

CAPÍTULO 6

1 / 1

» Cadenas E/S, Formateo y Análisis

La Clase String

El concepto clave para entender esto, es que una vez que se crea un objeto String, nunca se puede cambiar.

Las Cadenas son objetos inmutables

Manejar "cadenas" de caracteres es un aspecto fundamental de la mayoría de lenguajes de programación.

En Java cada carácter de una cadena es un carácter Unicode de 16 bits. Debido a que los caracteres Unicode son de 16 bits (no los escasos 7 o 8 bits que proporciona ASCII), un rico conjunto internacional de caracteres es fácilmente representado en Unicode.

En Java, las cadenas son objetos que pueden crear una instancia de una cadena con la nueva palabra clave, de la siguiente manera:

```
String s = new String();
```

Esta línea de código crea un nuevo objeto de clase String y la asigna a variables de referencia. Hasta ahora, los objetos String se parecen a otros objetos.

Ahora vamos a darle a la Cadena un valor:

```
s = "abcdef";
```

Como era de esperar, la clase String tiene alrededor de un trillón de constructores, por lo que se puede usar un atajo más eficiente:

```
String s = new String("abcdef");
```

Y como sólo se utilizan cadenas, incluso se puede de la siguiente manera:

```
String s = "abcdef";
```

Cada ejemplo crea un nuevo objeto String con un valor de "abcdef" y lo asignan a una variable de referencia s. Ahora si queremos referenciar al objeto String al que hace referencia s:

```
String s2 = s; //refer s2 to the same
```

Una vez que se le ha asignado un valor a una cadena, ese valor nunca puede cambiar; es immutable, sólido, congelado, no se mueve, fini, hecho.

La buena noticia es que si bien el objeto String es immutable, su variable de referencia no lo es, así que veamos el siguiente ejemplo:

```
s = s.concat (" more stuff"); //the concat() method  
'appends'  
// a literal to the end
```

La máquina virtual tomó el valor de String s y agregó "más cosas" al final.

Dado que Strings son inmutables, la máquina virtual no podría incluir este nuevo valor en la cadena anterior a la que se hace referencia por s, por lo que creó un nuevo objeto String, le dio el valor "abcdef*más cosas" e hizo s referirse a él.

Datos importantes sobre las cadenas y la memoria

Uno de los objetivos clave de cualquier buen lenguaje de programación es hacer un uso eficiente de memoria. A medida que las aplicaciones crecen, es muy común que los literales de cadena ocupen grandes cantidades de memoria de un programa, y a menudo hay mucha redundancia dentro del universo de literales String para un programa. Para hacer que Java sea más eficiente en la memoria, JVM reserva un área especial de memoria llamada "Grupo de constantes de cadena". Cuando el compilador encuentra un literal String idéntico ya existe, si se encuentra una coincidencia, la referencia al nuevo literal se dirige al String existente, y no se crea un nuevo objeto literal String.

Si varias variables de referencia se refieren a la misma cadena sin saberlo, sería muy malo si alguno de ellos pudiera cambiar el valor de la cadena.

Nadie puede anular el comportamiento de cualquier de los métodos String, por lo que puede estar seguro de que String los objetos con los que está contando que serán inmutables serán, de hecho, inmutables.

Crear Nuevas Cadenas

Veamos un par de ejemplos de cómo una cadena podría crearse, y supongamos además que no existen otros objetos String en la piscina:

String s = "abc" //Creates one string object and one
//reference variable

En este caso simple, "abc" iría al grupo y s se referiría a él.

String s = new String("abc"); //creates two objects
//and one reference
//variable

En este caso, debido a que usamos la nueva palabra clave, Java crearía un nuevo objeto String en la memoria normal (no agrupada), y s se referiría a él. Además, el literal "abc" se colocaría en la piscina.

Métodos importantes en la clase String

Los siguientes métodos son algunos de los métodos más utilizados en la clase String.

- * **charAt()** Devuelve el carácter ubicado en el índice especificado.
- * **concat()** Agrega una cadena al final de otra ("+" también funciona)
- * **equalsIgnoreCase()** Determina la igualdad de dos cadenas, ignorando el caso
- * **length()** Devuelve el número de caracteres de una cadena.
- * **replace()** Reemplaza las ocurrencias de un personaje con un nuevo personaje

- * **substring()** Devuelve una parte de una cadena.
- * **toLowerCase(CASEI)** Devuelve una cadena con caracteres en mayúscula convertidos.
- * **toString()** Devuelve el valor de una cadena.
- * **toUpperCase(CASEI)** Devuelve una cadena con caracteres en minúscula convertidos.
- * **trim()** Elimina los espacios en blanco de los extremos de una cadena.

public char charAt(int index)

Este método devuelve el carácter ubicado en el índice especificado de la cadena.

