GENERICOS EN JAVA

**Conceptos Básicos de Genéricos**

1. **Definición de Genéricos:** Los genéricos permiten que las clases, interfaces y métodos operen sobre tipos especificados por el usuario, proporcionando una forma de reutilizar el código y mejorar la seguridad de tipos en tiempo de compilación.
2. **Sintaxis Básica:** La sintaxis básica de los genéricos utiliza ángulos < > para definir un parámetro de tipo.

class Caja<T> {

private T contenido;

public void setContenido(T contenido) {

this.contenido = contenido;

}

public T getContenido() {

return contenido;

}

}

1. **Parámetros de Tipo:**
   * T: Tipo (Type)
   * E: Elemento (Element, usado en colecciones)
   * K: Clave (Key)
   * V: Valor (Value)
   * N: Número (Number)

**Ejemplos Prácticos**

**1. Clase Genérica Simple**

public class Caja<T> {

private T contenido;

public void setContenido(T contenido) {

this.contenido = contenido;

}

public T getContenido() {

return contenido;

}

public static void main(String[] args) {

Caja<String> cajaDeTexto = new Caja<>();

cajaDeTexto.setContenido("Hola, Genéricos!");

System.out.println(cajaDeTexto.getContenido());

Caja<Integer> cajaDeNumeros = new Caja<>();

cajaDeNumeros.setContenido(123);

System.out.println(cajaDeNumeros.getContenido());

}

}

**2. Métodos Genéricos**

public class Util {

public static <T> void imprimir(T elemento) {

System.out.println(elemento);

}

public static void main(String[] args) {

Util.imprimir("Hola, Mundo!");

Util.imprimir(123);

Util.imprimir(45.67);

}

}

1. **Interfaces Genéricas**

**CREAR UNA INTERFAZ EN EL MISMO PAQUEE DONDE RESIDE EL MAIN**

interface Contenedor<T> {

void agregar(T elemento);

T obtener();

}

**CREAR UNA CLASE QUE IMPEMENTE LA INTERFAZ**

class ContenedorSimple<T> implements Contenedor<T> {

private T elemento;

@Override

public void agregar(T elemento) {

this.elemento = elemento;

}

@Override

public T obtener() {

return elemento;

}

**PROBAR EL METODO MAIN**

public static void main(String[] args) {

Contenedor<String> contenedor = new ContenedorSimple<>();

contenedor.agregar("Elemento Genérico");

System.out.println(contenedor.obtener());

}

}

**Bounded Type Parameters (Parámetros de Tipo Acotados)**

Puedes restringir los tipos que se pueden usar como parámetros de tipo.

**CREAR UNA CLASE QUE SOLO HEREDE DE NUMBER EN EL MISMO PAQUETE DONDE RESIDE EL MAIN**

public class CajaNumerica<T extends Number> {

private T numero;

public void setNumero(T numero) {

this.numero = numero;

}

public T getNumero() {

return numero;

}

**PROBARLA EN EL MAIN**

public static void main(String[] args) {

CajaNumerica<Integer> cajaDeEnteros = new CajaNumerica<>();

cajaDeEnteros.setNumero(10);

System.out.println(cajaDeEnteros.getNumero());

CajaNumerica<Double> cajaDeDoubles = new CajaNumerica<>();

cajaDeDoubles.setNumero(10.5);

System.out.println(cajaDeDoubles.getNumero());

}

}

**Wildcards (Comodines)**

Los comodines (?) se utilizan cuando deseas especificar un parámetro de tipo desconocido.

**Upper Bounded Wildcards**

**CREAR UNA CLASE EN EL MISMO PAQUETE DONDE RESIDE EL MAIN**

public static void imprimirLista(List<? extends Number> lista) {

for (Number numero : lista) {

System.out.println(numero);

}

}

public static void main(String[] args) {

List<Integer> listaDeEnteros = Arrays.asList(1, 2, 3);

List<Double> listaDeDoubles = Arrays.asList(1.1, 2.2, 3.3);

imprimirLista(listaDeEnteros);

imprimirLista(listaDeDoubles);

}

**Lower Bounded Wildcards**

public static void agregarNumeroALista(List<? super Integer> lista) {

lista.add(10);

}

public static void main(String[] args) {

List<Number> listaDeNumeros = new ArrayList<>();

agregarNumeroALista(listaDeNumeros);

System.out.println(listaDeNumeros);

}

**Erasure (Eliminación de Tipos)**

Java implementa los genéricos usando "type erasure" (eliminación de tipos), lo que significa que la información de tipos genéricos es borrada en tiempo de ejecución, y las verificaciones de tipo se realizan en tiempo de compilación.

