Sesión 6

Uso de pilas y árboles

En nuestros programas manejaremos un tipo particular de clases predefinidas: aquellas que contienen la traducción a Java de los tipos abstractos de datos presentados en la teoría: pilas y árboles binarios. Dichas clases forman parte del paquete `tadsPM`. De ellas sólo conoceremos algunos detalles, como su nombre y las cabeceras de sus métodos públicos. Todo ello consta en los respectivos ficheros de documentación `.doc`.

6.1 Restricciones para el uso de los TADs en Java

Los TADs implementados en el paquete `tadsPM` se basan en las especificaciones de los módulos correspondientes mostrados en teoría. Ahora bien, debido a las particularidades de la orientación a objetos y del lenguaje Java, vistos en una sesión anterior, y a la genericidad, se han tomado algunas decisiones de implementación que influyen en su uso.

- En primer lugar, notad que al igual que en los ejemplos de sesiones anteriores, se han modificado las **cabeceras** de las operaciones adaptándolas a los convenios de Java, de forma que toda operación se aplica sobre un parámetro implícito, con una sintaxis de la forma

  ```
  <objeto>.<método>(<otros parámetros>)
  ```

  Por ejemplo, la instrucción $p :=$ apilar(x, p) de la notación algorítmica se traduce como `p.apilar(x)`.

- Para cada instrucción que pueda modificar un objeto de manera no deseada (típicamente, las llamadas a métodos), hay que plantearse la conveniencia de hacer una **copia** del objeto involucrado. Por ejemplo, si se desea algo equivalente a $q := \text{apilar}(x, p)$ sin modificar p, habrá que copiar p en q y después aplicar `q.apilar(x)`. Si nosotros mismos implementamos el método invocado, siempre podemos realizar la copia dentro del código del mismo.

  Del mismo modo, evitaremos las asignaciones $p := q$ entre objetos (obviamente, no entre variables y expresiones de tipos simples) y en su lugar aplicaremos copias.

- Si usamos pilas de tipos diferentes habrá que escribir separadamente los correspondientes **métodos de lectura y escritura**: métodos para leer o escribir pilas de enteros, otros para las pilas de pares de enteros, otros para las pilas de `Estudiante`, etc. Podéis estudiar el ejemplo de pilas de enteros en la clase `PilaIOint`.

- Se ha mantenido la **genericidad** de los tipos: en un objeto de la clase `Pila` o `Arbol` pueden almacenarse directamente objetos de cualquier clase. Al consultar un elemento de la estructura se ha de aplicar una **coerción**, es decir, hay que convertir la cima o raíz correspondientes al tipo al que realmente pertenecen. Por ejemplo, para obtener la cima de una pila de estudiantes habría que hacer

  ```
  Estudiante e = (Estudiante)p.cima()
  ```

  Notad que esta asignación entre objetos puede dar lugar a efectos secundarios si se modifica el estudiante `e`. Si no queremos que ello ocurra deberemos trabajar sobre una copia del mismo.

  ```
  e.copiar_estudiante((Estudiante)p.cima())
  ```

  Con los tipos simples hay que aplicar una envoltura antes de almacenar cada valor. Puede usarse una envoltura estándar, como `INTEGER` o `BOOLEAN` o una definida por nosotros, mediante una clase auxiliar. Por otro lado, al consultar un elemento, después de la coerción hay que deshacer la envoltura. Como ejemplo, ved los métodos `leer_pila_int` y `escribir_pila_int` en `PilaIOint`.

6.2 Uso de la clase `Pila`

El fichero `pertpila.java` contiene un programa que comprueba si un cierto número está en una pila de enteros. El programa usa una función estática que realiza la búsqueda. Por su parte, el método `main` se encarga de la entrada/salida y de invocar correctamente la función anterior. Dicha función se basa en el siguiente algoritmo.

funcion f_pert $(p : \text{pila}; x : \text{natural})$ **retorna** b : booleano
  {Pre: cierto}
  si es_nula(p) $\longrightarrow$ $b := \text{falso}$
  [] $\neg$es_nula(p) $\longrightarrow$
    $b :=$ f_pert (desapilar$(p), x$);
    {HI: b indica si x está en desapilar(p)}
    $b := b \vee x = \text{cima}(p)$
  fsi;
  {Decr: alt(p)}
  {Post: b indica si x está en p}

6.2.1 Ejercicio: Búsqueda en una pila de pares de enteros

Repetid el programa anterior considerando una pila de pares de enteros. Dado un número, comprobad si aparece como primer elemento de algún par de la pila y, en caso de éxito, escribid el par completo. Definid previamente una clase `parint` (“par de enteros”), como envoltura para apilar los números, y otra `PilaIOparint` para leer y escribir las pilas de `parint`.

6.2.2 Ejercicio: Búsqueda en una pila de estudiantes

Repetid el programa anterior con una pila de estudiantes. Dado un entero, buscad si hay algun estudiante en la pila que lo tenga como DNI. En caso de éxito hay que informar sobre la nota del estudiante. Necesitaréis un módulo `PilaIOEstudiante`.

6.2.3 Ejercicio: sumar un cierto valor a cada elemento de una pila

1. Con la misma estructura del ejemplo anterior, escribid un programa que dada una pila de naturales le sume un número k a todos sus elementos.

2. Repetid el ejercicio, pero usando esta vez pilas de pares de enteros y sumando k al segundo elemento de cada par.

3. Por último, aplicad las ideas de los problemas anteriores para redondear las notas de una pila de estudiantes.

Tomad como base el siguiente algoritmo. Podéis traducirlo como acción o función. En el segundo caso, poned especial cuidado en lograr que las pilas originales no se vean modificadas.

funcion f_incrementar $(p$: pila; k : natural$)$ **retorna** q : pila
  {Pre: cierto}
  si es_nula(p) $\longrightarrow$ $q :=$ p_nula
  [] $\neg$es_nula(p) $\longrightarrow$ $x :=$ cima$(p) + k$;
    $q :=$ f_incrementar(desapilar$(p), k$);
    {HI: q es la pila resultante de incrementar todo
    elemento de desapilar(p) en k unidades}
    $q :=$ apilar(x, q)
  fsi
  {Decr: alt(p)}
  {Post: q es la pila resultante de incrementar todo elemento de p en k unidades}

6.3 Uso de la clase `Arbol`

En el fichero `ArbolIOint` están definidos dos posibles métodos de lectura y escritura de árboles binarios de enteros, que a su vez utilizan la clase `Arbol` y sus métodos. Para probarlos, escribid programas que calculen operaciones sencillas sobre árboles como la altura, el tamaño u obtener el árbol resultante de sumar un valor k a todos los nodos de un árbol dado.

Por ejemplo, comenzad probando la siguiente función, situada en el fichero `tamarb.java` que calcula el número de elementos de un árbol:

funcion f_tam (a : arbol) **retorna** n : nat
  {Pre: cierto}
  si es_nulo(a) $\longrightarrow$ $n := 0$
  [] $\neg$es_nulo(a) $\longrightarrow$ $n_1 :=$ f_tam (hi(a));
    $n :=$ f_tam (hd(a));
    {HI: $n_1 =$ tam (hi(a)) $\wedge$ $n =$ tam (hd(a))}
    $n := 1 + n_1 + n$
  fsi
  {Decr: tam(a)}
  {Post: $n =$ tam (a)}

Resolved también algún problema con árboles de `parint` y de estudiantes.