Los genéricos en Java son una poderosa característica del lenguaje que permite a los desarrolladores escribir clases, interfaces y métodos con tipos parametrizados. Esto proporciona una mayor flexibilidad y seguridad en tiempo de compilación, ya que permite detectar errores de tipo antes de que el código se ejecute.

Los genéricos permiten definir clases, interfaces y métodos que operan sobre tipos parametrizados. En lugar de especificar un tipo concreto, se utiliza un parámetro de tipo que actúa como un marcador de posición para el tipo real que se utilizará.

Ejemplo de Genéricos

Por ejemplo podemos crear una clase llamada Box que podría tener un objeto de cualquier clase, es decir. Un objeto genérico.

```
public class Box<T> {
    private T item;

    public void set(T item) {
        this.item = item;
    }

    public T get() {
        return item;
    }
}
```

En este ejemplo, T es el parámetro de tipo. Cuando creas una instancia de Box, puedes especificar el tipo que deseas que contenga:

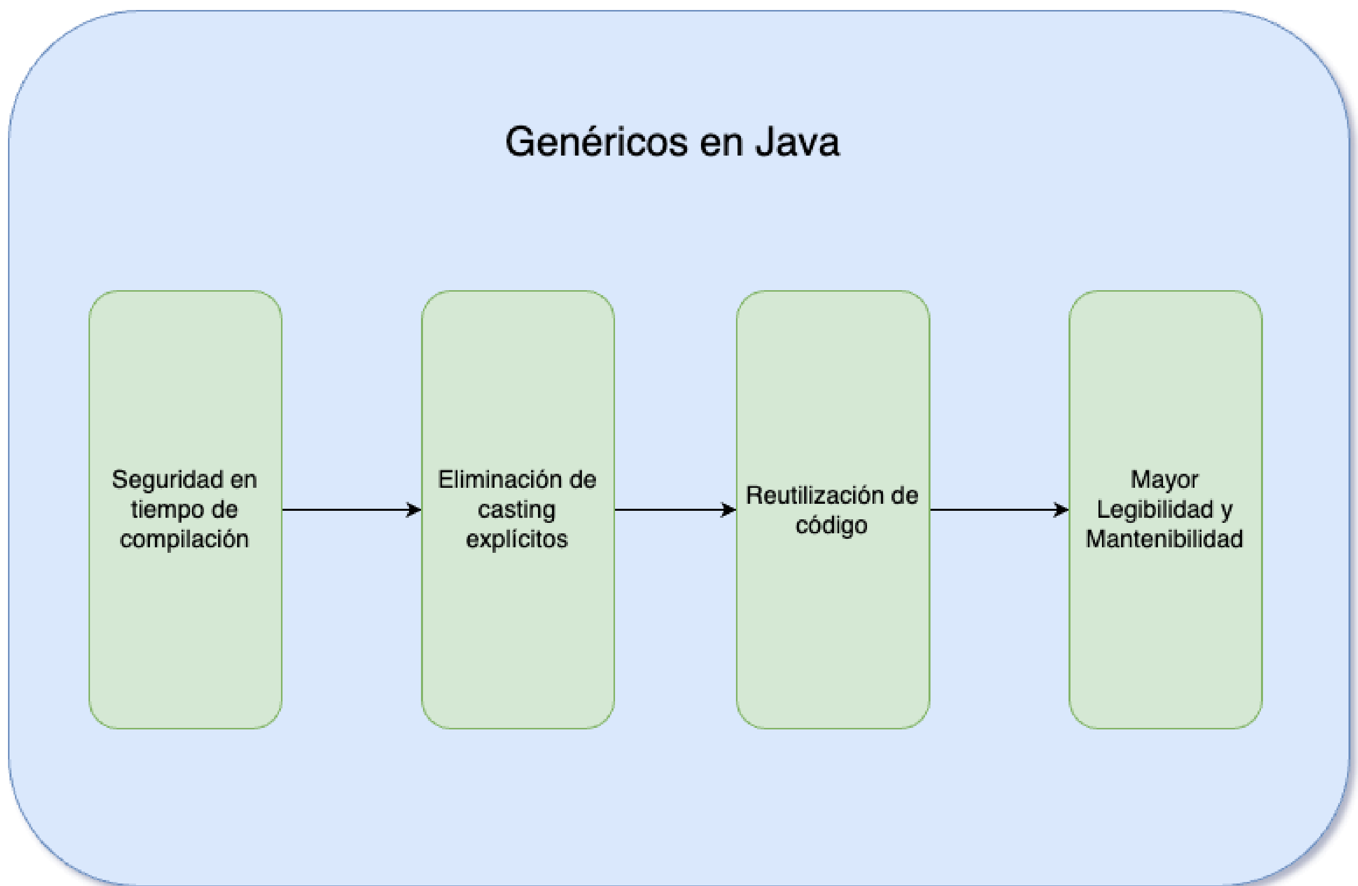
```
Box<String> stringBox = new Box<>();
stringBox.set("Hello");
String value = stringBox.get();

Box<Integer> integerBox = new Box<>();
```

```
integerBox.set(123);
Integer number = integerBox.get();
```

