A: Paso y Retorno de Objetos

Hasta este momento el lector debería sentirse razonablemente cómodo con la idea de que cuando se está "pasando" un objeto se está pasando de hecho una referencia.

En muchos lenguajes de programación se puede usar la forma "normal" de pasar objetos, y la mayoría de veces funciona bien. Pero siempre parece que llega un momento en el que hay que hacer algo que se sale de la norma, y de repente todo se vuelve algo más complicado (o, en el caso de C++, bastante complicado). Java no es una excepción, y es importante que se entienda exactamente lo que ocurre al pasar y manipular objetos. Este apéndice pretende dar esta visión.

Otra forma de afrontar la cuestión de este apéndice, si se proviene de un lenguaje de programación bien equipado es, "¿tiene Java punteros?". Hay quien dice que los punteros son difíciles y peligrosos, y por consiguiente, malos, y dado que Java es todo virtud y bondad, tanto que te llevará al paraíso de la programación, no puede tener de esas cosas tan malas. Sin embargo, es más exacto decir que Java tiene punteros; de hecho, todo identificador en Java (excepto en el caso de los datos primitivos), es uno de esos punteros, pero su uso está restringido y reservado no sólo al compilador sino también al sistema de tiempo de ejecución. Hasta la fecha, yo les he llamado "referencias", y se puede pensar que son "punteros de seguridad", no muy distintos de las tijeras con punta roma de pre-escolar —no están afiladas, y por tanto no te puedes cortar sencillamente, pero pueden ser en ocasiones lentas y tediosas.

Pasando referencias

Cuando se pasa una referencia a un método, se sigue apuntando al mismo objeto. Un experimento simple puede demostrar esto:

```
//: apendicea:PasarReferencias.java
// Pasando referencias.

public class PasarReferencias {
  static void f(PasarReferencias h) {
    System.out.println("h dentro de f(): " + h);
  }
  public static void main(String[] args) {
    PasarReferencias p = new PasarReferencias();
    System.out.println("p dentro de main(): " + p);
    f(p);
  }
```

```
} ///:~
```

En las sentencias de impresión se invoca automáticamente al método **toString( )**, y **PasarReferencias** hereda directamente de **Object** sin redefinir **toString( )**. Por consiguiente, se usa la versión de **toString( )** de **Object**, que imprime la clase del objeto seguida de la dirección donde está localizado el mismo (no la referencia, sino el almacenamiento de objetos en sí). La salida tiene este aspecto:

```
p dentro de main(): PasarReferencias@1653748
h dentro de f(): PasarReferencias@1653748
```

Como puede verse, tanto **p** como **h** hacen referencia al mismo objeto. Esto es mucho más eficiente que duplicar un nuevo objeto **PasarReferencias** de forma que se pueda enviar un parámetro a un método. Pero trae un aspecto muy importante.

Uso de alias

El uso de alias significa que hay más de una referencia vinculada al mismo objeto, como en el ejemplo de arriba. El problema con el uso de alias radica en que alguien *escriba* en ese objeto. Si los propietarios de las referencias no esperan que el objeto varíe, se sorprenderán. Esto puede demostrarse con este sencillo ejemplo:

```
//: apendicea:Alias1.java
// Uso de alias: dos referencias al mismo objeto.

public class Alias1 {
  int i;
  Alias1(int ii) { i = ii; }
  public static void main(String[] args) {
    Alias1 x = new Alias1(7);
    Alias1 y = x; // Asignar la referencia
    System.out.println("x: " + x.i);
    System.out.println("y: " + y.i);
    System.out.println("Incrementando x");
    x.i++;
    System.out.println("x: " + x.i);
    System.out.println("y: " + y.i);
  }
} ///:~
```

En la línea:

```
Alias1 y = x; // Asignar la referencia
```

se crea una nueva referencia **Alias1**, pero en vez de ser asignada a un nuevo objeto creado con **new**, se asigna a una referencia existente. Por tanto, los contenidos de la referencia **x**, que es la dirección a la que apunta el objeto **x**, se asignan a **y**, con lo que tanto **x** como **y** están asignadas al mismo objeto. Por tanto, cuando en la sentencia:

```
x.i++;
```

se incrementa la **i** de **x**, también se verá afectada la **i** de **y**. Esto puede verse en la salida:

```
x: 7
y: 7
Incrementando x
x: 8
y: 8
```

Una buena solución en este caso es simplemente no hacerlo: no establecer conscientemente un alias, o más de una referencia a un objeto dentro de un mismo ámbito. El código será así más fácil de depurar y entender. Sin embargo, cuando se está pasando una referencia como un argumento —que es como Java se supone que funciona— se está generando automáticamente un alias porque la referencia local creada puede modificar el "objeto externo" (el objeto que se creó fuera del ámbito del método). He aquí un ejemplo:

```
//: apendicea:Alias2.java
// Las llamadas a metodos implican alias
// de sus parámetros.

public class Alias2 {
  int i;
  Alias2(int ii) { i = ii; }
  static void f(Alias2 referencia) {
    referencia.i++;
  }
  public static void main(String[] args) {
    Alias2 x = new Alias2(7);
    System.out.println("x: " + x.i);
    System.out.println("Invocando a f(x)");
    f(x);
    System.out.println("x: " + x.i);
  }
} ///:~
```

La salida es:

```
x: 7
Invocando a f(x)
x: 8
```

El método modifica su argumento, el objeto externo. Cuando se da este tipo de situación, hay que decidir si tiene sentido, si el usuario lo espera, y si va a causar o no problemas.

En general, se invoca a un método para producir un valor de retorno y/o un cambio de estado en el objeto *por el que se llama al método.* (Un método es el cómo se "envía un mensaje" a ese objeto.) Es menos común llamar a un método para que manipule sus argumentos; a esto se le denomina "lla-

mar a un método por sus *efectos laterales*". Por consiguiente, al crear un método que modifica sus parámetros, el usuario debe estar instruido y advertido sobre el uso de ese método y sus potenciales sorpresas. Debido a posibles confusiones y fallos, es mucho mejor evitar cambiar el parámetro.

Si hay que modificar un parámetro durante una llamada a un método y no se pretende modificar el parámetro externo, entonces se debería proteger ese parámetro haciendo una copia dentro del método. Sobre esto versará gran parte de este apéndice.

Haciendo copias locales

Para repasar: todo paso de parámetros en Java se lleva a cabo pasando referencias. Es decir, al pasar "un objeto", verdaderamente sólo se está pasando una referencia a un objeto que reside fuera del método, por lo que si se llevan a cabo modificaciones con esa referencia, se modifica el objeto externo. Además:

- Durante el paso de parámetros se produce automáticamente el uso de alias.
- No hay objetos locales, sólo referencias locales.
- Las referencias tienen ámbitos, los objetos no.
- La longevidad de los objetos nunca es un problema en Java.
- No hay soporte por parte del lenguaje (por ejemplo, la palabra clave "const") para evitar que se modifiquen los objetos (es decir, para evitar los aspectos negativos del uso de alias).

Si sólo se está leyendo información de un objeto sin modificarlo, el paso de una referencia es la forma más eficiente de paso de parámetros. Está bien que la forma de hacer las cosas por defecto sea la más eficiente. Sin embargo, en ocasiones es necesario ser capaz de tratar el objeto como si fuera "local", de forma que los cambios que se hagan sólo afecten a la copia local, sin modificar el objeto externo. Muchos lenguajes de programación soportan la habilidad de hacer automáticamente una copia local del objeto externo, dentro del método[1]. Java no lo hace, pero te permite producir este efecto.

Paso por valor

Así surge el aspecto de la terminología, que siempre suele ser adecuado. El término "pasar por valor" y su significado dependen de cómo se perciba el funcionamiento del programa. El significado general es que se logra una copia de sea lo que sea lo que se pasa, pero la pregunta real es cómo se piensa en lo que se pasa. Cuando se "pasa por valor", hay dos visiones claramente distintas:

---

[1] En C, que generalmente manipula pequeñas cantidades de datos, el paso por defecto es por valor. C++ tenía que seguir este esquema, pero con objetos, el *paso por valor* no suele ser la forma más eficiente. Además, la codificación de clases para dar soporte en C++ al paso por valor, no supone sino quebraderos de cabeza.

1. Java pasa todo por valor. Cuando se pasan datos primitivos a un método, se logra una copia aparte del dato. Cuando se pasa una referencia a un método, se obtiene una referencia al método, se puede lograr una copia de la referencia. Ergo, todo se pasa por valor. Por supuesto, se supone que siempre se piensa (y se tiene cuidado en qué) que se están pasando referencias, pero parece que el diseño de Java ha ido mucho más allá, permitiéndote ignorar (la mayoría de las veces) que se está trabajando con una referencia. Es decir, parece permitirte pensar en la referencia como si se tratara "del objeto" puesto que implícitamente se desreferencia cuando se hace una llamada a un método.

2. Java pasa los tipos primitivos de datos por valor (sin que haya parámetros), pero los objetos se pasan por referencia. Ésta es la visión de que la referencia es un alias del objeto, por lo que *no* se piensa en el paso de referencias, sino que se dice "Estoy pasando el objeto". Dado que no se logra una copia local del objeto, cuando se pasa a un método, claramente, los objetos no se pasan por valor. Parece haber cierto soporte para esta visión por parte de Sun, pues una de las palabras clave "no implementada pero reservada" era **byvalue**. (No se sabe, sin embargo, si esa palabra clave algún día llegará a ver la luz).

Habiendo visto ambas perspectivas, y tras decir que "depende de cómo vea cada uno lo que es una referencia", intentaré dejar este aspecto de lado. Al final, no es *tan* importante —lo que es importante es que se entienda que pasar una referencia permite que el objeto que hizo la llamada pueda cambiar de forma inesperada.

