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

Segundo Parcial de Programación Orientada a Objetos

29 de Noviembre de 2012 – 14hs.

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.**
- **Además de las clases pedidas se pueden agregar las que se consideren necesarias.**

Ejercicio 1

Se tiene la interfaz Set (**no** es la interface Set que ofrece Java) que modela un conjunto inmutable y se muestra a continuación:

```
public interface Set<T> {

      /** Indicates whether or not element is contained in this set. */
      public boolean contains(T element);

      /** Returns a new set including the given element */
      public Set<T> include(T element);

      /** Returns a new set that has all the elements that belong to this set or the given one   */
      public Set<T> union(Set<T> that);

      /** Returns a new set that has all the elements that belong to this set and the given one. */
      public Set<T> intersect(Set<T> that);
}
```

Ninguna de las operaciones realizadas sobre el conjunto alteran su estado interno. Los métodos, **include**, **union** e **intersect** retornan nuevos conjuntos dejando intacto al receptor.

Escribir una implementación de dicha interfaz que utilice la técnica de definición de conjuntos por comprensión: en lugar de guardar explícitamente cada uno de los elementos que el conjunto contiene, se guarda un predicado que describe qué características deben tener los elementos. Por ejemplo, un posible predicado podría ser: "Todos los números naturales pares".

Realizar todo lo necesario para que al ejecutar el siguiente ejemplo de uso se obtenga la salida indicada en comentarios:

```
public class Example {
      public static void main(String[] args) {
               Predicate<Integer> oneToTenPredicate = new OneToTenPredicate();
               Predicate<Integer> evenPredicate = new EvenPredicate();
               Predicate<Integer> equalsToElevenPredicate = new EqualsToElevenPredicate();

               Set<Integer> set1 = new InmutableSet<Integer>(oneToTenPredicate);
               Set<Integer> set2 = new InmutableSet<Integer>(evenPredicate);
               Set<Integer> set3 = new InmutableSet<Integer>(equalsToElevenPredicate);

               System.out.println(set1.contains(0));       // false
               System.out.println(set1.contains(1));       // true
               System.out.println(set1.contains(2));       // true

               System.out.println(set2.contains(2));       // false
               System.out.println(set2.contains(3));       // true

               System.out.println(set3.contains(11));      // true

               Set<Integer> set4 = set1.include(0);
               System.out.println(set1.contains(0));       // false
               System.out.println(set4.contains(0));       // true

               Set<Integer> set5 = set1.union(set3);
               System.out.println(set5.contains(10));      // true
               System.out.println(set5.contains(11));      // true

               Set<Integer> set6 = set5.intersect(set2);
               System.out.println(set6.contains(10));      // false
               System.out.println(set6.contains(11));      // true
       }
}
```

Definir la interfaz **Predicate** y las clases **OneToTenPredicate**, **EvenPredicate**, **EqualsToElevenPredicate e InmutableSet**.

Ejercicio 2

Se tiene la interfaz **List** que modela una *lista lineal simplemente encadenada*.

```
public interface List<T> {

      /** Indicates whether or not this list is empty. */
      public boolean isEmpty();

      /** Returns the head of this list.
       * @throws NoSuchElementException if the list is empty. */
      public T head();

      /** Returns the tail of this list.
       * @throws NoSuchElementException if the list is empty. */
      public List<T> tail();

      /** Returns the number of elements of this list. */
      public int size();

      /** Indicates whether or not the given element is contained in this list. */
      public boolean contains(T elem);
}
```

Se desea escribir una implementación inmutable de dicha interfaz. Realizar todo lo necesario para que al ejecutar el siguiente fragmento de código se obtenga la salida indicada (y se cumpla con el contrato documentado):

```
public class Example {
  public static void main(String[] args) {
    List<Integer> list1 = new Empty<Integer>();
    List<Integer> list2 = new NonEmpty<Integer>(10, new NonEmpty<Integer>(20, new Empty<Integer>()));

    print(list1); // Imprime: ""
    print(list2); // Imprime: "10 20"

    System.out.println(list1.size()); // Imprime: "0"
    System.out.println(list2.size()); // Imprime: "2"

    List<Integer> list3 = new NonEmpty<Integer>(0, list2);
    List<Integer> list4 = new NonEmpty<Integer>(5, list2);
    print(list2); // Imprime: "10 20"
    print(list3); // Imprime: "0 10 20"
    print(list4); // Imprime: "5 10 20"

    System.out.println(list2.contains(0)); // Imprime: "false"
    System.out.println(list3.contains(0)); // Imprime: "true"
  }

  private static <T> void print(List<T> list) {
    while (!list.isEmpty()) {
      System.out.print(list.head() + " ");
      list = list.tail();
    }
    System.out.println();
  }
}
```

Ejercicio 3

Se cuenta con la clase **RankMap** que implementa un mapa que lleva registro de la fecha de último acceso a cada clave y del ranking de cada una (cantidad de accesos):

```
public class RankMap<K,V> {

       private Map<K,RankEntry> map = new HashMap<K, RankEntry>();

       public V get(K key) {
              RankEntry e = map.get(key);
              e.addAccess();
              return e.getValue();
       }

       public Date getLastAccess(K key) {
              RankEntry e = map.get(key);
              if (e != null) {
                     return map.get(key).getLast();
              }
              throw new NoSuchKeyException();
       }
```

```
	public int getRank(K key) {
		RankEntry e = map.get(key);
		if (e != null) {
			return map.get(key).getRank();
		}
		throw new NoSuchKeyException();
	}

	public void put(K key, V value) {
		RankEntry e = map.get(key);
		if (e == null) {
			e = new RankEntry(value);
			map.put(key, e);
		} else {
			e.setValue(value);
		}
	}

	public void remove(K key) {
		map.remove(key);
	}

	protected Map<K,RankEntry> getMap() {
		return map;
	}

	private class RankEntry {

		private V value;
		private int rank = 0;
		private Date last = null;

		RankEntry(V value) {
			this.value = value;
		}
		void setValue(V value) {
			this.value = value;
		}
		void addAccess() {
			rank++;
			last = new Date();
		}
		Date getLast() {
			return last;
		}
		int getRank() {
			return rank;
		}
		V getValue() {
			return value;
		}
	}
}
```

