Tema 2

Tipos de Datos y Expresiones en Java

Tipos de Datos

Cada uno de los datos (atributos, variables, parámetros, valor de regreso de un método, expresiones) tiene un tipo. El tipo de un dato establece:

- El rango de valores que puede tomar el dato
- Las operaciones que se pueden realizar con ese dato.

Java es un lenguaje fuertemente tipado, es decir cada uno de los datos tiene un tipo y cada tipo está definido estrictamente. En cada asignación ya sea explícita o a través del paso de parámetros en la invocación de un método se comprueba la compatibilidad de tipos. No hay conversiones implícitas de tipo que pueda causar un conflicto.

Tipos Primitivos

Los tipos de Java se dividen en dos grupos, los grupos primitivos y los objetos. Los tipos simples representan valores simples, no objetos. A pesar de que java es un lenguaje totalmente orientado a objetos, los tipos simples no lo están por razones de eficiencia.

Tipos Enteros

Los tipos enteros se utilizan para representar valores enteros con signo, esto es, pueden tomar valores tanto positivos como negativos. Los diferentes tipos de enteros se diferencian en el rango de valores que pueden tomar. La tabla siguiente muestra los tipos enteros y su rango:

Tabla 2.1. Tipos Enteros

Tipo	Especificador de tipo	Tamaño (en bytes)	Rango
byte	`byte`	1	-128 a 127
entero corto	`short`	2	-32768 a 32767
entero	`int`	4	-2,147,483,648 a 2,147,483,647
entero largo	`long`	8	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807

Tipos Reales

Los tipos reales o de punto flotante se utilizan cuando deseamos trabajar con números que tienen parte fraccionaria. La siguiente tabla muestra los tipos reales y su rango.

Tabla 2.2. Tipos Reales

Tipo	Especificador de tipo	Tamaño (en bytes)	Rango	Precisión (en cifras significativas)
flotante	`float`	4	$3.4e^{-38}$ a $3.4e^{+38}$	7
doble	`double`	8	$1.7e^{-308}$ a $1.7\ e^{+308}$	15

Tipo Booleano

El tipo booleano empleado para representar valores lógicos sólo puede tomar dos valores: **true** (verdadero) y **false** (falso).

Tabla 2.3. Tipo Booleano

Tipo	Especificador de tipo	Rango
Booleano	`boolean`	`true, false`

Tipo Carácter

El tipo carácter de Java nos permite almacenar caracteres y su tamaño es de dos bytes. Los caracteres en Java se codifican en Unicode, donde cada carácter es codificado como un número entero en el rango de 0 a 65,535. Los primeros 128 caracteres de Unicode corresponden a los caracteres del código ASCII. Unicode nos permite representar cualquier carácter de los diferentes alfabetos empleados en los diferentes lenguajes hablados en todo el mundo.

Tabla 2.4. Tipo Carácter

Tipo	Especificador de tipo	Tamaño (en bytes)	Rango
carácter	`char`	2	0 a 65535

Expresiones

Procesar datos implica combinarlos de alguna forma, dando como resultado otros datos. Para especificar como deben de combinarse esos datos se utilizan las expresiones.

Una expresión es una combinación de operandos y operadores. Los operandos representan valores y los operadores indican las operaciones a realizarse con ellos, para obtener un resultado.

Operandos

Hay tres tipos de operandos en Java: Literales, Variables e Invocaciones a Métodos.

Literales

Literal es un valor constante. Las literales en Java pueden ser de los siguientes tipos: Enteras, flotantes, booleanas, de carácter y de cadena.

Literales Enteras

Una literal entera es un número entero en el rango 0 al valor máximo permitido en un entero largo. Las literales enteras se crean del tipo int. Si la literal sobrepasa el rango permitido para un int debe agregársele el sufijo L o l para que el compilador la cree como long (por ejemplo 2500000000L).

