2 | Tipos primitivos y bits

En el momento en que se elige utilizar un sistema físico para realizar cómputos, éstos quedan sujetos a las leyes que rigen estos sistemas. Al elegir utilizar sistemas binarios con dos estados distinguibles, dichos cómputos quedan encuadrados por la lógica booleana; más aún, por una lógica booleana con un número grande, pero finito de símbolos disponibles.

Varios lenguajes de programación nos permiten acceder en forma privilegiada a aquellos tipos de datos que son representables con mayor facilidad dentro de este encuadre y gozan de una implementación particularmente eficiente, a estos tipos se les conoce como *tipos primitivos* y son los átomos a partir de los cuales se manejará cualquier tipo de información.

Meta

Que el alumno se familiarice con la representación en la computadora de los tipos primitivos de Java.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Visualizar adecuadamente cómo están siendo representados los valores de los tipos primitivos en la memoria de la computadora.
- Identificar el problema de *desbordamiento*[1] al realizar operaciones con números grandes.
- Utilizar apropiadamente la *conversión*[2] de tipos primitivos.
- Preveer los efectos de convertir datos de un tipo a otro tipo.
- Empacar y desempacar bits de información dentro de un tipo con más bits.

[1] En inglés *overflow*.
[2] En inglés *casting*.

ANTECEDENTES

Java tiene ocho tipos primitivos, los cuales se muestran en la Tabla 2.1.

Tabla 2.1 Tipos básicos de Java

Nombre	Tamaño	Representación en memoria
`byte`	1 byte	Entero con signo en complemento a 2
`short`	2 bytes	Entero con signo en complemento a 2
`int`	4 bytes	Entero con signo en complemento a 2
`long`	8 bytes	Entero con signo en complemento a 2
`float`	4 bytes	Racional de acuerdo al estándar IEEE 754-1985
`double`	8 bytes	Racional de acuerdo al estándar IEEE 754-1985
`boolean`	≈1 byte	Booleano `true` o `false`
`char`	2 bytes	Caracter Unicode 2.0

* Un byte son 8 bits

Enteros con signo

Java permite representar números enteros $\mathbb{Z}$ contenidos en intervalos de la forma $[A, B]$ con $A < 0, B > 0$ y $A, B \in \mathbb{Z}$. Los valores de A y B dependen del número de bits disponibles para almacenar estos números y del sistema utilizado para representar a los números negativos. El sistema elegido por `Java` es complemento a dos.

Para calcular la representación de un número negativo utilizando complemento a dos se pueden utilizar tres pasos:

1. Se escribe el número en sistema binario, utilizando tantos bits como indique el tipo entero utilizado. Los bits sobrantes a la izquierda valen cero. El bit más a la izquierda es el bit de signo, en este paso debe valer cero. Por ejemplo, el número 14 en un `byte` se escribe:

   00001110

2. Se calcula el complemento a uno: el valor de cada bit es invertido. Aquí ya aparece el signo menos, indicado por un uno en el bit más a la izquierda.

   11110001

3. Se le suma uno. Ésta es ahora la representación del -14.

   11110010

Existe también una receta corta para pasar de la representación binaria a complemento a dos:

Tabla 2.2 Enteros

Nombre	Tamaño	Intervalo
`byte`	1 byte	$[-128, 127]$
`short`	2 bytes	$[-32768, 32767]$
`int`	4 bytes	$[-2147483648, 2147483647]$
`long`	8 bytes	$[-9223372036854775808, 9223372036854775807]$

1. Se copian, de derecha a izquierda, todos los ceros hasta llegar al primer uno,
2. se copia ese uno y
3. a partir de ahí se invierten todos los demás bits.

Por ejemplo:

$$14_{10} = 00001110_2 \rightarrow 11110010_2 = -14_{10}$$

De acuerdo a esta representación, las capacidades de los diferentes tipos de enteros se muestra en la Tabla 2.2. Puedes acceder a estos valores desde el código de `Java`. Por ejemplo:

```
1  int max = Integer.MAX_VALUE;
```

Actividad 2.1

Revisa la documentación de las clases `Byte, Short, Integer y Long` de `Java` y revisa los atributos de clase que permiten acceder a esta información. ¿Cuáles encuentras relevantes?

Intenta sumarle uno a `max`, imprime el resultado en base 10 y su representación en binario. Explica qué pasó.

Números de punto flotante

Para representar números racionales, `Java` utiliza el estándar IEEE 754-1985. Éste especifica una serie de reglas para utilizar notación exponencial en base dos. Un ejemplo de número binario en notación exponencial sería:

$1.001101 \text{x} 10_2^{-101}$

Existen buenas referencias para quienes deseen conocer a detalle cómo funciona esta representación [Fish 2005]. De momento lo que nos interesa es lo siguiente:

Tabla 2.3 Flotantes

Nombre	Tamaño	Intervalo
`float`	4 bytes	$[1.4x10^{-45}, 3.4028235x10^{38}]$
`double`	8 bytes	$[4.9x10^{-324}, 1.79x10^{308}]$

- Se utiliza el bit más a la izquierda para representar el signo del número: 0 si es positivo y 1 si es negativo.
- Los siguientes X bits se utilizan para almacenar el exponente del número, en base dos. Uno de esos bits corresponde al signo del exponente.
- Los bits restantes almacenan la mantiza.

La diferencia entre `float` y `double` radica en el número de bits totales y, por ende, los dedicados al exponente y a la mantiza. Los intervalos aproximados de números que pueden ser representados con estos tipos se listan en la Tabla 2.3.

