Tipos de datos de Java

Parte 2

1. Representación de datos

Es importante que los estudiantes de **Programación II** comprendan que **la elección del tipo de dato no es solo un aspecto técnico, sino una *decisión de diseño* basada en la magnitud de la realidad que intentan modelar**.

Para representar valores extremos, Java nos ofrece tipos específicos. Aquí tiene una propuesta de ejemplos que puede comprobar usted mismo:

1.1. Valores Extremadamente Altos

Ejemplo: **Distancias Astronómicas**

Para distancias como la que separa a la Tierra del Sol, el tipo `int` (que llega hasta $2{,}1 \times 10^9$) se queda corto si usamos metros. Aquí es donde entra el `long`.

- **Distancia Tierra-Sol**

  Aproximadamente 150,000,000 km.

  - Si se expresa en metros, el valor es $150,000,000,000$. Esto supera el límite de un `int`.
  - Tipo recomendado: `long`

- **Año Luz**

  Aproximadamente $9{,}46 \times 10^{12}$ km.

  - Tipo recomendado: `double` (para notación científica) o `long` (si se requiere precisión entera exacta en kilómetros).

```
// Ejemplo en Java
long distanciaSolMetros = 150000000000L; // La 'L' es vital para definir el literal long
double añoLuzKm = 9.461e12;
```

1.2. Valores Extremadamente Bajos

- **Escala Atómica**

  Cuando hablamos de la distancia entre átomos, entramos en el terreno de los *nanómetros* o *picómetros*. Aquí la precisión decimal es la clave.

- **Distancia entre átomos de Hidrógeno en el agua**

  Aproximadamente 0.000000000095 metros (95 picómetros).

  - Tipo recomendado: `double`. Aunque existe `float`, el `double` es el estándar en Java para cálculos científicos por su precisión de 64 bits.

```
// Ejemplo en Java
double distanciaAtomosMetros = 0.000000000095;
// O usando notación científica, que es más legible para los alumnos:
double distanciaAtomosCientifica = 9.5e-11;
```

Magnitud	Ejemplo Real	Valor Aproximado	Tipo de Dato Java
Entero Grande	Población Mundial	8,000'000,000	`long`
Distancia Espacial	Tierra a Marte (m)	$2{,}25 \times 10^{11}$	`long` o `double`
Escala Micro	Diámetro de un ADN	0.000000002 m	`double`
Escala Atómica	Radio del Átomo	0.00000000005 m	`double`

Tip:

Si bien el `double` es genial para la ciencia, nunca deben usarlo para dinero (centavos), ya que la aritmética de punto flotante puede perder precisión. Para eso, Java nos ofrece el `BigDecimal`.

2. El problema del trigo y el tablero de ajedrez

En un tablero de ajedrez, de 64 casillas, se ha de colocar un grano de trigo en la primera casilla, esto es 2^0, en la segunda casilla, se ha de colocar 2 granos de trigo, esto es 2^1. El valor de la casilla *n* es 2^{n-1}.

2.1. Comportamiento matemático

La progresión es una potencia de 2. El valor de la casilla n es 2^{n-1}.

Casilla 1: $2^0 = 1$

Casilla 2: $2^1 = 2$

...

Casilla 10: $2^9 = 512$

...

Casilla 31: $2^{30} = 1,073,741,824$ (Último valor positivo que cabe en un `int` de 32 bits, que llega hasta $2{,}1 \times 10^9$).

2.2. El *Desbordamiento*/*Overflow*

En la casilla 32, el valor teórico es $2^{31} = 2,147,483,648$.

En Java, el tipo `int` es un entero con signo de 32 bits. Su valor máximo es $2^{31} - 1$.

Resultado en Java

Al intentar asignar 2^{31} a un `int`, el valor *da la vuelta* (*overflow*) y se convierte en **-2,147,483,648**. Ver un número negativo donde debería haber un crecimiento gigante es una clara señal de que algo no está bien.

2.3. ¿Qué pasa en la Casilla 64?

El valor de la última casilla es 2^{63}.

