INTERFACES LIBRERIA JAVA1.8

Comparadores.-

Comparable <*Clase*>

Uso: Sirva para mantener el orden natural de objetos de *Clase*. Se debe implementar en la propia *Clase* que se deseé crear un orden natural.

Función a implementar :

```
public int compareTo(Clase o)
```

Ejemplo de implementación:

```
public class Persona implements Comparable<Persona> {

	private String nombre;
	private int edad;
	private LocalDate fecha;

	…
	@Override
	public int compareTo(Persona o) {
		// TODO Auto-generated method stub
		return nombre.compareTo(o.nombre);
	}
}
```

Ejemplo de uso:

```
List<Persona> personas  = new ArrayList<>();

…

// Ordenar alfabéticamente, ordenar por orden natural
Collections.sort(personas);
// Mostrar lista
Pantalla.escribirString("\n\nLista ordenada alfabéticamente");
for (Persona p :personas)
	Pantalla.escribirString("\n"+ p);
```

Comparator <*Clase*>

Uso: Sirva para mantener el orden total de objetos de *Clase*. Se debe implementar como una clase diferente a *Clase*.

Función a implementar :

```
public int compare(Clase o1, Clase o2)
```

Ejemplo de implementación:

```
public class ComparadorPorEdad implements Comparator<Persona>
{
	@Override
	public int compare(Persona o1, Persona o2) {
		// TODO Auto-generated method stub
		return o1.getEdad() - o2.getEdad();
	}

}
```

donde la Persona se define como

```
public class Persona {

	private String nombre;
	private int edad;
	private LocalDate fecha;

	...
}
```

Ejemplo de uso:

```
List<Persona> personas  = new ArrayList<>();
…
// Ordenar por edad (de menor a mayor), orden total
ComparadorPorEdad comparadorPorEdad = new ComparadorPorEdad();
Collections.sort(personas, comparadorPorEdad);
Pantalla.escribirString("\n\nLista ordenada por edad");
for (Persona p :personas)
	Pantalla.escribirString("\n"+ p);
```

Consumidor.-

Consumer <*Clase*>

Uso: Sirva para realizar una acción con objetos de *Clase*. Se debe implementar en una clase diferente a *Clase*.

Función a implementar :

```
public void accept(Clase t)
```

Ejemplo de implementación:

```
public class AccionMostrarPersona implements Consumer<Persona>
{
	@Override
	public void accept(Persona t) {
		// TODO Auto-generated method stub
		Pantalla.escribirString("\n" + t);
	}
}
```

Ejemplo de uso:

```
List<Persona> personas  = new ArrayList<>();
...
// Mostrar lista usando accion
Pantalla.escribirString("\nLista sin filtrar");
accion = new AccionMostrarPersona();
personas.forEach(accion);
```

La función `forEach` de `List`, recorre todo el conjunto de elementos que lo componen y a cada elemento le aplica la acción, es decir ejecuta la función *accept* del objeto *accion*, en el caso del ejemplo ejecutará *Pantalla.escribir("\n" + t)* para cada elemento(*t*) del conjunto *personas*.

Predicado o Filtro

Predicate <*Clase*>

Uso: Sirva para comprobar si una instancia de *Clase* cumple una determinada condición. Se debe implementar en una clase diferente a *Clase*.

Función a implementar :

```
public boolean test(Clase t)
```

Ejemplo de implementación:

```
public class FiltroPorEdad implements Predicate<Persona> {

	private int edad;
	public FiltroPorEdad(int edad)
	{
		this.edad = edad;
	}
	public int getEdad() {
		return edad;
	}
	public void setEdad(int edad) {
		this.edad = edad;
	}

	@Override
	public boolean test(Persona t) {
		// TODO Auto-generated method stub
		return t.getEdad() >= edad;
	}
}
```

Ejemplo de uso:

```
List<Persona> personas  = new ArrayList<>();
...
// Filtrado por  edad >= 30
FiltroPorEdad filtro = new FiltroPorEdad (30);
personas.removeIf(filtro);
```

La función `removeIf` de `List`, recorre todo el conjunto de elementos que lo componen y cada elemento le aplica el filtro, es decir ejecuta la función *`test`* del objeto *`filtro`*, en el caso del ejemplo ejecutará *`return t.getEdad() >= edad;`* para cada elemento(*`t`*) del conjunto *`personas`*. Si la función devuelve true, el predicado se cumple cumple y por tanto el elemento es borrado de la lista, se filtra.