Static Features

El capitulo comienza con la introducción al problema respecto a variables que es única para cada objeto que vallamos a instanciar, por lo que le da introducción a las variables estaticas

Static Variables:

Este tipo de atributo lo usaremos para marcar una variable como “estatica”, refiriéndose a que lo que tenga por dentro esta variable (sea int, char, bool…) va a ser el mismo para todos los objetos que se vallan a crear de esa clase.
Pero esta puede ser editada y al momento de cambiarla se va a cambiar el valor para todos los demás objetos.

Del mismo modo podemos generar métodos estáticos, donde pueden ser invocados a modo general sobre una clase

7.5.4 RESTRICTIONS ON STATIC (Restricciones a los métodos estáticos)

Hay una restricción importante en los métodos estáticos: “dichos métodos no pueden acceder a las características no estáticas de la clase a la que pertenecen los métodos.”

El compilador generaría el siguiente mensaje de error con respecto a la instrucción println en el método de impresión estática:

```
non-static method getName() cannot be referenced from a static context
```

Las clases son plantillas vacías en lo que respecta a los atributos; no es hasta que instanciamos un objeto que se rellenan sus valores de atributo. Si se invoca un método estático en una clase en su conjunto, y ese método a su vez intentaba acceder al valor de un atributo, el valor de ese atributo no estaría definido para la clase, como se ilustra conceptualmente en la Figura 7-

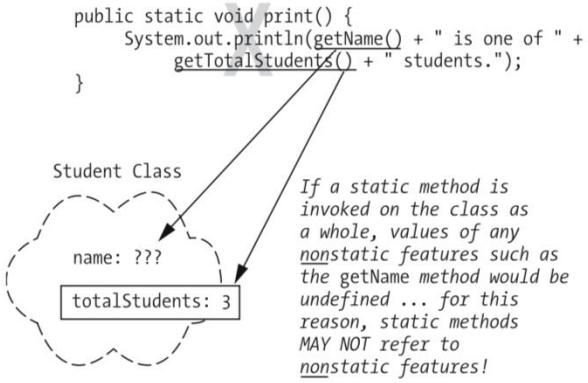

Figure 7-9. *Nonstatic attribute values are undefined in the context of a class.*

9.

Otra restricción en los métodos estáticos es que no se puede declarar que son abstractos.

El siguiente error del compilador resultaría:

```
illegal combination of modifiers: abstract and static
```

7.5.5 REVISITING THE SYNTAX OF PRINT STATEMENTS (Revisar la sintaxis de las declaraciones de impresión)

En el libro se ha usado la expresión *"System.out.println(expresión)"* para mostrar mensajes en la ventana de comandos. Resulta que

• La clase del sistema está integrada en el lenguaje Java principal (es decir, está definido en el paquete java.lang).

• Out es un atributo estático público de la clase System, declarado como de tipo PrintStream. Por lo tanto, la expresión System.out se refiere a un objeto de tipo PrintStream.

• El método de impresión de la clase PrintStream acepta el argumento de expresión y lo muestra en forma de cadena en la ventana de salida estándar.

Por lo tanto, no es necesario crear una instancia de un objeto System para imprimir mensajes en la pantalla; simplemente llamamos al método println en el atributo PrintStream estático público de la clase System.

7.5.6 UTILITY CLASSES (Clases de utilidad)

Las clases de utilidad son clases que proporcionan formas convenientes de realizar comportamientos de uso frecuente sin tener que instanciar un objeto para realizar dichos comportamientos. Estas clases a menudo se componen completamente de métodos estáticos y variables estáticas públicas.

El lenguaje Java incluye varias clases de utilidad predefinidas. Por ejemplo, la clase de matemáticas del paquete java.lang, la cual declara una variedad de métodos estáticos para calcular funciones trigonométricas, exponenciales, logarítmicas y de potencia; para redondear valores numéricos; y para generar números aleatorios.

Las clases de utilidad hacen uso de las variables finales.

