Nombre:..... Legajo:.....

Recuperatorio del Segundo Parcial de Programación Orientada a Objetos

7 de Diciembre de 2009

Ej. 1	Ej. 2	Ej. 3	Nota

- **Condición de aprobación: Tener dos ejercicios calificados como B o B-.**
- **Los ejercicios que no se ajusten estrictamente al enunciado, no serán aceptados.**
- **No es necesario agregar las sentencias import**

Ejercicio 1

Se cuenta con la interface **`Criteria`** que representa una condición aplicada a un objeto, la interface **`FilterList`** que utiliza a **`Criteria`** para filtrar los elementos de una lista y la clase **`ArrayListFiltered`** que implementa **`FilterList`**:

Criteria.java

```
/** Modela un criterio a aplicar a un objeto de tipo T
 */
public interface Criteria<T> {
         boolean satisfies(T obj);
}
```

FilterList.java

```
/** Modela una lista que puede ser filtrada según un criterio
 */
public interface FilterList<T> extends List<T>{
         public FilterList<T> filter(Criteria<T> criteria);
}
```

ArrayListFiltered.java

```
public class ArrayListFiltered<T> extends ArrayList<T> implements FilterList<T>{

        public FilterList<T> filter(Criteria<T> criteria) {
                FilterList<T> result = new ArrayListFiltered<T>();
                for(T elem: this) {
                        if (criteria.satisfies(elem)) {
                                result.add(elem);
                        }
                }
                return result;
        }
}
```

Se pide implementar lo que considere necesario, sin modificar nada de lo existente para poder aplicarle a una lista varias implementaciones de **`Criteria`** en forma simultánea obteniendo como resultado la sublista de elementos que cumplen con todas las condiciones.

Utilizar lo implementado para completar el código de ejemplo realizando lo indicado en el comentario:

```
public class TestFilterList {

        public static void main(String[] args) {
            FilterList<Integer> list = new ArrayListFiltered<Integer>();

            list.add(4);
            list.add(3);
            list.add(15);
            list.add(16);
            list.add(20);

            // A partir de acá completar lo necesario para filtrar la lista
            // quedándose en result con los números pares de dos cifras.

            FilterList<Integer> result = list.filter(criteria);
        }
}
```

Ejercicio 2

Se cuenta con la siguiente clase que modela un teléfono celular:

CellPhone.java

```
public abstract class CellPhone {

	private String number;
	private boolean isEnabled;

	public CellPhone(String number) {
		this.number = number;
		this.isEnabled = true;
	}

	public void enable() {
		this.isEnabled = true;
	}

	public void disable() {
		this.isEnabled = false;
	}

	public void makeCall(String toNumber) {
		if (!isEnabled) {
			throw new DisabledCellPhoneException();
		}
		onCall(toNumber);
	}

	public String getNumber() {
		return number;
	}

	protected abstract void onCall(String toNumber);

	@Override
	public String toString() {
		return number;
	}

	@Override
	public int hashCode() {
		return number.hashCode();
	}

	@Override
	public boolean equals(Object obj) {
		if (obj == null || !(obj instanceof CellPhone)) {
			return false;
		}
		return number.equals(((CellPhone)obj).number);
	}
}
```

Se quiere implementar una promoción en la cual existen grupos de celulares que comparten el crédito (se considera que se consume una unidad de crédito por llamada). Las llamadas entre celulares de un mismo grupo son gratuitas.

Para esto se deben implementar las clases **`FamilyCellPhone`** y **`FamilyCellPhoneGroup`**. **`FamilyCellPhone`** modela al celular que puede pertencer a un grupo y **`FamilyCellPhoneGroup`** modela al grupo de celulares que comparten el crédito.

El siguiente código muestra un ejemplo de uso de las clases mencionadas y su salida. No es necesario implementar algo que no se observe en el ejemplo.

```
public class TestCellPhone {

	public static void main(String[] args) {

		// Crea un grupo con crédito para 2 llamadas.
		FamilyCellPhoneGroup group = new FamilyCellPhoneGroup(2);

		// Crea y agrega tres celulares al grupo.
		CellPhone phone1 = new FamilyCellPhone(group, "111-111");
		CellPhone phone2 = new FamilyCellPhone(group, "222-222");
		CellPhone phone3 = new FamilyCellPhone(group, "333-333");

		phone1.makeCall("222-222");   // Estas dos llamadas no se cobran porque
		phone2.makeCall("111-111");   // son internas al grupo
		phone3.makeCall("555-555");
```

```
                // Crea otro grupo con crédito para 2 llamadas.
                FamilyCellPhoneGroup group2 = new FamilyCellPhoneGroup(2);

                // Imprime 1
                System.out.println(group.getAvailableCalls());

                // Imprime 2
                System.out.println(group2.getAvailableCalls());

                // Agrega crédito para 2 llamadas.
                group.loadCredit(2);

                phone1.makeCall("444-444");
                phone2.makeCall("777-777");
                phone3.makeCall("888-888");

                group.showPhones();   // Imprime:
                                      //     333-333
                                      //     222-222
                                      //     111-111

                phone1.makeCall("555-555");    // Lanza NoCreditException
        }
}
```

Ejercicio 3

Se cuenta con la interface **`Cache`** que permite almacenar pares clave-valor y es la siguiente:

```
public interface Cache<K,V> {

        /**
         * Agrega el par clave-valor al cache
         */
        public void add(K key, V value);

        /**
         * Devuelve el valor asociado a una clave o null si la clave no existe
         */
        public V get(K key);

        /**
         * Devuelve la cantidad de claves almacenadas
         */
        public int size();
}
```

Se pide implementar la clase **`LimitedCache`** que implementa la interface anterior y que tiene un límite de capacidad de pares. Al momento de agregar una clave, si se llegó al límite de la capacidad, se debe eliminar una clave existente. La clave a eliminar se elige considerando la cantidad de consultas que se hicieron sobre la misma. Se elimina la clave menos consultada.

El que sigue es un ejemplo de uso de dicha clase:

```
public class TestCache {

        public static void main(String[] args) {
                Cache<String, String> cache = new LimitedCache<String, String>(3);

                cache.add("Hola", "mundo");
                cache.add("Soy", "Laura");
                cache.add("key", "value");

                System.out.println(cache.get("key"));       //Imprime "value"
                System.out.println(cache.get("Soy"));       //Imprime "Laura"
                System.out.println(cache.get("Soy"));       //Imprime "Laura"

                cache.add("K", "V");                //Se elimina "Hola" por ser la menos consultada

                System.out.println(cache.get("Hola"));      //Imprime "null" porque "Hola" no está

                cache.add("72.33", "POO");          //Se elimina "K" por ser la menos consultada

                System.out.println(cache.get("K"));         //Imprime "null" porque "K" no está
                System.out.println(cache.get("72.33"));     //Imprime "POO"
                System.out.println(cache.get("72.33"));     //Imprime "POO"

                cache.add("x", "y");                //Se elimina "key" por ser la menos consultada
                System.out.println(cache.get("key"));       //Imprime "null"
        }
}
```