Capítulo 8. Modularidad.

Lenguajes de Programación

Carlos Ureña Almagro

Curso 2011-12

Contents

1 Introducción **2**

1.1 Compilación separada 3

1.2 Ocultamiento de información 3

1.3 Restricciones de acceso 4

1.4 Tipos abstractos de datos 4

2 Modularidad en Ada **5**

2.1 Los packages de Ada 5

2.2 Packages y compilación separada 8

2.3 Restricciones de acceso en Ada 10

2.4 Implementación de TADs en Ada 13

3 Modularidad en C/C++ **15**

3.1 Compilación separada 15

3.2 Implementación de TADs en C 17

3.3 Namespaces en C++ 19

4 Modularidad en Java **21**

4.1 Compilación separada en Java 21

4.2 Packages y restricciones de acceso de Java 22

1 Introducción

Programas grandes

Imaginemos un programa grande que puede estar compuesto básicamente de:

- declaraciones de tipos
- declaraciones de variables globales
- declaraciones de subprogramas

A cada uno de estos lo llamaremos elementos del programa

En principio, cualquier modificación en un elemento puede afectar a cualquier otro de ellos, ya que puede haber dependencias arbitrarias entre ellos (a esto se le llama cohesión)

- El programa es difícil de escribir y leer
- El programa es muy dificil de modificar o mantener
- El programa es poco fiable.

El objetivo anterior se puede conseguir diviendo un programa en partes, cada una de ellas siendo un grupo de elementos, de forma que:

- Cada elemento de un grupo esta fuertemente relaccionado con otros elementos de dicho grupo
- Las dependencias entre elementos de grupos distintos son las menores posibles, y en cualquier caso están bien documentadas.
- Cada parte puede ser reutilizable en más de un programa

La técnica anterior se llama modularidad, y es esencial para la construcción de programas grandes.

- El particionamiento adecuado es una tarea de diseño que puede ser llevada a la práctica usando casi cualquier lenguaje de programación
- La mayoría de los lenguajes contemplan mecanismos que, bien ayudan a realizar el particionamiento, o bien soportan explicitamente la modularidad.

A cada una de las partes de un programa se le llamará, en general, módulo.

En este capítulo veremos varios de dichos mecanismos:

- namespaces de C++ y C#

- packages de Ada
- packages de Java

Además, veremos los mecanismos para la compilación separada, ocultamiento de información y restricciones de acceso, mecanismos relacionados en distinto grado con los anteriores, y que sirven al mismo propósito.

1.1 Compilación separada

La compilación separada es la posibilidad de escribir un programa en más de un archivo fuente, de forma que cada uno de estos archivos puedan ser traducidos (compilado) independientemente en distintas ejecuciones del traductor.

- Cada uno de los archivos fuente se denomina una unidad de compilación
- Cada módulo de un programa se debe situar en una o varias unidades de compilación.

El mecanismo de compilación separada:

- Hace más eficiente en tiempo y memoria la traducción, ya que después de un cambio solo tiene que hacerse para las unidades de compilación afetectadas, en lugar de para todo el programa fuente.
- Se permite el trabajo simultáneo de varios programadores en distintas partes de un programa

En la mayoría de los lenguajes, los mecanismos de modularidad y compilación separada están relaccionados. Una excepción a esto es C/C++.

1.2 Ocultamiento de información

Una forma de prevenir que se establezcan dependencias innecesarias entre partes de programas es usar mecanismos que permitan o ayuden al programador de un módulo impedir el establecimiento de dichas dependencias desde otros módulos.

El ocultamiento de información (information hidding) es la capacidad de algunos lenguajes de impedir que desde un módulo A se use una declaración incluidas en otro módulo B. Esto se hace limitando el ámbito de dicha declaraciones (se dice que la declaración no es visible desde A).

Ventajas

- El ocultamiento de información ayuda a reducir la dependencias entre módulos, ya que, si se sabe que una declaración no es visible en otros módulos, podemos asumir que cualquier cambio que hagamos en la misma no afectará a esos otros módulos, sino únicamente a aquel módulo donde aparece.
- Este mecanismo permite al diseñador de un módulo seleccionar que declaraciones del mismo serán visibles desde otros módulos, por tanto, se hace explícito en el programa las dependencias que se pueden establecer

Implementación en distintos lenguajes

- Los lenguajes Ada y Java tienen mecanismos explícitos de ocultamiento de información, asociados a los mecanismos de modularidad (packages)
- En el lenguaje C/C++, el ocultamiento de información solo se puede lograr mediante el uso de compilación separada.