7.5.7 THE FINAL KEYWORD (La palabra clave final)

La palabra clave final de Java se puede aplicar a variables, métodos y clases en su conjunto. Una variable final es una variable a la que se le puede asignar un valor solo una

vez en un programa; después de esa primera asignación, el valor de la variable no se puede cambiar. Se declara dicha variable colocando la palabra clave final justo antes del tipo de la variable, de la siguiente manera:

```
public class Example {
    // A static variable can be declared to be final ...
    public static final int x;

    // ... as can a (nonstatic) attribute.
    private final int y;

    public void someMethod() {
        // Even a local variable may be declared to be final.
        final int z;
        // etc.
```

Mientras que a una variable final local se le puede asignar un valor por separado de su declaración, no se puede hacer para otras variables finales.

Para evitar tal problema, se deben asignar valores a las variables finales de clase e instancia en el momento en que se declaran.

Variables e interfaces finales estáticas públicas

A las interfaces no se les permite declarar variables, con una excepción: a las interfaces se les permite declarar variables finales estáticas públicas para que sirvan como constantes globales, es decir, valores constantes que están en el alcance y, por lo tanto, accesibles en toda una aplicación.

La convención de nomenclatura para una variable final estática pública de una clase o una interfaz es bastante inusual: tradicionalmente se usan todas las mayúsculas para nombrarlas y, por lo tanto, usamos caracteres de guión bajo (_) para separar visualmente las palabras en nombres de varias palabras, por ejemplo, FULL_TIME y PART_TIME.

La palabra clave final también se puede aplicar a los métodos y a las clases en su conjunto:

- Un método declarado como final no se puede anular en una subclase.
- Una clase declarada para ser final no puede ser subclasificada.

7.5.8 THE STATIC IMPORT FACILITY (La instalación de importación estática)

A partir de J2SE 5.0, se puede importar selectivamente miembros estáticos específicos de una clase o interfaz, en lugar de importar la clase o interfaz en su conjunto, utilizando una declaración estática de importación.

7.5.9 CUSTOM UTILITY CLASSES (Clases de utilidad personalizadas)

Se pueden aprovechar las mismas técnicas utilizadas para crear clases de utilidad de Java predefinidas, como la clase de matemáticas, para crear nuestras propias clases de utilidad personalizadas. Por ejemplo, supongamos que vamos a tener una necesidad frecuente de hacer conversiones de temperatura de grados Fahrenheit a grados centígrados y viceversa. Podríamos inventar una clase de utilidad llamada Temperatura.

```
// A utility class to provide F=>C and C=>F conversions.
public class Temperature {
  public static double FahrenheitToCentigrade(double tempF) {
    return (tempF - 32.0) * (5.0/9.0);
  }

  public static double CentigradeToFahrenheit(double tempC) {
    return tempC * (9.0/5.0) + 32.0;
  }
}
```

Ejemplo

Entonces, para usar esta clase, simplemente escribiríamos el código del cliente de la siguiente manera:

```
double degreesF = 212.0;
double degreesC = Temperature.FahrenheitToCentigrade(degreesF);
System.out.println("A temperature of " + degreesF + " degrees F = " +
                   degreesC + "degrees C");
```

Incluso podríamos desear incluir algunas constantes de uso común, por ejemplo, los puntos de ebullición y congelación del agua en los términos F y C, como variables finales

estáticas públicas en nuestra clase de utilidad:

```
// A utility class to provide F=>C and C=>F conversions.
public class Temperature {
  // We've added some public static final variables.
  public static final double FAHRENHEIT_FREEZING = 32.0;
  public static final double CENTIGRADE_FREEZING = 0.0;
  public static final double FAHRENHEIT_BOILING = 212.0;
  public static final double CENTIGRADE_BOILING = 100.0;

  public static double FahrenheitToCentigrade(double tempF) {
    // We can utilize our new attributes in our method code.
    return (tempF  - FAHRENHEIT_FREEZING) * (5.0/9.0);
  }

  public static double CentigradeToFahrenheit(double tempC) {
    // Ditto.
    return tempC * (9.0/5.0) + FAHRENHEIT_FREEZING;
  }
}
```

Entonces también podríamos aprovechar estas constantes en nuestro código de cliente:

```
Soup s = new Soup("chicken noodle");

// Bring the soup to a boil.
if (s.getTemperature() < Temperature.FAHRENHEIT_BOILING) {
  s.cook();
}
```

Debido a que todas las características de la clase Temperature son estáticas, nunca necesitamos instanciar un objeto Temperature en nuestra aplicación.

