BigDecimal

BigDecimal representa un número decimal con signo de precisión arbitraria inmutable. Está formado por dos partes:

Valor sin escalar: un entero de precisión arbitraria.

Escala: un número entero de 32 bits que representa el número de dígitos a la derecha del punto decimal.

Por ejemplo, BigDecimal 3.14 tiene el valor sin escalar de 314 y la escala de 2.

* Usamos BigDecimal para aritmética de alta precisión.
* También lo usamos para cálculos que requieren control sobre la escala y el comportamiento de redondeo.

Un ejemplo de ello son los cálculos que involucran transacciones financieras.

Podemos crear un objeto BigDecimal a partir de String, de un array de caracteres, int, long y BigInteger:

|  |
| --- |
| BigDecimal bdFromString = **new** BigDecimal("0.1");  BigDecimal bdFromCharArray = **new** BigDecimal(**new** **char**[] {'3','.','1','6','1','5'});  BigDecimal bdlFromInt = **new** BigDecimal(42);  BigDecimal bdFromLong = **new** BigDecimal(123412345678901L);  BigInteger bigInteger = BigInteger.probablePrime(100, **new** Random());  BigDecimal bdFromBigInteger = **new** BigDecimal(bigInteger); |

También se puede crear desde un tipo double:

BigDecimal bdFromDouble = **new** BigDecimal(0.1d);

Sin embargo, el resultado, en este caso, es diferente al esperado (es decir, 0,1). Debido a que:

* el constructor doble hace una traducción exacta
* 0.1 no tiene una representación exacta en doble

****

Por lo tanto, deberíamos usar el constructor String en lugar del constructor doble.

Además, podemos convertir double y long a BigDecimal usando el método estático valueOf que convierte double a String antes de convertirlo a BigDecimal:

|  |
| --- |
| BigDecimal bdFromLong1 = BigDecimal.valueOf(123412345678901L);  BigDecimal bdFromLong2 = BigDecimal.valueOf(123412345678901L, 2);  BigDecimal bdFromDouble = BigDecimal.valueOf(0.1d); |

Regla Práctica BigDecimal:

|  |
| --- |
| Utiliza preferiblemente el método valueOf frente al constructor de la clase BigDecimal para double |

De un BigDecimal puedes obtener la precisión, la escala y el signo mediante métodos.

Escala = numero de decimales

|  |
| --- |
| BigDecimal bd = **new** BigDecimal("-12345.6789");    assertEquals(9, bd.precision());  assertEquals(4, bd.scale());  assertEquals(-1, bd.signum()); |

Regla Práctica BigDecimal:

|  |
| --- |
| Las comparaciones se realizan siempre mediante el método compareTo (NO UTILIZAR equals!!) |

|  |
| --- |
| BigDecimal bd1 = **new** BigDecimal("1.0");  BigDecimal bd2 = **new** BigDecimal("1.00");  BigDecimal bd3 = **new** BigDecimal("2.0");   assertTrue(bd1.compareTo(bd3) < 0);  assertTrue(bd3.compareTo(bd1) > 0);  assertTrue(bd1.compareTo(bd2) == 0);  assertTrue(bd1.compareTo(bd3) <= 0);  assertTrue(bd1.compareTo(bd2) >= 0);  assertTrue(bd1.compareTo(bd3) != 0); |

Redondeo

Al redondear un número, lo reemplazamos por otro que tenga una representación más corta, más simple y más significativa. Por ejemplo, redondeamos 24.784917€ a 24.78€ porque no tenemos fracciones de céntimo.

El modo de precisión y redondeo a utilizar varía según el cálculo. Por ejemplo, en las declaraciones de impuestos y las facturas se especifica redondear los montantes usando HALF\_UP.

Hay dos clases que controlan el comportamiento de redondeo: RoundingMode y MathContext.

El enum RoundingMode proporciona ocho modos de redondeo:

*CEILING:*  redondea hacia el infinito positivo

*FLOOR* : redondea hacia el infinito negativo

*UP:* redondea desde cero

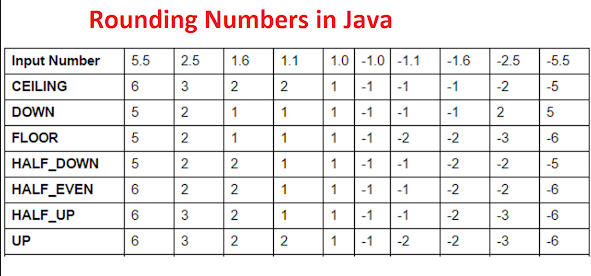
*DOWN:* redondea hacia cero

HALF\_UP: redondea hacia el "vecino más cercano" a menos que ambos vecinos sean equidistantes, en cuyo caso se redondea hacia arriba

HALF\_DOWN: redondea hacia el "vecino más cercano" a menos que ambos vecinos sean equidistantes, en cuyo caso se redondea hacia abajo

HALF\_EVEN: redondea hacia el "vecino más cercano" a menos que ambos vecinos sean equidistantes, en cuyo caso, redondea hacia par

*UNNECESSARY* : no es necesario redondear y se lanza ArithmeticException si no es posible un resultado exacto.



