TEMA 1. Clases y objetos

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

Respuesta

Las cuatro características fundamentales de la POO son: **abstracción, encapsulación, herencia y polimorfismo**.

La **abstracción** permite representar objetos del mundo real mediante clases que capturan sus características esenciales, ignorando los detalles innecesarios. Por ejemplo, una clase `Coche` puede tener atributos como velocidad y color sin necesidad de representar cada átomo del vehículo.

La **encapsulación** consiste en agrupar los datos (atributos) y las operaciones (métodos) que actúan sobre ellos dentro de una clase, ocultando los detalles internos del funcionamiento. Esto es similar a tener funciones y variables relacionadas juntas en C, pero con la capacidad de controlar quién puede acceder a qué mediante modificadores de visibilidad.

La **herencia** permite que una clase herede atributos y métodos de otra clase base, evitando duplicación de código. Una clase `Vehículo` podría tener subclases como `Coche` y `Moto` que heredan sus características comunes.

El **polimorfismo** permite que objetos de diferentes clases se comportan de formas distintas ante el mismo mensaje o llamada a método. Por ejemplo, un método `acelerar()` puede comportarse diferente en un `Coche` que en un `Barco`.

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

Respuesta

Cuatro lenguajes populares con soporte para POO son: **Java**, **C++**, **Python** y **C#**. Java es utilizado extensamente en aplicaciones empresariales; C++ añade características orientadas a objetos al C

tradicional que ya se conoce; Python es un lenguaje interpretado de propósito general muy utilizado en ciencia de datos; C# es similar a Java pero diseñado por Microsoft para la plataforma .NET.

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

Respuesta

La **programación estructurada** es un paradigma que utiliza estructuras de control como secuencias, condicionales (if-else) y bucles para organizar el flujo del programa, evitando saltos no controlados (goto). El código C que ya se conoce es un ejemplo de programación estructurada, donde los programas se escriben como una serie de instrucciones que se ejecutan en orden con ramificaciones lógicas.

La **programación modular** lleva la estructura un paso más allá al dividir el programa en módulos independientes (típicamente funciones), cada uno responsable de una tarea específica. En C, esto se consigue agrupando funciones relacionadas en archivos .c y .h separados. La modularidad mejora la reutilización de código y facilita el mantenimiento.

Ambos paradigmas siguen siendo válidos y se utilizan hoy en día, pero la POO introduce un nivel adicional de organización al agrupar datos y funciones relacionadas en objetos, lo que aumenta aún más la modularidad y reduce el acoplamiento entre componentes del programa.

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

Respuesta

Tres elementos definen a un objeto: **identidad, estado y comportamiento**.

La **identidad** es lo que distingue un objeto de otro. Cada objeto es único y ocupa una posición diferente en la memoria, incluso si dos objetos tienen exactamente los mismos atributos. Por ejemplo, dos objetos `Punto` con coordenadas (3, 4) son objetos diferentes aunque sus valores sean idénticos.

El **estado** es el conjunto de valores de los atributos del objeto en un momento determinado. En un objeto `Coche`, el estado podría incluir su velocidad actual, gasolina disponible y color.

El **comportamiento** es lo que el objeto puede hacer, definido por sus métodos. Un `Coche` tiene comportamientos como acelerar, frenar o girar. Estos tres elementos trabajan conjuntamente: los métodos modifican el estado del objeto, y su identidad permanece constante a lo largo de su ciclo de vida.

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 y el comportamiento de un tipo de objeto. Es similar a un `struct` en C, pero con la capacidad adicional de contener métodos (funciones) asociados. Una clase define qué atributos tendrá un objeto y qué operaciones puede realizar.

Un **objeto** es una instancia concreta de una clase: es un ente que existe en memoria con valores específicos para los atributos definidos en la clase. Si una clase es un plano de una casa, un objeto es una casa real construida según ese plano.

Una **instancia** es exactamente lo mismo que un objeto: es una copia concreta de la clase. Se usa el término "instancia" para enfatizar la relación entre el objeto y la clase de la que procede.

No todos los lenguajes orientados a objetos utilizan clases explícitamente. Algunos lenguajes como JavaScript utilizan **prototipos** en lugar de clases, donde los objetos heredan directamente de otros objetos sin necesidad de una clase intermedia. Sin embargo, Java, C++ y C# sí utilizan el concepto de clase.

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

Respuesta

En Java, los objetos se almacenan en el **heap** (montículo), que es una región de memoria dinámica. Esto es diferente a las variables primitivas simples, que en algunos contextos pueden almacenarse en la pila (stack). El heap permite que los objetos tengan un ciclo de vida flexible, existiendo mientras existan referencias a ellos.

Esto no es igual en todos los lenguajes. En C, se utilizan `malloc()` y `free()` para gestionar memoria manualmente en el heap. En C++, se pueden usar `new` y `delete` de forma similar, aunque también permite objetos en la pila. Java abstrae este proceso y el programador no necesita liberar memoria explícitamente.