13.1 Rounding Out Your Java Knowledge

Terminología especifica de Java

En la tabla 13-1 se compara la terminología OO genérica con la terminología especifica de Java

Table 13-1. *Comparing Generic OO vs. Java Terminology*

Generic OO Terminology Used in this Book	Formal Java-Specific Terminology Used by Sun Microsystems	Used to Describe the Following Notion
attribute	field, instance variable	A variable that is created once per object—that is, per each instance of a class. Each object has its own separate set of instance variables.
static variable (informal: static attribute)	static field, class variable	A variable that exists only once per class.
method	instance method	A function that is invoked on an object.
static method	class method	A function that can be called on a class as a whole, without reference to a specific object. Class methods can neither call instance methods nor access instance variables.
feature	member	Those components of a class that can potentially be inherited—for example, instance/class variables and instance/class methods, but ***not*** constructors.

El termino variable local se refiere a una variable que se declara dentro de un método, por lo tanto, tiene un ámbito local relativo a ese método. Las variables locales no son variables estáticas ni de instancia.

En este código se ilustran los tres tipos de variables:

```
public class Student {
  // Attributes.
  private String name;  // <== name is an instance variable
  private static int totalStudents;  // <== totalStudents is a
                                     //     class variable

  // Methods.
  public void foo(int y) { // <== parameter y is local to the method
    int x;  // <== x is a local variable
    // etc.
  }

  // etc.
}
```

La terminología especifica de Java, tal como establece en la JLS, no es tan intuitiva como la terminología genérica de OO para un concepto determinado

Ejemplo, la forma en que el termino “herencia” se utiliza en JLS en comparación con la discusión OO general.

En general en OO, si una clase A declara 3 atributos (variables de instancia), un constructor, y dos (instancia) métodos con las accesibilidades indicadas aquí:

```
public class A {
    private int x;
    protected int y;
    public int z;

    public A() { ... }

    private void foo() { ... }

    public void bar() { ... }

    // etc.
}
```

A continuacion se deriva una subclase B de esta clase de la siguiente manera:

```
public class B extends A {
    // Add one new attribute ...
    private int w;

    // ... and one new method.
    public void whatever() { ... }
}
```

Entonces, “la estructura ósea” (cap 5) de una instancia B consiste en cuatro elementos de datos int x, int y, int z e int w, pero x no será referenciable como símbolo dentro del ámbito de B porque tiene accesibilidad privada en A. Por lo tanto, usaremos métodos de acceso publico para x que heredamos de A siempre que queramos acceder al valor de x en B.

```
public class B extends A {
    // Add one new attribute ...
    private int w;

    // ... and one new method.
    public void whatever() {
        // Access our own value of x.
        System.out.println(getX());
        // etc.
    }
}
```

En términos generales, se puede describir este fenómeno de OO de la siguiente manera:

“la clase B hereda todos los atributos de A, pero solo puede referirse directamente a un atributo heredado por su nombre si no se declara que ese atributo tiene accesibilidad privada en A”.

La JLS reconoce que x se convierte en parte de la “estructura ósea” de una instancia de B, sin embargo, el JLS simplemente no proporciona una terminología alternativa para referirse a este fenómeno.

Por lo tanto, se usara el termino de “herencia” en el sentido general de OO, para referirme a cualquier cosa que se convierte en parte de la “estructura ósea” de una instancia de una subclase B en una función de su presencia en la superclase A, sea o no directamente accesible por su nombre en B, es decir, si su nombre, como símbolo, esta o no en el ámbito del compilador cuando se está compilando B.

