Introducción a los genéricos

En este tema voy a describir un mecanismo para definir clases e interfaces parametrizando su definición de acuerdo con un tipo genérico. **El tipo concreto se proporcionará cuando se declaren elementos de dicha clase o interfaz**. Por tanto, los genéricos son una abstracción en la definición de clases e interfaces.

Esta interfaz permite especificar una colección de objetos de un cierto tipo, pero sin especificarlo en la definición, sino en el uso. En efecto, si se imagina una colección de String y otra de Alumno, es fácil imaginar que el código es igual, pero haciendo referencia a String y a Alumno, según el caso.

El siguiente ejemplo de uso de la clase *ArrayList* en la que se especifica que se van a guardar objetos de la clase Alumno:

```
ArrayList<Alumno> miGrupo = new ArrayList<Alumno>(50);

    miGrupo.add(new Alumno("Pepito","Garcia",1998));
    miGrupo.add(new Alumno("Jakintsu","Iheslari",1980));

    Iterator<Alumno> a = miGrupo.iterator();
    while(a.hasNext()){
       Alumno al = a.next();
       try {
          al.ponGrupo("33345", Horario.TARDE);
       } catch (Exception e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
       }
    }
```

En las declaraciones, se pone el nombre de la clase genérica, y entre los signos de menor que y de mayor que, el nombre de la clase con la que se parametriza. El resultado es que no se dispone de un objeto *ArrayList* cualquiera, sino un *ArrayList* de alumnos, que se escribe como *ArrayList<Alumno>.* Es decir, *ArrayList* es una clase genérica que toma como un parámetro un tipo, en el ejemplo Alumno.

Definición de genéricos

Para ver como declarar genéricos, se presenta a continuación la definición del interfaz *Iterator* del paquete *java.util*:

```
public interface Iterator {
   boolean hasNext();
```

```
   E next();

   void remove();

}
```

La definición es muy similar a la de cualquier interfaz, excepto por, que quiere decir que este interfaz toma como parámetro un tipo que se denominará *E*. **A este tipo, se le llama tipo formal parámetro del genérico**. Se puede usar como si existiese ya declarado, con alguna excepción. **Por ejemplo, no puede aparecer en la declaración ni inicialización de un elemento de clase.**

Herencia de genéricos y conversión de tipos

Los genéricos se pueden heredar como cualquier otra clase o interfaz. Las reglas de extensión son las mismas que se aplican en la herencia. No obstante, las reglas de compatibilidad de tipos requieren una explicación detallada. En el siguiente código, se asignan referencias de genéricos:

```
List<Alumno> ls = new Vector<Alumno>();
List<Persona> lp = ls; //problemas en esta línea de código
```

La primera línea asigna a una lista de alumnos una instancia de vectores de alumnos. Efectivamente, Vector<E> extiende List<E>, siendo tipos compatibles. El problema aparece en la siguiente línea: el tipo List<Persona> no es un tipo compatible con List<Alumno>. Para entenderlo, piensa que en el objeto lp, de tipo List<Persona>, se pueden insertar objetos de la clase Persona, que no son compatibles con los objetos de ls, que al ser List<Alumno>, sólo admite insertar objetos de la clase Alumno.

Comodines

Dado que las instanciaciones de genéricos sólo son compatibles si el tipo parametrizado es el mismo, no habría forma de escribir código que sea válido para un tipo genérico si no se sabe cómo se instanciará. El siguiente es un ejemplo que intenta imprimir los datos de las personas de una lista:

```
void imprime(List<Persona> c){
      for(Persona p:c){
         System.out.println(p);
      }
   }
```

Sin embargo, el método anterior no es muy útil pues no se puede utilizar para imprimir una lista de alumnos, de profesores, de bedeles, dado que *List<Persona>* no es una clase compatible con *List<Alumno>, List<Profesor>*, etc. **No obstante, se incluye un**

mecanismo denominado de "comodines" que permite generalizar el uso de genéricos. El siguiente código permite imprimir los objetos de una lista:

```
void imprimeTodo(List<?> c){
      for(Object o:c){
         System.out.println(o);
      }
   }
```

El comodín se representa con el carácter de interrogación '?'. **De esta forma, *List<?>* representa una lista instanciada por cualquier tipo.** Sin embargo, si se observa el bucle *for*, los objetos de la lista sólo se pueden usar como referencias a *Object*.