Los **Lower Bounded Wildcards** (comodines acotados por abajo) en Java son una característica de los genéricos que te permite definir un tipo genérico que puede ser un tipo específico o cualquier superclase de ese tipo específico. Se utiliza la sintaxis ? super T, donde T es el tipo específico.

**Conceptos Clave**

* **Sintaxis:** List<? super T>
* **Propósito:** Permitir que el tipo sea T o cualquier superclase de T.
* **Uso Común:** Es útil cuando se necesita escribir métodos que pueden aceptar argumentos de múltiples tipos relacionados por herencia, pero se debe garantizar que los tipos pasen al menos una cierta clase base.

Los "Lower Bounded Wildcards" (comodines acotados por abajo) en Java se utilizan cuando deseas trabajar con un tipo genérico que es un supertipo (tipo padre) de un tipo específico. Esto te permite restringir un método para aceptar tipos que son supertypos de un tipo dado, brindando mayor flexibilidad.

**Sintaxis**

La sintaxis para un lower bounded wildcard es <? super T>, donde T es el tipo específico y el wildcard permitirá cualquier supertipo de T.

**Ejemplo de Uso**

Supongamos que tienes una lista y quieres agregar elementos a esta lista. Quieres asegurarte de que la lista pueda aceptar el tipo específico que deseas agregar o cualquier supertipo de ese tipo. Aquí hay un ejemplo:

**CREAR UNA CLASE EN EL MISMO PAQUETE DONDE RESIDE EL MAIN**

import java.util.ArrayList;

import java.util.List;

public class LowerBoundedWildcardExample {

public static void agregarNumeroALista(List<? super Integer> lista) {

lista.add(10);

lista.add(20);

}

public static void main(String[] args) {

List<Number> listaDeNumeros = new ArrayList<>();

agregarNumeroALista(listaDeNumeros);

System.out.println(listaDeNumeros);

List<Object> listaDeObjetos = new ArrayList<>();

agregarNumeroALista(listaDeObjetos);

System.out.println(listaDeObjetos);

}

}

**Explicación**

1. **Definición del Método**:

public static void agregarNumeroALista(List<? super Integer> lista)

Aquí, List<? super Integer> significa que el método agregarNumeroALista puede aceptar una lista de cualquier tipo que sea un supertipo de Integer. Esto incluye Integer, Number, y Object.

1. **Agregar Elementos a la Lista**:

lista.add(10);

lista.add(20);

Se agregan elementos de tipo Integer a la lista. Dado que la lista puede aceptar Integer o cualquier supertipo de Integer, esto es válido.

1. **Uso del Método**:

List<Number> listaDeNumeros = new ArrayList<>();

agregarNumeroALista(listaDeNumeros);

System.out.println(listaDeNumeros);

Una lista de Number se pasa al método, y es válida porque Number es un supertipo de Integer.

List<Object> listaDeObjetos = new ArrayList<>();

agregarNumeroALista(listaDeObjetos);

System.out.println(listaDeObjetos);

También se puede pasar una lista de Object, porque Object es un supertipo de Integer.

**Beneficios de los Lower Bounded Wildcards**

* **Flexibilidad**: Permiten que un método trabaje con una jerarquía de tipos en lugar de un solo tipo específico.
* **Seguridad de Tipos**: Ayudan a asegurar que los tipos de datos sean manejados correctamente, previniendo errores de tipo en tiempo de compilación.