1.3 Restricciones de acceso

En algunos casos, es conveniente que una declaración de un tipo incluida en un módulo sea visible desde otros, de forma que se puedan declarar variables de ese tipo, pero sin que se pueda acceder a la representación asociada al tipo. Esto permite:

- usar las variables del tipo sin tener que preocuparse de la representación, lo cual mejora la legibilidad y la fiabilidad.
- modificar la representación del tipo sin modificar los programas que usan variables de dicho tipo, con lo cual el mantenimiento se hace más fácil.

Se dice que estos tipos son tipos privados del módulo.

1.4 Tipos abstractos de datos

Las restricciones de acceso permiten la implementación de tipos abstractos de datos, que son tipos de datos (conjuntos) sobre los que existen un conjunto de funciones cumpliendo determinadas propiedades. Estas propiedades definen al tipo (no el conjunto de valores)

Los tipos abstractos de datos se implementan mediante tipos o clases definidas por el programador que pueden ser usadas de forma independiente de su representación a través de un conjunto de subprogramas.

Ejemplo Sea A un conjunto cualquiera, tal que existe otro conjunto no vacío B, una serie de funciones (a las que llamamos *vacia*, *inserta*, *extrae*, *tope*), y un elemento $nueva \in A$ tales que $\forall p \in A$ y $\forall x \in B$, se cumple que:

$$\begin{aligned} vacia &\in A \to \{\, false\,, true\,\} \\ & \quad vacia(nueva) = true \\ push &\in A \times B \to A \\ & \quad vacia(push(p,x)) = false \\ extrae &\in A - \{vacia\} \to A \\ & \quad extrae(push(p,x)) = p \\ tope &\in A - \{vacia\} \to B \\ & \quad tope(push(p,x)) = x \end{aligned}$$

Si A y B son dos conjuntos dotados de las funciones descritas anteriormente, entonces los elementos de A son las pilas LIFO con elementos de tipo B. Esto se puede afirmar independientemente de que representación se use para A, es decir, nos podemos abstraer de la representación.

Un TADs se puede implementar en cualquier lenguaje mediante un tipo definido por el usuario. El código que usa el tipo no necesita depender de la representación del tipo, aunque odría hacerlo.

Las restricciones de acceso y el ocultamiento de información permiten garantizar que el código que usa el tipo va a ser siempre independiente de la representación, ya que no va a acceder nunca a la misma.

Mecanismos para restricciones de acceso en varios lenguajes

- En C no existen mecanismos específicos
- En C++, existen mecanismos ligados al mecanismo de clase
- En Ada, existen mecanismos asociados a los package
- En Java, existen mecanismos asociados al mecanismo de clase y a los packages

2 Modularidad en Ada

El principal mecanismo para modularidad, compilación separada y ocultamiento de información en el lenguaje Ada es el package.

En esta sección veremos como se definen y usan los packages de Ada.

2.1 Los packages de Ada

Un package en Ada es un conjunto de declaraciones a las que se da un nombre, de forma que dichas declaraciones se pueden reusar en varios puntos de un programa.

Los packages de Ada son la herramienta adecuada para la implementación de la modularidad, pues permiten agrupar declaraciones, y formar programas uniendo varios packages.

Declaraciones de packages en Ada

Una declaración de package es un tipo de declaración, con la siguiente forma:

package *identificador* ***is***
  declaraciones
end [*identificador*] ;

- Cuando se elabora, se produce una asociación entre el nombre y el conjunto de declaraciones, y se elaboran las declaraciones,
- Cada declaración interna al package tiene como ámbito el interior del package más el ámbito de la declaración de package (en este segundo caso hay que anteponer el nombre del package como prefijo).

