15.11 Implementando TADs

Uno de los objetivos fundamentales de un TAD es el de separar los intereses del proveedor, que escribe el código que implementa el TAD, del cliente, que es quien lo usa. El proveedor sólo tiene que preocuparse de que la implementación sea correcta—de acuerdo con la especificación del TAD— y no de cómo será usado.

Por otro lado, el cliente *asume* que la implementación del TAD es correcta y no se preocupa por los detalles. Cuando estamos usando una de las clases preincorporadas de Java, podemos darnos el lujo de pensar exclusivamente como clientes.

Cuando implementamos un TAD, en cambio, debemos escribir código cliente para probarlo. En este caso, a veces hace falta pensar cuidadosamente qué rol estamos jugando en un instante dado.

En las siguientes secciones vamos a cambiar de rol e investigar una manera de implementar el TAD Pila usando un arreglo. Es hora de empezar a pensar como un proveedor.

15.12 Implementación del TAD Pila usando arreglos

Las variables de instancia para esta implementación son un arreglo de `Objects`, que contendrá los elementos de la pila, y un índice entero que llevará la cuenta de cuál es el siguiente espacio disponible en el arreglo. Inicialmente, el arreglo está vacío y el índice es `0`.

Para agregar un elemento a la pila (`push`), vamos a copiar una referencia a él en la pila e incrementar el índice. Para quitar un elemento (`pop`) tenemos que decrementar el índice primero y después copiar el elemento afuera.

Esta es la definición de la clase:

```
public class Stack {
    Object[] arreglo;
    int indice;

    public Stack () {
        this.arreglo = new Object[128];
        this.indice = 0;
    }
}
```

Como de costumbre, una vez que elegimos las variables de instancia, es un proceso mecánico el de escribir un constructor. Por ahora, el tamaño por defecto es de 128 ítems. Después vamos a considerar mejores formas de manejar esto.

Chequear si la pila está vacía es trivial:

```
public boolean isEmpty () {
    return indice == 0;
}
```

Es importante recordar, sin embargo, que el número de elementos en la pila no es el mismo que el tamaño del arreglo. Inicialmente el tamaño es 128, pero el número de elementos es 0.

Las implementaciones de push y pop surgen naturalmente de la especificación.

```
public void push (Object elem) {
    arreglo[indice] = elem;
    indice++;
}

public Object pop () {
    indice--;
    return arreglo[indice];
}
```

Para probar estos métodos, podemos sacar ventaja del código cliente que usamos en el ejercicio de la `Stack` preincorporada. Todo lo que tenemos que hacer es comentar la línea `import java.util.Stack`. Después, en vez de usar la implementación de la pila de `java.util` el programa va a usar la implementación que acabamos de escribir.

Si todo va de acuerdo al plan, el programa debería funcionar sin ningún cambio adicional. Otra vez, una de las fortalezas de usar un TAD es que es posible cambiar las implementaciones sin cambiar el código cliente.

15.13 Redimensionando el arreglo

Una debilidad de esta implementación es que elige un tamaño arbitrario para el arreglo cuando la `Stack` es creada. Si el usuario apila más de 128 ítems, causará una excepción de tipo `ArrayIndexOutOfBounds`.

Una alternativa es dejar que el código cliente especifique el tamaño del arreglo. Esto alivia el problema, pero requiere que el cliente sepa de antemano cuántos ítems va a necesitar, y eso no siempre es posible.

Una solución mejor es verificar si el arreglo está lleno y agrandarlo si es necesario. Ya que no tenemos idea de qué tan grande tiene que ser el arreglo, una estrategia razonable es empezar con un tamaño pequeño y

duplicarlo cada vez que se sobrepasa. Acá tenemos una versión mejorada de push:

```
public void push (Object item) {
   if (lleno ()) redimensionar ();

   // en este punto podemos probar
   // que indice < arreglo.length

   arreglo[indice] = item;
   indice++;
}
```

Antes de poner un nuevo ítem en el arreglo, debemos chequear si el arreglo está lleno. En ese caso, llamamos a `redimensionar`. Despues del `if`, sabemos que o bien (1) ya había tamaño en el arreglo, o (2) el arreglo fue redimensionado y ahora sí hay espacio. Si `lleno` y `redimensionar` son correctos, entonces podemos probar que `indice < arreglo.length`, y de esta manera, la siguiente sentencia no causará ninguna excepción. Ahora, todo lo que tenemos que hacer es implementar `lleno` y `redimensionar`.

```
private boolean lleno () {
   return indice == arreglo.length;
}

private void redimensionar () {
   Object[] nuevoArreglo = new Object[arreglo.length * 2];

   // asumimos que el arreglo anterior estaba lleno
   for (int i=0; i<arreglo.length; i++) {
      nuevoArreglo[i] = arreglo[i];
   }
   arreglo = nuevoArreglo;
 }
```

Ambos métodos son declarados `private`[4], lo cual significa que no pueden ser llamados desde otra clase, sólo desde esta. Esto es aceptable, ya que no hay razón para el código cliente de usar estas funciones, y a la vez es deseable, ya que refuerza la barrera entre el código proveedor y el cliente.

La implementación de `lleno` es trivial; simplemente chequea que si el índice ha pasado más allá del rango de índices válidos.

4. N.d.T.: Privados.

La implementación de `redimensionar` es directa, con la salvedad de que asume que el arreglo viejo está lleno. En otras palabras, esta presunción es una precondición de este método. Es simple ver que esta precondición es satisfecha, ya que la única manera de llamar a `redimensionar` es si `lleno` dio verdadero, lo cual sólo puede suceder si `indice` es igual a `arreglo.length`.

Al final de `redimensionar`, reemplazamos el arreglo viejo con el nuevo (causando que el viejo sea reclamado por el recolector de basura). La nueva `arreglo.length` es el doble de grande que la anterior, e `indice` no ha cambiado, por lo que ahora debe ser cierto que `index < array.length`. Esta aseveración es una **postcondición** de `redimensionar`: algo que debe ser cierto cuando este método está completo (siempre y cuando las precondiciones hayan sido satisfechas).

Precondiciones, postcondiciones e invariantes son herramientas útiles para analizar programas y demostrar su corrección. En este ejemplo hemos demostrado un estilo de programación que facilita el análisis de programas y un estilo de documentación que ayuda a demostrar corrección.

15.14 Glosario

tipo de dato abstracto (TAD): Tipo de dato (usualmente una colección de objetos) que está definido por un conjunto de operaciones, pero que puede ser implementado en una variedad de formas.

cliente: Programa que usa un TAD (o persona que escribió el programa).

proveedor: Código que implementa un TAD (o persona que lo escribió).

clase adaptadora: Una de las clases de Java, como `Double` e `Integer` que provee objetos para contener tipos primitivos y métodos que operan en ellos.

`private`**:** Palabra clave de Java que indica que un método o variable de instancia no puede ser accedido desde el exterior de la definición de la clase misma.

notación infija: Forma de escribir expresiones matemáticas con los operadores entre los operandos.

notación polaca: Forma de escribir expresiones matemáticas con los operadores después de los operandos.

parsear: Leer una cadena de caracteres o tokens y analizar su estructura gramatical.