Un `long` en Java tiene 64 bits y su valor máximo es $2^{63} - 1$.

Por lo tanto, la casilla 64 por sí sola causaría un `overflow` incluso en un `long` (se volvería negativa).

Casilla	Valor Real	¿Cabe en `int`?	¿Cabe en `long`?
1	1	Sí	Sí
31	1,073'741,824	Sí	Sí
32	2,147'483,648	**No** (Overflow)	Sí
63	$4{,}6110^{18}$	No	Sí
64	$9{,}2210^{18}$	No	**No** (Límite exacto)

2.4. La Solución en Java

Para problemas de este tipo, tenemos la clase **`BigInteger`**. A diferencia de los tipos primitivos, `BigInteger` no tiene un límite fijo de bits (está limitado solo por la memoria disponible del sistema).

```
import java.math.BigInteger;

public class EjemploAjedrez {
    public static void main(String[] args) {
        BigInteger granos = BigInteger.ONE;
        for (int i = 1; i <= 64; i++) {
            System.out.println("Casilla " + i + ": " + granos);
            granos = granos.multiply(BigInteger.valueOf(2));
        }
    }
}
```

Dato: El total de granos en el tablero sería $2^{64} - 1$.

¡Eso es más de 18 trillones de granos! Se dice que es más de lo que la humanidad ha producido en toda su historia.

3. Efecto Cuentakilómetros

Un tipo de dato primitivo funciona como el cuentakilómetros/*odómetro* de un auto viejo: cuando llega al máximo de su capacidad, no se detiene, sino que **da la vuelta** y comienza desde el valor más bajo posible (en este caso, el extremo negativo).

3.1. Justificación Técnica

La **trampa** del bit de signo:

Java usa el sistema de **Complemento a 2**. El bit más a la izquierda define si el número es positivo (0) o negativo (1). En la casilla 32 del ejemplo del tablero de ajedrez, el valor es exactamente 2^{31}. **Ese bit de signo cambia a 1**, y de pronto, un programa que contaba granos de trigo empieza a mostrar una deuda astronómica de granos.

3.2. Costo de Memoria vs. Precisión

Si bien la memoria RAM **sobra**, en sistemas de alto rendimiento o bases de datos con miles de millones de registros (como transacciones bancarias o *logs* de servidores), elegir un `long` cuando un `int` basta duplica el consumo de almacenamiento innecesariamente.

3.3. Tipos de Datos *Silenciosos*

A diferencia de otros errores de programación, **el desbordamiento de tipos primitivos no lanza una excepción** (`Exception`) en tiempo de ejecución. El programa sigue corriendo, pero con datos corruptos. Este es el riesgo más alto en ingeniería de software.

4. Saltar de un tipo de dato primitivo a otro

Tres aspectos críticos a tener en cuenta para saltar de un tipo a otro según el dominio del problema (financiero, científico, gestión)?

1. **El Silencio del Error** (*Silent Failure*)

   En Java, **el desbordamiento de un tipo primitivo no detiene el programa**. No hay un mensaje de error rojo. El sistema simplemente sigue calculando con valores falsos. Esto enseña que es el programador el responsable de validar los rangos, no el lenguaje.

2. **El Techo de Cristal de los 32 bits**

   Al llegar a la casilla 32, estamos físicamente el límite de la arquitectura de muchos sistemas antiguos o estándares. Es por ello qué se tuvo que migrar de sistemas de 32 bits a 64 bits: ¡simplemente nos quedamos sin *espacio* para contar cosas grandes!

3. **El Costo de la Decisión**

   Planteamos este dilema:

   Si usamoss `long` para todo *por si acaso*, **gastamos el doble de memoria** (64 bits vs 32 bits).

   En un arreglo de 64 posiciones no importa, pero ¿y si es una base de datos de millones de usuarios en un Workshop? Allí, esa *indiferencia* se traduce en **costos de servidor** y **lentitud**.

Un buen programador conoce las reglas del lenguaje; un buen ingeniero conoce los límites de la máquina.

Los principales prefijos métricos

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	,000,000,000,000,000,000,000	Yotta