TutorJava Nivel Básico
Autor-Traductor: Juan Antonio Palos (Ozito)
Puedes encontrar la Version Original en Ingles en (http://java.sun.com)

Indice de contenidos

Me ha parecido un tutorial, muy facil y sencillo de seguir.

TutorJava Nivel Básico

En esta página:

Conceptos básicos de Programacion Orientada a Objetos

¿Qué son los objetos?

En informática, un OBJETO es un conjunto de variables y de los métodos relacionados con esas variables.

Un poco más sencillo: un objeto contiene en sí mismo la información y los métodos o funciones necesarios para manipular esa información.

Lo más importante de los objetos es que permiten tener un control total sobre 'quién' o 'qué' puede acceder a sus miembros, es decir, los objetos pueden tener miembros públicos a los que podrán acceder otros objetos o miembros privados a los que sólo puede acceder él. Estos miembros pueden ser tanto variables como funciones.

El gran beneficio de todo esto es la encapsulación, el código fuente de un objeto puede escribirse y mantenerse de forma independiente a los otros objetos contenidos en la aplicación.

¿Qué son las clases?

Una CLASE es un proyecto, o prototipo, que define las variables y los métodos comunes a un cierto tipo de objetos.

Un poco más sencillo: las clases son las matrices de las que luego se pueden crear múltiples objetos del mismo tipo. La clase define las variables y los métodos comunes a los objetos de ese tipo, pero luego, cada objeto tendrá sus propios valores y compartirán las mismas funciones.

Primero deberemos crear una clase antes de poder crear objetos o ejemplares de esa clase.

¿Qué son los mensajes?

Para poder crear una aplicación necesitarás más de un objeto, y estos objetos no pueden estar aislados unos de otros, pues bien, para comunicarse esos objetos se envían mensajes.

Los mensajes son simples llamadas a las funciones o métodos del objeto con el se quiere comunicar para decirle que haga cualquier cosa.

¿Qué es la herencia?

Qué significa esto la herencia, quíen hereda qué; bueno tranquilo, esto sólo significa que puedes crear una clase partiendo de otra que ya exista.

Es decir, puedes crear una clase a través de una clase existente, y esta clase tendrá todas las variables y los métodos de su 'superclase', y además se le podrán añadir otras variables y métodos propios.

Se llama 'Superclase' a la clase de la que desciende una clase, puedes ver más sobre la declaración de clases en la página Declarar Clases.

TutorJava Nivel Básico

En esta página:

Variables y Tipos de Datos

Las variables son las partes importantes de un lenguaje de programación: ellas son las entidades (valores, datos) que actúan y sobre las que se actúa.

Una declaración de variable siempre contiene dos componentes, el tipo de la variable y su nombre.

```
tipoVariable nombre;
```

Tipos de Variables

Todas las variables en el lenguaje Java deben tener un tipo de dato. El tipo de la variable determina los valores que la variable puede contener y las operaciones que se pueden realizar con ella.

Existen dos categorias de datos principales en el lenguaje Java: los tipos primitivos y los tipos referenciados.

Los tipos primitivos contienen un sólo valor e incluyen los tipos como los enteros, coma flotante, los caracteres, etc... La tabla siguiente muestra todos los tipos primitivos soportados por el lenguaje Java, su formato, su tamaño y una breve descripción de cada uno.

Tipo	Tamaño/Formato	Descripción
(Números enteros)		
byte	8-bit complemento a 2	Entero de un Byte
short	16-bit complemento a 2	Entero corto
int	32-bit complemento a 2	Entero
long	64-bit complemento a 2	Entero largo
(Números reales)		
float	32-bit IEEE 754	Coma flotante de precisión simple
double	64-bit IEEE 754	Coma flotante de precisión doble
(otros tipos)		
char	16-bit Caracter	Un sólo carácter
boolean	true o false	Un valor booleano (verdadero o falso)

Los tipos referenciados se llaman así porque el valor de una variable de referencia es una referencia (un puntero) hacia el valor real. En Java tenemos los arrays, las clases y los interfaces como tipos de datos referenciados.

Nombres de Variables

Un programa se refiere al valor de una variable por su nombre. Por convención, en Java, los nombres de las variables empiezan con una letra minúscula (los nombres de las clases empiezan con una letra mayúscula).

Un nombre de variable Java.

1. debe ser un identificador legal de Java comprendido en una serie de caracteres Unicode. Unicode es un sistema de codificación que soporta texto escrito en distintos lenguajes humanos.Unicode perminte la codificación de 34.168 caracteres. Esto le permite utilizar en sus programas Java varios alfabetos

como el Japonés, el Griego, el Ruso o el Hebreo. Esto es importante para que los programadores pueden escribir código en su lenguaje nativo.

2. no puede ser el mismo que una palabra clave o el nombre de un valor booleano (true or false)
3. no deben tener el mismo nombre que otras variables cuyas declaraciones aparezcan en el mismo ámbito.

La regla número 3 implica que podría existir el mismo nombre en otra variable que aparezca en un ámbito diferente.

Por convención, los nombres de variables empiezan por un letra minúscula. Si una variable está compuesta de más de una palabra, como 'nombreDato' las palabras se ponen juntas y cada palabra después de la primera empieza con una letra mayúscula.

TutorJava Nivel Básico

En esta página:

Operadores de Java

Los operadores realizan algunas funciones en uno o dos operandos. Los operadores que requieren un operador se llaman operadores unarios. Por ejemplo, ++ es un operador unario que incrementa el valor su operando en uno.

Los operadores que requieren dos operandos se llaman operadores binarios. El operador = es un operador binario que asigna un valor del operando derecho al operando izquierdo.

Los operadores unarios en Java pueden utilizar la notación de prefijo o de sufijo. La notación de prefijo significa que el operador aparece antes de su operando.

```
operador operando
```

La notación de sufijo significa que el operador aparece después de su operando:

```
operando operador
```

Todos los operadores binarios de Java tienen la misma notación, es decir aparencen entre los dos operandos:

```
op1 operator op2
```

Además de realizar una operación también devuelve un valor. El valor y su tipo dependen del tipo del operador y del tipo de sus operandos. Por ejemplo, los operadores aritméticos (realizan las operaciones de aritmética básica como la suma o la resta) devuelven números, el resultado típico de las operaciones aritmétcias. El tipo de datos devuelto por los operadores aritméticos depende del tipo de sus operandos: si sumas dos enteros, obtendrás un entero. Se dice que una operación evalúa su resultado.

Es muy útil dividir los operadores Java en las siguientes categorías: aritméticos, relacionales y condicionales. lógicos y de desplazamiento y de asignación.

Operadores Aritméticos

El lenguaje Java soporta varios operadores aritéticos - incluyendo + (suma), - (resta), * (multiplicación), / (división), y % (módulo)-- en todos los números enteros y de coma flotante. Por ejemplo, puedes utilizar este código Java para sumar dos números:

```
sumaEsto + aEsto
```

O este código para calcular el resto de una división:

```
divideEsto % porEsto
```

Esta tabla sumariza todas las operaciones aritméticas binarias en Java.

Operador	Uso	Descripción
+	op1 + op2	Suma op1 y op2
-	op1 - op2	Resta op2 de op1
*	op1 * op2	Multiplica op1 y op2
/	op1 / op2	Divide op1 por op2

%	op1 % op2	Obtiene el resto de dividir op1 por op2

Nota: El lenguaje Java extiende la definición del operador + para incluir la concatenación de cadenas.

Los operadores + y - tienen versiones unarias que seleccionan el signo del operando.

Operador	Uso	Descripción
+	+ op	Indica un valor positivo
-	- op	Niega el operando

Además, existen dos operadores de atajos aritméticos, ++ que incrementa en uno su operando, y -- que decrementa en uno el valor de su operando.

Operador	Uso	Descripción
++	op ++	Incrementa op en 1; evalúa el valor antes de incrementar
++	++ op	Incrementa op en 1; evalúa el valor después de incrementar
--	op --	Decrementa op en 1; evalúa el valor antes de decrementar
--	-- op	Decrementa op en 1; evalúa el valor después de decrementar

Operadores Relacionales y Condicionales

Los valores relacionales comparan dos valores y determinan la relación entre ellos. Por ejemplo, != devuelve true si los dos operandos son distintos.

Esta tabla sumariza los operadores relacionales de Java.

Operador	Uso	Devuelve true si
>	op1 > op2	op1 es mayor que op2
>=	op1 >= op2	op1 es mayor o igual que op2
<	op1 < op2	op1 es menor que op2
<=	op1 <= op2	op1 es menor o igual que op2
==	op1 == op2	op1 y op2 son iguales
!=	op1 != op2	op1 y op2 son distintos

Frecuentemente los operadores relacionales se utilizan con otro juego de operadores, los operadores condicionales, para construir expresiones de decisión más complejas. Uno de estos operadores es && que realiza la operación Y booleana . Por ejemplo puedes utilizar dos operadores relacionales diferentes junto con && para determinar si ambas relaciones son ciertas. La siguiente línea de código utiliza esta técnica para determinar si un indice de un array está entre dos límites- esto es, para determinar si el indice es mayor que 0 o menor que NUM_ENTRIES (que se ha definido préviamente como un valor constante):

```
0 &lt; index && index &lt NUM_ENTRIES
```

Observa que en algunas situaciones, el segundo operando de un operador relacional no será evaluado. Consideremos esta sentencia:

```
((count &gt; NUM_ENTRIES) && (System.in.read() != -1))
```

Si count es menor que NUM_ENTRIES, la parte izquierda del operando de && evalúa a false. El operador && sólo devuelve true si los dos operandos son verdaderos. Por eso, en esta situación se puede deteminar el valor de && sin evaluar el operador de la derecha. En un caso como este, Java no evalúa el operando de la derecha. Así no se llamará a System.in.read() y no se leerá un carácter de la entrada estandard.

Aquí tienes tres operadores condicionales.

Operador	Uso	Devuelve true si
&&	op1 && op2	op1 y op2 son verdaderos
||	op1 || op2	uno de los dos es verdadero
!	! op	op es falso

El operador & se puede utilizar como un sinónimo de && si ambos operadores son booleanos. Similarmente, | es un sinonimo de || si ambos operandos son booleanos.

Operadores de Desplazamiento

Los operadores de desplazamiento permiten realizar una manipualción de los bits de los datos. Esta tabla sumariza los operadores lógicos y de desplazamiento disponibles en el lenguaje Java.

Operador	Uso	Descripción
>>	op1 >> op2	desplaza a la derecha op2 bits de op1
<<	op1 << op2	desplaza a la izquierda op2 bits de op1
>>>	op1 >>> op2	desplaza a la derecha op2 bits de op1(sin signo)
&	op1 & op2	bitwise and
|	op1 | op2	bitwise or
^	op1 ^ op2	bitwise xor
~	~ op	bitwise complemento

Los tres operadores de desplazamiento simplemente desplazan los bits del operando de la izquierda el número de posiciones indicadas por el operador de la derecha. Los desplazamientos ocurren en la dirección indicada por el propio operador. Por ejemplo:

```
13 &gt;&gt; 1;
```

desplaza los bits del entero 13 una posición a la derecha. La representación binaria del número 13 es 1101. El resultado de la operación de desplazamiento es 110 o el 6 decimal. Observe que el bit situado más a la derecha desaparece. Un desplazamiento a la derecha de un bit es equivalente, pero más eficiente que, dividir el operando de la izquierda por dos. Un desplazamiento a la izquierda es equivalente a multiplicar por dos.

Los otros operadores realizan las funciones lógicas para cada uno de los pares de bits de cada operando. La función "y" activa el bit resultante si los dos operandos son 1.

op1	op2	resultado
0	0	0
0	1	0
1	0	0
1	1	1

Supon que quieres evaluar los valores 12 "and" 13:

```
12 & 13
```

El resultado de esta operación es 12. ¿Por qué? Bien, la representación binaria de 12 es 1100 y la de 13 es 1101. La función "and" activa los bits resultantes cuando los bits de los dos operandos son 1, de otra forma el resultado es 0. Entonces si colocas en línea los dos operandos y realizas la función "and", puedes ver que los dos bits de mayor peso (los dos bits situados más a la izquierda de cada número) son 1 así el bit resultante de cada uno es 1. Los dos bits de menor peso se evalúan a 0 poque al menos uno de los dos operandos es 0:

```
        1101
  &     1100
        ------
        1100
```

El operador | realiza la operación O inclusiva y el operador ^ realiza la operación O exclusiva. O inclusiva significa que si uno de los dos operandos es 1 el resultado es 1.

op1	op2	resultado
0	0	0
0	1	1
1	0	1
1	1	1

O exclusiva significa que si los dos operandos son diferentes el resultado es 1, de otra forma el resultado es 0.

op1	op2	resultado
0	0	0
0	1	1
1	0	1
1	1	0

Y finalmente el operador complemento invierte el valor de cada uno de los bites del operando: si el bit del operando es 1 el resultado es 0 y si el bit del operando es 0 el resultado es 1.

Operadores de Asignación

Puedes utilizar el operador de asignación =, para asignar un valor a otro. Además del operador de asignación básico, Java proporciona varios operadores de asignación que permiten realizar operaciones aritmétcias, lógicas o de bits y una operación de asignación al mismo tiempo. Especificamente, supon que quieres añadir un número a una variable y asignar el resultado dentro de la misma variable, como esto:

```
i = i + 2;
```

Puedes ordenar esta sentencia utilizando el operador +=.

```
i += 2;
```

Las dos líneas de código anteriores son equivalentes.

Esta tabla lista los operadores de asignación y sus equivalentes.

Operador	Uso	Equivale a
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 &gt> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

TutorJava Nivel Básico

En esta página:

Expresiones Java

Definicion de Expresión

Las expresiones realizan el trabajo de un programa Java. Entre otras cosas, las expresiones se utilizan para calcular y asignar valores a las variables y para controlar el flujo de un programa Java. El trabajo de una expresión se divide en dos partes: realizar los cálculos indicados por los elementos de la expresión y devolver algún valor.

Definición: Una expresión es una serie de variables, operadores y llamadas a métodos (construida de acuerdo a la sintaxis del lenguaje) que evalúa a un valor sencillo.

El tipo del dato devuelto por una expresión depende de los elementos utilizados en la expresión. La expresión count++ devuelve un entero porque ++ devuelve un valor del mismo tipo que su operando y count es un entero. Otras expresiones devuelven valores booleanos, cadenas, etc...

Una expresión de llamada a un método devuelve el valor del método; así el tipo de dato de una expresión de llamada a un método es el mismo tipo de dato que el valor de retorno del método. El método System.in.read() se ha declarado como un entero, por lo tanto, la expresión System.in.read() devuelve un entero.

La segunda expresión contenida en la sentencia System.in.read() != -1 utiliza el operador !=.

Recuerda que este operador comprueba si los dos operandos son distintos. En esta sentencia los operandos son System.in.read() y -1.

System.in.read() es un operando válido para != porque devuelve un entero. Así System.in.read() != -1 compara dos enteros, el valor devuelto por System.in.read() y -1.

El valor devuelto por != es true o false dependiendo de la salida de la comparación.

Como has podido ver, Java te permite construir expresiones compuestas y sentencias a partir de varias expresiones pequeñas siempre que los tipos de datos requeridos por una parte de la expresión correspondan con los tipos de datos de la otra.

Tambíen habrás podido concluir del ejemplo anterior, el orden en que se evalúan los componentes de una expresión compuesta.

Por ejemplo, toma la siguiente expresión compuesta.

```
x * y * z
```

En este ejemplo particular, no importa el orden en que se evalúe la expresión porque el resultado de la multiplicación es independiente del orden. La salida es siempre la misma sin importar el orden en que se apliquen las multiplicaciones. Sin embargo, esto no es cierto para todas las expresiones. Por ejemplo, esta expresión obtiene un resultado diferente dependiendo de si se realiza primero la suma o la división.

```
x + y / 100
```

Puedes decirle directamente al compilador de Java cómo quieres que se evalúe una expresión utilizando los paréntesis (y).

Por ejemplo, para aclarar la sentencia anterior, se podría escribir: (x + y)/ 100.

Si no le dices explícitamente al compilador el orden en el que quieres que se realicen las operaciones, él decide basándose en la precedencia asignada a los operadores y otros elementos

que se utilizan dentro de una expresión.

Los operadores con una precedencia más alta se evalúan primero. Por ejemplo. el operador división tiene una precedencia mayor que el operador suma, por eso, en la expresión anterior x + y / 100, el compilador evaluará primero y / 100. Así

```
x + y / 100
```

es equivalente a.

```
x + (y / 100)
```

Para hacer que tu código sea más fácil de leer y de mantener deberías explicar e indicar con parétesis los operadores que se deben evaluar primero.

La tabla siguiente muestra la precedencia asignada a los operadores de Java. Los operadores se han listado por orden de precedencia de mayor a menor. Los operadores con mayor precedencia se evalúan antes que los operadores con un precedencia relativamente menor. Lo operadores con la misma precedencia se evalúan de izquierda a derecha.

Precedencia de Operadores en Java

operadores sufijo	[] . (params) expr++ expr--
operadores unarios	++expr --expr +expr -expr ~ !
creación o tipo	new (type)expr
multiplicadores	* / %
suma/resta	+ -
desplazamiento	<< >> >>>
relacionales	< > <= >= instanceof
igualdad	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
AND lógico	&&
OR lógico	||
condicional	? :
asignación	= += -= *= /= %= ^= &= |= <<= >>= >>>=

En esta página:

Sentencias de Control de Flujo en Java

Las sentencias de control de flujo determinan el orden en que se ejecutarán las otras sentencias dentro del programa. El lenguaje Java soporta varias sentencias de control de flujo, incluyendo.

Sentencias	palabras clave
toma de decisiones	if-else, switch-case
bucles	for, while, do-while
excepciones	try-catch-finally, throw
miscelaneas	break, continue, label:, return

Nota: Aunque goto es una palabra reservada, actualmente el lenguaje Java no la soporta. Podemos utilizar las rupturas etiquetadas en su lugar.

La sentencia if-else

La sentencia if-else de java proporciona a los programas la posibilidad de ejecutar selectivamente otras sentencias basándose en algún criterio.

Por ejemplo, supon que tu programa imprime información de depurado basándose en el valor de una variable booleana llamada DEBUG. Si DEBUG fuera verdadera true, el programa imprimiría la información de depurado, como por ejemplo, el valor de una variable como x. Si DEBUG es false el programa procederá normalmente. Un segmento de código que implemente esto se podría parecer a este.

```
. . .
if (DEBUG)
    System.out.println("DEBUG: x = " + x);
. . .
```

Esta es la versión más sencilla de la sentencia if: la sentencia gobernada por if se ejecuta si alguna codición es verdadera. Generalmente, la forma sencilla de if se puede escribir así.

```
if (expresión)
    sentencia
```

Pero, ¿y si quieres ejecutar un juego diferente de sentencias si la expresión es falsa? Bien, puedes utilizar la sentencia else. Echemos un vistazo a otro ejemplo. Supon que tu programa necesita realizar diferentes acciones dependiendo de que el usuario pulse el botón OK o el botón Cancel en un ventana de alarma. Se podría hacer esto utilizando una sentencia if.

```
    . . .

    // Respuesta dependiente del botoón que haya pulsado el usuario
    // OK o Cancel
    . . .
if (respuesta == OK) {
    . . .
```

```
    // Código para la acción OK
    . . .
} else {
    . . .
    // código para la acción Cancel
    . . .
}
```

Este uso particular de la sentencia else es la forma de capturarlo todo.

Existe otra forma de la sentecia else, else if que ejecuta una sentencia basada en otra expresión. Por ejemplo, supon que has escrito un programa que asigna notas basadas en la puntuación de un examen, un Sobresaliente para una puntuación del 90% o superior, un Notable para el 80% o superior y demás. odrías utilizar una sentencia if con una serie de comparaciones else if y una setencia else para escribir este código.

```
int puntuacion;
String nota;

if (puntuacion >= 90) {
    nota = "Sobresaliente";
} else if (puntuacion >= 80) {
    nota = "Notable";
} else if (puntuacion >= 70) {
    nota = "Bien";
} else if (puntuacion >= 60) {
    nota = "Suficiente";
} else {
    nota = "Insuficiente";
}
```

Una sentencia if puede tener cualquier número de sentencias de acompañamiento else if.

Podrías haber observado que algunos valores de puntuacion pueden satisfacer más una de las expresiones que componen la sentencia if. Por ejemplo, una puntuación de 76 podría evaluarse como true para dos expresiones de esta sentencia:puntuacion >= 70 y puntuacion >= 60.

Sin embargo, en el momento de ejecución, el sistema procesa una sentencia if compuesta como una sóla; una vez que se ha satisfecho una condición (76 >= 70), se ejecuta la sentencia apropiada (nota = "Bien";), y el control sale fuera de la sentencia if sin evaluar las condiciones restantes.

La sentencia switch

La sentencia switch se utiliza para realizar sentencias condicionalmente basadas en alguna expresión. Por ejemplo, supon que tu programa contiene un entero llamado mes cuyo valor indica el mes en alguna fecha. Supon que también quieres mostrar el nombre del mes basándose en su número entero equivalente. Podrías utilizar la sentencia switch de Java para realizar esta tarea.

```
int mes;
. . .
switch (mes) {
case 1:  System.out.println("Enero"); break;
case 2:  System.out.println("Febrero"); break;
case 3:  System.out.println("Marzo"); break;
case 4:  System.out.println("Abril"); break;
case 5:  System.out.println("May0"); break;
case 6:  System.out.println("Junio"); break;
case 7:  System.out.println("Julio"); break;
case 8:  System.out.println("Agosto"); break;
case 9:  System.out.println("Septiembre"); break;
case 10: System.out.println("Octubre"); break;
case 11: System.out.println("Noviembre"); break;
case 12: System.out.println("Diciembre"); break;
}
```

La sentencia switch evalúa su expresión, en este caso el valor de mes, y ejecuta la sentencia case apropiada.

Decidir cuando utilizar las sentencias if o switch dependen del juicio personal. Puedes decidir cual utilizar basándose en la buena lectura del código o en otros factores.

Cada sentencia case debe ser única y el valor proporcionado a cada sentencia case debe ser del mismo tipo que el tipo de dato devuelto por la expresión proporcionada a la sentencia switch.

Otro punto de interes en la sentencia switch son las sentencias break después de cada case.

La sentencia break hace que el control salga de la sentencia switch y continúe con la siguiente línea.

La sentencia break es necesaria porque las sentencias case se siguen ejecutando hacia abajo. Esto es, sin un break explícito, el flujo de control seguiría secuencialmente a través de las sentencias case siguientes.

En el ejemplo anterior, no se quiere que el flujo vaya de una sentencia case a otra, por eso se han tenido que poner las sentencias break.

Sin embargo, hay ciertos escenario en los que querrás que el control proceda secuencialmente a través de las sentencias case. Como este código que calcula el número de días de un mes de acuerdo con el ritmico refrán que dice "Treinta dias tiene Septiembre...".

```
int mes;
int numeroDias;
. . .
switch (mes) {
case 1.
case 3.
case 5.
case 7.
case 8.
case 10.
case 12.
    numeroDias = 31;
    break;
case 4.
case 6.
case 9.
case 11.
    numeroDias = 30;
    break;
case 2.
    if ( ((ano % 4 == 0) && !(ano % 100 == 0)) || ano % 400 == 0) )
        numeroDias = 29;
    else
        numeroDias = 28;
    break;
}
```

Finalmente, puede utilizar la sentencia default al final de la sentencia switch para manejar los valores que no se han manejado explícitamente por una de las sentencias case.

```
int mes;
. . .
switch (mes) {
case 1:  System.out.println("Enero"); break;
case 2:  System.out.println("Febrero"); break;
case 3:  System.out.println("Marzo"); break;
case 4:  System.out.println("Abril"); break;
case 5:  System.out.println("Mayo"); break;
case 6:  System.out.println("Junio"); break;
case 7:  System.out.println("Julio"); break;
case 8:  System.out.println("Agosto"); break;
case 9:  System.out.println("Septiembre"); break;
case 10: System.out.println("Octubre"); break;
case 11: System.out.println("Noviembre"); break;
case 12: System.out.println("Diciembre"); break;
default: System.out.println("Ee, no es un mes válido!");
    break;
}
```

Sentencias de Bucle

Generalmente hablando, una sentencia while realiza una acción mientras se cumpla una cierta condición. La sintaxis general de la sentencia while es.

```
while (expresión)
    sentencia
```

Esto es, mientras la expresión sea verdadera, ejecutará la sentencia.

sentencia puede ser una sóla sentencia o puede ser un bloque de sentencias. Un bloque de sentencias es un juego de sentencias legales de java contenidas dentro de corchetes('{'y '}').

Por ejemplo, supon que además de incrementar contador dentro de un bucle while también quieres imprimir el contador cada vez que se lea un carácter. Podrías escribir esto en su lugar.

```
. . .
while (System.in.read() != -1) {
    contador++;
    System.out.println("Se ha leido un el carácter = " + contador);
}
. . .
```

Por convención el corchete abierto '{' se coloca al final de la misma línea donde se encuentra la sentencia while y el corchete cerrado '}' empieza una nueva línea indentada a la línea en la que se encuentra el while.

