CAPÍTULO 6

En este capítulo, se exploran:

- Las características de tres tipos genéricos de colecciones: listas ordenadas, conjuntos y diccionarios.
- Los detalles de varios tipos/clases de colecciones predefinidos en Java, así como la manipulación de matrices clásicas en Java.
- La organización de clases lógicamente relacionadas en paquetes en Java y la importancia de importar paquetes para utilizar las clases que contienen.
- El uso de colecciones para modelar conceptos o situaciones complejas del mundo real.
- Técnicas de diseño para crear nuestro propio tipo de colección.

6.1 ¿QUÉ SON LAS COLECCIONES?

Son elementos que nos permiten tener una manera de reunir objetos a medida que se crean para poder gestionarlos como un grupo y operar con ellos de manera colectiva, además de poder referirnos a ellos individualmente cuando sea necesario.

Piensa en una colección como en una huevera, y los objetos que contiene como los huevos: tanto la huevera como los huevos son objetos, pero con propiedades decididamente diferentes.

Las colecciones están definidas por clases y deben ser instanciadas.

Java predefine varios tipos de clases de colección diferentes. Como con cualquier clase, un objeto de colección debe ser instanciado antes de poder ser utilizado.

```
CollectionType<elementType> x;
```

for example:

```
ArrayList<Student> x;  // ArrayList is one of Java's predefined collection types.
```

hasta que "entreguemos" a x un objeto de colección específico al que referirse, se dice que x está sin definir

```
x = new CollectionType<elementType>();
```

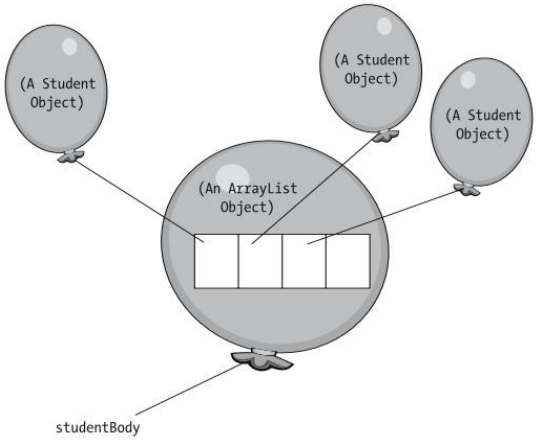
for example:

```
x = new ArrayList<Student>();
```

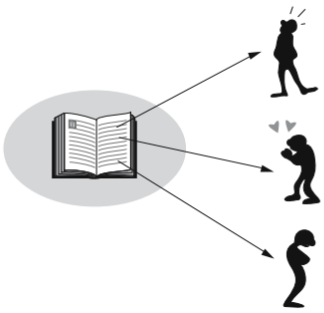
El objeto de colección recién creado es como una huevera vacía, mientras que la variable de referencia que lo señala actúa como una manija que nos permite encontrar y acceder a esta "huevera" en la memoria de la JVM en cualquier momento.

Las colecciones organizan referencias a otros objetos.

La analogía de "colección como huevera" es una simplificación excesiva, ya que en realidad no almacenamos físicamente objetos dentro de la colección, sino que almacenamos referencias a esos objetos. Los objetos organizados por una colección existen fuera de ella en la memoria de la JVM, mientras que solo sus referencias están dentro de la colección



Entonces, tal vez una analogía mejor que la de "colección como huevera" sería la de una colección como una libreta de direcciones: registramos una entrada en una libreta de direcciones (colección) para cada una de las personas (objetos) con las que deseamos poder contactar, pero las personas mismas están físicamente distantes.



Las colecciones están encapsuladas.

No necesitamos conocer los detalles privados de cómo se almacenan internamente las referencias de objetos en un tipo específico de colección para poder usar la colección correctamente; solo necesitamos conocer las características públicas de una colección, en particular, los encabezados de sus métodos públicos, para elegir un tipo de colección apropiado para una situación particular y utilizarlo de manera efectiva.

6.2 SE PRESENTAN TRES TIPOS GENÉRICOS DE COLECCIÓN

Listas Ordenadas: Permiten la inserción y recuperación de elementos en un orden específico. Los objetos pueden ser recuperados según su posición en la lista. Las listas ordenadas se expanden automáticamente al agregar nuevos elementos y se ajustan al eliminar elementos. Ejemplos de clases Java que las implementan son ArrayList, LinkedList, Stack y Vector.

