TEMA 1. Clases y objetos

1. ¿Cuáles son las cuatro características básicas de la programación orientada a objetos? Describe brevemente cada una

Respuesta:

📌 **Encapsulación**
Ocultamiento de los datos y detalles internos de un objeto, permitiendo acceso controlado mediante métodos públicos. Protege la integridad del objeto.

📌 **Abstracción**
Representación simplificada de entidades complejas, mostrando solo lo relevante. Un objeto expone su interfaz sin revelar cómo funciona internamente.

📌 **Herencia**
Capacidad de crear clases derivadas a partir de clases base, heredando atributos y métodos. Permite reutilizar código y establecer jerarquías.

📌 **Polimorfismo**
Un objeto puede tomar múltiples formas. Permite que métodos con el mismo nombre ejecuten comportamientos distintos según el objeto, mejorando flexibilidad.

2. Cita cuatro lenguajes populares que permitan la programación orientada a objetos

Respuesta:

🔷 **Java** — Lenguaje de propósito general, totalmente orientado a objetos, con gestión automática de memoria.

🔷 **Python** — Lenguaje dinámico y versátil que soporta POO junto con otros paradigmas, muy popular en ciencia de datos.

🔷 **C++** — Extensión de C con soporte para POO, permitiendo código de bajo nivel con estructuras orientadas a objetos.

🔷 **C#** — Lenguaje de Microsoft similar a Java, diseñado para el framework .NET, con características modernas de POO.

3. Los paradigmas anteriores a la POO, ¿Qué es la **programación estructurada**? y, todavía mejor, ¿Qué es la **programación modular**?

Respuesta:

Programación Estructurada
Es un paradigma que organiza el código mediante estructuras de control (if/else, while, for) de forma jerárquica. Descompone programas en bloques secuenciales evitando el uso de `goto`, mejorando legibilidad y mantenimiento. En C se sigue este paradigma, aunque sin objetos.

Programación Modular
Evolución de la programación estructurada que divide el programa en módulos independientes (funciones, librerías) reutilizables. Cada módulo tiene una responsabilidad específica, facilitando el desarrollo y testing. Es similar a lo que hacías en C con archivos `.h` y `.c` para separar interfaces de implementaciones.

4. ¿Qué tres elementos definen a un objeto en programación orientada a objetos?

Respuesta:

🎯 **Estado (Atributos/Propiedades)**
Son los datos que almacena el objeto. Definen sus características en un momento dado.

🎯 **Comportamiento (Métodos)**
Son las acciones que puede realizar el objeto. Definen qué puede hacer.

🎯 **Identidad**
Es lo que diferencia un objeto de otro, incluso si tienen el mismo estado y comportamiento. En memoria, cada objeto tiene una ubicación única (referencia).

5. ¿Qué es una clase? ¿Es lo mismo que un objeto? ¿Qué es una instancia? ¿Todos los lenguajes orientados a objetos manejan el concepto de clase?

Respuesta:

Una **clase** es un molde o plantilla que define la estructura (atributos) y comportamiento (métodos) de objetos. Un **objeto** es una instancia concreta de una clase con valores específicos. Por analogía: la clase es el plano de una casa, el objeto es la casa construida.

Una **instancia** es exactamente un objeto creado a partir de una clase. Múltiples instancias pueden crearse de la misma clase, cada una con su propio estado independiente.

No todos los lenguajes orientados a objetos manejan clases. Algunos como JavaScript usan **prototipos** en lugar de clases (aunque JavaScript moderno soporta sintaxis de clases). Este enfoque permite crear objetos a partir de otros objetos sin necesidad de una clase explícita.

6. ¿Dónde se almacenan en memoria los objetos? ¿Es igual en todos los lenguajes? ¿Qué es la **recolección de basura**?

Respuesta:

En lenguajes como Java, los objetos se almacenan en el **heap** (pila de memoria dinámica), mientras que las referencias (direcciones) se guardan en el **stack** (pila). Esto difiere en C/C++, donde puedes controlar manualmente si usar stack o heap. Otros lenguajes tienen estrategias propias: Python también usa heap, pero con conteo de referencias.

La **recolección de basura** es un mecanismo automático que libera memoria de objetos que ya no se usan (sin referencias activas). La JVM (Java Virtual Machine) ejecuta el recolector de basura periódicamente, eliminando la necesidad de desasignar memoria manualmente como en C++. Esto reduce bugs pero introduce ligera impredictibilidad temporal.

7. ¿Qué es un método? ¿Qué es la **sobrecarga de métodos**?

Respuesta:

Un **método** es una función definida dentro de una clase que actúa sobre los atributos del objeto. Es el equivalente a las funciones en C, pero asociado a un objeto específico. Los métodos acceden a los datos del objeto mediante `this` (la referencia al objeto actual).

