Generics

Generics

Generics es un mecanismo que permite a las clases Java (y también a otros lenguajes, el concepto excede a Java) manipular objetos en forma genérica, sin conocer de antemano (al momento de diseñar e implementar la clase que soporta Generics) cuál será la clase del objeto-elemento manipulado. Esto significa que el tipo de dato que se va a manipular no es explicitado cuando se diseña y define la clase, pero que sí será conocido cuando la clase se use.

¿Qué quiere decir esto? Posiblemente la mejor forma sea explicarlo a través de un ejemplo que ya conocemos: el uso de colecciones a partir de ArrayList.

Pero primero veamos la declaración técnica formal de la clase ArrayList según Oracle (https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html):

```
class ArrayList<E>;
```

En la declaración de la clase se puede observar la inclusión de `<E>`; esto no es más que una forma elegida por el lenguaje Java para recibir **parámetros de clase**. Esta notación suele conocerse como ***diamante*** o *paréntesis agudos*.

De antemano, **ArrayList** no sabe cuál es el tipo de elemento que se incluirá dentro de su colección de elementos. Pero para validar qué elementos puede recibir y devolver y asegurar así cierta seguridad de datos, ArrayList necesita que al momento de definir la colección concreta (la que vamos a usar) le indiquemos qué clase de elementos va a contener. Ese *"de qué clase se trata"* debemos decírselo pasándole un parámetro, el cual puede ser cualquier tipo de clase, incluso clases abstractas e interfaces.

Por ejemplo, si quisiéramos crear un ArrayList de series de tv con nuestras series favoritas, haríamos algo así:

```
ArrayList<SerieDeTV> misSeriesFavoritas = new ArrayList<>();
```

Así le decimos a ArrayList que el tipo de elementos que estará guardando serán de la clase **SerieDeTV**. Como se ve en el ejemplo, no es necesario aclarar al hacer el *new* cuál es el tipo de elemento, ya que se infiere directamente de la declaración de la variable.

Lo primero que debemos pensar es cuándo y por qué utilizar Generics. Hay algunos requisitos básicos e indispensables para que una clase pueda sacar provecho de este mecanismo. Si observamos detallada y críticamente las distintas declaraciones de las clases propias de Java que lo utilizan y sus métodos podremos ver que todas cumplen con los siguientes requisitos:

Generics

- La funcionalidad intrínseca de la clase no se ve afectada por el tipo de dato contenido (**ArrayList** guarda colecciones de elementos, **Stack** los apila, etc.).
- La clase contenedora (por ejemplo un ArrayList) no necesita conocer la clase del elemento contenido hasta el momento de ser utilizada, y hasta puede desconocerla por completo.
- La clase contenedora desconoce la estructura y el comportamiento (los atributos y métodos propios) de los elementos contenidos, y nunca manipula los datos (valores) de estos elementos. Solamente los recibe, los contiene y/o devuelve como unidad cuando es requerido y a través de los métodos declarados para ese fin.

Convenciones de nomenclatura

Por convención, para nombrar cada parámetro de Generics dentro de la clase se utiliza una única letra en mayúscula. Es completamente lo opuesto a la convención para la declaración de clases e interfaces y también para variables, atributos y parámetros de métodos. La razón primordial es facilitar la identificación de estos parámetros “de clase” de las propias clases e interfaces (no se declaran clases ni interfaces con nombres de una sola letra) y de atributos, variables y parámetros de métodos (que se declaran comenzando en minúscula).

Los nombres más utilizados (al menos en Java) son:

- **E** - Element (elemento). Usado extensivamente por Java Collections Framework.
- **K** - Key (clave). Se usa para la identificación de los nodos.
- **V** - Value (valor). Generalmente se usa en conjunto con K para indicar el tipo de dato del valor, más allá de la clave.
- **T** - Type (tipo). Usado principalmente cuando el componente es único (por ejemplo el contenido de una Caja).

Estos nombres (estas letras) se usarán dentro de la clase siempre que sea necesario referirse al tipo de dato del elemento, la clave, etc. Es decir que funcionan como comodines.

Manipulación de elementos

Veamos cómo es esto usando de ejemplo la clase ArrayList. ArrayList tiene varios métodos que nos permiten manipular los elementos de su colección. Cada operación que manipula algún elemento de la misma toma el tipo de dato del parámetro de clase indicado en **<E>**. Por ejemplo, para obtener el elemento de una posición determinada usamos *get()*, y su declaración es:

```
E get(int index)
```

Generics

En el caso de nuestras series favoritas de TV, donde tenemos una colección de SerieDeTV (y/o de sus clases derivadas, si las hubiese), el método *get()* devolverá un objeto que alguna de estas clases (SerieDeTV ó una de sus derivadas).