Diccionarios: También conocidos como mapas, almacenan referencias de objetos junto con claves únicas para una recuperación rápida. Las claves suelen basarse en valores de atributos del objeto. Los diccionarios permiten la iteración a través de los elementos en un orden definido. Ejemplos en Java incluyen HashMap, Hashtable y TreeMap.

Conjuntos: Colecciones desordenadas que no permiten duplicados. No hay forma de recuperar elementos por posición una vez que se insertan. Ejemplos de clases Java para conjuntos son HashSet y TreeSet.

6.3. ARREGLOS COMO COLECCIONES SIMPLES

En Java hay una manera de crear un array que parece más "tipo objeto", pero el código no es "bonito":

```
// Declare an array "x" of 20 Student references.
Object x  = Array.newInstance(Class.forName("Student"), 20);
```

Inicializando el contenido de un arreglo

Los valores pueden asignarse a elementos individuales de un array utilizando índices; de forma alternativa, podemos inicializar un array con un conjunto completo de valores con una única sentencia Java cuando el array se instancie por primera vez. En este último caso, los valores iniciales se proporcionan como una lista separada por comas y encerrada entre llaves.

Si no se proporciona un conjunto de valores iniciales separados por comas al instanciar una matriz por primera vez, los elementos de la matriz se inicializan automáticamente con sus valores equivalentes a cero:

- Una matriz int se inicializa para contener ceros enteros (0s).

- Una matriz double se inicializa para contener ceros de coma flotante (0.0s).

- Una matriz booleana se inicializaría para contener el valor false en cada celda.

Manipulando arreglos de objetos

En ocasiones en un arreglo se pueden presentar "minas terrestres", para solucionar este problema se puede incluir una verificación para estar seguro de que el objeto no es nulo:

```
// Step through all elements of the array.
for (int i = 0; i < studentBody.length; i++) {
    // Check for the presence of a valid object reference before trying to
    // "talk to" it via dot notation.
    if (studentBody[i] != null) {
        System.out.println(studentBody[i].getName());
    }
}
```

6.4 A MORE SOPHISTICATED TYPE OF COLLECTION: THE ARRAYLIST CLASS

Se habla de las caracteristicas del ArrayList que es basicamente que es flexible a la cantidad de objetos y a su agregacion o eliminacion de dicha arraylist. Se habla de la importancia de importar las librerias o paquetes en este caso usando el java.util* entre otras de las librerias que ya hemos usado.

se dan algunos ejemplos como:

```
// We can import individual classes by name, to better document where each class
// that we are using originates.
import java.util.ArrayList;
import java.util.Date;
import java.io.PrintWriter;
// etc.
```

tambien se habla de la importancia de que si solo usamos el .* no estamos completamente seguros de las lases que usaremos y nos permitiran hacer diferentes cosas, sin embargo si las definimos exactamente como en la imagen, entonces sabresmos donde buscar informacion para cuando no sepamos hacer algo en especifico.

The Namespace of a Class

Cuando hablamos de todos los nombres en una clase nos referimos a:

1. El nombre de la clase en sí (por ejemplo, Estudiante)

2. Los nombres de todas las características (atributos, métodos, etc.) declaradas por la clase.

3. Los nombres de cualquier variable local declarada dentro de cualquier método de la clase (incluido parámetros que se pasan)

4. Los nombres de todas las clases que pertenecen al mismo paquete que la clase que estamos compilando.

5. Los nombres de todas las clases públicas en el paquete java.lang: String, Math, System, etc.

6. Los nombres de todas las clases públicas en cualquier otro paquete que haya sido importado por la clase que estamos compilando

7. Los nombres de todas las características públicas (atributos, métodos) de las clases enumeradas en los puntos 5 y 6

Se habla sobre la importancia de referenciar y crear bien dichos nombres, osea definiendo tipos, los cuales deben ser permitidos por los paquetes importados, por ejemplo el crear un arraylist sin haber importado el paquete de ArrayList, a esto se refiere.

User-defined packages and the default package

Esta parte habla de como se pueden crear paquetes en Java, utilizando inicialmente el paquete por defecto creado para el usuario, y en cuyo caso necesitemos, podremos crear nuevos paquetes para relacionar categóricamente nuestras clases.

En cuyo caso no usemos paquetes, las clases podrán usar instancias suyas entre si sin necesidad de hacer importaciones.

Generic

En este apartado empiezan a hablar con las listas, antiguamente antes de Java 5, la forma de crear ArrayList, era de forma genérica, esto indica que en un arreglo se podía guardar diferentes instancias de diversas clases y herencias entre otros; pero después de Java 5 y actualmente los ArrayList pasan a instanciarse con un tipo predefinido de Objeto que va a guardar para que no hallan conflictos a la hora de recorrer el arreglo, por ejemplo.

