2 | Tipos primitivos y bits

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.

Código Auxiliar 2.1: Compiladores e intérpretes

https://github.com/computacion-ciencias/icc-tipos-primitivos

Antecedentes

En el momento en que se elige utilizar un sistema físico para realizar cómputos, éstos quedan sujetos a las leyes que rigen estos sistemas. Se vuelve necesario almacenar toda la información a procesar en esos sistemas. Actualmente se utilizan sistemas en los cuales se puedan distinguir fácilmente dos estados: encendido/apagado, cargado/descargado, pasa

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

corriente/no pasa corriente, etc. Al elegir utilizar estos sistemas binarios con dos estados distinguibles, los cómputos quedan encuadrados natualmente por la *lógica booleana* con los valores *verdader* y *falso*; más aún, por una lógica booleana con un número grande, pero finito de símbolos disponibles.

Varios lenguajes de programación nos permiten utilizar en forma privilegiada 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. `Java` tiene ocho tipos primitivos, que se forman al concatenar un número fijo de valores booleanos e interpretarlos como números enteros, racionales, valores booleanos o caracteres para escritura de textos. Estos tipos 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

Estas piezas de información se almacenan en espacios de memoria. Para apartar una localidad de memoria e indicar cómo deben interpretarse y operarse con los bits almacenados en ella se deben especificar tres cosas:

1. El tipo de datos que contendrá,
2. se le debe asignar un nombre y
3. asignársele un valor.

La forma de asignarles valores a variables de estos tipos se ilustra con los ejemplos siguientes:

```
1 byte ochobits = 5;
2 short cortito = 15;
3 int entero = 29328;
4 int binario = 0b111;
5 int octal = 0700;
6 int hexa = 0xff;
7 long largo = 2341;
8 float flotante = 3.14f;
9 double doble = 3.141592;
```

Una vez que se ha *inicializado* así una variable es posible utilizar el valor que contiene tan sólo escribiendo su nombre. También es posible modificar el valor que contiene asignándole uno nuevo. Por ejemplo:

```
int número = 17;       // Declaración y definición
número = 25;           // Asignación de valor nuevo
número = 24 + 3;       // Asignación de valor nuevo
número = número - 8;   // Lectura de valor anterior y asignación de
    ↪ valor nuevo
```

Para verlos dentro de un programa de `Java` hay que colocar las líneas anteriores dentro de un archivo de texto con terminación `.java`. Para el ejemplo siguiente el archivo debe llamarse `Tipos.java`. Véamos qué debe contener ese archivo.

Listado 2.1: Tipos.java

```
public class Tipos {
  public static void main(String[] args) {
    System.out.println("Primitivos");
    byte ochobits = 5;        // Asigna valores enteros
    short cortito = 15;
    int entero = 29328;
    int binario = 0b111;      // Asigna valor binario
    int octal = 0700;         // Asigna valor en base 8
    int hexa = 0xff;          // Asigna valor en base 16
    long largo = 2341L;       // Asigna entero long
    float flotante = 3.14f;   // Asigna flotante
    double doble = 3.141592;

    System.out.println("ochobits␣=␣" + ochobits);
    System.out.println("cortito␣␣␣=␣" + cortito);
    System.out.println("entero␣␣␣␣=␣" + entero);
    System.out.println("binario␣␣␣=␣" + binario);
    System.out.println("octal␣␣␣␣␣=␣" + octal);
    System.out.println("hexa␣␣␣␣␣␣=␣" + hexa);
    System.out.println("largo␣␣␣␣␣=␣" + largo);
    System.out.println("flotante␣=␣" + flotante);
    System.out.println("doble␣␣␣␣␣=␣" + doble);
  }
}
```

En resumen, contiene los siguientes elementos:

- Declaración y definición de la clase `Tipos`.

```
public class Tipos {
  ...
}
```

Todo el código va dentro de esta clase, porque todo código en Java va dentro de una clase.

