Centro Asociado Palma de Mallorca

Introducción Práctica de Programación Java

Antonio Rivero Cuesta

Sesión II

La Sintaxis de Java I

Tipos de Datos

Tipos de Datos Simples

Tipo	Descripción	Long	Rango
byte	byte	1 byte	−128 a 127
short	Entero corto	2 bytes	−32768 a 32767
int	Entero	4 bytes	-2^{31} a $2^{31}-1$
long	Entero largo	8 bytes	-2^{63} a $2^{63}-1$
float	Real Coma Flotante	32 bytes	$\pm 3{,}4 \cdot 10^{38}$ a $\pm 1{,}4 \cdot 10^{-38}$
double	Real Coma Flotante DP	64 bytes	$\pm 1{,}8 \cdot 10^{308}$ a $\pm 4{,}9 \cdot 10^{-324}$
char	Carácter	2 bytes	0 a 65.535
boolean	Lógico	1 bit	true o false

El resto de tipos de datos que no son simples, son considerados ***referenciales***.

Estos tipos son básicamente las clases, en las que se basa la programación orientada a objetos.

Al declarar un objeto perteneciente a una determinada clase, se está reservando una zona de memoria donde se almacenarán los atributos y otros datos pertenecientes a dicho objeto.

Lo que se almacena en el objeto en sí, es un puntero, referencia, a dicha zona de memoria.

Dentro de estos tipos pueden considerarse las ***interfaces***, los ***Strings*** y los ***vectores***, que son unas clases un tanto especiales, y que se verán en detalle posteriormente.

A diferencia de otros lenguajes de programación, los ***Strings*** en Java no son un tipo simple de datos sino un objeto.

Los valores de tipo String van entre comillas dobles (“Hola”), mientras que los de tipo char van entre comillas simples

Tipos de datos Referenciales

Tipo de datos simple	Clase equivalente
byte	**`java.lang.Byte`**
short	**`java.lang.Short`**
int	**`java.lang.Integer`**
long	**`java.lang.Long`**
float	**`java.lang.Float`**
double	**`java.lang.Double`**
char	**`java.lang.Character`**
boolean	**`java.lang.Boolean`**

Operadores

1. Aritméticos
2. Relaciónales
3. Lógicos
4. De bits
5. Asignación
6. Prioridad

Operadores Aritméticos

Operador	Formato	Descripción
+	op1 + op2	Suma aritmética de dos operandos
-	op1 - op2	Resta aritmética de dos operandos
-	-op1	Cambio de signo
*	op1 * op2	Multiplicación de dos operandos
/	op1 / op2	División entera de dos operandos
%	op1 % op2	Resto de la división entera o módulo
++	++op1 op1++	Incremento unitario
--	--op1 op1--	Decremento unitario

Operadores Relacionales

Operador	Formato	Descripción
`>`	`op1 > op2`	Devuelve `true` (cierto) si `op1` es mayor que `op2`
`<`	`op1 < op2`	Devuelve `true` (cierto) si `op1` es menor que `op2`
`>=`	`op1 >= op2`	Devuelve `true` (cierto) si `op1` es mayor o igual que `op2`
`<=`	`op1<= op2`	Devuelve `true` (cierto) si `op1` es menor o igualque `op2`
`==`	`op1 == op2`	Devuelve `true` (cierto) si `op1` es igual a `op2`
`!=`	`op1 != op2`	Devuelve `true` (cierto) si `op1` es distinto de `op2`

Operadores Lógicos

Operador	Formato	Descripción
`&&`	`op1 && op2`	Y lógico. Devuelve `true` si son ciertos `op1` y `op2`
`||`	`op1 || op2`	O lógico. Devuelve `true` si son ciertos `op1` o `op2`
`!`	`!op1`	Negación lógica. Devuelve `true` si es `false op1`.

Estos operadores actúan sobre operadores o expresiones lógicas, es decir, aquellos que se evalúan a cierto o falso (true / false).