ArrayList<students> = new ArrayList<Students>();

Seria la forma predefinida por Java para poder instanciar un arreglo, aunque también es posible en versiones mas modernas colocar *"new ArrayList<>()"*

Arraylist features

En los ArrayList existen 3 constructores sobrecargados, además de 38 metodos públicos

Explican a detalle el método add(E element); donde E debe ser del mismo tipo (también puede ser heredado) que la instanciación del arreglo.

Este objeto se va a agregar al final del arreglo

```
ArrayList<Student> students = new ArrayList<Student>();
Student s = new Student();
students.add(s);
// or:
students.add(new Student());
```

También esta add(int n, E element): donde hace lo mismo que el add anterior pero el "n" indica en que posición del arreglo vamos a meter el objeto.

Contains(Object element): este retorna true o false si el elemento que se desea buscar dentro del ArrayList esta verdaderamente referenciado o no.

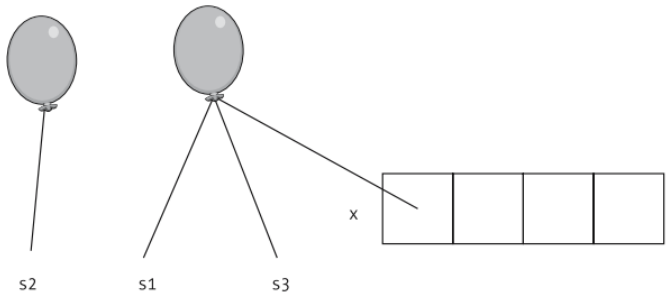
acá se muestra en representación grafica como se vería si instanciamos un objeto, lo insertamos a un arreglo y luego lo referenciamos a otra variable:

```
// Create a collection.
ArrayList<Student> x = new ArrayList<Student>();

// Instantiate two Students, but only add the FIRST of them
// to ArrayList x.
Student s1 = new Student();
Student s2 = new Student();
x.add(s1);

// Declare a third reference variable of type Student, and have it refer to
// the SAME student as s1:  that is, a Student whose reference has already been
// added to collection x.
Student s3 = s1;
```

The situation with regard to objects x, s1, s2, and s3 can be thought of conceptually as illustrated in Figure 6-10.



Con esta ultima parte de código, se puede mostrar como se instancian 2 objetos de la misma clase, se agregan a una lista y se elimina uno de ellos, con esto el arrayList solo contendrá una referencia a 1 Students y no a ambos.

```
// Create a collection.
ArrayList<Student> x = new ArrayList<Student>();

// Instantiate two Students, and add both to x.
Student s1 = new Student();
Student s2 = new Student();
x.add(s1);
x.add(s2);

// Remove s1.
x.remove(s1);
// x now only contains one reference, to s2.
```

Copying the contents of an arraylist into an array

Para copiar el contenido de una colección en una matriz se utiliza el método declarado por la clase *ArrayList* con el siguiente encabezado:

```
type[] toArray(type[] arrayRef)
```

Es decir, invocamos el método toArray en un objeto ArrayList, pasando una matriz del tipo deseado como argumento, y el método a su vez nos devolverá una matriz que contiene una copia del contenido de ArrayList, como sigue:

Si la matriz que pasamos tiene capacidad suficiente para contener el contenido de ArrayList, ese mismo objeto de matriz se llena y se devuelve. De lo contrario, se crea, completa y devuelve una matriz nueva del tipo y tamaño apropiados, y la que pasamos como argumento se ignora.

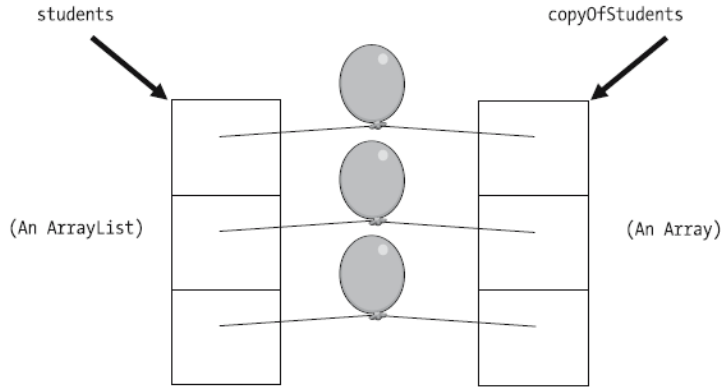


Figura 6-11. Usando el método toArray de la clase ArrayList, copiamos el contenido de un ArrayList a una matriz.