La **recolección de basura** (garbage collection) es un mecanismo automático que libera la memoria ocupada por objetos que ya no se utilizan. Cuando no hay ninguna referencia apuntando a un objeto, el recolector de basura identifica que es inalcanzable y libera su memoria automáticamente. Esto elimina la necesidad de llamar a `free()` como en C, pero tiene el costo de cierto overhead de ejecución cuando el recolector se ejecuta.

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

Respuesta

Un **método** es una función que pertenece a una clase y opera sobre los datos (atributos) de un objeto. Los métodos son el equivalente a las funciones que se creaban en C, pero asociadas a una estructura de datos específica (la clase). Los métodos tienen acceso directo a los atributos del objeto sin necesidad de pasarlos como parámetros.

La **sobrecarga de métodos** permite definir varios métodos con el mismo nombre en una clase, siempre que tengan diferentes firmas (diferentes parámetros en tipo o cantidad). Por ejemplo, se podría tener un método `suma(int a, int b)` y otro `suma(double a, double b)`. Cuando se llama al método, Java determina automáticamente cuál ejecutar según el tipo y número de argumentos proporcionados.

Esto proporciona una interfaz más intuitiva para el usuario de la clase, ya que puede llamar a métodos con el mismo nombre para operaciones similares sin importar los tipos de datos específicos. En C, esto se lograría creando funciones con nombres diferentes como `suma_int()` y `suma_double()`, lo que es más tedioso.

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

Aquí se presenta la clase `Punto` con sus atributos y método:

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

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

Para utilizar esta clase, se crea una instancia y se invoca el método:

```
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 al origen: " + distancia);
    }
}
```

En este ejemplo, se crea un objeto `Punto` llamado `p`, se asignan valores a sus atributos x e y, y luego se invoca el método `calculaDistanciaAOrigen()` que devuelve 5.0 (la distancia del punto (3, 4) al origen (0, 0)). La palabra clave `new` crea la instancia en el heap, similar a `malloc()` en C.

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 un programa Java es el método `main` con la firma `public static void main(String[] args)` . Cuando se ejecuta el programa, la máquina virtual Java busca y ejecuta este método automáticamente. Es equivalente a la función `main()` en C/C++.

La palabra clave `static` indica que un método o atributo pertenece a la clase misma, no a instancias individuales de la clase. Esto significa que se puede acceder a un método estático sin necesidad de crear un objeto de la clase. El método `main` es `static` porque debe ejecutarse antes de que existan instancias de cualquier clase. Además de `main` , `static` se usa frecuentemente en atributos de clase (variables compartidas por todas las instancias) y en métodos auxiliares que no necesitan operar sobre datos de un objeto específico.

Cuando se combina `static` con `final` , se obtiene una constante de clase: un valor que no puede cambiar y es compartido por toda la clase. Por ejemplo, `public static final double PI = 3.14159;` define una constante que todas las instancias pueden usar sin modificar. El modificador `final` simplemente significa que la variable no puede ser reasignada después de su inicialización.

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

Para compilar un programa Java desde la línea de comandos, se utiliza `javac` seguido del nombre del archivo:

```
javac MiPrograma.java
```

Esto genera un archivo `MiPrograma.class` en el mismo directorio. Para ejecutar el programa, se usa `java` seguido del nombre de la clase (sin la extensión `.class`):

```
java MiPrograma
```

Java es un lenguaje **compilado e interpretado**. El código fuente se compila en **byte-code**, que no es código máquina nativo sino un formato intermedio. Los archivos `.class` contienen este byte-code.

La **máquina virtual de Java** (JVM) es un programa que interpreta el byte-code. Cuando se ejecuta `java MiPrograma`, la JVM lee el archivo `.class` y ejecuta sus instrucciones traduciéndolas sobre la marcha a instrucciones nativas del procesador. Esto proporciona la ventaja de la portabilidad: el mismo `.class` funciona en cualquier sistema operativo que tenga una JVM instalada (write once, run anywhere). A diferencia de C/C++, no es necesario recompilar el código para cada plataforma, solo ejecutar con una JVM diferente.

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

La palabra clave `new` reserva espacio en el heap para un nuevo objeto e invoca el **constructor** de la clase. Es equivalente a usar `malloc()` en C para asignar memoria, pero con la capacidad adicional de inicializar los datos del objeto automáticamente.

Un **constructor** es un método especial que se ejecuta automáticamente cuando se crea una instancia de la clase. Su nombre es exactamente igual al de la clase, no tiene tipo de retorno (ni siquiera `void`), y se utiliza para inicializar los atributos del objeto en un estado válido. Si no se define un constructor, Java proporciona uno por defecto que no hace nada.

Aquí se muestra un ejemplo con la clase `Empleado`:

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

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

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

En este ejemplo, al ejecutarse `new Empleado(...)`, el constructor se invoca automáticamente con los argumentos proporcionados, inicializando los tres atributos del objeto de una sola vez.

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

La referencia `this` apunta al objeto actual en el que se está ejecutando el método. Es una forma de acceder a los atributos y métodos del objeto actual desde dentro de sus propios métodos. En C/C++, esto se logra con punteros `this->atributo`, pero con sintaxis diferente.