Operadores de Bits

Operador	Formato	Descripción
`>>`	`op1 >> op2`	Desplaza `op1`, `op2` bits a la derecha
`<<`	`op1 << op2`	Desplaza `op1`, `op2` bits a la izquierda
`>>>`	`op1 >>> op2`	Desplaza `op1`, `op2` bits a la derecha (sin signo)
`&`	`op1 & op2`	Realiza un Y (AND) a nivel de bits
`|`	`op1 | op2`	Realiza un O (OR) a nivel de bits
`^`	`op1 ^ op2`	Realiza un O exclusivo (XOR) a nivel de bits
`~`	`~op1`	Realiza el complemento de `op1` a nivel de bits.

Operadores de Asignación

Operador	Formato	Equivalencia
+=	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

Precedencia de Operadores en Java

Operadores postfijos	`[] . (paréntesis)`
Operadores unarios	`++expr --expr -expr ~ !`
Creación o conversión de tipo	`new (tipo)expr`
Multiplicación y división	`* / %`
Suma y resta	`+ -`
Desplazamiento de bits	`<< >> >>>`
Relacionales	`< > <= >=`
Igualdad y desigualdad	`== !=`
AND a nivel de bits	`&`
XOR a nivel de bits	`^`
OR a nivel de bits	`|`
AND lógico	`&&`
OR lógico	`||`
Condicional al estilo C	`? :`
Asignación	`= += -= *= /= %= ^= &= |= >>= <<= >>>=`

Las secuencias de escape son combinaciones del símbolo contrabarra \ seguido de una letra, y sirven para representar caracteres que no tienen una equivalencia en forma de símbolo.

Las posibles secuencias de escape son:

Secuencias de Escape

Secuencia	Significado
\’	Comillas simples
\”	Dobles comillas
\\	Contrabarra
\b	Retroceso
\n	Línea siguiente
\f	Form feed
\r	Retorno de carro
\t	Tabulador
\a	Alarma
\xxx	Carácter en octal
\0	Carácter nulo
\uxxxx	Carácter en hexadecimal Unicode

Palabras Reservadas

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Identificadores

Los identificadores son los nombres que se les da a

- Variables.
- Clases.
- Interfaces.
- Atributos.
- Métodos.

Usar nombres significativos

Usar nombres descriptivos para todos los identificadores (nombres de clases, de variables, de métodos).

Evitar ambigüedades.

Evitar abreviaturas.

Están formados por letras y dígitos.

No pueden empezar por un dígito numérico.

No pueden contener ninguno de los caracteres especiales.

Los caracteres especiales y signos de puntuación siguientes:

+ - * / = % & # ! ? ^ “ ‘ ~ \ | < > ( ) [ ] { } : ; . ,

No puede ser una palabra reservada de Java.

Java hace distinción entre mayúsculas y minúsculas.

Pueden estar formados por cualquiera de los caracteres del código Unicode.

No pueden utilizarse espacios en blanco ni símbolos coincidentes con operadores.

La longitud máxima de los identificadores es prácticamente ilimitada.

No pueden ser iguales a otro identificador declarado en el mismo ámbito.

Por convenio, los nombres de las variables y los métodos deberían empezar por una letra minúscula y los de las clases por mayúscula.

Si el identificador está formado por varias palabras la primera se escribe en minúsculas (excepto para las clases) y el resto de palabras se hace empezar por mayúscula.

Estas reglas no son obligatorias, pero son convenientes ya que ayudan al proceso de codificación de un programa, así como a su legibilidad.

Es más sencillo distinguir entre clases y métodos o variables.

Sentencias y Expresiones

Las ***sentencias de asignación*** almacenan el valor representado por el lado derecho de la sentencia en una variable nombrada a la izquierda.

```
int telefonoContacto = 0;
```

Las sentencias de asignación funcionan tomando el valor de lo que aparece del lado derecho del operador y copiando dicho valor en una variable ubicada en el lado izquierdo.

Normalmente, las sentencias se ponen unas debajo de otras, aunque sentencias cortas pueden colocarse en una misma línea.

La parte de la derecha se denomina ***expresión.***

Las expresiones son cosas que la computadora puede evaluar.

Una regla es que el ***tipo de una expresión*** debe coincidir con el ***tipo de la variable*** a la que es asignada.

Los métodos de modificación deben comenzar con el prefijo "***set***":

setAlgo(..)

Los métodos de acceso deben comenzar con el prefijo "***get***":

getAlgo(..)

Los métodos de acceso con valores de retorno booleanos generalmente comienzan con el prefijo "***es***":

esAlgo(..);

por ejemplo, esVacio( )

Los nombres de las clases comienzan con una letra mayúscula.

