Presentación

En estas sesiones supondremos que el alumno está familiarizado con los conceptos básicos del lenguaje Java y posee un mínimo dominio de los sistemas operativos Unix y Linux (editores, gestión de ficheros, etc). No obstante, cuando sea preciso presentaremos los conceptos y herramientas nuevos que consideremos necesarios para las sesiones de laboratorio y la práctica de la asignatura.

La documentación básica para el uso del sistema se encuentra en la página

`http://www.fib.upc.es/LCFIB`

a partir del recuadro de "Suport i documentació", siguiendo por "Preguntes més freqüents" hasta "Linux". Por otra parte, también es útil la "Guía dels sistemes informàtics del LCFIB (díptic)".

La documentación necesaria para las sesiones de laboratorio, incluyendo una copia de este manual en formato `PDF` se encuentra en el siguiente directorio público

`/home/soft/assig/pm`

Por último, toda la información nueva que vaya apareciendo sobre la práctica, convocatorias de exámenes, etc. se encontrará en la página de la asignatura

`http://www.lsi.upc.es/~albert/promet.html`

Índice

Capítulo 1

Primera sesión

Para empezar, copiad todo el subdirectorio `sesion1` con

```
cp -r /home/soft/assig/pm/sesion1 sesion1
```

1.1 Estructura de un programa en Java

La unidad básica de construcción de programas en Java es la *clase*. En un programa codificado en Java pueden aparecer una o varias clases, que podemos clasificar en los siguientes tipos:

- las clases estándar del lenguaje
- las clases definidas por el propio programador
- las clases definidas por otros programadores

Se supone que cada programa tiene una clase principal que es la que se ejecuta en primera instancia. Ésta puede utilizar otras y así sucesivamente.

El uso de una clase por parte de otra implica la importación, implícita o explícita, de la misma y sus métodos. Discutiremos la diferencia entre ambas en la próxima sesión.

1.2 Visibilidad de clases

Para poder utilizar una clase, ésta ha de ser *visible* por el programa que la importa. Esencialmente, una clase es visible desde el directorio de trabajo si ha sido declarada pública y si su correspondiente fichero `.class` cumple alguna de estas propiedades:

- está en el propio directorio de trabajo (en realidad, toda clase del directorio de trabajo es pública, salvo si se ha declarado explícitamente como privada)
- es del lenguaje, como la clase `Object`
- está en cualquiera de los directorios que aparecen en la definición de la varible de entorno `CLASSPATH`

En esta primera sesión trabajaremos sólo con clases instaladas en el directorio de trabajo, creadas por vosotros o copiadas del directorio de la asignatura y definidas como públicas en el código correspondiente.

1.3 Compilación

Para poder compilar con éxito un programa contenido en un fichero `.java`, las clases que éste importa han de ser visibles según la definición anterior. Si esto se cumple, para compilar un programa como `PM1` basta con hacer

```
javac PM1.java
```

y se generará el correspondiente `PM1.class`.

Si dentro del programa se definen clases públicas, se generarán también ficheros `*.class` para todas ellas. Si el programa usa clases definidas en otros `*.java` instalados en el directorio de trabajo, éstos se compilarán automáticamente, salvo que en su código haya una indicación expresa de lo contrario.

1.4 Ejecución de programas

Los programas que os pediremos codificar en PM no contendrán ningún elemento gráfico (ventanas, formularios, etc) para gestionar el flujo de información. Sólo trabajaremos con el canal estándar de entrada/salida, en ocasiones redireccionado a ficheros de texto.

Por ello, no será necesario importar paquetes como `java.awt, java.applet`, etc. La entrada y salida de los programas se gestiona con los métodos de `System.in` y `System.out` y las ejecuciones se realizan siempre desde la línea de comandos.

Por ejemplo, si queremos ejecutar un programa, contenido en el fichero `PM1.class`, que recibe sus datos por teclado y escribe sus resultados en pantalla, haremos

```
java PM1
```

y a continuación escribimos los datos del programa respetando el formato esperado por éste. Tras la ejecución, veremos escrita la salida del mismo.

Si deseamos ejecutar el mismo programa sobre los datos almacenados en el fichero `PM1.dat`, pero queremos ver los resultados por pantalla redireccionaremos el canal estándar de entrada hacia ese fichero:

```
java PM1 < PM1.dat
```

Si además queremos que los resultados no se escriban en pantalla sino en otro fichero `PM1.sal` escribiremos

```
java PM1 < PM1.dat > PM1.sal
```

Éstos son los mecanismos de redirección de los canales estándar de entrada y salida en Unix y Linux. Puede usarse cualquier combinación de ellos cuando sea preciso.

