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

Segundo Parcial de Programación Orientada a Objetos

25 de Junio de 2012 – 14hs.

Ej. 1	*Ej. 2*	*Ej. 3*	*Nota*

- **<u>Condición de aprobación: Tener dos ejercicios calificados como B o B-.</u>**
- **Los ejercicios que no se ajusten estrictamente al enunciado, <u>no serán aceptados</u>.**
- **No es necesario agregar las sentencias <u>import.</u>**
- **Además de las clases pedidas se pueden agregar las que se consideren necesarias.**

Ejercicio 1

Se tienen las clases **Course** y **Student** que modelan un curso y a un alumno respectivamente. El curso lleva registro de los inscriptos al mismo y el alumno de las materias que tiene aprobadas:

```
public class Course {
	private String subjectName;
	private List<Student> students;

	public Course(String subjectName) {
		this.subjectName = subjectName;
		this.students = new ArrayList<Student>();
	}

	public String getSubjectName() {
		return subjectName;
	}

	public void register(Student student) {
		if (!students.contains(student)) {
			students.add(student);
		}
	}

	public List<Student> getStudents() {
		return students;
	}

	public void printStudents() {
		System.out.print("Registered students: ");
		for (Student s: students) {
			System.out.print(s.getIdentification() + " ");
		}
		System.out.println();
	}
}
```

```
public class Student {
	private String identification;
	private Set<String> approvedSubjects;

	public Student(String identification) {
		this.identification = identification;
		this.approvedSubjects = new HashSet<String>();
	}

	public String getIdentification() {
		return identification;
	}

	public void addApprovedSubject(String subjectName) {
		approvedSubjects.add(subjectName);
	}

	public boolean hasApprovedSubject(String subjectName) {
		return approvedSubjects.contains(subjectName);
	}
}
```

Se quiere agregar la posibilidad de realizar diferentes validaciones antes de confirmar la inscripción de un alumno al curso:

- **Cupo:** No debe ser posible agregar un alumno al curso si el número de inscriptos ya es igual al cupo.
- **Correlativas:** No debe ser posible que un alumno se inscriba a un curso sino tiene aprobadas ciertas materias.

Estas validaciones se deben realizar en el orden en que son agregadas, y al primer fallo, no se debe continuar procesando las restantes. Asimismo se debe llevar registro para cada curso del número de veces que cada validador impidió una registración.

Escribir las clases necesarias para que al ejecutar el siguiente fragmento de código se obtenga la salida indicada. No es posible modificar las clases **Course** ni **Student.**

```
public class Example {
	public static void main(String[] args) {
		Student s1 = new Student("45001");
		s1.addApprovedSubject("pi");		// El alumno tiene aprobada pi

		Student s2 = new Student("45002");
		s2.addApprovedSubject("poo");

		Student s3 = new Student("45003");
		s3.addApprovedSubject("poo");

		Student s4 = new Student("45004");
		Student s5 = new Student("45005");
		Student s6 = new Student("45006");

		QuotaValidator quotaValidator = new QuotaValidator(2);

		EnhancedCourse mate1 = new EnhancedCourse("mate1");
		mate1.addValidator(quotaValidator);
		mate1.register(s1); mate1.register(s2);	// Pasan las validaciones
		mate1.register(s3);			// Falla por falta de cupo

		System.out.println("MATE1: ");
		mate1.printStudents();

		EnhancedCourse eda = new EnhancedCourse("eda");
		eda.addValidator(quotaValidator);
		List<String> dependencies = new ArrayList<String>();

		dependencies.add("poo");
		eda.addValidator(new DependenciesValidator(dependencies));

		eda.register(s1);			// Falla porque no tiene aprobado poo
		eda.register(s2); eda.register(s3);	// Pasan las validaciones
		eda.register(s4); eda.register(s5); eda.register(s6); // Fallan por falta de cupo

		System.out.println("\nEDA: ");
		eda.printStudents();
	}
}
```

```
MATE1:
Registered students: 45001 45002

EDA:
Registered students: 45002 45003
```

Ejercicio 2

Se tiene el siguiente código de ejemplo que muestra el comportamiento de un conjunto de clases que modelan el sistema de venta de tarjetas de subte. El mismo contempla dos tipos de tarjeta:

- *Por cantidad fija de viajes*: se adquiere con cierta cantidad de viajes, y una vez consumida dicha cantidad se desecha. Esta tarjeta no es afectada por modificaciones en el precio del pasaje.
- *Recargable*: es una tarjeta que almacena un saldo. Se adquiere inicialmente sin saldo, y se la puede recargar en cualquier momento. Al utilizarla para realizar un viaje se descuenta del saldo el precio actual del pasaje.