Volviendo a la definición de Sun del termino “miembro” para referirse a “una variable o función que potencialmente puede ser heredada por subclases”, los elementos de la clase A que serian o no un miembro:

```
public class B extends A { ... }

public class A {
    private int x;   // x is a member of A, but not of B
    protected int y; // y is a member of both A and B
    public int z;    // z is a member of both A and B

    public A() { ... } // not a member of A; and, since
                       // constructors are NOT members,
                       // they cannot be inherited by B,
                       // hence this is not a member of B

    private void foo() { ... } // a member of A but not B

    public void bar() { ... }  // a member of both A and B

    // etc.
}
```

Arquitectura de aplicaciones Java

En la parte 1 del libro se examinó la anatomía de un programa de java trivialmente simple, que consiste en un método principal contenido en la lógica de nuestro programa encapsulado dentro de una clase:

```
public class Simple {
    public static void main(String[] args) {
        System.out.println("I LOVE Java!!!");
    }
}
```

- El código fuente para declarar una clase de este tipo se encuentra en un archivo llamado “Classname.java”
- Compilaremos este código fuente mediante el comando javac Simple.java para producir un archivo de código de bytes llamado Simple.class
- Ejecutamos este programa lanzando la JVM para interpretar/ ejecutar este bytecode mediante el comando java Simple.

Una aplicación Java no trivial se compone de muchos archivos fuente Java que, al ser compilados, producen muchos archivos bytecode como se indica acá:

- Normalmente tenemos un archivo de código fuente. Java para cada una de las clases modelo que definimos en nuestro diagrama de clases UML, para la aplicación SRS por ejemplo, tendremos diez archivos:

```
Course.java
CourseCatalog.java
Faculty.java
Person.java
Professor.java
ScheduleOfClasses.java
Section.java
Student.java
Transcript.java
TranscriptEntry.java
```

Se revisará el código para cada una de estas clases modelo SRS en detalle en los cap 14 y 15.

- Normalmente necesitamos una o mas clases para establecer la conectividad con almacenamiento de datos persistente de algún tipo (archivos ASCII orientados a registros, archivos XML o una base de datos relacional).

- se suele crear un archivo de código fuente. Java separado para alojar la declaración de clase para cada una de las ventanas principales que componen la interfaz grafica del usuario de una aplicación, si las hay. Para la aplicación SRS, se declaran dos clases: MainFrame y PasswordPopup que revisan en detalle en el cap 16.

- normalmente tendremos un archivo código fuente. Java separado para alojar la clase que encapsula el método oficial main, que sirve como punto de partida de la aplicación.

- Una de las principales responsabilidades del método main es instanciar objetos centrales necesarios para cumplir la misión de un sistema

- el método main también puede ser responsable de mostrar la ventana de inicio de una GUI, si nuestra aplicación tiene una, establecer conexiones con almacenamiento persistente, si es necesario; y otros tipos de cosas de inicialización de la aplicación

- a menudo se tienen clases “ayudantes” adicionales para dar soporte a la aplicación entre bastidores

Archivos Java Archive (JAR)

El código de bytes Java que compone una aplicación se suele empaquetar y entregar en forma de archivo Java Archive (JAR). Utilicemos como ejemplo una aplicación sencilla que consta de lo siguiente:

- Tres tipos definidos por el usuario: las clases Person, Student y Professor.

- Una clase envolvente del método principal llamada MyApp

para ilustrar el uso de archivos JAR. El código de nuestro sencillo ejemplo es el siguiente

```
public class Person {
    private String name;

    public void setName(String n) {
        name = n;
    }
}

public class Student extends Person {
    private Professor advisor;

    public void setAdvisor(Professor p) {
        advisor = p;
    }
}

public class Professor extends Person {
    private String title;

    public void setTitle(String t) {
        title = t;
    }
}
```

This code supports the following program:

```
public class MyApp {
    public static void main(String[] args) {
        Professor p = new Professor();
        Student s = new Student();
        s.setAdvisor(p);
    }
}
```

Creando un archivo JAR

Comenzamos compilando nuestro código:

*javac *.java*

Luego, para crear un archivo JAR, escribimos

jar cvf jarfilename.jar list_of_files_to_be_included_in_jar_file

donde el argumento de línea de comandos cvf indica que

- Deseamos crear un archivo JAR.

- Deseamos que el comando sea detallado, es decir, que muestre todo lo que sucede mientras se ejecuta el archivo JAR. todo lo que sucede mientras se crea el archivo JAR.

- Estamos designando el (archivo) nombre del archivo JAR que se va a crear.

