Material descargable

Genéricos

egg

¡Hola! 👋 Te damos la bienvenida a Genéricos.

Imagina que estás desarrollando una aplicación en Java y necesitas una clase "Contenedor" que pueda contener o envolver cualquier tipo de objeto. Dado que todas las clases en Java son subclases de la clase Object, una opción que podrías considerar sería utilizar Object en tu clase Contenedor. De esta manera, tu clase podría aceptar y guardar cualquier tipo de objeto.

En una parte diferente de tu código, decides utilizar este Contenedor para guardar una cadena de texto "Hola Mundo". Más adelante, intentas recuperar este objeto y, dado que lo que guardaste fue una cadena de texto, esperas obtener una cadena de texto de vuelta.

Aquí es donde te encuentras con un problema. Como tu clase Contenedor utiliza Object, cuando obtienes el objeto de vuelta, tienes que hacer un casting a su tipo original. Pero, ¿qué sucede si no recuerdas qué tipo de objeto guardaste? O peor aún, ¿qué sucede si haces un casting al tipo incorrecto?.

Lo anterior conducirá a un ClassCastException, un error en tiempo de ejecución. Este problema surge debido a la falta de seguridad de tipo en tiempo de compilación en tu clase Contenedor. Para solucionar esto necesitamos una forma de proporcionar seguridad de tipo a nuestras clases y métodos en tiempo de compilación para evitar este tipo de errores en tiempo de ejecución.

Y aquí es donde entran los genéricos en Java.

¡Que comience el viaje! 🚀

Genéricos

Los genéricos fueron introducidos en Java 5 para proporcionar un mecanismo que permitiera la seguridad de tipo en tiempo de compilación. Con los genéricos,

podemos decirle al compilador de Java qué tipo de objeto se espera, permitiendo que el compilador prevenga errores de casting incorrectos.

Los genéricos nos permiten definir un tipo de dato como parámetro al definir clases, interfaces y métodos. Cuando utilizamos estos métodos o clases, podemos especificar el tipo de dato exacto que esperamos.

Por ejemplo, aquí podemos ver una comparativa de la clase Contenedor de la introducción, y la misma clase pero escrita para usar genéricos:

```
// Contenedor sin generico
public class Contenedor {
    private Object objeto;

    public void set(Object objeto) {
        this.objeto = objeto;
    }

    public Object get() {
        return this.objeto;
    }
}
public class Main {
    public static void main(String[] args) {
        Contenedor contenedor = new Contenedor();
        contenedor.guardar("Hola Mundo");
        Integer entero = (Integer) contenedor.obtener(); // Esto lanzará
ClassCastException
    }
}
```

```
// Contenedor con generico
public class Contenedor<T> {
    private T objeto;

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