El modo de redondeo HALF\_EVEN minimiza el sesgo debido a las operaciones de redondeo. Se usa con frecuencia. También se conoce como el redondeo bancario.

MathContext encapsula tanto el modo de precisión como el de redondeo. Hay algunos MathContexts predefinidos:

DECIMAL32 – Precisión de 7 dígitos y un modo de redondeo de HALF\_EVEN

DECIMAL64: precisión de 16 dígitos y un modo de redondeo de HALF\_EVEN

DECIMAL128: precisión de 34 dígitos y un modo de redondeo de HALF\_EVEN

*UNLIMITED* : aritmética de precisión ilimitada

Ejemplo 1:

BigDecimal resultado = numero1.divide(numero2, 2, RoundingMode.HALF\_UP); // Divide dos BigDecimal con una precisión o escala de 2 decimales y redondeo hacia arriba

Ejemplo de cálculo de total:

|  |
| --- |
| **public** **static** BigDecimal **calculateTotalAmount**(BigDecimal quantity,BigDecimal  unitPrice, BigDecimal discountRate, BigDecimal taxRate) {  BigDecimal amount = quantity.multiply(unitPrice);  BigDecimal discount = amount.multiply(discountRate);  BigDecimal discountedAmount = amount.subtract(discount);  BigDecimal tax = discountedAmount.multiply(taxRate);  BigDecimal total = discountedAmount.add(tax);   *// 2 decimales HALF\_EVEN*  BigDecimal roundedTotal = total.setScale(2, RoundingMode.HALF\_EVEN);    **return** roundedTotal; } |
|  |

SIEMPRE:

* Evita el constructor double

|  |
| --- |
| BigDecimal x = **new** BigDecimal(0.1); System.out.println("x=" + x); |

|  |
| --- |
| x=0.1000000000000000055511151231257827021181583404541015625 |

* Trabaja con el constructor de String en vez valueOf de double

|  |
| --- |
| BigDecimal x = BigDecimal.valueOf(1.01234567890123456789); BigDecimal y = **new** BigDecimal("1.01234567890123456789"); System.out.println("x=" + x); System.out.println("y=" + y); |

|  |
| --- |
| x=1.0123456789012346 // esto es un double 32bytes y=1.01234567890123456789// esto es de un String 64bytes |

El valor x ha perdido cuatro dígitos decimales porque un doble tiene una precisión de solo 15 a 17 dígitos (un flotante tiene una precisión de solo 6 a 9 dígitos), mientras que BigDecimal tiene una precisión arbitraria (limitada solo por la memoria).

* Utiliza compareTo para comparar (evita equals!!)

|  |
| --- |
| BigDecimal x = **new** BigDecimal("1"); BigDecimal y = **new** BigDecimal("1.0"); System.out.println(x.equals(y));  Aunque 1 y 1.0 son iguales la comparación con equals y bigDecimal lanza |

|  |
| --- |
| False |

* Utiliza setScale(2, RoundingMode.HALF\_UP) siempre
* mejor que round(new MathContext(2, RoundingMode.HALF\_UP))

|  |
| --- |
| BigDecimal x = **new** BigDecimal("12345.6789"); x = x.round(**new** MathContext(2, RoundingMode.HALF\_UP)); System.out.println("x=" + x.toPlainString()); System.out.println("scale=" + x.scale()); |

|  |
| --- |
| x=12000  scale=-3 |

|  |
| --- |
| BigDecimal x = **new** BigDecimal("12345.6789"); x = x.setScale(2, RoundingMode.HALF\_UP); System.out.println("x=" + x)); |

|  |
| --- |
| x=12345.68 |

Ejercicio 1. Factorial n!= 1· 2 · 3 … · n con el tipo BigInteger.

Ejercicio 2. Interés compuesto con BigDecimal.

Una persona invierte un ‘monto principal’ en una cuenta de ahorro que produce un % de interés dado.

Suponiendo que todo el interés se deposita en la cuenta, calcula y muestra el monto de dinero en la cuenta al final de ‘n’ años. Use la siguiente fórmula para determinar los montos anuales:

c = p(1 +r)^n

en donde

p es el monto que se invirtió originalmente (es decir, el monto principal)

r es la tasa de interés anual (por ejemplo, use 0.05 para el 5%)

n es el número de años

c es la cantidad depositada al final del año enésimo.

Ejercicio 3. Implementa el cálculo de la distancia entre 2 puntos en la tierra mediante la fórmula de Haversine, basándote en la implementación realizada en C# <https://www.genbeta.com/desarrollo/como-calcular-la-distancia-entre-dos-puntos-geograficos-en-c-formula-de-haversine>.

Para ello, tienes que portar esta implementación de C# a Java, utiliza BigDecimal para los cálculos en vez de precisión en punto flotante, crea un test de prueba de funcionamiento.

Ejercicio 4. Implementa en java el siguiente diagrama de clases UML sobre una tienda online, toma las decisiones de diseño que consideres oportunas.

¿Con qué tipo de Java modelamos el tipo UML Real para una aplicación con cálculos financieros/monetarios?

Crea los siguientes tests:

* Crear varios Web User, Customer, Account.
* Crear varios Product.
* Añadir varios Product a ShoppingCart de Web User y Account existente.
* Crear un Order y Payment asociado a un ShoppingCart existente.

