

OBJETOS BIGDECIMAL

Podemos decir que son una suerte de BigInteger con capacidad para memorizar "posiciones decimales". Sirven luego para representar números con un número fijo de decimales, lo que en los lenguajes de programación se llamaban números de coma fija (punto fijo en inglés). Por tanto, además de poder representar enteros gigantes, como en el ejemplo anterior, podemos trabajar con parte decimal, pero sin la falta de control sobre las imprecisiones de float y double. Con BigDecimal las imprecisiones inevitablemente también existe, por ejemplo es imposible representar infinitos decimales, pero la imprecisión puede estar bajo control del programador. Si escribieras una aplicación bancaria real, utilizarías BigDecimal, no float ni double.

IMPRECISIÓN DE COMA FLOTANTE

Hay múltiples ejemplos de los problemas de precisión de double. Son ejemplos "chocantes" y difíciles de entender porque nosotros razonamos en base 10 pero internamente el número está representado en base 2. Especificamos en nuestro código un número en base 10 y el compilador debe traducirlo a la representación binaria de coma flotante. Esta traducción a veces no es "exacta" y el porqué de esta inexactitud se sale de nuestro alcance ya que tendríamos que estudiar detenidamente la representación interna de coma flotante.

Ejemplo de imprecisión en double

```
class Unidad2 {  
    public static void main(String[] args) {  
        double unCentimo = 0.01;  
        double suma6Centimos = unCentimo + unCentimo + unCentimo + unCentimo + unCentimo + unCentimo;  
        System.out.println(suma6Centimos);  
    }  
}
```

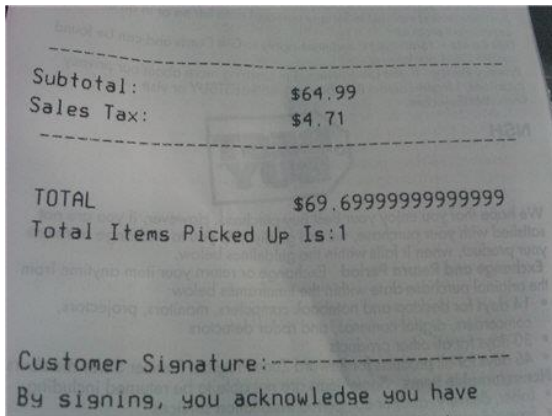
0.060000000000000005

¡Esperábamos 0.06 exacto!

si sumas de la forma anterior 7 y 8 Centimos da O.K., si sumas 9 de nuevo sale un resultado aproximado....No nos paramos en el porqué de esto ya que implica conocer en profundidad la representación interna de double, lo que se sale de nuestro alcance.

En muchas situaciones esta imprecisión no tiene la mayor importancia. Por ejemplo, si me debes 6 céntimos y yo te pido 0.060000000000000005€ seguro que no te enfadas, pero, si en una página web de compras haciendo una transacción esperamos 0.06 y nos aparece 0.060000000000000005 podemos desconfiar de que algo no va bien y anular la compra, igual que nos daría reparo y desconfianza comprar un burro en una web cuyo link principal fuera "Benta de vurros varatos" (impresionan las faltas de ortografía) .

Además, en las transacciones comerciales una ristra de números decimales causa mala impresión, observa esta foto de una factura real



intragable!

Podemos trabajar con double/float y formatear la salida enseñando por ejemplo sólo 2 decimales, también tenemos posibilidades de aplicar redondeos matemáticos, pero todo esto complica muchísimo el código, sencillamente: para los cálculos comerciales no se adapta bien la aritmética en coma flotante.

Veamos el ejemplo anterior sustituyendo a double por BigDecimal

```
import java.math.BigDecimal;
public class Unidad2 {
    public static void main(String[] args) {
        BigDecimal unCentimo = new BigDecimal("0.01");
        BigDecimal
suma6Centimos=unCentimo.add(unCentimo).add(unCentimo).add(unCentimo).add(unCentimo).add(unCentimo);
        System.out.println(suma6Centimos);
    }
}
```

0.06

Esto es una demostración para intuir que con BigDecimal puedo obtener el valor esperado.

