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

Recuperatorio del Segundo Parcial de Programación Orientada a Objetos

28/06/2011 – 15hs.

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

Ejercicio 1

Una bolsa es una colección de elementos sin orden que admite repetidos. Dadas las siguientes clases y el ejemplo de uso, implementar lo necesario para que el ejemplo funcione:

```
public interface Bag<T> {

	/** Agrega el elemento a la bolsa. */
	public void add(T elem);

	/** Elimina el elemento de la bolsa. Si la bolsa contiene muchas veces un mismo elemento
	 * serán necesario que se remueva muchas veces dicho elemento. */
	public void remove(T elem);

	/** Retorna el número de elementos equivalentes al recibido que hay en la bolsa. */
	public int count(T elem);

	/** Retorna el conjunto de elementos distintos de la bolsa. */
	public Set<T> distinct();
}
```

```
public class BagImpl<T> implements Bag<T> {

	private Map<T, Integer> values = new HashMap<T, Integer>();

	public void add(T elem) {
		if (values.get(elem) == null) {
			values.put(elem, 1);
		} else {
			values.put(elem, values.get(elem) + 1);
		}
	}

	public int count(T elem) {
		if (values.get(elem) == null) {
			return 0;
		} else {
			return values.get(elem);
		}
	}

	public void remove(T elem) {
		if (values.get(elem) == null) {
			throw new NoSuchElementException();
		} else if (values.get(elem) == 1) {
			values.remove(elem);
		} else {
			values.put(elem, values.get(elem) - 1);
		}
	}

	public Set<T> distinct() {
		return new HashSet<T>(values.keySet());
	}
}
```

```
public class BagExample {
	public static void main(String[] args) {
		IterableBagImpl<String> bag = new IterableBagImpl<String>();

		bag.add("perro"); bag.add("perro"); bag.add("perro");
		bag.add("gato");  bag.add("raton"); bag.add("raton");

		for (String s : bag) {		// imprime: gato raton raton perro perro perro
			System.out.println(s);
		}

		removeOnce(bag, "perro");
		removeOnce(bag, "gato");

		for (String s : bag) {		// imprime: raton raton perro perro
			System.out.println(s);
		}

	}

	private static void removeOnce(IterableBagImpl<String> bag, String elem) {
		Iterator<String> it = bag.iterator();
		while (it.hasNext()) {
			if (it.next().equals(elem)) {
				it.remove();
				return;
			}
		}
	}
}
```

Ejercicio 2

Se tienen las siguientes clases que modelan un banco, sus cuentas y transacciones:

```
public class Bank {

	private String name;
	private Set<Account> accounts;

	public Bank(String name) {
		this.name = name;
		this.accounts = new HashSet<Account>();
	}

	public Account openAccount(String code, String client) {
		Account account = createAccount(code, client);
		if (accounts.contains(account)) {
			throw new IllegalArgumentException("Duplicated client.");
		}
		accounts.add(account);
		return account;
	}

	protected Account createAccount(String code, String client) {
		return new Account(this, code, client);
	}

	public Set<Account> getAccounts() {
		return accounts;
	}

	public String getName() {
		return name;
	}
}
```

```
public abstract class Transaction {

	private double amount;
	private Account account;
	private String date;

	public Transaction(double amount, Account account, String date) {
		this.amount = amount;
		this.account = account;
		this.date = date;
	}

	public Account getAccount() {
		return account;
	}
```

```
	public double getAmount() {
		return amount;
	}
	public String getDate() {
		return date;
	}
	public String getDescription() {
		return amount + " on " + date;
	}
	public abstract double apply();
}
```

```
public class CreditTransaction extends Transaction {
	public CreditTransaction(double amount, Account account, String date) {
		super(amount, account, date);
	}
	public double apply() {
		return getAccount().getBalance() + getAmount();
	}
	public String getDescription() {
		return "Credit " + super.getDescription();
	}
}
```

```
public class DebitTransaction extends Transaction {
	public DebitTransaction(double amount, Account account, String date) {
		super(amount, account, date);
	}
	public double apply() {
		return getAccount().getBalance() - getAmount();
	}
	public String getDescription() {
		return "Debit " + super.getDescription();
	}
}
```

```
public class Account {

	private String code;
	private String client;
	private Bank bank;
	private double balance;
	private List<Transaction> transactions = new ArrayList<Transaction>();

	protected Account(Bank bank, String code, String client) {
		this.bank = bank;
		this.code = code;
		this.client = client;
	}
	public void credit(double amount, String date) {
		onTransaction(new CreditTransaction(amount, this, date));
	}
	public void debit(double amount, String date) {
		onTransaction(new DebitTransaction(amount, this, date));
	}
	protected void onTransaction(Transaction tr) {
		double newBalance = tr.apply();
		if (newBalance < 0) {
			throw new IllegalStateException("Not enough money.");
		}
		this.balance = newBalance;
		transactions.add(tr);
		System.out.println(tr.getDescription() + "--> Account " + code +
                                                    ": Balance: " + balance);
	}
	public void transfer(Account target, double amount, String date) {
		debit(amount, date);
		target.credit(amount, date);
	}
	public void reApplyTransactions(){
		balance = 0;
		for(Transaction tr : transactions) {
			balance = tr.apply();
		}
	}
```

```
	public Bank getBank() {
		return bank;
	}
	public String getClient() {
		return client;
	}

	/* Omitimos aquí el hashcode e equals. Asumir que están escritos correctamente */
}
```

Se pide implementar lo necesario para ofrecer un nuevo tipo de banco que pague intereses cuando se desee en base al dinero depositado en cada cuenta en ese instante, y además restrinja el monto que se retira de una cuenta por día. Se asume que las transacciones llegan en orden cronológico y no es necesario validarlo.