Por ejemplo, para colocar el bytecode de nuestra aplicación simple en un archivo JAR llamado MyJar.jar, escribiríamos

jar cvf MyJar.jar Person.class Student.class Professor.class MyApp.class

Al igual que con los archivos .zip, podemos almacenar literalmente cualquier tipo de archivo dentro de un archivo JAR: código fuente, código de bytes, archivos de imagen, ¡incluso otros archivos JAR!

Inspeccionar el contenido de un archivo JAR

Para inspeccionar y listar el contenido de un archivo JAR sin "desacoplar" (extraer) los archivos, utilizamos el comando comando

jar tvf jarfilename.jar

donde el argumento de línea de comandos t indica que deseamos ver una tabla de contenidos para el archivo JAR nombrado, por ejemplo:

jar tvf MiJar.jar

Uso del bytecode contenido en un archivo JAR

Para informar a la JVM de que queremos utilizar el bytecode contenido en un archivo JAR, utilizamos una opción de línea de comandos para establecer una variable de entorno llamada CLASSPATH de la siguiente manera:

java –cp path_to_jar_file class_containing_main_method

Por ejemplo, si almacenamos nuestro archivo MyJar.jar en un directorio compartido con el nombre de S:\applications,

entonces para ejecutar el programa MyApp que se almacena dentro de ese archivo JAR, escribiríamos

java –cp S:\applications\MyJar.jar MyApp

Extracción del contenido de un archivo JAR

Tenga en cuenta que no es necesario extraer el código de bytes de un archivo JAR con el fin de utilizarlo; la JVM es capaz de recuperar archivos bytecode individuales desde dentro de los archivos JAR según sea necesario. Sin embargo, si alguna vez Sin embargo, si alguna vez deseamos extraer archivos seleccionados de un archivo JAR, por ejemplo, si se incluyeron archivos de código fuente de Java y deseamos trabajar con archivos de código fuente individuales y quisiéramos trabajar con archivos fuente individuales, escribiríamos el comando

jar xvf jarfilename.jar space_separated_list_of_files_to_be_extracted

por ejemplo

jar xvf MiJar.jar Student.java Professor.java

"Jarring " Jerarquías de directorios completas

Es posible incorporar el contenido de toda una jerarquía de directorios (todas las subcarpetas) en un único archivo JAR mediante el comando

jar cvf jarFileName topLevelDirectoryName

Por ejemplo, los ejemplos de código SRS que acompañan a este libro están almacenados dentro de una jerarquía de directorios en UN ordenador bajo un directorio padre llamado

C:\My Documents\BJO Second Edition\Code

Para crear un archivo JAR que contenga todo el código, escribiría el comando

jar cvf BJOcode.jar "C:\My Documents\BJO Second Edition\Code"

Los comentarios de documentación de Java, también conocidos como comentarios de javadoc, son un tipo especial de comentario que permite generar automáticamente documentación HTML para una aplicación. Estos comentarios comienzan con una barra seguida de dos asteriscos (/**) y terminan con un asterisco seguido de barra (*/). Dentro del cuerpo de un comentario de javadoc, podemos usar una serie de etiquetas predefinidas de javadoc (cuyos nombres comienzan con @) para controlar el aspecto del HTML resultante.

Aquí tienes una clase simple Person que incorpora comentarios de javadoc:

```
// Person.java
/**
* Una persona es un ser humano. Podemos usar una Persona para representar un
estudiante
* o un profesor en un entorno académico.
*/
public class Person {
//------------
// Atributos.
//------------
/**
* El nombre legal de una persona. Normalmente representado como
* "Nombre Apellido".
*/
public String name;
/**
* La edad de una persona en años. No importa cuán inminente sea el próximo
cumpleaños de una persona,
* la edad de la persona siempre reflejará cuántos años tenía en su último cumpleaños.
*/
private int age;
//-------------
// Constructor.
//-------------
/**
* Este constructor inicializa los atributos nombre y edad.
* @param n El nombre de la Persona, en orden nombre - inicial del segundo nombre -
* apellido.
* @param a La edad de la Persona.
*/
public Person(String n, int a) {
name = n;
age = a;
}
/**
* Este método se utiliza para determinar la edad de una persona en años de perro.
*/
public double dogYears() {
return age/7.0;
```

```
}
}
```

Las características públicas de una clase se incluyen automáticamente en la documentación generada por javadoc. Sin embargo, las características privadas no se incluyen por defecto. Aunque hayamos documentado el atributo privado 'age' en estilo Javadoc, este no se reflejará en la documentación HTML resultante.