Clonando objetos

La razón más probable para hacer una copia local de un objeto es cuando se va a modificar este objeto y no se desea modificar el objeto llamador. Si se decide que se desea hacer una copia local, basta con usar el método **clone( )** para llevar a cabo la operación. Se trata de un método definido como **protected** en la clase base **Object**, y que hay que superponer como **public** en cualquier clase derivada que se desee clonar. Por ejemplo, la clase de biblioteca estándar **ArrayList** superpone **clone( )**, por lo que se puede llamar a **clone( )** para **ArrayList**:

```
//: apendicea:Clonar.java
// La operación clone() funciona sólo para unos pocos
// elementos en la biblioteca estándar de Java.
import java.util.*;

class Int {
  private int i;
  public Int(int ii) { i = ii; }
  public void incrementar() { i++; }
  public String toString() {
    return Integer.toString(i);
  }
}
```

```
public class Clonar {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++ )
      v.add(new Int(i));
    System.out.println("v: " + v);
    ArrayList v2 = (ArrayList)v.clone();
    // Incrementar todos los elementos de v2:
    for(Iterator e = v2.iterator();
        e.hasNext(); )
      ((Int)e.next()).incrementar();
    // Ver si los elementos de v han cambiado:
    System.out.println("v: " + v);
  }
} ///:~
```

El método **clone( )** produce un **Object**, que hay que convertir al método apropiado. El ejemplo muestra cómo el método **clone( )** de **ArrayList** *no* intenta clonar automáticamente todos los objetos contenidos en el **ArrayList** —el viejo **ArrayList** y el **ArrayList** clonado son alias del mismo objeto. A esto se le suele llamar hacer una *copia superficial*, puesto que sólo se copia la porción de "superficie" del objeto. El objeto en sí consiste en esta "superficie" más todos los objetos a los que apunten las referencias, más todos los objetos a los que *esos* objetos apuntan, etc. A esto se le suele llamar la "telaraña de objetos". Copiar absolutamente todo se llama hacer una *copia en profundidad.*

El efecto de la copia superficial puede verse en la salida, donde las acciones hechas sobre **v2** afectan a **v**:

```
v:  [0,  1,  2,  3,  4,  5,  6,  7,  8,  9]
v:  [1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
```

Ahora, intentar **clone( )** (clonar) los objetos del **ArrayList** es probablemente una simple presunción, puesto que no hay ninguna garantía de que estos objetos sean "clonables"[2].

Añadiendo a una clase la capacidad de ser clonable

Incluso aunque el método para clonar está definido en la clase **Object**, base de todas las clases, clonar *no* es algo que esté disponible automáticamente para todas las clases[3]. Esto parecería contra-intuitivo con la idea de que los métodos de la clase base siempre están disponibles para sus clases derivadas. En Java, clonar va contra esta idea; si se desea que exista para una clase, hay que añadir código de forma específica, para que la clonación sea posible.

---

[2] Ésta no es la palabra conforme aparece en el diccionario, pero es la usada en la librería Java, así que es la usada aquí con la esperanza de reducir la confusión.

[3] Aparentemente, se puede crear un ejemplo simple para esta sentencia, así (ver continuación de la nota en la página siguiente):

Usando un truco con **protected**

Para evitar tener la capacidad de clonar por defecto toda clase que se cree, el método **clone( )** es **protected** y pertenece a la clase base **Object**. Esto no sólo significa que no está disponible por defecto para el programador cliente que simplemente use la clase (aquél que no genera subclases de las mismas), sino que también significa que no se puede llamar a **clone( )** vía referencia a la clase base. (Aunque eso podría parecer útil en algunas situaciones, como pudiera ser clonar de forma polimórfica un conjunto de **Object**s.) Es, en efecto, una forma de proporcionar al programador, en tiempo de compilación, la información de que el objeto no es clonable —y de hecho, la mayoría de las clases de la biblioteca estándar de Java no son clonables. Por tanto, si se dice:

```
Integer x = new Integer(1);
x = x.clone();
```

En tiempo de compilación se obtendrá un mensaje de error que indica que **clone( )** no es accesible (puesto que **Integer** no lo superpone y por defecto se acude a la versión **protected**).

Si, sin embargo, se está en una clase derivada de **Object** (cosa que son todas las clases), se puede invocar a **Object.clone( )** puesto que es **protected** y la clase es una descendiente. La clase base **clone( )** tiene funcionalidad útil —lleva a cabo la duplicación del *objeto de la clase derivada*, actuando por consiguiente como la operación común de clonado. Sin embargo, hay que hacer **public** la operación de clonado de *cada uno* si se desea que ésta esté accesible. Por tanto, dos aspectos vitales al clonar son:

- Llamar casi siempre a **super.clone( )**
- Hacer **public** la función **clone** de cada uno.

Probablemente, se deseará superponer **clone( )** en subsiguientes clases derivadas, pues de otra forma se usará el nuevo (y ahora **public**) método **clone( )**, y eso podría no ser siempre lo correcto (aunque, puesto que **Object.clone( )** hace una copia del objeto, puede que sí). El truco **protected** sólo funciona una vez —la primera vez que se hereda de una clase que no es clonable y se desea hacer que sí lo sea. En cualquier clase heredada de la ya definida, estará disponible el método **clone( )** puesto que Java no puede reducir el acceso a métodos durante la derivación. Es decir, una

---

```
public class Cloneit implements Cloneable {
  public static void main (String [] args)
  throws CloneNotSupporteException {
    Cloneit a = new Cloneit();
    Cloneit b = (Cloneit)a.clone();
  }
}
```

Sin embargo, esto sólo funciona porque **main( )** es un método de **Cloneit** y, por consiguiente, tiene permiso para llamar al método **protected clone( )** de la clase base. Si se le llama desde una clase diferente, no compilará.

vez que una clase es clonable, todo lo que se derive de la misma también lo es, a no ser que se proporcionen mecanismos (descritos más adelante) para "desactivar" la "clonabilidad".

Implementando la interfaz Cloneable

Todavía se necesita algo más para completar la "clonabilidad" de un objeto: implementar la interfaz **Cloneable**. Esta **interfaz** es un poco extraña pues ¡está vacía!

```
interface Cloneable()
```

La razón de implementar esta interfaz vacía no es obviamente porque se vaya a aplicar un molde hacia arriba a **Cloneable** e invocar a uno de sus métodos. Aquí, el uso de la interfaz se considera como una especie de "intrusismo" pues usa una faceta para algo que no es su propósito original. Implementar la interfaz **Cloneable** actúa como indicador, vinculado al tipo de la clase.

Hay dos razones para la existencia de la interfaz **Cloneable**. En primer lugar, se podría tener una referencia convertida al tipo base sin saber si es posible hacer una clonación a ese objeto. En este caso, se puede usar la palabra clave **instanceof** (descrita en el Capítulo 12) para averiguar si la referencia está conectada a un objeto que puede clonarse:

```
if (miReferencia instanceof Cloneable) // ...
```

La segunda razón es que está en este diseño porque se pensaba que la "clonabilidad" probablemente no fuera deseable para todos los tipos de objetos. Por tanto, **Object.clone( )** verifica que una clase implemente la interfaz **Cloneable**. Si no, lanza una excepción **CloneNotSupportedException**. Por tanto, en general, uno se ve obligado a implementar **Cloneable** como parte del soporte a la clonación.

Clonación con éxito

Una vez comprendidos los detalles de implementación del método **clone( )**, ya se pueden crear clases que pueden duplicarse sencillamente para proporcionar una copia local:

```
//: apendicea:CopiaLocal.java
// Creando copias locales con clone().
import java.util.*;

class MiObjeto implements Cloneable {
  int i;
  MiObjeto(int ii) { i = ii; }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("MiObjeto no es clonable");
    }
```

```
    return o;
  }
  public String toString() {
    return Integer.toString(i);
  }
}

public class CopiaLocal {
  static MiObjeto g(MiObjeto v) {
    // El paso de una referencia modifica el objeto externo:
    v.i++;
    return v;
  }
  static MiObjeto f(MiObjeto v) {
    v = (MiObjeto)v.clone(); // Copia local
    v.i++;
    return v;
  }
  public static void main(String[] args) {
    MiObjeto a = new MiObjeto(11);
    MiObjeto b = g(a);
    // Probando la equivalencia de referencias, ,
    // no la equivalencia de objetos:
    if(a == b)
      System.out.println("a == b");
    else
      System.out.println("a != b");
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    MiObjeto c = new MiObjeto(47);
    MiObjeto d = f(c);
    if(c == d)
      System.out.println("c == d");
    else
      System.out.println("c != d");
    System.out.println("c = " + c);
    System.out.println("d = " + d);
  }
} ///:~
```

En primer lugar, **clone( )** debe ser accesible, por lo que hay que hacerlo **public**. En segundo lugar, para la parte inicial de la operación **clone( )** habría que invocar a la versión de **clone( )** de la clase base. El **clone( )** al que se está invocando aquí es el predefinido dentro de **Object**, y se puede invocar por ser **protected**, y por tanto, accesible en las clases derivadas.

El método **Object.clone( )** averigua lo grande que es el objeto, crea memoria suficiente para uno nuevo, y copia todos los bits del viejo al nuevo. A esto se le llama *copia bit a bit*, y es lo que generalmente se esperaría que hiciera un método **clone( )**. Pero antes de que **Object.clone( )** lleve a cabo sus operaciones, primero comprueba si una clase es **Cloneable** —es decir, si implementa o no la interfaz **Cloneable**. Si no lo hace, **Object.clone( )** lanza una **CloneNotSupportedException** para indicar que no se puede clonar. Por tanto, hay que llevar la llamada a **super.clone( )** con un bloque *try-catch*, que capture una excepción que nunca debería darse (por haber implementado la interfaz **Cloneable**).

En **CopiaLocal**, los dos métodos **g( )** y **f( )** demuestran la diferencia entre los dos enfoques en el paso de parámetros. El método **g( )** muestra el paso por referencia en el que se modifica el objeto exterior, devolviendo una referencia a ese objeto exterior, mientras que **f( )** clona el parámetro, desacoplándolo y dejando a salvo el objeto original. Después, puede hacer lo que desee, e incluso devolver una referencia al nuevo objeto sin crear efectos perniciosos al original. Nótese la sentencia, en cierta medida curiosa:

```
v = (MiObjeto)v.clone();
```

Es *aquí donde se crea la copia local*. Para evitar confusiones por sentencias así, recuérdese que este dialecto de codificación tan extraño está totalmente permitido en Java, puesto que todo identificador de objeto es de hecho una referencia al mismo. Por tanto, se usa la referencia a **v** para **clone( )** una copia de aquello a lo que hace referencia, y éste devuelve una referencia al tipo base **Object** (porque así está definido en **Object.clone( )**) que hay que convertir después al tipo adecuado.