6.5 THE HASHMAP COLLECTION CLASS

Un HashMap de Java es una colección de tipo diccionario, es decir, un HashMap nos brinda acceso directo a un objeto determinado en función de un valor clave único. Tanto la clave como el objeto en sí se puede declarar como de cualquier tipo.

Cuando se declara y crea una instancia de un HashMap, debemos especificar tipos para dos elementos: la clave y el valor que representa esta clave. *Ejemplo:*

```
HashMap<String, Student> students = new HashMap<String, Student>();
```

Usamos el método put para insertar un objeto en un HashMap:

```
students.put(s1.getIdNo(), s1);
```

Este método inserta el objeto representado por el segundo argumento (s1, en el ejemplo anterior) en la colección con un valor de clave de recuperación representado por el primer

argumento (el idNo de s1, recuperado llamando al método getIdNo, en el ejemplo anterior).

Si intentamos insertar un segundo objeto en un HashMap con un valor clave que duplica la clave de un objeto al que ya hace referencia el HashMap, el método put reemplazará silenciosamente la referencia del objeto original con la nueva referencia. Si queremos evitar este tipo de reemplazo involuntario de objetos en un HashMap, podemos usar el método containsKey, que devuelve un valor verdadero si una clave particular ya existe en el HashMap, y falso en caso contrario.

El método get se utiliza para recuperar una referencia de objeto del HashMap cuyo valor clave coincide con el valor pasado como argumento al método (Si no se encuentra ninguna coincidencia, se devuelve un valor nulo):

```
Student x = students.get(id);
```

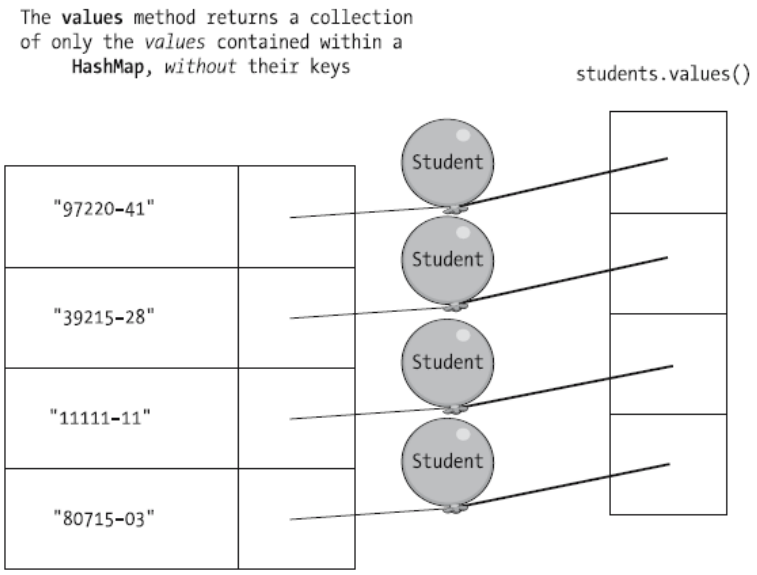


Figura 6-12. El método de valores devuelve una colección de valores (solo) de un HashMap.

Algunos de los otros métodos comúnmente utilizados declarados por la clase HashMap son los siguientes:

• *Object remove(Object key):* Elimina la referencia al objeto representado por la clave dada del HashMap.

• *boolean contains(Object value):* Devuelve verdadero si el HashMap ya hace referencia al objeto específico pasado como argumento al método, independientemente de cuál pueda ser su valor clave; de lo contrario, devuelve falso.

• *int size():* Devuelve un recuento del número de pares clave/objeto almacenados actualmente en el HashMap.

• *void clear():* Vacía el HashMap de todos los pares clave/objeto, como si acabara de crear una instancia.

• *boolean isEmpty():* Devuelve verdadero si el HashMap no contiene entradas; de lo contrario, devuelve falso.