La **sobrecarga de métodos** (method overloading) permite definir múltiples métodos con el mismo nombre pero diferente número o tipo de parámetros. Java selecciona automáticamente cuál ejecutar según los argumentos proporcionados. Por ejemplo, puede haber un método `calcular(int x)` y otro `calcular(double x, double y)` en la misma clase.

8. Ejemplo mínimo de clase en Java, que se llame Punto, con dos atributos, x e y, con un método que se llame `calculaDistanciaAOrigen`, que calcule la distancia a la posición 0,0. Por sencillez, los atributos deben tener visibilidad por defecto. Crea además un ejemplo de uso con una instancia y uso del método

Respuesta:

```
class Punto {
    double x;
    double y;

    double calculaDistanciaAOrigen() {
        return Math.sqrt(x * x + y * y);
    }
}
```

📝 **Ejemplo de uso:**

```
public class Main {
    public static void main(String[] args) {
        Punto p = new Punto();
        p.x = 3.0;
        p.y = 4.0;
```

```
        double distancia =
            p.calculaDistanciaAOrigen();
        System.out.println(
            "Distancia: " + distancia);
    }
}
```

Se crea una instancia `p` de la clase `Punto`, se asignan valores a sus atributos `x` e `y`, y se llama al método para calcular la distancia.

9. ¿Cuál es el punto de entrada en un programa en Java? ¿Qué es `static` y para qué vale? ¿Sólo se emplea para ese método `main`? ¿Para qué se combina con `final`?

Respuesta:

El punto de entrada en Java es el método **`main`**: `public static void main(String[] args)`. La JVM busca este método en la clase indicada para iniciar la ejecución del programa.

`static` significa que el método/atributo pertenece a la clase, no a instancias individuales. Se accede sin crear un objeto: `Clase.metodoStatico()`. Se usa en `main` porque debe ejecutarse antes de crear cualquier objeto. No es exclusivo de `main`; puedes tener atributos y métodos estáticos en cualquier clase (ej: contadores globales, utilidades).

Cuando **`static`** **se combina con** **`final`**, se crea una constante de clase. Por ejemplo: `public static final double PI = 3.14159;` es una constante que pertenece a la clase, no puede modificarse, y es accesible desde cualquier lugar sin crear instancias.

10. Intenta ejecutar un poco de Java de forma básica, con los comandos `javac` y `java`. ¿Cómo podemos compilar el programa y ejecutarlo desde linea de comandos? ¿Java es compilado? ¿Qué es la **máquina virtual**? ¿Qué es el *byte-code* y los ficheros `.class`?

Respuesta:

💻 **Compilación y ejecución desde línea de comandos:**

```
javac Archivo.java    # Compila a Archivo.class
java Archivo          # Ejecuta (sin .class)
```

Java es **semi-compilado**. El código fuente (`.java`) se compila a **bytecode** (código intermedio en archivos `.class`), que es independiente de la plataforma. Este bytecode se interpreta/compila en tiempo de ejecución por la **máquina virtual de Java (JVM)**, que sí es específica del sistema operativo.

La **JVM** es un programa que ejecuta bytecode. Traduce instrucciones bytecode a instrucciones nativas del procesador en tiempo real (compilación JIT). Gracias a esto, el mismo `.class` funciona en Windows, Linux,

macOS sin recompilar: "Write once, run anywhere".

11. En el código anterior de la clase `Punto` ¿Qué es `new`? ¿Qué es un **constructor**? Pon un ejemplo de constructor en una clase `Empleado` que tenga DNI, nombre y apellidos

Respuesta:

`new` es el operador que crea una nueva instancia de una clase en el heap y devuelve una referencia a ella. Sin `new`, solo tienes una variable declarada pero sin objeto asignado.

Un **constructor** es un método especial que se ejecuta automáticamente al crear un objeto con `new`. Tiene el mismo nombre que la clase y no retorna nada. Inicializa los atributos del objeto. Si no defines un constructor, Java proporciona uno vacío por defecto.

Ejemplo:

```
class Empleado {
    String dni;
    String nombre;
    String apellidos;

    Empleado(String dni, String nombre,
        String apellidos) {
        this.dni = dni;
        this.nombre = nombre;
        this.apellidos = apellidos;
    }
}

// Uso:
Empleado emp = new Empleado(
    "12345678X", "Juan", "Pérez");
```

12. ¿Qué es la referencia `this`? ¿Se llama igual en todos los lenguajes? Pon un ejemplo del uso de `this` en la clase `Punto`

Respuesta:

`this` es una referencia especial que apunta al objeto actual (la instancia). Permite distinguir entre atributos de la clase y variables locales, especialmente en constructores. Accede a métodos y atributos del objeto desde dentro.

En otros lenguajes se llama diferente: en C++ es `this`, en Python es `self`, en JavaScript es `this` (pero con semántica distinta). El concepto es universal: una forma de referirse al propio objeto.