Ejemplo de declaración y uso de un package Ada:

```
procedure Proc01 is
  package Ejemplo is
     pi : constant Float := 3.1415927 ;
     x : Float := pi ;
     type MiArray is array(1..10) of Float ;
  end  ;
  y : Float := 2.0 ;
  a : Ejemplo.MiArray := (1..10 => 5.0 ) ;
begin
      a(3) := y + Ejemplo.x ;
      Ejemplo.x := Ejemplo.pi ;
end  ;
```

La cláusula `use` en Ada

```
procedure Proc02 is
  package Ejemplo is
     pi : constant Float := 3.1415927 ;
     x : Float := pi ;
     type MiArray is array(1..10) of Float ;
  end  ;
  y : Float := 2.0 ;
  use Ejemplo ;
  a : MiArray := (1..10 => 5.0 ) ;
begin
      a(3) := y + x ;
      x := pi ;
end  ;
```

Colisiones de nombres en packages Ada

La aparición en un punto del programa de un mismo identificador de dos packages distintos es errónea si no se antepone el nombre del package, ya que ocurre una ambigüedad que el compilador no puede resolver.

```
procedure Proc03 is
  package P1 is x : Float   ; end ;
  package P2 is x : Integer ; end ;
  use P1 ; use P2 ;

begin
  x := 0 ;       -- error
  P1.x := 3.0 ; -- bien
  P2.x := 5 ;   -- bien
end  ;
```

Cabecera y cuerpo de los packages

En la forma vista, un package no puede incluir una declaración de un subprograma:

```
procedure Proc04 is
   package Ejemplo is
      procedure Nada is -- Error
      begin
         null ;
      end ;
   end  ;
begin
   null ;
end  ;
```

Cabecera y cuerpo de los packages. Sintaxis.

Para lograr lo anterior hay que dividir el package en dos partes, con la siguiente sintaxis:

package *identificador* ***is***
  declaraciones-1
end [*identificador*] ;
.....
package body *identificador* ***is***
  declaraciones-2
end [*identificador*] ;

Cabecera y cuerpo de los packages. Ambitos.

- La primera parte es llamada la cabecera del package,
- La segunda parte se llama cuerpo del package

- El ámbito de *declaraciones-1* es el mismo que el ámbito de la declaración del package, e incluye al cuerpo
- El ámbito de *declaraciones-2* es el solo el cuerpo del package (las declaraciones solo son visibles en el cuerpo).

Declaraciones incompletas en las cabeceras

En la cabecera puede aparecer una declaración incompleta de un subprograma, y entonces, en el cuerpo, debe aparecer la declaración completa:

```
procedure Proc04 is
   package Ejemplo is
      procedure Nada ;
   end ;
   package body Ejemplo is
      procedure Nada is
         begin null ; end ;
   end ;
begin
   null ;
end ;
```

2.2 Packages y compilación separada

Además de la forma vista antes (que es poco útil), un package Ada puede constituir por si mismo una unidad de compilación. En este caso se puede hacer ocultamiento de información. Es la forma más frecuente de declarar los packages.

- El package se incluye en archivos fuente distintos de los archivos que incluyen al programa que usa el package.
- En este caso, a la cabecera se le llama parte visible y al cuerpo, parte oculta

Ambitos y unidades de compilación

- Las declaraciones de la parte visible tienen como ámbito la parte oculta más otras unidades de compilación que declaren que usan el package (mediante la cláusula `with`)
- La parte visible debe ser leida por el compilador cuando traduzca otras unidades de compilación que usen el package
- No es necesario leer la parte oculta, y por tanto no es necesario proporcionarla para usar el package (permite el ocultamiento de información).

Estructura de una unidad de compilación

Archivo *id* – *package*`.ads` :

package *id-package* ***is***
declaraciones-parte-visible
end [*id-package*] ;

Archivo *id* – *package*`.adb` :

package body *id-package* ***is***
declaraciones-parte-oculta
end [*id-package*] ;

Uso de un package desde subprogramas

Desde otras unidad de compilación podemos usar un package con la palabra clave `with` incluida en la llamada cláusula de contexto

(a) Desde unidades de compilación formadas por un subprograma:

with *id-package* ;
[**use** *id-package* ;]
...
procedure Ejemplo ***is***
...
begin
..... *id-package.id* ....
end Ejemplo;

Uso de un package desde otros packages

Desde otras unidad de compilación podemos usar un package con la palabra clave `with` incluida en la llamada cláusula de contexto

(b) Desde la parte visible u oculta de otro package

with *id-package* ;
[**use** *id-package* ;]
...
package [**body**] Ejemplo ***is***
...
end Ejemplo;

