público'; // Accesible desde cualquier lugar
    private $privado = 'Privado'; // Solo accesible dentro de la clase
    protected $protegido = 'Protegido'; // Accesible dentro de la clase y las clases
    // No se especifica modificador, por lo que es público por defecto
    $sinModificador = 'Sin Modificador';

    public function mostrarPrivado() {
        // Podemos acceder al miembro privado dentro de la clase
        echo $this->privado;
    }
}

$objeto = new Ejemplo();
echo $objeto->publico; // Salida: Público
// echo $objeto->privado; // Error: No se puede acceder a un miembro privado desde fuera
// echo $objeto->protegido; // Error: No se puede acceder a un miembro protegido desde fuera
// echo $objeto->sinModificador; // Error: No se puede acceder a un miembro sin modificador
$objeto->mostrarPrivado(); // Salida: Privado
```

Métodos Getters y Setters:

Los métodos getters se utilizan para acceder a los valores de los atributos privados o protegidos, mientras que los métodos setters se utilizan para establecer o modificar esos valores.

php



Save to grepper



Copy code

```
class Persona {
    private $nombre;

    public function __construct($nombre) {
        $this->nombre = $nombre;
    }

    // Getter para obtener el nombre
    public function getNombre() {
        return $this->nombre;
    }

    // Setter para establecer el nombre
    public function setNombre($nuevoNombre) {
        $this->nombre = $nuevoNombre;
    }
}

$persona = new Persona("Juan");
echo $persona->getNombre(); // Salida: Juan
$persona->setNombre("Pedro");
echo $persona->getNombre(); // Salida: Pedro
```

Encapsulación de Datos:

La encapsulación de datos implica ocultar los detalles de implementación de una clase y exponer solo la interfaz pública. Esto se logra utilizando modificadores de acceso y proporcionando métodos públicos para interactuar con los datos internos de la clase.

```
php ✂ Save to grepper 📄 Copy code

class CuentaBancaria {
    private $saldo = 0;

    public function depositar($cantidad) {
        $this->saldo += $cantidad;
    }

    public function retirar($cantidad) {
        if ($cantidad <= $this->saldo) {
            $this->saldo -= $cantidad;
        } else {
            echo "Fondos insuficientes";
        }
    }

    public function obtenerSaldo() {
        return $this->saldo;
    }
}

$cuenta = new CuentaBancaria();
$cuenta->depositar(1000);
$cuenta->retirar(500);
echo $cuenta->obtenerSaldo(); // Salida: 500
```

4. Herencia

En la programación orientada a objetos, la herencia es un principio que permite que una clase (llamada clase hija o subclase) herede propiedades y métodos de otra clase (llamada clase padre o superclase). Esto facilita la reutilización de código y la organización de la jerarquía de clases.

Aquí tienes un ejemplo en PHP que ilustra estos conceptos:

```

<?php
// Definición de la clase padre
class Animal {
    protected $nombre;
    protected $tipo;

    public function __construct($nombre, $tipo) {
        $this->nombre = $nombre;
        $this->tipo = $tipo;
    }

    public function emitirSonido() {
        return "El $this->tipo $this->nombre emite un sonido.";
    }
}

// Definición de la clase hija
class Perro extends Animal {
    public function __construct($nombre) {
        parent::__construct($nombre, "perro");
    }

    // Sobrescritura del método emitirSonido
    public function emitirSonido() {
        return "El perro $this->nombre ladra.";
    }

    // Método específico de la clase Perro
    public function perseguirCola() {
        return "$this->nombre está persiguiendo su cola.";
    }
}

// Crear una instancia de la clase Animal
$animal = new Animal("Tigre", "felino");
echo $animal->emitirSonido() . "\n"; // Output: El felino Tigre emite un sonido.