Además de while Java tiene otros dos constructores de bucles que puedes utilizar en tus programas.

el bucle for y el bucle do-while.

Primero el bucle for. Puedes utilizar este bucle cuando conozcas los límites del bucle (su instrucción de inicialización, su criterio de terminación y su instrucción de incremento). Por ejemplo, el bucle for se utiliza frecuentemente para iterar sobre los elementos de un array, o los caracteres de una cadena.

```
// a es un array de cualquier tipo
. . .
int i;
int length = a.length;
for (i = 0; i < length; i++) {
    . . .
    // hace algo en el elemento i del array a
    . . .
}
```

Si sabes cuando estas escribiendo el programa que quieres empezar en el inicio del array, parar al final y utilizar cada uno de los elementos. Entonces la sentencia for es una buena elección. La forma general del bucle for puede expresarse asi.

```
for (inicialización; terminación; incremento)
    sentencias
```

inicialización es la sentencia que inicializa el bucle -- se ejecuta una vez al iniciar el bucle.

terminación es una sentecia que determina cuando se termina el bucle. Esta expresión se evalúa al principio de cada iteracción en el bucle. Cuando la expreión se evalúa a false el bucle se termina.

Finalmente, incremento es una expresión que se invoca en cada interacción del bucle. Cualquiera (o todos) de estos componentes puden ser una sentencia vacía (un punto y coma).

Java proporciona otro bucle, el bucle do-while,que es similar al bucle while que se vió al principio, excepto en que la expresión se avalú al final del bucle.

```
do {
    sentencias
} while (Expresión Booleana);
```

La sentencia do-while se usa muy poco en la construcción de bucles pero tiene sus usos. Por ejemplo, es conveniente utilizar la sentencia do-while cuando el bucle debe ejecutarse al menos una vez. Por ejemplo, para leer información de un fichero, sabemos que al menos debe leer un carácter.

```
int c;
InputStream in;
. . .
do {
    c = in.read();
    . . .
} while (c != -1);
```

Sentencias de Manejo de Excepciones

Cuando ocurre un error dentro de un método Java, el método puede lanzar una excepción para indicar a su

llamador que ha ocurrido un error y que el error está utilizando la sentencia throw.

El método llamador puede utilizar las sentencias try, catch, y finally para capturar y manejar la excepción.

Puedes ver Manejar Errores Utilizando Excepciones para obtener más información sobre el lanzamiento y manejo de excepciones.

Sentencias de Ruptura

Ya has visto la sentencia break en acción dentro de la sentencia switch anteriormente. Como se observó anteriormente, la sentencia break hace que el control del flujo salte a la sentencia siguiente a la actual.

Hay otra forma de break que hace que el flujo de control salte a una sentencia etiquetada.

Se puede etiquetar una sentencia utilizando un identificador legal de Java (la etiqueta) seguido por dos puntos (:) antes de la sentencia.

```
SaltaAqui: algunaSentenciaJava
```

Para saltar a la sentencia etiquetada utilice esta forma de la sentencia break.

```
break SaltaAqui;
```

Las rupturas etiquetadas son una alternativa a la sentencia goto que no está soportada por el lenguaje Java.

Se puede utiliza la sentencia continue dentro de un bucle para saltar de la sentencia actual hacia el principio del bucle o a una sentencia etiquetada.

Considera esta implementación del método indexOf() de la clase String que utiliza la forma de continue que continúa en una sentencia etiquetada.

```
public int indexOf(String str, int fromIndex) {
    char[] v1 = value;
    char[] v2 = str.value;
    int max = offset + (count - str.count);
  test.
    for (int i = offset + ((fromIndex < 0) ? 0 : fromIndex); i <= max ; i++) {
        int n = str.count;
        int j = i;
        int k = str.offset;
        while (n-- != 0) {
            if (v1[j++] != v2[k++]) {
                continue test;
            }
        }
        return i - offset;
    }
    return -1;
}
```

Nota: Sólo se puede llamar a la sentencia continue desde dentro de un bucle.

Y finalmente la sentencia return.

Esta sentencia se utiliza para salir del método actual y volver a la sentencia siguiente a la que originó la llamada en el método original.

Existen dos formas de return: una que devuelve un valor y otra que no lo hace.

Para devolver un valor, simplemente se pone el valor (o una expresión que calcule el valor) detrás de la palabra return.

```
return ++count;
```

El valor devuelto por return debe corresponder con el tipo del valor de retorno de la declaración del método.

Cuando un método se declara como void utiliza la forma de return que no devuelve ningún valor.

```
return;
```

TutorJava Nivel Básico

En esta página:

Arrays y Cadenas en Java

Al igual que otros lenguajes de programación, Java permite juntar y manejar múltiples valores a través de un objeto array (matriz). También se pueden manejar datos compuestos de múltiples caracteres utilizando el objeto String (cadena).

Arrays

Esta sección te enseñará todo lo que necesitas para crear y utilizar arrays en tus programas Java.

Como otras variables, antes de poder utilizar un array primero se debe declarar. De nuevo, al igual que otras variables, la declaración de un array tiene dos componentes primarios: el tipo del array y su nombre. Un tipo de array incluye el tipo de dato de los elementos que va contener el array. Por ejemplo, el tipo de dato para un array que sólo va a contener elementos enteros es un array de enteros. No puede existir un array de tipo de datos genérico en el que el tipo de sus elementos esté indefinido cuando se declara el array. Aquí tienes la declaración de un array de enteros.

```
int[] arrayDeEnteros;
```

La parte int[] de la declaración indica que arrayDeEnteros es un array de enteros. La declaración no asigna ninguna memoria para contener los elementos del array.

Si se intenta asignar un valor o acceder a cualquier elemento de arrayDeEnteros antes de haber asignado la memoria para él, el compilador dará un error como este y no compilará el programa.

```
testing.java:64: Variable arraydeenteros may not have been initialized.
```

Para asignar memoria a los elementos de un array, primero se debe ejemplarizar el array. Se puede hacer esto utilizando el operador new de Java. (Realmente, los pasos que se deben seguir para crear un array son similares a los se deben seguir para crear un objeto de una clase: declaración, ejemplarización e inicialización.

La siguiente sentencia asigna la suficiente memoria para que arrayDeEnteros pueda contener diez enteros.

```
int[] arraydeenteros = new int[10]
```

En general, cuando se crea un array, se utiliza el operador new, más el tipo de dato de los elementos del array, más el número de elementos deseados encerrado entre cochetes cuadrados ('[' y ']').

```
TipodeElemento[] NombredeArray = new TipodeElementos[tamanoArray]
```

Ahora que se ha asignado memoria para un array ya se pueden asignar valores a los elemetos y recuperar esos valores.

```
for (int j = 0; j < arrayDeEnteros.length; j ++) {
    arrayDeEnteros[j] = j;
    System.out.println("[j] = " + arrayDeEnteros[j]);
```

```
}
```

Como se puede ver en el ejemplo anterior, para referirse a un elemento del array, se añade corchetes cuadrados al nombre del array. Entre los corchetes caudrados se indica (bien con una variable o con una expresión) el índice del elemento al que se quiere acceder. Observa que en Java, el índice del array empieza en 0 y termina en la longitud del array menos uno.

Hay otro elemento interesante en el pequeño ejemplo anterior. El bucle for itera sobre cada elemento de arrayDeEnteros asignándole valores e imprimiendo esos valores. Observa el uso de arrayDeEnteros.length para obtener el tamaño real del array. length es una propiedad proporcionada para todos los arrays de Java.

Los arrays pueden contener cualquier tipo de dato legal en Java incluyendo los tipos de referencia como son los objetos u otros array. Por ejemplo, el siguiente ejemplo declara un array que puede contener diez objetos String.

```
String[] arrayDeStrings = new String[10];
```

Los elementos en este array son del tipo referencia, esto es, cada elemento contiene una referencia a un objeto String. En este punto, se ha asignado suficiente memoria para contener las referencias a los Strings, pero no se ha asignado memoria para los propios strings. Si se intenta acceder a uno de los elementos de arraydeStrings obtendrá una excepción 'NullPointerException' porque el array está vacio y no contiene ni cadenas ni objetos String. Se debe asignar memoria de forma separada para los objetos String.

```
for (int i = 0; i < arraydeStrings.length; i ++) {
    arraydeStrings[i] = new String("Hello " + i);
}
```

Strings

Una secuencia de datos del tipo carácter se llama un string (cadena) y en el entorno Java está implementada por la clase String (un miembro del paquete java.lang).

```
String[] args;
```

Este código declara explícitamente un array, llamado args, que contiene objetos del tipo String. Los corchetes vacios indican que la longitud del array no se conoce en el momento de la compilación, porque el array se pasa en el momento de la ejecución.

El segundo uso de String es el uso de cadenas literales (una cadena de caracteres entre comillas " y ").

```
"Hola mundo!"
```

El compilador asigna implicitamente espacio para un objeto String cuando encuentra una cadena literal.

Los objetos String son inmutables - es decir, no se pueden modificar una vez que han sido creados.

El paquete java.lang proporciona una clase diferente, StringBuffer, que se podrá utilizar para crear y manipular caracteres al vuelo.

Concatenación de Cadenas

Java permite concatenar cadenas facilmente utilizando el operador +. El siguiente fragmento de código concatena tres cadenas para producir su salida.

```
"La entrada tiene " + contador + " caracteres."
```

Dos de las cadenas concatenadas son cadenas literales: "La entrada tiene " y " caracteres.". La tercera cadena - la del medio- es realmente un entero que primero se convierte a cadena y luego se concatena con las otras.

En esta página:

Crear Objetos en Java

En Java, se crea un objeto mediante la creación de un objeto de una clase o, en otras palabras, ejemplarizando una clase. Aprenderás cómo crear una clase más adelante en Crear Clases.

Hasta entonces, los ejemplos contenidos aquí crean objetos a apartir de clases que ya existen en el entorno Java.

Frecuentemente, se verá la creación de un objeto Java con un sentencia como esta.

```
Date hoy = new Date();
```

Esta sentencia crea un objeto Date (Date es una clase del paquete java,util). Esta sentencia realmente realiza tres acciones: declaración, ejemplarización e inicialización.

Date hoy es una declaración de variable que sólo le dice al compilador que el nombre hoy se va a utilizar para referirse a un objeto cuyo tipo es Date, el operador new ejemplariza la clase Date (creando un nuevo objeto Date), y Date() inicializa el objeto.

Declarar un Objeto

Ya que la declaración de un objeto es una parte innecesaria de la creación de un objeto, las declaraciones aparecen frecuentemente en la misma línea que la creación del objeto. Como cualquier otra declaración de variable, las declaraciones de objetos pueden aparecer solitarias como esta.

```
Date hoy;
```

De la misma forma, declarar una variable para contener un objeto es exactamente igual que declarar una variable que va a contener un tipo primitivo.

```
tipo nombre
```

donde tipo es el tipo de dato del objeto y nombre es el nombre que va a utilizar el objeto. En Java, las clases e interfaces son como tipos de datos. Entonces tipo puede ser el nombre de una clase o de un interface.

Las declaraciones notifican al compilador que se va a utilizar nombre para referirse a una variable cuyo tipo es tipo. Las declaraciones no crean nuevos objetos. Date hoy no crea un objeto Date, sólo crea un nombre de variable para contener un objeto Date. Para ejemplarizar la clase Date, o cualquier otra clase, se utiliza el operador new.

Ejemplarizar una Clase

El operador new ejemplariza una clase mediante la asignación de memoria para el objeto nuevo de ese tipo. new necesita un sólo argumento: una llamada al método constructor. Los métodos constructores son métodos especiales proporcionados por cada clase Java que son reponsables de la inicialización de los nuevos objetos de ese tipo. El operador new crea el objeto, el constructor lo inicializa.

Aquí tienes un ejemplo del uso del operador new para crear un objeto Rectangle (Rectangle es una clase del paquete java.awt).

```
new Rectangle(0, 0, 100, 200);
```

En el ejemplo, `Rectangle(0, 0, 100, 200)` es una llamada al constructor de la clase Rectangle.

El operador `new` devuelve una referencia al objeto recien creado. Esta referencia puede ser asignada a una variable del tipo apropiado.

```
Rectangle rect = new Rectangle(0, 0, 100, 200);
```

(Recuerda que una clase esencialmente define un tipo de dato de referencia. Por eso, Rectangle puede utilizarse como un tipo de dato en los programas Java. El valor de cualquier variable cuyo tipo sea un tipo de referencia, es una referencia (un puntero) al valor real o conjunto de valores representado por la variable.

Inicializar un Objeto

Como mencioné anteriormente, las clases porporcionan métodos constructores para incializar los nuevos objetos de ese tipo. Una clase podría proporcionar múltiples constructores para realizar diferentes tipos de inicialización en los nuevos objetos.

Cuando veas la implementación de una clase, reconocerás los constructores porque tienen el mismo nombre que la clase y no tienen tipo de retorno. Recuerda la creación del objeto Date en el sección inicial. El constructor utilizado no tenía ningún argumento.

```
Date()
```

Un constructor que no tiene ningún argumento, como el mostrado arriba, es conocido como constructor por defecto. Al igual que Date, la mayoría de las clases tienen al menos un constructor, el constructor por defecto.

Si una clase tiene varios constructores, todos ellos tienen el mismo nombre pero se deben diferenciar en el número o el tipo de sus argumentos. Cada constructor inicializa el nuevo objeto de una forma diferente. Junto al constructor por defecto, la clase Date proporciona otro constructor que inicializa el nuevo objeto con un nuevo año, mes y día.

```
Date cumpleaños = new Date(1963, 8, 30);
```

El compilador puede diferenciar los constructores a través del tipo y del número de sus argumentos.

En esta página:

Usar Objetos Java

Una vez que se ha creado un objeto, probablemente querrás hacer algo con él. Supón, por ejemplo, que después de crear un nuevo rectángulo, quieres moverlo a una posición diferente (es decir, el rectángulo es un objeto en un programa de dibujo y el usuario quiere moverlo a otra posición de la página).

La clase Rectangle proporciona dos formas equivalentes de mover el rectángulo.

1. Manipular directamente las variables **x** e **y** del objeto.
2. Llamar el método **move()**.

La opción 2 se considera "más orientada a objetos" y más segura porque se manipulan las variables del objeto indirectamente a través de una capa protectora de métodos, en vez de manejarlas directamente. Manipular directamente las variables de un objeto se considera propenso a errores; se podría colocar el objeto en un estado de inconsistencia.

Sin embargo, una clase no podría (y no debería) hacer que sus variables estuvieran disponibles para la manipulación directa por otros objetos, si fuera posible que esas manipulaciones situaran el objeto en un estado de inconsistencia. Java proporciona un mecanismo mediante el que las clases pueden restringir o permitir el acceso a sus variables y métodos a otros objetos de otros tipos.

Esta sección explica la llamada a métodos y la manipulación de variables que se han hecho accesibles a otras clases. Para aprender más sobre el control de acceso a miembros puedes ir Controlar el Acceso a Miembros de una Clase.

Las variables x e y de Rectangle son accesibles desde otras clases. Por eso podemos asumir que la manipulación directa de estas variables es segura.

Referenciar Variables de un Objeto

Primero, enfoquemos cómo inspeccionar y modificar la posición del rectángulo mediante la manipulación directa de las variables x e y. La siguiente sección mostrará como mover el rectángulo llamando al método move().

Para acceder a las variables de un objeto, sólo se tiene que añadir el nombre de la variable al del objeto referenciado introduciendo un punto en el medio ('.').

`objetoReferenciado.variable`

Supón que tienes un rectángulo llamado rect en tu programa. puedes acceder a las variables x e y con rect.x y rect.y, respectivamente.

Ahora que ya tienes un nombre para las variables de rect, puedes utilizar ese nombre en sentencias y expresiones Java como si fueran nombres de variables "normales". Así, para mover el rectángulo a una nueva posición podrías escribir.

```
rect.x = 15;        // cambia la posición x
rect.y = 37;        // cambia la posición y
```

La clase Rectangle tiene otras dos variables--width y height--que son accesibles para objetos fuera de Rectangle. Se puede utilizar la misma notación con ellas: rect.width y rect.height. Entonces se puede calcular el área del rectángulo utilizando esta sentencia.

```
area = rect.height * rect.width;
```

Cuando se accede a una variable a través de un objeto, se está refiriendo a las variables de un objeto particular. Si cubo fuera también un rectángulo con una altura y anchura diferentes de rect, esta instrucción.

```
area = cubo.height * cubo.width;
```

calcula el área de un rectángulo llamado cubo y dará un resultado diferente que la instrucción anterior (que calculaba el área de un rectángulo llamado rect).

Observa que la primera parte del nombre de una variable de un objeto (el objetoReferenciado en objetoReferenciado.variable) debe ser una referencia a un objeto. Como se puede utilizar un nombre de variable aquí, también se puede utilizar en cualquier expresión que devuelva una referencia a un objeto. Recuerda que el operador new devuelve una referencia a un objeto. Por eso, se puede utilizar el valor devuelto por new para acceder a las variables del nuevo objeto.

```
height = new Rectangle().height;
```

Llamar a Métodos de un Objeto

Llamar a un método de un objeto es similar a obtener una variable del objeto. Para llamar a un método del objeto, simplemente se añade al nombre del objeto referenciado el nombre del método, separados por un punto ('.'), y se proporcionan los argumentos del método entre paréntesis. Si el método no necesita argumentos, se utilizan los paréntesis vacios.

```
objetoReferenciado.nombreMétodo(listaArgumentos);
   o
objetoReferenciado.nombreMétodo();
```

Veamos que significa esto en términos de movimiento del rectángulo. Para mover rect a una nueva posición utilizando el método move() escribe esto.

```
rect.move(15, 37);
```

Esta sentencia Java llama al método move() de rect con dos parámetros enteros, 15 y 37. Esta sentencia tiene el efecto de mover el objeto rect igual que se hizo en las sentencias anteriores en las que se moficaban directamente los valores x e y del objeto.

```
rect.x = 15;
rect.y = 37;
```

Si se quiere mover un rectángulo diferente, uno llamado cubo, la nueva posición se podría escribir.

```
cubo.move(244, 47);
```

Como se ha visto en estos ejemplos, las llamadas a métodos se hacen directamente a un objeto específico; el objeto especificado en la llamada al método es el que responde a la instrucción.

Las llamadas a métodos también se conocen como mensajes.

Como en la vida real, los mensajes se deben dirigir a un receptor particular.

Se pueden obtener distintos resultados dependiendo del receptor de su mensaje.

En el ejemplo anterior, se ha enviado el mensaje move() al objeto llamado rect para que éste mueva su posición.

Cuando se envía el mensaje move() al objeto llamado cubo, el que se mueve es cubo. Son resultados muy distintos.

Una llamada a un método es una expresión (puedes ver Expresiones para más información) y evalúa a algún valor. El valor de una llamada a un método es su valor de retorno, si tiene alguno.

Normalmente se asignará el valor de retorno de un método a una variable o se utilizará la llamada al método dentro del ámbito de otra expresión o sentencia.

El método move() no devuelve ningún valor (está declarado como void). Sin embargo, el método inside() de Rectangle si lo hace. Este método toma dos coordendas x e y, y devuelte true si este punto está dentro del rectángulo.

Se puede utilizar el método inside() para hacer algo especial en algún punto, como decir la

posición del ratón cuando está dentro del rectangulo.

```
if (rect.inside(mouse.x, mouse.y)) {
    . . .
        // ratón dentro del rectángulo
    . . .
} else {
    . . .
        // ratón fuera del rectángulo
    . . .
}
```

Recuerda que una llamada a un método es un mensaje al objeto nombrado. En este caso, el objeto nombrado es rect. Entonces.

```
rect.inside(mouse.x, mouse.y)
```

le pregunta a rect si la posición del cursor del ratón se encuentra entre las coordenadas mouse.x y mouse.y. Se podría obtener una respuesta diferente si envía el mismo mensaje a cubo.

Como se explicó anteriormente, el objetoReferenciado en la llamada al método objetoReferenciado.metodo() debe ser una referencia a un objeto. Como se puede utilizar un nombre de variable aquí, también se puede utilizar en cualquier expresión que devuelva una referencia a un objeto. Recuerda que el operador new devuelve una referencia a un objeto. Por eso, se puede utilizar el valor devuelto por new para acceder a las variables del nuevo objeto.

```
new Rectangle(0, 0, 100, 50).equals(anotherRect)
```

La expresión new Rectangle(0, 0, 100, 50) evalúa a una referencia a un objeto que se refiere a un objeto Rectangle.

Entonces, como verás, se puede utilizar la notación de punto ('.') para llamar al método equals() del nuevo objeto Rectangle para determinar si el rectangúlo nuevo es igual al especificado en la lista de argumentos de equals().

En esta página:

Eliminar Objetos Java

Muchos otros lenguajes orientados a objetos necesitan que se siga la pista de los objetos que se han creado y luego se destruyan cuando no se necesiten. Escribir código para manejar la memoria de esta es forma es aburrido y propenso a errores.

Java permite ahorrarse esto, permitiéndo crear tantos objetos como se quiera (sólo limitados por los que el sistema pueda manejar) pero nunca tienen que ser destruidos. El entorno de ejecución Java borra los objetos cuando determina que no se van autilizar más. Este proceso es conocido como recolección de basura.

Un objeto es elegible para la recolección de basura cuando no existen más referencias a ese objeto. Las referencias que se mantienen en una variable desaparecen de forma natural cuando la variable sale de su ámbito. O cuando se borra explícitamente un objeto referencia mediante la selección de un valor cuyo tipo de dato es una referencia a null.

Recolector de Basura

El entorno de ejecución de Java tiene un recolector de basura que periódicamente libera la memoria ocupada por los objetos que no se van a necesitar más.

El recolector de basura de Java es un barredor de marcas que escanea dinámicamente la memoria de Java buscando objetos, marcando aquellos que han sido referenciados. Después de investigar todos los posibles paths de los objetos, los que no están marcados (esto es, no han sido referenciados) se les conoce como basura y son eliminados.

El colector de basura funciona en un thread (hilo) de baja prioridad y funciona tanto síncrona como asíncronamente dependiendo de la situación y del sistema en el que se esté ejecutando el entorno Java.

El recolector de basura se ejecuta síncronamente cuando el sistema funciona fuera de memoria o en respuesta a una petición de un programa Java. Un programa Java le puede pedir al recolector de basura que se ejecute en cualquier momento mediante una llamada a System.gc().

Nota: Pedir que se ejecute el recolector de basura no garantiza que los objetos sean recolectados.

En sistemas que permiten que el entorno de ejecución Java note cuando un thread a empezado a interrumpir a otro thread (como Windows 95/NT), el recolector de basura de Java funciona asíncromamente cuando el sistema está ocupado. Tan pronto como otro thread se vuelva activo, se pedira al recolector de basura que obtenga un estado consistente y termine.

Finalización

Antes de que un objeto sea recolectado, el recolector de basura le da una oportunidad para limpiarse él mismo mediante la llamada al método finalize() del propio objeto. Este proceso es conocido como finalización.

Durante la finalización de un objeto se podrían liberar los recursos del sistema como son los ficheros, etc. y liberar referencias en otros objetos para hacerse elegible por la recolección de basura.

El método finalize() es un miembro de la clase java.lang.Object. Una clase debe sobreescribir el método finalize() para realizar cualquier finalización necesaria para los objetos de ese tipo.

TutorJava Nivel Básico

En esta página:

Declarar Clases Java

Ahora que ya sabemos como crear, utilizar y destruir objetos, es hora de aprender cómo escribir clases de las que crear esos objetos.

Una clase es un proyecto o prototipo que se puede utilizar para crear muchos objetos. La implementación de una clase comprende dos componentes: la declaración y el cuerpo de la clase.

```
DeclaraciónDeLaClase {
    CuerpoDeLaClase
}
```

La Declaración de la Clase

Como mínimo, la declaración de una clase debe contener la palabra clave class y el nombre de la clase que está definiendo. Así la declaración más sencilla de una clase se parecería a esto.

```
class NombredeClase {
    . . .
}
```

Por ejemplo, esta clase declara una nueva clase llamada NumeroImaginario.

```
class NumeroImaginario {
    . . .
}
```

Los nombres de las clases deben ser un identificador legal de Java y, por convención, deben empezar por una letra mayúscula. Muchas veces, todo lo que se necesitará será una declaración mínima. Sin embargo, la declaración de una clase puede decir más cosas sobre la clase. Más especificamente, dentro de la declaración de la clase se puede.

- declarar cual es la superclase de la clase.
- listar los interfaces implementados por la clase
- declarar si la clase es pública, abstracta o final

Declarar la Superclase de la Clase

En Java, todas las clases tienen una superclase. Si no se especifica una superclase para una clase, se asume que es la clase Object (declarada en java.lang). Entonces la superclase de NumeroImaginario es Object porque la declaración no explicitó ninguna otra clase. Para obtener más información sobre la clase Object, puede ver La clase Object.