La etiqueta @param es específica de Javadoc y se utiliza para definir el propósito de un parámetro particular de un método. La sintaxis general es @param nombreDelParámetro descripción.

Las líneas en blanco o los comentarios que no son de Javadoc no tienen impacto en la generación de la documentación con Javadoc.

Para generar la documentación HTML para una clase, se utiliza la utilidad de línea de comandos llamada 'javadoc', que viene incluida con el Kit de Desarrollo de Java (JDK). Por ejemplo, el comando javadoc Person.java generará la documentación para una sola clase. Si se desea generar documentación para varios archivos .java al mismo tiempo, se puede utilizar javadoc *.java.

La salida de este comando javadoc incluirá varios archivos HTML, que constituyen la documentación para la clase y sus miembros. Estos archivos incluirán resúmenes de campos, constructores y métodos, así como detalles adicionales sobre cada uno de ellos. También se generará un índice que facilita la navegación por la documentación.

Para ver la documentación resultante, puedes cargar el archivo index.html en un navegador web. Este archivo actúa como la "página de inicio" de nuestra documentación.

Aquí hay una exploración de la página:

- En la parte superior de la página se muestra la jerarquía de herencia a la que pertenece la clase Person (en este caso, se muestra que Person deriva directamente de la clase Object del paquete java.lang).
- A continuación, se encuentra la descripción narrativa de la clase a partir del comentario de javadoc que precede a la declaración public class Person { ... }.
- Al desplazarse hacia abajo en la página, se encuentran listas de todos los atributos públicos, constructores y métodos pertenecientes a esta clase bajo los encabezados Resumen de campo, Resumen de constructor y Resumen de método, respectivamente. Dado que fue declarado como un atributo privado, se omite de forma predeterminada. Para incluir todas las características en la documentación generada, ya sean públicas o no, simplemente incluye la bandera -private en el comando javadoc:

```
javadoc -private Person.java
```

- Al continuar desplazándose hacia abajo en la página, se pueden ver detalles adicionales sobre los campos (atributos), constructores y métodos.

- Al hacer clic en el enlace Índice en la parte superior de la página, se accede a una vista alternativa de la documentación, donde se pueden navegar a través de una lista alfabética de todos los nombres de clases, atributos, constructores y métodos.

13.5.1
Operations on strings

Hay varias operaciones que podemos hacer sobre un string para hallar algunos resultados como:

-**(+)** para concatenar cadenas de strings

-**variable.length()** da la longitud de el string

Dígamos que s = “HOLA”
-**s.startsWith(“HO”)** devuelve true si s comienza con “HO” en este caso

-**s.endsWith(“A”)** devuelve un true si s termina con “A” en este caso

-**s.contains(“OL”)** devuelve un true si s contiene la cadena dada osea “OL” en este caso

-**s.indexOf(“H”)** retorna valores enteros, de tal manera que si lo que entra como parametro esta dentro de el string, entonces le asigna un valor de indice a ese parametro, y si no esta le asigna un valor negativo, ejemplo:

s.indexOf(“H”);---------->> 0
s.indexOf(“OLA”);---------->> 2 aqui pasa eso por que solo se podria dividir el string en

[H, OLA]

s.indexOf(“OA”);---------->> -1

-s.replace(old,new) reemplaza todos los caracteres dados como parametro “old” que sean iguales por el nuevo “new”

String s = “01020304”;
String p = s.replace(‘0’,’x’);
esto dara —------> “x1x2x3x4”

-substring(int i) crea un nuevo objeto que contiene un substring comenzando desde la posicion “i” hasta el final de el string

String s = ”foobar”;
String p = s.substring(3); —------> p = “bar”