Los nombres de las clases son sustantivos en singular.

Los nombres de los métodos y de las variables comienzan con letras minúsculas.

Tanto los nombres de las clases, como los de los métodos y los de las variables, emplean letras mayúsculas entre medio para aumentar la legibilidad de los identificadores que lo componen; por ejemplo:

numeroDeElementos

Las constantes se escriben en MAYÚSCULAS

Ocasionalmente se utiliza el símbolo de subrayado en el nombre de una constante para diferenciar los identificadores que lo componen:

VALOR_MAXIMO

Esquema

Un nivel de indentación es de cuatro espacios

Todas las sentencias de un bloque se indentan un nivel

Las llaves de las clases y de los métodos se ubican solas en una línea

Las llaves que encierran el bloque de código de la clase y las de los bloques de código de los métodos se escriben en una sola línea y con el mismo nivel de indentación.

Por ejemplo:

```
public int getEdad(){
   sentencias
}
```

Para los restantes bloques de código, las llaves se abren al final de una línea

En todos los bloques de código restantes, la llave se abre al final de la línea que contiene la palabra clave que define al bloque.

La llave se cierra en una línea independiente, alineada con la palabra clave que define dicho bloque.

```
while(condición){
   sentencias
}
if(condición){
   sentencias
}
else{
   sentencias
}
```

Use siempre llaves en las estructuras de control.

Se usan llaves en las sentencias *if* y en los ciclos aun cuando el cuerpo esté compuesto por una única sentencia.

Use un espacio antes de la llave de apertura de un bloque de una estructura de control.

Use un espacio antes y después de un operador.

Use una línea en blanco entre los métodos (y los constructores).

Use líneas en blanco para separar bloques lógicos de código; es decir, use líneas en blanco por lo menos entre métodos, pero también entre las partes lógicas dentro de un mismo método.

Documentación

Cada clase tiene un comentario de clase en su parte superior

El comentario de clase contiene como mínimo

- Una descripción general de la clase
- El nombre del autor (o autores)
- Un número de versión

Cada persona que ha contribuido en la clase debe ser nombrada como un autor o debe ser acreditada apropiadamente de otra manera.

Un número de versión puede ser simplemente un número o algún otro formato.

Lo más importante es que el lector pueda reconocer si dos versiones no son iguales y determinar cuál es la más reciente.

Cada método tiene un comentario

Los comentarios son legibles para javadoc

Los comentarios de la clase y de los métodos deben ser reconocidos por javadoc; en otras palabras: deben comenzar con el símbolo de comentario «/**».

Comente el código sólo donde sea necesario

Se deben incluir comentarios en el código en los lugares en que no resulte obvio o sea difícil de comprender (y preferentemente, el código debe ser obvio o fácil de entender, siempre que sea posible) y

donde ayude a la comprensión de un método. No comente sentencias obvias, ¡asuma que el lector comprende Java!

Restricciones del Lenguaje

Orden de las declaraciones: campos, constructores, métodos.

Los elementos de una definición de clase aparecen (si se presentan) en el siguiente orden: sentencias de paquete, sentencias de importación, comentario de clase, encabezado de la clase, definición de campos, constructores, métodos.

Los campos no deben ser públicos (con excepción de los campos final).

Use siempre modificadores de acceso.

Especifique todos los campos y los métodos como privados, públicos o protegidos.

Nunca use el acceso por defecto *(package private).*

Importe las clases individualmente

Es preferible que las sentencias de importación nombren explícitamente cada clase que se quiere importar y no al paquete completo.

```
import java.util.ArrayList;
import java.util.HashSet;
```

Es mejor que:

```
import java.util.*;
```

Incluir siempre un constructor (aun cuando su cuerpo quede vacío).

Incluir siempre una llamada al constructor de una superclase.

En los constructores de las subclases no deje que se realice la inserción automática de una llamada a una superclase.

Incluir explícitamente la invocación super(...), aun cuando funcione bien sin hacerlo.

Inicialice todos los campos en el constructor.

Modismos del Código

Use iteradores en las colecciones

Para iterar o recorrer una colección, use un *ciclo for-each*.

Cuando la colección debe ser modificada durante una iteración, use un Iterator en lugar de un índice entero.