En **main( )**, se prueba la diferencia entre los efectos de los dos enfoques de paso de parámetros en los dos métodos. La salida es:

```
a == b
a = 12
b = 12
c != d
c = 47
d = 48
```

Es importante darse cuenta de que las pruebas de equivalencia en Java no comparan la parte interna de los objetos para ver si sus valores son el mismo. Los operadores == y != simplemente comparan las *referencias*. Si las direcciones contenidas en las referencias son iguales, entonces apuntan al mismo objeto siendo por tanto "iguales". Por tanto ¡lo que verdaderamente prueban los operadores es si las referencias son alias de un mismo objeto!

El efecto de **Object.clone( )**

¿Qué es lo que de verdad ocurre cuando se invoca a **Object.clone( )** que hace tan esencial llamar a **super.clone( )** cuando se superpone **clone( )** en una clase? El método **clone( )** de la clase raíz es el responsable de crear la cantidad de almacenamiento correcta y de hacer la copia bit a bit del objeto original al espacio de almacenamiento del nuevo objeto. Es decir, no simplemente crea el es-

pacio de almacenamiento y se encarga de la copia de un **Object** —de hecho averigua el tamaño exacto del objeto que está copiando y lo duplica. Dado que todo esto ocurre a partir del código del método **clone( )** definido en la clase raíz (que no tiene idea de qué es lo que se ha heredado de la misma), como puede adivinarse, el proceso implica RTTI para determinar el objeto en concreto que está siendo clonado, y hacer una copia de bits correcta para ese tipo.

Sea lo que sea lo que se haga, la primera parte del proceso de clonado debería ser normalmente una llamada a **super.clone( )**. Así se echan las bases de la operación de clonado creando un duplicado exacto. En este momento, se pueden llevar a cabo otras operaciones necesarias para completar el clonado.

Para asegurarse de cuáles son esas operaciones, hay que entender exactamente qué es lo que hace exactamente **Object.clone( )**. En concreto ¿clona automáticamente el destino de todas las referencias? El ejemplo siguiente permite probar la respuesta a esta pregunta:

```
//: apendicea:Serpiente.java
// Probar el clonado para ver si también
// se clona el destino de las referencias.

public class Serpiente implements Cloneable {
  private Serpiente siguiente;
  private char c;
  // Valor de I == número de segmentos
  Serpiente(int i, char x) {
    c = x;
    if(--i > 0)
      siguiente = new Serpiente(i, (char)(x + 1));
  }
  void incrementar() {
    c++;
    if(siguiente != null)
      siguiente.incrementar();
  }
  public String toString() {
    String s = ":" + c;
    if(siguiente != null)
      s += siguiente.toString();
    return s;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Serpiente no se puede clonar");
    }
```

```
    return o;
  }
  public static void main(String[] args) {
    Serpiente s = new Serpiente(5, 'a');
    System.out.println("s = " + s);
    Serpiente s2 = (Serpiente)s.clone();
    System.out.println("s2 = " + s2);
    s.incrementar();
    System.out.println(
      "tras s.incrementar, s2 = " + s2);
  }
} ///:~
```

Una **Serpiente** está compuesta por un conjunto de segmentos, cada uno de tipo **Serpiente**. Por consiguiente, es una lista simplemente enlazada. Los segmentos se crean de forma recursiva, decreciendo el primer parámetro al constructor en cada segmento hasta llegar a cero. Para dar a cada segmento una etiqueta única, se incrementa el segundo parámetro, un **char**, por cada llamada recursiva al constructor.

El método **incrementar( )** incrementa cada etiqueta de forma que pueda verse el cambio, y el **toString( )** imprime cada etiqueta de forma recursiva. La salida es:

```
s = :a:b:c:d:e
s2 = :a:b:c:d:e
tras s.incrementar, s2 = a:c:d:e:f
```

Esto significa que **Object.clone( )** sólo duplica el primer segmento, y por consiguiente, hace una copia superficial. Si se desea duplicar toda la serpiente —una copia en profundidad— hay que llevar a cabo las operaciones adicionales dentro del **clone( )** superpuesto.

Generalmente, se invocará a **super.clone( )** en cualquier clase derivada de una clase clonable para asegurarse de que se den todas las operaciones de la clase base (incluida **Object.clone( )**). A continuación se hace una llamada explícita a **clone( )** por cada referencia que haya en el objeto; de otra forma, estas referencias se convertirían en alias de las del objeto original. Es análogo a la forma de llamar a los constructores —primero el constructor de la clase base, después el constructor de la siguiente derivada, y así hasta el constructor de la última clase derivada. La diferencia reside en que **clone( )** no es un constructor, por lo que no hay nada que haga automáticamente. Hay que asegurarse de hacerlo a mano.

Clonando un objeto compuesto

Al intentar hacer una copia en profundidad de un objeto compuesto, hay un problema. Hay que asumir que el método **clone( )** de los objetos miembros de hecho llevará a cabo una copia en profundidad de *sus* referencias, y así sucesivamente. Esto es bastante comprometido. Significa de hecho que para que funcione una copia en profundidad, uno debe o controlar todo el código en todas sus clases, o al menos tener el conocimiento suficiente sobre todas las clases involucradas en la copia

en profundidad como para saber que están llevando a cabo su copia en profundidad de forma correcta.

Este ejemplo muestra qué es lo que hay que hacer para lograr una copia en profundidad cuando se manipule un objeto compuesto:

```
//: apendicea:CopiaProfundidad.java
// Clonando un objeto compuesto.

class LeerProfundidad implements Cloneable {
  private double Profundidad;
  public LeerProfundidad(double profundidad) {
    this.profundidad = profundidad;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      e.printStackTrace(System.err);
    }
    return o;
  }
}

class LeerTemperatura implements Cloneable {
  private long tiempo;
  private double temperatura;
  public LeerTemperatura(double temperatura) {
    tiempo = System.currentTimeMillis();
    this.temperatura = temperatura;
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      e.printStackTrace(System.err);
    }
    return o;
  }
}

class LeerOceano implements Cloneable {
  private LeerProfundidad profundidad;
```

```
  private LeerTemperatura temperatura;
  public LeerOceano(double datost, double datosp){
    temperatura = new LeerTemperatura(datost);
    profundidad = new LeerProfundidad(datosp);
  }
  public Object clone() {
    LeerOceano o = null;
    try {
      o = (LeerOceano)super.clone();
    } catch(CloneNotSupportedException e) {
      e.printStackTrace(System.err);
    }
    // Hay que clonar las referencias:
    o.profundidad = (LeerProfundidad)o.profundidad.clone();
    o.temperatura =
      (LeerTemperatura)o.temperatura.clone();
    return o; // Conversión de Nuevo a Object
  }
}

public class CopiaProfundidad {
  public static void main(String[] args) {
    LeerOceano leer =
      new LeerOceano(33.9, 100.5);
    // Ahora clonarlo:
    LeerOceano l =
      (LeerOceano)leer.clone();
  }
} ///:~
```

LeerProfundidad y **LeerTemperatura** son bastante similares; ambos contienen sólo tipos de datos primitivos. Por consiguiente, el método **clone( )** puede ser bastante simple: llama a **super.clone( )** y devuelve el resultado. Nótese que en ambos casos el código de **clone( )** es idéntico.

LeerOceano está compuesto de objetos **LeerTemperatura** y **LeerProfundidad**, y así, para hacer una copia en profundidad, su **clone( )** debe clonar las referencias incluidas en **LeerOceano**. Para lograr esto, hay que convertir el resultado de **super.clone( )** a un objeto **LeerOceano** (de forma que se pueda acceder a las referencias **profundidad** y **temperatura**).

Una copia en profundidad con ArrayList

Revisitemos el ejemplo de **ArrayList** visto anteriormente en este apéndice. Esta vez, la clase **Int2** es clonable, por lo que se puede hacer una copia en profundidad del **ArrayList**:

```
//: apendicea:AniadirClonado.java
// Hay que dar unas pocas vueltas
```

```
// para añadir el clonado a una clase propia.
import java.util.*;

class Int2 implements Cloneable {
  private int i;
  public Int2(int ii) { i = ii; }
  public void incrementar() { i++; }
  public String toString() {
    return Integer.toString(i);
  }
  public Object clone() {
    Object o = null;
    try {
      o = super.clone();
    } catch(CloneNotSupportedException e) {
      System.err.println("Int2 no se puede clonar");
    }
    return o;
  }
}

// Una vez que es clonable, la herencia no
// elimina la "clonabilidad":
class Int3 extends Int2 {
  private int j; // Duplicado automáticamente
  public Int3(int i) { super(i); }
}

public class AniadirClonado {
  public static void main(String[] args) {
    Int2 x = new Int2(10);
    Int2 x2 = (Int2)x.clone();
    x2.incrementar();
    System.out.println(
      "x = " + x + ", x2 = " + x2);
    // Todo lo heredado es también clonable:
    Int3 x3 = new Int3(7);
    x3 = (Int3)x3.clone();

    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++ )
      v.add(new Int2(i));
    System.out.println("v: " + v);
    ArrayList v2 = (ArrayList)v.clone();
    // Ahora, clonar cada elemento:
```

```
    for(int i = 0; i < v.size(); i++)
      v2.set(i, ((Int2)v2.get(i)).clone());
    // Incrementar todos los elementos de v2:
    for(Iterator e = v2.iterator();
        e.hasNext(); )
      ((Int2)e.next()).incrementar();
    // Ver si los elementos de v han cambiado:
    System.out.println("v: " + v);
    System.out.println("v2: " + v2);
  }
} ///:~
```

Int3 hereda de **Int2** y se añade un nuevo miembro primitivo **int j**. Podría pensarse que hay que superponer de nuevo **clone( )** para asegurarse de la copia de **j**, pero no es así. Cuando se invoca al **clone( )** de **Int2** como el **clone( )** de **Int3**, llama a **Object.clone( )**, que determina que está trabajando con un **Int3** y duplica todos los bits del **Int3**. En la medida en que no se añaden referencias que necesiten ser clonadas, la llamada a **Object.clone( )** lleva a cabo toda la duplicación necesaria, independientemente de lo lejos que esté definido **clone( )** dentro de la jerarquía.