BigDecimal "avisa" al programador ante situaciones especiales

un ejemplo: si al operar se generan infinitos decimales

```
import java.math.BigDecimal;
public class Unidad2 {
    public static void main(String[] args) {
        BigDecimal uno = new BigDecimal("1");
        BigDecimal tres = new BigDecimal("3");
        BigDecimal unTercio=uno.divide(tres);
        System.out.println(unTercio);
    }
}
```

run:

Exception in thread "main" java.lang.ArithmeticException: Non-terminating decimal expansion; no exact representable decimal result.

at java.math.BigDecimal.divide(BigDecimal.java:1616)
at Unidad2.main(Unidad2.java:7)

Es decir, como no se puede representar con decimales de forma exacta el resultado de $\frac{1}{3}$ java genera una excepción. Tienes que entender que es imposible representar algo infinito con algo finito(sin redondeos). Más adelante sabremos manejar las excepciones y tomar el control de la situación. Ahora simplemente, observa que como BigDecimal no puede trabajar con exactitud “avisa al programador”.

Crear objetos BigDecimal

Si consultas el API verás un montón de constructores. La forma más básica es a partir de un String y no a partir de un double ya que con double el BigDecimal tiene en cuenta la representación interna de ese double y la pasa directamente a un String. Puedes consultar al respecto

<https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html#BigDecimal-double->

Aquí nos comenta que curiosamente la constante double 0.1 no se puede representar exactamente con punto flotante. En el ejemplo también añadimos el println() de 1.0 y vemos que aparentemente en d se almacena 0.1, pero realmente no es así si no que él println() nos muestra 0.1 debido al funcionamiento del propio println()

```
import java.math.BigDecimal;
public class Unidad2 {
    public static void main(String[] args) {
        BigDecimal bd1 = new BigDecimal("0.1");
        BigDecimal bd2= new BigDecimal(0.1);
        System.out.println(bd1.toString());
        System.out.println(bd2.toString());
        double d=0.1;
        System.out.println(d);
    }
}
SALIDA
0.1
0.1000000000000000055511151231257827021181583404541015625
0.1
```

MÁS RAZONES PARA USAR BIGDECIMAL.

Ya vimos que son más precisos que double/float en ciertos contextos (ver ejemplos anteriores).

También hay que recordar que double/float son tipos primitivos y por tanto mucho más eficientes y que la mayoría de las imprecisiones que aportan suelen ser poco relevantes según el contexto.. Por otro lado los BiGDecimal se prefieren en ciertos contextos de trabajo, como los procesos financieros por dos razones:

1. Lo BigDecimal tiene la habilidad de poder especificar la escala, es decir, el número de dígitos significativos tras la coma decimal.
2. Lo BigDecimal tiene la habilidad de especificar un método de redondeo.

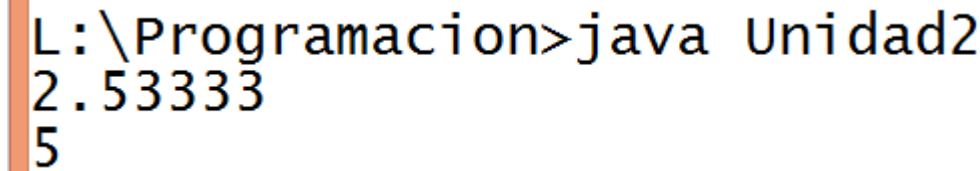
Ambas habilidades redundan en que aunque hay imprecisiones, están bajo control del programador de una forma cómoda, sii usamos double tenemos utilidades de formateo en pantalla y de redondeo, pero da mucho trabajo extra al programador, el camino fácil es usar BigDecimal

Escala

Escala=>número de decimales significativos

Al crear un BigDecimal desde un String, la escala se deduce del String

```
import java.math.BigDecimal;
public class Unidad2 {
    public static void main(String[] args) {
        BigDecimal a;
        a = new BigDecimal("2.53333");
        System.out.println(a);
        System.out.println(a.scale());
    }
}
```



consulta el método scale() en el API y observa que devuelve la escala del número sobre el que se invoca.