6.6 LA CLASE TREEMAP

Es otra colección de tipo diccionario, son muy similares a los HashMaps con una diferencia notable:

- Cuando iteramos a través de un treeMap, los objetos se recuperan automáticamente de la colección en orden de clave ascendente.
- Cuando iteramos en un HashMap, no se garantiza el orden en que se recuperan los elementos

Lo siguiente es un código fuente que demuestra estas diferencias mencionadas, en el programa instanciaremos uno de cada uno de estos dos tipos de colección. Esta vez, insertaremos cadenas en las colecciones en lugar de Estudiantes; dejaremos que la misma cadena sirva tanto de clave como de valor.

```
import java.util.*;

public class TreeHash {
    public static void main(String[] args) {
    // Instantiate two collections -- a HashMap and a TreeMap -- with
    // String as both the key type and the object type.
    HashMap<String, String> h = new HashMap<String, String>();
    TreeMap<String, String> t = new TreeMap<String, String>();

    // Insert several Strings into the HashMap, where the String serves
    // as both the key and the value.
    h.put("FISH", "FISH");
    h.put("DOG", "DOG");
    h.put("CAT", "CAT");
    h.put("ZEBRA", "ZEBRA");
    h.put("RAT", "RAT");

    // Insert the same Strings, in the same order, into the TreeMap.
    t.put("FISH", "FISH");
    t.put("DOG", "DOG");
    t.put("CAT", "CAT");
    t.put("ZEBRA", "ZEBRA");
    t.put("RAT", "RAT");

    // Iterate through the HashMap to retrieve all Strings ...
    System.out.println("Retrieving from the HashMap:");
    for (String s : h.values()) {
        System.out.println(s);
    }

    System.out.println();
```

```
        // ... and then through the TreeMap.
        System.out.println("Retrieving from the TreeMap:");
            for (String s : t.values()) {
                System.out.println(s);
            }
        }
    }
```

El output del código es el siguiente:

```
Retrieving from the HashMap:
ZEBRA
CAT
FISH
DOG
RAT

Retrieving from the TreeMap:
CAT
DOG
FISH
RAT
ZEBRA
```

Tener en cuenta que el treeMap ordeno las cadenas, mientras que las cadenas se recuperaron en un orden arbitrario, ni en el orden en que se insertaron, ni en orden de clasificación, desde el HashMap.

Todos los otros métodos que se han discutido para la clase HashMap funcionan de la misma manera para TreeMaps.

Si los treeMaps son idénticos a los HashMaps con la ventaja de la iteración ordenada, ¿Por qué no ignoramos la clase HashMap y siempre usamos la treeMap para crear colecciones de diccionarios?

La respuesta esta en el hecho de que los diccionarios pueden utilizar cualquier tipo de objeto como clave.

Si utilizamos cadenas como claves, como hemos hecho en todos los ejemplos hasta ahora, un TreeMap no tiene problemas para determinar como ordenarlas, porque la clase String define un método “compareTo” que la clase TreeMap aprovecha. Pero si utilizamos un tipo definido por el usuario como clave, la carga recae en nosotros para definir programáticamente lo que significa ordenar ese tipo de objeto.

Ejemplo

Creamos una colección diccionario en la que un objeto Departamento sirve de clave, y el Profesor que preside el departamento es el valor referenciado por una clave dada. Si declaramos que la colección es un TreeMap, debemos definir lo que significa que un departamento este “antes que” otro de manera ordenada si queremos iterar la

colección.El código para esto es bastante avanzado, con lo que sabemos ahora, basta decir que si realmente no necesitamos iterar a través de un diccionario en orden de clave, no vale la pena la molestia adicional de utilizar TreeMap cuando HashMap lo haría bastante bien.

6.7 EL MISMO OBJETO PUEDE SER REFERENCIADO POR VARIAS COLECCIONES

Cuando hablamos de insertar un objeto en una colección, lo que queremos decir es que estamos insertando una referencia al objeto, no el objeto en sí.

Esto implica que el mismo objeto puede ser referenciado por múltiples colecciones simultaneamente

Piensa en una persona como un objeto, y en su numero de telefono como una referencia para llegar a esa persona.

Ahora piensa en una agenda de direcciones como una colección:

Es fácil ver que el numero de teléfono de esa misma persona(referencia) puede estar registrado en muchas otras libretas de direcciones(colecciones) simultáneamente.

Ejemplo relacionado Con el SRS:

Dado los alumnos inscritos para asistir a un curso concreto, podemos mantener simultáneamente lo siguiente:

- Una lista ordenada de estos estudiantes para saber quien matriculo primero en un curso
- Un diccionario que nos permita recuperar un objeto Estudiante determinado basándonos en su nombre
- Quizá un segundo diccionario para todo el SRS que organice a todos los estudiantes de la universidad basándose en sus numero de identificación

Representación conceptual:

