Ejercicios: Clases internas, anónimas y reflexión

Semana del 13 de Abril

1) **Clases anónimas/internas**: Crea una clase `Store`, que puede almacenar de manera secuencial, y permitiendo repeticiones elementos de cualquier tipo. La clase `Store` debe ser compatible con `List`, y debe admitir un filtro (objetos compatibles con la interfaz `IFilter`) que deben pasar los elementos que se desan añadir.

Se pide: el código de la clase `Store`, y cambiar el siguiente programa para que, en vez de usar una clase interna estática, use una clase anónima.

```
interface IFilter<T> { boolean check(T elem); }

public class AnonymousClass {
	static class FilterEven implements IFilter<Integer> {
		@Override public boolean check(Integer elem) { return elem%2==0; }
	}

	public static void main(String[] args) {
		Store<Integer> store = new Store<>(new FilterEven()); // change to anonymous
		store.add(1);
		store.add(2);
		System.out.println(store);
	}
}
```

Salida esperada:

```
[2]
```

Solución:

```
public class Store<T> extends ArrayList<T>{
	private IFilter<? super T> filter;

	public Store(IFilter<? super T> filter) {
		this.filter = filter;
	}
	@Override public boolean add(T i) {
		if (this.filter.check(i)) return super.add(i);
		else return false;
	}
}

public static void main(String[] args) {
		Store<Integer> store = new Store<Integer>(new IFilter<Integer>() {
			@Override public boolean check(Integer elem) {
				return elem%2==0;
			}
		}); // change to anonymous
		//…
Nota: Podemos usar también una expresión lambda en vez de una clase anónima:
Store<Integer> store = new Store<Integer>(elem -> elem%2==0);
```

2) **Reflexión/instanceof:** Frecuentemente, el uso del operador `instanceof` indica un error en el diseño. ¿Sucede eso en el siguiente programa? Reorganiza su código para eliminar ese operador y mejorar la extensibilidad del código.

```
abstract class BankCard {
	protected String number;
	protected double amount;

	public BankCard(String number, double amount) {
		this.number = number; this.amount = amount;
	}
	public void withdraw(double qty) { this.amount -= qty;} // error control omited
}
class CreditCard extends BankCard {
	public CreditCard(String number, long amount) { super(number, amount); }
}
class DebitCard extends BankCard {
	public DebitCard(String number, long amount) { super(number, amount); }
}

public class Banking {
  public void charge(BankCard[] cards, int qty) {
	for (BankCard bc : cards) {
	  if (bc instanceof CreditCard) bc.withdraw(0.1*qty+qty);  // get rid of this!
	  else if (bc instanceof DebitCard) {                      // get rid of this!
		if (qty>100) bc.withdraw(qty);
		else bc.withdraw(qty+1);
	  }
	}
  }
/*…*/}
```

Solución: Efectivamente, el código anterior muestra un diseño muy malo, ya que dificulta la extensibilidad y hace el código innecesariamente complicado. La lógica del cargo a las tarjetas no está bien situada en Banking.charge, y ha de moverse a las clases tarjetas. De esta manera, creamos un método abstracto charge en BankCard y lo sobreescribimos en las dos clases hijas:

```
abstract class BankCard {
	protected String number;
	protected double amount;

	public BankCard(String number, double amount) {
		this.number = number; this.amount = amount;
	}
	public void withdraw(double qty) { this.amount -= qty;} // error control omited
	public abstract void charge(int qty);
}
class CreditCard extends BankCard {
	public CreditCard(String number, long amount) { super(number, amount); }
	@Override public void charge(int qty) { this.withdraw(0.1*qty+qty); }
}
class DebitCard extends BankCard {
	public DebitCard(String number, long amount) { super(number, amount); }

	@Override public void charge(int qty) {
		if (qty>100) this.withdraw(qty);
		else this.withdraw(qty+1);
	}
}

public class Banking {
  public void charge(BankCard[] cards, int qty) {
	for (BankCard bc : cards)
	  bc.charge(qty);
  }
}
```

3) **Clases internas:** Las clases internas se utilizan a veces para crear clases auxiliaries, que no son de interés para otra clase más que la clase contenedora. Así, en este ejercicio queremos crear una clase `Book`, que internamente contenga objetos de tipo `Chapter`, pero no queremos exponer esa clase al resto del programa.

Se pide: usando clases internas, completar el siguiente programa para que dé la salida de más abajo.

```
public class Books {
  public static void main(String... args) {
       Book b = new Book("The Book of why", "Judea Pearl");
       b.addChapter("Introduction", 22).  // internally creates a Chapter object
         addChapter("The ladder of causation", 31).
         addChapter("Paradoxes Galore", 30);
       System.out.println(b);
  }
}
```

Salida esperada:

```
'The Book of why' by Judea Pearl, with chapters:
[Introduction (22 pp), The ladder of causation (31 pp), Paradoxes Galore (30 pp)]
```

Solución:

```
public class Book {
	private String title;
	private String author;
	private List<Chapter> chapters = new ArrayList<>();

	public Book(String title, String author) {
		this.title = title;
		this.author = author;
	}

	public Book addChapter(String chapTitle, int np) {
		new Chapter(chapTitle, np);
		return this;
	}

	public String toString() {
		return "'"+this.title+"' by "+this.author+", with chapters:\n"+this.chapters;
	}

	private class Chapter {
		private String title;
		private int pages;

		public Chapter(String chapTitle, int np) {
			this.title = chapTitle;
			this.pages = np;
			chapters.add(this);
		}

		@Override public String toString() {
			return this.title+" ("+this.pages+" pp)";
		}
	}
}
```