Los índices de cadenas se basan en cero, por ejemplo,

```
String x = "airplane";
System.out.println(x.charAt(2)); // output is 'i'
```

public String concat(String s)

Este método devuelve una cadena con el valor del String pasado al método adjunto al final del String utilizado para invocar el método, por ejemplo,

```
String x = "taxi";
System.out.println(x.concat(" cab")); // output is
// "taxi cab"
```

Los operadores + y += sobrecargados realizan funciones similares a concat(), el método, por ejemplo

```
String x = "libravu";
System.out.println(x + " card"); // output is "libravu card"
```

```
String x = "Atlantic";
x += " ocean";
System.out.println(x); // output is "Atlantic ocean"
```

El operador `+=` es un operador de asignación, por lo que la línea 2 es realmente creando una nueva cadena "Océano Atlántico" y asignándola a la variable `x`. Después la linea 2 se ejecuta, la cadena `x` original a la que se refería, "Atlántico" se abandona.

public boolean equalsIgnoreCase (String s)

Este método devuelve un valor booleano (verdadero o falso) dependiendo de si el valor de String en el argumento es el mismo que el valor de String usado para invocar el método.

Este método devolverá verdadero incluso cuando los caracteres de los objetos String que se comparan tengan casos diferentes, por ejemplo:

```
String x = "Exit";
System.out.println(x.equalsIgnoreCase("EXIT"));
                    // is "true"
System.out.println(x.equalsIgnoreCase("fixe"));
                    // is "false"
```

`public int length()`

Este método devuelve la longitud de la cadena utilizada para invocar el método, por ejemplo:

```
String x = "01234567"
```

```
System.out.println(x.length()); // returns "8"
```

`public String replace (char old, char new)`

Este método devuelve un String cuyo valor es el del String utilizado para invocar el método, actualizado para que cualquier ocurrencia del char en el primer argumento sea reemplazada por el char en el segundo argumento, por ejemplo:

```
String x = "oxoxoxoxox";
```

```
System.out.println(x.replace('x', 'x')); // output is  
// "oxoxoxoxox"
```

`public String substring (int begin)`

`public String substring (int begin, int end)`

El método `substring()` se utiliza para devolver una parte (o subcadena) de la cadena utilizada para invocar el método. El primer argumento representa la ubicación inicial (basada en cero) de la subcadena devuelta incluyendo los caracteres hasta el final de la cadena original. Si la llamada tiene dos argumentos, la subcadena devuelta terminaría con el carácter ubicado en la enésima posición de la cadena original, donde n es el segundo argumento. Desafortunadamente, el argumento final es 7, el último carácter de la cadena devuelta estaba en la posición 7 de la

cadena original, que es el índice 6 (auch).
Veamos algunos ejemplos

```
String x = "0123456789"; //as if by magic, the value  
//of each char  
//is the same as its index!
```

```
System.out.println(x.substring(5)); //output is "56789"
```

```
System.out.println(x.substring(5, 8)); //output is "567"
```

El primer ejemplo debería ser fácil:
comienza en el índice 5 y devuelve el resto de la
cadena. En el segundo ejemplo se debe de leer de la
siguiente manera: comienza en el índice 5 y regresa
los caracteres hasta la octava posición inclusive la
misma octava posición (índice 7).

public String toLower case()

Este método devuelve una cadena cuyo valor es la
cadena utilizada para invocar al método, pero con
los caracteres en mayúscula convertidos a minúsculas,
por ejemplo:

```
String x = "A new moon";
```

```
System.out.println(x.toLowerCase()); //output is  
// "a new moon"
```

public String toString()

Este método devuelve el valor de la cadena utiliza-
da para invocar el método. ¿Qué? Por qué necesi-
taria un método aparentemente de "no hacer nada"?