-substring(int i , int j) igual que el anterior, pero va desde la posicion “i” hasta antes de la posicion “j”

String s = ”foobar”;
String p = s.substring(1,5); —------> p = “ooba”
-charAt(int index) retorna el caracter en la posicion dada como int index

-equals(String) booleano que dice si dos cadenas si son iguales o no

String s = “dog”
String p = “cat”

s.equals(p) —--> false hay que usar esto si queremos comparar strings, evitar usar “==”

The StringBuffer Class

se importa asi: import java.util.StringBuffer;

StringBuffer x = new StringBuffer();

Esta clase es como la de String, pero mas flexible ya que podemos modificar el contenido sin necesidad de crear un objeto nuevo, como tendriamos que hace con String, tambien tiene varios metodos para poder ser modificada, como append(), insert(), delete() y cosas asi, uno de estos metodos es importante como el toString() que permite convertir todo su contenido en una forma de String

7) 13.5.4 The StringTokenizer Class - 13.5.6 Testing the Equality of Strings [Pag. 462-470]

la clase StringTokenizer

ejemplo:

con esta clase se puede partir un string en subcadenas basados en limitaciones arbitrarias

```
String s = "This is a test.";
StringTokenizer st = new StringTokenizer(s);

while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

Here's the output:

```
This
is
a
test.
```

- se usa el método boolean hasMoreTokens de StringTokenizer, este método retorna true si hay más subcadenas o falso si ya se recorrió todo el string

- el constructor de String Tokenizer toma un argumento y parte por los espacios en blanco a la cadena dada
- el método de String nextToken se usa para extraer la próxima subcadena

una segunda forma del constructor es StringTokenizer(String s, String delimiter) si se quiere especificar un delimitador para partir la cadena, ejemplo:

```
String date = "11/17/1985";

// Note use of double quotes below.
StringTokenizer st = new StringTokenizer(date, "/");

while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

Here's the output:

```
11
17
1985
```

Creación de instancias de cadenas y el grupo de literales de cadenas

Hay dos maneras de instanciar un string

método abreviado:

```
String s = "I am a String!";
```

método formal:

```
String t = new String("I am a String, too!");
```

Al crear una cadena con el método abreviado, se verifica en el grupo de literales de cadenas si ya existe. Si no existe, se crea un nuevo objeto; en cambio, si ya existe, se asignan los valores del objeto existente.

Cuando se crea un objeto String de manera formal, siempre se generará un nuevo objeto. Si se crea otro objeto con el mismo valor, en lugar de asignar el mismo objeto, se creará uno nuevo.

Probando la equivalencia de Strings

para comparar string se pueden usar dos operadores

- == la comparación será verdadera si ambas variables hacen referencia al mismo objeto string en específico. Será falso si, a pesar de tener el mismo string hacen referencia a objetos string diferentes

- .equals() , esta operación será verdadera si los objetos string, a pesar de ser diferentes, tienen guardada la misma cadena string

  por esta razon, al comparar strings, se recomienda usar en la mayoría de las veces .equals()

Message Chains (13.6):

En Java y otros lenguajes de programación orientados a objetos, es común construir expresiones complejas mediante la concatenación y a veces anidación de llamadas a métodos. En este ejemplo, se crean instancias de las clases Student, Professor y Department, y se establecen diversos valores de atributos. La expresión a evaluar es:

s.setMajor(s.getAdvisor().getDepartment().getName());

Se realiza la evaluación de la expresión de izquierda a derecha. Primero, s.getAdvisor() devuelve una referencia a un objeto Professor, simplificando la expresión a s.setMajor(p.getDepartment().getName()). Luego, se aplica getDepartment() a este Professor, obteniendo una referencia a un objeto Department y simplificando la expresión a s.setMajor(d.getName()). Finalmente, se aplica getName() a este Department, obteniendo una referencia a un objeto String con el valor "Math". La expresión se simplifica aún más a s.setMajor("Math"), y finalmente, se asigna el valor "Math" al campo major del objeto Student s.

Determing the Class That an Objects Belongs To (13.11.1):

En Java se puede determinar la clase a la que pertenece un objeto utilizando el método getClass() heredado de la clase Object. Este método devuelve un objeto de tipo Class que representa la clase a la que pertenece el objeto.