```
    public T get() {
        return this.objeto;
    }
}
public class Main {
    public static void main(String[] args) {
        Contenedor<String> contenedor = new Contenedor<>();
        contenedor.guardar("Hola Mundo");
        String cadena = contenedor.obtener();
    }
}
```

Aquí, T es un parámetro de tipo que se reemplaza por un tipo real cuando creamos una instancia de Contenedor. Si intentamos almacenar cualquier otro tipo de objeto, obtendremos un error de compilación, por lo tanto, los genéricos nos ayudan a evitar errores en tiempo de ejecución. También hacen que nuestro código sea más legible y reutilizable, ya que podemos utilizar la misma clase o método con diferentes tipos de datos.

Sintaxis de los Genéricos en Java

Los genéricos en Java tienen una sintaxis específica que vamos a desglosar a continuación.

- **Símbolos <>:** En la declaración de genéricos, utilizamos los símbolos "<>" (menor que y mayor que) para indicar que estamos utilizando genéricos. Por ejemplo, en la declaración de una clase genérica Caja<T>, los "<>" indican que T es un parámetro de tipo.
- **Identificadores de los tipos de parámetro**: El identificador “T” que utilizamos entre los símbolos "<>" es un nombre de tipo de parámetro. Podríamos declarar y usar un identificador llamado “TIPO” pero “T” es comúnmente utilizado por convención y representa "Type" (Tipo). Sin embargo, no estamos limitados a usar solo “T”, podemos usar cualquier identificador válido en Java. Algunos identificadores comunes incluyen:
    - E: Elemento (usado en general en contextos de colecciones)
    - K: Key (Llave, en el caso de mapas)
    - V: Value (Valor, también usado en mapas)
    - N: Number (Número, para tipos numéricos)

- S, U, V, etc: Para representar el segundo, tercero, cuarto, etc. tipos en casos donde se necesita más de un parámetro de tipo.

Además de estos identificadores, también está el wildcard **"?"**, que es un símbolo especial utilizado en genéricos para representar "cualquier tipo". El comodín es particularmente útil cuando estás trabajando con objetos de varios tipos diferentes y no necesitas (o quieres) especificar un tipo concreto.

Por ejemplo, si tienes una clase Caja**<T>**, y quieres un método que pueda trabajar con una caja que contenga cualquier tipo de objeto, podrías usar el comodín "**?",** en la firma del método, así:

```
public void algunMetodo(Caja<?> caja) {
    // Aquí puedes trabajar con la caja, pero no sabes de qué tipo son los objetos que contiene ni puedes hacer referencia a su tipo
}
```

Mientras que los identificadores de tipo como T permiten definir y referenciar un tipo específico en varios lugares en tu código, el comodín "**?"** te permite trabajar con objetos de cualquier tipo, pero sin la capacidad de referirte a ese tipo específicamente. Ambos conceptos son fundamentales para entender y trabajar con genéricos en Java.

Uso de múltiples parámetros de tipo

Podemos utilizar más de un parámetro de tipo en nuestros genéricos. Por ejemplo, podríamos tener una clase Pareja **<T, U>** que almacena dos objetos de diferentes tipos:

```
public class Pareja<T, U> {
    private T primerElemento;
    private U segundoElemento;
}
```

Limitaciones de Tipo Superior e Inferior: extends y super

En el contexto de genéricos, a veces es necesario restringir los tipos de parámetros que puedes pasar a un método o clase. Java proporciona dos mecanismos para lograr esto: extends (para limitaciones de tipo superior) y super (para limitaciones de tipo inferior).

- **Extends (Limitación de Tipo Superior):** La palabra clave extends se utiliza para indicar que un tipo genérico debe ser una subclase de un tipo específico, o implementar una interfaz específica. Esto se conoce como limitación de tipo superior.

  Por ejemplo, supongamos que tenemos una caja genérica que puede contener cualquier tipo de número:

```
public class Caja<T extends Number> {
    private T contenido;

    public Caja(T contenido) {
        this.contenido = contenido;
    }

    public T obtener() {
        return contenido;
    }
}
```

  Aquí, T extends Number significa que el tipo T debe ser Number o una subclase de Number. Por lo tanto, podrías tener una Caja<Integer>, una Caja<Double>, etc., pero no una Caja<String> o Caja<Object>.

- **Super (Limitación de Tipo Inferior):** La palabra clave super se utiliza para indicar que un tipo genérico debe ser una superclase de un tipo específico. Esto se conoce como limitación de tipo inferior.

  Por ejemplo, siguiendo con el ejemplo anterior:

```
public class Caja<T super Integer> {
    private T contenido;

    public Caja(T contenido) {
        this.contenido = contenido;
    }

    public T obtener() {
        return contenido;
    }
```

```
}
```

Ahora, "super Integer" significa que puedes pasar al método una lista de cualquier tipo que sea una superclase de Integer. Por lo tanto, podrías pasar una Caja<Number> o Caja<Object>, pero no una Caja<Double> o Caja<String>, ya que Double y String no son superclases de Integer.

Una de las ventajas de usar extends y super es que ahora tienes acceso a los métodos de la clase que definiste como límite.

Por ejemplo si tienes una caja genérica que limita su tipo de parámetro a Animal o subclases de Animal, y animal tiene un método comer(), ahora puedes acceder a él en la clase genérica.