Los objetos BigInteger y BigDecimal son inmutables, igual que los objetos String. El concepto de inmutabilidad se estudia un poco más adelante, por el momento inmutable quiere decir que un objeto que una vez creado no se puede modificar, u otra forma de verlo, es un objeto que una vez creado es de sólo lectura. El "truco" para modificarlo es crear uno nuevo a partir del viejo con las modificaciones

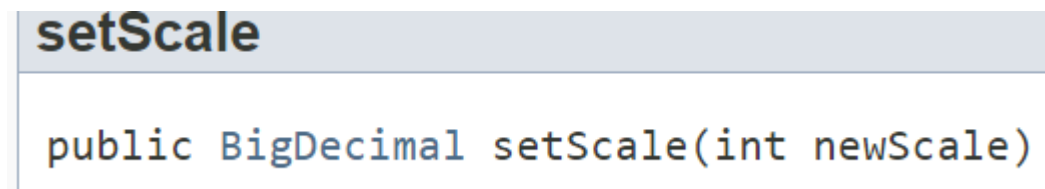
modificar la escala de un objeto BigDecimal

Para modificar la escala utilizamos setScale().

Con el siguiente código no estoy cambiando la escala de a, ya que el objeto referenciado por a es inmutable

```
import java.math.BigDecimal;
public class Unidad2 {
    public static void main(String[] args) {
        BigDecimal a;
        a = new BigDecimal("2.53");
        a.setScale(4);
        System.out.println(a);
        System.out.println(a.scale());
    }
}
```

Observa que setScale() realmente devuelve un "nuevo" BigDecimal con la escala actualizada



Así sí que cambiamos la escala

```
a=a.setScale(4);
```

Realmente a.setScale(4); devuelve un nuevo BigDecimal y finalmente a se engancha a ese nuevo objeto, hay cierto paralelismo a cuando trabajamos con tipos primitivos:

```
int a=7;
a=a+3;
```

```
import java.math.BigDecimal;
public class Unidad2 {
    public static void main(String[] args) {
        BigDecimal a;
        a = new BigDecimal("2.53");
        a=a.setScale(4);
        System.out.println(a);
        System.out.println(a.scale());
    }
}
```

Redondeo

Recordemos que “lo bueno” de BigDecimal es que permite al programador tener bajo su control la imprecisión inevitable al representar valores reales. Este control se hace con la combinación de escala y redondeo. Cuando creamos un BigDecimal desde un String se deduce del literal una escala, pero no se puede deducir redondeo. El redondeo es un parámetro que hace falta para prácticamente cualquier operación con BigDecimal.

Por ejemplo si quiero cambiar de scale un BigDecimal a un número con menor scala tengo que decirle a BigDecimal como quiero hacer el redondeo.

Lo siguiente da error:

```
import java.math.BigDecimal;
public class Unidad2 {
    public static void main(String[] args) {
        BigDecimal a;
        a = new BigDecimal("2.5333");
        a=a.setScale(2);
        System.out.println(a);
    }
}
```

¿Con número de dos decimales se queda?, es decir, ¿cómo redondea?

El redondeo se puede indicar también en setScale()

```
import java.math.BigDecimal;
public class Unidad2 {
    public static void main(String[] args) {
        BigDecimal a = new BigDecimal("2.5333");
        a=a.setScale(2,BigDecimal.ROUND_DOWN);
        System.out.println(a);
        BigDecimal b = new BigDecimal("2.5333");
        b=b.setScale(2,BigDecimal.ROUND_UP);
        System.out.println(b);
        //y hay más tipos de redondeo ...
    }
}
```

Operaciones aritmética e inmutabilidad.

Cuando hacemos operaciones con números BigDecimal llegamos a las mismas consideraciones que para setScale(), por ejemplo el método add se ejecuta sobre un objeto pero no lo modifica, lo que hace add() es devolver un BigDecimal nuevo. En el siguiente ejemplo add() no modifica unoYPico si no que crea un nuevo objeto BigDecimal, con scala 3 que deduce del contexto ya que hay un operando con 3 decimales. El nuevo objeto BigDecimal pasa a ser referenciado por suma.

```
import java.math.BigDecimal;
public class Unidad2 {
    public static void main(String[] args) {
        BigDecimal unoYPico = new BigDecimal("1.244");
        BigDecimal dosYPico = new BigDecimal("2.11");
```

```

BigDecimal suma= unoYPico.add(dosYPico);

System.out.println("unoYPico: "+unoYPico);
System.out.println("dosYPico: "+dosYPico);
System.out.println("suma de unoYPico y dosYPico: "+suma);

suma=suma.setScale(2,BigDecimal.ROUND_UP);
System.out.println("el valor anterior con escala 2 y ROUND_UP:"+suma);
}
}

```

Operaciones aritmética y escala.

