Conceptos básicos de Programación 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, quien 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.

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 categorías 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 permite 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 una 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.

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 aparecen 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 aritméticas. 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 aritméticos - 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 < index && index < NUM_ENTRIES
```

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

```
((count > 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 manipulación 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 >> 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 aritméticas, lógicas o de bits y una operación de asignación al mismo tiempo. Específicamente, supón 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 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

Expresiones Java

Definición 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.

También 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 paréntesis 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	= += -= *= /= %= ^= &= |= <<= >>= >>>=

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.

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;
```

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.

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.

Hasta entonces, los ejemplos contenidos aquí crean objetos a partir 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 responsables 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 recién 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 proporcionan métodos constructores para inicializar 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.

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.

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 vacíos.

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

Veamos qué 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 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 coordenadas x e y, y devuelto **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 rectángulo.

```
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()**.

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, permitiendo 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 utilizar 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 asíncronamente cuando el sistema está ocupado. Tan pronto como otro thread se vuelva activo, se pedirá 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.

Declarar Clases Java

Ahora que ya sabemos cómo 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 específicamente, 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 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..

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 cláusula **implements** sigue a la cláusula **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.

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.

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.

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.

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

De todos estos ítems, 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 abstracta y que no implementa interfaces).

Controlar el Acceso a los Miembros de la Clase

Uno de los beneficios 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 compañía.

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 permitidos 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()**.

Constructores

Todas las clases Java tienen métodos especiales llamados Constructores que se utilizan para inicializar un objeto nuevo de ese tipo. Los constructores 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 por qué 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.

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.

Crear Subclases

Se declara que una 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 cláusula **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 característica 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 sobrescribe 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.

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 Sobreescibir

- 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.

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 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, inconsistentes 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 quisiéramos 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 creación 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) {
    }
    . . .
}
```

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 proporciona 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 substanciales: 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 cómo 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.