Además, la clase Class tiene un método llamado getName() que devuelve el nombre completo de la clase. Combinando ambos métodos, se puede obtener el nombre de la clase a partir de un objeto con la siguiente expresión: reference.getClass().getName().

Por ejemplo, si tenemos un objeto Professor pr = new Professor();, podemos obtener el nombre de la clase utilizando pr.getClass().getName(), que devolverá "Professor".

También se puede utilizar la expresión instanceof para verificar si un objeto pertenece a una clase en particular. Por ejemplo:

if (x instanceof Student) {

```
  System.out.println("x es un estudiante");
}
if (x instanceof Professor) {
  System.out.println("x es un profesor");
}
if (x instanceof Person) {
  System.out.println("x es una persona");
}
```

Testing the Equality of Objects (13.11.2):

Dos referencias de objetos son consideradas iguales si apuntan exactamente al mismo objeto en memoria, lo cual se puede verificar utilizando el operador ==. También se puede usar el método equals(), heredado de la clase Object, para comparar la igualdad de objetos. Sin embargo, es importante destacar que el comportamiento de equals() puede variar según la implementación de la clase.

En el caso de las cadenas (String), el método equals() compara los valores de las cadenas en lugar de las identidades de los objetos, lo que significa que dos cadenas con el mismo valor serán consideradas iguales, incluso si se crean como instancias separadas.

En general, se recomienda que las clases personalizadas implementen su propio método equals() para definir cómo se debe realizar la comparación de igualdad para objetos de esa clase.

13.11.3 Overriding the equals Method

En esta sección se aborda la importancia de sobrescribir el método 'equals' en clases personalizadas para realizar comparaciones relevantes entre objetos. Se presenta un ejemplo con la clase Person, donde se sobrescribe el método para comparar los valores del atributo. Al intentar comparar dos objetos, se muestra cómo se maneja la excepción al intentar convertir referencias de objetos. El código de ejemplo ilustra la implementación del método en la clase para determinar la igualdad basada en el número de seguro social (ssn).

```
public class Person {
    private String ssn;
    private String name;
    // etc.

    // Constructor.
    public Person(String s, String n) {
        this.setSsn(s);
        this.setName(n);
    }

    public String getSsn() {
        return ssn;
    }

    // etc.

    // Overriding the equals method that we inherited from the Object class.
    public boolean equals(Object o) {
        boolean isEqual;

        // Try to cast the Object reference into a Person reference.
        // If this fails, we'll get a ClassCastException.
        try {
            Person p = (Person) o;

            // If we make it to this point in the code, we know we're
            // dealing with a Person object; next, we'll compare ssn's.
            if (this.getSsn().equals(p.getSsn())) {
                // We'll deem p equal to THIS person.
                isEqual = true;
            }
            else {
                isEqual = false;
            }
        }
        catch (ClassCastException e) {
            // They're not equal - o isn't even a Person!
            isEqual = false;
        }

        return isEqual;
    }
}
```

13.11.4 Overriding the toString Method

En esta sección se explora la personalización de la representación de objetos al imprimirlos mediante la sobrescritura del método . Se muestra un ejemplo con la clase , donde se sobrescribe el método para imprimir el nombre y el número de seguro social del estudiante. Se destaca que el método heredado de la clase imprime la clase y el ID interno del objeto, y al sobrescribirlo, se puede definir una representación personalizada del objeto al imprimirlo.

```
Student s = new Student("Harvey", "123-45-6789");

// Try to print the object reference directly.
System.out.println(s);
```

De esta instrucción obtendremos

```
Student@71f71130
```

donde Estudiante@71f71130 representa un ID de objeto interno relevante sólo para la JVM.

Podemos solucionarlo sobreescribiendo el método toString de la clase Student para definir lo que que deseamos imprimir como representación de un Student. Por ejemplo, la clase Student puede sobrescribir el método toString como sigue:

```
public String toString() {
  return this.getName() + " (" + this.getSsn() + ")");
}
```

Volviendo a ejecutar la primera instrucción, obtendríamos:

```
Harvey (123-45-6789)
```