Si lo analizas bien hay muchas clases e interfaces que usamos día a día que están construidas a partir de genéricos como por ejemplo List, ArrayList, Map, HashMap, entre muchas otras.

Beneficios/Ventajas de los genéricos



1. Seguridad de Tipo en Tiempo de Compilación

Los genéricos permiten detectar errores de tipo en tiempo de compilación en lugar de en tiempo de ejecución. Esto reduce la probabilidad de errores y excepciones relacionadas con tipos.

```
Box<String> stringBox = new Box<>();
stringBox.set("Hello");
// stringBox.set(123); // Error de compilación: incompatible types
```

Sin el uso de genéricos necesitaríamos primero compilar la aplicación, hacer un cast a nivel de código y esperar que una vez ejecutado el programa nos mostrará un error de cast.

2. Eliminación de Castings Explícitos

Ampliando un poco más lo mencionado al final del punto anterior con los genéricos, no es necesario realizar castings explícitos, lo que hace que el código sea más limpio y fácil de leer.

Sin genéricos:

```
List list = new ArrayList();
list.add("Hello");
String value = (String) list.get(0); // Casting explícito
```

Con genéricos:

```
List<String> list = new ArrayList<>();
list.add("Hello");
String value = list.get(0); // No se necesita casting
```

3. Reutilización de Código

Los genéricos permiten escribir código más reutilizable y flexible. Puedes crear clases, métodos e interfaces que funcionen con cualquier tipo de dato.

Veamos primero una clase para “simular” lo que hacemos con un Map

```
public class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
```

```
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}
```

Si logras imaginarlo esa clase que acabamos de crear es altamente reutilizable, al igual que lo es la interfaz Map y todas sus implementaciones, esa es la magia y el poder que tienen los generics.

Uso de la clase “Pair”

```
Pair<String, Integer> pair = new Pair<>("Age", 30);
String key = pair.getKey();
Integer value = pair.getValue();
```

Por si fuera poco también podemos definir **métodos genéricos** aunque la clase de la que son parte no utilice genéricos en su definición, esto es muy común por ejemplo en clases de utilidades.

```
public class Util {
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.println(element);
        }
    }
}
```

Y así lo usaríamos:

```
String[] stringArray = {"Hello", "World"};
Integer[] intArray = {1, 2, 3};

Util.printArray(stringArray);
```

```
Util.printArray(intArray);
```

Otra opción es hacer uso de interfaces genéricas, creo que para implementar algo con interfaces genéricas es necesario un poco de conocimiento en diseño avanzado de clases, pero vamos a hacer un ejemplo simulando lo que hacemos con la interfaz java.util.List.

```
public interface Container<T> {
    void add(T item);
    T get(int index);
}
```