Todos los objetos en Java deben tener un método `toString()`, que normalmente devuelve un `String` que de alguna manera significativa describe el objeto en cuestión. En el caso de un objeto `String`, ¿qué forma más significativa que el valor de `String`? En aras de la coherencia, aquí hay un ejemplo:

```
String x = "big surprise";
System.out.println(x.toString()); // output -
// reader's exercise
```

`public String toUpperCase()`

Este método devuelve una cadena cuyo valor es la cadena utilizada para invocar el método, pero con los caracteres en minúscula convertidos a mayúsculas, por ejemplo:

```
String x = "A New Moon";
System.out.println(x.toUpperCase()); // output is
// "A NEW MOON"
```

`public String trim()`

Este método devuelve una cadena cuyo valor es la cadena utilizada para invocar el método, pero con los espacios en blanco o espacios iniciales o finales eliminados, por ejemplo:

```
String x = "    hi    ";
System.out.println(x + "x"); // result is "    hi x"
System.out.println(x.trim() + "x"); // result is "hi x"
```

The StringBuffer and StringBuilder classes

Las clases `java.lang.StringBuffer` y `java.lang.StringBuilder` deben usarse cuando tenga que hacer muchas modificaciones en cadenas de caracteres.

Los objetos `String` son inmutables, por lo que si elige hacer muchas manipulaciones con objetos `String`, terminaría con muchos objetos `String` abandonados en el grupo `String`. (Incluso en estos días de gigabytes de RAM, no es una buena idea desperdiciar memoria valiosa en objetos `String` pool descartados).

Por otro lado, los objetos de tipo `StringBuffer` y `StringBuilder` se pueden modificar una u otra vez sin dejar una gran efuencia de objetos `String` descartados.

`StringBuffer` vs `StringBuilder`

La clase `StringBuilder` se agregó en Java 5. Tiene exactamente la misma API que la clase `StringBuffer`, excepto que `StringBuilder` no es seguro para subprocesos.

En otras palabras, sus métodos no están sincronizados.

Se recomienda que utilice `StringBuilder` en lugar de `StringBuffer` siempre que sea posible porque `StringBuilder` se ejecutaba más rápido (y tal vez salte más alto). Entonces, aparte de la sincronización, todo lo que digamos sobre los métodos de `StringBuilder` es válido para los métodos de `StringBuffer` y viceversa.

CAPÍTULO 7

1 1

» Genéricos y Colecciones

» Generics and Collections

El método `toString()`

Anule `toString()` cuando desee que un simple mortal pueda leer algo significativo sobre los objetos de su clase. El código puede llamar a `toString()` en su objeto cuando quiere leer detalles útiles sobre su objeto. Cuando pasa una referencia de objeto al método `System.out.println()`, por ejemplo, se llama al método `toString()` del objeto y se devuelve `toString()`, se muestra en el siguiente ejemplo:

```
public class HardToRead {  
    public static void main (String [] args) {  
        HardToRead h = new HardToRead ();  
        System.out.println (h);  
    }  
}
```

Ejecutar la clase `HardToRead` nos blinda lo encantador u significativo,

```
% java HardToRead  
HardToRead@ a47c0
```

El resultado anterior es lo que obtiene cuando no anula el método `toString()` de la clase `Object`. Le da el nombre de la clase (al menos eso es significativo) seguido del símbolo `@`, seguido de

la representación hexadecimal sin firmar del código hash del objeto.

METHOD	DESCRIPTION
boolean equals(Object obj)	Decide si dos objetos son significativamente equivalentes.
void finalize()	Conocido como recolector de basura cuando el recolector de basura ve que no se puede hacer referencia al objeto.
int hashCode()	Devuelve un valor int de código hash para un objeto, de modo que el objeto se pueda usar en clases de colección que usan hash, incluidos Hashtable, HashMap y HashSet.
final void notify()	Despierta un hilo que está esperando el bloqueo de este objeto.
final void notifyAll()	Despierta todos los subprocesos que esperan el bloqueo de este objeto.
final void wait()	Hace que el subproceso actual espere hasta que otro subproceso llame a notify() o notifyAll() en este objeto.

METHOD	DESCRIPTION
String toString()	Devuelve una "representación de texto" del objeto.

Overriding equals()

Discutimos cómo comparar dos referencias de objeto usando el operador == que evalúa como verdadero solo cuando ambas referencias se refieren al mismo objeto (porque == simplemente mira los bits en la variable, y si son idénticas o no).