Aquí puede deducirse qué es necesario para hacer una copia en profundidad de un **ArrayList**: una vez clonado el **ArrayList**, hay que recorrer cada uno de los objetos apuntados por el **ArrayList**. Habría que hacer algo similar a esto si se desea hacer una copia en profundidad de un **HashMap**.

El resto del ejemplo muestra que se dio el clonado, mostrando que, una vez clonado un objeto, es posible modificarlo sin que el original se vea alterado.

Copia en profundidad vía serialización

Cuando se considera la serialización de objetos de Java (presentada en el Capítulo 11) podría observarse que si un objeto se serializa y después se deserializa, de hecho, está siendo clonado.

Por tanto ¿por qué no usar la serialización para llevar a cabo copias en profundidad? He aquí un ejemplo que comprueba los dos enfoques, cronometrándolos:

```
//: apendicea:Competir.java
import java.io.*;

class Cosa1 implements Serializable {}
class Cosa2 implements Serializable {
  Cosa1 o1 = new Cosa1();
}

class Cosa3 implements Cloneable {
  public Object clone() {
    Object o = null;
    try {
```

```
        o = super.clone();
      } catch(CloneNotSupportedException e) {
        System.err.println("Cosa3 no se puede clonar");
      }
      return o;
    }
  }

  class Cosa4 implements Cloneable {
    Cosa3 o3 = new Cosa3();
    public Object clone() {
      Cosa4 o = null;
      try {
        o = (Cosa4)super.clone();
      } catch(CloneNotSupportedException e) {
        System.err.println("Cosa4 no se puede clonar");
      }
      // Clonar también el campo:
      o.o3 = (Cosa3)o3.clone();
      return o;
    }
  }

  public class Competir {
    static final int TAMANIO = 5000;
    public static void main(String[] args)
    throws Exception {
      Cosa2[] a = new Cosa2[TAMANIO];
      for(int i = 0; i < a.length; i++)
        a[i] = new Cosa2();
      Cosa4[] b = new Cosa4[TAMANIO];
      for(int i = 0; i < b.length; i++)
        b[i] = new Cosa4();
      long t1 = System.currentTimeMillis();
      ByteArrayOutputStream buf =
        new ByteArrayOutputStream();
      ObjectOutputStream salida =
        new ObjectOutputStream(buf);
      for(int i = 0; i < a.length; i++)
        salida.writeObject(a[i]);
      // Ahora, hacerse con las copias:
      ObjectInputStream entrada =
        new ObjectInputStream(
          new ByteArrayInputStream(
            buf.toByteArray()));
```

```
    Cosa2[] c = new Cosa2[TAMANIO];
    for(int i = 0; i < c.length; i++)
      c[i] = (Cosa2)entrada.readObject();
    long t2 = System.currentTimeMillis();
    System.out.println(
      "Duplicacion via serializacion: " +
      (t2 - t1) + " Milisegundos");
    // Ahora, intentar clonar:
    t1 = System.currentTimeMillis();
    Cosa4[] d = new Cosa4[TAMANIO];
    for(int i = 0; i < d.length; i++)
      d[i] = (Cosa4)b[i].clone();
    t2 = System.currentTimeMillis();
    System.out.println(
      "Duplicacion via clonado: " +
      (t2 - t1) + " Milisegundos");
  }
} ///:~
```

Cosa2 y **Cosa4** contienen objetos miembro, de forma que se copie algo en profundidad. Es interesante darse cuenta de que mientras que es fácil configurar las clases **Serializable**, hay que hacer mucho más trabajo para duplicarlas. El clonado implica mucho trabajo para configurar las clases, pero la duplicación de objetos es relativamente simple. Los resultados hablan por sí solos. He aquí los resultados de tres ejecuciones:

```
Duplicacion via serializacion: 940 Milisegundos
Duplicacion via clonado: 50 Milisegundos

Duplicacion via serializacion: 710 Milisegundos
Duplicacion via clonado: 60 Milisegundos

Duplicacion via serializacion: 770 Milisegundos
Duplicacion via clonado: 50 Milisegundos
```

A pesar de la diferencia de tiempo tan significativa entre la serialización y el clonado, también se verá que la técnica de serialización parece fluctuar más en cuanto a duración, mientras que el clonado parece ser más estable.

Añadiendo "clonabilidad" a lo largo de toda una jerarquía

Si se crea una clase nueva, su clase base por defecto es **Object**, y por tanto, no clonable (como se verá en la sección siguiente). Mientras no se añada explícitamente "clonabilidad", ésta no surgirá por sí sola. Pero se puede añadir en cualquier capa y todas sus descendientes serán clonables:

```
//: apendicea:GolpeHorror.java
// La Clonabilidad puede insertarse
// en cualquier nivel de la herencia.
import java.util.*;

class Persona {}
class Heroe extends Persona {}
class Cientifico extends Persona
    implements Cloneable {
  public Object clone() {
    try {
      return super.clone();
    } catch(CloneNotSupportedException e) {
      // Esto no debería ocurrir nunca:
      // ¡Ya es clonable!
      throw new InternalError();
    }
  }
}
class CientificoLoco extends Cientifico {}

public class GolpeHorror {
  public static void main(String[] args) {
    Persona p = new Persona();
    Heroe h = new Heroe();
    Cientifico s = new Cientifico();
    CientificoLoco m = new CientificoLoco();

    // p = (Persona)p.clone(); // Error de compilación
    // h = (Heroe)h.clone(); // Error de compilación
    s = (Cientifico)s.clone();
    m = (CientificoLoco)m.clone();
  }
} ///:~
```

Antes de añadir "clonabilidad", el compilador evitaba que clonases cosas. Cuando se añadió "clonabilidad" a **Cientifico**, tanto esta clase como todas sus descendientes se convirtieron en clonables.

¿Por qué un diseño tan extraño?

Si todo esto parece seguir un esquema extraño, es porque así lo es. Uno podría preguntarse por qué es así. ¿Qué hay detrás de un diseño así?

Originalmente, Java se diseñó como un lenguaje para controlar cajas hardware y, desde luego, no con Internet en mente. En un lenguaje de propósito general como éste tiene sentido que el progra-

mador pueda clonar cualquier objeto. Por ello, se ubicó **clone( )** en la clase raíz **Object**, *pero* era un método **public** de forma que siempre se pudiera clonar cualquier objeto. Éste parecía ser el enfoque más flexible, y después de todo ¿qué daño podría hacer?

Bien, cuando Java se empezó a contemplar como el último lenguaje de programación de Internet, las cosas cambiaron. De repente, hay aspectos de seguridad, y por supuesto, estos objetos están relacionados con el uso de objetos, y necesariamente no se desea que nadie sea capaz de clonar los objetos de seguridad. Por tanto, lo que se ven son un montón de parches aplicados al esquema original, simple y directo: ahora **clone( )** es **protected** en **Object**. Hay que superponerlo *e* **implementar Cloneable** *y* hacer frente a las posibles excepciones.

Merece la pena reseñar que hay que usar la interfaz **Cloneable** *sólo* si se va a invocar al método **clone( )** de **Object**, puesto que este método comprueba en tiempo de ejecución si la clase implementa **Cloneable**. Pero por motivos de consistencia (y puesto que de cualquier forma, **Cloneable** está vacío) hay que implementarlo.

Controlando la "clonabilidad"

Uno podría sugerir que para eliminar la "clonabilidad" simplemente hace falta que se haga **private** el método **clone( )**, pero esto no funcionaría puesto que no se puede tomar un método de la clase base y hacerlo menos accesible en una clase derivada. Por tanto, no es tan simple. Y lo que es más, hay que poder controlar si se puede clonar el objeto. De hecho, en una clase que uno diseñe se pueden tomar varias actitudes a este respecto:

1. Indiferencia. No se hace nada con el clonado, lo que significa que tu clase no puede clonarse pero a una clase que se derive de la misma se le podría añadir clonación si se desea. Esto sólo funciona si el **Object.clone( )** por defecto hace algo razonable con todos los campos de la clase.

2. Soportar **clone( )**. Seguir la práctica estándar de implementar **Cloneable** y superponer **clone( )**. En el **clone( )** superpuesto se invoca a **super.clone( )** y se capturan todas las excepciones (de forma que el **clone( )** superpuesto no lance ninguna excepción).

3. Soportar el clonado condicionalmente. Si tu clase tiene referencias a otros objetos que podrían o no ser clonables (por ejemplo una clase contenedora) tu **clone( )** puede intentar clonar todos los objetos para los que se tenga referencias, y si lanzan excepciones, simplemente pasárselas al programador. Por ejemplo, considérese un tipo de **ArrayList** especial que intenta clonar todos los objetos que guarda. Cuando se escriba un **ArrayList** así, no se puede saber el tipo de objetos que el programador cliente podría poner en el **ArrayList**, por lo que no se sabe si pueden ser clonados.

4. No implementar **Cloneable** pero superponer **clone( )** como **protected**, produciendo el comportamiento de copia correcto para todos los campos. De esta forma, cualquiera que herede de esta clase puede superponer **clone( )** e invocar a **super.clone( )** para producir el comportamiento de copia correcto. Nótese que una implementación puede y debería invocar a **super.clone( )** incluso aunque ese método espere un objeto **Cloneable** (en caso contrario, lan-

zará una excepción), porque nadie la invocará directamente con un objeto del tipo creado. Sólo será invocada a través de clases derivadas, que, si se desea que funcione correctamente, deberán implementar **Cloneable**.

5. Intentar evitar el clonado no implementando **Cloneable** y superponiendo **clone( )** para que lance una excepción. Esto sólo tiene éxito si cualquier clase derivada de ésta llama a **super.clone( )** en su redefinición de **clone( )**. De otra forma, un programador podría volver a ello.

6. Evitar el clonado haciendo que la clase sea **final**. Hacer la clase **final** es la única forma de garantizar que se evite el clonado. Además, al tratar con objetos de seguridad o cualquier otra situación en la que se desee controlar el número de objetos que se crean, habría que hacer **private** todos los constructores y proporcionar uno o más métodos especiales para crear objetos. De esa forma, estos métodos pueden restringir el número de objetos que se crean y las condiciones en las que se crean. (Un caso particular de esto es el patrón *singleton* que aparece en *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*.)