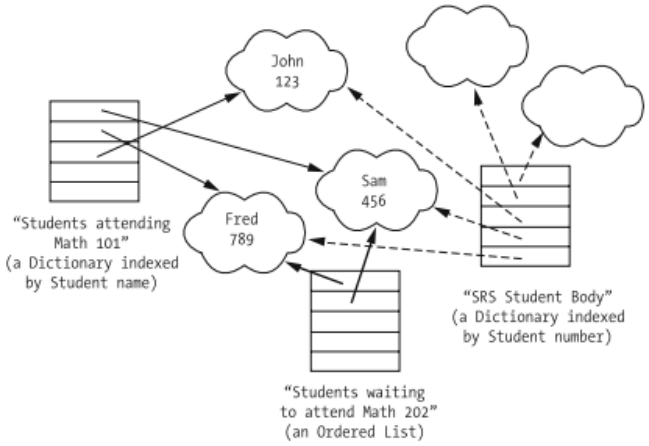


Figure 6-14. *A given object may be referenced by multiple collections simultaneously.*

Uno error común es asumir que si una colección dada se vacía (mediante una llamada explicita a su método clear) entonces los objetos a los que la colección hacía referencia previamente serán recolectados. Recordad cap 3 sobre recolección de basura, solo cuando ya no hay ningún manejador sobre un objeto dado, se recolecta la basura.

Su memoria será reciclada por la JVM. Dado que los objetos son a menudo referenciados por múltiples colecciones simultáneamente, no podemos asumir que borrar una sola colección liberara los objetos a los que referenciaba. Ejemplo, si borramos el contenido de la colección "Students wating to attend Math 202" de la figura 6-14, los objetos estudiante "John", "Fred" y "Sam" todavía serán referenciados por otras dos colecciones. A menos que estos objetos Estudiante fueran eliminados posteriormente de esas otras colecciones, no serían recolectados.

6.8 INVENTAR NUESTROS PROPIOS TIPOS DE COLECCIÓN

Los distintos tipos de colecciones tienen propiedades y comportamientos diferentes.

Por lo tanto, debe familiarizarse cono los distintos tipos de colecciones predefinidas disponibles para el lenguaje OO que elijas, y elegir el mas apropiado para lo que necesites en una situación dada.

Si no le conviene alguno, invente uno, Aquí es donde empezamos a darnos cuenta del potencial de un lenguaje OO, tenemos la capacidad de inventar nuestros propios tipos definidos por el usuario, por supuesto tenemos via libre para definir nuestros propios tipos de colección.

Tenemos varias formas de crear nuestras propias clases de colección:

1. Enfoque #1: podemos diseñar una nueva clase collection desde cero.
2. Enfoque#2: podemos utilizar las técnicas del camp 5 para extender una colección predefinida.

3. Enfoque #3: crear una clase “envoltorio” que encapsule uno de los tipos de colección incorporados para “abstraer” algunos de los detalles involucrados en la manipulación de la colección.

Vamos a discutir cada uno de estos enfoques:

Enfoque # 1: Podemos diseñar una nueva clase collection desde cero.

Suele llevar bastante trabajo. Dado que la mayoría de lenguajes OO proporcionan una amplia gama de tipos de colección predefinidos, casi siempre es posible encontrar un tipo de colección preexistente para utilizar como punto de partida, en cuyo caso casi siempre se preferiría uno de los otros dos enfoques

Enfoque #2: Extendiendo una clase colección predefinida

La clase `MyIntCollection` extiende la clase `ArrayList` para crear una colección personalizada que puede realizar las mismas operaciones que un `ArrayList` estándar, pero también realiza un seguimiento del valor más pequeño, el valor más grande y el promedio de los enteros almacenados en la colección. Para lograr esto, se utilizan clases de envoltura como `Integer` para almacenar los valores enteros en la colección, ya que las colecciones en Java solo pueden contener tipos de referencia. El constructor de `MyIntCollection` utiliza `super()` para llamar al constructor de la clase base `ArrayList`. Se anula el método `add` para incluir la lógica necesaria para rastrear el valor más pequeño, el valor más grande y la suma total de los enteros agregados a la colección. Además, se proporcionan métodos para obtener el valor más pequeño, el valor más grande y el promedio de los enteros en la colección. Finalmente, se muestra un ejemplo de cómo usar `MyIntCollection` para agregar enteros, obtener el tamaño de la colección y obtener estadísticas adicionales sobre los enteros almacenados.

