Tema 4 –Colecciones y Genericidad

Programación Orientada a Objetos

Grado en Ingeniería Informática

DIS
Departamento de Informática y Sistemas

UNIVERSIDAD DE MURCIA

Anexo

- **Genericidad**:
  - Genericidad y sistema de tipos. Tipo comodín.
  - Genericidad restringida.
  - Genericidad y herencia.

Genericidad y sistema de tipos

```
List<Burbuja> burbujas = new LinkedList<Burbuja>();    //OK

Collections.addAll(burbujas, basica, limitada, debil,
                             creciente, sensible);


LinkedList<BurbujaDebil> debiles =
                             new LinkedList<BurbujaDebil>();

Collections.addAll(debiles, debil, creciente);



burbujas = debiles;          //error en tiempo de compilación
```

Genericidad y sistema de tipos

- Las reglas del **polimorfismo** se mantienen entre clases genéricas.

- Sin embargo, en una asignación polimórfica **no está permitido que tengan distintos parámetros**.
- En el ejemplo:
  - `LinkedList` es compatible con `List` y han sido parametrizadas al mismo tipo (`Burbuja`).
  - En la última asignación, aunque `LinkedList` es compatible con `List`, están parametrizadas a tipos distintos (`Burbuja` y `BurbujaDebil`).
    - No importa que `Burbuja` y `BurbujaDebil` sean compatibles.

- Es una **limitación en el paso de parámetros**.

Genericidad y sistema de tipos

- **Problema**: el método sólo permite variables cuyo tipo estático sea compatible con `List<Burbuja>`.

```
public class PruebaSimulador {

  private static void simular(List<Burbuja> burbujas){

    Simulador simulador = new Simulador(710, 710);

    for (Burbuja burbuja : burbujas)

        simulador.simular(burbuja);

  }

}
```

- ¿Cómo podemos pasar una variable de tipo `List<BurbujaDebil>`?

Genericidad y sistema de tipos

- **Solución**: **tipo comodín**.

```
private static void simular(
                          List<? extends Burbuja> burbujas){
```

- En el ejemplo significa: permite cualquier lista genérica parametrizada a la clase `Burbuja` o a un tipo compatible (subclase).

- El tipo comodín se puede usar **también para declarar variables locales o atributos**.

- **No se puede utilizar para construir objetos**.

- Si se indica simplemente `<?>`, significa “cualquier tipo”.

Limitaciones del tipo comodín

- En las colecciones declaradas utilizando el tipo comodín no es posible añadir nuevos objetos.

```
LinkedList<BurbujaDebil> debiles = new LinkedList<>();
BurbujaDebil debil = new BurbujaDebil(…);
BurbujaCreciente creciente = new BurbujaCreciente(…);

Collections.addAll(debiles,  debil, creciente);

LinkedList<? extends Burbuja> burbujas = debiles;

BurbujaLimitada limitada = new BurbujaLimitada(…);

burbujas.add(limitada);          //Error de compilación

burbujas.add(debil.clone());     //Error de compilación
```

Genericidad restringida

- **Objetivo**: limitar los tipos a los que puede ser parametrizada una clase genérica.

- Al restringir los tipos obtenemos el **beneficio** de poder **aplicar métodos** (además de los de `Object`) **sobre los objetos del tipo parametrizado**.

- Una clase con genericidad restringida sólo permite ser parametrizada con tipos **compatibles** **con el de la restricción** (clase o interfaz).

Genericidad restringida

- **Ejemplo**: la clase `Escenario` sólo puede ser parametrizada con tipos compatibles con `Animable`.
- `Animable` es una interfaz que declara los métodos necesarios para animar un elemento:

```
public class Escenario<T extends Animable> {

        private LinkedList<T> elementos;
        …
        public void accion() {
                for (T elemento : elementos)
                        elemento.animar();
        …}
}
```

Genericidad restringida

- **Ejemplo**: Si queremos utilizar burbujas en el escenario, debemos hacer que la clase `Burbuja` implemente la interfaz `Animable`:
  - La implementación de `animar` podría llamar a `ascender`.
- Una clase genérica puede estar **restringida por varios tipos**:

```
public class Escenario<T extends Animable & Atrapable>
```

- ➔ Las operaciones disponibles para objetos de tipo `T` es la unión de todos los tipos de la restricción.
  - En el ejemplo, todas las operaciones de la interfaz `Animable` y la interfaz `Atrapable`.

Genericidad y herencia

- Una clase puede heredar de una clase genérica.
- Una clase puede implementar una interfaz genérica.
- En cualquiera de los dos casos, si no se establece el tipo del parámetro, la clase descendiente sigue siendo genérica.
- Al heredar se puede reducir el número de parámetros.

Ejemplo herencia

```
public class Par<T, P> {
    private T valor1;
    private P valor2;
    public Par(T valor1, P valor2) {
        this.valor1 = valor1;
        this.valor2 = valor2;
    }
    public T getValor1() { return valor1; }
    public P getValor2() {  return valor2; }
    public void setValor1(T valor1) {
        this.valor1 = valor1; }
    public void setValor2(P valor2) {
        this.valor2 = valor2; }
     //continúa ...
}
```

Ejemplo herencia

```
public class Par<T, P> {


    //...
    @Override
    public String toString() {
            return getClass().getName()
                            + "[valor1=" + valor1
                            + ", valor2=" + valor2
                            + "]";
    }
}
```

Ejemplo herencia

```
public class ParUniforme<T> extends Par<T, T> {

    public ParUniforme(T valor1, T valor2) {
        super(valor1, valor2);
    }

    public boolean contiene(T valor) {
        return getValor1().equals(valor) ||

               getValor2().equals(valor);
    }
}
```

Ejemplo herencia

```
public class ParEntero extends ParUniforme<Integer>{

    public ParEntero(Integer valor1, Integer valor2) {
        super(valor1, valor2);
    }

    public ParEntero suma(ParEntero otroPar){
        return new ParEntero(
                getValor1() + otroPar.getValor1(),
                getValor2() + otroPar.getValor2());
    }
}
```

Ejemplo herencia

```
public class PruebaPar {
   public static void main(String[] args) {

      ParEntero par1 = new ParEntero(3,5);

      ParUniforme<Integer> par2 = par1;

      System.out.println(par2.contiene(5));

      ParEntero par3 = new ParEntero(2,6);

      System.out.println(par1.suma(par3));
   }
}
```