**Casos de Uso Comunes**

* **Métodos de Adición**: Cuando necesitas agregar elementos a una colección y quieres aceptar colecciones que pueden contener el tipo específico o cualquier supertipo de ese tipo.
* \*\*Procesamiento

**Recursos Adicionales**

Para profundizar en los genéricos en Java, aquí tienes algunos recursos recomendados:

* **Libros:**
  + "Effective Java" de Joshua Bloch (Capítulo sobre Genéricos)
  + "Java Generics and Collections" de Maurice Naftalin y Philip Wadler
* **Documentación Oficial:**
  + [Tutorial de Genéricos en Java](https://docs.oracle.com/javase/tutorial/java/generics/index.html)
  + [Java SE Documentation - Generics](https://docs.oracle.com/javase/specs/jls/se11/html/jls-4.html#jls-4.5)
* **Artículos y Blogs:**
  + Guía de Genéricos en Java
  + Java Generics Tutorial

**Como crear una clase Genérica que simule las operaciones de un CRUD en memoria**

**Veamos la siguiente clase Genérica llamada InMemoryRepository<T> en el mismo paquete donde reside el main:**

public class InMemoryRepository<T> {  
 protected Map<Long, T> data = new HashMap<>();  
 protected AtomicLong idGenerator = new AtomicLong();  
  
 public T save(T entity) {  
 long id = idGenerator.incrementAndGet();  
 *// Suponiendo que las entidades tienen un método setId* try {  
 String clase;  
 entity.getClass().getMethod("setId", Long.class).invoke(entity, id);  
 clase = entity.getClass().getName();  
 System.*out*.println(clase + " id :" + id);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
  
 data.put(id, entity);  
 return entity;  
 }  
  
 public Optional<T> findById(Long id) {  
 return Optional.*ofNullable*(data.get(id));  
 }  
  
  
  
 public List<T> findAll() {  
 return new ArrayList<>(data.values());  
 }  
  
  
 public Optional<T> genericUpdate(Long id, T updatedEntity) {  
 if (!data.containsKey(id)) {  
 return Optional.*empty*();  
 }  
  
 try {  
 *// Establecer el mismo ID en la entidad actualizada para mantener la coherencia* Method setIdMethod = updatedEntity.getClass().getMethod("setId", Long.class);  
 setIdMethod.invoke(updatedEntity, id);  
  
 data.put(id, updatedEntity);  
 return Optional.*of*(updatedEntity);  
 } catch (Exception e) {  
 e.printStackTrace();  
 return Optional.*empty*();  
 }  
 }  
  
 public Optional<T> genericDelete(Long id) {  
 if (!data.containsKey(id)) {  
 return Optional.*empty*();  
 }  
 return Optional.*ofNullable*(data.remove(id));  
 }  
  
 public List<T> genericFindByField(String fieldName, Object value) {  
 List<T> results = new ArrayList<>();  
 try {  
 for (T entity : data.values()) {  
 Method getFieldMethod = entity.getClass().getMethod("get" + capitalize(fieldName));  
 Object fieldValue = getFieldMethod.invoke(entity);  
 if (fieldValue != null && fieldValue.equals(value)) {  
 results.add(entity);  
 }  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 return results;  
 }  
  
 private String capitalize(String str) {  
 if (str == null || str.isEmpty()) {  
 return str;  
 }  
 return str.substring(0, 1).toUpperCase() + str.substring(1);  
 }  
}

EXPLICACION

**La clase InMemoryRepository<T> es un repositorio genérico en memoria que permite gestionar entidades de cualquier tipo T. Esta clase proporciona métodos para guardar, buscar, actualizar y eliminar entidades, así como para encontrar entidades por campos específicos. Aquí está una explicación detallada de cada componente y método de la clase:**

**Componentes de la Clase**

1. **protected Map<Long, T> data = new HashMap<>();**
   * Un mapa que almacena las entidades en memoria. La clave es un Long que representa el ID de la entidad y el valor es la propia entidad.
2. **protected AtomicLong idGenerator = new AtomicLong();**
   * Un generador de IDs que se incrementa automáticamente cada vez que se guarda una nueva entidad, asegurando que cada entidad tenga un ID único.