Ejemplo en Punto:

```
class Punto {
    double x;
    double y;

    Punto(double x, double y) {
        this.x = x;  // atributo
        this.y = y;
    }

    void mostrar() {
        System.out.println("Punto: (" +
            this.x + ", " + this.y + ")");
    }
}
```

13. Añade ahora otro nuevo método que se llame `distanciaA`, que reciba un `Punto` como parámetro y calcule la distancia entre `this` y el punto proporcionado

Respuesta:

```
double distanciaA(Punto otro) {
    double dx = this.x - otro.x;
    double dy = this.y - otro.y;
    return Math.sqrt(dx * dx + dy * dy);
}
```

📐 **Uso:**

```
Punto p1 = new Punto(0, 0);
Punto p2 = new Punto(3, 4);
double dist = p1.distanciaA(p2);  // Devuelve 5.0
```

El método accede a los atributos del punto actual mediante `this` y a los del punto parámetro mediante la referencia `otro`. Calcula la distancia euclidiana entre ambos puntos.

14. El paso del `Punto` como parámetro a un método, es **por copia** o **por referencia**, es decir, si se cambia el valor de algún atributo del punto pasado como parámetro, dichos cambios afectan al objeto fuera del método? ¿Qué ocurre si en vez de un `Punto`, se recibiese un entero (`int`) y dicho entero se modificase dentro de la función?

Respuesta:

En Java, el paso es **siempre por valor**, pero el valor de los objetos es la **referencia** (la dirección en memoria). Esto significa que cambios en atributos del `Punto` sí afectan al objeto original fuera del método, pero reasignar la referencia no afecta. Ejemplo:

```
// Esto SÍ modifica el objeto original:
void mover(Punto p) { p.x = 10; }  // El Punto original cambia

// Esto NO afecta al original (solo dentro del método):
void reasignar(Punto p) { p = new Punto(5, 5); }
```

Con un `int`, los cambios **no afectan** al original porque `int` se pasa por valor (copia). Modificar el entero dentro del método es como cambiar una copia local que se descarta al salir. Esta es la diferencia clave: **objetos vs tipos primitivos**.

15. ¿Qué es el método `toString()` en Java? ¿Existe en otros lenguajes? Pon un ejemplo de `toString()` en la clase `Punto` en Java

Respuesta:

`toString()` es un método especial que devuelve una representación en texto del objeto. Se llama automáticamente cuando imprimes un objeto o lo concatenas con strings. Por defecto, devuelve algo como `Punto@1a2b3c`, poco útil. Puedes sobrescribirlo para personalizar la salida.

Existe en otros lenguajes: Python tiene `__str__()` y `__repr__()`, C++ tiene `<<` (operator overloading), etc. Es un patrón común para depuración.

📝 **Ejemplo:**

```
class Punto {
    double x, y;

    @Override
    public String toString() {
        return "Punto(" + x + ", " + y + ")";
    }
}

// Uso:
Punto p = new Punto(3, 4);
System.out.println(p);  // Salida: Punto(3.0, 4.0)
```

*** End Patch

16. Reflexiona: ¿una clase es como un `struct` en C? ¿Qué le falta al `struct` para ser como una clase y las variables de ese tipo ser instancias?

Respuesta:

Sí, conceptualmente una **clase** es como un `struct` en C: ambos agrupan datos (miembros/atributos). Pero un `struct` de C es solo un contenedor de datos, carece de métodos integrados, encapsulación real y los beneficios de POO.

Al `struct` le faltan:

- **Métodos integrados**: funciones que operan sobre los datos sin código externo.
- **Encapsulación**: control de acceso (private/public) a miembros.
- **Constructores y destructores**: inicialización/limpieza automática.
- **Herencia y polimorfismo**: relación entre tipos.

C++ añade exactamente esto a los structs (de hecho, en C++, `struct` y `class` son prácticamente lo mismo, solo que struct es público por defecto). Java eleva este concepto al máximo: una clase es un tipo de datos con comportamiento completo.

17. Quitemos un poco de magia a todo esto: ¿Como se podría "emular", con `struct` en C, la clase `Punto`, con su función para calcular la distancia al origen? ¿Qué ha pasado con `this`?

Respuesta:

💡 **En C, emulamos la clase así:**

```c
#include <math.h>

typedef struct {
    double x;
    double y;
} Punto;

double calculaDistanciaAOrigen(
    Punto* this) {
    return sqrt(this->x * this->x +
        this->y * this->y);
}

// Uso:
int main() {
    Punto p = {3.0, 4.0};
    double dist =
        calculaDistanciaAOrigen(&p);
    printf("Distancia: %f\n", dist);
}
```

🔍 **¿Qué ha pasado con `this`?**
En C no existe implícitamente. Debemos pasarlo explícitamente como parámetro (por referencia con `*` y `&`) a cada función. Esto revela la "magia" de Java: el compilador inserta automáticamente `this` en cada llamada. En

Java, `p.calculaDistanciaAOrigen()` equivale en C a `calculaDistanciaAOrigen(&p)`. Java simplifica enormemente la sintaxis ocultando este trabajo manual.