He aquí un ejemplo que muestra las varias formas de implementar el clonado, y luego, cómo ser "deshabilitado" más abajo en la jeraquía:

```
//: apendicea:ComprobarCloneable.java
// Comprobar si se puede clonar una referencia.

// No se puede clonar porque no
// superpone clone():
class Ordinario {}

// Superpone clone, pero no implementa
// Cloneable:
class ClonErroneo extends Ordinario {
  public Object clone()
       throws CloneNotSupportedException {
    return super.clone(); // Lanza excepcion
  }
}

// Hace todo lo necesario para el clonado:
class EsClonable extends Ordinario
    implements Cloneable {
  public Object clone()
       throws CloneNotSupportedException {
    return super.clone();
  }
}

// Desconectar el clonado lanzando la excepción:
class NoMas extends EsClonable {
```

```
  public Object clone()
       throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
  }
}

class ProbarMas extends NoMas {
  public Object clone()
       throws CloneNotSupportedException {
    // Llama a NoMas.clone(), lanza excepción:
    return super.clone();
  }
}

class Retornar extends NoMas {
  private Retornar duplicar(Retornar b) {
    // Hacer de alguna forma una copia de b
    // y devolverla. Ésta es una copia estúpida
    // simplemente para que se vea:
    return new Retornar();
  }
  public Object clone() {
    // No llama a NoMas.clone():
    return duplicar(this);
  }
}

// No se puede heredar de éste, así que no
// puede superponer el método clone como en Retornar:
final class RealmenteNoMas extends NoMas {}

public class ComprobarCloneable {
  static Ordinario intentarClonar(Ordinario ord) {
    String id = ord.getClass().getName();
    Ordinario x = null;
    if(ord instanceof Cloneable) {
      try {
        System.out.println("Intentando " + id);
        x = (Ordinario)((esClonable)ord).clone();
        System.out.println("Clonado " + id);
      } catch(CloneNotSupportedException e) {
        System.err.println("No se pudo clonar "+id);
      }
    }
    return x;
```

```
  }
  public static void main(String[] args) {
    // Molde hacia arriba:
    Ordinary[] ord = {
      new EsClonable(),
      new ClonErroneo(),
      new NoMas(),
      new ProbarMas(),
      new Retornar(),
      new RealmenteNoMas(),
    };
    Ordinario x = new Ordinario();
    // Esto no compilara, pues clone() es
    // protected en Object:
    //! x = (Ordinario)x.clone();
    // intentarClonar() Primero comprueba si una
    // clase implementa Cloneable:
    for(int i = 0; i < ord.length; i++)
      intentarClonar(ord[i]);
  }
} ///:~
```

La primera clase, **Ordinario**, representa los tipos de clases que hemos venido viendo a lo largo de todo el libro: sin soporte para el clonado, pero como se ve, tampoco hay ninguna prevención contra el mismo. Si se tiene una referencia a un objeto **Ordinario** que podría haber sufrido una conversión hacia arriba desde una clase aún más derivada, no puede decirse si puede o no ser clonada.

La clase **ClonErroneo** muestra una forma incorrecta de implementar el clonado. Superpone **Object.clone( )** y convierte en **public** ese método, pero no implementa **Cloneable**, por lo que cuando se llama a **super.clone( )** (lo que a efectos es una llamada a **Object.clone( )**), se lanza **CloneNotSupportedException** de forma que el clonado no funcionará.

En **EsClonable** puede verse cómo se llevan a cabo las acciones pertinentes para el clonado: se superpone **clone( )** y se implementa **Cloneable**. Sin embargo, este método **clone( )** y otros muchos que siguen este ejemplo, *no* capturan **CloneNotSupportedException**, sino que en su lugar se lo pasan al llamador, que deberá envolverlo con un bloque *try-catch*. En tus métodos **clone( )**, generalmente tendrás que capturar **CloneNotSupportedException** *dentro* de **clone( )** en vez de pasarlo. Como se verá, en este ejemplo es más informativo pasar las excepciones.

La clase **NoMas** intenta "desactivar" el clonado de la forma que pretendían los diseñadores de Java: en la clase derivada **clone( )** se lanza **CloneNotSupportedException**. El método **clone( )** de la clase **NoMas** llama adecuadamente a **super.clone( )**, y éste resuelve a **NoMas.clone( )**, que lanza una excepción y evita el clonado.

Pero ¿qué ocurre si el programador no sigue la pauta "adecuada" de llamar a **super.clone( )** dentro del método **clone( )** superpuesto? En **Retornar**, puede verse cómo puede ocurrir esto. Esta clase usa un método **duplicar( )** separado para hacer una copia del objeto actual y llama a este método

dentro de **clone( )** *en vez de* invocar a **super.clone( )**. La excepción no se lanza nunca y la clase nueva es clonable. No se puede confiar en que se lance una excepción para evitar hacer clonable una clase. La única solución a prueba de bombas es la mostrada en **RealmenteNoMas**, que es **final**, y de la que por consiguiente no se puede heredar. Esto significa que si **clone( )** lanza una excepción en la clase **final**, no puede ser modificada con herencia y se asegura prevenir el clonado. (No se puede invocar explícitamente a **Object.clone( )** desde una clase que tiene un nivel de herencia arbitrario; uno esta limitado a invocar a **super.clone( )** que tiene acceso sólo a la clase base directa.) Por consiguiente, si se construye cualquier objeto que involucre aspectos de seguridad, se deseará que esas clases sean **final**.

El primer método que se ve en la clase **ComprobarCloneable** es **intentarClonar()**, que toma un objeto **Ordinario** y comprueba si es clonable o no con **instanceof**. Si lo es, convierte el objeto a un **EsClonable**, llama a **clone( )** y convierte el resultado de nuevo a **Ordinario**, capturando cualquier excepción que se lance. Nótese el uso de la identificación de tipos en tiempo de ejecución (ver Capítulo 12) para imprimir el nombre de la clase de forma que pueda verse lo que está ocurriendo.

En **main( )**, se crean distintos tipos de objetos **Ordinario** que son convertidos a **Ordinario** en la definición del array. Las dos primeras líneas de código siguientes crean un objeto **Ordinario** tal cual e intentan clonarlo. Sin embargo, este código no compilará pues **clone( )** es un método **protected** en **Object**. El resto del código recorre el array e intenta clonar cada objeto, informando del éxito o fracaso de cada caso. La salida es:

```
Intentando EsClonable
Clonado EsClonable
Intentando NoMas
No se pudo clonar NoMas
Intentando ProbarMas
No se pudo clonar ProbarMas
Intentando Retornar
Clonado Retornar
Intentando RealmenteNoMas
No se pudo clonar RealmenteNoMas
```

Por tanto, para resumir, si se desea que una clase sea clonable:

1. Implementar la interfaz **Clonable**.
2. Superponer **clone( )**.
3. Invocar a **super.clone()** dentro del **clone( )** propio.
4. Capturar las excepciones dentro del **clone( )** propio.

Esto producirá los efectos más convenientes.

El constructor de copia

El clonado puede parecer un proceso complicado de configurar. Podría parecer que debería haber alguna alternativa. Un enfoque que se le podría ocurrir a alguien (especialmente si se trata de un

programador de C++) es hacer un constructor especial cuyo trabajo sea duplicar un objeto. En C++, a esto se le llama un *constructor de copia*. A primera vista, ésta parece una solución obvia, pero de hecho, no funciona. He aquí un ejemplo:

```
//: apendicea:ConstructorCopia.java
// Un constructor para copiar un objeto del mismo tipo,
// como un intento de crear una copia local.

class CualidadesFruta {
  private int peso;
  private int color;
  private int solidez;
  private int madurez;
  private int olor;
  // etc.
  CualidadesFruta() { // Constructor por defecto
    // Hacer algo que tenga sentido...
  }
  // Otros constructores:
  // ...
  // Constructor de copia:
  CualidadesFruta(CualidadesFruta f) {
    peso = f.peso;
    color = f.color;
    solidez = f.solidez;
    madurez = f.madurez;
    olor = f.olor;
    // etc.
  }
}

class Semilla {
  // Miembros...
  Semilla() { /* Constructor por defecto */ }
  Semilla(Semilla s) { /* Constructor de copia */ }
}

class Fruta {
  private CualidadesFruta cf;
  private int semillas;
  private Semilla[] s;
  Fruta(CualidadesFruta c, int conteoSemilla) {
    cf = c;
    Semillas = conteoSemillas;
    s = new Semilla[semillas];
    for(int i = 0; i < semillas; i++)
```

```
      s[i] = new Semilla();
  }
  // Otros constructores:
  // ...
  // Constructor de copia:
  Fruta(Fruta f) {
    cf = new CualidadesFruta(f.cf);
    semillas = f.semillas;
    // Llamar a todos los constructores de copia de Semilla:
    for(int i = 0; i < semillas; i++)
      s[i] = new Semilla(f.s[i]);
    // Otras actividades de construccion de copias...
  }
  // Para permitir a los constructores derivados (u otros
  // métodos) poner distintas calidades:
  protected void aniadirCualidades(CualidadesFruta c) {
    cf = c;
  }
  protected CualidadesFruta obtenerCualidades() {
    return cf;
  }
}

class Tomate extends Fruta {
  Tomate() {
    super(new CualidadesFruta(), 100);
  }
  Tomate(Tomate t) { // Constructor de copia
    super(t); // Molde hacia arriba para los constructores de copia base
    // Otras actividades de construcción de copias...
  }
}

class CualidadesZebra extends CualidadesFruta {
  private int rayas;
  CualidadesZebra() { // Constructor por defecto
    // hacer algo que tenga sentido...
  }
  CualidadesZebra(CualidadesZebra z) {
    super(c);
    rayas = c.rayas;
  }
}

class ZebraVerde extends Tomate {
```

```
  ZebraVerde() {
    aniadirCualidades(new CualidadesZebra());
  }
  ZebraVerde(ZebraVerde z) {
    super(z); // Invoca Tomate(Tomate)
    // Restaurar las actividades correctas:
    aniadirCualidades(new CualidadesZebra());
  }
  void evaluar() {
    CualidadesZebra cz =
      (CualidadesZebra)obtenerCualidades();
    // Hacer algo con las cualidades
    // ...
  }
}

public class ConstructorCopia {
  public static void madurar(Tomate t) {
    // Usar el constructor de copia:
    t = new Tomate(t);
    System.out.println("En madurar, t es un " +
      t.getClass().getName());
  }
  public static void cortar(Fruta f) {
    f = new Fruta(f); // Ummm... ¿funcionara esto?
    System.out.println("En cortar, f es un " +
      f.getClass().getName());
  }
  public static void main(String[] args) {
    Tomate tomate = new Tomate();
    madurar(tomate); // OK
    cortar(tomate); // OOPS!
    ZebraVerde z = new ZebraVerde();
    madurar(z); // OOPS!
    cortar(z); // OOPS!
    z.evaluar();
  }
} ///:~
```

Esto parece un poco extraño a primera vista. Es verdad que *fruta* tiene *cualidades*, pero ¿por qué no poner datos miembro representando las cualidades directamente en la clase **Fruta**? Hay dos razones potenciales. La primera es que se podría querer insertar o modificar las cualidades de forma sencilla. Nótese que **Fruta** tiene un método **protected aniadirCualidades( )** para permitir que las clases derivadas sí lo hagan. (Se podría pensar que lo lógico es tener un constructor **protected** en **Fruta** que tome un parámetro **CualidadesFruta**, pero los constructores no se heredan por lo que

no estaría disponible en clases heredadas de ésta.) Haciendo que las cualidades de la fruta sean una clase separada, se tiene mayor flexibilidad, incluyendo la potestad para cambiar las cualidades a mitad de camino en la vida de un objeto **Fruta** en particular.