Los comodines pueden ser ligados, especificando que los tipos sean de una clase o interfaz cualquiera clases derivadas. Esto se escribe como:

- List<? extends nombreClase>
- List<? extends nombreInterfaz>

En el siguiente ejemplo, se presenta un método que imprime los apellidos y año de nacimiento de una lista de personas.

```
void imprimePersonas(List<? extends Persona> p) {

      for (Persona per : p) {
         System.out.println(per.getApellidos() + " " + per.getAnioNacimiento());
      }
   }
```

Métodos genéricos

Para acabar con la teoría de genéricos, sólo nos falta ver cómo declarar métodos que usen la clase o el interfaz genérico. Es posible, especificando que un método va a usar un tipo genérico, como presenta el siguiente ejemplo:

```
public static <T> void insertar(List<T> lp, T o) {
      lp.add(o);
   }
```

Se define un método genérico, insertar. Se indica que es genérico, añadiendo la especificación de parámetros de tipo (uno o más), antes del valor de retorno del método. En este caso, <T>. El método acepta una lista genérica, *List<T>* y un objeto de tipo T. De hecho, se pueda usar el tipo T en el cuerpo del método, como muestra el siguiente ejemplo, que inserta un *array* de alumnos en una lista:

```
public static<T> void inserta(List<T> lp, T[] arr ){
      for(T o:arr){
         lp.add(o);
      }
   }
```

Así, como con *? extends T* se pone un límite superior en la jerarquía de tipos aceptada por un método genérico, o en una declaración de clase o interfaz genérico, también es necesario poner un límite inferior en los tipos aceptados. Para ello, se utiliza la sintaxis:

```
TreeSet(Comparator<? super E> c);
```

En este ejemplo, se indica que se acepta que la clase para la ordenación de los elementos sea un comparador del tipo E o de cualquiera de sus ancestros. Por último, indicar que es posible mezclar el uso de métodos genéricos y comodines ligados.

La interfaz *Collections* muestra un buen ejemplo de ello, como el método copy:

```
public static <T> void copy(List<? super T> dest, List<? extends T> src)
```

Este método acepta dos listas de elementos y copia todos los elementos de la lista src en dest. Para ello, el método se declara genérico, parametrizado por el tipo T. La lista origen restringe a T y sus derivados. La lista destino puede ser T o cualquiera de sus ancestros. De esta forma, se obtiene una gran flexibilidad.

Vista la teoría, ahora toca la práctica

Ejercicio 1

Escribe una clase Pila genérica usando para ello un atributo del tipo LinkedList. La clase Pila tendrá los siguientes métodos:

- **estaVacia():** devuelve true si la pila está vacía y false en caso contrario.
- **extraer():** devuelve y elimina el primer elemento de la colección.
- **primero():** devuelve el primer elemento de la colección
- **aniadir():** añade un objeto por el extremo que corresponda.
- **toString()**: devuelve en forma de String la información de la colección

Como puede hacerse

Vamos a aprovechar que la clase LinkedList es genérica, sólo se tiene que pasar como tipo genérico del atributo LinkedList como parámetro genérico de la clase Pila<E>.

- **estaVacía():** utiliza el método isEmpty() de LinkedList.
- **extraer():** utiliza el método removeFirst(), devuelve lo mismo que este método.
- **primero():** utiliza el método getFirst(), devuelve lo mismo que este método.
- **aniadir()**: utiliza el método addFirst().
- **toString()**: devuelve lo mismo que el método toString().