Ejemplo de uso de un package

Aquí, usamos el package `Text_IO` desde el subprograma `Proc06`

```
with Text_IO ;

procedure Proc06 is
begin
   Text_IO.Put_Line("Hola Mundo!") ;
end  ;
```

Uso de un package desde la cabecera de otro

Archivo `pack01.ads`

```
package pack01 is
    type Entero is new Integer ;
end ;
```

Archivo `pack02.adb`

```
with pack01 ;
package pack02 is
   i : pack01.Entero ;
   procedure Ejemplo( p : pack01.Entero ) ;
end ;
```

Ejemplo de uso de un package desde el cuerpo de otro

Archivo `pack02.adb`

```
with Text_IO ;
package body pack02 is
   procedure Ejemplo( p : pack01.Entero ) is
   begin
       i := p ;
       Text_IO.Put_Line("hola!");
   end ;
end ;
```

2.3 Restricciones de acceso en Ada

Restricciones de acceso en Ada

La cabecera de un package puede contener una sección privada, que puede contener declaraciones de tipos accesibles exclusivamente desde el cuerpo del package:

package *id-package* ***is***
  declaraciones-públicas
  private
    declaraciones-privadas
end [*id-package*] ;

A la parte anterior a `private` se le llama sección pública

Ejemplo de restricciones de acceso

Archivo `pack03.ads`

```
package pack03 is
   type EnteroPublico is new Integer ;
   private
       type EnteroPrivado is new Integer ;
end ;
```

Archivo `proc07.adb`

```
with pack03 ;
procedure Proc07 is
  i : pack03.EnteroPublico ; -- bien
  j : pack03.EnteroPrivado ; -- error
begin   null ; end  ;
```

Tipos privados en packages Ada

La forma anterior no es muy útil, ya que se podría haber definido `EnteroPrivado` en la parte oculta (el cuerpo del package). La utilidad de la sección privada proviene de la posibilidad de uso de declaraciones de tipos privados:

package *id-package* ***is***
  type *id-tipo* ***is*** **private** ;
  ...
  private
    type *id-tipo* ***is*** *descriptor-tipo* ;
  end [*id-package*] ;

Se pueden declarar variables del tipo privado y copiarlas, pero no se puede acceder a la representación del tipo.

Ejemplo de tipo privado en un package

Un ejemplo de un tipo privado es el tipo `Rep` en el siguiente package:

Archivo `pack04.ads`

```
package pack04 is
   type T is private ;
   private
       type T is record
          x : Integer ;
       end record ;
end ;
```

Uso de tipos privados

El anterior package puede ser usado para declarar variables de tipo `T`, pero no para acceder a su estructura:

```
with Pack04 ;

procedure Proc08 is

  r,s : Pack04.T ; -- bien
  i   : Integer ;

begin
   r := s ; -- bien
   i := r.x ;  -- error
end  ;
```

Uso de variables de tipos privados

El uso de variables de tipos privados solo se puede hacer mediante subprogramas declarados en la sección pública:

```
package Pack05 is
   type T is private ;
   procedure Asignar( r : out T ; v : Integer ) ;
   function Leer( r : T ) return Integer ;
   private
       type T is record
          x : Integer ;
       end record ;
end ;
```

Acceso a tipos privados en la parte oculta

La implementación de los subprogramas aparece en la parte oculta:

```
package body Pack05 is
   procedure Asignar( r : out T ; v : Integer ) is
   begin
      r.x := v ;
   end ;
   function Leer( r : T ) return Integer is
   begin
      return r.x ;
   end ;
end ;
```

Ejemplo de uso de tipos privados

Ahora podemos declarar variables tipo `T` y acceder a las mismas via los subprogramas anteriores:

```
with Pack05 ;

procedure Proc09 is

  r,s : Pack05.T ;
  i   : Integer ;

begin
   Pack05.Asignar(r,3) ;
   i := Pack05.Leer(r) ;
   Pack05.Asignar(s,i) ;
end  ;
```

Tipos `limited private` en packages

En algunos casos, es conveniente no permitir asignaciones de variables del tipo privado (por ejemplo, una cabecera de una lista).