La segunda razón para hacer **CualidadesFruta** un objeto independiente es por si se desea añadir cualidades nuevas o cambiar el comportamiento vía la herencia y el polimorfismo. Nótese que en el caso de **ZebraVerde** (que *verdaderamente* es un tipo de tomate —yo los he cultivado y son geniales), el constructor llama a **aniadirCualidades( )** y le pasa un objeto **CualidadesZebra**, que se deriva de **CualidadesFruta**, de forma que se puede adjuntar a la referencia a **CualidadesFruta** en la clase base. Por supuesto, cuando **ZebraVerde** usa el **CualidadesFruta**, debe hacer una conversión del mismo hacia el tipo correcto (como se vio en **evaluar( )**), pero siempre sabe que ese tipo es **CualidadesZebra**.

También se verá que hay una clase **Semilla**, y que **Fruta** (que por definición lleva sus propias *seeds* o semillas[4]) contiene un array de **Semillas**.

Finalmente, fíjese que cada clase tiene un constructor de copia, y que cada uno se encarga de llamar a los constructores de copia correspondientes a la clase base y los objetos miembros para lograr una copia en profundidad. El constructor de copia se prueba dentro de la clase **ConstructorCopia**. El método **madurar( )** toma un parámetro **Tomate** y hace una construcción de una copia del mismo para duplicar el objeto:

```
t = new Tomate(t);
```

mientras que **cortar( )** toma un objeto **Fruta** más genérico, duplicándolo también:

```
f = new Fruta(f);
```

Éstos se prueban en **main( )** condistintos tipos de **Fruta**. He aquí la salida:

```
En madurar, t es un Tomate
En cortar, f es una Fruta
En madurar, t es un Tomate
En cortar, f es una Fruta
```

Es aquí donde aflora el problema. Tras la construcción de copia que se da al **Tomate** dentro de **cortar( )**, el resultado deja de ser un objeto **Tomate** para ser simplemente una **Fruit**. Ha perdido todas las características que lo hacían ser un Tomate. Y lo que es más, al tomar una **ZebraVerde**, tanto **madurar( )** como **cortar( )** la convierten en un **Tomate** y una **Fruta** respectivamente. Por consiguiente, desgraciadamente, el esquema del constructor de copia no es bueno en Java cuando lo que se pretende es hacer una copia local de un objeto.

¿Por qué funciona en C++ y no en Java?

El constructor de copia es una parte fundamental de C++, puesto que hace automáticamente una copia de un objeto. No obstante, el ejemplo anterior muestra que no funciona en Java. ¿Por qué? En

[4] Excepto el pobre aguacate, que ha sido reclasificado a simplemente "grasa".

Java todo lo que manipulamos son referencias, mientras que en C++ se puede tener entidades que parecen referencias y se puede *también* pasar los objetos directamente. Esto es para lo que sirve este constructor: cuando se desea tomar un objeto y pasarlo por valor, duplicando por consiguiente el objeto. Por tanto, funciona bien en C++, pero debería tenerse en cuenta que este esquema falla en Java, por lo que no debe usarse en este lenguaje.

Clases de sólo lectura

Mientras que la copia local producida por **clone( )** da los resultados deseados en los casos apropiados, es un ejemplo de obligar al programador (el autor del método) a responsabilizarse de prevenir los efectos negativos del uso de alias. ¿Qué ocurre si se está construyendo una biblioteca de propósito tan general y uso tan frecuente que no se puede suponer que las operaciones de clonado se harán siempre en los lugares adecuados? O más probablemente, ¿qué ocurre si se *desea* permitir el *uso de alias* para lograr eficiencia —para evitar la duplicación innecesaria de objetos—, pero no se desean los tan negativos efectos laterales del uso de alias?

Una solución es crear *objetos inmutables* que pertenezcan a clases de sólo lectura. Se puede definir una clase de forma que ningún método de la misma cause cambios al estado interno del objeto. En una clase así, el uso de alias no tiene impacto puesto que sólo se puede *leer* el estado interno, de forma que muchos fragmentos de código puedan estar leyendo el mismo objeto sin problemas.

Como ejemplo simple de los objetos inmutables, la biblioteca estándar de Java contiene las clases "envoltorio" para todos los tipos primitivos. Se podría ya haber descubierto que, si se desea almacenar un **int** dentro de un contenedor como una **ArrayList** (que sólo guarda **referencias a Object**), se puede envolver el **int** dentro de la clase **Integer** de la biblioteca estándar:

```
//: apendicea:EnteroInmutable.java
// No se puede alterar la clase Integer.
import java.util.*;

public class EnteroInmutable {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new Integer(i));
    // ¿Pero cómo se cambia el int
    // interno al Integer?
  }
} ///:~
```

La clase **Integer** (además de todas las clases "envoltorio" primitivas) implementa la inmutabilidad de forma simple: no tienen métodos que permitan modificar el objeto.

Si se desea un objeto que guarde un tipo primitivo que se pueda modificar, hay que crearlo a mano. Afortunadamente, esto es trivial:

```
//: apendicea:EnteroMutable.java
// Una clase envoltorio modificable.
import java.util.*;

class ValorInt {
  int n;
  ValorInt(int x) { n = x; }
  public String toString() {
    return Integer.toString(n);
  }
}

public class EnteroMutable {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 10; i++)
      v.add(new ValorInt(i));
    System.out.println(v);
    for(int i = 0; i < v.size(); i++)
      ((ValorInt)v.get(i)).n++;
    System.out.println(v);
  }
} ///:~
```

Nótese que **n** es amigo para simplificar la codificación.

ValorInt puede incluso ser más simple si la inicialización a cero por defecto es la adecuada (momento en el que el constructor deja de ser necesario) y no hay que preocuparse de su impresión (por lo que no se necesita el **toString( )**).

```
class ValorInt { int n; }
```

Coger el elemento y convertirlo es un poco molesto, pero eso es cosa de **ArrayList**, no de **ValorInt**.

Creando clases de sólo lectura

Es posible crear clases de sólo lectura. He aquí un ejemplo:

```
//: apendicea:Inmutable1.java
// Objetos que no pueden ser modificados
// e inmunes al uso de alias.

public class Inmutable1 {
  private int datos;
  public Inmutable1(int valIni) {
    datos = valIni;
```

```
  }
  public int leer() { return datos; }
  public boolean nocero() { return datos != 0; }
  public Inmutable1 cuadruple() {
    return new Inmutable1(datos * 4);
  }
  static void f(Inmutable1 i1) {
    Inmutable1 cuad = i1.cuadruple();
    System.out.println("i1 = " + i1.leer());
    System.out.println("cuad = " + cuad.leer());
  }
  public static void main(String[] args) {
    Inmutable1 x = new Inmutable1(47);
    System.out.println("x = " + x.leer());
    f(x);
    System.out.println("x = " + x.leer());
  }
} ///:~
```

Todos los datos son **private**, y se verá que ninguno de los métodos **public** modifica esos datos. De hecho, el método que parece modificar un objeto es **cuadruple( )**, pero éste crea un nuevo objeto **Inmutable1** y deja el original intacto.

El método **f( )** toma un objeto **Inmutable1** y lleva a cabo varias operaciones sobre él mismo, y la salida de **main( )** demuestra que no se hace ningún cambio a **x**. Por consiguiente, el objeto de **x** podría recibir tantos alias como se desee sin que esto suponga daño alguno, pues la clase **Inmutable1** está diseñada para garantizar que los objetos no se modifiquen.

Los inconvenientes de la inmutabilidad

Crear una clase inmutable parece en primera instancia proporcionar una solución elegante. Sin embargo, siempre que se necesita un objeto modificado del tipo nuevo, hay que sufrir la sobrecarga debida a la creación del nuevo objeto, además de causar potencialmente más recolecciones de basura. En algunas clases esto no es un problema, pero en otras (como la clase **String**) es prohibitivamente caro.

La solución es crear una clase amiga que *puede* modificarse. Así, al hacer muchas modificaciones, se puede pasar a usar la clase amiga modificable y volver a la clase inmutable cuando se haya acabado.

