Ejercicios Excepciones

Semana del 23 de Marzo

1) Modifica clase `SemiGroup` que hiciste en los ejercicios de interfaces para añadir control de errores mediante excepciones. Debes diseñar una jerarquía de excepciones que controle las siguientes situaciones:

- El método `calculate` debe señalar dos tipos de errores: que los parámetros no pertenezcan al conjunto, o que el resultado no pertenezca al conjunto.
- El constructor de `SemiGroup` debe lanzar una excepción si operar con alguna combinación de sus elementos resulta en un valor que no pertenece al conjunto.

Tus excepciones deben asegurar que el siguiente programa obtenga la salida de más abajo

```
package semiGroup;
import java.util.Arrays;
import semiGroup.exceptions.*;

public class SemiGroupMain {
  public static void main(String[] args) {
      try {
            SemiGroup sg = new SemiGroup(new AddModulo(4), Arrays.asList(0, 1, 2));
      }
      catch (IllFormedSemiGroupException ex) {
            System.err.println("Malformed!: "+ex);
      }
      catch (SemiGroupException ex) {
            System.err.println(ex);
      }

      try {
            SemiGroup sg = new SemiGroup(new AddModulo(3), Arrays.asList(0, 1, 2));
            sg.calculate(0, 3);
      } catch (SemiGroupException ex) {
            ex.printStackTrace();
      }
  }
}
```

Salida esperada:

```
Malformed!: Ill-formed semigroup: operating 1 and 2 yields 3 which is not an element of
the semigroup
Parameter 3 is not an element of the semigroup
      at semiGroup.SemiGroup.calculate(SemiGroup.java:32)
        at semiGroup.SemiGroupMain.main(SemiGroupMain.java:20)
```

Posible mejoras: Podemos hacer un semi-grupo de cualquier tipo de datos, no necesariamente enteros. Para ello, usaremos clases e interfaces genéricas en la próxima sección.

2) Queremos crear una clase `PasswordChecker` que compruebe la validez de una clave respecto a una serie configurable de políticas, como tener una longitud mínima, contener dígitos, tener caracteres en minúscuas y mayúsculas, etc.

Para ello, cada política se implementa como una clase conforme a la siguiente interfaz:

```
public interface IPasswordPolicy {
	void check(String password) throws InvalidPasswordException;
}
```

Cada política debe implementar el método `check` de manera que compruebe la condición necesaria, y lanzar una subclase de `InvalidPasswordException`, si no se cumple.

Crea las clases necesarias, para que el siguiente programa devuelva la salida de más abajo.

```
import password.exceptions.InvalidPasswordException;
import password.policies.*;

public class PasswordMain {
  public static void main(String[] args) {
	// las claves deben tener al menos longitud 5, y algún dígito
	PasswordChecker pc = new PasswordChecker(new MinimumLengthPolicy(5),
                                                 new HasDigitPolicy());
	try {
		pc.check("admin");
	} catch (InvalidPasswordException ex) {
		System.err.println(ex);
	}
  }
}
```

Salida esperada:

```
Invalid password: admin, reason:
Has no digits
```

Mejoras: Crear una clase, tal como `MinimumLengthPolicy` o `HasDigitPolicy`, sólo con el objetivo de implementar un método requiere mucha "ceremonia", más si no se va a usar en ningún otro sitio. Más adelante, veremos que las expresiones lambda evitan esta necesidad, siendo una notación mucho más concisa.