Algunas operaciones aritméticas pueden deducir automáticamente la escala que le corresponde al resultado

```

import java.math.BigDecimal;
class Unidad2{
    public static void main(String[] args) {
        BigDecimal a = new BigDecimal("1.1");
        BigDecimal b = new BigDecimal("2.2");
        BigDecimal c= new BigDecimal("3.33");
        BigDecimal sumaAyB=a.add(b);
        BigDecimal sumaAyC=a.add(c);
        BigDecimal multiplicaAyB=a.multiply(b);
        BigDecimal multiplicaAyC=a.multiply(c);
        System.out.println("sumaAyB: "+sumaAyB);
        System.out.println("sumaAyC: "+sumaAyC);
        System.out.println("multiplicaAyB: "+multiplicaAyB);
        System.out.println("multiplicaAyC: "+multiplicaAyC);
    }
}

```

Es importante que analices la salida del programa anterior y “te salgan las cuentas”, al respecto de la escala que tienen los números resultado. Debes consultar el API de BigDecimal y encontrar en add() y multiply() la justificación de los resultados

Ejercicio 1

Con numero1 y numero2

```

BigDecimal numero1 = new BigDecimal("10.7");
BigDecimal numero2 = new BigDecimal("5.4");

```

Realiza las siguientes operaciones:

- numero1+numero2
- numero1-numero2
- numero1*numero2
- numero1/numero2

Consulta el API de BigDecimal para descubrir los métodos que te permiten hacer las operaciones anteriores.

La operación de división genera una excepción, intenta razonar por qué y consultando el API escoge la versión del método que divide que te permite solucionar el problema.

Ejercicio 2:

Utiliza ahora en el ejemplo anterior double en el parámetro del constructor

```

BigDecimal numero1 = new BigDecimal(10.7);
BigDecimal numero2 = new BigDecimal(5.4);

```

y explica la nueva salida por pantalla aunque el resto del código sea el mismo

Ejercicio 3: comprueba la salida terrorífica del siguiente código y obtén la salida "esperada" usando BigDecimal en lugar de double.

```
class Unidad2 {  
    public static void main(String args[]) {  
        System.out.println(2.00 - 1.1);  
        System.out.println(2.00 - 1.2);  
        System.out.println(2.00 - 1.3);  
        System.out.println(2.00 - 1.4);  
        System.out.println(2.00 - 1.5);  
        System.out.println(2.00 - 1.6);  
        System.out.println(2.00 - 1.7);  
        System.out.println(2.00 - 1.8);  
        System.out.println(2.00 - 1.9);  
        System.out.println(2.00 - 2);  
    }  
}
```

Ejercicio 4:

Leemos del teclado el precio de un artículo con nextBigDecimal(consulta API).

Leemos del teclado el impuesto que queremos aplicarle, también con nextBigDecimal calculamos el pvp con métodos bigdecimal de forma traducimos a "lenguaje BigDecimal"

$pvp = precio + precio * impuesto / 100$

ejemplo de posible salida

```
L:\Programacion>java Unidad2  
precio:  
10.55  
impuesto:  
10.0  
pvp: 11.605
```

Ejercicio 5:

Repita el ejercicio anterior sin nextBigDecimal().

Ejercicio 6:

Repetimos el ejercicio anterior pero ahora con la siguiente consideración.

Suponemos que trabajamos en un contexto financiero, por ejemplo un banco, que se rige por las siguiente norma: siempre se trabaja con 2 decimales y si hay que hacer redondeo siempre usamos Round_Up, de forma tal que cada operación(cada suma, cada multiplicación, ...) que se hace se le aplica redondeo para quedarnos con dos decimales.
un ejemplo de ejecución

```
L:\Programacion>java Unidad2  
precio en euros:  
10.542  
impuesto en %:  
10.0  
precio redondeado a dos decimales: 10.55  
impuesto en euros sobre precio: 1.06  
pvp: 11.61
```