Tipos de datos paramétricos

El uso de tipos de datos paramétricos no es exclusivo de Java. Incluso es una incorporación reciente al lenguaje, mientras que otros lenguajes (C++, D, Eiffel, Delphi) ofrecen esta herramienta hace muchos años.

Algunos lenguajes llaman a esta herramienta "Genericity" dado que se establece un tipo de dato genérico para la estructura de datos o clase que se está definiendo. En Java, se lo denomina de manera similar: **"Generics"** consiste en diferir el verdadero tipo de un objeto hasta que se use. Entonces, el "tipo" del objeto se deja como un parámetro más, como si fuera un parámetro de un método, que el programador establecerá al momento de trabajar con el código correspondiente a ese objeto.

Definición y uso

Para ver cómo definir un tipo de dato paramétrico y cómo se usa, pensemos en un objeto sencillo: un Balde. En un balde, se pueden colocar muchas cosas: podemos llenarlo de algo sólido, como tierra, arena, escombros, o de algo líquido, como agua, combustible, etc. De esta forma, si llenamos el balde con una cosa particular, al vaciarlo solo podremos obtener lo mismo.

Por ejemplo, si llenamos un balde con agua, cuando volquemos el contenido, volcaremos agua. De la misma forma, si estamos lidiando con un balde de agua, no deberemos llenarlo con combustible, tampoco queremos poner un poco de agua y un poco de combustible. No queremos lavar el piso con un líquido potencialmente inflamable. Entonces, si bien un balde puede contener cualquier cosa, en un punto determinado debemos establecer reglas específicas sobre el contenido del balde. Las operaciones que podemos hacer con el balde estarán dirigidas por estas reglas también.

Veamos cómo poner esto en código:

```
public class Balde<T> {
  private T contenido;
    public Balde() {
    }
```

```
    public void llenar(T contenido) {
      this.contenido = contenido;
    }
    public T obtenerContenido() {
      return contenido;
    }
}
```

Lo que definimos no es otra cosa que una clase, tal cual la conocemos. La diferencia es el agregado de "<T>". Esto significa que el Balde tendrá un tipo específico en su interior. De igual manera, si observamos el atributo "contenido" no establecimos el tipo de dato. O, mejor dicho, sí lo hicimos, pero el tipo es "genérico". En este caso, "diferimos" el tipo hasta el momento de usar el Balde. Es decir, dejamos el tipo como **parámetro**. Ese parámetro está definido por la letra T. La letra puede ser cualquier letra del abecedario, menos Ñ, por supuesto. Normalmente se usa T por "type". Lo mismo ocurre con las operaciones sobre el contenido; como dijimos, estas están reguladas por el tipo genérico. Entonces, limitamos el uso de estas operaciones de acuerdo con el tipo del atributo contenido.

Hagamos un ejemplo:

```
public class Agua{
  //atributos
  …
  // getters & setters
}
```

```
public class Agua{
  //atributos
  …
  // getters & setters
  public void encender() {
    //boom
  }
```

```
}
```

```
public class Test {
  public static void main(String[] args) {
    Agua a = new Agua();
    Combustible c = new Combustible();

    Balde<Agua> b = new Balde<>();
    b.llenar(a);

    // NO COMPILA -> No puedo poner combustible en un balde de
agua!
    //b.setContenido(c); //no puedo "mezclar" contenidos

    // si el balde es de agua, siempre obtendré agua de él
    System.out.println("Voy a tomar" + b.obtenerContenido());

    Balde<Combustible> danger = new Balde<Combustible>();
    danger.setContenido(c);
    danger.obtenerContenido().encender();
  }
}
```

Utilidad de Generics

¿Cómo sería posible obtener el ejemplo anterior si no existiera Generics? Según lo que vimos, todas las clases en Java heredan de Object, con lo cual podríamos haber hecho lo siguiente:

```
public class Balde {
  private Object contenido;
    public Balde() {
```

```
    }
    public void llenar(Object contenido) {
      this.contenido = contenido;
    }
    public Object obtenerContenido() {
      return contenido;
    }
}
```

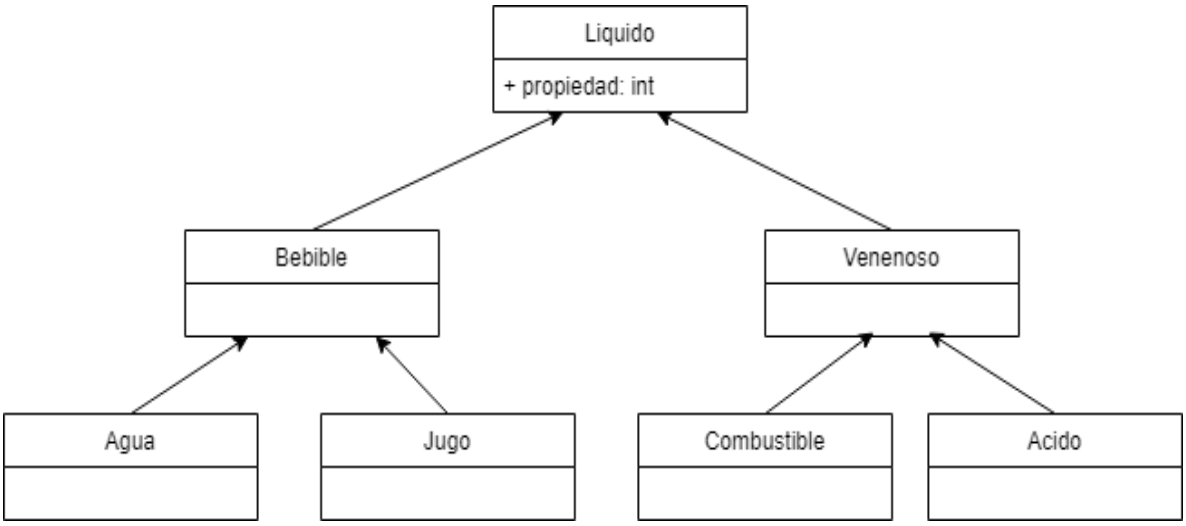
Y no habría problemas, podríamos hacer (casi) lo mismo que hicimos en el main anterior. Sin embargo, si observamos detenidamente, para hacer lo que hicimos en el main anterior esto traería problemas:

```
Balde danger = new Balde<Combustible>();
danger.setContenido(c);
danger.obtenerContenido().encender();
```