Literales de punto flotante

Las literales de punto flotante contienen un punto decimal (por ejemplo 123.4) o un exponente (por ejemplo 1e-2) o ambos (por ejemplo 1.234e2). Las literales de punto flotante se crean del tipo double. Podemos forzar a una literal al tipo float usando el sufijo f o F (por ejemplo 2.36e-3f).

Literales Booleanas

Sólo hay dos literales booleanas: **true** (verdadero) y **false** (falso).

Literales de carácter

Una literal de carácter se almacena en un entero de dos bytes sin signo y representa el código asociado al carácter Unicode. Una literal de carácter puede escribirse de las siguientes formas:

- Un carácter delimitado por apóstrofes. Por ejemplo: 'a', 'Z'.

- Los caracteres de control y algunos caracteres difíciles de imprimir pueden representarse mediante una secuencia de escape delimitada por apóstrofes. Una secuencia de escape son ciertos caracteres precedidos por una diagonal invertida (\) o exactamente cuatro dígitos hexadecimales precedidos de los caracteres \u.

  Por ejemplo: '\n', '\a', '\u0041'

En la tabla siguiente se muestra la lista de las secuencias de escape:

Tabla 2.5. Caracteres Expresados Como Secuencias de Escape

Secuencia	Significado
\b	Regresa un espacio
\f	Salto de página
\n	Salto de línea
\r	Regreso de carro
\t	Tab horizontal
****	Diagonal invertida
\'	Apóstrofe
\"	Comillas
\uxHHHH	Valor hexadecimal de exactamente cuatro dígitos

Literales de tipo cadena

Una literal de tipo cadena es una secuencia de caracteres delimitada por comillas. La secuencia de caracteres puede contener secuencias de escape. Por ejemplo:

```
"Hola"
"Instituto Tecnológico de Sonora"
"\nEn una nueva línea\nEn una nueva línea"
```

Variables

Un dato que puede variar su valor durante la ejecución de un programa es considerado como una variable. Podemos tener variables de cualquier tipo de dato primitivo, s.

Todas las variables deben declararse antes de usarse. Al declarar una variable estamos haciendo dos cosas: Primero, le informamos al compilador de la existencia de una variable cuyo nombre y tipo se especifican en la declaración; y segundo le pedimos al compilador que nos reserve espacio en la memoria para la variable. La sintaxis para declarar una variable es la siguiente:

```
tipo nomVar1[ = exp1][, nomVar2[ = exp2]...]
```

nomVar1, *nomVar2* ... son los nombre de las variables que se están declarando. Las variables son del tipo de dato *tipo* que puede ser cualquier tipo primitivo, el nombre de una clase o interfaz.

Las variables pueden inicializarse al momento de su declaración. *exp1*, *exp2*, ... son las expresiones a cuyos valores se inicializan las variables *nomVar1*, *nomVar2* ..., respectivamente. Por lo general *exp1*, *exp2*, ... son literales aunque las variables locales pueden inicializarse a cualquier expresión que pueda evaluarse al momento en que la variable es creada.

Por ejemplo:

```
int a, b, c, x = 10;
float l, resultado, producto = 8.75;
char respuesta;
```

Calificador final en Declaración de Variables

El calificador **final** puede aplicarse a la declaración de cualquier variable para especificar que su valor no cambia, comportándose como una literal.

Por ejemplo:

```
final int x = 8;
final float g = 9.81;
```

Una variable con el calificador final debe ser inicializada al momento de su declaración. Cualquier intento de modificar su valor posteriormente, genera un error por parte del compilador.

Ámbito

El ámbito de un identificador define la región de código donde se conoce dicho identificador.

Ámbito de una Variable Local

Una variable local es aquella que se declara dentro de un bloque. Un bloque es conjunto de sentencias delimitada por llaves ({, }) dentro de la definición de un método. El ámbito de una variable local empieza con la declaración de la variable y termina al final del bloque en donde está declarada la variable.

Tiempo de Vida

El tiempo de vida establece cuando una variable es creada y cuando es destruida.

Tiempo de Vida de una Variable Local

Una variable local es creada en el momento en que se declara y es destruida en el momento en que el control del programa sale del bloque en que está declarada la variable.

