UNIVERSIDAD TECNOLÓGICA DEL PERÚ

Facultad de Ingeniería

Programacion Orientada a Objetos

Sesión 12: Clases genéricas

Recordando:

¿Que vimos la clase pasada?

Universidad Tecnológica del Perú

Logro de aprendizaje

Al finalizar la sesión, el estudiante soluciona problemas aplicando las clases genéricas usando java en la resolución de ejercicios.

Universidad Tecnológica del Perú

Utilidad

- Las clases genéricas son una herramienta fundamental en la programación orientada a objetos y ofrecen múltiples beneficios para los estudiantes de ingeniería de sistemas.
- Para los estudiantes, el conocimiento y uso de clases genéricas es esencial. Les permite escribir código más limpio, eficiente y seguro. Además, fomenta buenas prácticas de programación que son cruciales para el desarrollo de software a gran escala y la gestión de sistemas complejos.

Agenda

- Wildcards (Comodines).
- Implementación de Clases Genéricas en Colecciones.

Wildcards (Comodines)

Definición y Propósito:
Los wildcards, también conocidos como comodines, son una característica importante en la programación genérica de Java que permite trabajar con tipos genéricos de manera más flexible. Los wildcards se representan con el símbolo? y se utilizan en diferentes contextos para indicar "cualquier tipo". Hay dos tipos principales de wildcards en Java: ? Extends Tipo y ? super Tipo.

Universidad Tecnológica del Perú

Wildcards

https://www.youtube.com/watch?v=LiXW4OHgJBk

? extends Tipo (Wildcard con límite superior)

Esto significa que el tipo genérico es un subtipo de "Tipo" o "Tipo" mismo. En otras palabras, permite que el tipo genérico sea cualquier subtipo de "Tipo". Por ejemplo:

```
public void imprimirLista(List<? extends
Number> lista) {
    for (Number elemento : lista) {
        System.out.println(elemento);
    }
}
```

En este ejemplo, ? Extends Number permite que la lista contenga cualquier tipo que sea un subtipo de Number, como Integer o Double.

? super Tipo (Wildcard con límite inferior)

Esto significa que el tipo genérico es un supertipo de "Tipo" o "Tipo" mismo. En otras palabras, permite que el tipo genérico sea cualquier supertipo de "Tipo". Por ejemplo:

```
public void agregarElemento(List<?
super Integer> lista) {
    lista.add(42);
}
```

En este ejemplo, ? super Integer permite que la lista acepte cualquier tipo que sea un supertipo de Integer, como Object.

Captura de Tipos (Type Capture)

La captura de tipos es un concepto relacionado con los wildcards. Cuando trabajas con un wildcard en un método, Java captura el tipo específico que se utiliza en lugar del comodín y lo almacena en una variable temporal. Esto es útil cuando deseas trabajar con ese tipo dentro del método. Por ejemplo:

```
public void procesoLista(List<?> lista) {
    // Java captura el tipo real y lo almacena en una variable temporal
    Object elemento = lista.get(0);
    // Ahora puedes trabajar con 'elemento' como si fuera del tipo capturado
    System.out.println(elemento.toString());
}
```

Uso de Wildcards

Los wildcards son especialmente útiles cuando deseas escribir métodos o clases genéricas que sean más flexibles en cuanto a los tipos de datos que pueden manejar.

Los métodos que aceptan wildcards pueden trabajar con una amplia gama de tipos sin conocer el tipo exacto en tiempo de compilación.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);
// Uso de wildcard con límite superior
imprimirLista(numeros);
List<Number> listaNumero = new ArrayList<>();
agregarElemento(listaNumero);
```

Wildcards

https://www.youtube.com/watch?v=becfU6P5o1c

Uso de Wildcards

Los wildcards en Java permiten trabajar con tipos genéricos de manera flexible y pueden ser de gran utilidad cuando necesitas escribir métodos o clases genéricas que pueden manejar una variedad de tipos de datos sin conocer el tipo exacto de antemano.
La captura de tipos (type capture) es un concepto relacionado que permite trabajar con el tipo capturado dentro del método que utiliza wildcards.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);
// Uso de wildcard con límite superior
imprimirLista(numeros);
List<Number> listaNumero = new ArrayList<>();
agregarElemento(listaNumero);
```

Implementación de Clases Genéricas en Colecciones

Una de las áreas más comunes donde se utilizan clases genéricas en Java es en la implementación de colecciones genéricas.

Las colecciones son estructuras de datos ampliamente utilizadas para almacenar y manipular grupos de objetos en Java, y el uso de clases genéricas mejora significativamente su flexibilidad y seguridad de tipos.