Solución

```
import java.util.LinkedList;

public class Pila<E> {

   private LinkedList<E> lista;

   public Pila() {
      lista = new LinkedList<E>();
   }

   public void aniadir(E o) {
      lista.addFirst(o);
   }

   public E primero() {
      return lista.getFirst();
   }

   public E extraer() {
      return lista.removeFirst();
   }

   public boolean estaVacia() {
      return lista.isEmpty();
   }

   public String toString() {
      return lista.toString();
   }
}
```

Ejercicio 2

Implementa una pila utilizando como atributos un array genérico y un entero que cuente el número de objetos insertados. La clase se debe llamar PilaArray y tiene los mismos métodos que la pila del ejercicio anterior.

Como puede hacerse

1. La clase PilaArray **tiene un parámetro genérico *E.***
2. Esta clase tiene como atributos un array del mismo tipo genérico *E* que la clase y un entero que sirve de contador de objetos. El constructor recibe por parámetro el tamaño máximo de la pila. **El método estaVacia() comprueba si el contador es 0.**
3. **El método aniadir()** recibe por parámetro un objeto de tipo *E*, comprueba que hay espacio libre y, si es así, lo añade en la celda que indica el contador. Posteriormente incrementa el valor del contador. Si se ha añadido, devuelve *true*. Si no se ha podido añadir, devuelve false.
4. **primero():** si está vacía, lanza NoSuchElementException. Si no está vacía, devuelve el elemento que está en el contador. El método se declara de tipo *E*, ya que los objetos que va a devolver son del tipo parámetro.
5. **extraer():** si está vacía, lanza NoSuchElementException. Si no está vacía, decrementa el contador y devuelve el elemento que está en la celda indicada por el contador después de decrementarse. Extraer se declara también de tipo *E*. ***Recuerda que NoSuchElementException hereda de RunTimeException, por lo que no se declara en la cláusula throws.***
6. Se sobreescribe el método **toString()**

Solución

```
import java.util.NoSuchElementException;

public class PilaArray {

   private E[] arrayGenerico; // Array de tipo genérico
   private int contador;

   public PilaArray(int capacidadMaxima) {
      arrayGenerico = (E[]) new Object[capacidadMaxima];
      contador = 0;
   }

   public boolean estaVacia() {
      return contador == 0;
   }

   public boolean aniadir(E objeto) {
      if (contador == arrayGenerico.length) {
         return false;
      } else {
         arrayGenerico[contador] = objeto;
         contador++;
         return true;
```

```
    }
  }

  public E primero() {
    if (estaVacia()) {
      throw new NoSuchElementException();
    } else {
      return arrayGenerico[–contador];
    }
  }

  public String toString() {
    return arrayGenerico.toString();
  }
}
```

NOTA: No se pueden crear array de tipo genérico, aunque sí se pueden declarar. Para superar esta dificultad, se crea un array de *Object* y se transforma a un array del tipo genérico *E*.

Ejercicio 3

Escribe una clase Matriz genérica con los siguientes métodos:

- constructor que recibe por parámetro el número de filas y columnas de la matriz.
- **set()** recibe por parámetro la fila, la columna y el elemento a insertar. El elemento es de tipo genérico. Este método inserta el elemento en la posición indicada.
- **get()** recibe por parámetro la fila y la columna. Devuelve el elemento en esa posición. El elemento devuelto es genérico.
- **columnas()** devuelve el número de columnas de la matriz.
- **filas()** devuelve el número de filas de las matriz.
- **toString()** devuelve en forma de String la información de la matriz.

Como puede hacerse

La matriz tiene un parámetro genérico *E*. El método set() recibe por parámetro, además de la posición, un elemento de tipo *E*. Y el método get() devuelve un elemento de tipo *E*.