**Métodos de la Clase**

1. **public T save(T entity)**
   * Guarda una entidad en el repositorio.
   * Genera un nuevo ID para la entidad utilizando idGenerator.
   * Utiliza reflexión para invocar el método setId de la entidad y asignarle el ID generado.
   * Almacena la entidad en el mapa data.
   * Retorna la entidad guardada.

public T save(T entity) {

long id = idGenerator.incrementAndGet();

try {

entity.getClass().getMethod("setId", Long.class).invoke(entity, id);

System.out.println(entity.getClass().getName() + " id :" + id);

} catch (Exception e) {

e.printStackTrace();

}

data.put(id, entity);

return entity;

}

1. **public Optional<T> findById(Long id)**
   * Busca una entidad por su ID.
   * Retorna un Optional que puede contener la entidad si se encuentra, o estar vacío si no se encuentra.

public Optional<T> findById(Long id) {

return Optional.ofNullable(data.get(id));

}

1. **public List<T> findAll()**
   * Retorna una lista con todas las entidades almacenadas en el repositorio.

public List<T> findAll() {

return new ArrayList<>(data.values());

}

1. **public Optional<T> genericUpdate(Long id, T updatedEntity)**
   * Actualiza una entidad existente en el repositorio.
   * Comprueba si el ID existe en el mapa data.
   * Si existe, utiliza reflexión para establecer el mismo ID en la entidad actualizada.
   * Almacena la entidad actualizada en el mapa y retorna un Optional que contiene la entidad actualizada.
   * Si no existe, retorna un Optional.empty().

public Optional<T> genericUpdate(Long id, T updatedEntity) {

if (!data.containsKey(id)) {

return Optional.empty();

}

try {

Method setIdMethod = updatedEntity.getClass().getMethod("setId", Long.class);

setIdMethod.invoke(updatedEntity, id);

data.put(id, updatedEntity);

return Optional.of(updatedEntity);

} catch (Exception e) {

e.printStackTrace();

return Optional.empty();

}

}

1. **public Optional<T> genericDelete(Long id)**
   * Elimina una entidad del repositorio por su ID.
   * Retorna un Optional que puede contener la entidad eliminada si se encuentra, o estar vacío si no se encuentra.

public Optional<T> genericDelete(Long id) {

if (!data.containsKey(id)) {

return Optional.empty();

}

return Optional.ofNullable(data.remove(id));

}

1. **public List<T> genericFindByField(String fieldName, Object value)**
   * Busca entidades en el repositorio por un campo específico y su valor.
   * Utiliza reflexión para obtener el valor del campo especificado en cada entidad.
   * Retorna una lista de entidades que tienen el valor especificado en el campo.

public List<T> genericFindByField(String fieldName, Object value) {

List<T> results = new ArrayList<>();

try {

for (T entity : data.values()) {

Method getFieldMethod = entity.getClass().getMethod("get" + capitalize(fieldName));

Object fieldValue = getFieldMethod.invoke(entity);

if (fieldValue != null && fieldValue.equals(value)) {

results.add(entity);

}

}

} catch (Exception e) {

e.printStackTrace();

}

return results;

}

1. **private String capitalize(String str)**
   * Capitaliza la primera letra de una cadena. Este método es útil para construir nombres de métodos getter basados en nombres de campos.

private String capitalize(String str) {

if (str == null || str.isEmpty()) {

return str;

}

return str.substring(0, 1).toUpperCase() + str.substring(1);

}

**Ejemplo de Uso del InMemoryRepository aplicado a la clase persona**

La clase InMemoryRepository permite gestionar entidades en memoria de manera genérica. Por ejemplo, puedes crear un repositorio para una clase Persona de la siguiente manera:

**CREAR LA CLASE PERSONA EN EL MISMO PAQUETE DONDE RESIDE EL MAIN**

public class Persona {

private Long id;

private String nombre;

private String apellido;

private Integer edad;

// Getters y setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

// Otros getters y setters, así como métodos adicionales

}

**EN EL MAIN**

public class Main {

public static void main(String[] args) {

InMemoryRepository<Persona> repo = new InMemoryRepository<>();

Persona persona = new Persona();

persona.setNombre("Juan");

persona.setApellido("Perez");

persona.setEdad(30);

// Guardar persona

repo.save(persona);

// Encontrar por ID

Optional<Persona> encontrada = repo.findById(1L);

encontrada.ifPresent(p -> System.out.println(p.getNombre()));

// Encontrar todas

List<Persona> todas = repo.findAll();

todas.forEach(p -> System.out.println(p.getNombre()));

}

}

Este código muestra cómo crear un repositorio para la clase Persona, guardar una instancia, buscarla por ID y obtener todas las instancias almacenadas.