La interfaz java.util.List tiene varias implementaciones pero la más populares es ArrayList, nosotros crearemos algo similar:

```
public class ListContainer<T> implements Container<T> {
    private List<T> list = new ArrayList<>();

    @Override
    public void add(T item) {
        list.add(item);
    }

    @Override
    public T get(int index) {
        return list.get(index);
    }
}
```

Y si queremos usar la interfaz con su implementación lo haríamos así:

```
Container<String> stringContainer = new ListContainer<>();
stringContainer.add("Hello");
String value = stringContainer.get(0);
```

La “magia” aquí es que podrías crear otra implementación que solo agregue items pero que no implemente ninguna función de “get”, combinar las interfaces con genéricos puede llevar tu código a un nivel superior.

Wildcards en Genéricos en Java

La traducción puede resultar compleja pero los wildcards son una especie de “comodines” que nos permiten extender un poco más la funcionalidad de los genéricos, ten presente que generalmente están representados por un signo de interrogación de cierre (?) y hay 3 tipos.

Unbounded Wildcards

El wildcard sin límite (?) se utiliza cuando cualquier tipo es aceptable. A diferencia de T incluso en tiempo de compilación puedes tratar su valor como un “Object”, se usa básicamente cuando no sabes que vas a recibir. Mientras que el tipo T se usa cuando tienes claro el tipo de objeto que vas a recibir o el tipo que deseas contener en tu genérico.

```
public void printList(List<?> list) {
    for (Object elem : list) {
        System.out.println(elem);
    }
}
```

Bounded Wildcards

Los wildcards con límite se utilizan para restringir el tipo permitido. Hay dos tipos:

1. **Upper Bounded Wildcards** (<? extends T>): Permite cualquier tipo que sea una subclase de T.

```
public void processList(List<? extends Number> list) {
    for (Number num : list) {
        System.out.println(num);
    }
}
```

Recuerda pensar de forma genérica, T puedes tomar el valor de cualquier tipo de clase, por eso aquí T pasa a ser Number pero podría ser Integer, String, Double, en fin. Cualquier tipo de clase.

2. **Lower Bounded Wildcards** (<? super T>): Permite cualquier tipo que sea una superclase de T.

```
public void addNumbers(List<? super Integer> list) {
    list.add(1);
    list.add(2);
}
```

¿Por qué siempre los genéricos en Java se definen como T? ¿Hay más letras acaso?

La respuesta es sí. De hecho hay una convención común de letras que solemos utilizar al momento de crear una clase, interfaz o método genérico.

1. **T – Type**: Se usa generalmente para definir un tipo genérico, recuerda que en POO cuando hablamos de Type generalmente hace referencia a cualquier clase, ya sean las que vienen por defecto en el lenguaje o alguna que nosotros podamos crear.

```
public class Box<T> {
    private T item;
    public void set(T item) { this.item = item; }
    public T get() { return item; }
}
```

2. **E – Element**: Se usa generalmente en colecciones, por ejemplo la clase java.util.Collection recibe E como un atributo genérico en su definición.

```
public interface Collection<E> {
    void add(E element);
    E get(int index);
}
```

3. **K – Key** y **V – Value**: Se usa generalmente en objetos de tipo clave valor, mayor representante de esto son los mapas.

```
public interface Map<K, V> {
    void put(K key, V value);
    V get(K key);
}
```

4. **N – Number**: Se usa generalmente para genéricos que reciben números.

```java
public class NumericBox<N extends Number> {
    private N number;
    public void set(N number) { this.number = number; }
    public N get() { return number; }
}
```

Entender estas convenciones te ayuda a dos cosas por un lado entiendes más con que tipo de genérico estás tratando y por otro lado te puede dar una mejor perspectiva al diseñar tus propios genéricos.

¿Otros lenguajes implementan genéricos además de Java?

Sí, si no me equivoco cualquier lenguaje de programación que soporte programación orientada a objetos, tiene genéricos. Personalmente he implementado generics en TypeScript, PHP, Java y Python.

Así que te ánimo a explorarlo, ten presente que no solo se recomienda usarlo con POO puedes mezclarlos con otros estilos de programación.