El atributo donde se guarda la información es un array bidimensional de tipo *E*. Al crearlo, **como Java no permite crear arrays de un tipo genérico, se crea un array bidimensional de tipo *Object* y luego se convierte a array bidimensional de E.**

El método toString() utiliza un String auxiliar donde se acumula la información de cada celda, para separar unas celdas de otras, podemos utilizar el carácter '\t' (o si quieres, un espacio)

Solución

```
public class Matriz {

   private E[][] tabla;

   public Matriz(int filas, int columnas) {
      tabla = (E[][]) new Object[filas][columnas]; // NO SE PUEDEN CREAR
                                                   // ARRAYS
      // DE TIPO GENERICO, POR LO QUE SE UTILIZA LA CONVERSION
   }

   public void set(int fila, int columna, E elemento) {
      tabla[fila][columna] = elemento;
   }

   public E get(int fila, int columna) {
      return tabla[fila][columna];
   }

   public int columnas() {
      return tabla[0].length;
   }

   public int filas() {
      return tabla.length;
   }

   public String toString() {
      String s = "";
      for (int i = 0; i < tabla.length; i++) {
         for (int j = 0; j < tabla[0].length; j++) {
            s += tabla[i][j] + "\t";
         }
      }
      return s;
   }
}
```

NOTA: Las celdas de la matriz no se inicializan a ningún valor, por lo que el resultado de get() podría ser *null*. Al llamar al método toString(), también cabe esa posibilidad de que devuelva *null* para alguna de las celdas

Ejercicio 4

Escribe una aplicación que:

1. Cree una matriz de Integer de 4 filas y 2 columnas
2. Rellénala con números consecutivos comenzando por el 1.
3. Muestra por pantalla la matriz.
4. Muestra por pantalla el contenido de la celda en la fila 1, columna 2

Como puede hacerse

- Se crea la matriz de Integer, pasando las dimensiones por parámetro.
- Para rellenarla, se realizan dos *for* anidados que vayan pasando por todas las celdas de la matriz. Para el número a insertar se declara una variable de tipo *int*, que se incrementa después de cada inserción.
- Para mostrar la matriz, se utiliza el método toString().
- Para acceder a una celda determinada, se utiliza el método get().

Solución

```
public class MatrizMain {

   public static void main(String[] args) {

      Matriz matriz = new Matriz(4, 2);
      int numero = 1;

      for (int i = 0; i < matriz.filas(); i++) {
         for (int j = 0; j < matriz.columnas(); j++) {
            matriz.set(i, j, numero++);
         }
      }
      System.out.println(matriz.toString());
      System.out.println(matriz.get(1, 2));

   }

}
```

Ejercicio 5

Escribe una interfaz ColeccionSimpleGenerica, que como su propio nombre indica, es genérica, con los siguientes métodos:

- **estaVacia():** devuelve true si la pila está vacía y false en caso contrario

- **extraer():** devuelve y elimina el primer elemento de la colección.
- **primero():** devuelve el primer elemento de la colección.
- **aniadir():** añade un objeto por el extremo que corresponda.

Como puede hacerse

Se realiza igual que la interfaz *ColeccionSimple*, **pero declarando el tipo genérico en la cabecera de la interfaz**. Los parámetros y valores de retorno de los métodos son del tipo genérico en lugar de ser *Object*.

Solución

```
public interface ColeccionSimpleGenerica {

   boolean estaVacia();

   boolean aniadir(E o);

   E primero();

   E extraer();

}
```

Ejercicio 6

Escribe una clase genérica ListaOrdenada con un tipo parametrizado *E* que sea Comparable. La clase debe tener lo siguiente:

- Un constructor
- void add(E o)
- E get(int index)
- int size()
- boolean isEmpty()
- boolean remove(E o)
- int indexOf(E o)
- String toString()

Como puede hacerse

1. Para implementar esta ListaOrdenada, se declara un atributo de tipo *List*.
2. Todos los métodos, excepto el método add(), consisten en una llamada al método correspondiente de *List*.
3. En el método add() se recorre el atributo comparando el objeto a insertar con los objetos de la lista. Como ya se conoce que la clase de los objetos que se

insertan implementan Comparable, **se puede usar el método compareTo() para comparar los objetos.** Cuando se encuentra su posición se llama al método add(int index, E o) de *List,* para añadirlo en la posición específica. Este método ya se encarga de desplazar a la derecha el resto de los elementos de la lista. Si se termina de recorrer la lista, sin encontrar su posición es porque hay que añadirlo al final. Para eso, se llama al método add(E o) de *List*.