Implementar todo lo necesario para que el siguiente código de ejemplo imprima la salida indicada:

```
public static void main(String[] args) {

	SubwayCentral central = new SubwayCentral(1.1);			// El precio del pasaje es $1.10

	FixedSubwayCard card1 = central.buyFixedSubwayCard(3);	// Se compran 3 viajes

	try {
		card1.register("Bulnes");				// Registra un viaje desde Bulnes
		card1.register("9 de julio");
		central.changePrice(2.5);				// El precio del pasaje pasa a $2.50
		card1.register("Bulnes");
		card1.printTransactions();
		card1.register("9 de julio");
	} catch (RideDeniedException e) {
		System.out.println("No hay saldo suficiente.");
	}

	central.changePrice(1.1);

	RechargeableSubwayCard card2 = central.buyRechargeableSubwayCard();  // Tarjeta recargable

	card2.charge(4.0);		// Se le cargan $4 a la tarjeta

	try {
		card2.register("Carlos Gardel");
		central.changePrice(2.5);
		card2.register("Alem");
		card2.printTransactions();
		card2.register("Catedral");
	} catch (RideDeniedException e) {
		System.out.println("No hay saldo suficiente.");
	}

	card2.charge(3.0);
	try {
		card2.register("Catedral");
		card2.printTransactions();
		card2.register("Carranza");
	} catch (RideDeniedException e) {
		System.out.println("No hay saldo suficiente.");
	}
}
```

```
Viaje desde estacion Bulnes ($1.1)
Viaje desde estacion 9 de julio ($1.1)
Viaje desde estacion Bulnes ($1.1)
No hay saldo suficiente.
Recarga ($-4.0)
Viaje desde estacion Carlos Gardel ($1.1)
Viaje desde estacion Alem ($2.5)
No hay saldo suficiente.
Recarga ($-4.0)
Viaje desde estacion Carlos Gardel ($1.1)
Viaje desde estacion Alem ($2.5)
Recarga ($-3.0)
Viaje desde estacion Catedral ($2.5)
No hay saldo suficiente.
```

Ejercicio 3

Se tiene la siguiente interfaz junto con una clase que la implementa:

```
public interface SimpleList<T> extends Iterable<T> {
	/** Agrega un elemento al final de la lista. */
	public void append(T elem);
}
```

```
public class SimpleArrayList<T> extends ArrayList<T> implements SimpleList<T> {
	public void append(T elem) {
		add(elem);
	}
}
```

Se quiere dar la posibilidad al usuario de concatenar múltiples **SimpleList**. Implementar todo lo necesario para que el siguiente código de ejemplo compile e imprima lo que se indica en los comentarios. **No es posible modificar el código de SimpleList ni de SimpleArrayList.**

```
public class Test {
	public static void main(String[] args) {

		SimpleList<String> list1 = new SimpleArrayList<String>();
		SimpleList<String> list2 = new SimpleArrayList<String>();
		SimpleList<String> list3 = new SimpleArrayList<String>();

		list1.append("A"); list1.append("B"); list1.append("C");
		list2.append("D"); list2.append("E");
		list3.append("1"); list3.append("2");list3.append("3"); list3.append("4");

		SimpleList<String> list4 = new ConcatNSimpleList<String>();

		((ConcatNSimpleList<String>)list4).add(list1).add(list2).add(list3);

		show(list4);		/* La salida es: ABCDE1234 */

		list1.append("H");
		list2.append("I");

		show(list4);		/* La salida  es: ABCHDEI1234 */

		Iterator<String> it = list4.iterator();
		while (it.hasNext()) {
			String s = it.next();
			if (s.equals("B") || s.equals("D")) {
				it.remove();
			}
		}

		list4.append("J");

		show(list1);		/* La salida es: ACH */
		show(list2);		/* La salida es: EI */
		show(list3);		/* La salida es: 1234J */
	}

	private static void show(SimpleList<String> list) {
		for (String s : list) {
			System.out.print(s);
		}
		System.out.println();
	}
}
```