**Ejercitación Propuesta:**

**Tomando la estructura anterior crear 5 personas distintas y probar toda la funcionalidad de los métodos de la clase InMemoryRepository.**

**Cuestionario sobre Genéricos en Java**

1. **¿Qué son los genéricos en Java?**
   * A) Una forma de escribir código independiente del tipo.
   * B) Una forma de evitar errores de compilación.
   * C) Una manera de mejorar la legibilidad del código.
   * D) Un tipo de colección en Java.
   * **Respuesta Correcta: A**
2. **¿Qué permite el uso de genéricos en Java?**
   * A) El uso de operadores sobrecargados.
   * B) La reutilización del código con diferentes tipos de datos.
   * C) La creación de nuevos tipos de datos.
   * D) La optimización del rendimiento.
   * **Respuesta Correcta: B**
3. **¿Cuál es la sintaxis correcta para declarar una clase genérica en Java?**
   * A) class MyClass<T> { }
   * B) class MyClass[T] { }
   * C) class MyClass { <T> }
   * D) class MyClass { T }
   * **Respuesta Correcta: A**
4. **¿Qué significa <T extends Number> en una declaración de genéricos?**
   * A) T puede ser cualquier tipo.
   * B) T debe ser una subclase de Number.
   * C) T debe ser una superclase de Number.
   * D) T debe ser exactamente Number.
   * **Respuesta Correcta: B**
5. **¿Cuál es el propósito de los "wildcards" (comodines) en genéricos?**
   * A) Para permitir cualquier tipo en un método.
   * B) Para restringir el tipo de datos a un conjunto específico.
   * C) Para facilitar el uso de herencia con tipos genéricos.
   * D) Para optimizar el rendimiento del código.
   * **Respuesta Correcta: C**
6. **¿Qué representa List<?> en una declaración de método?**
   * A) Una lista de cualquier tipo específico.
   * B) Una lista que no acepta elementos.
   * C) Una lista de tipos desconocidos.
   * D) Una lista vacía.
   * **Respuesta Correcta: C**
7. **¿Qué son los "upper bounded wildcards" en genéricos?**
   * A) Wildcards que permiten cualquier tipo.
   * B) Wildcards que permiten supertypos.
   * C) Wildcards que permiten subtipos.
   * D) Wildcards que restringen el tipo a ser exactamente uno.
   * **Respuesta Correcta: C**
8. **¿Cómo se declara un método que acepta cualquier supertipo de Integer?**
   * A) public void metodo(List<? super Integer> list)
   * B) public void metodo(List<Integer> list)
   * C) public void metodo(List<? extends Integer> list)
   * D) public void metodo(List<?> list)
   * **Respuesta Correcta: A**
9. **¿Cuál de las siguientes declaraciones es correcta para crear una instancia de una clase genérica?**
   * A) MyClass<int> obj = new MyClass<>();
   * B) MyClass<T> obj = new MyClass<T>();
   * C) MyClass<String> obj = new MyClass<>();
   * D) MyClass<?> obj = new MyClass<?>()
   * **Respuesta Correcta: C**
10. **¿Qué beneficio proporciona el uso de genéricos en colecciones como ArrayList?**
    * A) Mayor rendimiento.
    * B) Seguridad de tipos en tiempo de compilación.
    * C) Menor uso de memoria.
    * D) Capacidad de almacenar solo tipos primitivos.
    * **Respuesta Correcta: B**

Este cuestionario cubre una variedad de conceptos clave relacionados con los genéricos en Java, incluyendo su sintaxis, uso, y los beneficios que proporcionan.