El que sigue es un programa de ejemplo y luego la salida:

```
public class Test {

	public static void main(String[] args) {

		SpecialBank bank1 = new SpecialBank("Banco 1", 1000.0); /* Limite de retiro 1000 */
		Bank bank2 = new Bank("Banco 2");

		Account account1 = bank1.openAccount("001", "Juan");
		Account account2 = bank1.openAccount("002", "Ana");
		Account account3 = bank2.openAccount("003", "Pablo");

		account1.credit(3000.0, "2011-06-20");
		account2.credit(2500.0, "2011-06-20");
		account3.credit(3500.0, "2011-06-21");

		account1.debit(600, "2011-06-24");
		account1.debit(700, "2011-06-25");
		account2.debit(1000, "2011-06-25");
		account1.debit(200, "2011-06-25");

		try {
			account1.debit(200, "2011-06-25");
		} catch (LimitExceededException e) {
			System.out.println("Can't debit $200 from account 001");
		}

		try {
			account1.transfer(account2, 500, "2011-06-25");
		} catch (LimitExceededException e) {
			System.out.println("Can't transfer $500 from account 001 to account 002");
		}

		account2.transfer(account1, 500, "2011-06-26");

		bank1.payInterest(1.05, "2011-06-27");

		try {
			account2.transfer(account3, 500, "2011-06-27");
		} catch (LimitExceededException e) {
			System.out.println("Can't transfer $500 from account 002 to account 003");
		}
	}
}
```

```
Credit 3000.0 on 2011-06-20--> Account 001: Balance: 3000.0
Credit 2500.0 on 2011-06-20--> Account 002: Balance: 2500.0
Credit 3500.0 on 2011-06-21--> Account 003: Balance: 3500.0
Debit 600.0 on 2011-06-24--> Account 001: Balance: 2400.0
Debit 700.0 on 2011-06-25--> Account 001: Balance: 1700.0
Debit 1000.0 on 2011-06-25--> Account 002: Balance: 1500.0
Debit 200.0 on 2011-06-25--> Account 001: Balance: 1500.0
Can't debit $200 from account 001
Can't transfer $500 from account 001 to account 002
Debit 500.0 on 2011-06-26--> Account 002: Balance: 1000.0
Credit 500.0 on 2011-06-26--> Account 001: Balance: 2000.0
Interest 1.05 on 2011-06-27--> Account 001: Balance: 2100.0
Interest 1.05 on 2011-06-27--> Account 002: Balance: 1050.0
Debit 500.0 on 2011-06-27--> Account 002: Balance: 550.0
Credit 500.0 on 2011-06-27--> Account 003: Balance: 4000.0
```

Ejercicio 3

Se tiene la interfaz **ReversibleMap** que modela un mapa reducido que ofrece operaciones simples para manipular claves con valores asociados, y permite obtener un mapa "reverso" del mismo. Esto es un mapa donde las claves pasan a ser los valores del mapa original y los valores son las claves del mapa original:

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

	/** Indica si el mapa contiene la clave recibida. */
	public boolean containsKey(K key);

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

	/** Agrega al mapa la clave con el valor asociado (si la clave ya se encuentra
	 * en el mapa, se sobreescribe el valor). Si el valor ya se encuentra asociado
	 * a alguna otra clave se lanza la excepción DuplicatedElementException. */
	public void put(K key, V value);

	/** Elimina la clave y su valor asociado del mapa. Si no existe, no hace nada. */
	public void remove(K key);

	/** Obtiene un mapa reverso del mapa que recibe el mensaje.  Las operaciones que se
	 * realicen sobre el objeto devuelto se propagan sobre el mapa original. */
	public ReversibleMap <V, K> reverse();  /* Importante: es V,K y no K,V*/
}
```

Se pide escribir la clase **ReversibleHashMap** que implementa la interfaz definida. No es necesario implementar la excepción DuplicatedElementException. A continuación se muestra un programa de ejemplo:

```
public class ReversibleMapExample {
	public static void main(String[] args) {
		ReversibleMap<Integer, String> map = new ReversibleHashMap<Integer, String>();

		map.put(1, "hola");
		map.put(2, "mundo");

		ReversibleMap<String, Integer> reverseMap = map.reverse();

		System.out.println(reverseMap.get("hola"));		// imprime 1
		System.out.println(reverseMap.get("casa"));		// imprime null
		System.out.println(reverseMap.containsKey("mundo"));	// imprime true

		reverseMap.put("casa", 8);
		reverseMap.put("hola", 4);

		System.out.println(map.get(1));	// imprime null
		System.out.println(map.get(4));	// imprime hola
		System.out.println(map.get(8));	// imprime casa

		reverseMap.remove("hola");

		System.out.println(map.containsKey(4));	// imprime false

		map.remove(8);
		System.out.println(reverseMap.get("casa")); // imprime null

		reverseMap.put("clave", 2);		// Lanza DuplicatedElementException
	}
}
```

No es una solución aceptable que la clase `ReversibleHashMap` utilice dos mapas