Sabemos que la clase String y las clases contenidas han anulado el método equals() (heredado de la clase Object), de modo que se puede comparar dos objetos diferentes pero del mismo tipo para ver si sus contenidos son significativamente equivalentes.

Si dos instancias de Integer diferentes tienen el valor 5, en lo que a nosotros respecta, son iguales. El hecho de que el valor 5 viva en dos objetos separados no importa.

Cuando realmente necesitamos saber si dos referencias son idénticas, use ==. Pero cuando necesitamos saber si dos objetos en sí (no las referencias) son iguales, use el método equals(). Para cada clase que escriba, debe decidir si tiene sentido considerar dos instancias diferentes iguales. Para algunas clases, puede decidir que dos objetos nunca pueden ser iguales. Por ejemplo, imagina un auto de clase que tiene variables de instancia para cosas como marca, modelo, año, configuración;

Ciertamente no quiere que tu auto de repente sea tratado como el mismo auto que alguien con un auto que tiene atributos idénticos. Su automóvil es su automóvil y no quiere que su vecino Billy conduzca en él solo porque, "que, en realidad es el mismo automóvil", el método `equals()` lo dice". Así que no hay dos coches que se consideren exactamente iguales. Si dos referencias se refieren a un automóvil, entonces sabe que ambos se refieren a un automóvil, no a dos automóviles que tienen los mismos atributos. Por lo tanto, en el caso de un automóvil, es posible que nunca necesite, o desee, anular el método `equals()`. Por supuesto, sabes que ese no es el final de la historia.

What it means if you don't override `equals()`

Hay una limitación potencial: si no se anula el método `equals()` de una clase, no se podrán usar esos objetos como una clave en una tabla hash y probablemente no obtendrá conjuntos precisos, tales que hay sin duplicados conceptuales.

El método `equals()` en la clase `Object` usa solo el operador `=` para las comparaciones, por lo que a menos que anule `equals()`, dos objetos se consideran iguales solo si las dos referencias se refieren al mismo objeto. Si desea que los objetos de su clase se utilicen como claves para una tabla hash (o como elementos en cualquier estructura de datos que utilice la equivalencia para buscar y/o recuperar un objeto), entonces debe anular `equals()` para que dos instancias diferentes puedan considerarse iguales.

Implementación de un método equals()