Invocaciones a Métodos

Un método representa una funcionalidad de una clase (una de las cosas que sabe hacer un objeto de esa clase). La invocación a ese método es la orden para que ejecute esa funcionalidad. La sintaxis de la invocación de un método tiene dos variantes:

```
nomObjeto.nomMetodo(lista de argumentos)
nomClase.nomMetodoEstatico(lista de argumentos)
```

Donde `nomMetodo` es el nombre del método y *`nomObjeto`* es el nombre de su objeto. `nomMetodo` es el nombre de un método estático y *`nomClase`* es el nombre de su Clase. *`lista de argumentos`* es la lista de los argumentos, esto es, la lista de los valores que se le envían al método a invocarlo y que el método recibe en sus parámetros. Debe de haber una correspondencia en número y tipo entre los argumentos y los parámetros.

La invocación de un método que regresa un valor se interpreta como un operando cuyo valor y tipo son el valor y tipo regresado por el método. La invocación de un método aparece como parte de una expresión. Al evaluar la expresión, el programa substituye la invocación del método por el valor que regresado por éste.

Por ejemplo suponga la siguiente expresión:

```
3.0 + Math.sqrt(4.0)
```

Al estar evaluando esta expresión, la invocación al método estático sqrt() de la clase Math que nos regresa la raíz cuadrada de su argumento, es substituida por el valor que regresa el método, 2.0 en este caso. La expresión, por lo tanto se convierte a:

```
3.0 + 2.0
```

la cual es evaluada a 4.0 posteriormente.

Operadores

En el lenguaje Java, cada uno de los tipos de datos primitivos: Enteros y flotantes, tiene su propia aritmética, la aritmética de enteros y la aritmética de punto flotante. Cada aritmética tiene sus propias reglas y opera sólo sobre los datos de su mismo tipo. La aritmética de enteros opera sólo con enteros y produce resultados enteros. Lo mismo sucede con los flotantes. No se permiten mezclas de enteros y flotantes. Cuando una operación existe en ambas aritméticas, éstas comparten el mismo símbolo para el

operador. Por ejemplo el símbolo para la suma en las dos aritméticas es (+). El compilador sabe que aritmética aplicar de acuerdo al tipo de los operandos.

Algunos de los operadores de Java son unarios, es decir, operan sobre un sólo operando, otros son binarios, operan sobre dos operandos y hay un operador terciario, trabaja sobre tres operandos.

Operadores Aritméticos

Operadores positivo (+) y negativo (-)

El operador unario negativo, -, le cambia el signo al operando al que está asociado. Si el operando es un entero, el resultado es un entero, si el operando es flotante el resultado es flotante. El operador unario positivo, +, sólo existe por simetría ya que no hace nada.

Ejemplos:

```
-56
-34.78
+25         igual a 25
```

Operadores de multiplicación (*), división (/) y módulo (%)

El operador de multiplicación, *, nos da el producto de sus operandos. Si los operandos son enteros, el resultado es un entero, si los operandos son flotantes el resultado es flotante.

El operador de división, /, nos da la parte entera del cociente si sus operandos son enteros. Si los operandos son flotantes el resultado es flotante.

El operador módulo, %, sólo opera con operandos enteros y nos da el residuo de la división de sus operandos.

Ejemplos:

Expresión	Resultado
`45 * 23`	`1035`
`18.27 * 2.975`	`54.3532`
`25 / 4`	`6`
`25.0 / 4.0`	`6.25`
`25 % 4`	`1`

Operadores de suma (+) y resta (-)

Los operadores de suma, +, y resta, -, nos dan la suma y la diferencia de sus operandos, respectivamente. Si los operandos son enteros, el resultado es un entero, si los operandos son flotantes el resultado es flotante.