Por lo tanto, si quisiéramos recuperar la primera serie de nuestra lista de series favoritas, podríamos guardarla en una variable de la clase SerieDeTV escribiendo una línea de código como...

```
SerieDeTV miFavorita = misSeriesFavoritas.get(0);
```

De la misma forma, para recibir el elemento que se debe agregar a la colección, el método *add()* es:

```
public boolean add(E e)
```

… donde **E** es la clase del elemento y ***e*** (también de ***elemento***) el nombre del argumento recibido que representa al objeto a agregar en la lista.

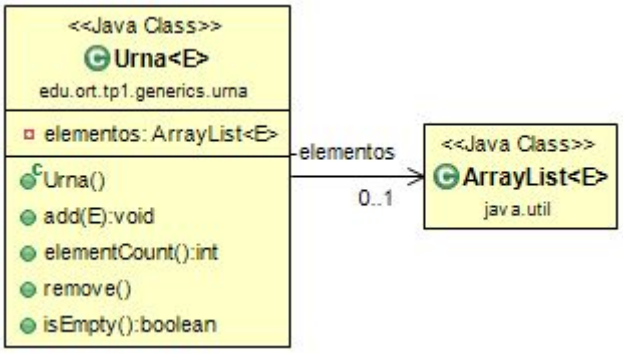
Declarar una clase que utiliza Generics

Para declarar que una clase utiliza uno o más parámetros de clase alcanza con incluir variables de tipo encerradas en el *diamante* (entre los paréntesis agudos “<” y “>”) de su declaración:

```
class NombreClase<variableDeClase>
```

Para usar de ejemplo vamos a crear una clase llamada **Urna**. Esta clase, que internamente tiene un ArrayList, sirve para guardar elementos que luego serán sacados de a uno y al azar. Este comportamiento se puede encontrar en el bingo, en un sorteo con papelitos o pelotitas (como se hace en los torneos deportivos), etc.

Esta clase no es muy grande, tiene apenas algunos métodos y su diseño es el siguiente...



Generics

… y la declaración de la clase es la que sigue:

```
class Urna<E>
```

Así, ***E*** recibirá el tipo de dato que se asigne en la declaración de la **Urna** y será trasladada al ArrayList contenido:

```
public class Urna<E> {

	private ArrayList<E> elementos;

	public Urna() {
		elementos = new ArrayList<>();
	}
```

Entonces, para crear una urna que guardará una serie de números para sacarlos al azar, declararemos:

```
Urna<Integer> numerosSorteo = new Urna<>();
```

Manipular y devolver elementos genéricos

Para manipular internamente los elementos del tipo referido por el parámetro de clase indicado en **<E>** (por ejemplo), la clase siempre usa esta referencia (**E**) para indicar el tipo de dato. Así, si hubiera que declarar un auxiliar interno (por ejemplo en los métodos ***remove()*** de cualquiera de estas estructuras, la variable auxiliar debe utilizar en su declaración el nombre del parámetro de clase que corresponda (en este caso **E**).

Mostramos aquí ejemplo una versión extendida del método *remove()* de **Urna**, que elige al azar una posición cualquiera de la colección y extrae ese elemento. Puede verse tanto en la declaración del método como en la de la variable local cómo se usa el comodín E, que al momento de usar la urna se reemplazará por el tipo de dato real (en el caso que nombramos antes sería *Integer*.

```
	public E remove() {
		E elemento = null;
		if (!isEmpty()) {
			int pos = (int) (Math.random() * elementos.size());
			elemento = elementos.remove(pos);
		}
```

Generics

```
            return elemento;
      }
```

Generics y tipos de datos indefinidos

Cuando trabajamos con Generics no siempre sabemos qué tipo de elementos estamos manejando, y en determinados momentos probablemente no nos importe demasiado. Supongamos que tenemos que mostrar todos los elementos guardados en un ArrayList. ¿Podemos hacer un método genérico que funcione con cualquier ArrayList sin saber de antemano cuál será la clase de sus elementos?

Se puede. La razón principal es que independientemente de cuál sea el objeto contenido, el mismo es un objeto. De hecho, antes de existir el concepto de Generics, en las primeras versiones de Java sólo se creaban colecciones de **Object** que luego indefectiblemente debían *enmascararse (casting)* para acceder al dato.

Pero siempre hay algún "pero". El primero es que no sabremos cómo definir dentro del diamante el tipo de elemento *genérico* que contendrá el ArrayList recibido como parámetro, pues desconocemos en tiempo de diseño cuál será este Arraylist.

La forma de hacerlo es utilizar el operador **?**, el cual indica que desconocemos el tipo del parámetro formal y que éste debe resolverse en tiempo de ejecución.

Claro que al hacer esto, dentro del método no podremos acceder a los métodos propios del elemento manipulado. En caso de recorrer la colección con un ***for-each***, lo que se hará es mapear momentáneamente los elementos con la clase **Object**. Pero si no queremos hacer esto, si recorremos la estructura a partir de la posición (por ejemplo con un ***for***) y obtenemos el elemento con ***get()***, este mapeo es innecesario.