El código no compila porque obtenerContenido() devuelve un Object, y el método encender() no existe allí. Deberíamos **castear** el contenido a Combustible y solo así tendríamos acceso al método encender(). Hay un problema adicional. El Balde, así definido, acepta "mezclas". Puedo en el mismo balde poner combustible y luego agua. Eso indicaría que no solo debemos castear, sino también preguntar con el instanceof si podemos hacerlo. Es decir, esta solución con "Object" puede traernos problemas en tiempo de ejecución si no somos cuidadosos.

Volviendo al Balde<T> genérico, podemos observar que no hubo casteos para operar con el contenido; por tanto, no hay peligros de errores al momento de la ejecución. Adicionalmente, no podemos poner algo que no corresponda dentro del balde.

Una ventaja adicional es que los tipos genéricos aceptan jerarquías. El hecho de especificar un tipo de dato al momento de usar el balde es relativamente flexible, el tipo no queda "fijo". Supongamos que tenemos las clases Liquido, del cual tenemos dos hijas, Bebible y Venenoso, que a su vez tienen hijas: Agua y Jugo por un lado, y Combustible y Acido por otro:



Según lo que vimos, si quisiera poder tomar del contenido del balde, debería tener dos baldes, uno para jugo y otro para agua.

```
Agua a = new Agua();
Jugo j = new Jugo();
Balde<Agua> b = new Balde<>();
b.setContenido(a);
//b.setContenido(c); error! no puedo poner jugo en un balde de agua!

// si el balde es de agua, siempre obtendré agua de él
System.out.println("Voy a tomar" + b.obtenerContenido());
```

Entonces, a simple vista, establecer el tipo de antemano para el balde parece ser un poco rígido. Pero los tipos paramétricos nos dan herramientas como para "dar a entender" qué queremos poner en el balde. Por ejemplo, queremos poner cualquier cosa que podamos beber:

```
public class Test {
  public static void main(String[] args) {
    Agua a = new Agua();
    Jugo c = new Jugo();
    Balde<Bebible> b = new Balde<> ();
```

```
    //agua es un Bebible, pero esto no compila
    b.setContenido(a);
    //jugo es un Bebible, pero esto no compila
    b.setContenido(c);
    //lo único que podemos hacer es
    Bebible bebida = new Bebible();
    b.setContenido(bebida);
    System.out.println("Voy a tomar" + b.obtenerContenido());
```

```
    Agua a = new Agua();
    Jugo c = new Jugo();
    Balde<? extends Bebible> b = new Balde<> ();
    //agua es un Bebible, esto es valido
    b.setContenido(a);
    //jugo es un Bebible, esto es valido
    b.setContenido(c);
    //lo único que podemos hacer es
    Bebible bebida = new Bebible();
    // ya no es valido, el tipo es cualquier "hija de Bebible"
    //b.setContenido(bebida);
    System.out.println("Voy a tomar" + b.obtenerContenido());
  }
}
```

Lo que hicimos aquí es definir el tipo genérico con "?". Esto se denomina "comodín" o "wildcard". En este caso, estamos diciendo que el tipo será "'cualquier cosa que' extienda de Bebible" o "cualquier cosa que cumpla con la prueba es-un Bebible". ¡Pero atención!, porque se está aclarando que el tipo será cualquier cosa que extienda de Bebible, pero no una instancia de Bebible propiamente dicha. La regla es clara: subclases de bebible. Otra forma de establecer esta regla sería en la clase Balde:

```
public class Balde<T extends Bebible> {
  private T contenido;
    public Balde() {
    }
    public void llenar(T contenido) {
```

```
        this.contenido = contenido;
    }
    public T obtenerContenido() {
        return contenido;
    }
}
```

Aunque limitamos un poco el uso del Balde. El Balde ahora podrá contener cualquier líquido bebible, pero no combustible ni arena… Como siempre, donde pongamos la regla y qué restricciones queramos imponer, dependerá del problema a resolver.

La utilidad final es que todos los chequeos de tipos son en tiempo de compilación. Cuando el programa se ejecuta todos estos chequeos ya no se hacen, porque ya no son necesarios. Cualquier error de parte del programador hubiera saltado al momento de escribir el código. De hecho, si pudiéramos ver el bytecode ya ni siquiera quedan rastros de la sintaxis “<” y “>”. Esto se conoce como “type erasure”, que significa que los tipos paramétricos desaparecen (“se borran”) al momento de ejecutar el código.

Te invitamos a reflexionar sobre lo visto durante la lectura: ¿Consideras que Generics es útil para la OOP? ¿Crees que es necesario para lenguajes fuertemente tipados, como Java, o crees que sería aplicable a cualquier lenguaje? ¡Cuéntanos!