Instructivo de Generics

El siguiente instructivo es una traducción de varias secciones de la lección de Oracle:
https://docs.oracle.com/javase/tutorial/java/generics/index.html

¿Por qué usar Generics?

Los Generics (genéricos) permiten que los *tipos* (clases e interfaces) sean parámetros cuando se definen clases, interfaces y métodos. Como los parámetros formales, más familiares, usados en la declaración de métodos, los parámetros de tipos proveen una forma de reutilizar el mismo código con diferentes entradas. La diferencia es que las entradas de los parámetros formales son valores, mientras que las entradas de los parámetros de tipos son tipos.

El código que utiliza Generics tiene muchos beneficios sobre el código que no los utiliza:

- Chequeos más fuertes de tipos en tiempo de compilación.
  El compilador de java aplica el chequeo fuerte de tipos al código genérico y muestra los errores si el código viola el chequeo. Arreglar los errores en tiempo de compilación es más fácil que arreglar errores de tiempo de ejecución, que pueden ser difíciles de encontrar.
- Se elimina la utilización de *casts*
  El siguiente snippet sin Generics requiere de casting:

  ```
  List list = new ArrayList();
  list.add("hello");
  String s = (String) list.get(0);
  ```

  Cuando se lo reescribe utilizando Generics, el código no requiere de casting:

  ```
  List<String> list = new ArrayList<String>();
  list.add("hello");
  String s = list.get(0); // No hay cast
  ```
- Permite a los desarrolladores implementar algoritmos genéricos:
  Mediante el uso de los Generics, los desarrolladores pueden implementar algoritmos genéricos que funcionan con colecciones de diferentes tipos, pueden ser adaptados, son seguros en cuanto al chequeo de tipos y son fáciles de leer.

Tipos genéricos

Un tipo genérico es una clase o interfaz genérica que es parametrizada mediante tipos. La siguiente clase *Box* será modificada para demostrar el concepto.

Una simple clase Box

Empiece por examinar una clase *Box* no genérica que opera con objetos de cualquier tipo. Sólo necesita de dos métodos: set, que agrega un objeto a la caja, y get, que lo obtiene:

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Como sus métodos aceptan o devuelven un Object, eres libre de pasarle cualquier cosa que quieras, siempre y cuando no sea uno de los tipos primitivos (en Java todas las clases heredan de Object). No hay forma de verificar, en tiempo de compilación, cómo se utiliza la clase. Una parte del código podría poner un Integer in la caja y esperar obtener Integers de la misma, mientras otra parte del código podría, por error, pasarle un String, resultando en un error en tiempo de ejecución.

Una versión genérica de la clase Box

Una clase genérica es definida mediante el siguiente formato:

```
class Name<T1, T2, ..., Tn> { /* ... */ }
```

La sección de parámetros de tipo, delimitada por los brackets (<>), se encuentra a continuación del nombre de la clase. Especifica los parámetros de tipo (también llamados variables de tipo) T1, T2 … Tn.

Para actualizar la clase Box y utilizar Generics, creas una declaración de un tipo genérico, cambiando el código “public class Box” a “public class Box<T>”. Esto introduce la variable de tipo, T, que puede ser utilizada en cualquier lugar dentro de la clase.

Con este cambio, la clase Box cambia a:

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Como puedes ver, todas las ocurrencias de Object son reemplazadas con T. Una variable de tipo puede ser cualquier tipo **no primitivo** que especifiques: cualquier tipo de clase, cualquier tipo de interfaz, array, o inclusive otra variable de tipo.

Esta misma técnica se puede aplicar a las interfaces genéricas.

Convención de nombres para parámetros de tipo

Por convención, los parámetros de tipo se nombran con una única letra mayúscula. Esto contrasta mucho con las convenciones de nombres de variables que ya conocen de Java, y por una buena razón: Sin esta convención, sería difícil distinguir mediante una variable de tipo y el nombre de una clase o interfaz.

Los nombres más comunes usados para parámetros de tipo son:

- E – Elemento (usado extensivamente en Java Collections)
- K – Key (clave)
- N – Number (número)
- T – Tipo
- V – Valor
- S, U, V etc – 2do, 3er y 4to tipo

Verás estos nombres a través de toda la API de Java SE.

Invocando e instanciando un Tipo Genérico

Para hacer referencia a la clase Box desde dentro de tu código, deberás realizar una invocación de tipo genérico, que reemplaza T con algún valor concreto, como Integer.

```
Box<Integer> integerBox;
```

Podés pensar en la invocación de un tipo genérico como algo similar a la invocación a un método cualquiera, pero, en vez de pasarle un argumento al método, le estarás pasando un argumento de tipo – Integer en este caso – a clase Box misma.

Terminología: Parámetro de Tipo y Argumento de Tipo: muchos desarrolladores usan los términos parámetro de tipo y argumento de tipo como si fueran sinónimos, pero estos términos no son iguales. Cuando uno programa, provee argumentos de tipo para crear un tipo parametrizado o parametrizable. Entonces, la T en Foo<T> es un parámetro de tipo y String en Foo<String> es un argumento de tipo. En este instructivo se respeta esta definición cuando se utilizan estos términos.

Como cualquier otra declaración de variable, este código no crea nuevo objeto Box. Simplemente declara que integerBox va a mantener una referencia a “Una Caja de Integer”, que es como se leería Box<Integer>.

La invocación a un tipo genérico se conoce comúnmente como parameterized type (tipo parametrizado).