Para especificar explícitamente la superclase de una clase, se debe poner la palabra clave extends más el nombre de la superclase entre el nombre de la clase que se ha creado y la llave abierta que abre el cuerpo de la clase, así.

```
class NombredeClase extends NombredeSuperClase {
    . . .
}
```

Por ejemplo, supon que quieres que la superclase de NumeroImaginario sea la clase Number en vez de la clase Object. Se podría escribir esto.

```
class NumeroImaginario extends Number {
```

```
    . . .
}
```

Esto declara explícitamente que la clase Number es la superclase de NumeroImaginario. (La clase Number es parte del paquete java.lang y es la base para los enteros, los números en coma flotante y otros números).

Declarar que Number es la superclase de NumeroImaginario declara implícitamente que NumeroImaginario es una subclase de Number. Una subclase hereda las variables y los métodos de su superclase.

Crear una subclase puede ser tan sencillo como incluir la clausula extends en su declaración de clase. Sin embargo, se tendrán que hacer otras provisiones en su código cuando se crea una subclase, como sobreescribir métodos. Para obtener más información sobre la creación de subclases, puede ver Subclases, Superclases, y Herencia.

Listar los Interfaces Implementados por la Clase

Cuando se declara una clase, se puede especificar que interface, si lo hay, está implementado por la clase. Pero, ¿Qué es un interface? Un interface declara un conjunto de métodos y constantes sin especificar su implementación para ningún método. Cuando una clase exige la implementación de un interface, debe proporcionar la implementación para todos los métodos declarados en el interface.

Para declarar que una clase implementa uno o más interfaces, se debe utilizar la palabra clave implements seguida por una lista de los interfaces implementados por la clase delimitados por comas. Por ejemplo, imagina un interface llamado Aritmetico que define los métodos llamados suma(), resta(), etc... La clase NumeroImaginario puede declarar que implementa el interface Aritmetico de esta forma.

```
class NumeroImaginario extends Number implements Aritmetico {
    . . .
}
```

se debe garantizar que propociona la implementación para los métodos suma(), resta() y demás métodos declarados en el interface Aritmetico. Si en NumeroImaginario falta alguna implementación de los métodos definidos en Aritmetico, el compilador mostrará un mensaje de error y no compilará el programa.

```
nothing.java:5: class NumeroImaginario must be declared abstract. It does not define
java.lang.Number add(java.lang.Number, java.lang.Number) from interface Aritmetico.
class NumeroImaginario extends Number implements Aritmetico {
      ^
```

Por convención, la clausula implements sigue a la clausula extends si ésta existe.

Observa que las firmas de los métodos declarados en el interface Aritmetico deben corresponder con las firmas de los métodos implementados en la clase NumeroImaginario. Tienes más información sobre cómo crear y utilizar interfaces en Crear y Utilizar Interfaces.

Clases Public, Abstract, y Final

Se puede utilizar uno de estos tres modificadores en una declaración de clase para declarar que esa clase es pública, abstracta o final. Los modificadores van delante de la palabra clave class y son opcionales.

El modificador public declara que la clase puede ser utilizada por objetos que estén fuera del paquete actual. Por defecto, una clase sólo puede ser utilizada por otras clases del mismo paquete en el que están declaradas.

```
public class NumeroImaginario extends Number implements Aritmetico {
    . . .
}
```

Por convención, cuando se utiliza la palabra public en una declaración de clase debemos asegurarnos de que es el primer item de la declaración.

El modificador abstract declara que la clase es una clase abstracta. Una clase abstracta podría contener métodos abstractos (métodos sin implementación). Una clase abstracta está diseñada para ser una superclase y no puede ejemplarizarse. Para una discusión sobre las clases abstractas y cómo escribirlas puedes ver Escribir Clases y Métodos Abstractos.

Utilizando el modificador final se puede declarar que una clase es final, que no puede tener subclases. Existen (al menos) dos razones por las que se podría querer hacer esto: razones de seguridad y razones de diseño. Para una mejor explicación sobre las clases finales puedes ver Escribir Clases y Métodos Finales.

Observa que no tiene sentido para una clase ser abstracta y final. En otras palabras, una clase que contenga métodos no implementados no puede ser final. Intentar declarar una clase como final y abstracta resultará en un error en tiempo de compilación.

Sumario de la Daclaración de una Clase

En suma, una declaración de clase se parecería a esto.

```
[ modificadores ] class NombredeClase [ extends NombredeSuperclase ]
 [ implements NombredeInterface ] {
    . . .
}
```

Los puntos entre [y] son opcionales. Una declaración de clase define los siguientes aspectos de una clase.

- **modificadores** declaran si la clase es abstracta, pública o final.
- **NombredeClase** selecciona el nombre de la clase que está declarando
- **NombredeSuperClase** es el nombre de la superclase de NombredeClase
- **NombredeInterface** es una lista delimitada por comas de los interfaces implementados por NombredeClase

De todos estos items, sólo la palabra clave class y el nombre de la clase son necesarios. Los otros son opcionales. Si no se realiza ninguna declaración explícita para los items opcionales, el compilador Java asume ciertos valores por defecto (una subclase de Object no final, no pública, no obstracta y que no implementa interfaces).

En esta página:

El Cuerpo de una Clase Java

Anteriormente se vió una descripción general de la implementación de una clase.

```
DeclaraciondeClase {
    CuerpodeClase
}
```

La página anterior describe todos los componentes de la declaración de una clase. Esta página describe la estructura general y la organización del cuerpo de la clase.

El cuerpo de la clase compone la implementación de la propia clase y contiene dos secciones diferentes: la declaración de varibles y la de métodos. Una variable miembro de la clase representa un estado de la clase y sus métodos implemetan el comportamiendo de la clase. Dentro del cuerpo de la clase se definen todas las variables miembro y los métodos soportados por la clase.

Típicamente, primero se declaran las variables miembro de la clase y luego se porporcionan las declaraciones e implementanciones de los métodos, aunque este orden no es necesario.

```
DeclaracióndeClase {
    DeclaracionesdeVariablesMiembros
    DeclaracionesdeMétodos
}
```

Aquí tienes una pequeña clase que declara tres variables miembro y un método.

```
class Ticket {
    Float precio;
    String destino;
    Date fechaSalida;
    void firma(Float forPrecio, String forDestino, Date forFecha) {
        precio = forPrecio;
        destino = forDestino;
        fechaSalida = forFecha;
    }
}
```

Para más información sobre cómo declarar variables miembro, puedes ver Declarar Variables Miembro. Y para obtener más información sobre cómo implementar métodos, puedes ver Implementar Métodos.

Además de las variables miembro y los métodos que se declaran explícitamente dentro del cuerpo de la clase, una clase puede heredar algo de su superclase. Por ejemplo, todas las clases del entorno Java son una descendencia (directa o indirecta) de la clase Object. La clase Object define el estado básico y el comportamieto que todos los objetos deben tener como habilidad para comparar unos objetos con otros, para convertir una cadena, para esperar una condición variable, para notificar a otros objetos que una condición variable ha cambiado, etc... Así, como descendentes de esta clase, todos los objetos del entorno Java heredan sus comportamientos de la clase Object.

En esta página:

Declarar Variables Java

Como mínimo, una declaración de variable miembro tiene dos componentes: el tipo de dato y el nombre de la variable.

```
tipo nombreVariable;        // Declaración mínima de una variable miembro
```

Una declaración mínima de variable miembro es como la declaración de variables que se escribe en cualquier otro lugar de un programa Java, como las variables locales o los parámetros de los métodos. El siguiente código declara una variable miembro entera llamada unEntero dentro de la clase ClaseEnteros.

```
class ClaseEnteros {
    int unEntero;
    . . .
    // define los métodos aquí
    . . .
}
```

Observa que la declaración de variables miembro aparece dentro de la implementación del cuerpo de la clase pero no dentro de un método. Este posicionamiento dentro del cuerpo de la clase determina que una variable es una variable miembro.

Al igual que otras variables en Java, las variables miembro deben tener un tipo. Un tipo de variable determina los valores que pueden ser asignados a las variables y las operaciones que se pueden realizar con ellas. Ya deberías estar familiarizado con los tipos de datos en Java mediante la lectura de la lección anterior: Variables y Tipos de Datos.

Un nombre de una variable miembro puede ser cualquier identificador legal de Java y por convención empieza con una letra minúscula (los nombres de clase típicamente empiezan con una letra mayúscula). No se puede declarar más de una variable con el mismo nombre en la misma clase. Por ejemplo, el siguiente código es legal.

```
class ClaseEnteros {
    int unEntero;
    int unEntero() {        // un método con el mismo nombre que una variable
        . . .
    }
}
```

Junto con el nombre y el tipo, se pueden especificar varios atributos para las variables miembro cuando se las declara: incluyendo si los objetos pueden acceder a la variable, si la variable es una variable de clase o una variable de ejemplar, y si la variable es una constante.

Una declaración de variable se podría parecer a esto.

```
[especificadordeAcceso] [static] [final] [transient] [volatile] tipo nombredeVaraible
```

Los puntos entre [y] son opcionales. Los items en negrita se deben reemplazar por palabras clave o por nombres.

Una declaración de variable miembro define los siguientes aspectos de la variable.

- **especificadordeAcceso** define si otras clases tienen acceso a la variable. Se puede controlar el acceso a los métodos utilizando los mismos especificadores, por eso Controlar el Acceso a Variables Miembro de una Clase cubre cómo se puede controlar el acceso a las variables miembro o los métodos.
- **static** indica que la variable es una variable miembro de la clase en oposición a una variable miembro del ejemplar. Se puede utilizar **static** para declarar métodos de clase. Miembros de Clase y de Ejemplar explica la declaración de variables de clase y de ejemplar y escribir métodos de ejemplar o de clase.
- **final** indica que la variable es una constante

- **transient** la variable no es una parte persistente del estado del objeto
- **volatile** significa que la variable es modificada de forma asíncrona.

La explicación de las variables final, transient, y volatile viene ahora.

Declarar Constantes

Para crear una variable miembro constante en Java se debe utilizar la palabra clave final en su declaración de variable. La siguiente declaración define una constante llamada AVOGADRO cuyo valor es el número de Avogadro (6.023 x 10^23) y no puede ser cambiado.

```
class Avo {
    final double AVOGADRO = 6.023e23;
}
```

Por convención, los nombres de los valores constantes se escriben completamene en mayúsculas. Si un programa intenta cambiar una variable, el compilador muestra un mensaje de error similar al siguiente, y rehusa a compilar su programa.

```
AvogadroTest.java:5: Can't assign a value to a final variable: AVOGADRO
```

Declarar Variables Transitorias

Por defecto, las variables miembro son una parte persistente del estado de un objeto, Las variables que forman parte persistente del estado del objeto deben guardarse cuando el objeto se archiva. Se puede utilizar la palabra clave transient para indicar a la máquina virtual Java que la variable indicada no es una parte persistente del objeto.

Al igual que otros modificadors de variables en el sistema Java, se puede utilizar transient en una clase o declaración de variable de ejemplar como esta.

```
class TransientExample {
    transient int hobo;
    . . .
}
```

Este ejemplo declara una variable entera llamada hobo que no es una parte persistente del estado de la claseTransientExample.

Declarar Variables Volatiles

Si una clase contiene una variable miembro que es modificada de forma asíncrona, mediante la ejecución de threads concurrentes, se puede utilizar la palabra clave volatile de Java para notificar esto al sistema Java.

La siguiente declaración de variable es un ejemplo de como declarar que una variable va a ser modificada de forma asíncrona por threads concurrentes.

```
class VolatileExample {
    volatile int contador;
    . . .
}
```

En esta página:

Implementar Métodos Java

Similarmente a la implementación de una clase, la implementación de un método consiste en dos partes, la declaración y el cuerpo del método.

```
declaracióndeMétodo {
        cuerpodeMétodo
}
```

La Declaración de Método

Una declaración de método proporciona mucha información sobre el método al compilador, al sistema en tiempo de ejecución y a otras clases y objetos.

Junto con el nombre del método, la declaración lleva información como el tipo de retorno del método, el número y el tipo de los argumentos necesarios, y qué otras clases y objetos pueden llamar al método.

Los únicos elementos necesarios para una declaración de método son el nombre y el tipo de retorno del método. Por ejemplo, el código siguiente declara un método llamado estaVacio() en la clase Pila que devuelve un valor booleano (true o false).

```
class Pila {
    . . .
    boolean estaVacio() {
        . . .
    }
}
```

Devolver un Valor desde un Método

Java necesita que un método declare el tipo de dato del valor que devuelve. Si un método no devuelve ningún valor, debe ser declarado para devolver void (nulo).

Los métodos pueden devolver tipos de datos primitivos o tipos de datos de referencia. El método estaVacio() de la clase Pila devuelve un tipo de dato primitivo, un valor booleano.

```
class Pila {
    static final int PILA_VACIA = -1;
    Object[] stackelements;
    int topelement = PILA_VACIA;
    . . .
    boolean estaVacio() {
        if (topelement == PILA_VACIA)
            return true;
        else
            return false;
    }
}
```

Sin embargo, el método pop de la clase PILA devuelve un tipo de dato de referencia: un objeto.

```
class Pila {
    static final int PILA_VACIA = -1;
    Object[] stackelements;
    int topelement = PILA_VACIA;
    . . .
    Object pop() {
        if (topelement == PILA_VACIA)
            return null;
        else {
            return stackelements[topelement--];
        }
    }
}
```

Los métodos utilizan el operador return para devolver un valor. Todo método que no sea declarado como void debe contener una sentencia return.

El tipo de dato del valor devuelto por la sentencia return debe corresponder con el tipo de dato que el método tiene que devolver; no se puede devolver un objeto desde un método que fue declarado para devolver un entero.

Cuando se devuelva un objeto, el tipo de dato del objeto devuelto debe ser una subclase o la clase exacta indicada. Cuando se devuelva un tipo interface, el objeto retornado debe implementar el interface especificado.

Un Nombre de Método

Un nombre de método puede ser cualquier indentificador legal en Java. Existen tres casos especiales a tener en cuenta con los nombres de métodos.

1. Java soporta la sobrecarga de métodos, por eso varios métodos pueden compartir el mismo nombre. Por ejemplo, supon que se ha escrito una clase que puede proporcionar varios tipos de datos (cadenas, enteros, etc...) en un área de dibujo. Se podría escribir un método que supiera como tratar a cada tipo de dato. En otros lenguajes, se tendría que pensar un nombre distinto para cada uno de los métodos. **dibujaCadena()**, **dibujaEntero**, etc... En Java, se puede utilizar el mismo nombre para todos los métodos pasándole un tipo de parámetro diferente a cada uno de los métodos. Entonces en la clase de dibujo, se podrán declarar tres métodos llamados **draw()** y que cada uno aceptara un tipo de parámetro diferente.

   ```
   class DibujodeDatos {
       void draw(String s) {
           . . .
       }
       void draw(int i) {
           . . .
       }
       void draw(float f) {
           . . .
       }
   }
   ```

   Nota: La información que hay dentro de los paréntesis de la declaración son los argumentos del método. Los argumentos se cubren en la siguiente página: Pasar Información a un Método.

   Los métodos son diferenciados por el compilador basándose en el número y tipo de sus argumentos. Así draw(String s)y draw(int i) son métodos distintos y únicos. No se puede declarar un método con la misma firma: draw(String s)y draw(String t) son idénticos y darán un error del compilador.

   Habrás observado que los métodos sobrecargados deben devolver el mismo tipo de dato, por eso void draw(String s) e int draw(String t) declarados en la misma clase producirán un error en tiempo de compilación.
2. Todo método cuyo nombre sea igual que el de su clase es un **constructor** y tiene una tarea especial que realizar. Los constructores se utilizan para inicializar un objeto nuevo del tipo de la clase. Los constructores sólo pueden ser llamados con el operador **new**. Para aprender cómo escribir un constructor, puedes ver Escribir un Método Constructor.
3. Una clase puede sobreescribir un método de sus superclases. El método que sobreescribe debe tener el mismo, nombre, tipo de retorno y lista de parámetros que el método al que ha sobreescrito. Sobreescribir Métodos te enseñará como sobreescribir los métodos de una superclase.

Características Avanzadas de la Declaración de Métodos

Junto con los dos elementos necesarios, una declaración de método puede contener otros elementos. Estos elementos declaran los argumentos aceptados por el método, si el método es un método de clase, etc...

Juntándolo todo, una declaración de método se parecería a esto.

[**especificadordeAcceso**] [static] [abstract] [final] [native] [synchronized]
tipodeRetorno nombredelMétodo ([**listadeparámetros**]) [throws **listadeExcepciones**]

Cada uno de estos elementos de una declaración se cubre en alguna parte de este tutorial.

En esta página:

Pasar Información a un Método

Cuando se escribe un método, se declara el número y tipo de los argumentos requeridos por ese método. Esto se hace en la firma del método. Por ejemplo, el siguiente método calcula el pago mensual de una hipoteca basándose en la cantidad prestada, el interés, la duración de la hipoteca (número de meses) y el valor futuro de la hipoteca (presumiblemente el valor futuro sea cero, porque al final de la hipoteca, ya la habrás pagado).

```
double hipoteca(double cantidad, double interes, double valorFinal, int numPeriodos)
{
    double I, parcial1, denominador, respuesta;

    I = interes / 100.0;
    parcial1 = Math.pow((1 + I), (0.0 - numPeriodos));
    denominador = (1 - parcial1) / I;
    respuestar = ((-1 * cantidad) / denominador) - ((valorFinal * parcial1) /
denominador);
    return respuesta;
}
```

Este método toma cuatro argumentos: la cantidad prestada, el interés, el valor futuro y el número de meses. Los tres primeros son números de coma flotante de doble precisión y el cuarto es un entero.

Al igual que este método, el conjunto de argumentos de cualquier método es una lista de declaraciones de varibales delimitadas por comas donde cada declaración de variable es un par tipo/nombre.

`tipo nombre`

Como has podido ver en el ejemplo anterior, sólo tienes que utilizar el nombre del argumento para referirte al valor del argumento.

Tipos de Argumentos

En Java, se puede pasar como argumento a un método cualquier tipo de dato válido en Java. Esto incluye tipos primitivos, como enteros, dobles, etc.. y tipos de referencia como arrays, objetos, etc...

Aquí tienes un ejemplo de un constructor que acepta un array como argumento. En este ejemplo el constructor inicializa un objeto Polygon a partir de una lista de puntos (Point es una clase del paquete java.awt que representa una coordenada xy).

```
Polygon polygonFrom(Point[] listadePuntos) {
    . . .
}
```

Al contrario que en otros lenguajes, no se puede pasar un método a un método Java. Pero si se podría pasar un objeto a un método y luego llamar a los métodos del objeto.

Nombres de Argumentos

Cuando se declara un argumento para un método Java, se proporciona el nombre para ese argumento. Este nombre es utilizado dento del cuerpo del método para referirse al valor del argumento.

Un argumento de un método puede tener el mismo nombre que una variable de la clase. En este caso, se dice que el argumento oculta a la variable miembro. Normalmente los argumentos que ocultan una variable miembro se utilizan en los constructores para inicializar una clase. Por ejemplo, observa la clase Circle y su constructor.

```
class Circle {
    int x, y, radius;
```

```
    public Circle(int x, int y, int radius) {
        . . .
    }
}
```

La clase Circle tiene tres variables miembro x, y y radius. Además, el constructor de la clase Circle acepta tres argumentos cada uno de los cuales comparte el nombre con la variable miembro para la que el argumento proporciona un valor inicial.

Los nombres de argumentos ocultan los nombres de las variables miembro. Por eso utilizar x, y o radius dentro del cuerpo de la función, se refiere a los argumentos, no a las variables miembro. Para acceder a las variables miembro, se las debe referenciar a través de this--el objeto actual.

```
class Circle {
    int x, y, radius;
    public Circle(int x, int y, int radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
}
```

Los nombres de los argumentos de un método no pueden ser el mismo que el de otros argumentos del mismo método, el nombre de cualquier variable local del método o el nombre de cualquier parámetro a una clausula catch() dentro del mismo método.

Paso por Valor

En los métodos Java, los argumentos son pasados por valor. Cuando se le llama, el método recibe el valor de la variable pasada. Cuando el argumento es de un tipo primitivo, pasar por valor significa que el método no puede cambiar el valor. Cuando el argumento es del tipo de referencia, pasar por valor significa que el método no puede cambiar el objeto referenciado, pero si puede invocar a los métodos del objeto y puede modificar las variables accesibles dentro del objeto.

Consideremos esta serie de sentencias Java que intentan recuperar el color actual de un objeto Pen en una aplicación gráfica.

```
. . .
int r = -1, g = -1, b = -1;
pen.getRGBColor(r, g, b);
System.out.println("red = " + r + ", green = " + g + ", blue = " + b);
. . .
```

En el momento que se llama al método getRGBColor(), las variables r, g, y b tienen un valor de -1. El llamador espera que el método getRGBColor() le devuelva los valores de rojo, verde y azul para el color actual en las variables r, g, y b.

Sin embargo, el sistema Java pasa los valores de las variables(-1) al método getRGBColor(); no una referencia a las variables r, g, y b.

Con esto se podría visualizar la llamada a getRGBColor() de esta forma.

```
getRGBColor(-1, -1, -1)
```

Cuando el control pasa dentro del método getRGBColor(), los argumentos entran dentro del ámbito (se les asigna espacio) y son inicializados a los valores pasados al método.

```
class Pen {
    int valorRojo, valorVerde, valorAzul;
    void getRGBColor(int rojo, int verde, int azul) {
        // rojo, verde y azul han sido creados y sus valores son  -1
        . . .
    }
}
```

Con esto getRGBColor() obtiene acceso a los valores de r, g, y b del llamador a tavés de sus argumentos rojo, verde, y azul, respectivamente.

El método obtiene su propia copia de los valores para utilizarlos dentro del ámbito del método. Cualquier cambio realizado en estas copias locales no sera reflejado en las variables originales del llamador.

Ahora veremos la implementación de getRGBColor() dentro de la clase Pen que implicaba la firma de método anterior.

```
class Pen {
    int valorRojo, valorVerde, valorAzul;
    . . .
        // Este método no trabaja como se espera
    void getRGBColor(int rojo, int verde, int azul) {
```

```
        rojo = valorRojo;
        verde=valorVerde;
        azul=valorAzul;
    }
}
```

Este método no trabajará como se espera. Cuando el control llega a la sentencia println() en el siguiente fragmento de código, los argumentos rojo, verde y azul de getRGBColor() ya no existen. Por lo tanto las asignaciones realizadas dentro del método no tendrán efecto; r, g, y b seguiran siendo igual a -1.

```
. . .
int r = -1, g = -1, b = -1;
pen.getRGBColor(r, g, b);
System.out.println("rojo = " + r + ", verde = " + g + ", azul = " + b);
. . .
```

El paso de las varibales por valor le ofrece alguna seguridad a los programadores: los métodos no puede modificar de forma no intencionada una variable que está fuera de su ámbito.

Sin embargo, alguna vez se querrá que un método modifique alguno de sus argumentos. El metodo getRGBColor() es un caso apropiado. El llamador quiere que el método devuelva tres valores a través de sus argumentos. Sin embargo, el método no puede modificar sus argumentos, y, además, un método sólo puede devolver un valor a través de su valor de retorno. Entonces, ¿cómo puede un método devolver más de un valor, o tener algún efecto (modificar algún valor) fuera de su ámbito?

Para que un método modifique un argumento, debe ser un tipo de referencia como un objeto o un array. Los objetos y arrays también son pasados por valor, pero el valor de un objeto es una referencia. Entonces el efecto es que los argumentos de tipos de referencia son pasados por referencia. De aquí el nombre. Una referencia a un objeto es la dirección del objeto en la memoria. Ahora, el argumento en el método se refiere a la misma posición de memoria que el llamador.

Reescribamos el método getRGBColor() para que haga lo que queremos. Primero introduzcamos un nuevo objeto RGBColor, que puede contener los valores de rojo, verde y azul de un color en formato RGB.

```
class RGBColor {
    public int rojo, verde, azul;;
}
```

Ahora podemos reescribir getRGBColor() para que acepte un objeto RGBColor como argumento. El método getRGBColor() devuelve el color actual de lápiz, en los valores de las variables miembro rojo, verde y azul de su argumento RGBColor.

```
class Pen {
    int valorRojo, valorVerde, valorAzul;
    void getRGBColor(RGBColor unColor) {
        unColor.rojo = valorRojo;
        unColor.verde = valorVerde;
        unColor.azul = valorAzul;
    }
}
```

Y finalmente, reescribimos la secuencia de llamada.

```
. . .
RGBColor penColor = new RGBColor();
pen.getRGBColor(penColor);
System.out.println("ojo = " + penColor.rojo + ", verde = " + penColor.verde + ",
                   azul = " + penColor.azul);
. . .
```

Las modificaciones realizadas al objeto RGBColor dentro del método getRGBColor() afectan al objeto creado en la secuencia de llamada porque los nombres penColor (en la secuencia de llamada) y unColor (en el método getRGBColor()) se refieren al mismo objeto.