El ejemplo de arriba puede modificarse así:

```
//: apendicea:Immutable2.java
// Una clase clase amiga para hacer
// cambios a objetos inmutables.
```

```
class Mutable {
  private int datos;
  public Mutable(int valIni) {
    datos = valIni;
  }
  public Mutable sumar(int x) {
    datos += x;
    return this;
  }
  public Mutable multiplicar(int x) {
    datos *= x;
    return this;
  }
  public Inmutable2 hacerInmutable2() {
    return new Inmutable2(datos);
  }
}

public class Inmutable2 {
  private int datos;
  public Inmutable2(int valIni) {
    datos = valIni;
  }
  public int leer() { return datos; }
  public boolean nocero() { return datos != 0; }
  public Inmutable2 sumar(int x) {
    return new Inmutable2(datos + x);
  }
  public Inmutable2 multiplicar(int x) {
    return new Inmutable2(datos * x);
  }
  public Mutable hacerMutable() {
    return new Mutable(datos);
  }
  public static Inmutable2 modificar1(Inmutable2 y){
    Inmutable2 val = y.sumar(12);
    val = val.multiplicar(3);
    val = val.sumar(11);
    val = val.multiplicar(2);
    return val;
  }
  // Esto produce el mismo resultado:
  public static Inmutable2 modificar2(Inmutable2 y){
    Mutable m = y.hacerMutable();
    m.sumar(12).multiplicar(3).sumar(11).multiplicar(2);
```

```
    return m.hacerInmutable2();
  }
  public static void main(String[] args) {
    Inmutable2 i2 = new Inmutable2(47);
    Inmutable2 r1 = modificar1(i2);
    Inmutable2 r2 = modificar2(i2);
    System.out.println("i2 = " + i2.leer());
    System.out.println("r1 = " + r1.leer());
    System.out.println("r2 = " + r2.leer());
  }
} ///:~
```

Inmutable2 contiene métodos que, como antes, preservan la inmutabilidad de los objetos produciendo nuevos objetos siempre que se desee una modificación. Se trata de los métodos **sumar( )** y **multiplicar( )**. La clase amiga se llama **Mutable** y también tiene métodos **sumar( )** y **multiplicar( )**, que modifican el objeto **Mutable** en vez de construir uno nuevo. Además, **Mutable** tiene un método que usa sus datos para producir un objeto **Inmutable2** y viceversa.

Los dos métodos estáticos **modificar1( )** y **modificar2( )** muestran dos enfoques distintos para producir el mismo resultado. En **modificar1( )**, todo se hace dentro de la clase **Inmutable2** y puede verse que en el proceso se crean cuatro objetos **Immutable2** nuevos. (Y cada vez que se reasigna **val**, el objeto anterior se convierte en basura.)

En el método **modificar2( )** puede verse que lo primero que se hace es tomar **Inmutable2 y** y producir un **Mutable** a partir de él mismo. (Esto es simplemente como llamar a **clone( )** como se vio anteriormente, pero esta vez se crea un tipo de objeto distinto.) Después se usa el objeto **Mutable** para llevar a cabo muchas operaciones de cambio *sin* precisar la creación de muchos objetos nuevos. Finalmente, se vuelve a convertir en **Inmutable2**. Aquí, se crean dos objetos nuevos (el **Mutable** y el resultado **Inmutable2**) en vez de cuatro.

Por tanto, este enfoque tiene sentido cuando:

1. Se necesitan objetos inmutables y
2. A menudo son necesarias muchas modificaciones o
3. Es caro crear objetos inmutables nuevos.

Strings inmutables

Considérese el código siguiente:

```
//: apendicea:Encadenador.java

public class Encadenador {
  static String mayusculas(String s) {
    return s.toUpperCase();
```

```
  }
  public static void main(String[] args) {
    String q = new String("hola");
    System.out.println(q); // hola
    String qq = mayusculas(q);
    System.out.println(qq); // HOLA
    System.out.println(q); // hola
  }
} ///:~
```

Cuando se pasa **q** en **mayusculas( )** es de hecho una copia de la referencia a **q**. El objeto al que está conectado esta referencia sigue en una única ubicación física. Se copian las referencias al ser pasadas.

Echando un vistazo a la definición de **mayusculas( )**, se puede ver que la referencia que se pasa es **s**, y que existe sólo mientras se está ejecutando el cuerpo de **mayusculas( )**. Cuando acaba **mayusculas( )**, la referencia local **s** se desvanece. El método **mayusculas( )** devuelve el resultado, que es la cadena de texto original con todos los caracteres en mayúsculas. Por supuesto, de hecho devuelve una referencia al resultado. Pero resulta que esa referencia que devuelve apunta a un nuevo objeto, quedando el **q** original de lado. ¿Cómo ocurre esto?

Constantes implícitas

Si se dice:

```
String s = "asdf";
String x = Encadenador.mayusculas(s);
```

¿se desea verdaderamente que el método **mayusculas( )** *modifique* el parámetro? En general, no, porque un parámetro suele parecer al lector del código como un fragmento de información proporcionado al método, no algo que pueda modificarse. Ésta es una garantía importante, puesto que hace que el código sea más fácil de leer y entender.

En C++, disponer de esta garantía era lo suficientemente importante como para incluir la palabra clave **const**, que permitía al programador asegurar que no se pudiera usar una referencia (en C++, puntero o referencia) para modificar el objeto original. Entonces, se pedía al programador de C++ que fuera diligente y usara **const** en todas partes. Esto puede confundir, además de ser fácil de olvidar.

Sobrecarga de "+" y el **StringBuffer**

Los objetos de la clase **String** están diseñados para ser inmutables, usando la técnica recién mostrada. Si se examina la documentación en línea de la clase **String** (como se resumirá más adelante en este capítulo), se verá que todo método de la clase que sólo aparenta modificar un **String** verdaderamente crea y devuelve un objeto **String** completamente nuevo que contiene la modificación. El **String** original queda intacto. Por consiguiente, no hay una faceta en Java que, como **const** en C++, permita al compilador soportar la inmutabilidad de los objetos. Si se desea lograrla, hay que implementarla a mano, como hace **String**.

Dado que los objetos **String** son inmutables, se pueden establecer tantos alias a un **String** como se desee. Dado que es de sólo lectura, no se puede dar el caso de que una referencia altere algo que afecte a otras. Por tanto, un objeto de sólo lectura soluciona de forma elegante el problema del uso de alias.

También parece posible manejar todas las clases en las que se necesita un objeto modificado, creando una versión completamente nueva del objeto con sus modificaciones, como hace **String**. Sin embargo, para algunas operaciones, esto no es eficiente. En este sentido, destaca el operador "+", sobrecargado para objetos **String**. La sobrecarga implica que se le da un significado extra cuando se usa con cierta clase en concreto. (Los operadores "+" y "+=" para **String** son los únicos sobrecargados en Java, y Java no permite al programador sobrecargar ninguno más)[5].

Cuando se usa con objetos **String**, el "+" permite concatenar **String**s:

```
String s = "abc" + foo + "def" + Integer.toString(47);
```

Puede imaginarse cómo *debería* funcionar esto: el **String** "abc" podría tener un método **append( )** que creara un nuevo objeto **String** que contuviera "abc" concatenado con los contenidos de **foo**. El nuevo objeto **String** crearía a continuación otro **String** con la adición de "def" y así sucesivamente.

Esto funcionaría, pero requiere de la creación de muchos objetos **String** simplemente para componer este nuevo **String**, y después hay un conjunto de objetos **String** intermedios que deberían ser eliminados por el recolector de basura. Sospecho que los diseñadores de Java intentaron primero este enfoque (que no es sino una lección en diseño de software —no se sabe nada sobre un sistema hasta que se prueba a codificar en él y se tiene algo funcionado). También sospecho que descubrieron que implicaba un rendimiento inaceptable.

La solución es una clase amiga mutable semejante a la ya vista. En el caso de **String**, esta clase compañero se llama **StringBuffer**, y el compilador crea automáticamente un **StringBuffer** para evaluar ciertas expresiones, en particular cuando se usan los operadores sobrecargados + y += con objetos **String**. El ejemplo muestra lo que ocurre:

```
//: apendicea:CadenasInmutables.java
// Demostrando StringBuffer.

public class CadenasInmutables {
  public static void main(String[] args) {
    String foo = "foo";
    String s = "abc" + foo +
```

[5] C++ permite al programador sobrecargar operadores según desee. Debido a que éste puede ser un proceso complicado (ver Capítulo 10 de *Thinking in C++, 2nd edition,* Prentice-Hall, 2000), los diseñadores de Java lo consideraron una faceta "negativa" que no debería incluirse en este lenguaje. No fue algo tan malo pues acabaron haciéndolo ellos mismos, e irónicamente, la sobrecarga sería mucho más fácil de usar en C++ que en Java. Esto se puede ver en Phyton (*http://www.Python.org*), que tiene sobrecargado, por ejemplo, el recolector de basura.

```
        "def" + Integer.toString(47);
      System.out.println(s);
      // El "equivalente" usando StringBuffer:
      StringBuffer sb =
        new StringBuffer("abc"); // ¡Crea un String!
      sb.append(foo);
      sb.append("def"); // ¡Crea un String!
      sb.append(Integer.toString(47));
      System.out.println(sb);
    }
} ///:~
```

En la creación del **String s**, el compilador está haciendo el equivalente al código subsecuente que usa **sb**: se crea un **StringBuffer** y se usa **append( )** para añadir nuevos caracteres de forma directa al mismo (en vez de hacer una copia del mismo cada vez). Mientras que esto es más eficiente, merece la pena recalcar que cada vez que se crea una cadena de caracteres entre comillas, como **"abc"** y **"def"**, el compilador las convierte en objetos **String**. Por tanto, puede que se creen más objetos de los esperados, a pesar de la eficiencia lograda con el uso de **StringBuffer**.

Las clases **String** y **StringBuffer**

He aquí un repaso de los métodos disponibles tanto para **String** como para **StringBuffer** de forma que se pueda tener una idea de cómo interactúan. Estas tablas no contienen todos y cada uno de los métodos, sino aquéllos que son importantes en esta discusión. Los métodos sobrecargados vienen resumidos en una única fila.

En primer lugar, la clase **String**:

Método	Parámetros, Sobrecarga	Uso
boolean	–	–
Constructor	Sobrecargados: Default, **String**, **StringBuffer**, **char**, arrays, byte **arrays.**	Creación de objetos **String**.
length( )		Número de caracteres del **String**.
charAt( )	**int** Indice	El char en cierta posición del **String**.
getChars( ), getBytes()	El principio y el fin del que copiar, el array en el que copiar, un índice al array destino.	Copiar **chars** o **bytes** a un array externo.