El ejemplo a continuación ilustra el caso y muestra las dos posibilidades. Ambos métodos obtienen exactamente el mismo resultado.

```
package edu.ort.generics.ejemplos;

import java.util.ArrayList;

public class ListadorDeArrayLists {

      public static void main(String[] args) {
            ArrayList<Number> numeros = new ArrayList<>();
            numeros.add(1);
            numeros.add(-32);
            numeros.add(14.5);
            ArrayList<String> palabras = new ArrayList<>();
            palabras.add("hola");
            palabras.add("que");
            palabras.add("tal");
```

Generics

```
		// invocamos al método listarElementosConForEach para mostrar
		// el contenido de cada colección.
		listarElementosConForEach(numeros);
		listarElementosConForEach(palabras);

		// invocamos al método listarElementosSinUsarObject para mostrar
		// el contenido de cada colección.
		listarElementosSinUsarObject(numeros);
		listarElementosSinUsarObject(palabras);
	}

	private static void listarElementosConForEach(ArrayList<?> lista) {
		// como el método requiere una variable local para guardar
		// el elemento debemos usar Object para referirnos a él.
		for (Object objeto : lista) {
			System.out.println(objeto);
		}
	}

	private static void listarElementosSinUsarObject(ArrayList<?> lista) {
		for (int i = 0; i < lista.size(); i++) {
			System.out.println(lista.get(i));
		}
	}

}
```

El otro “pero” es que, si tuviésemos que devolver un elemento, el tipo declarado no puede ser otro que **Object**.

Interfaces con Generics

Las interfaces también pueden usar Generics y, de hecho, conocemos y utilizamos varias de ellas. La declaración de una interfaz que haga uso de Generics es exactamente la misma que la de una clase con Generics. Por ejemplo, la declaración de la interfaz ***GuardaCosas*** para una hipotética clase que permita guardar cosas es la siguiente:

```
public interface GuardaCosas<E> {
	void guardar(E elemento);
	E extraer();
	boolean isEmpty();
	boolean isFull();
}
```

Generics

En caso de que la clase que implementa la interfaz espere recibir el parámetro de tipo, ésta hará de “pasamanos” y reenviará este tipo a la interfaz. Por ejemplo, la declaración de la clase **Armario** que implementa **GuardaCosas** es:

```
public class Armario<E> implements GuardaCosas<E> {
	...
	...
}
```

Una aclaración importante, antes de continuar y a modo de recordatorio, es que si bien todas estas estructuras de las que venimos nombrando sirven para contener colecciones o conjuntos de elementos, el mecanismo de Generics no está intrínsecamente relacionado a la manipulación de colecciones o conjuntos. Podríamos tener una **Caja** que sirva para guardar una sola cosa, claro ejemplo de agregación en el que podemos usar Generics y en el que no hay ninguna colección, sino un único elemento/objeto asociado.

Estructuras con Generics anidados

Sabemos que cualquier clase puede ser atributo de otra clase e ir a parar “adentro del *diamante*”. Eso incluye a las clases e interfaces cuya declaración incluye Generics. ¿Cómo hacemos, entonces, para declarar una lista de pilas, o cualquier combinación que se nos ocurra donde todas las clases implicadas tienen su propio *diamante*?

Parece complicado cuando lo leemos por primera vez, pero si lo leemos con cuidado veremos que simplemente las definiciones se *anidan* una dentro de la otra.

Veamos esto paso a paso para evitar confusiones. Observemos primero la declaración de ***Caja*** y sus métodos públicos (tranquilamente Caja podía ser una interfaz).

```
public class Caja<T> {
	T contenido;

	public Caja() {
		contenido = null;
	}

	public boolean estaVacia() {
		return contenido == null;
	}

	public boolean estaLlena() {
		return contenido != null;
	}

	void guardar(T contenido) {
```

Generics

```
            if (contenido == null) {
                  throw new RuntimeException("Debe agregarse un contenido");
            }
            if (estaLlena()) {
                  throw new RuntimeException("La caja ya esta llena");
            }
            this.contenido = contenido;
      }

      T extraer() {
            if (estaVacia()) {
                  throw new RuntimeException("La caja esta vacia");
            }
            return contenido;
      }
};
```

Así, para crear una caja de zapatos vacía no tendríamos más que hacer:

```
Caja<ParDeZapatos> cajaDeZapatos = new Caja<>();
```

Pero si quisiéramos guardar varias cajas de zapatos en un arraylist, ¿cuál sería la declaración correcta?