En esta página:

El Cuerpo de un Método

En el siguiente ejemplo, el cuerpo de método para los métodos estaVacio() y poner() están en negrita.

```
class Stack {
    static final int PILA_VACIA = -1;
    Object[] elementosPila;
    int elementoSuperior = PILA_VACIA;
    . . .
    boolean estaVacio() {
        if (elementoSuperior == PILA_VACIA)
            return true;
        else
            return false;
    }
    Object poner() {
        if (elementoSuperior == PILA_VACIA)
            return null;
        else {
            return elementosPila[elementoSuperior--];
        }
    }
}
```

Junto a los elementos normales del lenguaje Java, se puede utilizar this en el cuerpo del método para referirse a los miembros del objeto actual.

El objeto actual es el objeto del que uno de cuyos miembros está siendo llamado. También se puede utilizar super para referirse a los miembros de la superclase que el objeto actual haya ocultado mediante la sobreescritura. Un cuerpo de método también puede contener declaraciones de variables que son locales de ese método.

this

Normalmente, dentro del cuerpo de un método de un objeto se puede referir directamente a las variables miembros del objeto. Sin embargo, algunas veces no se querrá tener ambigüedad sobre el nombre de la variable miembro y uno de los argumentos del método que tengan el mismo nombre.

Por ejemplo, el siguiente constructor de la clase HSBColor inicializa alguna variable miembro de un objeto de acuerdo a los argumentos pasados al constructor. Cada argumento del constructor tiene el mismo nombre que la variable del objeto cuyo valor contiene el argumento.

```
class HSBColor {
    int hue, saturacion, brillo;
    HSBColor (int luminosidad, int saturacion, int brillo) {
        this.luminosidad = luminosidad;
```

```
        this.saturacion = saturacion;
        this.brillo = brillo;
    }
}
```

Se debe utilizar this en este constructor para evitar la ambigüedad entre el argumento luminosidad y la variable miembro luminosidad (y así con el resto de los argumentos). Escribir luminosidad = luminosidad; no tendría sentido. Los nombres de argumentos tienen mayor precedencia y ocultan a los nombres de las variables miembro con el mismo nombre. Para referirise a la variable miembro se debe hacer explicitamente a través del objeto actual--this.

También se puede utilizar this para llamar a uno de los métodos del objeto actual. Esto sólo es necesario si existe alguna ambigüedad con el nombre del método y se utiliza para intentar hacer el código más claro.

super

Si el método oculta una de las variables miembro de la superclase, se puede referir a la variable oculta utilizando super. De igual forma, si el método sobreescribe uno de los métodos de la superclase, se puede llamar al método sobreescrito a través de super.

Consideremos esta clase.

```
class MiClase {
    boolean unaVariable;
    void unMetodo() {
        unaVariable = true;
    }
}
```

y una subclase que oculta unaVariable y sobreescribe unMetodo().

```
class OtraClase extends MiClase {
    boolean unaVariable;
    void unMetodo() {
        unaVariable = false;
        super.unMetodo();
        System.out.println(unaVariable);
        System.out.println(super.unaVariable);
    }
}
```

Primero unMetodo() selecciona unaVariable (una declarada en OtraClase que oculta a la declarada en MiClase) a false. Luego unMetodo() llama a su método sobreescrito con esta sentencia.

```
super.unMetodo();
```

Esto selecciona la versión oculta de unaVariable (la declarada en MiClase) a true.

Luego unMetodo muestra las dos versiones de unaVariable con diferentes valores.

```
false
true
```

Variables Locales

Dentro del cuerpo de un método se puede declarar más variables para usarlas dentro del método. Estas variables son variables locales y viven sólo mientras el control permanezca dentro del método. Este método declara un variable local i y la utiliza para operar sobre los elementos del array.

```
Object encontrarObjetoEnArray(Object o, Object[] arrayDeObjetos) {
    int i;      //  variable local
    for (i = 0; i < arrayDeObjetos.length; i++) {
        if (arrayDeObjetos[i] == o)
            return o;
    }
    return null;
```

```
}
```

Después de que este método retorne, i ya no existirá más.

En esta página:

Miembros de la Clase y del Ejemplar

Cuando se declara una variable miembro como unFloat en MiClase.

```
class MiClase {
    float unFloat;
}
```

declara una variable de ejemplar. Cada vez que se crea un ejemplar de la clase, el sistema crea una copia de todas las variables de ejemplar de la clase. Se puede acceder a una variable del ejemplar del objeto desde un objeto como se describe en Utilizar Objetos.

Las variables de ejemplar están en constraste con las variables de clase (que se declaran utilizando el modificador static). El sistema asigna espacio para las variables de clase una vez por clase, sin importar el número de ejemplares creados de la clase. Todos los objetos creados de esta clase comparten la misma copia de las variables de clase de la clase, se puede acceder a las variables de clase a través de un ejemplar o a través de la propia clase.

Los métodos son similares: una clase puede tener métodos de ejemplar y métodos de clase. Los métodos de ejemplar operan sobre las variables de ejemplar del objeto actual pero también pueden acceder a las variables de clase. Por otro lado, los métodos de clase no pueden acceder a las variables del ejemplar declarados dentro de la clase (a menos que se cree un objeto nuevo y acceda a ellos a través del objeto). Los métodos de clase también pueden ser invocados desde la clase, no se necesita un ejemplar para llamar a los métodos de la clase.

Por defecto, a menos que se especifique de otra forma, un miembro declarado dentro de una clase es un miembro del ejemplar. La clase definida abajo tiene una variable de ejemplar -- un entero llamado x -- y dos métodos de ejemplar -- x() y setX() -- que permite que otros objetos pregunten por el valor de x.

```
class UnEnteroLlamadoX {
    int x;
    public int x() {
        return x;
    }
    public void setX(int newX) {
        x = newX;
    }
}
```

Cada vez que se ejemplariza un objeto nuevo desde una clase, se obtiene una copia de cada una de las variables de ejemplar de la clase. Estas copias están asociadas con el objeto nuevo. Por eso, cada vez que se ejemplariza un nuevo objeto UnEnteroLlamadoX de la clase, se obtiene una copia de x que está asociada con el nuevo objeto UnEnteroLlamadoX.

Todos los ejemplares de una clase comparten la misma implementación de un método de ejemplar; todos los ejemplares de UnEnteroLlamadoX comparten la misma implementación de x() y setX(). Observa que estos métodos se refieren a la variable de ejemplar del objeto x por su nombre. "Pero, ¿si todos los ejemplares de UnEnteroLlamadoX comparten la misma implementación de x() y setX() esto no es ambigüo?" La respuesta es no. Dentro de un método de ejemplar, el nombre de una variable de ejemplar se refiere a la variable de ejemplar del objeto actual (asumiendo que la variable de ejemplar no está ocultada por un parámetro del método). Ya que, dentro de x() y setX(), x es equivalente a this.x.

Los objetos externos a UnEnteroLlamadoX que deseen acceder a x deben hacerlo a través de un ejemplar particular de UnEnteroLlamadoX. Supongamos que este código estuviera en otro método del objeto. Crea dos objetos diferentes del tipo UnEnteroLlamadoX, y selecciona sus valores de x a diferente valores y luego lo muestra:

```
. . .
UnEnteroLlamadoX miX = new UnEnteroLlamadoX();
UnEnteroLlamadoX otroX = new UnEnteroLlamadoX();
miX.setX(1);
```

```
otroX.x = 2;
System.out.println("miX.x = " + miX.x());
System.out.println("otroX.x = " + otroX.x());
. . .
```

Observa que el código utilizado en setX() para seleccionar el valor de x para miX pero sólo asignando el valor otroX.x directamente. De otra forma, el código manipula dos copias diferentes de x: una contenida en el objeto miX y la otra en el objeto otroX. La salida producida por este código es.

```
miX.x = 1
otroX.x = 2
```

mostrando que cada ejemplar de la clase UnEnteroLlamadoX tiene su propia copia de la variable de ejemplar x y que cada x tiene un valor diferente.

Cuando se declara una variable miembro se puede especificar que la variable es una variable de clase en vez de una variable de ejemplar. Similarmente, se puede especificar que un método es un método de clase en vez de un método de ejemplar. El sistema crea una sola copia de una variable de clase la primera vez que encuentra la clase en la que está definida la variable. Todos los ejemplares de esta clase comparten la misma copia de las variables de clase. Los métodos de clase sólo pueden operar con variables de clase -- no pueden acceder a variables de ejemplar definidas en la clase.

Para especificar que una variable miembro es una variable de clase, se utiliza la palabra clave static. Por ejemplo, cambiemos la clase UnEnteroLlamadoX para que su variable x sea ahora una variable de clase.

```
class UnEnteroLlamadoX {
    static int x;
    public int x() {
        return x;
    }
    public void setX(int newX) {
        x = newX;
    }
}
```

Ahora veamos el mismo código mostrado anteriormente que crea dos ejemplares de UnEnteroLlamadoX, selecciona sus valores de x, y muestra esta salida diferente.

```
miX.x = 2
otroX.x = 2
```

La salida es diferente porque x ahora es una variable de clase por lo que sólo hay una copia de la variable y es compartida por todos los ejemplares de UnEnteroLlamadoX incluyendo miX y otroX.

Cuando se llama a setX() en cualquier ejemplar, cambia el valor de x para todos los ejemplares de UnEnteroLlamadoX.

Las variables de clase se utilizan para aquellos puntos en los que se necesite una sola copia que debe estar accesible para todos los objetos heredados por la clase en la que la variable fue declarada. Por ejemplo, las variables de clase se utilizan frecuentemente con final para definir constantes (esto es más eficiente en el consumo de memoria, ya que las constantes no pueden cambiar y sólo se necesita una copia).

Similarmente, cuando se declare un método, se puede especificar que el método es un método de clase en vez de un método de ejemplar. Los métodos de clase sólo pueden operar con variables de clase y no pueden acceder a las variables de ejemplar definidas en la clase.

Para especificar que un método es un método de clase, se utiliza la palabra clave static en la declaración de método. Cambiemos la clase UnEnteroLlamadoX para que su variable miembro x sea de nuevo una variable de ejemplar, y sus dos métodos sean ahora métodos de clase.

```
class UnEnteroLlamadoX {
    private int x;
    static public int x() {
        return x;
    }
    static public void setX(int newX) {
        x = newX;
    }
}
```

Cuando se intente compilar esta versión de UnEnteroLlamadoX, se obtendrán errores de compilación.

```
UnEnteroLlamadoX.java:4: Can't make a static reference to nonstatic variable x in
class UnEnteroLlamadoX.
        return x;
               ^
```

```
UnEnteroLlamadoX.java:7: Can't make a static reference to nonstatic variable x in
class UnEnteroLlamadoX.
        x = newX;
        ^
2 errors
```

Esto es porque los métodos de la clase no pueden acceder a variables de ejemplar a menos que el método haya creado un ejemplar de UnEnteroLlamadoX primero y luego acceda a la variable a través de él.

Construyamos de nuevo UnEnteroLlamadoX para hacer que su variable x sea una variable de clase.

```
class UnEnteroLlamadoX {
    static private int x;
    static public int x() {
        return x;
    }
    static public void setX(int newX) {
        x = newX;
    }
}
```

Ahora la clase se compilará y el código anterior que crea dos ejemplares de UnEnteroLlamadoX, selecciona sus valores x, y muestra en su salida los valores de x.

```
miX.x = 2
otroX.x = 2
```

De nuevo, cambiar x a través de miX también lo cambia para los otros ejemplares de UnEnteroLlamadoX.

Otra diferencia entre miembros del ejemplar y de la clase es que los miembros de la clase son accesibles desde la propia clase. No se necesita ejemplarizar la clase para acceder a los miembros de clase.

Reescribamos el código anterior para acceder a x() y setX() directamente desde la clase UnEnteroLlamadoX.

```
. . .
UnEnteroLlamadoX.setX(1);
System.out.println("UnEnteroLlamadoX.x = " + UnEnteroLlamadoX.x());
. . .
```

Observa que ya no se tendrá que crear miX u otroX. Se puede seleccionar x y recuperarlo directamente desde la clase UnEnteroLlamadoX. No se puede hacer esto con miembros del ejemplar. Solo se puede invocar métodos de ejemplar a través de un objeto y sólo puede acceder a las variables de ejemplar desde un objeto. Se puede acceder a las variables y métodos de clase desde un ejemplar de la clase o desde la clase misma.

En esta página:

Controlar el Acceso a los Miembros de la Clase

Uno de los beneficos de las clases es que pueden proteger sus variables y métodos miembros frente al acceso de otros objetos. ¿Por qué es esto importante? Bien, consideremos esto. Se ha escrito una clase que representa una petición a una base de datos que contiene toda clase de información secreta, es decir, registros de empleados o proyectos secretos de la compañia.

Ciertas informaciones y peticiones contenidas en la clase, las soportadas por los métodos y variables accesibles públicamente en su objeto son correctas para el consumo de cualquier otro objeto del sistema. Otras peticiones contenidas en la clase son sólo para el uso personal de la clase. Estas otras soportadas por la operación de la clase no deberían ser utilizadas por objetos de otros tipos. Se querría proteger esas variables y métodos personales a nivel del lenguaje y prohibir el acceso desde objetos de otros tipos.

En Java se pueden utilizar los especificadores de acceso para proteger tanto las variables como los métodos de la clase cuando se declaran. El lenguaje Java soporta cuatro niveles de acceso para las variables y métodos miembros: private, protected, public, y, todavía no especificado, acceso de paquete.

La siguiente tabla le muestra los niveles de acceso pemitidos por cada especificador.

Especificador	clase	subclase	paquete	mundo
private	X			
protected	X	X*	X	
public	X	X	X	X
package	X		X	

La primera columna indica si la propia clase tiene acceso al miembro definido por el especificador de acceso. La segunda columna indica si las subclases de la clase (sin importar dentro de que paquete se encuentren estas) tienen acceso a los miembros. La tercera columna indica si las clases del mismo paquete que la clase (sin importar su parentesco) tienen acceso a los miembros. La cuarta columna indica si todas las clases tienen acceso a los miembros.

Observa que la intersección entre protected y subclase tiene un '*' - este caso de acceso particular tiene una explicación en más detalle más adelante.

Echemos un vistazo a cada uno de los niveles de acceso más detalladamente.

Private

El nivel de acceso más restringido es private. Un miembro privado es accesible sólo para la clase en la que está definido. Se utiliza este acceso para declarar miembros que sólo deben ser utilizados por la clase. Esto incluye las variables que contienen información que si se accede a ella desde el exterior podría colocar al objeto en un estado de inconsistencia, o los métodos que llamados desde el exterior pueden poner en peligro el estado del objeto o del programa donde se está ejecutando. Los miembros privados son como secretos, nunca deben contarsele a nadie.

Para declarar un miembro privado se utiliza la palabra clave private en su declaración. La clase siguiente contiene una variable miembro y un método privados.

```
class Alpha {
    private int soyPrivado;
    private void metodoPrivado() {
        System.out.println("metodoPrivado");
    }
```

```
}
```

Los objetos del tipo Alpha pueden inspeccionar y modificar la variable soyPrivado y pueden invocar el método metodoPrivado(), pero los objetos de otros tipos no pueden acceder. Por ejemplo, la clase Beta definida aquí.

```
class Beta {
    void metodoAccesor() {
        Alpha a = new Alpha();
        a.soyPrivado = 10;      // ilegal
        a.metodoPrivado();      // ilegal
    }
}
```

no puede acceder a la variable soyPrivado ni al método metodoPrivado() de un objeto del tipo Alpha porque Beta no es del tipo Alpha.

Si una clase está intentando acceder a una variable miembro a la que no tiene acceso--el compilador mostrará un mensaje de error similar a este y no compilará su programa.

```
Beta.java:9: Variable iamprivate in class Alpha not accessible from class Beta.
        a.iamprivate = 10;     // ilegal
         ^
1 error
```

Y si un programa intenta acceder a un método al que no tiene acceso, generará un error de compilación parecido a este.

```
Beta.java:12: No method matching privateMethod() found in class Alpha.
        a.privateMethod();         // ilegal
1 error
```

Protected

El siguiente especificador de nivel de acceso es 'protected' que permite a la propia clase, las subclases (con la excepción a la que nos referimos anteriormente), y todas las clases dentro del mismo paquete que accedan a los miembros. Este nivel de acceso se utiliza cuando es apropiado para una subclase de la clase tener acceso a los miembros, pero no las clases no relacionadas. Los miembros protegidos son como secretos familiares - no importa que toda la familia lo sepa, incluso algunos amigos allegados pero no se quiere que los extraños lo sepan.

Para declarar un miembro protegido, se utiliza la palabra clave protected. Primero echemos un vistazo a cómo afecta este especificador de acceso a las clases del mismo paquete.

Consideremos esta versión de la clase Alpha que ahora se declara para estar incluida en el paquete Griego y que tiene una variable y un método que son miembros protegidos.

```
package Griego;

class Alpha {
    protected int estoyProtegido;
    protected void metodoProtegido() {
        System.out.println("metodoProtegido");
    }
}
```

Ahora, supongamos que la clase Gamma, también está declarada como miembro del paquete Griego (y no es una subclase de Alpha). La Clase Gamma puede acceder legalmente al miembro estoyProtegido del objeto Alpha y puede llamar legalmente a su método metodoProtegido().

```
package Griego;

class Gamma {
    void metodoAccesor() {
        Alpha a = new Alpha();
        a.estoyProtegido = 10;    // legal
        a.metodoProtegido();    // legal
    }
}
```

Esto es muy sencillo. Ahora, investiguemos cómo afecta el especificador protected a una subclase de Alpha.

Introduzcamos una nueva clase, Delta, que desciende de la clase Alpha pero reside en un paquete diferente - Latin. La clase Delta puede acceder tanto a estoyProtegido como a metodoProtegido(), pero solo en

objetos del tipo Delta o sus subclases. La clase Delta no puede acceder a estoyProtegido o metodoProtegido() en objetos del tipo Alpha. metodoAccesor() en el siguiente ejemplo intenta acceder a la variable miembro estoyProtegido de un objeto del tipo Alpha, que es ilegal, y en un objeto del tipo Delta que es legal.

Similarmente, metodoAccesor() intenta invocar a metodoProtegido() en un objeto del tipo Alpha, que también es ilegal.

```
import Griego.*;

package Latin;

class Delta extends Alpha {
    void metodoAccesor(Alpha a, Delta d) {
        a.estoyProtegido = 10;    // ilegal
        d.estoyProtegido = 10;    // legal
        a.metodoProtegido();    // ilegal
        d.metodoProtegido();    // legal
    }
}
```

Si una clase es una subclase o se cuentra en el mismo paquete de la clase con el miembro protegido, la clase tiene acceso al miembro protegido.

Public

El especificador de acceso más sencillo es 'public'. Todas las clases, en todos los paquetes tienen acceso a los miembros públicos de la clase. Los miembros públicos se declaran sólo si su acceso no produce resultados indeseados si un extraño los utiliza. Aquí no hay secretos familiares; no importa que lo sepa todo el mundo.

Para declarar un miembro público se utiliza la palabra clave public. Por ejemplo,

```
package Griego;

class Alpha {
    public int soyPublico;
    public void metodoPublico() {
        System.out.println("metodoPublico");
    }
}
```

Reescribamos nuestra clase Beta una vez más y la ponemos en un paquete diferente que la clase Alpha y nos aseguramos que no están relacionadas (no es una subclase) de Alpha.

```
import Griego.*;

package Romano;

class Beta {
    void metodoAccesor() {
        Alpha a = new Alpha();
        a.soyPublico = 10;       // legal
        a.metodoPublico();       // legal
    }
}
```

Como se puede ver en el ejemplo anterior, Beta puede inspeccionar y modificar legalmente la variable soyPublico en la clase Alpha y puede llamar legalmente al método metodoPublico().

Acceso de Paquete

Y finalmente, el último nivel de acceso es el que se obtiene si no se especifica ningún otro nivel de acceso a los miembros. Este nivel de acceso permite que las clases del mismo paquete que la clase tengan acceso a los miembros. Este nivel de acceso asume que las clases del mismo paquete son amigas de confianza. Este nivel de confianza es como la que extiende a sus mejores amigos y que incluso no la tiene con su familia.

Por ejemplo, esta versión de la clase Alpha declara una variable y un método con acceso de paquete. Alpha reside en el paquete Griego.

```
package Griego;

class Alpha {
```

```
    int estoyEmpaquetado;
    void metodoEmpaquetado() {
        System.out.println("metodoEmpaquetado");
    }
}
```

La clase Alpha tiene acceso a estoyEmpaquetado y a metodoEmpaquetado().

Además, todas las clases declaradas dentro del mismo paquete como Alpha también tienen acceso a estoyEmpaquetado y metodoEmpaquetado().

Supongamos que tanto Alpha como Beta son declaradas como parte del paquete Griego.

```
package Griego;

class Beta {
    void metodoAccesor() {
        Alpha a = new Alpha();
        a.estoyEmpaquetado = 10;      // legal
        a.metodoEmpaquetado();     // legal
    }
}
```

Entonces Beta puede acceder legalmente a estoyEmpaquetado y metodoEmpaquetado().

En esta página:

Constructores

Todas las clases Java tienen métodos especiales llamados Constructores que se utilizan para inicializar un objeto nuevo de ese tipo. Los contructores tienen el mismo nombre que la clase --el nombre del constructor de la clase Rectangle es Rectangle(), el nombre del constructor de la clase Thread es Thread(), etc...

Java soporta la sobrecarga de los nombres de métodos, por lo que una clase puede tener cualquier número de constructores, todos los cuales tienen el mismo nombre. Al igual que otros métodos sobrecargados, los constructores se diferencian unos de otros en el número y tipo de sus argumentos.

Consideremos la clase Rectangle del paquete java.awt que proporciona varios constructores diferentes, todos llamados Rectangle(), pero cada uno con número o tipo diferentes de argumentos a partir de los cuales se puede crear un nuevo objeto Rectangle. Aquí tiene las firmas de los constructores de la clase java.awt.Rectangle.

```
public Rectangle()
public Rectangle(int width, int height)
public Rectangle(int x, int y, int width, int height)
public Rectangle(Dimension size)
public Rectangle(Point location)
public Rectangle(Point location, Dimension size)
```

El primer constructor de Rectangle inicializa un nuevo Rectangle con algunos valores por defecto razonables, el segundo constructor inicializa el nuevo Rectangle con la altura y anchura especificadas, el tercer constructor inicializa el nuevo Rectangle en la posición especificada y con la altura y anchura especificadas, etc...

Típicamente, un constructor utiliza sus argumentos para inicializar el estado del nuevo objeto. Entonces, cuando se crea un objeto, se debe elegir el constructor cuyos argumentos reflejen mejor cómo se quiere inicializar el objeto.

Basándose en el número y tipos de los argumentos que se pasan al constructor, el compilador determina cual de ellos utilizar, Así el compilador sabe que cuando se escribe.

```
new Rectangle(0, 0, 100, 200);
```

el compilador utilizará el constructor que requiere cuatro argumentos enteros, y cuando se escribe.

```
new Rectangle(miObjetoPoint, miObjetoDimension);
```

utilizará el constructor que requiere como argumentos un objeto Point y un objeto Dimension.

Cuando escribas tus propias clases, no tienes porque proporcionar constructores. El constructor por defecto, el constructor que no necesita argumentos, lo proporciona automáticamente el sistema para todas las clases. Sin embargo, frecuentemente se querrá o necesitará proporcionar constructores para las clases.

Se puede declarar e implementar un constructor como se haría con cualquier otro método en una clase. El nombre del constructor debe ser el mismo que el nombre de la clase y, si se proporciona más de un constructor, los argumentos de cada uno de los constructores deben diferenciarse en el número o tipo. No se tiene que especificar el valor de retorno del constructor.

El constructor para esta subclase de Thread, un hilo que realiza animación, selecciona algunos

valores por defecto como la velocidad de cuadro, el número de imágenes y carga las propias imágenes.

```
class AnimationThread extends Thread {
    int framesPerSecond;
    int numImages;
    Image[] images;

    AnimationThread(int fps, int num) {
        int i;

        super("AnimationThread");
        this.framesPerSecond = fps;
        this.numImages = num;

        this.images = new Image[numImages];
        for (i = 0; i <= numImages; i++) {
            . . .
            // Carga las imágenes
            . . .
        }
    }
}
```

Observa cómo el cuerpo de un constructor es igual que el cuerpo de cualquier otro método -- contiene declaraciones de variables locales, bucles, y otras sentencias. Sin embargo, hay una línea en el constructor de AnimationThread que no se verá en un método normal--la segunda línea.

```
super("AnimationThread");
```

Esta línea invoca al constructor proporcionado por la superclase de AnimationThread--Thread. Este constructor particular de Thread acepta una cadena que contiene el nombre del Thread. Frecuentemente un constructor se aprovechará del código de inicialización escrito para la superclase de la clase.