ArrayList Genérico

El ArrayList genérico es una de las implementaciones más comunes de listas en Java.
Permite almacenar elementos de un tipo específico y garantiza que solo se puedan agregar elementos del tipo declarado.

```
ArrayList<String> listaDeNombres = new ArrayList<>();
listaDeNombres.add("Alice");
listaDeNombres.add("Bob");
```

En este ejemplo, ArrayList<String> garantiza que solo se puedan agregar cadenas a la lista listaDeNombres.

Uso de clases genéricas en Mapas

También es posible utilizar clases genéricas en la implementación de mapas, como HashMap o TreeMap. Esto permite asignar valores a claves específicas mientras se mantiene la seguridad de tipos.

```
HashMap<String, Integer> mapaDeEdades = new HashMap<>();
mapaDeEdades.put("Alice", 30);
mapaDeEdades.put("Bob", 25);
```

En este caso, HashMap<String, Integer> asegura que solo se puedan asociar valores enteros con claves de tipo cadena.

Iteración y acceso a elementos

Cuando se utilizan colecciones genéricas, no es necesario realizar conversiones de tipos al acceder o iterar a través de los elementos, ya que la información de tipo se conserva en tiempo de compilación.
Esto aumenta la seguridad de tipos y mejora la legibilidad del código.

```
ArrayList<Integer> listaDeNumeros = new ArrayList<>();
listaDeNumeros.add(1);
listaDeNumeros.add(2);
listaDeNumeros.add(3);
for (Integer numero : listaDeNumeros) {
    System.out.println(numero);
}
```

Ventajas de usar colecciones genéricas

- **Seguridad de tipos:** Las colecciones genéricas garantizan que solo se puedan agregar elementos del tipo especificado, lo que reduce los errores de tipo en tiempo de ejecución.
- **Legibilidad del código:** El uso de colecciones genéricas hace que el código sea más legible y autoexplicativo, ya que se especifica claramente el tipo de datos que se almacena en la colección.
- **Reducción de conversiones de tipos:** No es necesario realizar conversiones de tipos al acceder a elementos de la colección, ya que el tipo está garantizado en tiempo de compilación.

Ejemplo de uso de colecciones genéricas

```
List<Double> listaDeNotas = new ArrayList<>();
listaDeNotas.add(8.5);
listaDeNotas.add(9.0);
listaDeNotas.add(7.5);
double suma = 0.0;
for (Double nota : listaDeNotas) {
    suma += nota;
}
double promedio = suma / listaDeNotas.size();
System.out.println("Promedio de notas: " + promedio);
```

Ejemplo de uso de colecciones genéricas

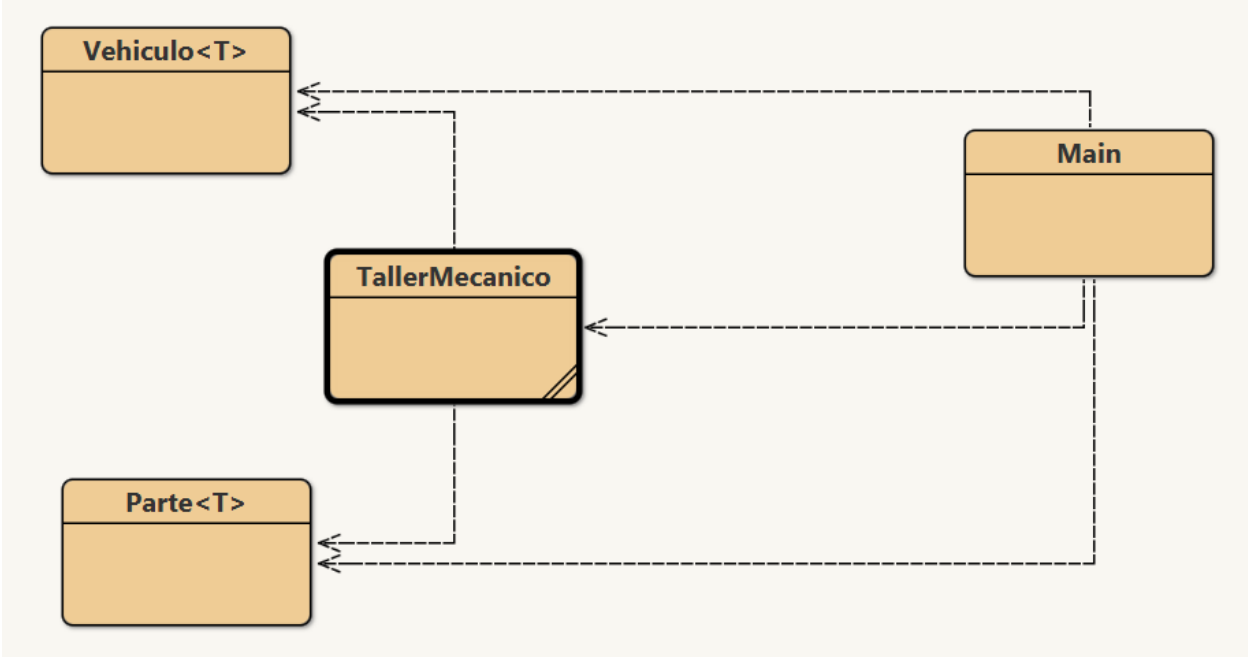
- La implementación de clases genéricas en colecciones, como ArrayList o HashMap, proporciona seguridad de tipos y legibilidad al código, lo que facilita la manipulación y gestión de datos de manera eficiente y sin necesidad de conversiones de tipos innecesarias.
- Esto hace que las colecciones genéricas sean una parte fundamental de la programación en Java.