Para instanciar esta clase, utiliza la palabra reservada “new”, como siempre, pero pon <Integer> en medio del nombre de la clase y los paréntesis:

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

El Diamante

A partir de Java SE 7 y posterior, se puede reemplazar los argumentos de tipo requeridos para invocar el constructor de una clase genérica con un conjunto vacío de argumentos de tipo (<>), siempre y cuando el compilador pueda determinar, o inferir, los argumentos de tipo a través del contexto. Este par de brackets, <>, se conocen informalmente como “el diamante”. Por ejemplo, se puede crear una instancia de Box<Integer> con la siguiente sentencia:

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

Múltiples parámetros de tipo

Como se mencionó anteriormente, una clase genérica puede tener múltiples parámetros de tipo. Por ejemplo, la clase genérica OrderedPair, que implementa la interfaz genérica Pair:

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

Las siguientes sentencias crean dos instancias de la clase OrderedPair:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even",
8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello",
"world");
```

El código, new OrderedPair<String, Integer>, instancia K como un String y V como un Integer. De esta forma, los parámetros de tipos de OrderedPair son String e Integer, respectivamente.

Gracias al autoboxing (el mecanismo por el cual un tipo primitivo es “convertido” en un objeto que lo envuelve), es válido pasarle un String y un int (en vez de un Integer directamente) a la clase.

Como se mencionó en la sección del Diamante, como el compilador de Java puede inferir los tipos de K y V a partir de la declaración OrderedPair<String, Integer>, estas sentencias se pueden acortar utilizando la notación de diamante:

```
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
```

Para crear una interfaz genérica, sigue las mismas convenciones que para crear una clase genérica.

Tipos parametrizados

También se puede sustituir un parámetro de tipo (ej K o V) con un tipo parametrizado (por ejemplo List<String>). Por ejemplo, usando OrderedPair<K,V>:

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new
Box<Integer>(...));
```

Métodos Genéricos

Los métodos genéricos son métodos que introducen sus propios parámetros de tipo. Esto es similar a declarar un tipo genérico, pero el alcance del parámetro de tipo estará limitado al método en el que se declare. Se permite tanto para métodos estáticos como no estáticos, así como también para constructores de clases genéricas.

La sintaxis para un método genérico introduce un parámetro de tipo, dentro de los brackets, y aparece antes del tipo de retorno del método. Para métodos estáticos, la sección de parámetro de tipo debe aparecer antes del tipo de retorno.

La clase Util, incluye un método genérico, compare, que compara dos objetos Pair:

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2)
{
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}

public class Pair<K, V> {

    private K key;
    private V value;

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

    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

La sintaxis completa para invocar este método sería:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

El tipo ha sido provisto de forma explícita, como se muestra en negrita. Generalmente esto se puede dejar para que el compilador infiera el tipo que se necesita:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

Parámetros de Tipo "Ligados/Limitados"

Hay veces en que podés querer restringir los tipos que se pueden usar como argumentos de tipo en un tipo parametrizado. Por ejemplo, un método que opera con números puede querer aceptar sólo instancias de Number o sus subclases. Para esto que son los parámetros de tipo ligados (Bounded Type Parameters en inglés).

Para declarar un parámetro de tipo ligado, debes listar el nombre del parámetro, seguido de la palabra reservada extends (extiende/hereda), seguido de su límite superior, que en este ejemplo es Number. Nótese que en este contexto, extends, es usado en el sentido general para expresar que extends (hereda/extiende como en clases) o implements (implementa interfaz).

```
public class Box<T> {

    private T t;

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

    public T get() {
        return t;
    }

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // error: esto sigue siendo
un String!
    }
}
```

Al modificar nuestro método para incluir un parámetro de tipo ligado, la compilación ahora fallará, ya que nuestra invocación de inspect todavía incluye un String:

```
Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot
  be applied to (java.lang.String)
                        integerBox.inspect("10");
                                   ^
1 error
```

Además de limitar los tipos que se pueden usar para instanciar un tipo genérico, los parámetros ligados permiten invocar métodos definidos dentro de los límites:

```
public class NaturalNumber<T extends Integer> {

    private T n;

    public NaturalNumber(T n)  { this.n = n; }

    public boolean isEven() {
        return n.intValue() % 2 == 0;
    }

    // ...
}
```

isEven invoca el método intValue definido en Integer a través de n.

Múltiple

El ejemplo anterior muestra el uso de un parámetro de tipo con un único límite/ligamiento, pero un parámetro de tipo puede tener múltiples:

```
<T extends B1 & B2 & B3>
```

Una variable con multiples ligamientos es un subtipo de todas los tipos listados en el ligamiento. Si uno de ellos es una clase, debe ser especificado primero:

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

Si A no es especificado antes que los demás, se obtendrá un error en tiempo de compilación:

```
class D <T extends B & A & C> { /* ... */ }  // error
```

Métodos genéricos y parámetros ligados

Los parámetros de tipo ligados son clave para la implementación de algoritmos genéricos. Considere el siguiente método que cuenta la cantidad de elementos en un array T[] que son más grandes que un elemento “elem”.

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // compiler error
            ++count;
    return count;
}
```

La implementación del método es bastante simple, pero no compila porque el operador (>) sólo opera con tipos primitivos como short, int, doublé, long, float, byte y char. No se puede usar el operador (>) para comparar objetos. Para arreglar el problema, unamos un parámetro de tipo ligado por la interfaz Comparable<T>:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

El código resultante será:

```
public static <T extends Comparable<T>> int countGreaterThan(T[]
anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```