// Crear una instancia de la clase Perro
$perro = new Perro("Max");
echo $perro->emitirSonido() . "\n"; // Output: El perro Max ladra.
echo $perro->perseguirCola() . "\n"; // Output: Max está persiguiendo su cola.
?>

```

En este ejemplo, la clase `Animal` es la clase padre que tiene propiedades y métodos comunes a todos los animales. La clase `Perro` es una clase hija que hereda de la

clase `Animal`. La clase `Perro` sobrescribe el método `emitirSonido` para que un perro emita ladridos en lugar de un sonido genérico.

La palabra clave `parent` se utiliza dentro de la clase hija para llamar a métodos de la clase padre. En el constructor de la clase `Perro`, se utiliza `parent::__construct()` para llamar al constructor de la clase `Animal` y establecer el nombre y el tipo del perro.

Espero que esta explicación y el ejemplo de código te hayan ayudado a comprender el concepto de herencia en PHP. Si tienes alguna pregunta adicional, no dudes en preguntar.

5. Polimorfismo

Concepto de Polimorfismo:

El polimorfismo es un principio de la programación orientada a objetos que permite que objetos de diferentes clases sean tratados de manera uniforme si implementan una interfaz común o si son subclases de una clase base. Esto significa que un objeto puede comportarse de diferentes maneras dependiendo del contexto en el que se utiliza. Aquí tienes un ejemplo:

php

✂ Save to grepper

📄 Copy code

```
// Definición de la clase abstracta
abstract class Animal {
    // Método abstracto
    abstract public function hacerSonido();

    // Método con implementación por defecto
    public function moverse() {
        echo "El animal se está moviendo.";
    }
}

// Clase concreta que extiende la clase abstracta
class Perro extends Animal {
    // Implementación del método abstracto
    public function hacerSonido() {
        echo "Guau";
    }
}

// Clase concreta que extiende la clase abstracta
class Gato extends Animal {
    // Implementación del método abstracto
    public function hacerSonido() {
        echo "Miau";
    }
}

// Creación de objetos y llamada a métodos
$perro = new Perro();
$perro->hacerSonido(); // Salida: Guau
$perro->moverse(); // Salida: El animal se está moviendo.

$gato = new Gato();
$gato->hacerSonido(); // Salida: Miau
$gato->moverse(); // Salida: El animal se está moviendo.
```

Implementación de Interfaces:

En PHP, las interfaces definen un conjunto de métodos que una clase debe implementar. Esto permite el polimorfismo al proporcionar un contrato que las clases deben seguir. Aquí tienes un ejemplo:

```
php ✂ Save to grepper 📄 Copy code

interface Animal {
    public function hacerSonido();
}

class Perro implements Animal {
    public function hacerSonido() {
        echo "Guau";
    }
}

class Gato implements Animal {
    public function hacerSonido() {
        echo "Miau";
    }
}
```

Uso de la Herencia para el Polimorfismo:

La herencia es otro mecanismo que permite el polimorfismo. Una subclase puede sobrescribir los métodos de la clase base para proporcionar un comportamiento específico. Aquí hay un ejemplo:

php



Save to grepper



Copy code

```
class Figura {
    public function area() {
        return 0;
    }
}

class Circulo extends Figura {
    private $radio;

    public function __construct($radio) {
        $this->radio = $radio;
    }

    public function area() {
        return pi() * $this->radio * $this->radio;
    }
}

class Cuadrado extends Figura {
    private $lado;

    public function __construct($lado) {
        $this->lado = $lado;
    }

    public function area() {
        return $this->lado * $this->lado;
    }
}
```

Métodos y Propiedades Estáticas:

Los métodos y propiedades estáticas pertenecen a la clase en lugar de a una instancia específica de la clase. Esto significa que pueden ser llamados sin necesidad de crear una instancia de la clase. El polimorfismo también se puede aplicar a métodos estáticos. Aquí tienes un ejemplo:

