**Java: Un Ejemplo Detallado**

En Java, los genéricos nos permiten escribir código más seguro y reutilizable, trabajando con tipos de forma más abstracta. Sin embargo, a veces necesitamos más flexibilidad sobre los tipos exactos que pueden ser usados. Aquí es donde entran en juego los *wildcards* (comodines).

Un Lower Bounded Wildcard (? super T) se utiliza para indicar que un tipo genérico puede ser el tipo T o **cualquier supertipo de T**. Esto es particularmente útil cuando queremos *agregar* elementos a una colección de forma segura, garantizando que el tipo de elemento que agregamos sea compatible con el tipo que la lista espera.

**El Código**

Aquí está el código mejorado y completo para ilustrar ? super Integer:

Java

import java.util.ArrayList; // Necesario para crear instancias de ArrayList

import java.util.List; // Interfaz List, buena práctica usar interfaces

/\*\*

\* Clase que demuestra el uso de Lower Bounded Wildcard (? super T).

\*/

public class LowerBoundedWildcardExample {

/\*\*

\* Método que acepta una lista que puede contener Integer o cualquier supertipo de Integer.

\* Esto significa que podemos agregar de forma segura objetos de tipo Integer

\* o cualquier subtipo de Integer a esta lista.

\*

\* @param lista La lista a la que se agregarán los números enteros.

\*/

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

// Podemos agregar Integer directamente

lista.add(10);

lista.add(20);

// También podríamos agregar subtipos de Integer, aunque en este caso Integer no tiene subtipos directos.

// Por ejemplo, si tuviéramos un MiInteger extends Integer (no posible para Integer final class)

// lista.add(new MiInteger(30));

System.out.println("Elementos agregados a la lista en el método: " + lista);

}

/\*\*

\* Método principal para ejecutar el ejemplo.

\* Muestra cómo diferentes tipos de listas (Number, Object) pueden ser pasadas

\* al método con el wildcard lower bounded.

\*/

public static void main(String[] args) {

System.out.println("--- Demostración con List<Number> ---");

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

System.out.println("Lista inicial de números: " + listaDeNumeros);

agregarNumeroALista(listaDeNumeros);

System.out.println("Lista de números después de agregar: " + listaDeNumeros); // Output: [10, 20]

System.out.println("\n--- Demostración con List<Object> ---");

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

listaDeObjetos.add("Hola"); // Podemos agregar Strings a una List<Object>

listaDeObjetos.add(3.14); // Podemos agregar Doubles a una List<Object>

System.out.println("Lista inicial de objetos: " + listaDeObjetos);

agregarNumeroALista(listaDeObjetos);

System.out.println("Lista de objetos después de agregar: " + listaDeObjetos); // Output: [Hola, 3.14, 10, 20]

// --- Ejemplos de lo que NO se puede pasar ---

System.out.println("\n--- Ejemplos de compatibilidad (errores de compilación) ---");

// List<Double> listaDeDoubles = new ArrayList<>();

// agregarNumeroALista(listaDeDoubles); // ERROR DE COMPILACIÓN: Double no es supertipo de Integer

// List<String> listaDeStrings = new ArrayList<>();

// agregarNumeroALista(listaDeStrings); // ERROR DE COMPILACIÓN: String no es supertipo de Integer

}

}

**Explicación Detallada**

**1. La Firma del Método: public static void agregarNumeroALista(List<? super Integer> lista)**

* **List**: Indica que el método espera un objeto que implementa la interfaz List.
* **<?>**: El signo de interrogación ? es el comodín (*wildcard*). Significa "algún tipo desconocido".
* **super Integer**: Esta es la clave. La palabra clave super define un *límite inferior* para el tipo del comodín. Significa que el tipo de la lista (?) debe ser Integer o **un supertipo de Integer**.
  + **¿Qué significa esto para la lista?**
    - Una List<Integer> es válida, ya que Integer es Integer.
    - Una List<Number> es válida, ya que Number es un supertipo de Integer.
    - Una List<Object> es válida, ya que Object es el supertipo más alto de Integer (y de todo).
    - Una List<Double> **NO** es válida, porque Double no es un supertipo de Integer (son hermanos, ambos subtipos de Number).
    - Una List<String> **NO** es válida, porque String no es un supertipo de Integer.

**2. La Operación lista.add(10);**

* Dentro del método agregarNumeroALista, podemos llamar a lista.add(10) (donde 10 es un Integer).
* **¿Por qué es esto seguro?** El compilador sabe que lista es una colección que puede contener Integers o supertipos de Integer. Si la lista es List<Integer>, puede tomar Integer. Si la lista es List<Number>, puede tomar Integer (ya que Integer es un Number). Si la lista es List<Object>, puede tomar Integer (ya que Integer es un Object). En todos los casos, agregar un Integer es una operación válida y segura para el tipo subyacente de la lista.
* Con ? super T, siempre puedes agregar elementos de tipo T (o subtipos de T) a la colección. Esto es lo que se conoce como un **"Consumer" (Consumidor)**: la lista está siendo utilizada para "consumir" o "recibir" elementos.

**3. El Método main**

* **List<Number> listaDeNumeros = new ArrayList<>();**: Aquí creamos una lista que está declarada para contener objetos Number. Number es un supertipo de Integer. Cuando pasamos listaDeNumeros a agregarNumeroALista, cumple con el requisito List<? super Integer>.
* **List<Object> listaDeObjetos = new ArrayList<>();**: De manera similar, Object es el supertipo más alto, por lo que List<Object> también es compatible con List<? super Integer>.
* **System.out.println(...)**: Después de llamar a agregarNumeroALista, se imprimen los contenidos de las listas, mostrando que los Integers (10 y 20) han sido agregados exitosamente.
* **Comentarios de errores de compilación**: Las líneas comentadas en el main (List<Double>, List<String>) ilustran que estos tipos de listas no son compatibles con List<? super Integer>, ya que Double y String no son supertipos de Integer.

**El Principio PECS (Producer Extends, Consumer Super)**

Este ejemplo es una aplicación directa del principio **PECS (Producer Extends, Consumer Super)**, una regla mnemotécnica para recordar cuándo usar extends y cuándo usar super con *wildcards*:

* **P**roducer **E**xtends: Si tu colección va a **producir** (devolver) elementos del tipo T, usa ? extends T. (Puedes leer T o subtipos de T, pero no puedes agregar nada más que null).
* **C**onsumer **S**uper: Si tu colección va a **consumir** (aceptar/agregar) elementos del tipo T, usa ? super T. (Puedes agregar T o subtipos de T, pero al leer obtendrás Object).

En este ejemplo, agregarNumeroALista es un **consumidor** de Integers, por lo tanto, usamos ? super Integer.

Este ejemplo demuestra cómo ? super T proporciona la flexibilidad necesaria para trabajar con jerarquías de tipos al momento de agregar elementos a las colecciones, garantizando la seguridad de tipos en tiempo de compilación.