En realidad, algunas clases deben llamar al constructor de su superclase para que el objeto trabaje de forma apropiada. Típicamente, llamar al constructor de la superclase es lo primero que se hace en el constructor de la subclase: un objeto debe realizar primero la inicialización de nivel superior.

Cuando se declaren constructores para las clases, se pueden utilizar los especificadores de acceso normales para especificar si otros objetos pueden crear ejemplares de su clase.

private

Niguna otra clase puede crear un objeto de su clase.

La clase puede contener métodos públicos y esos métodos pueden construir un objeto y devolverlo, pero nada más.

protected

Sólo las subclases de la clase pueden crear ejemplares de ella.

public

Cualquiera puede crear un ejemplar de la clase.

package-access

Nadie externo al paquete puede construir un ejemplar de su clase.

Esto es muy útil si se quiere que las clases que tenemos en un paquete puedan crear ejemplares de la clase pero no se quiere que lo haga nadie más.

En esta página:

Escribir un Método finalize()

Antes de que un objeto sea recolectado por el recolector de basura, el sistema llama al método finalize(). La intención de este método es liberar los recursos del sistema, como ficheros o conexiones abiertas antes de empezar la recolección.

Una clase puede proporcionar esta finalización simplemente definiendo e implementando un método llamado finalize(). El método finalize() debe declararse de la siguiente forma.

```
protected void finalize () throws throwable
```

Esta clase abre un fichero cuando se construye.

```
class AbrirUnFichero {
    FileInputStream unFichero = null;
    AbrirUnFichero (String nombreFichero) {
        try {
            unFichero = new FileInputStream(nombreFichero);
        } catch (java.io.FileNotFoundException e) {
        System.err.println("No se pudo abrir el fichero " + nombreFichero);
        }
    }
}
```

Para un buen comportamiento, la clase AbrirUnFichero debería cerrar el fichero cuando haya finalizado. Aquí tienes el método finalize() para la clase AbrirUnFichero.

```
protected void finalize () throws throwable {
    if (unFichero != null) {
        unFichero.close();
        unFichero = null;
    }
}
```

El método finalize() está declarado en la clase java.lang.Object. Así cuando escribas un método finalize() para tus clases estás sobreescribiendo el de su superclase. En Sobreescribir Métodos encontrarás más información sobre la sobreescritura de métodos.

Si la superclase tiene un método finalize(), probablemente este método deberá llamar al método finalize() de su superclase después de haber terminado sus tareas de limpieza. Esto limpiará cualquier recurso obtenido sin saberlo a través de los métodos heredados desde la superclase.

```
protected void finalize() throws Throwable {
    . . .
    // aquí va el código de limpieza de esta clase
    . . .
    super.finalize();
}
```

En esta página:

Subclases, Superclases y Herencia

En Java, como en otros lenguajes de programación orientados a objetos, las clases pueden derivar desde otras clases. La clase derivada (la clase que proviene de otra clase) se llama subclase. La clase de la que está derivada se denomina superclase.

De hecho, en Java, todas las clases deben derivar de alguna clase. Lo que nos lleva a la cuestión ¿Dónde empieza todo esto?. La clase más alta, la clase de la que todas las demás descienden, es la clase Object, definida en java.lang. Object es la raíz de la herencia de todas las clases.

Las subclases heredan el estado y el comportamiento en forma de las variables y los métodos de su superclase. La subclase puede utilizar los ítems heredados de su superclase tal y como son, o puede modificarlos o sobreescribirlos. Por eso, según se va bajando por el árbol de la herencia, las clases se convierten en más y más especializadas.

Definición:

Una subclase es una clase que desciende de otra clase. Una subclase hereda el estado y el comportamiento de todos sus ancestros. El término superclase se refiere a la clase que es el ancestro más directo, así como a todas las clases ascendentes.

TutorJava Nivel Básico

En esta página:

Crear Subclases

Se declara que un clase es una subclase de otra clase dentro de La declaración de Clase. Por ejemplo, supongamos que queremos crear una subclase llamada SubClase de otra clase llamada SuperClase. Se escribiría esto.

```
class SubClass extends SuperClass {
    . . .
}
```

Esto declara que SubClase es una subclase de SuperClase. Y también declara implícitamene que SuperClase es la superclase de SubClase. Una subclase también hereda variables y miembros de las superclases de su superclase, y así a lo largo del árbol de la herencia. Para hacer esta explicación un poco más sencilla, cuando este tutorial se refiere a la superclase de una clase significa el ancestro más directo de la clase así como a todas sus clases ascendentes.

Una clase Java sólo puede tener una superclase directa. Java no soporta la herencia múltiple.

Crear una subclase puede ser tan sencillo como incluir la clausula `extends` en la declaración de la clase. Sin embargo, normalmente se deberá realizar alguna cosa más cuando se crea una subclase, como sobreescribir métodos, etc...

¿Qué variables miembro hereda una subclase?

Regla:

Una subclase hereda todas las variables miembros de su superclase que puedan ser accesibles desde la subclase (a menos que la variable miembro esté oculta en la subclase).

Esto es, las subclases.

- heredan aquellas variables miembros declaradas como **public** o **protected**
- heredan aquellas variables miembros declaradas sin especificador de acceso (normalmente conocidas como "Amigas") siempre que la subclase esté en el mismo paquete que la clase
- no hereda las variables miembros de la superclase si la subclase declara una variable miembro que utiliza el mismo nombre. La variable miembro de la subclase se dice que oculta a la variable miembro de la superclase.
- no hereda las variables miembro **private**

Ocultar Variables Miembro

Como se mencionó en la sección anterior, las variables miembros definidas en la subclase ocultan las variables miembro que tienen el mismo nombre en la superclase.

Como esta característica del lenguaje Java es poderosa y conveniente, puede ser una fuente de errores: ocultar una variable miembro puede hacerse deliberadamente o por accidente. Entonces, cuando nombres tus variables miembro se cuidadoso y oculta sólo las variables miembro que realmente deseas ocultar.

Una caracteristica interesante de las variables miembro en Java es que una clase puede acceder a una variable miembro oculta a través de su superclase. Considere este pareja de superclase y subclase.

```
class Super {
    Number unNumero;
}
class Sub extends Super {
    Float unNumero;
}
```

La variable unNumero de Sub oculta a la variable unNumero de Super. Pero se puede acceder a la variable de la superclase utilizando.

```
super.unNumero
```

super es una palabra clave del lenguaje Java que permite a un método referirse a las variables ocultas y métodos sobreescritos de una superclase.

¿Qué métodos hereda una Subclase?

La regla que especifica los métodos heredados por una subclase es similar a la de las variables miembro.

> Regla:
>
> Una subclase hereda todos los métodos de sus superclase que son accesibles para la subclase (a menos que el método sea sobreescrito por la subclase).

Esto es, una Subclase.

- hereda aquellos métodos declarados como **public** o **protected**
- hereda aquellos métodos sin especificador de acceso, siempre que la subclase esté en el mismo paquete que la clase.
- no hereda un método de la superclase si la subclase declara un método que utiliza el mismo nombre. Se dice que el método de la subclase sobreescribe al método de la superclase.
- no hereda los métodos **private**.

Sobreescribir Métodos

La habilidad de una subclase para sobreescribir un método de su superclase permite a una clase heredar de su superclase aquellos comportamientos "más cercanos" y luego suplementar o modificar el comportamiento de la superclase.

En esta página:

Sobreescribir Métodos

Una subclase puede sobreescribir completamente la implementación de un método heredado o puede mejorar el método añadiendole funcionalidad.

Reemplazar la Implementación de un Método de una Superclase

Algunas veces, una subclase querría reemplazar completamente la implementación de un método de su superclase. De hecho, muchas superclases proporcionan implementaciones de métodos vacías con la esperanza de que la mayoría, si no todas, sus subclases reemplacen completamente la implementación de ese método.

Un ejemplo de esto es el método `run()` de la clase Thread. La clase Thread proporciona una implementación vacía (el método no hace nada) para el método `run()`, porque por definición, este método depende de la subclase. La clase Thread posiblemente no puede proporcionar una implementación medianamente razonable del método `run()`.

Para reemplazar completamente la implementación de un método de la superclase, simplememte se llama a un método con el mismo nombre que el del método de la superclase y se sobreescribe el método con la misma firma que la del método sobreescrito.

```
class ThreadSegundoPlano extends Thread {
    void run() {
        . . .
    }
}
```

La clase ThreadSegundoPlano sobreescribe completamente el método `run()` de su superclase y reemplaza completamente su implementación.

Añadir Implementación a un Método de la Superclase

Otras veces una subclase querrá mantener la implementación del método de su superclase y posteriormente ampliar algún comportamiento específico de la subclase. Por ejemplo, los métodos constructores de una subclase lo hacen normalmente--la subclase quiere preservar la inicialización realizada por la superclase, pero proporciona inicialización adicional específica de la subclase.

Supongamos que queremos crear una subclase de la clase Windows del paquete java.awt. La clase Windows tiene un constructor que requiere un argumento del tipo Frame que es el padre de la ventana.

```
public Window(Frame parent)
```

Este constructor realiza alguna inicialización en la ventana para que trabaje dentro del sistema de ventanas. Para asegurarnos de que una subclase de Windows también trabaja dentro del sistema de ventanas, deberemos proporcionar un constructor que realice la misma inicialización.

Mucho mejor que intentar recrear el proceso de inicialización que ocurre dentro del constructor de Windows, podríamos utilizar lo que la clase Windows ya hace. Se puede utilizar el código del constructor de Windows llamámdolo desde dentro del constructor de la subclase Window.

```
class Ventana extends Window {
    public Ventana(Frame parent) {
        super(parent);
```

```
        . . .
            // Ventana especifica su inicialización aquí
        . . .
    }
}
```

El constructor de Ventana llama primero al constructor de su superclase, y no hace nada más. Típicamente, este es el comportamiento deseado de los constructores--las superclases deben tener la oportunidad de realizar sus tareas de inicialización antes que las de su subclase. Otros tipos de métodos podrían llamar al constructor de la supeclase al final del método o en el medio.

Métodos que una Subclase no Puede Sobreescribir

- Una subclase no puede sobreescribir métodos que hayan sido declarados como **final** en la superclase (por definición, los métodos finales no pueden ser sobreescritos). Si intentamos sobreescribir un método final, el compilador mostrará un mensaje similar a este y no compilará el programa.

  ```
  FinalTest.java:7: Final methods can't be overriden. Method void iamfinal()
  is final in class ClassWithFinalMethod.
      void iamfinal() {
           ^
  1 error
  ```

  Para una explicación sobre los métodos finales, puedes ver: Escribir Métodos y Clases Finales.

- Una subclase tampoco puede sobreescribir métodos que se hayan declarado como **static** en la superclase. En otras palabras, una subclase no puede sobreescribir un método de clase. Puedes ver Miembros del Ejemplar y de la Clase para obtener una explicación sobre los métodos de clase.

Métodos que una Subclase debe Sobreescribir

Las subclases deben sobreescribir aquellos métodos que hayan sido declarados como abstract en la superclase, o la propia subclase debe ser abstracta. Escribir Clases y Métodos Abstractos explica con más detalle los métodos y clases abstractos.

TutorJava Nivel Básico

En esta página:

Escribir Clases y Métodos Finales

Se puede declarar que una clase sea final; esto es, que la clase no pueda tener subclases. Existen (al menos) dos razones por las que se querría hacer esto: razones de seguridad y de diseño.

Seguridad: Un mecanismo que los hackers utilizan para atacar sistemas es crear subclases de una clase y luego sustituirla por el original. Las subclases parecen y sienten como la clase original pero hacen cosas bastante diferentes, probablemente causando daños u obteniendo información privada. Para prevenir esta clase de subversión, se puede declarar que la clase sea final y así prevenir que se cree cualquier subclase.

La clase String del paquete java.lang es una clase final sólo por esta razón. La clase String es tan vital para la operación del compilador y del intérprete que el sistema Java debe garantizar que siempre que un método o un objeto utilicen un String, obtenga un objeto java.lang.String y no algún otro string. Esto asegura que ningún string tendrá propiedades extrañas, incosistentes o indeseables.

Si se intenta compilar una subclase de una clase final, el compilador mostrará un mensaje de error y no compilará el programa. Además, los bytescodes verifican que no está teniendo lugar una subversión, al nivel de byte comprobando que una clase no es una subclase de una clase final.

Diseño: Otra razón por la que se podría querer declarar una clase final son razones de diseño orientado a objetos. Se podría pensar que una clase es "perfecta" o que, conceptualmente hablando, la clase no debería tener subclases.

Para especificar que una clase es una clase final, se utiliza la palabra clave final antes de la palabra clave class en la declaración de la clase. Por ejemplo, si quisieramos declarar AlgoritmodeAjedrez como una clase final (perfecta), la declaración se parecería a esto.

```
final class AlgoritmodeAjedrez {
    . . .
}
```

Cualquier intento posterior de crear una subclase de AlgoritmodeAjedrez resultará en el siguiente error del compilador.

```
Chess.java:6: Can't subclass final classes: class AlgoritmodeAjedrez
class MejorAlgoritmodeAjedrez extends AlgoritmodeAjedrez {
      ^
1 error
```

Métodos Finales

Si la creacción de clases finales parece algo dura para nuestras necesidades, y realmente lo que se quiere es proteger son algunos métodos de una clase para que no sean sobreescritos, se puede utilizar la palabra clave final en la declaración de método para indicar al compilador que este método no puede ser sobreescrito por las subclases.

Se podría desear hacer que un método fuera final si el método tiene una implementación que no debe ser cambiada y que es crítica para el estado consistente del objeto. Por ejemplo, en lugar de hacer AlgoritmodeAjedrez como una clase final, podríamos hacer siguienteMovimiento() como un método final.

```
class AlgoritmodeAjedrez {
    . . .
    final void siguienteMovimiento(Pieza piezaMovida,
                        PosicionenTablero nuevaPosicion) {
    }
    . . .
}
```

En esta página:

Escribir Clases y Métodos Abstractos

Algunas veces, una clase que se ha definido representa un concepto abstracto y como tal, no debe ser ejemplarizado. Por ejemplo, la comida en la vida real. ¿Has visto algún ejemplar de comida? No. Lo que has visto son ejemplares de manzanas, pan, y chocolate. Comida representa un concepto abstracto de cosas que son comestibles. No tiene sentido que exista un ejemplar de comida.

Similarmente en la programación orientada a objetos, se podrían modelar conceptos abstractos pero no querer que se creen ejemplares de ellos. Por ejemplo, la clase Number del paquete java.lang representa el concepto abstracto de número. Tiene sentido modelar números en un programa, pero no tiene sentido crear un objeto genérico de números. En su lugar, la clase Number sólo tiene sentido como superclase de otras clases como Integer y Float que implementan números de tipos específicos. Las clases como Number, que implementan conceptos abstractos y no deben ser ejemplarizadas, son llamadas clases abstractas. Una clase abstracta es una clase que sólo puede tener subclases--no puede ser ejemplarizada.

Para declarar que una clase es un clase abstracta, se utiliza la palabra clave `abstract` en la declaración de la clase.

```
abstract class Number {
    . . .
}
```

Si se intenta ejemplarizar una clase abstracta, el compilador mostrará un error similar a este y no compilará el programa.

```
AbstractTest.java:6: class AbstractTest is an abstract class. It can't be
instantiated.
        new AbstractTest();
        ^
1 error
```

Métodos Abstractos

Una clase abstracta puede contener métodos abstractos, esto es, métodos que no tienen implementación. De esta forma, una clase abstracta puede definir un interface de programación completo, incluso porporciona a sus subclases la declaración de todos los métodos necesarios para implementar el interface de programación. Sin embargo, las clases abstractas pueden dejar algunos detalles o toda la implementación de aquellos métodos a sus subclases.

Veamos un ejemplo de cuando sería necesario crear una clase abstracta con métodos abstractos. En una aplicación de dibujo orientada a objetos, se pueden dibujar círculos, rectángulos, líneas, etc.. Cada uno de esos objetos gráficos comparten ciertos estados (posición, caja de dibujo) y comportamiento (movimiento, redimensionado, dibujo). Podemos aprovecharnos de esas similitudes y declararlos todos a partir de un mismo objeto padre-ObjetoGrafico.

Sin embargo, los objetos gráficos también tienen diferencias subtanciales: dibujar un círculo es bastante diferente a dibujar un rectángulo. Los objetos gráficos no pueden compartir estos tipos de estados o comportamientos. Por otro lado, todos los ObjetosGraficos deben saber como dibujarse a sí mismos; se diferencian en cómo se dibujan unos y otros. Esta es la situación perfecta para una clase abstracta.

Primero se debe declarar una clase abstracta, ObjetoGrafico, para proporcionar las variables miembro y los métodos que van a ser compartidos por todas las subclases, como la posición actual y

el método moverA().

También se deberían declarar métodos abstractos como dibujar(), que necesita ser implementado por todas las subclases, pero de manera completamente diferente (no tiene sentido crear una implementación por defecto en la superclase). La clase ObjetoGrafico se parecería a esto.

```
abstract class ObjetoGrafico {
    int x, y;
    . . .
    void moverA(int nuevaX, int nuevaY) {
        . . .
    }
    abstract void dibujar();
}
```

Todas las subclases no abstractas de ObjetoGrafico como son Circulo o Rectangulo deberán proprocionar una implementación para el método dibujar().

```
class Circulo extends ObjetoGrafico {
    void dibujar() {
        . . .
    }
}
class Rectangulo extends ObjetoGrafico {
    void dibujar() {
        . . .
    }
}
```

Una clase abstracta no necesita contener un método abstracto. Pero todas las clases que contengan un método abstracto o no proporcionen implementación para cualquier método abstracto declarado en sus superclases debe ser declarada como una clase abstracta.

En esta página:

La Clase Object

La clase Object está situada en la parte más alta del árbol de la herencia en el entorno de desarrollo de Java. Todas las clases del sistema Java son descendentes (directos o indirectos) de la clase Object. Esta clase define los estados y comportamientos básicos que todos los objetos deben tener, como la posibilidad de compararse unos con otros, de convertirse a cadenas, de esperar una condición variable, de notificar a otros objetos que la condición varible a cambiado y devolver la clase del objeto.

El método equals()

equals() se utiliza para comparar si dos objetos son iguales. Este método devuelve true si los objetos son iguales, o false si no lo son. Observe que la igualdad no significa que los objetos sean el mismo objeto. Consideremos este código que compara dos enteros.

```
Integer uno = new Integer(1), otroUno = new Integer(1);

if (uno.equals(otroUno))
    System.out.println("Los objetos son Iguales");
```

Este código mostrará Los objetos son Iguales aunque uno y otroUno referencian a dos objetos distintos. Se les considera iguales porque su contenido es el mismo valor entero.

Las clases deberían sobreescribir este método proporcionando la comprobación de igualdad apropiada. Un método equals() debería comparar el contenido de los objetos para ver si son funcionalmente iguales y devolver true si es así.

El método getClass()

El método getClass() es un método final (no puede sobreescribirse) que devuelve una representación en tiempo de ejecución de la clase del objeto. Este método devuelve un objeto Class al que se le puede pedir varia información sobre la clase, como su nombre, el nombre de su superclase y los nombres de los interfaces que implementa. El siguiente método obtiene y muestra el nombre de la clase de un objeto.

```
void PrintClassName(Object obj) {
    System.out.println("La clase del Objeto es " +
                obj.getClass().getName());
}
```

Un uso muy manejado del método getClass() es crear un ejemplar de una clase sin conocer la clase en el momento de la compilación. Este método de ejemplo, crea un nuevo ejemplar de la misma clase que obj que puede ser cualquier clase heredada desde Object (lo que significa que podría ser cualquier clase).

```
Object createNewInstanceOf(Object obj) {
    return obj.getClass().newInstance();
}
```

El método toString()

Este método devuelve una cadena de texto que representa al objeto. Se puede utilizar toString para mostrar un objeto. Por ejemplo, se podría mostrar una representación del Thread actual de la siguiente forma.

```
System.out.println(Thread.currentThread().toString());
System.out.println(new Integer(44).toString());
```

La representación de un objeto depende enteramente del objeto. El String de un objeto entero es el valor del entero mostrado como texto. El String de un objeto Thread contiene varios atributos sobre el thread, como su nombre y prioridad. Por ejemplo, las dos líneas anteriores darían la siguiente salida.

```
Thread[main,5,main]
4
```

El método toString() es muy útil para depuración y también puede sobreescribir este método en todas las clases.

Otros métodos de Object cubiertos en otras lecciones o secciones

La clase Object proporciona un método, finalize() que limpia un objeto antes de recolectar la basura. Este método se explica en la lección Eliminar Objetos no Utilizados. También en: Escribir un Método finalize() puedes ver cómo sobreescribir este método para manejar las necesidades de finalización de las clases

La clase Object tambiém proporciona otros cinco métodos.

- **notify()**
- **notifyAll()**
- **wait()** (tres versiones)

que son críticos cuando se escriben programas Java con múltiples thread. Estos métodos ayudan a asegurarse que los thread están sincronizados y se cubren en Threads de Control.

TutorJava Nivel Básico

En esta página:

¿Qué es un Interface

Definición:

Un interface es una colección de definiciones de métodos (sin implementaciones) y de valores constantes.

Los interfaces se utilizan para definir un protocolo de comportamiento que puede ser implementado por cualquier clase del árbol de clases.

Los interfaces son útiles para.

- capturar similitudes entre clases no relacionadas sin forzar una relación entre ellas.
- declarar métodos que una o varias clases necesitan implementar.
- revelar el interface de programación de un objeto sin recelar sus clases (los objetos de este tipo son llamados objetos anónimos y pueden ser útiles cuando compartas un paquete de clases con otros desarrolladores).

En Java, un interface es un tipo de dato de referencia, y por tanto, puede utilizarse en muchos de los sitios donde se pueda utilizar cualquier tipo (como en un argumento de métodos y una declaración de variables). Podrás ver todo esto en: Utilizar un Interface como un Tipo.

Los Interfaces No Proporcionan Herencia Múltiple

Algunas veces se tratra a los interfaces como una alternativa a la herencia múltiple en las clases. A pesar de que los interfaces podrían resolver algunos problemas de la herencia múltiple, son animales bastantes diferentes. En particular.

- No se pueden heredar variables desde un interface.
- No se pueden heredar implementaciones de métodos desde un interface.
- La herencia de un interface es independiente de la herencia de la clase--las clases que implementan el mismo interface pueden o no estar relacionadas a través del árbol de clases.

TutorJava Nivel Básico

En esta página:

Definir un Interface

Para crear un Interface, se debe escribir tanto la declaración como el cuerpo del interface.

```
declaraciondeInterface {
    cuerpodeInterface
}
```

La Declaración de Interface declara varios atributos del interface, como su nombre o si se extiende desde otro interface. El Cuerpo de Interface contiene las constantes y las declaraciones de métodos del Interface.

La Declaración de Interface

Como mínimo, una declaración de interface contiene la palabra clave interface y el nombre del interface que se va a crear.

```
interface Contable {
    . . .
}
```

Nota: Por convención, los nombres de interfaces empiezan con una letra mayúscula al igual que las clases. Frecuentemente los nombres de interfaces terminan en "able" o "ible".

Una declaración de interface puede tener otros dos componentes: el especificador de acceso public y una lista de "superinterfaces". Un interface puede extender otros interfaces como una clase puede extender o subclasificar otra case. Sin embargo, mientras que una clase sólo puede extender una superclase, los interfaces pueden extender de cualquier número de interfaces. Así, una declaración completa de interface se parecería a esto.

```
[public] interface Nombredenterface [extends listadeSuperInterfaces] {
    . . .
}
```

El especificador de acceso public indica que el interface puede ser utilizado por todas las clases en cualquier paquete. Si el interface no se especifica como público, sólo será accesible para las clases definidas en el mismo paquete que el interface.

La clausula extends es similar a la utilizada en la declaración de una clase, sin embargo, un interface puede extender varios interfaces (mientras una clase sólo puede extender una), y un interface no puede extender clases. Esta lista de superinterfaces es un lista delimitada por comas de todos los interfaces extendidos por el nuevo interface.

Un interface hereda todas las constantes y métodos de sus superinterfaces a menos que el interface oculte una constante con el mismo nombre o redeclare un método con una nueva declaración.

El cuerpo del Interface

El cuerpo del interface contiene las declaraciones de métodos para los métodos definidos en el interface. Implementar Métodos muestra cómo escribir una declaración de método. Además de las declaraciones de los métodos, un interface puede contener declaraciones de constantes. En Declarar

Variables Miembros existe más información sobre cómo construir una declaración de una variable miembro.

Nota:

Las declaraciones de miembros en un interface no permiten el uso de algunos modificadores y desaconsejan el uso de otros. No se podrán utilizar transient, volatile, o synchronized en una declaración de miembro en un interface. Tampoco se podrá utilizar los especificadores private y protected cuando se declaren miembros de un interface.

Todos los valores constantes definidos en un interface son implícitamente públicos, estáticos y finales. El uso de estos modificadores en una declaración de constante en un interface está desaconsejado por falta de estilo. Similarmente, todos los métodos declarados en un interface son implícitamente públicos y abstractos.

