Genericidad

Clases y métodos genéricos, cuando usarlos y porque.

Utilizando Object para datos genéricos

Supongamos que queremos crear una clase que tenga una lista de números, de cualquier tipo, y que calcule la suma y el promedio de dicha lista. Si está última es una atributo, se debe especificar que tipo de dato va a alojar, es decir, si es de tipo Integer, Float o Double. Para poder realizar las operaciones con distintos tipos de números tendríamos que crear distintas clases donde lo único que cambiaría sería el tipo de dato de la lista.

Para poder evitar el exceso innecesario de código, existe la posibilidad de utilizar la superclase Object. Se podría crear una sola clase que contenga una lista de tipo Object y pueda almacenar todo tipo de datos, y así ahorrarnos el tener que crear distintas clases. Luego en cada operación se castea el dato de clase Object a uno numérico.

```java
public class CalculadoraObject {

    private ArrayList<Object> lista;

    public CalculadoraObject() {}

    public CalculadoraObject(ArrayList<Object> lista) {
        this.lista = lista;
    }

    public double suma() {

        int suma = 0;
        for (Object numero : lista) {
            suma += (double) numero;
        }

        return suma;
    }

    public double promedio() {
        return suma() / lista.size();
    }
}
```

```java
public class Main {
    public static void main(String[] args) {

        ArrayList<Object> lista = new ArrayList<>(Arrays.asList(1.6,2,3.5,4,5.55));

        CalculadoraObject calculadora = new CalculadoraObject(lista);

        System.out.println(calculadora.suma());

    }
}
```

El problema de usar Object

Pérdida de seguridad en tiempo de compilación: Cuando se utiliza Object como referencia genérica, los errores de tipo pueden pasar desapercibidos hasta que se produzcan en tiempo de ejecución. En el ejemplo anterior, al querer castear el objeto Object a un tipo double, Java nos arroja un error de tipo *ClassCastException.*

Dificultad para mantener y modificar el código: El código puede ser más difícil de mantener y modificar, ya que no se especifica de forma clara y explícita el tipo de datos que se están utilizando. Esto puede dificultar la lectura y comprensión del código, y aumentar la probabilidad de introducir errores.

Pérdida de reutilización de código: El código no se puede reutilizar fácilmente en otros contextos.

Pérdida de eficiencia: Al utilizar Object, se pierde la eficiencia en tiempo de ejecución. Esto se debe a que, al utilizar Object, es necesario hacer una conversión de tipos en tiempo de ejecución, lo que puede tener un impacto negativo en el rendimiento del programa.

Genericidad

La **genericidad** en Java es una característica que permite trabajar con tipos de datos de forma más flexible y segura en tiempo de compilación.

La genericidad se implementa mediante la inclusión de tipos de datos parametrizados en las clases, interfaces y métodos. Es decir, en lugar de trabajar con tipos de datos específicos, se pueden definir clases e interfaces que acepten cualquier tipo de dato, y luego especificar el tipo de dato concreto cuando se crea una instancia de la clase o se llama a un método.

La razón de la genericidad se basa principalmente en el hecho de que los algoritmos de resolución de numerosos problemas no dependen del tipo de dato que procesan. Por ejemplo: un algoritmo que implementa una pila de caracteres es igual al que se usa para implementar una pila de números enteros. **La programación genérica significa escribir un código que pueda reutilizar muchos tipos diferentes de objetos.**

Ventajas de la Genericidad

Seguridad en tiempo de compilación: La genericidad ayuda a detectar y prevenir errores en tiempo de compilación, ya que los tipos de datos se comprueban en tiempo de compilación en lugar de en tiempo de ejecución.

Flexibilidad: La genericidad permite trabajar con una amplia variedad de tipos de datos sin tener que crear una versión específica para cada uno de ellos.

Reutilización de código: La genericidad permite escribir clases y métodos genéricos que pueden ser reutilizados en diferentes contextos.

Mejora de la legibilidad del código: La genericidad permite especificar de manera clara y explícita los tipos de datos que se están utilizando en el código.

Sintaxis de la genericidad y clases

La sintaxis para trabajar con Genericidad implica el uso de parámetros de tipo (por ejemplo, <T> o <E>) que se definen al crear una clase o método genérico. Estos parámetros de tipo se utilizan para especificar el tipo de datos concreto que se utilizará cuando se cree una instancia de la clase o se llame al método. La convención que se utiliza para los datos de tipo genéricos son:

- E ⟶ elemento de una colección.
- K ⟶ clave.
- V ⟶ valor.
- N ⟶ número.
- T ⟶ tipo.
- S, U, V, etc. ⟶ para segundos, terceros y cuartos tipos.

Cuando se crea una instancia de la clase genérica, se especifica el tipo de dato concreto que se utilizará.

En el siguiente ejemplo se crea una clase llamada **MiClase** que recibe como parámetro un dato de tipo genérico T. Cuando se instancie esta clase, el programador deberá indicar el tipo de dato de T. Si queremos limitar T a un determinado tipo de dato, debemos hacer un extends de la clase de ese dato. En la segunda foto estamos indicando que la clase **Miclase** va a trabajar con objetos de tipo Persona o de subclases de esta última.

```java
public class MiClase<T> {
  private T dato;

  public void setDato(T dato) {
    this.dato = dato;
  }

  public T getDato() {
    return this.dato;
  }
}
```

```java
public class MiClase<T extends Persona>
{
  private T dato;

  public void setDato(T dato) {
    this.dato = dato;
  }

  public T getDato() {
    return this.dato;
  }
}
```

```java
MiClase<String> instancia = new MiClase<String>();
```

Métodos genéricos

Los métodos genéricos son otra manera de implementar la genericidad. En la sintaxis se debe indicar el o los tipos de genéricos que se van a utilizar en el método, ya sea para su retorno o parámetros. Se escribe luego del modificador de acceso y antes del tipo de dato que retorna. Para indicar otro genérico se escribe una coma y luego el otro genérico con una letra distinta. El método puede retornar un tipo de dato genérico si es que este ya se declaró luego del modificador de acceso. Hay que recordar que los tipos genéricos pueden heredar de otras clases para limitar el tipo de dato. Ejemplos de métodos genéricos:

public <**T**> List<**T**> *fromArrayToList*(**T**[] c) {} // *El método va a retornar una lista del tipo de dato que le pasemos como parámetro. Si le pasamos un arreglo de tipo Integer, va a devolver una lista de ese tipo.*

public <**T** extends **Number**, **G**> List<T> *fromArrayToList*(**T**[] c, **G**[] x) {} // *Se indica que se van a trabajar con dos genéricos, donde T puede ser todas las clases que heredan de Number. El método va a retornar una lista de tipo T.*