TEMA 6. Genericidad

1. Empleando void* o Object, pon un ejemplo en C++ y en Java, de una estructura de datos, que empleando un array primitivo, permita alojar cualquier tipo de dato.

Respuesta En C++, se puede emplear `void*` para crear una estructura de datos genérica. Un ejemplo básico sería un array que almacena punteros a cualquier tipo de dato. Sin embargo, al usar `void*`, se pierde la información del tipo, por lo que es necesario realizar un casting explícito al acceder a los elementos.

```cpp
#include <iostream>
using namespace std;

void* array[3];

int main() {
    int a = 10;
    double b = 20.5;
    char c = 'z';

    array[0] = &a;
    array[1] = &b;
    array[2] = &c;

    cout << "Int: " << *(int*)array[0] << endl;
    cout << "Double: " << *(double*)array[1] << endl;
    cout << "Char: " << *(char*)array[2] << endl;

    return 0;
}
```

En Java, se puede usar `Object` para lograr un comportamiento similar. Al igual que en C++, se pierde la información del tipo, y es necesario realizar un casting explícito al acceder a los elementos.

```java
public class Main {
    public static void main(String[] args) {
        Object[] array = new Object[3];

        array[0] = 10; // Autoboxing de int a Integer
        array[1] = 20.5; // Autoboxing de double a Double
        array[2] = 'z'; // Autoboxing de char a Character

        System.out.println("Int: " + (Integer) array[0]);
        System.out.println("Double: " + (Double) array[1]);
        System.out.println("Char: " + (Character) array[2]);
    }
}
```

Ambos ejemplos muestran cómo se puede almacenar cualquier tipo de dato en un array genérico. Sin embargo, el uso de `void*` en C++ y `Object` en Java tiene limitaciones, como la falta de seguridad en tiempo de compilación y la necesidad de realizar conversiones explícitas, lo que puede llevar a errores en tiempo de ejecución.

2. Brevemente, ¿Qué significa la "programación genérica"? ¿Es el ejemplo anterior un ejemplo básico de programación genérica?

Respuesta La programación genérica es un paradigma que permite escribir código que puede operar con diferentes tipos de datos sin necesidad de duplicar el código para cada tipo. Esto se logra mediante el uso de parámetros de tipo, que actúan como marcadores de posición para los tipos concretos que se especificarán en tiempo de compilación. Este enfoque mejora la reutilización del código, la seguridad de tipos y la legibilidad, ya que evita el uso de conversiones explícitas y reduce los errores en tiempo de ejecución.

El ejemplo anterior, que utiliza `void*` en C++ y `Object` en Java, no es un ejemplo de programación genérica en el sentido estricto. Aunque permite almacenar diferentes tipos de datos en una misma estructura, no proporciona seguridad de tipos en tiempo de compilación ni evita la necesidad de realizar conversiones explícitas. Por lo tanto, se considera una solución genérica en un sentido más básico, pero no aprovecha las ventajas de los mecanismos de programación genérica que ofrecen lenguajes como C++ (con plantillas) o Java (con generics).

En resumen, el ejemplo anterior es una aproximación rudimentaria a la programación genérica, pero carece de las características clave que definen este paradigma, como la seguridad de tipos y la eliminación de conversiones explícitas. Para implementar programación genérica de manera adecuada, se deben utilizar herramientas como plantillas en C++ o generics en Java.

3. Reflexionemos con otro ejemplo. Si creo un array de Persona, y luego lo recorro invocando su método saludar(), el cual puede ser sobreescrito en las subclases, el algoritmo de recorrido, programado contra abstracciones (Persona), donde se pueden implementar distintos tipos de Persona, ¿es una forma de programación de un algoritmo genérico simple?

Respuesta El ejemplo planteado, donde se recorre un array de objetos de tipo `Persona` y se invoca el método `saludar()`, puede considerarse una forma de programación genérica simple basada en el uso de abstracciones y polimorfismo. En este caso, el algoritmo de recorrido no depende de las implementaciones concretas de las subclases de `Persona`, sino que opera sobre la interfaz común definida por la clase base. Esto permite que el mismo código funcione con cualquier objeto que sea una instancia de `Persona` o de sus subclases.

Este enfoque aprovecha el polimorfismo para delegar la implementación específica del método `saludar()` a las subclases, lo que elimina la necesidad de realizar comprobaciones explícitas de tipo o conversiones. Aunque no utiliza parámetros de tipo como en la programación genérica moderna, sigue siendo un ejemplo de cómo programar contra abstracciones para lograr flexibilidad y reutilización del código.

Sin embargo, este tipo de programación genérica tiene limitaciones. Por ejemplo, no proporciona la misma seguridad de tipos que los mecanismos de programación genérica basados en parámetros de tipo, como los generics en Java o las plantillas en C++. Además, el uso de un array de `Persona` implica que todos los elementos deben ser instancias de `Persona` o de sus subclases, lo que restringe la generalidad del algoritmo en comparación con una solución basada en parámetros de tipo.

En conclusión, aunque este enfoque puede considerarse una forma básica de programación genérica, no alcanza el nivel de flexibilidad y seguridad de tipos que ofrecen los mecanismos modernos de programación genérica. Es una solución adecuada para escenarios donde el polimorfismo es suficiente para manejar la variabilidad de los tipos.