1.5 Definición y uso de clases: la clase Estudiante

En ocasiones, la facultad manipula las notas de sus estudiantes en un formato bastante reducido. Por ejemplo, al final de cada cuatrimestre, se requiere que cada asignatura proporcione un fichero con tan sólo el DNI y la nota de cada estudiante. Para simular alguna de estas manipulaciones creamos una clase `Estudiante`.

Para cada estudiante, definido de la manera mencionada, necesitamos algunas operaciones primitivas, tales como la constructora `estudiante_vacio`, las modificadoras `ingresar_dni` e `ingresar_nota` y las consultoras `consultar_dni`, `consultar_nota` y `es_vacio`. Si en el futuro se necesitan más operaciones o campos, se pueden usar los diversos mecanismos que Java proporciona para extenderla.

Empleamos además otras dos clases, que sólo contienen métodos. En primer lugar, creamos la clase `EstudianteIO`, que define métodos estáticos para gestionar la lectura y escritura de los datos de un estudiante sin mezclar dicha gestión con las operaciones primitivas.

Esta clase usa la anterior (en particular, necesita los métodos en ella definidos para consultar o modificar los campos de un estudiante) y también usa la clase `LeerNumeros`. Esta segunda clase contiene métodos estáticos para leer números enteros y reales con coma flotante.

1.5.1 Ejemplo: Redondeo de la nota de una secuencia de estudiantes

Empleando las clases anteriores, vamos a escribir una clase cuyo método `main` permita sustituir la nota de una secuencia de estudiantes por su correspondiente redondeo al semientero mas próximo, mediante la fórmula:

```
nota_red = ((int) (2*(nota+.25f))) / 2.0f
```

(cuidado con el último `2.0f`: si usamos un entero la división no será exacta; la `f` es para que tome el 2 como `real` y no como `double`)

Realizaremos la sustitución mediante un **método estático** que aplica el redondeo a cada estudiante. El código resultante se encuentra en el fichero `redsec.java` del directorio de la sesión. En él se presentan dos métodos `redondear_e_a` y `redondear_e_f`, que se pueden usar indistintamente aunque se invocan de distinta manera.

Completad el fichero `redsec.dat` y usadlo como canal de entrada. Emplead otro fichero `redsec.sal` como canal de salida. Probad los métodos que redondean la nota para comprobar que funcionan igual.

1.6 Acciones y funciones

Una característica de la asignatura es que todas las variables que intervienen en las funciones que diseñamos han de aparecer en la cabecera de éstas o bien ser locales (no se permiten variables globales). Mantendremos esta línea al codificar nuestros algoritmos en Java.

Por otro lado, aunque en la asignatura solamente se diseñan funciones, en ocasiones puede interesar codificarlas como acciones (métodos que no retornan ningun resultado). Para ello empleamos parámetros de entrada/salida (parámetros cuyo valor queda modificado tras la ejecución del método). Con ellas también aplicaremos la norma del párrafo anterior.

Por ejemplo, el método estático `redondear_e_a` del fichero `redsec.java` es una acción porque modifica un campo del estudiante que se le pasa como parámetro. Tal modificación es posible porque disponemos de las operación modificadora `ingresar_nota` de la clase `Estudiante`. La misma situación se produce si la clase del parámetro modificado está definida en el mismo programa o si sus campos son públicos. Por último, también se pueden modificar posiciones de parámetros de tipo vector, pero no parámetros de tipos simples: para ello se necesita envolver éstos en una clase.

1.6.1 Ejercicio: Redondeo de la nota de un vector de estudiantes

Modificad el programa anterior, de forma que la sustitución se realice mediante un método estático que actúe sobre un vector de estudiantes. Elegid la dimensión del vector adecuadamente respecto al número de estudiantes que se usarán en la ejecución.

Codificad primero una función y comprobad si hay alguna mejora si después lo hacemos con una acción.

1.6.2 Ejercicio: Búsqueda lineal en un vector de estudiantes

Implementad el algoritmo de búsqueda lineal sobre un vector de estudiantes. Realizad una primera versión empleando una función que sólo retorne un booleano. ¿Qué ocurre si queremos que la función retorne dos resultados, un booleano y un índice? ¿Cómo cambia la situación si lo implementamos como acción?