Enfoque #3: Encapsulando una colección estándar

El enfoque #3 para crear una colección personalizada implica encapsular una instancia de la clase ArrayList en lugar de extenderla. Esta vez, creamos una clase llamada MyIntCollection2 que encapsula un ArrayList como un atributo. Eliminamos la extensión de la clase ArrayList y creamos métodos personalizados para realizar las operaciones deseadas. Aquí está el resumen de los cambios realizados:

- Eliminamos la extensión de ArrayList en la declaración de clase.
- Encapsulamos un ArrayList<Integer> como un atributo llamado numbers.
- En el constructor, instanciamos el ArrayList.
- Creamos un método size() para obtener el tamaño de la colección delegando al ArrayList encapsulado.
- Creamos un método add(int i) para agregar elementos a la colección, manteniendo el seguimiento de los valores más pequeños, más grandes y la suma total.

- Mantenemos los métodos existentes como getSmallestInt(), getLargestInt(), y getAverage() sin cambios.
- La forma de usar MyIntCollection2 en el cliente es idéntica a la forma de usar MyIntCollection.

Este enfoque muestra el poder de la encapsulación al permitir la creación de una colección personalizada con funcionalidades específicas sin depender directamente de la implementación de la clase base.

Trade-offs of approach #2 vs. approach #3 (ventajas y desventajas del enfoque #2 y frente al enfoque #3)

El enfoque #2, que trata de extender una clase de colección predefinida, es más económico en cuanto a uso de memoria, ya que al crear una instancia de la clase MyIntCollection, que hereda de ArrayList, se crea sólo un objeto en memoria. Por el contrario, al usar el enfoque #3 de encapsulamiento, como en MyIntCollection2, se crean dos objetos en memoria al instanciar la colección encapsulada junto con la clase que la envuelve.

Alternativamente, el enfoque #3 ofrece la ventaja de poder controlar qué comportamientos públicos de la colección encapsulada se exponen al código del cliente. Mientras que MyIntCollection hereda todos los 30 comportamientos públicos de ArrayList, MyIntCollection2 puede exponer solo los métodos necesarios, haciendo más sencillo el uso desde el punto de vista del cliente. Además, mediante la encapsulación, es posible "disfrazar" estos métodos dándoles diferentes nombres de la siguiente manera:

```
public class MyIntCollection2 {
    // details omitted ...

    // This was formerly the size() method ...
    public int getIntCount() {
        // DELEGATION!
        return numbers.size();
    }
```

En resumen, un método no es muy superior a otro y al entender las sutiles diferencias entre ambos podremos elegir el que mejor se ajuste a nuestras necesidades.

6.9 COLLECTIONS AS METHOD RETURN TYPES (COLECCIONES COMO TIPOS DE RETORNO DE MÉTODOS)

Las colecciones pueden ser utilizadas como tipos de retorno de métodos para superar la limitación de que los métodos solo pueden retornar un único resultado. En este caso, se muestra un ejemplo utilizando la clase Course que tiene un método getRegisteredStudents que retorna una colección (ArrayList) de objetos Student que están registrados en un curso. A continuación, se muestra como el código cliente podría usar el método luego de haberlo definido:

```
// The following method returns a reference to an entire collection
// containing however many Students are registered for the Course in question.
public ArrayList<Student> getRegisteredStudents() {
    return enrolledStudents;
}

// Instantiate a course and several students.
Course c = new Course();
Student s1 = new Student();
Student s2 = new Student();
Student s3 = new Student();

// Enroll the students in the course.
c.enroll(s1);
c.enroll(s2);
c.enroll(s3);

// Now, ask the course to give us a handle on the collection of
// all of its registered students and iterate through the collection,
// printing out a grade report for each student.
for (Student s : c.getRegisteredStudents()) {
    s.printGradeReport();
}
```

En resumen, el texto destaca cómo el uso de colecciones como tipos de retorno de métodos permite a los programas manejar y manipular conjuntos de objetos de manera más eficiente, especialmente cuando se trata de estructuras de datos complejas como grupos de estudiantes en un curso.

6.10 COLLECTIONS OF DERIVED TYPES

Las colecciones pueden contener objetos de tipos derivados de una superclase común. Por ejemplo, si declaramos una colección para contener objetos de una superclase Person, podemos insertar objetos de tipo Person o cualquier tipo derivado de Person, como UndergraduateStudent, GraduateStudent, y Professor. Esto es posible por la naturaleza de la herencia, donde los objetos de las subclases son casos especiales de objetos de la superclase. A continuación, veremos un ejemplo de lo anterior que también funciona de manera similar con un ArrayList añadiendo los elementos con add():

```
Person[] people = new Person[100];

Professor p = new Professor();
UndergraduateStudent s1 = new UndergraduateStudent();
GraduateStudent s2 = new GraduateStudent();

// Add a mixture of professors and students in random order to the array.

people[0] = s1;
people[1] = p;
people[2] = s2;
// etc.
```

6.11 REVISITING OUR STUDENT CLASS DESIGN (REVISANDO EL DISEÑO DE NUESTRA CLASE DE ESTUDIANTES)

Ejemplos con la clase Student:

Declarar ArrayList como tipo de un atributo:

```
import java.util.ArrayList;

public class Student {
  private String name;
  private String studentId;
  private ArrayList<Course> courseLoad;
  // etc.
```

La figura 6-17 ilustra cómo los objetos Student, ArrayList, TranscriptEntry y Course estarían "conectados" en la memoria en tiempo de ejecución.