Ejemplos:

Expresión	Resultado
`4 + 23`	`27`
`8.2 + 29.75`	`37.95`
`25 – 4`	`21`
`2.5 - 1.4`	`1.1`

Precedencia y asociatividad de operadores

Los operadores tienen reglas de precedencia y asociatividad que determinan precisamente como se evalúan las expresiones. La siguiente tabla resume las reglas de precedencia y asociatividad de todos los operadores de Java, incluyendo algunos que aún no han sido tratados. Los operadores situados en la misma línea tienen la misma precedencia, las filas están colocadas por orden de precedencia decreciente.

Si una expresión contiene más de un operador, el orden en que se efectúan las operaciones está dado por las siguientes reglas.

- Las expresiones encerradas entre paréntesis son evaluadas primero.

Tabla 2.6 Precedencia y Asociatividad de los Operadores de Java

Operador	Asociatividad
`! ~ ++ -- + - (`*`tipo`*`)`	Derecha a izquierda
`* / %`	Izquierda a derecha
`+ -`	Izquierda a derecha
`<< >> >>>`	Izquierda a derecha
`< <= > >=`	Izquierda a derecha
`== !=`	Izquierda a derecha
`&`	Izquierda a derecha
`^`	Izquierda a derecha
`|`	Izquierda a derecha
`&&`	Izquierda a derecha
`||`	Izquierda a derecha
`?:`	Derecha a izquierda
`= += -= *= /= %= <<= >>= &= ^= |=`	Derecha a izquierda
`,`	Izquierda a derecha

- Si en una expresión hay operadores de diferente precedencia, la operación del operador de mayor precedencia se realiza primero.

- Si en una expresión hay operadores de igual precedencia, las operaciones se realizan en el orden dado por la asociatividad de los operadores.

Conversión de Tipos Implícita

Cuando un operador tiene operandos de diferente tipo, éstos se convierten a un tipo común, en forma implícita o automática, de acuerdo con las siguientes reglas:

Promoción entera

Un byte o un short se convierten a int. Este proceso se conoce como **promoción entera**.

Conversiones aritméticas

- Si algún operando es de tipo double, el otro operando es convertido a double y el resultado es de tipo double.

- Si no se aplica la regla anterior y si algún operando es de tipo float, el otro se convierte a float y el resultado es float.

- Si no se aplica la regla anterior, y si algún operando es de tipo long, el otro se convierte a long y el resultado es long.

- Si no se pueden aplicar ninguna de las reglas anteriores, los operandos deben de ser del tipo int y el resultado será de tipo int.

En resumen cuando un operador tiene operandos de diferente tipo, el operando del tipo menor es promovido al tipo mayor y el resultado es del tipo mayor.

Suponga que una variable es el operando al que se le cambia el tipo en una conversión implícita. Esto no quiere decir que la variable ni su contenido cambia de tipo. Lo que cambia de tipo es una copia del valor de la variable la cual se almacena en una localidad de memoria temporal (que puede ser en la memoria principal o en un registro en el CPU) y la cual se utiliza para efectuar los cálculos.

Ejemplos:

Expresión	Resultado

`12 + 3.45 * 2`	`12 + 6.9000000000000000000`
	`18.900000000000000`
	`Un doble.`

```
4 % 3 - sqrt(45.89)          4 % 3 - 6.774216000000000000
                             1 - 6.774216000000000000
                             -5.774216000000000000
                             Un doble.

(89 / 7)/(95 % 15)           12 / 5
                             2
                             Un entero.
```

Conversión de tipos explícita

Se puede forzar la conversión explícita de una expresión a un tipo dado usando el operador cast, cuya sintaxis es la siguiente:

```
(tipo)expresión
```

En este caso la expresión se convierte al tipo dado de acuerdo a las reglas ya establecidas. Por ejemplo supongamos la siguiente declaración:

```
int x = 1000000, y = 4000;
```

la expresión

```
x * y
```

no da como resultado 4,000,000,000 ya que como x e y son enteros su producto es un entero. El número 4,000,000,000 excede el rango de los enteros. Una solución es usar el operador cast para cambiar uno de los operandos a long de manera que dada las conversiones implícitas la operación sea entre largos y el resultado sea largo.

```
(long)x * y
```