Ejercicio-Taller mecánico

Supongamos que queremos modelar un taller mecánico que trabaja con vehículos de diferentes tipos y partes del vehículo.

- Utilizaremos dos clases genéricas: Vehiculo y Parte
- 2 métodos genéricos: repararVehiculo y reemplazarParte.

Ejercicio-Taller mecánico



```
public class Vehiculo<T> {
    private String nombre;
    private T tipo;
    public Vehiculo(String nombre, T tipo) {
        this.nombre = nombre;
        this.tipo = tipo;
    }
    public String getNombre() {
        return nombre;
    }
    public T getTipo() {
        return tipo;
    }
    public void mostrarDetalles() {
        System.out.println("Vehículo: " + nombre);
        System.out.println("Tipo: " + tipo);
    }
}
```

Ejercicio:
Clase Genérica
Vehículo

```
public class Parte<T> {
    private String nombre;
    private T tipo;
    public Parte(String nombre, T tipo) {
        this.nombre = nombre;
        this.tipo = tipo;
    }
    public String getNombre() {
        return nombre;
    }
    public T getTipo() {
        return tipo;
    }
    public void mostrarDetalles() {
        System.out.println("Parte: " + nombre);
        System.out.println("Tipo: " + tipo);
    }
}
```

Ejercicio:
Clase Genérica
Vehículo

Ejercicio: Clase TallerMecanico: Método Genérico repararVehiculo / reemplazarParte

```
public class TallerMecanico {
    public static <T> void repararVehiculo(Vehiculo<T> vehiculo) {
        System.out.println("Reparando vehículo: " + vehiculo.getNombre());
        System.out.println("Tipo: " + vehiculo.getTipo());
        System.out.println("Reparación realizada.");
        System.out.println();
    }
public static <T> void reemplazarParte(Parte<T> parte) {
        System.out.println("Reemplazando parte: " + parte.getNombre());
        System.out.println("Tipo: " + parte.getTipo());
        System.out.println("Parte reemplazada con éxito.");
        System.out.println();
    }
}
```

Ejercicio: Clase Principal (Main)

```
public class Main {
    public static void main(String[] args) {
        Vehiculo<String> vehiculo1 = new Vehiculo<>("Auto", "Sedán");
        Vehiculo<Integer> vehiculo2 = new Vehiculo<>("Camión", 5);
        Parte<String> parte1 = new Parte<>("Motor", "Diesel");
        Parte<Double> parte2 = new Parte<>("Batería", 12.0);
        TallerMecanico.repararVehiculo(vehiculo1);
        TallerMecanico.repararVehiculo(vehiculo2);
        TallerMecanico.reemplazarParte(parte1);
        TallerMecanico.reemplazarParte(parte2);
    }
}
```

Ejercicio

- En este ejemplo, hemos creado las clases genéricas Vehiculo y Parte, y los métodos genéricos repararVehiculo y reemplazarParte en la clase TallerMecanico. Luego, en la clase principal (Main), hemos creado instancias de vehículos y partes con diferentes tipos de datos y hemos utilizado los métodos genéricos para reparar vehículos y reemplazar partes.
- Esto demuestra cómo se pueden usar clases genéricas y métodos genéricos en un contexto de taller mecánico en NetBeans.

Practica

Implementa el ejercicio visto.

Conclusiones

- A lo largo de esta presentación, hemos explorado la sintaxis de clases genéricas, los métodos genéricos, el uso de wildcards y cómo aplicar clases genéricas en colecciones.
- Con estos conocimientos, los programadores de Java están mejor preparados para escribir código más robusto y flexible.

Cierre

¿Qué hemos aprendido hoy?

Bibliografía

- MORENO PÉREZ, J. “Programación orientada a objetos”. RA-MA Editorial. https://tubiblioteca.utp.edu.pe/cgi-bin/koha/opac-detail.pl?biblionumber=31933
- Vélez Serrano, José. “Diseñar y programar, todo es empezar: una introducción a la Programación Orientada a Objetos usando UML y Java”. Dykinson. https://tubiblioteca.utp.edu.pe/cgi-bin/koha/opac-detail.pl?biblionumber=36368

Universidad Tecnológica del Perú