Podéis basaros los siguientes algoritmos:

funcion busqueda_lin (v : vector; x : natural) **retorna** b : booleano
{Pre: cierto}
var i : natural **fvar**
$i := 0$;
$b :=$ falso;
{Inv: $b = \exists \alpha : 1 \leq \alpha \leq i : v[\alpha] = x \ \wedge \ i \leq N$}
{Cota: $N - i$}
mientras $i < N \ \wedge \ \neg b$ **hacer**
$b := v[i+1] = x$;
$i := i + 1$
fmientras;
{Post: $b = \exists \alpha : 1 \leq \alpha \leq N : v[\alpha] = x$}

funcion busqueda_lin_2 (v : vector; x : natural) **retorna** b : booleano; i : natural
{Pre: cierto}
$i := 0$;
$b :=$ falso;
{Inv: $b = \exists \alpha : 1 \leq \alpha \leq i : v[\alpha] = x \ \wedge \ i \leq N \ \wedge \ b \Rightarrow v[i] = x$}
{Cota: $N - i$}
mientras $i < N \ \wedge \ \neg b$ **hacer**
$b := v[i+1] = x$;
$i := i + 1$
fmientras;
{Post: $b = \exists \alpha : 1 \leq \alpha \leq N : v[\alpha] = x \ \wedge \ b \Rightarrow v[i] = x$}

Capítulo 2

Segunda sesión

Comencemos por copiar el subdirectorio `sesion2` con

```
cp -r /home/soft/assig/pm/sesion2 sesion2
```

2.1 TADs, clases y packages

En nuestros programas manejaremos un tipo particular de clases predefinidas: aquellas que contienen la representación de los tipos abstractos de datos presentados en la teoría (pilas, colas, árboles binarios, ...).

De dichas clases sólo conoceremos algunos detalles, como el nombre de las mismas y las cabeceras de sus métodos públicos. Todo ello constará en los respectivos ficheros de documentación `.doc`, que estarán instalados, junto con los correspondientes `.class`, en la zona pública que la asignatura tiene definida en el sistema.

Recordemos que una manera de hacer visible una clase alojada fuera del directorio de trabajo era utilizando la variable de entorno `CLASSPATH`.

Para poder trabajar con los TADs de la asignatura, definimos la variable `CLASSPATH` así

```
setenv CLASSPATH .:/home/soft/assig/pm/
```

Esta definición se convertirá en permanente si en lugar de teclearla cada vez, la instalamos en el fichero `.tcshrc`.

Un **package** es un grupo de clases públicas que forman una unidad lógica. Los packages se definen para facilitar el uso de sus clases; ello se realiza mediante la declaración `import`. Las clases de un package han de estar instaladas físicamente en un subdirectorio del directorio de trabajo o de uno que aparezca en la definición de `CLASSPATH`.

Con ello, las clases visibles por las nuevas clases que definamos son:

- las del sistema,
- las que están en el directorio de trabajo,
- las del directorio `/home/soft/assig/pm/`,
- las de los subdirectorios de ambos que hayan sido definidos como packages.

Si una clase es visible y ha sido declarada pública, puede usarse sin más que invocar su nombre. Lo mismo ocurre con sus métodos públicos, excepto los que sean estáticos, que han de ir precedidos del nombre de la clase. Por ejemplo, recordad las clases `Estudiante` y `EstudianteIO` de la sesión anterior y cómo las usábamos en el programa `redsec`.

2.1.1 Ejemplo: el package `tadsPM` y la clase `Pila`

En el subdirectorio `/home/soft/assig/pm/tadsPM` están los ficheros `.class` y `.doc` de las clases que implementan los tads de la asignatura junto con algunas clases auxiliares. Tal directorio ha sido definido como un package, por lo que para usar esas clases hay que comenzar nuestros programas con la declaración

```
import tadsPM.*;
```

También se pueden importar las clases una por una según nuestras necesidades:

```
import tadsPM.Pila; import tadsPM.EmptyStackException; ....
```

Pasemos a estudiar una clase concreta, por ejemplo la que representa al TAD *pila*. En el fichero `Pila.doc` tenemos toda la información necesaria sobre la clase `Pila` y sus métodos. Leedlo con atención.

Observamos que en un objeto de la clase `Pila` pueden almacenarse objetos de cualquier clase: por ello decimos que la clase `Pila` es genérica. La única excepción son los tipos simples: si se desean, por ejemplo, pilas de enteros antes de apilar hay que convertir cada número en un objeto de la clase envoltura `Integer` o a otra definida expresamente por nosotros. Por otro lado, si se quiere operar con la cima de una pila, lo primero que hay que hacer es obtener su valor numérico, es decir, deshacer la conversión anterior.