Como segundo ejemplo suponga que se va a promediar una serie de números que representan las edades de los alumnos de un grupo y que la suma de las edades y el número de alumnos se encuentra en las variables enteras sumaEdades y nAlumnos. El cociente:

```
sumaEdades/nAlumnos
```

producirá como resultado sólo la parte entera del cociente. Si se desea un resultado con parte fraccionaria, podemos hacer lo siguiente:

```
(double)sumaEdades/nAlumnos
```

Al igual que en el caso de la conversión implícita, la conversión de tipo explícita tampoco modifica el tipo ni contenido de la variable cuyo valor es convertido. Sólo se modifica una copia temporal usada para realizar los cálculos.

Operadores relacionales

Los **operadores relacionales** nos permiten comparar dos valores. El resultado de la comparación es un valor booleano: false si es falso o true si es verdadero. Todos los operadores relacionales de Java son binarios. Esto es, operan sobre dos operandos. Los operadores relacionales son:

Operador menor que (<)

El operador menor que, < da como resultad**o** true (verdadero) si su operando izquierdo es menor que el derecho y false (falso) en caso contrario.

Operador menor o igual que (<=)

El operador menor o igual que, <= da como resultado true si su operando izquierdo es menor o igual que el derecho y false (falso) en caso contrario.

Operador mayor que (>)

El operador mayor que, > da como resultado true si su operando izquierdo es mayor que el derecho y false en caso contrario.

Operador mayor o igual que (>=)

El operador mayor o igual que, >= da como resultado true si su operando izquierdo es mayor o igual que el derecho y false en caso contrario.

Ejemplos:

Expresión	Resultado
`4 < 3`	`false`
`5 >= 2`	`true`

Operadores de igualdad

Al igual que los operadores relacionales, los operadores de igualdad nos permiten comparar dos valores. El resultado de la comparación es un booleano: false si es falso o true si es verdadero. Los operadores relacionales también son binarios. Los operadores relacionales son:

Operador igual a (==)

El operador igual a, == da como resultado true si su operando izquierdo es igual al derecho y false en caso contrario.

Operador diferente a (!=)

El operador diferente a, != da como resultado true si su operando izquierdo es diferente al derecho y false en caso contrario.

Ejemplos:

Expresión	Resultado
`4 == 3`	`false`
`5 != 2`	`true`

Operadores lógicos

Los operadores lógicos sólo operan sobre operandos booleanos. Los operandos lógicos normalmente se utilizan para construir expresiones más complejas a partir de expresiones que contienen los operadores relacionales o de igualdad.

Operador de negación (!)

El operador unario de negación, !, convierte un operando con valor de falso (false) a verdadero (true) y un valor de verdadero (true) a falso (false).

Operador de intersección (&&)

El operador de intersección, &&, nos da como resultado verdadero (true) sólo sí los dos operandos son verdaderos y falso (false) en caso contrario.

Operador de unión (||)

El operador de unión, ||, nos da como resultado verdadero (true) si al menos uno de los dos operandos es verdadero y falso (false) en caso contrario.

Evaluación en corto circuito

En la evaluación de expresiones que tienen operandos && y ||, el proceso de evaluación se detiene tan pronto se conoce si el resultado es falso o verdadero. Esto se conoce **evaluación en corto circuito**. Suponga que tenemos la siguiente expresión:

```
exp1 && exp2
```

donde *exp1* y *exp2* son expresiones. Supóngase que al comenzar a evaluar la expresión anterior se encuentra que *exp1* es falsa (vale false), entonces *exp2* no evalúa ya que independientemente del valor de *exp2*, el valor de *exp1* **&&** *exp2* es falso. En forma similar, en la evaluación de la expresión

exp1 **||** *exp2*

si se encuentra que *exp1* es verdadera, entonces *exp2* no evalúa ya que independientemente del valor de *exp2*, el valor de *exp1* || *exp2* es verdadero.

Ejemplos:

Expresión	Resultado	
`!true`	`false`	
`13 > 9 && 9 > 4`	`true && true` `true`	
`3 != 4 || 7 < 15`	`true`	Aquí sólo fue necesario evaluar 3 != 4 ya que al dar como resultado verdadera, toda la expresión es verdadera.