Método	Parámetros, Sobrecarga	Uso
toCharArray( )		Produce un **char[]** que contiene los caracteres del **String**.
equals( ), equalsIgnoreCase( )	Un **String** con el que compararse.	Una comprobación de igualdad sobre los contenidos de dos **String**s.
compareTo( )	Un **String** con el que compararse.	El resultado es negativo, cero o positivo dependiendo del orden lexicográfico del **String** y del parámetro. ¡Distingue mayúsculas y minúsculas!
regionMatches( )	Desplazamiento en este **String**, el otro **String**, y su desplazamiento y longitud para comparar. La sobrecarga añade "ignorar las mayúsculas / minúsculas".	El resultado **boolean** indica si la región coincide.
startsWith( )	**String** con el que podría empezar. La sobrecarga añade desplazamiento al parámetro.	El resultado **boolean** indica si el **String** comienza con el parámetro.
endsWith( )	**String** que podría ser sufijo de este **String.**	El resultado **boolean** indica si el parámetro es o no un sufijo del **String**.
indexOf( ), lastIndexOf( )	Sobrecargado: **char**, **char** e índice de comienzo, **String**, **String** e índice de comienzo.	Devuelve -1 si no se encuentra el parámetro dentro del **String**, de otra forma devuelve el índice en el que comienza el parámetro. **lastIndexOf( )** busca desde atrás hacia delante.
substring( )	Sobrecargado: índice de comienzo, índices de comienzo y final.	**String** que contiene el conjunto de caracteres especificado.

Método	Parámetros, Sobrecarga	Uso
concat( )	El **String** a concatenar.	Devuelve un nuevo objeto **String** conteniendo los caracteres del **String** original seguidos de los caracteres del parámetro.
replace( )	El carácter que buscar, y el nuevo con el que reemplazarlo.	Devuelve un nuevo objeto **String** en el que se han hecho los reemplazos. Usa el **String** viejo si no se encuentra ninguna ocurrencia.
toLowerCase( ), toUpperCase( )		Devuelve un nuevo objeto **String** con todas las letras cambiadas a mayúsculas o minúsculas, devolviendo el original si no se hacen cambios.
trim( )		Devuelve un nuevo objeto **String** quitándole los espacios en blanco del final, o el mismo si no se hacen cambios.
valueOf( )	Sobrecargado: **Object**, **char[]**, **char[]** y desplazamiento y cuenta, **boolean**, **char**, **int**, **long**, **float**, **double**.	Devuelve un **String** que contiene una representación del parámetro en forma de caracteres.
intern( )		Produce una y sólo una referencia a **String** por cada secuencia de caracteres.

Como puede verse, todo método de **String** devuelve cuidadosamente un nuevo objeto **String** cuando es necesario cambiar su contenido. Nótese también que si los contenidos no varían el método, simplemente devuelve una referencia al **String** original. Esto ahorra espacio de almacenamiento y sobrecarga.

He aquí la clase **StringBuffer**:

Método	Parámetros, Sobrecarga	Uso
Constructor	Sobrecargados: Default, longitud del espacio de almacenamiento intermedio a crear, **String** del que crearlo.	Creación de objetos **StringBuffer**.
toString( )		Crea un **String** a partir del **StringBuffer**.
length( )		Número de caracteres del **StringBuffer**.
capacity( )		Devuelve la cantidad de espacios asignados.
ensure-Capacity( )	Entero que indica la capacidad deseada.	Hace que el **StringBuffer** almacene al menos el número de espacios deseado.
setLength( )	Entero que indica la nueva longitud de la cadena de caracteres del espacio de almacenamiento intermedio.	Trunca o expande la cadena de caracteres anterior. Si se expande, se rellena con valores nulos.
charAt( )	Entero que indica la posición del elemento deseado.	Devuelve el **char** que hay en esa posición del espacio de almacenamiento intermedio.
setCharAt( )	Entero que indica la posición del elemento deseado y el nuevo valor **char** para ese elemento.	Modifica el valor de esa posición.
getChars( )	El principio y el final desde el que copiar, el array en el que copiar, un índice al array de destino.	Copia **char**s a un array externo. No hay **getBytes( )** como en **String**.
append( )	Sobrecargado: **Object**, **String**, **char[]**, **char[]** con desplazamiento y longitud, **boolean**, **char**, **int**, **long**, **float**, **double**.	El parámetro se convierte a String y se añade al final del espacio de almacenamiento intermedio actual, incrementándolo si es necesario.

Método	Parámetros, Sobrecarga	Uso
insert( )	Sobrecargado, cada uno con un primer parámetro del desplazamiento en el que empezar a insertar: **Object**, **String**, **char[]**, **boolean**, **char**, **int**, **long**, **float**, **double**.	El segundo parámetro se convierte a String y se inserta en el espacio de almacenamiento intermedio actual a partir del desplazamiento, incrementando el espacio de almacenamiento intermedio si es necesario.
reverse( )		Se invierte el orden de los caracteres en el espacio de almacenamiento intermedio.

El método más comúnmente usado es **append( )**, usado por el compilador al evaluar expresiones **String** que contienen los operadores "+" y "+=". El método **insert( )** tiene una forma similar y ambos métodos llevan a cabo manipulaciones significativas sobre el espacio de almacenamiento intermedio en vez de crear nuevos objetos.

Los **Strings** son especiales

Hasta el momento se ha visto que la clase **String** no es simplemente otra clase en Java. Hay muchos casos especiales en **String**, siendo todos ellos clases predefinidas y fundamentales para Java. Después está el hecho de que una cadena de caracteres entre comillas se convierte en **String**, por parte del compilador, además de los operadores + y +=. En este apéndice se ha visto el caso especial que quedaba: la inmutabilidad construida tan cuidadosamente usando la clase amiga **StringBuffer** y alguna magia extra por parte del compilador.

Resumen

Dado que en Java todo son referencias, y dado que todo objeto se crea en el montículo y es eliminado por el recolector de basura sólo cuando se deja de usar, el sentido de la manipulación de objetos varía, especialmente al pasar y retornar objetos. Por ejemplo, en C y C++, si se desea inicializar algún espacio de almacenamiento en un método, generalmente se pide que el usuario pase la dirección de ese espacio al método. Si no, habría que preocuparse de quién es responsable de destruir ese espacio cuando se acabe con él. Por consiguiente, la interfaz y el entendimiento de esos métodos es más complicado. Pero en Java no hay que preocuparse nunca por la responsabilidad o por si un objeto seguirá existiendo cuando sea necesario, puesto que el propio lenguaje se encarga de todo ello. Se puede crear un objeto en el momento en que se necesite, y no antes, y nunca preocuparse de la mecánica del paso de responsabilidades relativas a ese objeto: simplemente se pasa la referencia. En ocasiones la simplificación que esto proporciona pasa desapercibida, otras veces es abrumadora.

El inconveniente de toda esta magia subyacente es doble:

1. Siempre se tiene una disminución de eficiencia por la gestión extra de memoria (aunque esta disminución puede no ser muy grande), y siempre hay cierta cantidad de incertidumbre sobre el tiempo que puede llevar ejecutar algo (puesto que puede forzarse a que actúe el recolector de basura siempre que a uno le quede poca memoria). En la mayoría de aplicaciones, los beneficios son superiores a los inconvenientes, y en particular, es posible escribir secciones críticas en el tiempo utilizando métodos **native** (ver Apéndice B).

2. Uso de alias: en ocasiones se puede acabar teniendo de forma accidental dos referencias al mismo objeto, lo cual sólo es un problema si se presupone que ambas apuntan a objetos *diferentes*. Es aquí donde es necesario prestar un poco más de atención y, cuando sea necesario, **clone( )** (clonar) un objeto para evitar que otra referencia se sorprenda por un cambio inesperado. De forma alternativa, se puede soportar el uso de alias persiguiendo eficiencia creando objetos inmutables cuyas operaciones pueden devolver nuevos objetos del mismo o de otro tipo, pero nunca cambiar el objeto original, de forma que nadie que haga referencia al mismo perciba los cambios.

Algunos opinan que en Java la clonación es precisamente un alarde de buen diseño, por lo que, en aras de usarlo, implementan su propia versión del clonado[6] no llamando nunca al método **Object.clone( )**, y eliminando así la necesidad de implementar **Cloneable**, y capturar la **CloneNotSupportedException**. Éste es un enfoque bastante razonable y dado que **clone( )** tiene tan poco soporte en la biblioteca estándar de Java, es también bastante seguro. Pero en la medida en que no se invoque a **Object.clone( )** no hay por qué implementar **Cloneable** o capturar la excepción, por lo que esto también parece algo aceptable.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Demostrar un segundo nivel de uso de alias. Crear un método que tome una referencia a un objeto pero no modifique el objeto de la misma. Sin embargo, el método invoca a un segundo método, pasándole la referencia, y este segundo método sí que modifica el objeto.

2. Crear una clase **miCadena** que contenga un objeto **String** que se inicialice en el constructor, usando el parámetro al mismo. Añadir un método **toString( )** y un método **concatenar( )** que añada un objeto **String** a la cadena de caracteres interna. Implementar **clone( )** en **miCadena**. Crear dos métodos **static**, cada uno de los cuales tome como parámetro una referencia **miCadena x**, e invocar a **x.concatenar("prueba")**, pero en el segundo método, llamando primero a **clone( )**. Probar ambos métodos y mostrar los distintos efectos entre sí.

---

[6] Doug Lea, que me ayudó a resolver este aspecto, me lo sugirió, diciéndome que simplemente crea una función de nombre **duplicate( )** para cada clase.

3. Crear una clase denominada **Pila** que contenga un **int** que es un número de pila (un identificador único). Hacerlo clonable y darle un método **toString( )**. Crear ahora una clase llamada **Juguete** que contenga un array de **Pila** y un **toString( )** que imprima todas sus pilas. Escribir un método **clone()** para **Juguete** que clone automáticamente todos sus objetos **Pila**. Probarlo clonando **Juguete** e imprimiendo el resultado.

4. Cambiar **ComprobarCloneable.java** de forma que todos los métodos **clone( )** capturen la **CloneNotSupportedException** en vez de pasársela al llamador.

5. Utilizando la técnica de la clase amigo mutable, construir una clase inmutable que contenga un **int**, un **double** y un array de **char**.

6. Modificar **Competir.java** para añadir más objetos miembros a las clases **Cosa2** y **Cosa4** y ver si se puede determinar cómo varían los tiempos con la complejidad —si se trata de una relación lineal o parece algo más complicada.

7. A partir de **Serpiente.java**, crear una versión de copia profunda de la serpiente.

8. Heredar un **ArrayList** y hacer que su método **clone( )** lleve a cabo una copia en profundidad.