Este código define un nuevo interface llamado coleccion que contiene un valor constante y tres declaraciones de métodos.

```
interface coleccion {
    int MAXIMO = 500;

    void añadir(Object obj);
    void borrar(Object obj);
    Object buscar(Object obj);
    int contadorActual();
}
```

El interface anterior puede ser implementado por cualquier clase que represente una colección de objetos como pueden ser pilas, vectores, enlaces, etc...

Observa que cada declaración de método está seguida por un punto y coma (;) porque un interface no proporciona implementación para los métodos declarados dentro de él.

En esta página:

Utilizar un Interface

Para utilizar un interface se debe escribir una clase que lo implemente. Una clase declara todos los interfaces que implementa en su declaración de clase. Para declarar que una clase implementa uno o más interfaces, se utiliza la palabra clave implements seguida por una lista delimitada por comas con los interfaces implementados por la clase.

Por ejemplo, consideremos el interface coleccion presentado en la página anterior. Ahora, supongamos que queremos escribir una clase que implemente un pila FIFO (primero en entrar, primero en salir). Como una pila FIFO contiene otros objetos tiene sentido que implemente el interface coleccion. La clase PilaFIFO declara que implementa el interface coleccion de esta forma.

```
class PilaFIFO implements coleccion {
    . . .
    void añadir(Object obj) {
        . . .
    }
    void borrar(Object obj) {
        . . .
    }
    Object buscar(Object obj) {
        . . .
    }
    int contadorActual() {
        . . .
    }
}
```

así se garantiza que proporciona implementación para los métodos añadir(), borrar(), buscar() y contadorActual().

Por convención, la clausula implements sigue a la clausula extends si es que ésta existe.

Observa que las firmas de los métodos del interface coleccion implementados en la clase PilaFIFO debe corresponder exactamente con las firmas de los métodos declarados en la interface coleccion.

En esta página:

Utilizar un Interface como un Tipo

Como se mencionó anteriormente, cuando se define un nuevo interface, en esencia se está definiendo un tipo de referencia. Se pueden utilizar los nombres de interface en cualquier lugar donde se usaría un nombre de dato de tipos primitivos o un nombre de datos del tipo de referencia.

Por ejemplo, supongamos que se ha escrito un programa de hoja de cálculo que contiene un conjunto tabular de celdas y cada una contiene un valor. Querríamos poder poner cadenas, fechas, enteros, ecuaciones, en cada una de las celdas de la hoja. Para hacer esto, las cadenas, las fechas, los enteros y las ecuaciones tienen que implementar el mismo conjunto de métodos. Una forma de conseguir esto es encontrar el ancestro común de las clases e implementar ahí los métodos necesarios. Sin embargo, esto no es una solución práctica porque el ancestro común más frecuente es Object. De hecho, los objetos que puede poner en las celdas de su hoja de cálculo no están relacionadas entre sí, sólo por la clase Object. Pero no puede modificar Object.

Una aproximación podría ser escribir una clase llamada ValordeCelda que representara los valores que pudiera contener una celda de la hoja de cálculo. Entonces se podrían crear distintas subclases de ValordeCelda para las cadenas, los enteros o las ecuaciones. Además de ser mucho trabajo, esta aproximación arbitraria fuerza una relación entre esas clases que de otra forma no sería necesaria, y debería duplicar e implementar de nuevo clases que ya existen.

Se podría definir un interface llamado CellAble que se parecería a esto.

```
interface CellAble {
    void draw();
    void toString();
    void toFloat();
}
```

Ahora, supongamos que existen objetos Linea y Columna que contienen un conjunto de objetos que implementan el interface CellAble. El método setObjectAt() de la clase Linea se podría parecer a esto.

```
class Linea {
    private CellAble[] contents;
    . . .
    void setObjectAt(CellAble ca, int index) {
        . . .
    }
    . . .
}
```

Observa el uso del nombre del interface en la declaración de la variable miembro contents y en la declaración del argumento ca del método. Cualquier objeto que implemente el interface CellAble, sin importar que exista o no en el árbol de clases, puede estar contenido en el array contents y podría ser pasado al método setObjectAt().

En esta página:

Crear Paquetes

Los paquetes son grupos relacionados de clases e interfaces y proporcionan un mecanismo conveniente para menejar un gran juego de clases e interfaces y evitar los conflictos de nombres. Además de los paquetes de Java puede crear tus propios paquetes y poner en ellos definiciones de clases y de interfaces utilizando la sentencia package.

Supongamos que se está implementando un grupo de clases que representan una colección de objetos gráficos como círculos, rectángulos, líneas y puntos. Además de estas clases tendrás que escribir un interface Draggable para que en las clases que lo implementen pueda moverse con el ratón. Si quieres que estas clases estén disponibles para otros programadores, puedes empaquetarlas en un paquete, digamos, graphics y entregar el paquete a los programadores (junto con alguna documentación de referencia, como qué hacen las clases y los interfaces y qué interfaces de programación son públicos).

De esta forma, otros programadores pueden determinar fácilmente para qué es tu grupo de clases, cómo utilizarlos, y cómo relacionarlos unos con otros y con otras clases y paquetes.

Los nombres de clases no tienen conflictos con los nombres de las clases de otros paquetes porque las clases y los interfaces dentro de un paquete son referenciados en términos de su paquete (técnicamente un paquete crea un nuevo espacio de nombres).

Se declara un paquete utilizando la sentencia package.

```
package graphics;

interface Draggable {
    . . .
}

class Circle {
    . . .
}

class Rectangle {
    . . .
}
```

La primera línea del código anterior crea un paquete llamado graphics. Todas las clases e interfaces definidas en el fichero que contiene esta sentencia son miembros del paquete. Por lo tanto, Draggable, Circle,y Rectangle son miembros del paquete graphics.

Los ficheros .class generados por el compilador cuando se compila el fichero que contiene el fuente para Draggable, Circle y Rectangle debe situarse en un directorio llamado graphics en algún lugar del path CLASSPATH. CLASSPATH es una lista de directorios que indican al sistema donde ha instalado varias clases e interfaces compiladas Java. Cuando busque una clase, el intérprete Java busca un directorio en su CLASSPATH cuyo nombre coincida con el nombre del paquete del que la clase es miembro. Los ficheros .class para todas las clases e interfaces definidas en un paquete deben estar en ese directorio de paquete.

Los nombres de paquetes pueden contener varios componentes (separados por puntos). De hecho, los nombres de los paquetes de Java tienen varios componentes: java.util, java.lang, etc...

Cada componente del nombre del paquete representa un directorio en el sistema de ficheros. Así, los ficheros .class de java.util están en un directorio llamado util en otro directorio llamado java en algún lugar del CLASSPATH.

CLASSPATH

Para ejecutar una aplicación Java, se especifica el nombre de la aplicación Java que se desea ejecutar en el interprete Java. Para ejecutar un applet, se especifica el nombre del applet en una etiqueta <APPLET> dentro de un fichero HTML. El navegador que ejecute el applet pasa el nombre del applet al intérprete Java. En cualquier caso, la aplicación o el applet que se está ejecutando podría estár en cualquier lugar del sistema o de la red. Igualmemente, la aplicación o el applet pueden utilizar otras clases y objetos que están en la misma o diferentes localizaciones.

Como las clases pueden estar en cualquier lugar, se debe indicar al interprete Java donde puede encontrarlas. Se puede hacer esto con la variable de entorno CLASSPATH que comprende una lista de directorios que contienen clases Java compiladas. La construcción de CLASSPATH depende de cada sistema.

Cuando el interprete obtiene un nombre de clase, desde la línea de comandos, desde un navegador o desde una aplicación o un applet, el interprete busca en todos los directorios de CLASSPATH hasta que encuentra la clase que está buscando.

Se deberá poner el directorio de nivel más alto que contiene las clases Java en el CLASSPATH. Por convención, mucha gente tiene un directorio de clases en su directorio raíz donde pone todo su código Java. Si tu tienes dicho directorio, deberías ponerlo en el CLASSPATH. Sin embargo, cuando se trabaja con applets, es conveniente poner el applet en un directorio clases debajo del directorio donde está el fichero HTML que contiene el applet. Por esta, y otras razones, es conveniente poner el directorio actual en el CLASSPATH.

Las clases incluidas en el entorno de desarrollo Java están disponibles automáticamente porque el interprete añade el directorio correcto al CLASSPATH cuando arranca.

Observa que el orden es importante. Cuando el interprete Java está buscando una clase, busca por orden en los directorios indicados en CLASSPATH hasta que encuentra la clase con el nombre correcto. El interprete Java ejecuta la primera clase con el nombre correcto que encuentre y no busca en el resto de directorios. Normalmente es mejor dar a las clases nombres únicos, pero si no se puede evitar, asegúrate de que el CLASSPATH busca las clases en el orden apropiado. Recuerda esto cuando selecciones tu CLASSPATH y el árbol del código fuente.

Nota:

Todas las clases e interfaces pertenecen a un paquete. Incluso si no especifica uno con la sentencia package. Si no se especifica las clases e interfaces se convierten en miembros del paquete por defecto, que no tiene nombre y que siempre es importado.

En esta página:

Utilizar Paquetes

Para importar una clase específica o un interface al fichero actual (como la clase Circle desde el paquete graphics creado en la sección anterior) se utiliza la sentencia de import.

```
import graphics.Circle;
```

Esta sentencia debe estar al principio del fichero antes de cualquier definición de clase o de interface y hace que la clase o el interface esté disponible para su uso por las clases y los interfaces definidos en el fichero.

Si se quieren importar todas las clases e interfaces de un paquete, por ejemplo, el paquete graphics completo, se utiliza la sentencia import con un caracter comodín, un asterisco '*'.

```
import graphics.*;
```

Si intenta utilizar una clase o un interface desde un paquete que no ha sido importado, el compilador mostrará este error.

```
testing.java:4: Class Date not found in type declaration.
        Date date;
        ^
```

Observa que sólo las clases e intefaces declarados como públicos pueden ser utilizados en clases fuera del paquete en el que fueron definidos.

El paquete por defecto (un paquete sin nombre) siempre es importado. El sistema de ejecución también importa automáticamente el paquete java.lang.

Si por suerte, el nombre de una clase de un paquete es el mismo que el nombre de una clase en otro paquete, se debe evitar la ambigüedad de nombres precediendo el nombre de la clase con el nombre del paquete. Por ejemplo, previamente se ha definido una clase llamada Rectangle en el paquete graphics. El paquete java.awt también contiene una clase Rectangle. Si estos dos paquetes son importados en la misma clase, el siguiente código sería ambigüo.

```
Rectangle rect;
```

En esta situación se tiene que ser más específico e indicar exactamente que clase Rectangle se quiere.

```
graphics.Rectangle rect;
```

Se puede hacer esto anteponiento el nombre del paquete al nombre de la clase y separando los dos con un punto.

En esta página:

Los Paquetes Java

El entorno de desarrollo estandard de Java comprende ocho paquetes.

El Paquete de Lenguaje Java

El paquete de lenguaje Java, también conocido como java.lang, contiene las clases que son el corazón del lenguaje Java. Las clases de este paquete se agrupan de la siguiente manera.

Object

El abuelo de todas las clases--la clase de la que parten todas las demás. Esta clase se cubrió anteriormene en la lección La Clase Object.

Tipos de Datos Encubiertos

Una colección de clases utilizadas para encubrir variables de tipos primitivos: Boolean, Character, Double, Float, Integer y Long. Cada una de estas clases es una subclase de la clase abstracta Number.

Strings

Dos clases que implementan los datos de caracteres. Las Clases String y StringBuffer es una lección donde aprenderás el uso de estos dos tipos de Strings.

System y Runtime

Estas dos clases permiten a los programas utilizar los recursos del sistema. System proporciona un interface de programación independiente del sistema para recursos del sistema y Runtime da acceso directo al entorno de ejecución específico de un sistema. Utilizar Recursos del Sistema Describe las clases System y Runtime y sus métodos.

Thread

Las clases Thread, ThreadDeath y ThreadGroup implementan las capacidades multitareas tan importantes en el lenguaje Java. El paquete java.lang también define el interface Runnable. Este interface es conveniente para activar la clase Java sin subclasificar la clase Thread. A través de un ejemplo de aproximación Threads de Control te enseñará los Threads Java.

Class

La clase Class proporciona una descripción en tiempo de ejecución de una clase y la clase ClassLoader permite cargar clases en los programas durante la ejecución.

Math

Una librería de rutinas y valores matemáticos como pi.

Exceptions, Errors y Throwable

Cuando ocurre un error en un programa Java, el programa lanza un objeto que indica qué problema era y el estado del interprete cuando ocurrió el error. Sólo los objetos derivados de la clase Throwable pueden ser lanzados. Existen dos subclasses principales de Throwable: Exception y Error. Exception

es la forma que deben intentar capturar los programas normales. Error se utiliza para los errores catastróficos--los programas normales no capturan Errores. El paquete java.lang contiene las clases Throwable, Exception y Error, y numerosas subclases de Exception y Error que representan problemas específicos. Manejo de Errores Utilizando Excepciones te muestra cómo utilizar las excepciones para manejar errores en sus programas Java.

Process

Los objetos Process representan el proceso del sistema que se crea cuando se utiliza el sistema en tiempo de ejecución para ejecutar comandos del sistema. El paquete java.lang define e implementa la clase genérica Process.

El compilador importa automáticamente este paquete. Ningún otro paquete se importa de forma automática.

El Paquete I/O de Java

El paquete I/O de Java (java.io) proporciona un juego de canales de entrada y salida utilizados para leer y escribir ficheros de datos y otras fuentes de entrada y salida. Las clases e interfaces definidos en java.io se cubren completamente en Canales de Entrada y Salida.

El Paquete de Utilidades de Java

Este paquete, java.util, contiene una colección de clases útiles. Entre ellas se encuentran muchas estructuras de datos genéricas (Dictionary, Stack, Vector, Hashtable) un objeto muy útil para dividir cadenas y otro para la manipulación de calendarios. El paquete java.util también contiene el interface Observer y la clase Observable que permiten a los objetos notificarse unos a otros cuando han cambiado. Las clases de java.util no se cubre en este tutorial aunque algunos ejemplos utilizan estas clases.

El Paquete de Red de Java

El paquete java.net contiene definiciones de clases e interfaces que implementan varias capacidades de red. Las clases de este paquete incluyen una clase que implementa una conexión URL. Se pueden utilizar estas clases para implementar aplicaciones cliente-servidor y otras aplicaciones de comunicaciones. Conectividad y Seguridad del Cliente tiene varios ejemplos de utilización de estas clases, incluyendo un ejemplo cliente-servidor que utiliza datagramas.

El Paquete Applet

Este paquete contiene la clase Applet -- la clase que se debe subclasificar si se quiere escribir un applet. En este paquete se incluye el interface AudioClip que proporciona una abstración de alto nivel para audio.Escribir Applets.

Los Paquetes de Herramientas para Ventanas Abstractas

Tres paquetes componen las herramientas para Ventanas Abstractas: java.awt, java.awt.image, y java.awt.peer.

El paquete AWT

El paquete **java.awt** proporciona elementos GUI utilizados para obtener información y mostrarla en la pantalla como ventanas, botones, barras de desplazamiento, etc..

El paquete AWT Image

El paquete **java.awt.image** contiene clases e interfaces para manejar imágenes de datos, como la selección de un modelo de color, el cortado y pegado, el filtrado de colores, la selección del valor de un pixel y la grabación de partes de la pantalla.

El paquete AWT Peer

El paquete **java.awt.peer** contiene clases e interfaces que conectan los componentes AWT independientes de la plataforma a su implementación dependiente de la plataforma (como son los controles de Microsoft Windows).

En esta página:

Cambios en el JDK 1.1 que afectan a Objetos, Clases e Interfaces

La declaración de Clase El JDK 1.1 permite declarar clases internas. Puedes ver Cambios en el JDK 1.1: Clases Internas.

Los paquetes de Java Se han añadido quince nuevos paquetes al juego conjunto java.* y se ha eliminado uno. Puedes ver Cambios en el JDK 1.1: los Paquetes java.*.

El Paquete java.lang Se han añadido tres clases al paquete java.lang. Puedes ver Cambios en el JDK 1.1: el paquete java.lang.

El paquete java.io Se han añadido nuevas clases al paquete java.io para soportar la lectura y escritura de caracteres de 16 Bits Unicode. Puedes ver Cambios en el JDK 1.1: el paquete java.io.

El paquete java.util Se han añadido nuevas clases al paquete java.util para soporte de internacionalización y manejo de eventos. Puedes ver Cambios en el JDK 1.1: el paquete java.util.

El paquete java.net Se han añadido muchas clases al paquete java.net.

El paquete Applet Para un sumario de los cambios en el paquete java.applet puedes ver, [PENDIENTE].

El Paquete java.awt Para información sobre los cambios en los paquetes java.awt puedes ver las páginas Cambios en el GUI: el AWT crece

En esta página:

Cambios en el JDK 1.1: Clases Internas

Las versiones anteriores del lenguaje Java requerían que todas las clases fueran declaradas como miembros de un paquete (llamadas clases de nivel superior). La versión del JDK 1.1 elimina esta restricción y permite que las clases sean declaradas en cualquier ámbito (llamadas clases internas).

A parte de otros usos, la clases internas se han proporcionado para simplificar la sintaxis de creación de clases adaptadoras -- clases que implementan un interface (o clase) requeridas por un API, y delega el flujo de control en un objeto "principal" que las encierra. Utilizar Adaptadores y Clases Internas para Manejar eventos del aWT explica la utilización de las clases internas para éste propósito y proprociona varios ejemplo de ello. O si prefieres ir directo al; código, puedes ver alguno de estos ejemplos.

- La clase **MyAdapter** definida en la clase **SoundCanvas**
- Las clases **MyTextListener** y **MyTextActionListener** definidas en La clase **TextDemo**

Para más detalles sobre la síntaxis, las restricciones, etc, puedes ver la Especificación de las Clases Internas en la site de Sun.

En esta página:

Cambios en el JDK 1.1: Cambios en los Paquetes Java

Nuevos Paquetes java.*

Se han añadido los siguientes paquetes al JDK 1.1.

- java.awt.datatransfer, y java.awt.event
- java.beans
- java.lang.reflect
- java.math
- java.rmi, java.rmi.dgc, java.rmi.registry, y java.rmi.server
- java.security, java.security.acl, y java.security.interfaces
- java.sql
- java.text
- java.util.zip

Paquetes Eliminados de java.*

Se han eliminado los siguientes paquetes en la versión 1.1 del JDK.

- java.awt.peer

En esta página:

Cambios en el JDK 1.1: El paquete Java.Lang

Se han añadido tres nuevas clases de tipos de datos al paquete java.lang del JDK 1.1.

- **Byte**
- **Short**
- **Void**

Byte y Short son subclases de Number. Void es una subclase de Object. El tutorial no cubre ninguna de estas tres clases.

En esta página:

Cambios en el JDK 1.1: El Paquete Java.Util

Se han añadido ocho nuevas clases al paquete java.util para soportar internacionalización.

- **Calendar**
- **GregorianCalendar**
- **ListResourceBundle**
- **Locale**
- **PropertyResourceBundle**
- **ResourceBundle**
- **SimpleTimeZone**
- **TimeZone**

Se han añadido dos clases al paquete java.util para soportar manejo de eventos.

- **EventListener**
- **EventObject**

En esta página:

Las Clases String y StringBuffer

El paquete java.lang contiene dos clases de cadenas: String y StringBuffer. Ya hemos visto la clase String en varias ocasiones en este tutorial. La clase String se utiliza cuando se trabaja con cadenas que no pueden cambiar. Por otro lado, StringBuffer, se utiliza cuando se quiere manipular el contenido de una cadena.

El método reverseIt() de la siguiente clase utiliza las clases String y StringBuffer para invertir los caracteres de una cadena. Si tenemos una lista de palabras, se puede utilizar este método en conjunción de un pequeño programa para crear una lista de palabras rítmicas (una lista de palabras ordenadas por las silabas finales). Sólo se tienen que invertir las cadenas de la lista, ordenar la lista e invertir las cadenas otra vez.

```
class ReverseString {
    public static String reverseIt(String source) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);

        for (i = (len - 1); i >= 0; i--) {
            dest.append(source.charAt(i));
        }
        return dest.toString();
    }
}
```

El método reverseIt() acepta un argumento del tipo String llamado source que contiene la cadena que se va a invertir. El método crea un StringBuffer, dest, con el mismo tamaño que source. Luego hace un bucle inverso sobre los caracteres de source y los añade a dest, con lo que se invierte la cadena. Finalmente el método convierte dest, de StringBuffer a String.

Además de iluminar las diferencias entre String y StringBuffer, esta lección ilustra varias caracteristicas de las clases String y StringBuffer: Creacción de Strings y StringBuffers, utilizar métodos accesores para obtener información sobre String o StringBuffer, modificar un StringBuffer y convertir un tipo String a otro.

En esta página:

¿Por qué dos clases String?

```
class ReverseString {
    public static String reverseIt(String source) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);

        for (i = (len - 1); i >= 0; i--) {
            dest.append(source.charAt(i));
        }
        return dest.toString();
    }
}
```

El entorno de desarrollo Java proporciona dos clases para manipular y almacenar datos del tipo carácter: String, para cadenas constantes, y StringBuffer, para cadenas que pueden cambiar.

String se utiliza cuando no se quiere que cambie el valor de la cadena. Por ejemplo, si escribimos un método que necesite una cadena de caracteres y el método no va a modificar la cadena, deberíamos utilizar un objeto String. Normalmente, querremos utilizar Strings para pasar caracteres a un método y para devolver caracteres desde un método. El método `reverseIt()` toma un String como argumento y devuelve un String.

La clase StringBuffer proporcinada para cadenas variables; se utiliza cuando sabemos que el valor de la cadena puede cambiar. Normalmente utilizaremos StringBuffer para constuir datos de caracteres como en el método `reverseIt()`.

Como son constantes, los Strings son más económicos (utilizan menos memoria) que los StringBuffers y pueder ser compartidos. Por eso es importante utilizar String siempre que sea apropiado.

En esta página:

Crear String y StringBuffer

```
class ReverseString {
    public static String reverseIt(String source) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);

        for (i = (len - 1); i >= 0; i--) {
            dest.append(source.charAt(i));
        }
        return dest.toString();
    }
}
```

El método reverseIt() crea un StringBuffer llamado dest cuya longitud inicial es la misma que la de source. StringBuffer dest declara al compilador que dest se va a utilizar para referirse a un objeto del tipo StringBuffer, el operador new asigna memoria para un nuevo objeto y StringBuffer(len) inicializa el objeto. Estos tres pasos--declaración, ejemplarización e inicialziación-- se describen en: Crear Objetos.

Crear un String

Muchos Strings se crean a partir de cadenas literales. Cuando el compilador encuentra una serie de caracteres entre comillas (" y "), crea un objeto String cuyo valor es el propio texto. Cuando el compilador encuentra la siguente cadena, crea un objeto String cuyo valor es Hola Mundo.

```
"Hola Mundo."
```

También se pueden crear objetos String como se haría con cualquier otro objeto Java: utilizando new.

```
new String("Hola Mundo.");
```

Crear un StringBuffer

El método constructor utilizado por reverseIt() para incializar dest requiere un entero como argumento que indique el tamaño inicial del nuevo StringBuffer.

```
StringBuffer(int length)
```

reverseIt() podría haber utilizado el constructor por defecto para dejar indeterminada la longitud del buffer hasta un momento posterior. Sin embargo, es más eficiente especificar la longitud del buffer si se conoce, en vez de asignar memoria cada vez que se añadan caracteres al buffer.

En esta página:

Métodos Accesores

```
class ReverseString {
    public static String reverseIt(String source) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);

        for (i = (len - 1); i >= 0; i--) {
            dest.append(source.charAt(i));
        }
        return dest.toString();
    }
}
```

Las variables de ejemplar de un objeto están encapsuladas dentro del objeto, ocultas en su interior, seguras frente a la inspección y manipulación por otros objetos. Con ciertas excepciones bien definidas, los métodos del objeto no son los únicos a través de los cuales un objeto puede inspeccionar o alterar las variables de otro objeto. La encapsulación de los datos de un objeto lo protege de la corrupción de otros objetos y oculta los detalles de implementación a los objetos extraños. Esta encapsulación de datos detrás de los métodos de un objeto es una de las piedras angulares de la programación orientada a objetos.

Los métodos utilizados para obtener información de un objeto son conocidos como métodos accesores. El método reverseIt() utiliza dos métodos accesores de String para obtener información sobre el string source.

Primero utiliza el método accesor: length() para obtener la longitud de la cadena source.

```
int len = source.length();
```

Observa que a reverseIt() no le importa si el String mantiene su longitud como un entero, como un número en coma flotante o incluso si calcula la longitud al vuelo. reverseIt() simplemente utiliza el interface público del método length() que devuelve la longitud del String como un entero. Es todo lo que necesita saber reverseIt().

