Capítulo 15

Pilas

15.1 Tipos de datos abstractos

Los tipos de datos que hemos mirado hasta ahora son todos concretos, en el sentido de que hemos especificado completamente cómo se implementan. Por ejemplo, la clase `Carta` representa una carta usando dos enteros. Como discutimos en su momento, esa no es la única forma de representar una carta: hay muchas implementaciones alternativas.

Un **tipo de dato abstracto**, o TAD, especifica un conjunto de operaciones (o métodos) y la semántica de las operaciones (que es lo que hacen) pero no especifica la implementación de las mismas. Eso es lo que las hace abstractas.

¿Por qué es útil?

- Simplifica la tarea de especificar un algoritmo si podemos denotar las operaciones que necesitamos sin tener que pensar al mismo tiempo cómo esas operaciones se realizan.
- Ya que hay usualmente muchas maneras distintas de implementar un TAD, podría ser útil escribir un algoritmo que pueda ser usado con cualquiera de las posibles implementaciones.
- Existen TADs comúnmente reconocidos, como el TAD `Pila` en este capítulo, que a menudo son implementados en bibliotecas estándar de manera que sean escritos una sola vez y luego sean reutilizados por muchos programadores.
- Las operaciones en TADs proveen un lenguaje de alto nivel para especificar y hablar de algoritmos.

Cuando hablamos de TADs, a menudo distinguimos el código que usa al TAD, llamado el **código cliente**, del código que implementa al TAD, llamado el **código proveedor**, ya que provee un conjunto de servicios estándar.

15.2 El TAD Pila

En este capítulo vamos a analizar un TAD muy común, el de las pilas. Una pila es una colección, lo cual significa que es una estructura de datos que contiene múltiples elementos. Otras colecciones que hemos visto son los arreglos y las listas.

Como dijimos antes, un TAD se define por un conjunto de operaciones. Las pilas pueden realizar las siguientes operaciones:

constructor: Crea una nueva pila vacía.

apilar: Agregar un nuevo elemento al final de la pila.

desapilar: Quitar y devolver un ítem de la pila. El ítem devuelto es siempre el último que fue agregado.

estaVacia: Responder si la pila está vacía.

Una pila es a veces llamada una estructura LIFO, del inglés "last in, first out,", es decir "último en entrar, primero en salir", porque el último ítem agregado es el primero en ser removido.

15.3 El objeto Stack de Java

Java provee un tipo de objeto preincorporado llamado `Stack` [1] que implementa el TAD Pila. Deberías hacer un esfuerzo por mantener esas dos cosas—el TAD y la implementación Java—en su lugar. Antes de usar la clase `Stack`, tenemos que importarla desde `java.util`.

Las operaciones del TAD en la clase `Stack` de Java tienen los siguientes nombres:

apilar: `push`

desapilar: `pop`

estaVacia: `isEmpty`

Entonces la sintaxis para construir una nueva `Stack` es

1. N.d.T.: Stack: Pila en inglés.

```
Stack pila = new Stack ();
```

Inicialmente la pila está vacía, como podemos confirmar con el método `isEmpty`, que devuelve un `boolean`:

```
System.out.println (pila.isEmpty ());
```

Una pila es una estructura de datos genérica, lo cual significa que podemos agregar cualquier tipo de ítems a ella. En la implementación de Java, sin embargo, sólo podemos agregar objetos y no valores de tipos nativos.

Para nuestro primer ejemplo, vamos a usar objetos de tipo `Node`, como definimos en el capítulo anterior. Empecemos creando e imprimiendo una lista corta.

```
ListaInt lista = new ListaInt();
lista.agregarAdelante (3);
lista.agregarAdelante (2);
lista.agregarAdelante (1);
lista.imprimir ();
```

La salida es `(1, 2, 3)`. Para poner un objeto de tipo `Nodo` en la pila, usamos el método `push`:

```
pila.push (lista.cabeza);
```

El siguiente ciclo recorre la lista y apila todos los nodos en la pila:

```
for (Nodo nodo = lista.cabeza; nodo != null; nodo = nodo.prox) {
    pila.push (nodo);
}
```

Podemos remover un elemento de la pila con el método `pop`.

```
Object obj = pila.pop ();
```

El tipo de retorno de `pop` es `Object`! Esto es porque la implementación de la pila no sabe exactamente de qué tipo son los objetos que contiene. Cuando apilamos los objetos de tipo `Nodo`, son convertidos automáticamente en `Objects`. Cuando los obtenemos de nuevo desde el stack tenemos que castearlos de nuevo a `Nodo`.

```
Nodo nodo = (Nodo) obj;
System.out.println (nodo);
```

Desafortunadamente el manejo cae en el programador, que debe llevar la cuenta de los objetos en la pila y castearlos al tipo original cuando son removidos. Si se trata de castear un objeto a un tipo incorrecto se obtiene una `ClassCastException`.

El siguiente ciclo es la forma usual de recorrer la pila, desapilando todos los elementos y frenando cuando queda vacía:

```
while (!pila.isEmpty ()) {
    Node nodo = (Nodo) pila.pop ();
    System.out.print (nodo + " ");
}
```

La salida es `3 2 1`. En otras palabras, acabamos de usar una pila para imprimir los elementos de una lista de atrás para adelante! Por supuesto, este no es el formato estándar para imprimir una lista, pero usando una pila fue realmente fácil de hacer.

Podrías comparar este código con las implementaciones del capítulo anterior de `imprimirInverso`. Hay un paralelismo natural entre la versión recursiva de `imprimirInverso` y el algoritmo que usa una pila aquí. La diferencia es que `imprimirInverso` usa el stack de tiempo de ejecución para llevar la cuenta de los nodos mientras recorre la lista, y después los imprime a la vuelta de la recursión. El algoritmo que usa la pila hace lo mismo, pero usando un objeto de tipo `Stack` en vez de la pila de tiempo de ejecución.

15.4 Clases adaptadoras

Para cada tipo primitivo en Java, hay un tipo de objeto preincorporado llamado una **clase adaptadora**[2]. Por ejemplo, la clase adaptadora de `int` se llama `Integer`; la de `double` se llama `Double`.

Las clases adaptadoras son útiles por varias razones:

- Es posible instanciar una clase adaptadora y crear objetos que contengan valores primitivos. En otras palabras, se puede adaptar un valor primitivo a un objeto, lo cual es útil si se quiere llamar a un método que requiere algo de tipo objeto.
- Cada clase adaptadora contiene valores especiales (como el mínimo y el máximo valor para ese tipo), y métodos que son útiles para convertir entre tipos.

2. N.d.T.: Del inglés Wrapper class.