```
public class Caja<T extends Animal> {
    private T contenido;

    public Caja(T contenido) {
        this.contenido = contenido;
    }

    public void hacerComerAlAnimal() {
        contenido.comer();
    }
}
```

Diferencia entre <T> y <?>

La diferencia es bastante sutil:

- **<T>** es un parámetro de tipo que representa un tipo específico.
- **<?>** es un comodín que representa "cualquier tipo".

Veamos unos ejemplos:

```
public class ClaseGenerica {
    public <T> void metodo1(Caja<T> caja) {
        T contenido = caja.obtener();
        System.out.println(contenido.toString()); // Esto está bien
```

```
    }

    public void metodo2(Caja<?> caja) {
        Object contenido = caja.obtener();
        System.out.println(contenido.toString()); // Esto está bien
    }
}
```

Aquí podemos observar que ambos métodos, pueden aceptar una Caja que contenga cualquier tipo de objeto. La diferencia es que en metodo1(), puedes referirte al tipo de los objetos en la caja usando T, mientras que en metodo2(), debes referirte a los objetos en la caja como Object.

Vamos a hacer un ejemplo similar, pero usando extends. Imaginemos que ahora Caja<T> está limitada a clases que extienden de Number:

```
public class ClaseGenerica {
    public <T extends Number> void metodo1(Caja<T> caja) {
        T contenido = caja.obtener();
        System.out.println(contenido.doubleValue()); // Esto está bien
    }

    public void metodo2(Caja<? extends Number> caja) {
        Number contenido = caja.obtener();
        System.out.println(contenido.doubleValue()); // Esto está bien
    }
}
```

Como puedes ver, ambos métodos ahora sólo pueden aceptar una Caja que contenga una subclase de Number. Sin embargo, metodo1() puede referirse al tipo de los objetos en la caja usando T, mientras que metodo2() debe referirse a los objetos en la caja como Number.

Por lo tanto, en metodo1, podrías hacer cosas que son específicas para la subclase exacta de Number que T representa (si conoces cuál es esa subclase), mientras que en metodo2, sólo puedes hacer cosas que se pueden hacer con un Number.

Entonces la diferencia entre ambos está en que “T” se utiliza para declarar un parámetro de tipo que puedes usar en el resto de tu clase o método. Mientras

que “?” se utiliza como un comodín para decir "cualquier tipo" o "cualquier subtipo de" sin que puedas referirte a ese tipo en otro lugar. Y cuando usas “extends” con “T” vas a poder seguir refiriéndote a ese tipo como “T”, mientras que con **“?”**, tendrás que referirte al tipo del cual extiendes.

Inferencia de tipos

Es la capacidad del compilador de Java para determinar los tipos de parámetro basándose en el contexto. Por ejemplo:

```
public class Application {
    public static void main(String[] args) {
        Caja<Integer> caja = new Caja<>();
        caja.guardar(123);
    }
}
```

Aquí, el compilador deduce que estamos trabajando con un Caja<Integer> basado en el tipo de dato que guardamos en la caja.

Consideraciones importantes

- Los parámetros de tipo sólo existen en tiempo de compilación. En tiempo de ejecución, son eliminados por el compilador (esto se llama "type erasure" o "borrado de tipo").
- Como los parámetros de tipo no existen en tiempo de ejecución, no puedes, por ejemplo, hacer new T(), T.class o instanceof T.
- No puedes crear un array de tipo T, por ejemplo new T[10], debido a la misma limitación mencionada anteriormente.
- No puedes usar tipos primitivos como parámetros de tipo. En su lugar, debes usar sus equivalentes de wrapper, por ejemplo, en lugar de int, usarías Integer.