```
php ✂ Save to grepper 📋 Copy code

class Utilidades {
    public static function sumar($a, $b) {
        return $a + $b;
    }
}

class CalculadoraAvanzada extends Utilidades {
    public static function sumar($a, $b) {
        return $a + $b + 10; // Sobrescribiendo el método sumar
    }
}

echo Utilidades::sumar(5, 3); // Salida: 8
echo CalculadoraAvanzada::sumar(5, 3); // Salida: 18
```

En resumen, el polimorfismo en PHP se puede lograr mediante la implementación de interfaces, el uso de la herencia y la sobrescritura de métodos, así como el uso de métodos y propiedades estáticas. Esto permite que diferentes objetos sean tratados de manera uniforme, lo que aumenta la flexibilidad y la reutilización del código.

6. Abstracción

Abstracción en PHP:

La abstracción es un concepto fundamental en la programación orientada a objetos que se refiere a la capacidad de enfocarse en los aspectos importantes de un objeto y ocultar los detalles innecesarios. En PHP, podemos lograr la abstracción mediante el uso de interfaces y clases abstractas.

Interfaces y Clases Abstractas:

Interfaces:

Una interfaz en PHP define un conjunto de métodos que una clase debe implementar. Sin embargo, no proporciona ninguna implementación concreta de esos métodos. En otras palabras, una interfaz establece un contrato que las clases deben cumplir.

php



Save to grepper

Copy code

```
interface Animal {  
    public function hacerSonido();  
}
```

Clases Abstractas:

Una clase abstracta en PHP es una clase que no puede ser instanciada directamente. Puede contener métodos abstractos (métodos sin implementación) y métodos concretos (con implementación). Las clases que heredan de una clase abstracta deben implementar todos los métodos abstractos definidos en la clase abstracta.

php



Save to grepper

Copy code

```
abstract class Figura {  
    abstract public function calcularArea();  
}
```

Diferencias entre Interfaces y Clases Abstractas:

- Interfaces:
 - Pueden contener solo métodos abstractos (sin implementación).
 - Una clase puede implementar múltiples interfaces.
 - Se utilizan para definir contratos que las clases deben cumplir.
- Clases Abstractas:
 - Pueden contener tanto métodos abstractos como métodos concretos.
 - Una clase puede heredar de una sola clase abstracta.
 - Se utilizan cuando queremos proporcionar una implementación básica de ciertos métodos y requerir que las clases hijas implementen ciertos métodos.

Implementación de Interfaces:

Para implementar una interfaz en una clase, simplemente utilizamos la palabra clave `implements` seguida del nombre de la interfaz. La clase debe proporcionar una implementación para todos los métodos definidos en la interfaz.

php



Save to grepper



Copy code

```
class Perro implements Animal {  
    public function hacerSonido() {  
        echo "Woof";  
    }  
}
```

En este ejemplo, la clase `Perro` implementa la interfaz `Animal` y proporciona una implementación para el método `hacerSonido`.

7. Manejo de Errores

Excepciones en PHP:

Las excepciones en PHP se pueden lanzar usando la palabra clave `throw`. Esto permite a los desarrolladores indicar que ha ocurrido un error o una condición excepcional en su código.

php



Save to grepper



Copy code

```
<?php  
// Lanzar una excepción  
function divide($dividendo, $divisor) {  
    if ($divisor == 0) {  
        throw new Exception("División por cero");  
    }  
    return $dividendo / $divisor;  
}  
  
try {  
    echo divide(10, 0);  
} catch (Exception $e) {  
    echo "Se ha producido un error: " . $e->getMessage();  
}  
?>
```

Uso de try-catch para manejar errores:

El bloque `try` se utiliza para envolver el código que puede lanzar una excepción. El bloque `catch` se utiliza para manejar la excepción lanzada dentro del bloque `try`.

```
php ✂ Save to grepper 📋 Copy code  
  
<?php  
try {  
    // Código que puede lanzar una excepción  
    $resultado = divide(10, 0);  
    echo $resultado;  
} catch (Exception $e) {  
    // Manejo de la excepción  
    echo "Se ha producido un error: " . $e->getMessage();  
}  
?>
```

Personalización de excepciones:

Los desarrolladores pueden personalizar las excepciones creando sus propias clases de excepción. Esto permite definir tipos específicos de errores y proporcionar información adicional sobre el error.