Lo anterior se puede conseguir declarando el tipo privado como `limited private` en lugar de solo `private`, es decir, con una clausula de la forma:

type *id-tipo* ***is*** **limited private** ;

2.4 Implementación de TADs en Ada

Los TADs se pueden implementar mediante tipos privados en Ada. Hay dos opciones:

- como tipos-valor, con la representación en la sección privada

- como tipos-referencia, con la representación en la parte oculta

Ejemplo: TAD pila lifo de enteros (tipo-referencia)

```
package PilasEnteros is

   type PilaEnt is private ;

   function  Nueva return PilaEnt ;
   procedure Push ( p : PilaEnt ; v : Integer ) ;
   procedure Pop  ( p : PilaEnt ) ;
   function  Tope ( p : PilaEnt ) return Integer ;
   function  Vacia( p : PilaEnt ) return Boolean ;

   private
       type ReprPilaEnt ;
       type PilaEnt is access ReprPilaEnt ;
end ;

package body PilasEnteros is

   type Nodo ;
   type ReprPilaEnt is access Nodo ;

   type Nodo is record
       v   : Integer ;
       sig : ReprPilaEnt ;
   end record ;

   function Nueva return PilaEnt is
   begin
      return new ReprPilaEnt'(null) ;
   end ;

   .....

   ....

   procedure Push( p : PilaEnt ; v : Integer ) is
   begin
      p.all := new Nodo'(v,p.all);
   end ;

   procedure Pop( p : PilaEnt ) is
   begin
      p.all := p.all.sig ;
   end ;
```

```
    ....

    ...

    function Tope( p : PilaEnt ) return Integer is
    begin
       return p.all.v ;
    end ;

    function Vacia( p : PilaEnt ) return Boolean is
    begin
       return p.all = null ;
    end ;

end PilasEnteros ;
```

3 Modularidad en C/C++

En esta sección veremos los mecanismos de compilación separada y ocultamiento de información en C/C++, así como namespaces de C++

3.1 Compilación separada

- En C/C++, una unidad de compilación es una archivo con una serie de declaraciones.
- Las declaraciones son declaraciones de tipos, valores (constantes), subprogramas (y clases).
- las declaraciones son en principio invisibles desde otras unidades de compilación, con lo cual se implementa el ocultamiento de información

En los lenguajes C y C++

- No existen mecanismos explícitos para definir modulos
- No es posible indicar explícitamente que declaraciones de una unidad de compilación son visibles desde otras unidades
- No es posible indicar explicitamente en una unidad de compilación que otras unidades de compilación o declaraciones de las mismas se usan.

Sin embago en C y C++ es posible usar declaraciones de una unidad en otras:

- Para que una declaración de una unidad de compilación A sea usable en (visible desde) otra unidad de compilación B, es necesario duplicar en B el texto de dicha declaración (solo la cabecera en el caso de subprogramas o declaraciones incompletas de tipo).
- El ocultamiento de información se implementa dando a conocer únicamente las declaraciones de B que su programador desee publicar, y no proporcionando el texto de dicha unidad B

archivo `A.c`

```
typedef int Entero ;
const int dos = 2 ;
void dobla( Entero * j ) { *j *= dos ; }
```

archivo `B.c`

```
typedef int Entero ;
void dobla( Entero * j ) ;

int main() {
   Entero k = 3 ;
   dobla(&k) ;
   return 0 ;
}
```

Los traductores o compiladores de C/C++ suelen usarse conjuntamente con otro programa llamado preprocesador

- El preprocesador es una herramienta que ayuda a realizar esos duplicados de forma automática, mediante el uso de directivas include en el código fuente.
- Estas directivas son independientes del lenguaje en si, sirven simplemente para sustituir sus apariciones en el texto de un programa por una secuencia de carácteres obtenida de un archivo.

El esquema anterior es causa de frecuentes errores en programas medianos y grandes con muchas unidades de compilación ya que:

- El preprocesador es transparente para el compilador o traductor, de forma que este última programa procesa las distintas apariciones de las declaraciones en distintas unidades de compilación de forma completamente independiente.
- El compilador no tiene información explícita alguna de la relación de uso entre unidades de compilación o módulos.

La directiva `#include` permite repetir en un programa un conjunto de declaraciones. Cada unidad de compilación suele situarse en dos archivos fuente:

- uno de extensión `.c`, que contiene todas las declaraciones completas (visibles y ocultas) del módulo

- otro (para el cual se usa, por convención, la extensión `.h`) con aquellas declaraciones del archivo `.c` que son públicas. A este archivo se le llama archivo de cabecera. Es el que se proporciona a los usuarios del módulo.