`this` no se llama igual en todos los lenguajes. En C++ se usa `this->`, en Python se usa `self` de forma explícita, en JavaScript se usa `this`, y en C# también se usa `this`.

En la clase `Punto`, `this` es útil en el constructor para diferenciar los parámetros de los atributos de la clase:

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

    public Punto(double x, double y) {
        this.x = x;  // this.x es el atributo, x es el parámetro
        this.y = y;
    }

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

Aunque en `calculaDistanciaAOrigen()` el uso de `this` es opcional (se puede escribir `Math.sqrt(x * x + y * y)`), es una buena práctica usarlo para que sea explícito que se accede a los atributos del objeto.

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

Aquí se muestra el método `distanciaA` añadido a la clase `Punto`:

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

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

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

    public 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 `distanciaA` recibe un objeto `Punto` como parámetro, accede a sus atributos públicos `x` e `y`, y calcula la distancia euclidiana entre el punto actual (`this`) y el punto proporcionado. Este ejemplo demuestra cómo los métodos pueden operar sobre múltiples objetos de la misma clase.

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, los objetos como `Punto` se pasan **por referencia**: se pasa una referencia (similar a un puntero en C) al objeto original. Si se modifica un atributo del objeto dentro del método, dichos cambios afectan al objeto original fuera del método. Esto es diferente a C/C++, donde se pueden pasar referencias explícitamente con `&` o usar punteros.

Los tipos primitivos como `int`, `double`, `boolean`, etc., se pasan **por copia**: se crea una copia del valor, y cualquier modificación dentro del método no afecta la variable original. Esto es similar al paso por valor en C/C++.

```
public void modificaPunto(Punto p) {
    p.x = 100;  // Afecta al objeto original
}

public void modificaEntero(int valor) {
    valor = 100;  // No afecta a la variable original
}

// Uso:
Punto p = new Punto(5, 5);
modificaPunto(p);
// Ahora p.x es 100

int num = 5;
modificaEntero(num);
// num sigue siendo 5
```

Esta distinción es fundamental: los objetos son referencias (paso por referencia), mientras que los primitivos son valores (paso por copia).

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

Respuesta

El método `toString()` es un método especial heredado de la clase base `Object` en Java que devuelve una representación textual del objeto. Cuando se imprime un objeto usando `System.out.println()`, Java automáticamente invoca `toString()` para convertir el objeto en una cadena de texto. Por defecto, devuelve algo como `Punto@1a2b3c`, que no es muy útil.

El concepto existe en otros lenguajes con nombres diferentes. En C++, se sobrecarga el operador `<<` o se define un método similar. En Python, se sobrecarga el método `__str__()`. La idea es la misma: proporcionar una forma legible de representar el estado del objeto como texto.

Aquí se muestra un ejemplo de `toString()` sobrecargado en la clase `Punto`:

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

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

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

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

La anotación `@Override` es opcional pero indica explícitamente que se está sobrescribiendo un método de la clase base. Ahora cuando se imprime el objeto, se obtiene una representación clara y legible.

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

Una clase es conceptualmente similar a un `struct` en C en que ambas agrupan datos relacionados en una única entidad. Sin embargo, existen diferencias significativas. Un `struct` en C es principalmente un contenedor pasivo de datos, mientras que una clase incluye comportamiento (métodos).

Para que un `struct` en C fuera equivalente a una clase, le faltaría: **(1) métodos o funciones miembro** que operen sobre los datos; **(2) encapsulación y control de acceso** mediante modificadores de visibilidad como `private` y `public` ; y **(3) constructores y destructores** para inicializar y limpiar automáticamente los datos.

En C es posible emular algunas de estas características pasando un puntero a la estructura como primer parámetro a una función (similar a `this`), pero esto es manual y poco elegante. Java proporciona estos mecanismos de forma integrada y automática, lo que hace que las variables de tipo clase sean verdaderas instancias con comportamiento, no solo contenedores de datos. Esto es una de las ventajas fundamentales de la POO sobre la programación estructurada.

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

Se puede emular la clase `Punto` de Java usando un `struct` en C y funciones que reciben un puntero al struct como primer parámetro:

```c
#include <math.h>

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

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

// Uso:
Punto p;
p.x = 3.0;
p.y = 4.0;
double distancia = Punto_calculaDistanciaAOrigen(&p);
```

En este ejemplo, `this` ha sido reemplazado explícitamente por un puntero al `struct` que se pasa como parámetro. El nombre de la función incluye el nombre del struct (`Punto_`) para simular un espacio de nombres, ya que C no tiene soporte nativo para métodos.

La diferencia principal es que en Java, `this` es implícito y automático, mientras que en C debe ser explícitamente pasado como parámetro. Esto ilustra cómo la POO en Java oculta una gran cantidad de detalles de bajo nivel que en C deben gestionarse manualmente. Java hace que el código sea más legible (`p.calculaDistanciaAOrigen()` frente a `Punto_calculaDistanciaAOrigen(&p)`) y proporciona mejor encapsulación.