Solución

```
import java.util.ArrayList;
import java.util.List;

public class ListaOrdenada> {
   private List lista;

   public ListaOrdenada() {
      lista = new ArrayList();
   }

   public boolean add(E objeto) {
      for (int i = 0; i < lista.size(); i++) {
         if (objeto.compareTo(lista.get(i)) < 0) {
            lista.add(i, objeto);
            return true;
         }
      }
      lista.add(objeto);
      return true;
   }

   public E get(int index) {
      return lista.get(index);
   }

   public int size() {
      return lista.size();
   }

   public boolean remove(E objeto) {
      return lista.remove(objeto);
   }

   public boolean isEmpty() {
      return lista.isEmpty();
   }
```

```
    public int indexOf(E objeto) {
        return lista.indexOf(objeto);
    }

    public String toString() {
        String s = "";
        for (int i = 0; i < lista.size(); i++) {
            s += lista.get(i) + "\n";
        }
        return s;
    }

}
```

NOTA: El método indexOf() de las listas busca un elemento en la lista. Para comparar el objeto que entrar por parámetro con los objetos utiliza el método equals(). Recuerda que el método equals() de *Object* sólo compara referencias a objetos, como el operador ==

Ejercicio 7

Escribe una clase, de nombre ArrayListOrdenado, que herede de ArrayList. Esta clase sólo debe aceptar, como parámetro genérico de tipo, clases que implementen Comparable. La clase ArrayListOrdenado debe sobreescribir el método add(E objeto) para que añada los elementos en orden.

Como puede hacerse

1. La clase ArrayListOrdenado **hereda de ArrayList**. Recibe como parámetro genérico un tipo E que implemente Comparable. Este parámetro E se pasa en la cabecera de la clase a ArrayList.
2. En el método add() se recorre el atributo comparando el objeto a insertar con los objetos de la lista. Como ya se conoce que la clase de los objetos que se insertan implementa Comparable, **se puede usar el método compareTo() para comparar los objetos.** Cuando se encuentra su posición, se llama al método add(int index, E objeto) heredado de ArrayList, para añadirlo en la posición específica. Este método ya se encarga de desplazar a la derecha el resto de los elementos de la lista. Si se termina de recorrer la lista sin encontrar su posición, es porque hay que añadirlo al final. Para ésto, se llama al método add(E objeto) de la superclase.

Solución

```
import java.util.ArrayList;
```

```
public class ArrayListOrdenado> extends ArrayList {

   public boolean add(E objeto) {
      for (int i = 0; i < size(); i++) {
         if (objeto.compareTo(get(i)) < 0) {
            add(i, objeto);
            return true;
         }
      }
      super.add(objeto);
      return true;
   }

}
```

Nota: Aunque se haya sobreescrito este método, add(), la lista podría no mantenerse ordenada, si se utilizasen otros métodos para añadir heredados de ArrayList. Para solucionarlo, se podrían sobreescribir todos los métodos que permitan añadir y modificar.

Ejercicio 8

Escribe una interfaz genérica Operable, que sea genérica y que declare las cuatro operaciones básicas: suma, resta, producto y división.

Como puede hacerse

Se escribe la interfaz Operable, con un tipo genérico *E*. Todos los métodos de esta interfaz reciben un objeto de tipo *E* y devuelven un objeto de tipo *E*. De esta manera, se opera el objeto que llame al método con el que se recibe por parámetro y se devuelve el resultado. Ambos, operandos y el resultado, son del mismo tipo.

Solución

```
public interface Operable {
   E suma(E objeto);

   E resta(E objeto);

   E producto(E objeto);

   E division(E objeto);

}
```