Ejemplo de una unidad de compilación con declaraciones públicas y ocultas:

Archivo `cmod01.h`

```
typedef int Entero ;
void dobla( Entero * j ) ;
```

Archivo `cmod01.c`

```
#include "cmod01.h"

const int dos = 2 ;
void dobla( Entero * j ) { *j *= dos ; }
```

Ejemplo de uso de las declaraciones públicas de `cmod01`

```
#include "cmod01.h"

int main() {
   Entero k = 3 ;
   dobla(&k) ;
   return 0 ;
}
```

3.2 Implementación de TADs en C

Una TAD puede ser implementado en C usando una unidad de compilación que conforma un módulo.

- Las declaraciones públicas del módulo se incluirán en un archivo de cabecera `.h`, y las ocultas en un `.c` (es conveniente dar el mismo nombre base al archivo `.c` y al `.h`, este nombre es el nombre del módulo)
- Al igual que en Ada, se puede lograr que los módulos que usen el TAD sean completamente independientes de la representación usada, dejando dicha representación en la parte oculta.

La implementación del TAD pila LIFO de enteros en C puede hacerse como sigue:

Archivo `PilasEnteros.h`

```
typedef struct ReprPilaEnt * PilaEnt ;

PilaEnt  Nueva() ;
void     Push ( PilaEnt p, int v ) ;
void     Pop  ( PilaEnt p ) ;
int      Tope ( PilaEnt p ) ;
int      Vacia( PilaEnt p ) ;
```

Archivo `PilasEnteros.c`

```
#include <stdlib.h>
#include "PilasEnteros.h"

struct ReprPilaEnt
{
    struct Nodo * cabe ;
} ;

struct Nodo
{
   int     v ;
   struct Nodo * sig ;
} ;

...


...
PilaEnt Nueva()
{
   PilaEnt p = (PilaEnt) malloc(sizeof(struct ReprPilaEnt)) ;
   p->cabe = NULL ;
   return p ;
}

void Push( PilaEnt p, int v )
{
   struct Nodo * n = (struct Nodo *) malloc(sizeof(struct Nodo)) ;
   n->v = v ;
   n->sig = p->cabe ;
   p->cabe = n ;
}
....


....
void Pop( PilaEnt p )
```

```
{
   struct Nodo * n = p->cabe ;
   p->cabe = n->sig ;
   free(n) ;
}

int Tope ( PilaEnt p )
{
   return p->cabe->v ;
}

int Vacia( PilaEnt p )
{
   return p->cabe == NULL ;
}
```

Una ejemplo de unidad de compilación que use el modulo de pilas podría ser este:

```
#include <stdio.h>
#include "PilasEnteros.h"

int main()
{
  PilaEnt p = Nueva()  ;
  int     i ;

  Push(p,5);
  i = Tope(p);
  Pop(p) ;
  if (Vacia(p)) printf("p esta vacía\n");
}
```

3.3 Namespaces en C++

Uno de los principales problemas que se plantean en C para la construcción de programas grandes, dotados de muchos módulos, son las colisiones de nombres.

- Ocurre cuando dos modulos o unidades de comilación distintas producen distintas asociaciones para un mismo identificador
- Cuando se hace un `#include` del segundo de ellos, se produce un error o un efecto indeterminado, con lo cual es imposible que un tercer módulo use estos dos módulos simultáneamente
- Lo anterior reduce notablemente la reusabilidad de los módulos.

- Una solución consiste en anteponer un prefijo común a cada identificador declarado en un módulo. Soluciona el problema en muchos casos, pero no siempre, y además, hace los programas menos legibles (los identificadores tienden a ser muy largos)
- En una revisión de C++ se introdujo el mecanismo de la namespaces para solucionar este problema.

Al igual que los packages de Ada, un namespace en C++ es un conjunto de declaraciones a las que se les asocia un identificador (su nombre). Un namespace se introduce con una declaración como la siguiente:

namespace *id-namespace*
{
declaraciones
}

Donde *id-namespace* es el nombre del namesapce y *declaraciones* es el conjunto de declaraciones asociadas.