Se pide implementar la clase **RankIterable** que ofrece un **iterador sobre los elementos con N o más accesos**. A continuación se muestra un posible programa de ejemplo con su correspondiente salida:

RankMapExample

```
public class RankMapExample {

	public static void main(String[] args) {

		RankMap<String, String> map = new RankIterable<String, String>();

		map.put("google", "www.google.com");
		map.put("gmail", "gmail.google.com");
		map.put("iol", "www.iol.itba.edu.ar");
		map.put("twitter", "www.twitter.com");
		map.put("hola", "www.holamundo.com");
		map.put("hello", "www.helloworld.com");

		/* Se pide un iterador sobre claves que tengan 4 o más accesos */
		Iterator<String> it = ((RankIterable<String, String>) map).iterator(4);

		map.get("gmail"); map.get("gmail");
		map.get("iol");
		map.get("google"); map.get("google"); map.get("google"); map.get("google");
		map.get("hola"); map.get("hola"); map.get("hola");map.get("hola"); map.get("hola");
		map.get("google"); map.get("google"); map.get("google"); map.get("google");
		map.get("twitter"); map.get("twitter"); map.get("twitter"); map.get("twitter");
		map.get("hello"); map.get("hello"); map.get("hello");
```

```
                System.out.println("4 o más consultas");
                while (it.hasNext()) {
                        String key = it.next();
                        System.out.println( key + " fue consultado " + map.getRank(key) + " veces");
                        if (map.getRank(key) % 2 == 0) {
                                it.remove();
                        }
                }

                System.out.println("\n4 o más consultas luego del remove");
                it = ((RankIterable<String, String>) map).iterator(4);
                while (it.hasNext()) {
                        String key = it.next();
                        System.out.println( key + " fue consultado " + map.getRank(key) + " veces");
                }

                it = ((RankIterable<String, String>) map).iterator(4);
                try {
                        it.remove();
                } catch (IllegalStateException e) {
                        System.out.println("Lanzo excepción");
                }
        }
}
```

Salida

```
4 o más consultas
twitter fue consultado 4 veces
google fue consultado 8 veces
hola fue consultado 5 veces

4 o más consultas luego del remove
hola fue consultado 5 veces
Lanzo excepción
```

Se quiere modelar el menú de una pizzería. Se tienen 2 tipos de pizza: al horno y a la parrilla y varios agregados que se le pueden poner encima a la pizza: extra queso, tomate y cebolla. A continuación se muestra los precios de los tipos de pizza y de los toppings:

- Pizza al horno: $10
- Pizza a la parrila: $15
- Extra queso: $2
- Tomate: $3
- Cebolla: $1

Cada cliente puede Realizar todo lo necesario para que al ejecutar el siguiente fragmento de código se obtenga la salida indicada.

```
public class Example {
  public static void main(String[] args) {
    // barbacue pizza with tomatoe topping
    Pizza pizza1 = new TomatoTopping(new BarbacuePizza());

    // barbacue pizza with tomato, onion and extra cheese topping
    Pizza pizza2 = new ExtraCheeseTopping(new OnionTopping(new TomatoTopping(new BarbacuePizza())));

    // oven pizza with onion and extra cheese topping
    Pizza pizza3 = new ExtraCheeseTopping(new OnionTopping(new OvenPizza()));

    System.out.println(pizza1.getDescription() + ": " + pizza1.getPrice());
    System.out.println(pizza2.getDescription() + ": " + pizza2.getPrice());
    System.out.println(pizza3.getDescription() + ": " + pizza3.getPrice());
  }
}
```

```
Pizza a la parrila con tomate: 18.0
Pizza a la parrila con tomate, cebolla, extra queso: 21.0
Pizza al horno con cebolla, extra queso: 13.0
```

Variante con 2 métodos más en Pizza

```
public class Example {
  public static void main(String[] args) {
    // barbacue pizza with tomatoe topping
    Pizza pizza1 = new TomatoTopping(new BarbacuePizza());

    // barbacue pizza with tomato, onion and extra cheese topping
    Pizza pizza2 = new ExtraCheeseTopping(new OnionTopping(new TomatoTopping(new BarbacuePizza())));

    // oven pizza with onion and extra cheese topping
    Pizza pizza3 = new ExtraCheeseTopping(new OnionTopping(new OvenPizza()));

    print(pizza1);
    print(pizza2);
    print(pizza3);
  }

  private static void print(Pizza p) {
    System.out.println(p.getDescription() + ": " + p.getPrice() + " - " +
        p.getPizzaType() + "(" + p.getNumOfToppings() + ")");
  }
}
```

```
Pizza a la parrila con tomate: 18.0 - Parrilla(1)
Pizza a la parrila con tomate, cebolla, extra queso: 21.0 - Parrilla(3)
Pizza al horno con cebolla, extra queso: 13.0 – Horno(2)
```