La declaración e instanciación correcta de la colección sería...

```
ArrayList<Caja<ParDeZapatos>> miColeccionDeZapatos = new ArrayList<>();
```

Así, para agregar una caja con un par de zapatos negros en la colección hacemos…

```
      ParDeZapatos zapatos = new ParDeZapatos("Negros");
      Caja<ParDeZapatos> cajaDeZapatos = new Caja<>();
      cajaDeZapatos.guardar(zapatos);
      ArrayList<Caja<ParDeZapatos>> miColeccionDeZapatos = new ArrayList<>();
      miColeccionDeZapatos.add(cajaDeZapatos);
```

Como se puede apreciar en el ejemplo, cada vez que se crea un objeto de una clase que usa Generics se puede emplear el *diamante* vacío (<>), pues el tipo del dato se infiere de la declaración de la variable.

Generics

Arrays de datos que utilizan Generics

Arrays y Generics no se llevan bien en Java, al menos al momento de declarar e inicializar el array. Esto se debe a lo que cada estructura necesita resolver en tiempo de compilación y cómo maneja la memoria para sí misma y para su contenido. Podemos resumirlo en que mientras que a un ArrayList (o cualquier estructura que use Generics) no le importa el tipo del dato contenido, a un Array sí le importa.

Sabemos el tipo de dato de un objeto porque lo hemos proporcionado concreta y explícitamente cuando el objeto se creó. Lo mismo se aplica para los elementos de un arreglo: cuando se crea un Array (no cuando se declara, sino cuando lo instanciamos) debemos especificar el tipo de elemento de manera explícita. Sin embargo, los tipos genéricos en el código son una "ilusión" en tiempo de compilación, "comodines" que serán reemplazados por la clase real, y ahí está la "magia". Cuando tenemos una variable de tipo como ***T*** (suponiendo que declaramos *Clase*<***T***>), el código usa este tipo como "comodín". No puede saber realmente a qué tipo se refiere realmente ***T***, y sin embargo, sea cual fuere la clase que ocupe el lugar de ***T***, el código funcionará sin problemas.

Pero mientras una clase que usa generics no necesita conocer realmente qué elemento tiene, un arreglo necesita conocer explícitamente el tipo de dato, no puede "dejarlo para después". Y aquí chocan ambos mecanismos.

La solución más simple es crear una clase intermedia que "esconda" la definición con el diamante. Siguiendo el ejemplo anterior, si quisiéramos hacer un Array de cajas de zapatos, podríamos hacer:

```
public class CajaDeZapatos extends Caja<ParDeZapatos> { }
```

No hace falta más que la definición de la nueva clase, pues mantiene toda la funcionalidad original de una **Caja** que sabe guardar instancias de **ParDeZapatos**. Así, al quedar escondida la definición genérica inicial, se puede crear e inicializar un Array de la nueva clase sin problemas porque la nueva clase ya es de un **tipo concreto**:

```
CajaDeZapatos [] zapatero = new CajaDeZapatos[CANTIDAD_DE_CAJAS];
```

Dado que la creación de la clase **CajaDeZapatos** apenas es algo más que una declaración que "fija" el tipo de dato genérico y lo reemplaza por **ParDeZapatos**, todos los métodos públicos de la clase genérica quedan disponibles como si directamente usáramos **Caja**, aunque aquellos métodos de **Caja** que devolvían un elemento genérico (***T*** en la declaración) ahora solamente trabajarán con un objeto de clase **ParDeZapatos**.

Las líneas que siguen hace algo similar a lo que hacíamos en el caso anterior, pero ahora trabajando con el array recién declarado:

Generics

```
ParDeZapatos zapatos = new ParDeZapatos("Negros");
CajaDeZapatos [] zapatero = new CajaDeZapatos[CANTIDAD_DE_CAJAS];
CajaDeZapatos caja = new CajaDeZapatos();
caja.guardar(zapatos);
zapatero[0] = caja;
```

Generics y Herencia

Si vemos en detalle lo que hicimos en el punto anterior veremos que no es más que la declaración de una clase derivada de **Caja<T>** que sabe guardar pares de zapatos.

El mecanismo de Generics puede combinarse, claro está, para la creación de nuevas clases que definan el valor del parámetro de clase (eliminando el diamante en su propia declaración, como pasa con la caja de zapatos, o bien, si no necesita conocer el tipo de dato contenido. puede continuar teniendo su propio diamante y pasarle el parámetro del clase a su clase base o interfaz (lo mismo que hace la clase **Armario** respecto a **GuardaCosas)**.

Conclusión

El mecanismo de Generics es muy útil, y aunque no es más que una forma de pasar clases de objetos como parámetros, es interesante conocerlo pues permite que clases “contenedoras” puedan guardar elementos de clases no definidas y/o desconocidas por ellas de antemano.

Conocer este mecanismo es importante no sólo para no depender exclusivamente de la herencia, sino porque está bastante presente, entre otras, en las implementaciones de las estructuras múltiples basadas en colecciones.