Los namespaces en C++ se pueden introducir en varias declaraciones separadas, es decir, es válido escribir:

namespace *id-namespace*
{
declaraciones-1
}
....

namespace *id-namespace*
{
declaraciones-2
}

- la primera declaración introduce el namespace y le asocia el con de declaraciones *declaraciones-1*
- la segunda declaración añade *declaraciones-2* al namespace.
- lo anterior implica que en distintos entornos un mismo namespace puede tener asociado distintos conjuntos de declaraciones

Ejemplo de namespace:

```
namespace Ejemplo
{   const double pi = 3.1415927 ;
    double  x = pi ;
    typedef double MiArray[10] ;
} ;
double y = 2.0 ;
Ejemplo::MiArray a = {5,5,5,5,5, 5,5,5,5,5 } ;
```

```
int main()
{   a[3] = y + Ejemplo::x ;
    Ejemplo::x = Ejemplo::pi ;
    return 0 ;
}
```

La clausula `using` permite acortar la notación:

```
namespace Ejemplo
{   const double pi = 3.1415927 ;
    double  x = pi ;
    typedef double MiArray[10] ;
} ;
double y = 2.0 ;
Ejemplo::MiArray a = {5,5,5,5,5, 5,5,5,5,5 } ;

int main()
{   a[3] = y + Ejemplo::x ;
    using namespace Ejemplo ;
    x = pi ;
    return 0 ;
}
```

4 Modularidad en Java

En los lenguajes orientados a objetos, como Java, el concepto esencial es el de clase, y los programas consisten básicamente en conjuntos de declaraciones de clases. En la mayoría de los lenguajes OO, existen, sin embargo, mecanismos para agrupar las clases relaccionadas en conjuntos de clases, asimilables a los módulos de los lenguajes no OO.

En Java, cabe destacar, en esta línea, las siguientes posibilidades:

- Compilación separada
- Packages y restricciones de acceso

4.1 Compilación separada en Java

- En Java, una unidad de compilación es un archivo de extensión `.java` que contiene una o varias declaraciones de clases y/o interfaces. Cada unidad de compilación puede ser traducida independientemente de otras.
- La traducción de un `.java` produce un archivo con código intermedio (de extensión `.class`) por cada clase contenida en el archivo `.java` original.

- Los archivos `.class` son interpretables con el interprete de código intermedio de Java.
- Las declaraciones de una unidad de compilación son visibles desde otras unidades de compilación, siempre y cuando al traducir esas otras unidades se tenga acceso los archivos `.class` obtenidos de traducir la primera.

4.2 Packages y restricciones de acceso de Java

En Java, las declaraciones de clases e interfaces se pueden agrupar en packages. El mecanismo de los packages de Java es similar a los namespaces de C++ (evita colisiones de nombres), excepto que añade la posibilidad de restriciones de acceso.

Las declaraciones de una unidad de compilación se añaden a un package si al inicio del texto aparece una clausula de la forma:

package *id-package* ;

En Java:

- Al igual que ocurre en C++ con los namespaces, un package puede estar definido en más de una unidad de compilación
- Las declaraciones de un package son por defecto ocultas, es decir, su ámbito es exclusivamente el package que las declara.
- En un package, las declaraciones de clases etiquetadas como `public` son acesibles desde otras unidades de compilación. Estas clases deben ser declaradas en un archivo que solo las contenga a ellas.

A modo de ejemplo, veamos un package Java (`P1`)

Archivo `P1/Privadas.java`

```
package P1 ;

class Privada1
{  public void escribe()
   { System.out.println("clase privada 1") ; }
}
class Privada2
{  public void escribe()
   { System.out.println("clase privada 2") ; }
}
```

Archivo `P1/Publica.java`

```
package P1 ;

public class Publica
{  public void escribe()
   {
      Privada1 p1 = new Privada1() ;
      Privada2 p2 = new Privada2() ;
      System.out.println("clase pública:") ;
      p1.escribe() ;
      p2.escribe() ;
   }
}
```

El package anterior podría usarse desde una unidad de compilación como esta:

```
class Main
{
   public static void main( String[] args )
   {
      P1.Publica p = new P1.Publica() ;
      p.escribe() ;
   }
}
```

La clausula `import` permite acortar la notación:

```
import P1.* ;

class Main2
{
   public static void main( String[] args )
   {
      Publica p = new Publica() ;
      p.escribe() ;
   }
}
```

fin del capítulo.