Segundo, utiliza el método accesor: charAt() que devuelve el carácter que está situado en la posición indicada en su argumento.

```
source.charAt(i)
```

El carácter devuelto por charAt() es el que se añade al StringBuffer dest. Como la variable del bucle i empieza al final de source y avanza hasta el principio de la cadena, los caracteres se añaden en orden inverso al StringBuffer.

Más Métodos Accesores

Además de length() y charAt(), String soporta otros métodos accesores que proporcionan acceso a subcadenas y que indican la posición de caracteres específicos en la cadena. StringBuffer tiene sus propios métodos accesores similares.

Para la Clase String

Además de los accesores length() y charAt() que hemos visto antes, la clase String proporciona dos accesores que devuelven la posición dentro de una cadena de un carácter específico: indexOf() y lastIndexOf(). El método indexOf() busca empezando desde el principio de la cadena y lastIndexOf() busca empezando desde el final.

Los métodos indexOf() y lastIndexOf() se utilizan frecuentemente con substring(), que devuelve una subcadena de una cadena. La siguiente clase ilustra el uso de lastIndexOf() y substring() para aislar diferentes partes de un nombre de fichero.

Nota: Estos métodos no dan ningún error de chequeo y asumen que el argumento contiene un path completo con nombres de directorios y un nombre de fichero con extensión. Si estos métodos se utilizaran en algún programa deberían comprobar que los argumentos estuvieran construidos de forma apropiada.

```
class NombreFichero {
    String path;
    char separador;

    NombreFichero(String str, char sep) {
        path = str;
        separador = sep;
    }

    String extension() {
        int punto = path.lastIndexOf('.');
        return path.substring(punto + 1);
    }

    String NombreFichero() {
        int punto = path.lastIndexOf('.');
        int sep = path.lastIndexOf(separador);
        return path.substring(sep + 1, punto);
    }

    String path() {
        int sep = path.lastIndexOf(separador);
        return path.substring(0, sep);
    }
}
```

Aquí hay un pequeño programa que construye un objeto NombreFichero y llama a todos sus métodos.

```
class NombreFicheroTest {
    public static void main(String[] args) {
        NombreFichero miHomePage = new NombreFichero("/home/index.html", '/');
        System.out.println("Extension = " + miHomePage.extension());
        System.out.println("Nombre de Fichero = " + miHomePage.NombreFichero());
        System.out.println("Path = " + miHomePage.path());
    }
}
```

Aquí tienes la salida del programa.

```
Extension = html
NombreFichero = index
Path = /home
```

El método extension() utiliza lastIndexOf() para localizar la última aparición de un punto ('.') en el nombre del fichero. Luego substring() utiliza el valor devuelto por lastIndexOf() para extraer la extensión del fichero--esto es, una subcadena desde el punto ('.') hasta el final de la cadena. Este código asume que el nombre del fichero tiene realmente un puno ('.'), si no lo tiene lastIndexOf() devuelve -1, y el método substring() lanzará una StringIndexOutOfBoundsException.

Observa, también que extension() utiliza dot + 1 como el argumento para substring(). Si el punto es el último carácter de la cadena, dot + 1 es igual a la longitud de la cadena que es uno más que la longitud del texto de la cadena (porque los indices empiezan en 0). Sin embargo, substring() accepta un índice igual (pero no mayor) que la longitud de la cadena y lo interpreta como "el final de la cadena".

Intenta esto: Inspeccione los otros métodos utilizados en la clase NombreFichero y observa cómo los métodos lastIndexOf() y substring() trabajan juntos para aislar diferentes partes de un nombre de fichero.

Mientras los méodos del ejemplo anterior sólo utilizan una versión del método lastIndexOf(), la clase String realmente soporta cuatro diferentes versiones de los métodos indexOf() y lastIndexOf(). Las cuatro versiones trabajan de la siguiente forma.

indexOf(int character)

lastIndexOf(int character)

Devuelve el índice de la primera (o última) ocurrencia del carácter especificado.

indexOf(int character, int from)

lastIndexOf(int character, int from)

Devuelve el índice de la primera (o última) ocurrencia del carácter escpecificado desde el índice especificado.

indexOf(String string)

lastIndexOf(String string)

Devuelve el índice de la primera (o última) ocurrencia de la Cadena especificada.

indexOf(String string, int from)

lastIndexOf(String string, int from)

Devuelve el índice de la primera (o última) ocurrencia de la cadena escpecificada desde el índice especificado.

Para la Clase StringBuffer

Al igual que String, StringBuffer porporciona los métodos accesores length() y charAt(). Además de estos dos, accesores, StringBuffer también tiene un método llamado capacity(). Este método es diferente de length() en que devuelve la cantidad de espacio asignado actualmente para el StringBuffer en vez de la cantidad de espacio utilizado. Por ejemplo, la capacidad del StringBuffer de reverseIt() no cambia nunca, aunque la longitud del StringBuffer se incremente en cada vuelta del bucle.

```
class ReverseString {
    public static String reverseIt(String source) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);

        for (i = (len - 1); i >= 0; i--) {
            dest.append(source.charAt(i));
        }
        return dest.toString();
    }
}
```

En esta página:

Modificar un StringBuffer

```
class ReverseString {
    public static String reverseIt(String source) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);

        for (i = (len - 1); i >= 0; i--) {
            dest.append(source.charAt(i));
        }
        return dest.toString();
    }
}
```

El método reverseIt() utiliza el método append() de StringBuffer para añadir un carácter al final de la cadena de destino: dest. Si la adicción de caracteres hace que aumente el tamaño de StringBuffer más allá de su capacidad actual, el StringBuffer asigna más memoria. Como la asignación de memoria es una operación relativamente cara, debemos hacer un código más eficiente inicializando la capacidad del StringBuffer de forma razonable para el primer contenido, así minimizaremos el número de veces que se tendrá que asignar memoria.

Por ejemplo, el método reverseIt() construye un StringBuffer con una capacidad inicial igual a la de la cadena fuente, asegurándose sólo una asignación de memoria para dest.

La versión del método append() utilizado en reverseIt() es sólo uno de los métodos de StringBuffer para añadir datos al final de un StringBuffer. Existen varios métodos append() para añadir varios tipos, como float, int, boolean,e incluso objetos, al final del StringBuffer. El dato es convertido a cadena antes de que tenga lugar la operación de adicción.

Insertar Caracteres

Algunas veces, podríamos querer insertar datos en el medio de un StringBuffer. Se puede hacer esto utilizando el método insert(). Este ejemplo ilustra cómo insertar una cadena dentro de un StringBuffer.

```
StringBuffer sb = new StringBuffer("Bebe Caliente!");
sb.insert(6, "Java ");
System.out.println(sb.toString());
```

Este retazo de código imprimerá.

```
Bebe Java Caliente!
```

Con muchos métodos insert() de StringBuffer se puede especificar el índice anterior donde se quiere insertar el dato. En el ejemplo anterior: "Java " tiene que insertarse antes de la 'C' de "Caliente". Los índices empiezan en 0, por eso el índice de la 'C' es el 6. Para insertar datos al principio de un StringBuffer se utiliza el índice 0. Para añadir datos al final del StringBuffer se utiliza un índice con la longitud actual del StringBuffer o append().

Seleccionar Caracteres

Otro modificador muy útil de StringBuffer es setCharAt(), que selecciona un carácter en la posición especificada del StringBuffer. setCharAt() es útil cuando se reutiliza un StringBuffer.

En esta página:

Convertir Objetos a Strings

```
class ReverseString {
    public static String reverseIt(String source) {
        int i, len = source.length();
        StringBuffer dest = new StringBuffer(len);

        for (i = (len - 1); i >= 0; i--) {
            dest.append(source.charAt(i));
        }
        return dest.toString();
    }
}
```

El Método toString()

A veces es conveniente o necesario convertir un objeto a una cadena o String porque se necesitará pasarlo a un método que sólo acepta Strings. Por ejemplo, System.out.println() no acepta StringBuffers, por lo que necesita convertir el StringBuffer a String para poder imprimirlo. El método reverseIt() utiliza el método toString() de StringBuffer para convertirlo en un String antes de retornar.

```
return dest.toString();
```

Todas las clases heredan toString() desde la clase Object y muchas clases del paquete java.lang sobreescriben este método para proporcionar una implementación más acorde con la propia clase. Por ejemplo, las clases Character, Integer, Boolean, etc.. sobreescriben toString() para proporcionar una representación en String de los objetos.

El Método valueOf()

Como es conveniente, la clase String proporciona un método estático valueOf(). Se puede utilizar este método para convertir variables de diferentes tipos a un String. Por ejemplo, para imprimir el número pi.

```
System.out.println(String.valueOf(Math.PI));
```

Convertir Cadenas a Números

La clase String no porporciona ningún método para convertir una cadena en un número. Sin embargo, cuatro clases de los "tipos envolventes" (Integer, Double, Float, y Long) proporcionan unos métodos de clase llamados valueOf() que convierten una cadena en un objeto de ese tipo. Aquí tenemos un pequeño ejemplo del método valueOf() de la clase Float.

```
String piStr = "3.14159";
Float pi = Float.valueOf(piStr);
```

TutorJava Nivel Básico

En esta página:

Los Strings y el Compilador Java

El compilador de Java utiliza las clases String y StringBuffer detrás de la escena para manejar las cadenas literales y la concatenación.

Cadenas Literales

En Java se deben especificar las cadenas literales entre comillas.

```
"Hola Mundo!"
```

Se pueden utilizar cadenas literales en cualquier lugar donde se pueda utilizar un objeto String. Por ejemplo, System.out.println() acepta un argumenteo String, por eso se puede utilizar una cadena literal en su lugar.

```
System.out.println("Hola Mundo!");
```

También se pueden utilizar los métodos de String directamente desde una cadena literal.

```
int len = "Adios Mundo Cruel".length();
```

Como el compilador crea automáticamente un nuevo objeto String para cada cadena literal que se encuentra, se puede utilizar una cadena literal para inicializar un String.

```
String s = "Hola Mundo";
```

El constructor anterior es equivalente pero mucho más eficiente que este otro, que crea dos objetos String en vez de sólo uno.

```
String s = new String("Hola Mundo");
```

El compilador crea la primera cadena cuando encuentra el literal "Hola Mundo!", y la segunda cuando encuentra new String().

Concatenación y el Operador +

En Java, se puede utilizar el operador + para unir o concatenar cadenas.

```
String cat = "cat";
System.out.println("con" + cat + "enacion");
```

Esto decepciona un poco porque, como ya se sabe, los Strings no puden modificarse. Sin embargo detrás de la escena el compilador utiliza StringBuffer para implementar la concatenación. El ejemplo anterior se compilaría de la siguiente forma.

```
String cat = "cat";
System.out.println(new StringBuffer().append("con").append(cat).append("enation"));
```

También se puede utilizar el operador + para añadir valores a una cadena que no son propiamente cadenas.

```
System.out.println("Java's Number " + 1);
```

El compilador convierte el valor no-cadena (el entero 1 en el ejemplo anterior) a un objeto String antes de realizar la concatenación.

En esta página:

Cambios en el JDK 1.1: La Clase String

La versión del JDK 1.1 añade muchas características para hacer algo más sencillo a los programadores el desarrollo de programas intenacionalizados. La clase String necesitaba algunos cambios para ser un mejor ciudadano internacional. De echo, todos los cambios realizados en la clase String están relacionados con la internacionalizacion.

Ten cuidado: Algunos usos de las clases String y StringBuffer podrían comprometer la "Cualidad Internacional" de tu programa.

Métodos Obsoletos

La primera columna de la siguiente tabla lista los contructores y métodos de la clase String que se han queda obsoletos en el JDK 1.1. La segunda columna lista las alternativas a estos constructores y métodos.

Métodos Obsoletos	Alternativas
String(byte[], int)	String(byte[]) o String(byte[], String)
String(byte[], int, int, int)	String(byte[], int, int) o String(byte[], int, int, String)
getBytes(int, int, byte[], int)	byte[] getBytes(String) o byte[] getBytes()

Estos métodos han caducado porque no convierten de forma apropiada los bytes en caracteres. Los nuevos constructores y métodos utilizan una codificación de caracteres indicada o una por defecto para hacer la conversión.

Nuevos métodos

Estos constructores y métodos se han añadido a la clase String en el JDK 1.1.

```
String(byte[])
String(byte[], int, int)
String(byte[], String)
String(byte[], int, int, String)
byte[] getBytes(String)
byte[] getBytes()
String toLowerCase(Locale)
String toUpperCase(Locale)
```

Los cuatro nuevos constructores y los dos núevos métodos getBytes se describen en la sección anterior. Los otros dos métodos, toLowerCase y toUpperCase convierten el String a mayúsculas o minúsculas de acuerdo con las reglas de la especificación Local.

En esta página:

Seleccionar Atributos del Programa

Los programas Java se ejecutan dentro de algún entorno. Esto es, en un entorno donde hay una máquina, un directorio actual , preferencias del usuario, el color de la ventana, la fuente, el tamaño de la fuente y otros atributos ambientales. Además de estos atributos del sistema, un programa puede activar ciertos atributos configurables específicos del programa. Los atributos del programa son frecuentemente llamados preferencias y permiten al usuario configurar varias opciones de arranque.

Un programa podría necesitar información sobre el entorno del sistema para tomar decisiones sobre algo o como hacerlo. Un programa también podría modificar ciertos atributos propios o permitir que el usuario los cambie. Por eso, un programa necesita poder leer y algunas veces modificar varios atributos del sistema y atributos específicos del programa. Los programas Java puede manejar atributos del programa a través de tres mecanismos: propiedades, argumentos de la línea de comandos de la aplicación y parámetros de applets.

TutorJava Nivel Básico

En esta página:

Seleccionar y Utilizar Propiedades

En Java, los atributos del programa están representados por la clase Properties del paquete java.util. Un objeto Propierties contiene un juego de parejas clave/valor. Estas parejas clave/valor son como las entradas de un diccionario: la clave es la palabra, y el valor es la definición.

Tanto la clave como el valor son cadenas. Por ejemplo, os.name es la clave para una de las propiedades del sistema por defecto de Java--el valor contiene el nombre del sistema operativo actual. Utilice la clave para buscar una propiedad en la lista de propiedades y obtener su valor. En mi sistema, cuando busco la propiedad os.name, su valor Windows 95/NT. El tuyo probablemente será diferente.

Las propiedades específicas de un programa deben ser mantenidas por el propio programa. Las propiedades del sistema las mantiene la clase java.lang.System. Para más información sobre las propiedades del sistema puedes referirte a: Propiedades del Sistema en la lección Utilizar los Recursos del Sistema.

Se puede utilizar la clase Propierties del paquete java.util para manejar atributos específicos de un programa. Se puede cargar los pares clave/valor dentro de un objeto Propierties utilizando un stream, grabar las propiedades a un stream y obtener información sobre las propiedades representadas por el objeto Propierties.

Seleccionar un Objeto Properties

Frecuentemente, cuando un programa arranca, utiliza un código similar a este para seleccionar un objeto Propierties.

```
. . .
    // selecciona las propiedades por defecto
Properties defaultProps = new Properties();
FileInputStream defaultStream = new FileInputStream("defaultProperties");
defaultProps.load(defaultStream);
defaultsStream.close();

    // selecciona las propiedades reales
Properties applicationProps = new Properties(defaultProps);
FileInputStream appStream = new FileInputStream("appProperties");
applicationProps.load(appStream);
appStream.close();
. . .
```

Primero la aplicación selecciona un objeto Properties para las propiedades por defecto. Este objeto contiene un conjunto de propiedades cuyos valores no se utilizan explícitamente en ninguna parte. Este fragmenteo de código utiliza el método load() para leer el valor por defecto desde un fichero de disco llamado defaultProperties. Normalmente las aplicaciones guardan sus propiedades en ficheros de disco.

Luego, la aplicación utiliza un constructor diferente para crear un segundo objeto Properties applicationProps. Este objeto utiliza defaultProps para proporcionarle sus valores por defecto.

Después el código carga un juego de propiedades dentro de applicationProps desde un fichero llamado appProperties. Las propiedades cargadas en appProperties pueden seleccionarse en

base al usuairo o en base al sistema, lo que sea más apropiado para cada programa. Lo que es importante es que el programa guarde las Propiedades en un posición "CONOCIDA" para que la próxima llamada al programa pueda recuperarlas. Por ejemplo, el navegador HotJava graba las propiedades en el directorio raiz del usuario.

Se utiliza el método save() para escribir las propiedades en un canal.

```
FileOutputStream defaultsOut = new FileOutputStream("defaultProperties");
applicationProps.save(defaultsOut, "---No Comment---");
defaultsOut.close();
```

El método save() necesita un stream donde escribir, y una cadena que se utiliza como comentario al principio de la salida.

Obtener Información de las Propiedades

Una vez que se han seleccionado las propiedades de un pograma, se puede pedir información sobre la propiedades contenidas. La clase Properties proporciona varios métodos para obtener esta información de las propiedades.

getProperty() (2 versiones)

Devuelve el valor de la propiedad especificada. Una versión permite un valor por defecto, si no se encuentra la clase, se devuelve el valor por defecto.

list()

Escribe todas las propiedades en el canal especificado. Esto es útil para depuración.

propertyNames()

Devuelve una lista con todas las claves contenidas en el objeto Properties.

Consideraciones de Seguridad:

Observa que el acceso a las propiedades está sujeto a la aprobación del manejador de Seguridad. El programa de ejemplo es una aplicación solitaria, que por defecto, no tiene manejador de seguridad. Si se intenta utilizar este código en un applet, podría no trabajar dependiendo del navegador. Puedes ver: Entender las Capacidades y Restricciones de un Applet para obtener más información sobre las restricciones de seguridad en los applets.

TutorJava Nivel Básico

En esta página:

Argumentos de la Línea de Comandos

Una aplicación Java puede aceptar cualquier número de argumentos desde la línea de comandos. Los argumentos de la línea de comandos permiten al usuario variar la operación de una aplicación, Por ejemplo, una aplicación podría permitir que el usuario especificara un modo verboso--esto es, especificar que la aplicación muestre toda la información posible-- con el argumento -verbose.

Cuando llama a una aplicación, el usuario teclea los argumentos de la línea de comandos después del nombre de la aplicación. Supongamos, por ejemplo, que existe una aplicación Java, llamada Sort, que ordena las líneas de un fichero, y que los datos que se quiere ordenar están en un fichero llamado friends.txt. Si estuvieramos utilizando Windows 95/NT, llamaría a la aplicación Sort con su fichero de datos de la siguiente forma.

```
C:\> java Sort friends.txt
```

En el lenguaje Java, cuando se llama a una aplicación, el sistema de ejecución pasa los argumentos de la línea de comandos al método main de la aplicación, mediante un array de Strings. Cada String del array contiene un argumento. En el ejemplo anterior, los argumentos de la línea de comandos de la aplicación Sort son un array con una sola cadena que contiene "friends.txt".

Ejemplo de Argumentos

Esta sencilla aplicación muestra todos los argumentos de la línea de comandos uno por uno en cada línea.

```
class Echo {
    public static void main (String[] args) {
        for (int i = 0; i < args.length; i++)
            System.out.println(args[i]);
    }
}
```

Intenta Esto: Llama a la aplicación Echo. Aquí tiened un ejemplo de como llamarla utilizando Windows 95/NT.

```
C:\> java Echo Bebe Java Caliente
Bebe
Java
Caliente
```

Habrás observado que la aplicación muestra cada palabra en una línea distinta. Esto es así porque el espacio separa los argumentos de la línea de comandos. Si quieres que Bebe Java Caliente sea interpretado como un sólo argumento debes ponerlo entre comillas.

```
% java Echo "Bebe Java Caliente"
Bebe Java Caliente
```

Convenciones

Existen varias convenciones que se deberán observar cuando se acepten y procesen argumentos de la línea de comandos con una aplicación Java.

Analizar Argumentos de la Línea de Comandos

La mayoría de las aplicaciones aceptan varios argumentos de la línea de comandos que le permiten al usuario variar la ejecución de la aplicación, Por ejemplo, el comando UNIX que imprime el contenido de un directorio-- ls --accepta argumentos que determinan qué atributos de ficheros se van a mostrar y el orden en que lo van a hacer. Normalmente, el usuairo puede especificar los argumentos en cualquier orden por lo tanto requiere que la aplicación sea capaz de analizarlos.

En esta página:

Convenciones para los Argumentos de la Línea de Comandos

El lenguaje Java sigue las convenciones de UNIX que definen tres tipos diferentes de argumentos.

- Palabras (también conocidos como opciones)
- Argumentos que requieren argumentos
- Banderas

Además, una aplicación debe observar las siguientes convenciones para utilizar los argumentos de la línea de comandos en Java.

- El guión (-) precede a las opciones, banderas o series de banderas.
- Los argumentos pueden ir en cualquier orden, excepto cuando sea un argumento que requiere otros argumentos.
- Las banderas pueden listarse en cualquier orden, separadamente o combinadas.

  -xn o -nx o -x -n.
- Típicamente los nombres de fichero van al final.
- El programa imprime un mensaje de error de utilización cuando no se reconoce un argumento de la línea de comandos. Estas sentencias pueden tener esta forma.

  ```
  utilización: nombre_aplicación [ argumentos_opcionales ] argumentos_requeridos
  ```

Opciones

Los argumentos como -verbose son argumentos de palabra y deben especificarse completamente en la línea de comandos. Por ejemplo, -ver no correspondería con -verbose.

Se pueden utilizar sentencias como esta para comprobar los argumentos de palabras.

```
if (argument.equals("-verbose"))
    vflag = true;
```

Esta sentencia comprueba si la palabra -verbose está en la línea de argumentos y activa una bandera en el programa para que este se ejecute en modo verboso.

Argumentos que Requieren Argumentos

Algunos argumentos necesitan más informaión. Por ejemplo, un argumento como -output podría permitir que el usuario redirigiera la salida del programa. Sin embargo, la opción -output en solitario no ofrece la información suficiente a la aplicación: ¿Cómo sabe la aplicación dónde redirigir la salida? Por lo tanto el usuario debe especificar también un nombre de fichero. Normalmente, el ítem siguiente de la línea de comandos proporciona la información adicional para el argumento que así lo requiere. Se puede utilizar la siguiente sentencia para emparejar argumentos que requieren argumentos.

```
if (argument.equals("-output")) {
    if (nextarg < args.length)
        outputfile = args[nextarg++];
    else
        System.err.println("-output requiere un nombre de fichero");
}
```

Observa que el código se asegura de que el usuario ha especificado realmente un argumento siguiente antes de intentar utilizarlo.

Banderas

Lss banderas son carácteres que modifican el comportamieneo del programa de alguna manera. Por ejemplo, la

bandera -t proporcionada al comando ls de UNIX indica que la salida debe ordenarse por la fecha de los ficheros. La mayoría de las aplicaciones permiten al usuario especificar banderas separadas en cualquier orden.

`-x -n` **o** `-n -x`

Además, para hacer un uso más sencillo, las aplicciones deberán permitir que el usuario concatene banderas y las especifique en cualquier orden.

`-nx` **o** `-xn`

El programa de ejemplo descrito en la página siguiente implementa un sencillo algoritmo para procesar banderas que pueden especificarse en cualquier orden, separadas o de forma combinada.

En esta página:

Analizar Argumentos de la Línea de Comandos

Este programa, llamado ParseCmdLine, proporciona una forma básica para construir tu propio analizador de argumentos.

```
class ParseCmdLine {
    public static void main(String[] args) {

        int i = 0, j;
        String arg;
        char flag;
        boolean vflag = false;
        String ficheroSalida = "";

        while (i < args.length && args[i].startsWith("-")) {
            arg = args[i++];

    // Utiliza este tipo de chequeo para argumentos de "palabra"
            if (arg.equals("-verboso")) {
                System.out.println("modo verboso mode activado");
                vflag = true;
            }

    // Utiliza este tipo de chequeo para argumentos que requieren argumentos
            else if (arg.equals("-output")) {
                if (i < args.length)
                    ficheroSalida = args[i++];
                else
                    System.err.println("-output requiere un nombre de fichero");
                if (vflag)
                    System.out.println("Fichero de Salida = " + ficheroSalida);
            }

    // Utiliza este tipo de chequeo para una serie de banderas
            else {
                for (j = 1; j < arg.length(); j++) {
                    flag = arg.charAt(j);
                    switch (flag) {
                    case 'x'.
                        if (vflag) System.out.println("Opción x");
                        break;
                    case 'n'.
                        if (vflag) System.out.println("Opción n");
                        break;
                    default.
                        System.err.println("ParseCmdLine: opción ilegal " + flag);
                        break;
                    }
                }
            }
        }
        if (i == args.length)
            System.err.println("Utilización: ParseCmdLine [-verboso] [-xn]
                        [-output unfichero] nombre de Fichero");
```

```
        else
            System.out.println("Correcto!");
    }
}
```

Acepta un argumento de cada uno de los tipos: un argumento de palabra, un argumento que requiere un argumento y dos banderas. Además, este programa necesita un nombre de fichero. Aquí tienes una sentencia de utilización de este programa.

```
Utilización: ParseCmdLine [-verboso] [-xn] [-output unfichero] nombrefichero
```

Los argumentos entre los corchetes son opciones: el argumento nombrefichero es obligatorio.