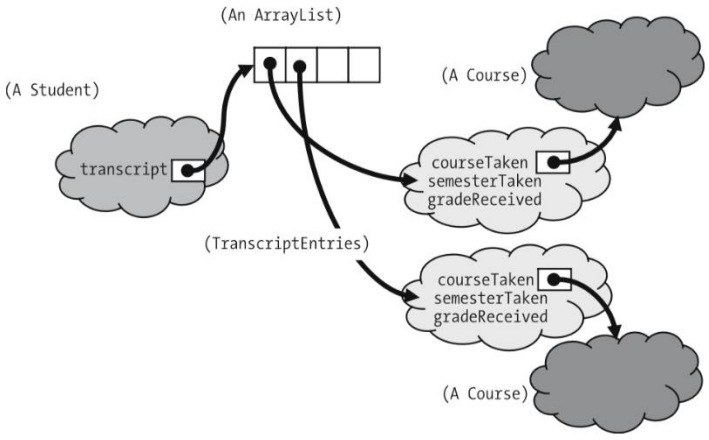


Figura 6-17. Como "se encuentran conectados" en memoria, un Student referencia una ArrayList, que a su vez hace referencia a los objetos de TranscriptEntry. Estos, a su vez, hacen referencia a un objeto del Curso.

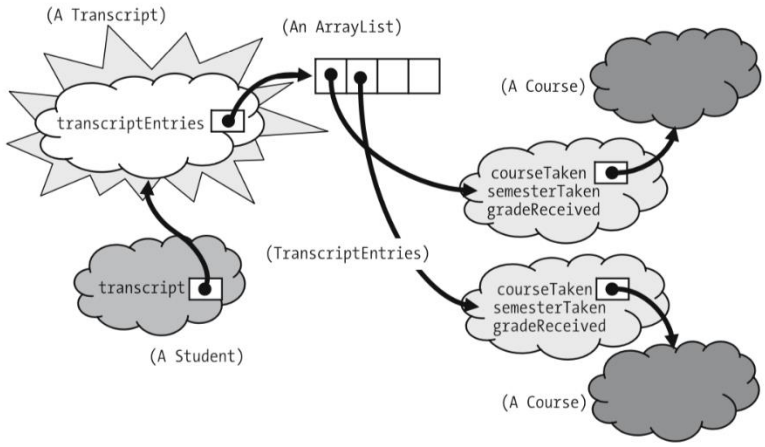


Figura 6-18. La introducción de otro nivel de abstracción en la forma de la clase de Transcript simplifica en última instancia el código del cliente, que es un objetivo de diseño importante.

RESUMEN DEL CAPÍTULO

• Las colecciones son tipos especiales de objetos que se utilizan para agrupar y gestionar referencias a otros objetos.

• La mayoría de los idiomas son compatibles con tres tipos genéricos de colección: Listas ordenadas, Conjuntos, Diccionarios (también conocidos como mapas)

• Las matrices son un tipo de colección que tiene algunas limitaciones, pero también tenemos otros tipos de colección más potentes para aprovechar con los lenguajes OO, como ArrayLists de Java, HashMaps, TreeMaps, etc.

• Es importante familiarizarse con las características únicas de cualquier tipo de colección que esté disponible para un idioma OO en particular, con el fin de hacer la selección más informada de qué tipo de colección usar para una circunstancia particular.

• Puedes inventar tus propios tipos de colección, ya sea extendiendo clases de colección predefinidas o creando "clases de envoltura" para encapsular una instancia de una clase de colección predefinida, y las sutiles diferencias entre los dos enfoques.

• Puedes evitar la limitación de que un método puede devolver solo un resultado haciendo que ese resultado sea una colección.

• Puedes crear clases compuestas sofisticadas mediante el uso de colecciones como atributos.

• "Enterrar" aumentando los niveles de detalle dentro de la capa de los servicios de abstracción para simplificar el código del cliente.