```
php ✂ Save to grepper 📄 Copy code

<?php
// Definir una clase de excepción personalizada
class MiExcepcion extends Exception {
    public function __construct($mensaje, $codigo = 0, Exception $anterior = null) {
        parent::__construct($mensaje, $codigo, $anterior);
    }

    public function __toString() {
        return __CLASS__ . ": [{".$this->code}]: {".$this->message}.\n";
    }

    public function mensajePersonalizado() {
        return "Error personalizado: {".$this->message}";
    }
}

try {
    // Lanzar una excepción personalizada
    throw new MiExcepcion("Este es un error personalizado");
} catch (MiExcepcion $e) {
    // Manejo de la excepción personalizada
    echo $e->mensajePersonalizado();
}

?>
```

8. Programación Orientada a Objetos Avanzada

Traits en PHP:

Los traits permiten la reutilización de métodos en diversas clases. Veamos un ejemplo:

```
php Save to grepper Copy code

<?php
trait Saludo {
    public function decirHola() {
        echo '¡Hola!';
    }
}

class MiClase {
    use Saludo;
}

$objeto = new MiClase();
$objeto->decirHola(); // Salida: ¡Hola!
?>
```

Namespaces:

Los namespaces permiten organizar y encapsular clases, evitando conflictos de nombres.

```
php Save to grepper Copy code

<?php
namespace MiProyecto;

class MiClase {
    public function __construct() {
        echo 'Instanciando MiClase';
    }
}

$objeto = new MiProyecto\MiClase(); // Instanciando MiClase
?>
```

Clases finales y métodos finales en PHP:

Clases finales:

Las clases finales no pueden ser heredadas.

```
php ✖ Save to grepper 📄 Copy code

<?php
final class MiClaseFinal {
    public function decirHola() {
        echo '¡Hola!';
    }
}

// Esto dará un error ya que no se puede heredar de una clase final
class OtraClase extends MiClaseFinal {
}

?>
```

Métodos finales:

Los métodos finales no pueden ser sobrescritos por las clases hijas.

```
php ✖ Save to grepper 📄 Copy code

<?php
class MiClase {
    final public function decirHola() {
        echo '¡Hola!';
    }
}

class OtraClase extends MiClase {
    // Esto dará un error ya que no se puede sobrescribir un método final
    public function decirHola() {
        echo '¡Hola desde la clase hija!';
    }
}

?>
```

Restricciones y ventajas de usar clases y métodos finales en PHP:



- Restricciones:
 - No se pueden heredar clases finales.

- No se pueden sobrescribir métodos finales.
- Ventajas:
 - Ayuda a definir una estructura de clases más robusta y predecible.
 - Evita que clases o métodos específicos sean modificados por clases hijas, lo que puede ayudar a mantener la integridad del código.

9. Relación entre Clases en PHP

Asociación: En la asociación, dos clases están relacionadas entre sí, pero no dependen la una de la otra. Pueden interactuar, pero cada una tiene su propia existencia independiente. Por ejemplo, una clase `Usuario` podría tener una asociación con una clase `Comentario`, donde un usuario puede tener muchos comentarios.

php

 Save to grepper Copy code

```
class Usuario {
    private $nombre;

    public function __construct($nombre) {
        $this->nombre = $nombre;
    }

    public function getNombre() {
        return $this->nombre;
    }
}

class Comentario {
    private $contenido;

    public function __construct($contenido) {
        $this->contenido = $contenido;
    }

    public function getContenido() {
        return $this->contenido;
    }
}

// Asociación entre Usuario y Comentario
$usuario = new Usuario("Juan");
$comentario = new Comentario("¡Hola mundo!");
echo $usuario->getNombre() . " ha comentado: " . $comentario->getContenido();
```

Composición: En la composición, una clase está compuesta por una o más instancias de otras clases. La existencia de los objetos contenidos depende de la clase que los contiene. Por ejemplo, una clase `Automovil` puede estar compuesta por instancias de las clases `Motor`, `Rueda`, etc.