Operadores de asignación

Los operadores de asignación permiten reemplazar el valor de una variable. Estos son:

`=  +=  -=  *=  /=  %=`

El **operador de asignación =** debe de tener como operando izquierdo una variable y como operando derecho una expresión:

nomVar **=** *expresión*

La operación de asignación reemplaza el valor que tiene la variable por el valor de la expresión. Además, *nomVar* **=** *expresión* constituye a su vez una expresión cuyo valor es *expresión* de tal forma que se permiten construcciones como:

nomVar1 **=** *nomVar2* **=** *expresión*

En este caso, tanto *nomVar1* como *nomVar2* toman el valor de *expresión*.

Si en una asignación el tipo del valor de *expresión* es diferente al tipo de *nomVar*, tienen lugar una conversión implícita. El valor de *expresión* se convierte al tipo de *nomVar* de acuerdo a las siguientes reglas:

Conversiones enteras

Cuando cualquier tipo entero se convierte a otro tipo entero, el valor no se cambia si puede ser representado en el nuevo tipo. En el otro caso el resultado no está definido.

Enteros y flotantes

Cuando un valor de tipo flotante se convierte a un tipo entero, la parte fraccionaria se descarta; si el valor resultante no puede ser representado en el tipo entero, el resultado no está definido.

Conversiones flotantes

Cuando un valor flotante menos preciso se convierte a un flotante igual o más preciso, el valor no se modifica. Cuando un flotante más preciso se convierte a un flotante menos preciso, y el valor esta dentro del rango representable, entonces el valor se redondea a la menor precisión. Si el resultado esta fuera de rango el comportamiento no esta definido.

Las expresiones como

```
i = i + 2
```

donde la variable de la izquierda se repite en la derecha, pueden escribirse en forma más compacta como:

```
i += 2
```

mediante el operador de asignación +=. La mayor parte de los operadores binarios poseen un correspondiente operador de asignación, cuya sintaxis es la siguiente:

```
nomVar operador= expresión
```

que es equivalente a la expresión:

```
nomVar = nomVar operador expresión
```

donde *operador* puede ser uno de los siguientes:

```
+   -   *   /   %
```

Operadores de incremento y decremento

Los operadores de incremento y decremento son operadores unarios que solo se aplican a variables enteras. El operador de incremento ++ es un operador que le suma 1 a su operando. El operador de decremento -- es un operador que le resta 1 a su operando.

Estos tipos de operadores se pueden usar como prefijos (antes de su operando, como en ++n ó --n) o como sufijos (después de su operando, como en n++ ó n--). En ambos casos se produce el mismo efecto, incrementar o decrementar el operando en 1. Sólo que, si el

operador se usa como prefijo, el operando es incrementado o decrementado antes de ser usado y si el operador se usa como sufijo, el operando es incrementado o decrementado después de ser usado. Por ejemplo considera las siguientes asignaciones:

```
n = 5;        n = 5;
x = n++;      y = ++n;
```

en ambos casos el valor final de n es 6. El valor asignado a x es 5 y el valor asignado a y es 6.

Operador condicional

El operador condicional, **?:** es un operador ternario y su sintaxis es la siguiente:

```
exp1 ? exp2 : exp3
```

exp1 debe ser una expresión booleana, esto es, que se pueda evaluar a falso (false) o verdadero (true). *exp2* y *exp3* pueden ser de cualquier tipo.

El valor de *exp1* **?** *exp2* **:** *exp3* se determina de la siguiente manera: Primero se evalúa *exp1*, si es verdadera *exp1* **?** *exp2* **:** *exp3* toma el valor de *exp2*. Si es falsa *exp1* **?** *exp2* **:** *exp3* toma el valor de *exp3*

Por ejemplo:

```
y = x > 7? 0: 10;
```

Aquí, y tomará el valor de 0 para x > 7 y de 10 para x ≤ 7.