Digamos que decide anular equals() en su clase. Podría verse así:

```
public class EqualsTest {  
    public static void main (String [] args) {  
        Moof one = new Moof (8);  
        Moof two = new Moof (8);  
        if (one.equals (two)) {  
            System.out.println ("one and two are equal");  
        }  
    }  
}  
  
class Moof {  
    private int moofValue;  
    Moof (int val) {  
        moofValue = val;  
    }  
    public int getMoofValue () {  
        return moofValue;  
    }  
    public boolean equals (Object o) {  
        if ((o instanceof Moof) && ((Moof)o).getMoofValue()  
            == this.moofValue) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

En el método main() de EqualsTest, creamos dos instancias de Moof, pasando el mismo valor 8 al constructor de Moof. La clase Moof lo que hace con ese argumento de constructor es asignar el valor a la variable de instancia moofValue.

El contrato de equals()

The equals() Contract

Extraído directamente de los documentos de Java, el contrato equals() dice:

- Es **reflexivo**. Para cualquier valor de referencia x, x.equals(x) debería devolver verdadero.
- Es **simétrico**. Para cualquier valor de referencia x y y, x.equals(y) debe devolver verdadero si y solo si y.equals(x) devuelve verdadero.
- Es **transitivo**. Para cualquier valor de referencia, x y y z, si x.equals(y) devuelve verdadero y y.equals(z) devuelve verdadero, entonces x.equals(z) debe devolver verdadero.
- Es **consistente**. Para cualquier valor de referencia x y y, múltiples invocaciones de x.equals(y) devuelven constantemente verdadero o falso, siempre que no se modifique la información utilizada en las comparaciones de iguales en el objeto.
- Para cualquier valor de referencia no nulo x, x.equals(null) debe devolver falso.

El método `equals()` y `hashCode()` están unidos por un contrato conjunto que especifica si dos objetos se consideran iguales usando el método `equals()`, entonces deben tener valores de código hash idénticos. Entonces, para estar realmente seguro, su regla general debería ser, si anula `equals()`, anular también `hashCode()`.

Overriding `hashCode()`

Los `hashCode` se utilizan normalmente para aumentar el rendimiento de grandes colecciones de datos. Algunas clases de colección utilizan el valor del código hash de un objeto (veremos las colecciones más adelante). Aunque puede considerarlo como una especie de número de identificación de objeto, no es necesariamente único. Las colecciones como `HashMap` y `HashSet` usan el valor del código hash de un objeto para determinar como se debe almacenar el objeto en la colección, y el código hash se usa nuevamente para ayudar a ubicar el objeto en la colección.

Es perfectamente legal tener un método de código hash terriblemente ineficiente en su clase, siempre que no viole el contrato especificado en la documentación de la clase `Object`. Entonces, por si se llega a pedir que se elija un uso apropiado o correcto de `hashCode`, no hay que confundir apropiado con legal o eficiente.

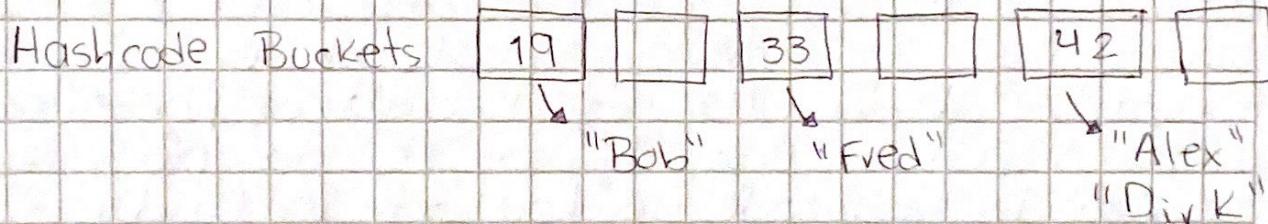
Entendiendo HashCodes

Para comprender que es apropiado y correcto, debemos observar como algunas de las colecciones usan códigos hash.

Imaginense un conjunto de cubos alineados en el suelo. Alguien le entrega una hoja de papel con un nombre. Toma el nombre y calcula un código entero a partir de él usando A es 1, B es 2, y así sucesivamente, y sumando los valores numéricos de todas las letras del nombre juntas. Un nombre de pila siempre resultaba en el mismo código, como la continuación:

Key	HashCode Algorithm	HashCode
Alex	A(1) + L(12) + E(5) + X(24)	= 42
Bob	B(2) + O(15) + B(2)	= 19
Dirk	D(4) + I(9) + R(18) + K(11)	= 42
Fred	F(6) + R(18) + E(5) + D(4)	= 33

HashMap Collection



No introducimos nada al azar, simplemente tenemos un algoritmo que siempre se ejecutará de la misma manera dada una entrada específica, por lo que la salida siempre será idéntica para dos entradas idénticas.

Implementando hashCode()

¿Cómo se ve un algoritmo de código hash real? Las personas obtienen su doctorado en algoritmos hash, por lo que, desde el punto de vista de las ciencias de la computación, está más allá del alcance del examen. La parte que nos importa aquí es la cuestión de si cumple con el contrato. Y para seguir el contrato, piense en lo que hace con el método equals(). Compára atributos. Porque esa comparación casi siempre involucra valores de variables de instancia.

La implementación hashCode() debe usar las mismas variables de instancia. Por ejemplo:

```
class HastHash {
    public int x;
    HastHash (int xVal) { x = xVal; }

    public boolean equals (Object o) {
        HastHash h = (HastHash) o; //Don't try at home
        //without instanceof test
        if (h.x == this.x) {
            return true;
        } else {
    }
```

```
    } return false;  
}  
public int hashCode() { return (x * 17); }  
}
```

Este método equals() dice que dos objetos son iguales si tienen el mismo valor x, por lo que los objetos con el mismo valor x tendrán que devolver códigos hash idénticos.

El contrato de hashCode() The hashCode Contract

- Siempre que se invoca en el mismo objeto más de una vez durante la ejecución de una aplicación Java, el método hashCode() debe devolver constantemente el mismo número entero, siempre que no se modifique la información utilizada en las comparaciones equals() del objeto. Este número entero no necesita permanecer consistente de una ejecución de una aplicación a otra ejecución de la misma aplicación.
- Si dos objetos son iguales según el método equals(Object), entonces llamar al método hashCode() en cada uno de los dos objetos debe producir el mismo resultado entero.