4. Indica los problemas respecto al chequeo de tipos, de emplear void* o Object cuando se crean estructuras de datos genéricas.

Respuesta El uso de `void*` en C++ y `Object` en Java para crear estructuras de datos genéricas presenta varios problemas relacionados con el chequeo de tipos. En ambos casos, se pierde la información del tipo en tiempo de compilación, lo que obliga a realizar conversiones explícitas (casting) al acceder a los elementos. Esto introduce un riesgo significativo de errores en tiempo de ejecución si el tipo de dato no coincide con el esperado.

En C++, al usar `void*`, el programador debe recordar manualmente el tipo de cada elemento almacenado en la estructura. Si se realiza un casting incorrecto, el comportamiento del programa es indefinido, lo que puede provocar fallos difíciles de depurar. Además, no hay ninguna verificación en tiempo de compilación que garantice que los datos almacenados y recuperados sean del mismo tipo.

En Java, el uso de `Object` tiene problemas similares. Aunque el lenguaje lanza una excepción `ClassCastException` si el casting es incorrecto, esto ocurre en tiempo de ejecución, no en tiempo de compilación. Esto significa que los errores relacionados con tipos no se detectan hasta que el programa se ejecuta, lo que puede ser problemático en aplicaciones grandes o críticas.

En ambos casos, la falta de seguridad de tipos en tiempo de compilación dificulta el mantenimiento del código y aumenta la probabilidad de errores. Estos problemas se resuelven en gran medida mediante el uso de mecanismos de programación genérica, como plantillas en C++ y generics en Java, que permiten definir estructuras de datos genéricas con seguridad de tipos en tiempo de compilación.

5. Vamos entonces con mecanismos de mejora de la programación genérica ¿Qué son los "parámetros de tipo"?

Respuesta

6. En Java existe "generics", en C++ existen "templates". Pon un ejemplo de uso de programación genérica en ambos, instanciando una lista o vector dinámico que solo admite String. Introduce valores, y luego haz un recorrido de ellos mostrando cómo cada elemento es del tipo concreto con seguridad.

Respuesta

7. Sobre el funcionamiento de la programación genérica. Qué hace el compilador cuando se instancia una clase que tiene parámetros de tipo? Hace lo mismo C++ y Java? Qué es el "type erasure" de Java y la "instanciación de plantillas" de C++?

Respuesta

8. Vamos a crear una nueva clase con parámetros de tipo. Define en Java una clase "Par", que permite alojar dos valores de tipos diferentes. Incluye un constructor y un getter para cada tipo. Pon un ejemplo de uso de ese Par, por ejemplo para especificar el tipo de retorno de una función que devuelve en un Par la media y desviación típica de un array de double.

Respuesta

9. En Java, se pueden declarar parámetros de tipo también a nivel de método, no solo a nivel de clase. Pon un ejemplo con un método genérico "seleccionaUno", que pasados dos objetos del mismo tipo, te devuelva aleatoriamente uno de ellos. Muestra la diferencia de definirlo con dos Objects, a definirlo con dos parámetros de tipo, en terminos de (i) evitar downcasting y (ii) forzar que ambos objetos sean del mismo tipo.

Respuesta

10. Se pueden establecer restricciones en los parámetros de tipo? Por ejemplo, si quiero definir un tipo genérico T, puedo decir que tenga que ser un número? Pon un ejemplo en Java de un Punto con dos coordenadas, metodos getX, getY, y una función calcularDistanciaA otro Punto. Permite que esas coordenadas sean cualquier tipo de número. Pon dos soluciones: una simplemente creando coordenadas de tipo Number y otra añadiendo generics para reforzar el chequeo de tipos y saber exactamente con qué tipo de número trabaja el Punto.

Respuesta

11. Sobre las soluciones anteriores. Si bien ambas permiten trabajar con distintos tipos de número sin duplicar la clase Punto, reflexiona sobre el refuerzo del chequeo de tipos con generics. ¿Permiten ambas crear un punto con una coordenada de tipo entero y la otra coordenada de tipo double? ¿Qué tipo devuelve el getX con la solucion sin generics y qué tipo devuelve el que tiene la solución con generics?

Respuesta

12. Hagamos un ejemplo avanzado. El siguiente código, con interfaz Punto, que define un método calcularDistanciaA(Punto p), junto con las implementaciones Punto2D y Punto3D. Añade generics para asegurarnos que la sobreescritura del método calcular distancia a otro Punto siempre es sobre un Punto del mismo tipo, evitando instanceof y el downcasting.

```java
public interface Punto {
    public double distanciaA(Punto p);
}

public class Punto2D implements Punto {
     private final double x, y;
     public Punto2D(double x, double y) {
        this.x = x; this.y = y;
    }

    @Override
    public double distanciaA(Punto p) {
        if (p instanceof Punto2D) {
            Punto2D p2d = (Punto2D) p;
            return Math.sqrt(Math.pow(x - p2d.x, 2)
                    + Math.pow(y - p2d.y, 2));
        } else {
            throw new RuntimeException("p debe ser Punto 2D");
        }
    }
}
```

```
22 public class Punto3D implements Punto {
23     // Igual que Punto2D, pero con tres coordenadas
24     ...
25 }
```

Respuesta