```
php ✖ Save to grepper 📋 Copy code

class Motor {
    public function encender() {
        echo "Motor encendido.";
    }
}

class Automovil {
    private $motor;

    public function __construct() {
        $this->motor = new Motor();
    }

    public function encenderMotor() {
        $this->motor->encender();
    }
}

$auto = new Automovil();
$auto->encenderMotor();
```

Agregación: En la agregación, un objeto está compuesto por otros objetos, pero esos objetos pueden existir independientemente del objeto principal. Por ejemplo, un `Equipo` puede estar compuesto por varios objetos `Jugador`, pero los jugadores pueden existir fuera del equipo.

```

class Jugador {
    private $nombre;

    public function __construct($nombre) {
        $this->nombre = $nombre;
    }

    public function getNombre() {
        return $this->nombre;
    }
}

class Equipo {
    private $jugadores = array();

    public function agregarJugador(Jugador $jugador) {
        $this->jugadores[] = $jugador;
    }

    public function listarJugadores() {
        foreach ($this->jugadores as $jugador) {
            echo $jugador->getNombre() . "<br>";
        }
    }
}

// Agregación entre Equipo y Jugador
$equipo = new Equipo();
$jugador1 = new Jugador("Lionel Messi");
$jugador2 = new Jugador("Cristiano Ronaldo");
$equipo->agregarJugador($jugador1);
$equipo->agregarJugador($jugador2);
echo "Jugadores del equipo:<br>";
$equipo->listarJugadores();

```

10. Dependencia entre Clases

Concepto de Dependencia entre Clases:

La dependencia entre clases se refiere a la relación en la que una clase (o componente) depende de otra para su funcionamiento. Esto significa que si modificamos la clase en la que hay una dependencia, puede afectar el comportamiento de la clase que la utiliza.

Por ejemplo, considera una clase `Car` que tiene una dependencia de una clase `Engine`. Sin el motor, el automóvil no puede funcionar. Si cambiamos algo en la clase `Engine`, como su interfaz o comportamiento interno, es posible que necesitemos ajustar la clase `Car` para que siga funcionando correctamente.

Uso de Inyección de Dependencias:

La inyección de dependencias es un patrón de diseño que nos permite desacoplar las clases al pasar las dependencias como argumentos en lugar de instanciarlas dentro de la clase misma. Esto hace que nuestras clases sean más flexibles, mantenibles y fáciles de probar.

Veamos un ejemplo en código PHP:

```
php ✂ Save to grepper 📄 Copy code

// Clase Engine
class Engine {
    public function start() {
        echo "Engine started!";
    }
}

// Clase Car con dependencia inyectada
class Car {
    private $engine;

    public function __construct(Engine $engine) {
        $this->engine = $engine;
    }

    public function start() {
        $this->engine->start();
    }
}

// Creamos una instancia del motor
$engine = new Engine();

// Creamos una instancia del automóvil pasando el motor como dependencia
$car = new Car($engine);

// Iniciamos el automóvil
$car->start();
```

En este ejemplo, la clase `Car` depende de la clase `Engine` para funcionar. En lugar de instanciar un motor dentro de la clase `Car`, lo pasamos como un argumento en su constructor. Esto hace que la clase `Car` no esté acoplada directamente a una implementación específica de `Engine`, lo que la hace más flexible y fácil de mantener.

Usando la inyección de dependencias, podemos cambiar el tipo de motor que utiliza `Car` simplemente pasando una instancia diferente de `Engine` en su constructor, sin necesidad de modificar la clase `Car` en sí misma. Esto facilita la modificación y extensión de nuestro código.