En el fichero `PilaIOint.java` están definidos dos métodos de lectura y escritura para las pilas de enteros, que a su vez utilizan la clase `Pila` y sus métodos.

2.1.2 Ejemplo: Búsqueda en una pila de enteros

Escribamos un programa `f_pert` que compruebe si un valor está en una pila de enteros. El programa se basará en una función estática que obtenga dicho resultado. Por su parte, el método `main` leerá una pila, la escribirá, leerá el elemento a buscar, invocará al método anterior para realizar la búsqueda, informará sobre el resultado de ésta y escribirá otra vez la pila original.

Notad que si usamos pilas de tipos diferentes habrá que escribir 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 booleanos, etc.

Trabajaremos a partir del siguiente código. Se supone que las operaciones ocultas *pert* y *alt* ya han sido especificadas. Notad que en este caso la traducción a función es directa y no hace falta codificar una acción. La solución se encuentra en el fichero `pertpila.java`.

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

2.1.3 Ejercicio: Búsqueda en una pila de pares de enteros

Repetid el programa anterior considerando una pila de pares de enteros y buscando el elemento dado en la segunda componente de los pares de la pila. Definid previamente una clase privada "par de enteros" dentro del mismo fichero.

2.2 Notas sobre el uso de las implementaciones de los TADs

Los TADs implementados en el paquete `tadsPM` se basan en las especificaciones algebraicas del libro "Programación Metódica" de J.L. Balcázar. Sin embargo, se han tomado algunas decisiones que influyen en su uso, debidas a restricciones del lenguaje Java y a la genericidad:

Restricciones debidas a la sintaxis de las clases en Java: Las cabeceras de las operaciones se han traducido al formato típico de los métodos Java para exportar, es decir, en lugar de emplear un estilo funcional con parametrización clásica, los métodos se aplican sobre un objeto de la clase correspondiente, distinguido por la notación `<objeto>.<método>(<otros parámetros>)`. Por ejemplo, la instrucción $p :=$ apilar(x, p) de la notación algorítmica pasa a ser `p.apilar(x)`.

Restricciones debidas a la implementación de los TADs: Dado que el usuario no conoce la implementación del TAD, ha de proteger a los datos de posibles efectos laterales y alias. 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 $q :=$ apilar(x, p) sin modificar p, habrá que copiar p en q y después aplicar `q.apilar(x)`.

En los programas anteriores hay más ejemplos: notad que el método `escribir_pila_int` consulta y desapila los elementos de `p`, que iría vaciándose; por ello hay que hacer una copia de `p`. En otros casos es más conveniente obtener una copia antes de la llamada y aplicar el método sobre ésta, como hacemos en `f_pert`.

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.

Restricciones debidas a la genericidad: Las operaciones que obtienen elementos almacenados en los TADs devuelven objetos de la clase `Object`. Para poder operar con ellos como objetos de su clase real hay que realizar la conversión correspondiente. Ver el método `escribir_pila_int` por ejemplo.

Por último, emplearemos acciones en vez de funciones siempre que ello nos evite realizar cálculos innecesarios, como en el siguiente ejercicio.

2.2.1 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. Recordad que hay que usar clases envoltura para apilar los números.

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

Podéis partir del siguiente código, donde se supone que la operación oculta *incrementar* ha sido previamente especificada. Realizad dos versiones de cada apartado, la primera codificando directamente una función y la segunda con una acción que sustituya la pila original por la modificada. Notad que este segundo método nos ahorra toda la gestión de la pila del resultado.

```
funcion f_incrementar (p : pila; k : natural) retorna q :
    pila
    {Pre: cierto}

    si
       es_nula(p)  −→  q := p_nula
    [] ¬es_nula(p) −→  x := cima(p) + k;
                       q := f_incrementar(desapilar(p), k);
                       {HI: q = incrementar(desapilar(p), k)}
                       q := apilar(x, q)
    fsi
    {Decr: alt(p)}
    {Post: q = incrementar(p, k)}
```

2.2.2 Ejercicio: operaciones con árboles

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 o o sumar un valor k a todos los nodos.

Por ejemplo, comenzad implementando y probando la siguiente función

```
funcion f_tam (a : arbol) retorna n : nat
    {Pre: cierto}

    si
       es_nulo(a)  −→  n := 0
    [] ¬es_nulo(a) −→  n1 := f_tam (hi(a));
                       n := f_tam (hd(a));
                       {HI: n1 = tam (hi(a)) ∧ n = tam (hd(a))}
                       n := 1 + n1 + n
    fsi
    {Decr: tam(a)}
    {Post: n = tam (a)}
```