GENÉRICOS

Java permite definir clases con variables o parámetros que representan tipos y se conocen como clases genéricas. Esto es posible, cuando el tipo de un dato no afecta al tratamiento que se le va a dar a ese dato y suele darse al utilizar clases para implementar los TADs clásicos (Pila, Cola, Lista, etc.) mediante clases contenedoras.

Antes se utilizaba la clase Object como "comodín" para simular los genéricos, pero esto conllevaba:

- Se podían mezclar objetos de las clases más dispares en una misma estructura contenedora
- Había que realizar muy a menudo castings (conversión de Object a cualquier otra clase)

Ante esto surgen los genéricos, que permiten parametrizar el "tipo de contenido" de una clase contenedora **Ejemplo**: List<T> permite crear una lista de elementos de tipo T. De esta forma se verifica en tiempo de compilación el código, evitando mezclas en las clases contenedoras y la realización de castings.

NOTA: Permiten hacerlo "a la antigua usanza" pero el compilador nos avisa con un warning si no parametrizamos

Supongamos que en un programa necesitamos una clase ParEnteros:

```
public class ParEnteros {
  private int a, b;
  public ParEnteros(int a,int b) {
    this.a = a;
    this.b = b;
  }
  public ParEnteros swap () {
    return new ParEnteros(b, a);
  }
}
```

Pero también necesitamos un clase ParCaracteres con los mismos métodos que la anterior:

```
public class ParCaracteres {
  private char a, b;
  public ParCaracteres(char a,char b) {
    this.a = a;
    this.b = b;
  }
  public ParCaracteres swap () {
    return new ParCaracteres(b, a);
  }
}
```

La implementación de los métodos es la misma, porque no depende del tipo del dato manipulado.

SOLUCIÓN → Abstraemos el tipo convirtiéndolo en un parámetro genérico T.

```
public class Par<T> {
  private T a, b;
  public Par(T a, T b){
    this.a = a;
    this.b = b;
  }
  public Par<T> swap (){
    return new Par<T>(b, a);
  }
}
```

Donde T es el identificador que se asigna al tipo que se pasará cuando se instancie. Se puede poner cualquier identificador válido en Java, por ejemplo, Datos

```
public class Prueba1 {
   public static void main(String[] args) {
     Par<String> p1 = new Par<>("uno","dos");
     Par<Fecha> p2 = new Par<>(new Fecha(1,2,2003), new Fecha(4,5,2006));
     Par<Integer> p3 = new Par<>(2,3); // NO SE PUEDEN USAR OBJETOS DE TIPO BÁSICO
     System.out.println ("Primer elemento de p1: " + p1.getA());
     System.out.println ("Primer elemento de p2: " + p2.getA());
     System.out.println ("Primer elemento de p3: " + p3.getA());
     p1 = p1.swap();
     p2 = p2.swap();
     P3 = p3.swap();
     System.out.println ("Primer elemento de p1: " + p1.getA());
     System.out.println ("Primer elemento de p2: " + p2.getA());
     System.out.println ("Primer elemento de p3: " + p3.getA());
   }
}
```

Java no permite crear arrays de elementos genéricos.

```
public class ParArrayGenerico<T> {
   private T[] par = new T[2]; // ERROR
   ....
}
```

Solución 1: array de objects y downcasting.
→ LA QUE UTILIZAREMOS EN CLASE

```
public class ParArrayGenerico<T> {
  private T[] par;
  public ParArrayGenerico () {
    this.par = (T[]) new Object[2];
  }
}
```

Solución 2: reflexión y la clase array de la API.
→ MEJOR

```
public class ParArrayGenerico<T> {
  private T[] par;
  public ParArrayGenerico (Class<T> clase) {
    this.par = (T[]) Array.newInstance(clase, 2);
  }
}
```

Ejercicio: Implementar un TAD Pila acotada genérico BoundedStack<E>

- Constructor: recibe el tamaño máximo de la pila.
- Operaciones observadoras: isEmpty(), isFull()
- push(E elem): lanza la excepción FullStackExp si la pila está llena.
- pop(), E peek(): lanzan la excepción EmptyStackExp si la pila está vacía.