De entre estos números se cuenta con los *normalizados*, cuya mantiza siempre comienza con 1.????. y los no-normalizados, que pueden tener un cero a la izquierda del punto.

Otro aspecto importante de este tipo de datos, es que no sólo representan números fraccionarios, sino también algunos símbolos especiales:

- `NaN`, que quiere decir "*Not a number*". Se utiliza para representar un resultado numérico inválido, como el obtenido al dividir $0/0$. Para acceder a este valor en `Java` escribe:

```
1 double nan = Double.NaN;
```

- $\pm\infty$. Para acceder a estos valores en `Java` escribe:

```
1 double minf = Double.NEGATIVE_INFINITY;
2 double pinf = Double.POSITIVE_INFINITY;
```

- ± 0. Como se utiliza la representación signo-maginitud, hay dos ceros. Para acceder a estos valores en `Java` escribe:

```
1 double cero = 0.0;
2 double mcero = -0.0;
```

Afortunadamente `Java` dice que ambos son iguales. Aunque el bit de signo en la computadora es diferente.

Actividad 2.2

Utiliza la clase `ImpresoraBinario` para visualizar la representación interna de estos valores especiales.

Boolean

El tipo `boolean` es utilizado para operaciones lógicas. Aunque en la lógica booleana sólo hay dos símbolos: $verdadero$ y $falso$, la representación en la computadora ocupa más espacio que un bit. Este espacio se desperdicia, pero se hace para que las operaciones sean más rápidas, debido a que la computadora puede direccionar un byte con mucha mayor facilidad que un único bit.

Cuando se almacenarán muchos de estos valores y no se operará con ellos frecuentemente, es común que el programador *empaque* los valores booleanos en los bits de un tipo numérico. Por ejemplo, los códigos de permisos para lectura, escritura y ejecución de los archivos de Linux utilizan esta técnica. Imaginemos cómo puede lograrse esto:

Hay un total de nueve permisos que manejar, agrupados por tipo de usuario: el *dueño*, su *grupo* de usuarios y *otros* usuarios. Por cada grupo hay tres permisos: *lectura*, *escritura* y *ejecución*. Como cada permiso puede ser *otorgado* o *denegado*, la lógica booleana es adecuada para trabajar con ellos, pero preferiríamos usar sólo un bit por cada permiso, de modo que sólo se requieran nueve bits. Si quisiéramos trabajar en `Java`, podríamos codificar estos bits dentro de un `short`.

```
1 short permisos;
```

Por ejemplo, para decir que un archivo puede ser leído por todos, escrito por el dueño y ejecutado por el dueño y su grupo, necesitaríamos escribir 111101100, donde los tres primeros bits son los permisos del dueño, los siguientes tres los del grupo y los últimos los de otros. Así quedan registrados `rwx r-x r-`. En base ocho esto es 754, en `Java` se puede asignar:

```
1 permisos = 0754;
```

Actividad 2.3

Imprime cómo se ve esto en binario. ¿Cuánto vale `permisos` en base 10? (Sólo mándalo imprimir, `Java` trabaja por defecto en base 10).

OPERADORES SOBRE BITS

Cuando se desea manipular los datos bit a bit, se utilizan los siguientes operadores:

Tabla 2.4 Operadores sobre bits en orden de precedencia.

Operandos	Asociatividad	Símbolo	Descripción
unario prefijo	izquierda	~<expresión>	complemento en bits
binario infijo	izquierda	<<	corrimiento de bits a la izquierda
binario infijo	izquierda	>>	corrimiento de bits a la derecha llenando con ceros
binario infijo	izquierda	>>>	corrimiento de bits a la derecha propagando signo
binario infijo	izquierda	&	AND de bits
binario infijo	izquierda	^	XOR de bits
binario infijo	izquierda	|	OR de bits

Actividad 2.4

Realiza ahora pruebas con los operadores de corrimiento «, », »>. Recorre los números del iniciso anterior por uno y tres bits. Se usa así:

```
1  int num = 345;
2  int resultado = num << 3;
```

Imprime tus resultados. Asegúrate de que entiendes lo que hizo cada operador.

Actividad 2.5

Ahora toma los permisos de la sección anterior: ¿qué operaciones necesitas hacer para que todos los usuarios tengan permiso de escritura? Verfica tu respuesta imprimiendo las representaciones binarias.

EJERCICIOS

1. Junto con esta práctica se te entrega el código de la clase `ImpresoraBinario`. En el método `main` de la clase de uso `PruebasPrimitivos` utiliza a `ImpresoraBinario` para imprimir en pantalla la representación con bits de los siguientes números:
   - El `int` 456

- El mismo número pero en una variable tipo `long`
- El mismo número pero en una variable tipo `float`
- El mismo número pero en una variable tipo `double`

Explica ¿qué diferencias observas?

2. Repite los mismos pasos, pero ahora con -456

3. Repite los mismos pasos, pero con -456.601. Ojo, en este caso deberás hacer un casting para guardar el número en los tipos correspondientes a enteros y perderás la parte fraccionaria. ¿Qué número queda almacenado?

4. Finalmente, crea un `int` llamado `máscara` cuyos últimos cuatro dígitos sean unos. Ahora utiliza el operador de corrimiento necesario, para colocarlos en las posiciones 4 a la 7 (contado de derecha a izquierda). ¿Qué número obtienes?

5. Dado un `int num = 1408`, realiza la operación `num & máscara`. Imprime los bits resultantes y el valor numérico de tu resultado. Repite el ejercicio con `|` y `~`.