- Declaración y definición del método `main`. La ejecución del programa comienza por la primer línea dentro de este método.

```
1   public static void main(String[] args) {
2     ...
3   }
```

- Cuerpo del método, todo lo que está escrito entre las llaves:
  - ⋆ Declaraciones de las variables de tipos primitivos y definición, al asignarles un valor.
  - ⋆ Instrucciones para imprimir mensajes en pantalla donde se muestre su valor.

Si escribes el código anterior en el archivo, lo compilas y corres con:

```
$ javac Tipos.java
$ java Tipos
```

Se imprime en la terminal:

```
Primitivos
ochobits = 5
cortito  = 15
entero   = 29328
binario  = 7
octal    = 448
hexa     = 255
largo    = 2341
flotante = 3.14
doble    = 3.141592
```

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:

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?

Entregable: En el paquete de código que recibes para esta práctica, realiza lo siguiente:

1. Crea un archivo `Prueba.java` al lado de `ImpresoraBinario.java`.
2. Observa que, al encontrarse dentro de los directorios:

   `icc/primitivos`

la clase que crearás en este archivo pertenece al paquete

`icc.primitivos`

por lo que la primer línea de código en el archivo de texto deberá decir:

`package icc.primitivos;`

3. Agrega ahora un método `main`, como el del ejemplo anterior, pero su contenido será diferente.

4. Ahí, crea e imprime la variable `max`, mencionada arriba.

5. Para imprimir también su representación en base 2 utiliza la clase

   `ImpresoraBinario`.

   Para ello deberás crear lo que llamaremos un *objeto* de tipo `ImpresoraBinario` dentro de tu método `main` y podrá ser utilizado en las líneas subsecuentes. El código siguiente muestra cómo crear y usar este objeto:

```
1 int uno = 1;
2 ImpresoraBinario p = new ImpresoraBinario();
3 p.imprime(uno);
```

6. Veamos ahora una forma más para compilar y ejecutar este código en forma manual. Con la terminal en el directorio que acabas de clonar de git (si ejecutas `ls` en ese directorio verás que ahí están `build.xml` y `README.md`, entre otros), crea un directorio llamado `classes` con:

```
mkdir classes
```

   ahora ejecuta al compilador del modo siguiente:

```
javac --source-path src -d classes src/icc/primitivos/Prueba
    ↪ .java
```

   observa que las opciones `-source-path` y `-d` indican en qué directorio buscar los archivos fuente y dónde colocar el `bytecode` generado, en forma equivalente a lo que has hecho con `ant`. Ahora corre tu código con:

```
java -cp classes icc.primitivos.Prueba
```

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	$[-9223372036854775808, 9223372036854775807]$

La opción `-cp` es una abreviatura de `classpath` y le indica a la máquina virtual en qué directorio puede encontrar el programa solicitado, ya compilado.

NOTA: Cuando hayas terminado esta práctica, antes de enviar tu código, no olvides borrar la carpeta `classes`.

7. Para sumarle un número a una variable puedes usar alguna de las alternativas siguientes:

```
// opción 1
int número = 10;
número = número + 15;
// opción 2
int otro = 5;
otro += 9;
// opción 3, para sumar 1
otro++;
// opción 4, para sumar 1
++otro;
```

Ahora intenta sumarle uno a `max`, imprime el resultado en base 10 y su representación en binario.

8. Explica qué pasó con estos números.

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:

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

donde identificamos tres datos:

Tabla 2.3 Flotantes

Nombre	Tamaño	Intervalo
`float`	4 bytes	$[1.4\text{x}10^{-45}, 3.4028235\text{x}10^{38}]$
`double`	8 bytes	$[4.9\text{x}10^{-324}, 1.79\text{x}10^{308}]$

1. **Signo:** $+$
2. **Mantiza:** 1.001101
3. **Exponente:** -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:

- 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 de la base dos, en base dos. El primero 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 `Double.NaN`. Por ejemplo:

```
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:

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

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

Para el caso de los números tipo `float`, necesitas agregar la `f`:

```
float cero = 0.0f;
float mcero = -0.0f;
```

Actividad 2.2

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

Entregable: En tu clase `Prueba.java` imprime las representaciones binarias de `Nan`, `NEGATIVE_INFINITY`, `POSITIVE_INFINITY`, cero y cero negativo para `double` y `float`.

Boolean

El tipo `boolean` es utilizado para operaciones lógicas. Aunque en la lógica booleana sólo hay dos símbolos: $\mathtt{verdadero}$ y $\mathtt{falso}$, 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`.