En esta página:

Cambios en el JDK 1.1: La clase Properties

Nuevos métodos

Se han añadido los siguientes métodos a la clase Properties.

`list(PrintWriter)`

El nuevo método list escribe su salida en un PrintWriter, una nueva clase introducida en el paquete java.io del JDK 1.1. Este nuevo método es similar al otro método list que escribia en un PrintStream.

Las propiedades se utilizan extensivamente en programas internacionalizados para manejar Datos sensitivos a la localización

En esta página:

Utilizar los Recursos del Sistema

Algunas veces, un programa necesita acceder a los recursos del sistema, como los canales de I/O estandard o la hora del sistema. Un programa podría utilizar estos recursos directamente desde el entorno de ejecución, pero sólo podría ejecutarse en el entorno donde fue escrito. Cada vez que se quisiera ejecutar un programa en un nuevo entorno se deberá "portar" el programa reescribiendo las secciones del código dependientes del sistema.

El entorno de desarrollo de Java resuelve este problema permitiendo a los programas que utilicen los recursos del sistema a través de un interface de programación independiente del sistema implementado por la clase System (miembro del paquete java.lang).

En esta página:

Utilizar la Clase System

Al contrario que la mayoría de las clases, no se debe ejemplarizar la clase System para utilizarla. Para ser más precisos, no se puede ejemplarizar-- es una clase final y todos sus constructores son privados.

Todas las variables y métodos de la clase System son métodos y variables de clase -- están declaradas como static. Para una completa explicación sobre las variables y métodos de clase y en qué se diferencian de las variables y métodos de ejemplar, puede referirse a Miembros del Ejemplar y de la Clase.

Para utilizar una variable de clase, se usa directamente desde el nombre de la clase utilizando la notación de punto ('.') de Java. Por ejemplo, para referirse a la variables out de la clase System, se añade el nombre de la variable al nombre de la clase separados por un punto. Así.

```
System.out
```

Se puede llamar a los métodos de clase de una forma similar. Por ejemplo, para llamar al método getProperty() de la clase System se añade el nombre del método al nombre de la clase separados por un punto.

```
System.getProperty(argument);
```

El siguente programa Java utiliza dos veces la clase System, primero para obtener el nombre del usuario actual y luego para mostrarlo.

```
class UserNameTest {
    public static void main(String[] args) {
        String name;
        name = System.getProperty("user.name");
        System.out.println(name);
    }
}
```

Habrás observado que el programa nunca ejemplariza un objeto de la clase System. Solo referencia al método getProperty() y la variable out directamente desde la clase.

El ejemplo anterior utiliza el método getProperty() para buscar en la base de datos de propiedades una propiedad llamada "user.name". Propiedades del Sistema más adelante en esta lección cuenta más cosas sobre las propiedades del sistema y el método getProperty().

El ejemplo también utliza System.out, un PrintStream que implementa el canal de salida estandard. El método println() imprime el argumento en el canal de salida estandard. La siguiente página de está lección explica el canal de salida estandard y los otros dos canales proporcionados por la clase System.

En esta página:

Los Canales de I/O Estándar

Los conceptos de los canales de I/O estandard son un concepto de la librería C que ha sido asimilado dentro del entorno Java. Existen tres canales estandard, todos los cuales son manejados por la clase java.lang.System.

Entrada estandard --referenciado por **System.in**

utilizado para la entrada del programa, típicamente lee la entrada introducida por el usuario.

Salida estandard --referenciado por **System.out**

utilizado para la salida del programa, típicamente muestra información al usuario.

Error estandard --referenciado por **System.err**

utilizado para mostrar mensajes de error al usuario.

Canal de Entrada Estandard

La clase System proporciona un canal para leer texto -- el canal de entrada estandard. El programa de ejemplo de Tuercas y Tornillos del Lenguaje Java utiliza el canal del entrada estandard para contar los caracteres tecleados por el usuario.

Los Canales de Salida y de Error Estandards

Probablemente los puntos más utilizados de la clase System sean los canales de salida y de error estandard, que se utilizan para mostrarle texto al usuario.

El canal de salida estandard se utiliza normalmente para las salidas de comandos, esto es, para mostrar el resultado de un comando del usuario. El canal de error estandard se utiliza para mostrar cualquier error que ocurra durante la ejecución del programa.

Los métodos print(), println(), y write()

Tanto la salida estandard como el error estandard derivan de la clase PrintStream. Así, se utiliza uno de los tres métodos de PrintStream para imprimir el texto en el canal: print(), println(), y write().

Los métodos print() y println() son esencialmente el mismo: los dos escriben su argumento String en el canal. La única diferencia entre estos dos métodos es que println() añade un carácter de nueva línea al final de la salida, y print() no lo hace. En otras palabras, esto

```
System.out.print("Duke no es un Pingüino!\n");
```

es equivalente a esto

```
System.out.println("Duke no es un Pingüino!");
```

Observa el carácter extra \n en la llamada al primer método; es el carácter para nueva línea. println() añade automáticamente este carácter a su salida.

El método write() es menos utilizado que los anteriores y se utiliza para escribir bytes en un canal. write() se utiliza para escribir datos que no sean ASCII.

Argumentos para print() y println()

Los métodos print() y println() toman un sólo argumento. Este argumento puede ser cualquiera de los siguientes tipos: Object, String, char[], int, long, float, double, y boolean. Además existe una versión extra de println() que no tiene argumentos que imprime una nueva línea en el canal.

Imprimir Objetos de Diferentes Tipos de Datos

El siguiente programa utiliza println() para sacar datos de varios tipos por el canal de salida estandard.

```
class DataTypePrintTest {
    public static void main(String[] args) {

        Thread objectData = new Thread();
        String stringData = "Java Mania";
        char[] charArrayData = { 'a', 'b', 'c' };
        int integerData = 4;
        long longData = Long.MIN_VALUE;
        float floatData = Float.MAX_VALUE;
        double doubleData = Math.PI;
        boolean booleanData = true;

        System.out.println(objectData);
        System.out.println(stringData);
        System.out.println(charArrayData);
        System.out.println(integerData);
        System.out.println(longData);
        System.out.println(floatData);
        System.out.println(doubleData);
        System.out.println(booleanData);
    }
}
```

El programa anterior tendría esta salida.

```
Thread[Thread-4,5,main]
Java Mania
abc
4
-9223372036854775808
3.40282e+38
3.14159
true
```

Observa que se puede imprimir un objeto -- el primer println() imprime un objeto Thread y el segundo imprime un objeto String. Caundo se utilice print() o println() para imprimir un objeto, los datos impresos dependen del tipo del objeto. En el ejemplo, imprimir un String muestra el contenido de la cadena. Sin embargo, imprimir un Thread muestra una cadena con este formato.

```
ThreadClass[nombre,prioridad,grupo]
```

En esta página:

Propiedades del Sistema

La clase System mantiene un conjunto de propiedades -- parejas clave/valor -- que definen atributos del entorno de trabajo actual. Cuando arranca el sistema de ejecución por priemra vez, las propiedades del sistema se inicializan para contener información sobre el entorno de ejecución. Incluyendo información sobre el usuario actual, la versión actual del runtime de Java, e incluso el carácter utilizado para separar componentes en un nombre de fichero.

Aquí tiene una lista completa de las propiedades del sistema que puede obtener el sistema cuando arranca y lo que significan.

Clave	Significado	Acceden los Applets
"file.separator"	File separator (e.g., "/")	si
"java.class.path"	Java classpath	no
"java.class.version"	Java class version number	si
"java.home"	Java installation directory	no
"java.vendor"	Java vendor-specific string	si
"java.vendor.url"	Java vendor URL	si
"java.version"	Java version number	si
"line.separator"	Line separator	si
"os.arch"	Operating system architecture	si
"os.name"	Operating system name	si
"path.separator"	Path separator (e.g., ":")	si
"user.dir"	User's current working directory	no
"user.home"	User home directory	no
"user.name"	User account name	no

Los programas Java pueder leer o escribir las propiedades del sistema a través de varios métodos de la clase System. Se puede utilizar una clave para buscar una propiedad en la lista de propiedades, o se puede obtener el conjunto completo de propiedades de una vez. También se puede cambiar el conjunto de propiedades completamente.

Consideraciones de Seguridad:

Los Applets pueden acceder a las propiedades del sistema pero no a todas. Para obtener una lista completa de las propieades del sistema que pueden y no pueden ser utilizadas por los applets, puedes ver :Leer las Propiedades del Sistema. Los applets no pueden escribir las propiedades del sistema.

Leer las Propiedades del Sistema

La clase System tiene dos métodos que se pueden utilizar para leer las propiedades del sistema: getProperty() y getProperties.

La clase System tiene dos versiones diferentes de getProperty(). Ambas versiones devuelven el valor de la propiedad nombrada en la lista de argumentos. La más simple de las dos getProperty() toma un sólo argumento: la clave de la propiedad que quiere buscar. Por ejemplo, para obtener el valor de path.separator, utilizamos la siguiente sentencia.

```
System.getProperty("path.separator");
```

Este método devuelve una cadena que contiene el valor de la propiedad. Si la propiedad no existe, esta versión de getProperty() devuelve null.

Lo que nos lleva a la siguiente versión de getProperty(). Esta versión requiere dos argumentos String: el primer argumento es la clave que buscamos y el segundo es el valor por defecto devuelto si la clave no se encuentra o no tiene ningún valor. Por ejemplo, esta llamada a getProperty() busca la propiedad del sistema llamada subliminal.message.

Esto no es una propiedad válida del sistema, por lo que en lugar de devolver null, este método devolverá el valor proporcionado por el segundo argumento: "Compra Java ahora".

```
System.getProperty("subliminal.message", "Compra Java ahora!");
```

Se deberá utilizar esta versión de getProperty() si no se quiere correr el riesgo de una excepción NullPointerException, o si realmente se quiere proporcionar un valor por defecto para una propiedad que no tiene valor o que no ha podido ser encontrada.

El último método proporcionado por la clase System para accer a los valores de las propiedades es el método getProperties() que devuelve Propiedad un objeto que contiene el conjunto completo de las propiedades del sistema. Se pueden utilizar varios métodos de la clase Propierties para consultar valores específicos o para listar el conjunto completo de propiedades. Para más información sobre la clase Propierties, puedes ver Seleccionar y utilizar Propiedades.

Escribir Propiedades del Sistema

Se puede modificar el conjunto existente de las propiedades del sistema, utilizando el método setProperties() de la clase System. Este método toma un objeto Propierties que ha sido inicializado para contener parejas de clave/valor para las propiedades que se quieren modificar. Este método reemplaza el conjunto completo de las propiedades del sistema por las nuevas representadas por el objeto Propierties.

Aquí tenemos un pequeño programa de ejemplo que crea un objeto Propierties y lo inicializa desde un fichero.

```
subliminal.message=Buy Java Now!
```

El programa de ejemplo utiliza System.setProperties() para instalar el nuevo objeto Propierties como el conjunto actual de propiedades del sistema.

```
import java.io.FileInputStream;
import java.util.Properties;

class PropertiesTest {
    public static void main(String[] args) {
        try {
                // selecciona el nuevo objeto propierties a partir de
"myProperties.txt"
            FileInputStream propFile = new FileInputStream("myProperties.txt");
            Properties p = new Properties(System.getProperties());
            p.load(propFile);

                // selecciona las propiedades del sistema
            System.setProperties(p);
            System.getProperties().list(System.out);    // selecciona las nuevas
propiedades
        } catch (java.io.FileNotFoundException e) {
            System.err.println("Can't find myProperties.txt.");
        } catch (java.io.IOException e) {
            System.err.println("I/O failed.");
        }
    }
}
```

Observa que el programa de ejemplo crea un objeto Properties, p, que se utiliza como argumento para setProperties().

```
Properties p = new Properties(System.getProperties());
```

Esta sentencia inicializa el nuevo objeto Propierties, p con el conjunto actual de propiedades del sistema, que en el caso de este pequeño programa es el juego de propiedades inicializado por el sistema de ejecución. Luego el programa carga las propiedades adicionales en p desde el fichero myProperties.txt y selecciona las propieades del sistema en p. Esto tiene el efecto de añadir las propiedades listadas en myProperties.txt al juego de propiedades creado por el sistema de ejecución durante el arranque. Observa que se puede crear p sin ningún objeto Properties como este.

```
Properties p = new Properties();
```

Si se hace esto la aplicación no tendrá acceso a las propieades del sistema.

Observa también que las propiedades del sistema se pueden sobreescribir! Por ejemplo, si myProperties.txt contiene la siguiente línea, la propiedad del sistema java.vendor será sobreescrita.

```
java.vendor=Acme Software Company
```

En general, ten cuidado de no sobreescribir las propiedades del sistema.

El método setProperties() cambia el conjunto de las propiedades del sistema para la aplicación que se está ejecutando. Estos cambios no son persistentes. Esto es, cambiar las propiedades del sistema dentro de una aplicación no tendrá ningún efecto en próximas llamadas al intérprete Java para esta u otras aplicaciones. El sistema de ejecución re-inicializa las propiedades del sistema cada vez que arranca. Si se quiere que los cambios en las propiedades del sistema sean persistentes, se deben escribir los valores en un fichero antes de salir y leerlos de nuevo cuando arranque la aplicación.

En esta página:

Forzar la Finalización y la Recolección de Basura

El sistema de ejecución de Java realiza las tareas de manejo de memoria por tí. Cuando un programa ha terminado de utilizar un objeto-- esto es, cuando ya no hay más referencias a ese objeto- el objeto es finalizado y luego se recoge la basura.

Estas tareas suceden asíncronamente en segundo plano. Sin embargo, se puede forzar la finalización de un objeto y la recolección de basura utilizando los métodos apropiados de la clase System.

Finalizar Objetos

Antes de recolectar la basura de un objeto, el sistema de ejecución de Java le da la oportunidad de limpiarse a sí mismo. Este paso es conocido como finalización y se consigue mediante una llamada al método finalize() del objeto. El objeto debe sobreescribir este método para realizar cualquier tarea de limpieza final como la liberación de recursos del sistema como ficheros o conexiones. Para más informaciónsobre el método finalize() puedes ver: Escribir un método finalize().

Se puede forzar que ocurra la finalización de un objeto llamando al método runFinalization() de la clase System.

```
System.runFinalization();
```

Este método llama a los métodos finalize() de todos los objetos que están esperando para ser recolectados.

Ejecutar el Recolector de Basura

Se le puede pedir al recolector de basura que se ejecute en cualquier momento llamando al método gc() de la clase System.

```
System.gc();
```

Se podría querer ejecutar el recolector de basura para asegurarnos que lo hace en el mejor momento para el programa en lugar de hacerlo cuando le sea más conveniente al sistema de ejecución.

Por ejemplo, un programa podría desear ejecutar el recolector de basura antes de entrar en un cálculo o una sección de utilización de memoria extensiva, o cuando sepa que va a estar ocupado algun tiempo. El recolector de basura requiere unos 20 milisegundos para realizar su tarea, por eso un programa sólo debe ejecutarlo cuando no tenga ningún impacto en su programa -- esto es, que el programa anticipe que el recolector de basura va a tener tiempo suficiente para terminar su trabajo.

En esta página:

Otros Métodos de la Clase System

La clase System contiene otros métodos que proporcionan varias funcionalidades incluyendo la copia de arrays y la obtención de la hora actual.

Copiar Arrays

El método arraycopy() de la clase System se utiliza para realizar una copia eficiente de los datos de un array a otro. Este método requiere cinco argumentos.

```
public static
    void copiaarray(Object fuente, int indiceFuente, Object destino, int
indiceDestino, int longitud)
```

Los dos argumentos del tipo Object indican los array de origen y de destino. Los tres argumentos enteros indican la posición en los array de origen y destino y el número de elementos a copiar.

El siguiente programa utiliza copiaarray() para copiar algunos elementos desde copiaDesde a copiaA.

```
class Prueba {
    public static void main(String[] args) {
        byte[] copiaDesde = { 'd', 'e','s', 'c', 'a', 'f', 'e', 'i', 'n', 'a', 'd',
'o' };
        byte[] copiaA = new byte[7];

        System.copiaarray(copiaDesde, 3, copiaA, 0, 7);
        System.out.println(new String(copiaA, 0));
    }
}
```

Con la llamada al método copiaarray() en este programa de ejemplo comienza la copia en el elemento número 3 del array fuente -- recuerda que los índices de los arrays empiezan en cero, por eso la copia empieza en el elemento 'c'. Luego copiaarray() pone los elementos copiados en la posición 0 del array destino copiaA. Copia 7 elementos: 'c', 'a', 'f', 'e', 'i', 'n', y 'a'. Efectivamente, el método copiaarray() saca "cafeina" de "descafeinado".

Observa que el array de destino debe ser asignado antes de llamar a arraycopy() y debe ser lo suficientemente largo para contener los datos copiados.

Obtener la Hora Actual

El método currentTimeMillis() devuelve la hora actual en milisegundos desde las 00:00:00 del 1 de Enero de 1970. Este método se utiliza comunmente en pruebas de rendimiento; obtener la hora actual, realizar la operación que se quiere controlar, obtener de nuevo la hora actual--la diferencia entre las dos horas es el tiempo que ha tardado en realizarse la operación.

En interfaces gráficos de usuarios el tiempo entre dos pulsaciones del ratón se utiliza para determinar su el usuario ha realizado un doble click. El siguiente applet utiliza currentTimeMillis() para calcular el número de milisegundos entre los dos clicks del ratón. Si el tiempo entre los dos clicks es menor de 200 milisegundos, los dos clicks de ratón se interpretan como un doble click.

Aquí tienes el código fuente para el applet TimingIsEverything.

```
import java.awt.*;

public class TimingIsEverything extends java.applet.Applet {
```

```
    public long firstClickTime = 0;
    public String displayStr;

    public void init() {
        displayStr = "Ha un Doble Click aquí";
    }
    public void paint(Graphics g) {
        Color fondo = new Color(255, 204, 51);
        g.setColor(fondo);
        g.drawRect(0, 0, size().width-1, size().height-1);
        g.setColor(Color.black);
        g.drawString(displayStr, 40, 30);
    }
    public boolean mouseDown(java.awt.Event evt, int x, int y) {
        long clickTime = System.currentTimeMillis();
        long clickInterval = clickTime - firstClickTime;
        if (clickInterval < 200) {
            displayStr = "Doble Click!! (Intervalo = " + clickInterval + ")";
            firstClickTime = 0;
        } else {
            displayStr = "Un solo Click!!";
            firstClickTime = clickTime;
        }
        repaint();
        return true;
    }
}
```

Se podría utilizar el valor devuelto por este método para calcular la hora y fecha actuales. Sin embargo, encontrarás más conveniente obtener la hora y fecha actuales desde la clase Date del paquete java.util.

Salir del Entorno de Ejecución.

Para salir del intérprete Java, llama al método System.exit(). Debes pasarle un código de salida en un entero y el interprete saldrá con ese código de salida.

```
System.exit(-1);
```

Nota: El método exit() hace salir del intérprete Java, no sólo del programa Java-- ten cuidado cuando utilice está función.

Consideraciones de Seguridad:

La llamada al método exit() está sujeta a las restricciones de seguridad. Por eso dependiendo del navegador donde se esté ejecutando el applet, una llamada a exit() desde un applet podría resultar en una excepción SecurityException.

Selecionar y Obtener el Manejador de Seguridad

El controlador de seguridad es un objeto que refuerza cierta vigilancia de seguridad para una aplicación Java. Se puede seleccionar el controlador de seguridad actual para una aplicación utilizando el método setSecurityManager() de la clase System, y se puede recuperar el controlador de seguridad actual utilizando el método getSecurityManager().

Consideraciones de Seguridad:

El controlador de seguridad de una aplicación sólo puede seleccionarse una vez. Normalmente, un navegador selecciona su controlador de seguridad durante el arranque. Por eso, la mayoría de las veces, los applets no pueden seleccionar el controlador de seguridad porque ya ha sido seleccionado. Si un applet hace esto resultará una SecurityException.

En esta página:

Cambios en el JDK 1.1: Utilizar los Recursos del Sistema

Los Streams de I/O Estándards La clase System soporta tres nuevos métodos que permiten aumentan los streams de I/O estándards. Puedes ver Cambios en el JDK 1.1:La clase System.

Propiedades del Sistema El método getenv ha sido eliminado. Puedes ver Cambios en el JDK 1.1: La clase System.

Finalizar Objectos El nuevo método runFinalizersOnExit, de la clase System permite seleccionar si los finalizadores son llamados cuando tu programa finalice. Puedes ver Cambios en el JDK 1.1: La clase System.

Métodos Misceláneos del Sistema

El ejemplo de copia de Array

ArrayCopyTest.java utiliza un constructor de **String** que ha sido eliminado. Puedes ver Cambios en el JDK 1.1: Ejemplo de copia de Arrays.

El Appet TimingIsEverything

TimingIsEverything.java utiliza un API del AWT que está obsoleto. Puedes ver Cambios en el JDK 1.1: El Applet TimingIsEverything.

En esta página:

Cambios en el JDK 1.1: La clase System

Métodos Obsoletos

La primera columna de la siguiente tabla lista los métoso de la clase System que se han elimnado en el JDK 1.1. La segunda columna lista las altrnativas a estos métodos.

Métodos Eliminados	Alternativa
getenv(String)	getProperty(String)

El método getenv ha estado fuera de uso desde hace varias versiones del JDK. La versión 1.1 formaliza este cambio, eliminado el método. Los programadores deberían utilizar Properties para almacenar información entre invocaciones de un programa. Seleccionar y utilizar Properties explica como utilizar la clase Properties y Manejar datos Sensibles a la Localización muestra cómo utilizar las propiedades para ayudar a la internacionalización de tus programas Java.

Nuevos métodos

Se han añadido los siguientes métodos a la clase System en el JDK 1.1.

```
setErr(PrintStream)
setIn(InputStream)
setOut(PrintStream)
runFinalizersOnExit(boolean)
identityHashb(Object)
```

Los métodos setErr, setIn, y setOut reasignan los canales de error, entrada y salida estándards, respectivamente. Los Streams de I/O Estándard en esta Lección explica estos streams.

runFinalizersOnExit determina si los métodos finalize son invocados o no cuando un programa finaliza. Por defecto esta desactivado Eliminar Objetos no Utilizados explica los métodos finalize y la recolección de basura en más detalle.

En esta página:

Cambios en el JDK 1.1: Ejemplo copia de Arrays

La última línea de código en el ejemplo ArrayCopyTest utiliza un constructor String que crea un objeto String desde un array de bytes. Este constructor no covertía correctamente los bytes en caracteres ha sido eliminado. La clase String proporciona dos constructores alternativos.

```
String(byte[])
   o
String(byte[], String)
```

El primer constructor convierte el array de bytes en caracteres utilizando la codificación de caracteres por defecto. El segundo convierte el array de bytes en caracteres utilizado la codificación de caracteres especificada.

Aquí tienes una nueva versión de este ejemplo utilizando el constructor que utiliza la codificación de caracteres por defecto.

```
class ArrayCopyTest {
    public static void main(String[] args) {
        byte[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't',
'e', 'd' };
        byte[] copyTo = new byte[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```

Para ver más detalles sobre éste y otros cambios en la clase String , Cambios en el JDK 1.1: La Clase String.

En esta página:

Cambios en el JDK 1.1: El Applet TimingIsEverything

El applet TimingIsEverything utiliza un API desfasado. Primero utiliza el viejo mecanismo de manejo de eventos. Segundo utiliza el método size que ha sido eliminado en el JDK 1.1 en fabor del nuevo método getSize.

Hemos escrito una nueva versión del ejemplo TimingIsEverything que toma ventana del nuevo sistema de manejo de eventos y utilizar el nuevo método getSize. Aquí tienes el nuevo applet en acción.

Aquí tienes el código fuente de la versión 1.1 del applet TimingIsEverything.

```
import java.awt.Graphics;
import java.awt.Dimension;

import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class TimingIsEverything1_1 extends java.applet.Applet {

    public long firstClickTime = 0;
    public String displayStr;

    public void init() {
        displayStr = "Double Click Me";
        addMouseListener(new MyAdapter());
    }
    public void paint(Graphics g) {
        g.drawRect(0, 0, getSize().width-1, getSize().height-1);
        g.drawString(displayStr, 40, 30);
    }
    class MyAdapter extends MouseAdapter {
        public void mouseClicked(MouseEvent evt) {
            long clickTime = System.currentTimeMillis();
            long clickInterval = clickTime - firstClickTime;
            if (clickInterval < 200) {
                displayStr = "Double Click!! (Interval = " + clickInterval + ")";
                firstClickTime = 0;
            } else {
                displayStr = "Single Click!!";
                firstClickTime = clickTime;
            }
            repaint();
        }
    }
}
```

Para ver más detalles sobre estos y otros cambios en el AWT puedes ver la página Cambios en el GUI: el AWT Crece.