```
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;
```

Observa el cero 0 al inicio del número. Éste le indica a `Java` que estamos escribiendo en base 8. Si no lo pones, creerá que quieres guardar un 754_{10} y los bits no quedarán colocados correctamente.

Actividad 2.3

Entregable: Dentro del archivo `Prueba.java`, imprime cómo se ve el valor de `permisos` en binario. ¿Cuánto vale en base 10? (Para esto último sólo mándala imprimir, `Java` trabaja por defecto en base 10).

Operadores sobre bits

Consideraremos dos tipos de operadores: unarios y binarios. Los operadores unarios actúan sobre un único valor, pueden colocarse a su izquierda (prefijos) o a su derecha (postfijos). Los operadores binarios infijos se colocan entre dos operandos y devuelven un valor nuevo como resultado de la operación. 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

Tabla 2.5 Tablas para los operadores binarios sobre bits

Valores		&	^	|
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

El efecto de las tres últimas operaciones en la Tabla 2.4 se suele expresar con una tabla que indica el resultado de la operación para cada par de valores posibles de los operandos de un solo bit (Tabla 2.5). Para números compuestos por secuencias de bits la operación se aplica bit por bit, como en el ejemplo siguiente.

Ejemplo 2.1. *Sean los números* 00101010 *y* 11010101, *los resultados se obtienen bit con bit:*

```
  00101010          00101010          00101010
& 11100101        ^ 11100101        | 11100101
------------      ------------      ------------
  00100000          11001111          11101111
```

Actividad 2.4

Realiza ahora pruebas con los operadores de corrimiento <<, >>, >>>. Recorre los números del inciso anterior (la variable `permisos`) por uno y tres bits. Se usa así:

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

Asegúrate de que entiendes lo que hizo cada operador.

Entregable: Dentro del mismo método `main` incluye el código para:

1. Imprimir los valores originales en decimal y binario.
2. Realizar los corrimientos.
3. Imprimir los valores resultantes en decimal y binario.

Actividad 2.5

Ahora toma los permisos de la sección anterior (el valor original de la variable `permisos`): ¿qué operaciones necesitas hacer para que todos los usuarios tengan permiso de escritura, sin modificar sus otros permisos?

Entregable: En el mismo método de tus pruebas anteriores incluye código que:

1. Imprima el valor original de los permisos en binario.
2. Realize las operaciones para agregar los permisos indicados.
3. Imprima el valor modificado, también en binario, éste debe contener ahora los permisos que querías.

Desarrollo

Junto con esta práctica se te entrega el código de la clase `ImpresoraBinario` y el cascarón de la clase `PruebasPrimitivos`. El `build.xml` configura a `ant` para que puedas compilar y ejecutar este archivo como se vio en la práctica anterior, así que ya no necesitarás hacerlo manualmente.

1. 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 una conversión de tipos (*casting*) para guardar el número en los tipos correspondientes a enteros y perderás la parte fraccionaria. Una conversión de tipos primitivos obliga a `Java` a convertir un tipo primitivo en otro, incluso si se pierde información en el proceso. Para realizar una conversión se coloca el nombre del tipo de deseado entre paréntesis a la izquierda del valor a convertir, como en el ejemplo siguiente:

```
1 int x = (int)-456.601;
```

   ¿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 (comenzando por cero y contado de derecha a izquierda, es decir 0...011110000). ¿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 `^`. Calcula `~num`.

Entregables

Las actividades con resultados que deberás entregar son:

1. 2.1
2. 2.2 y 2.3
3. 2.4 y 2.5

Adicionalmente, debes completar los ejercicios de la sección anterior.