OBJETOS BIGDECIMAL

Podemos decir que son una suerte de BigInteger con capacidad para memorizar "posiciones decimales". Sirven por tanto para representar números con un número fijo de decimales, lo que en lenguajes de programación más antiguos se llamaban números de coma fija(punto fijo en inglés). Por tanto, además de poder representar enteros gigantes, como en el boletín 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 existen, por ejemplo es imposible representar infinitos decimales y hay que redondear un número periódico, pero la imprecisión puede estar bajo control del programador. Si escribieras una aplicación bancaria o de facturación real, utilizarías BigDecimal, no float ni double. Pero si escribieras una aplicación que requiere cálculos científicos usarías double y float.

ARITMÉTICA DE COMA(PUNTO) FIJA

nota:como el punto decimal inglés se corresponde con la coma decimal europea, se dice indistintamente:

- aritmética de coma flotante o aritmética de punto flotante.
- aritmética de coma fija o aritmética de punto fijo .

La aritmética de punto flotante se refiere a utilizar los complicados algoritmos a nivel procesador para representar números reales con bits. Se emplea en cálculos científicos

La aritmética de punto fijo, mucho más simple, coincide con el método clásico de trabajar con un número fijo de decimales e ir haciendo redondeos. Se emplea en contextos financieros y comerciales y desde java para trabajar con esta aritmética tenemos la inestimable ayuda de los objetos BigDecimal.

LA IMPRECISIÓN DE COMA FLOTANTE

Hay múltiples ejemplos sobre la imprecisión de double/float. 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

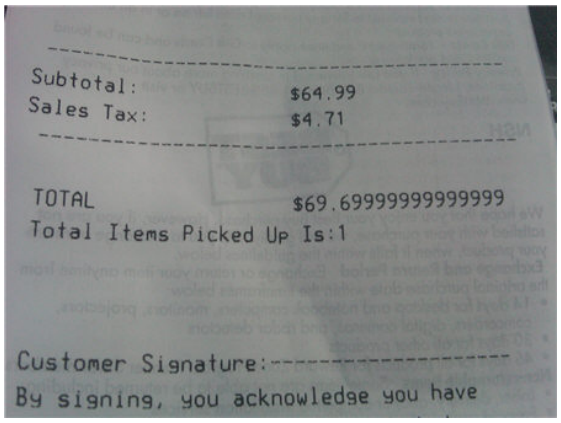
```
class Unidad2 {
   public static void main(String[] args) {
     double unCentimo = 0.01;
     double suma6Centimos=unCentimo+unCentimo+unCentimo+unCentimo+unCentimo+unCentimo;
     System.out.println(suma6Centimos);
   }
}
```

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….

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



¡queda fatal!

LA IMPRECISIÓN DE BIGDECIMAL
BigDecimal también es inevitablemente impreciso, por ejemplo, si divides 1 / 3 el resultado tiene infinitos decimales y el resultado hay que redondearlo de alguna forma.
La diferencia es que la imprecisión usando la aritmética de punto flotante es más difícil de controlar por el programador ya que las imprecisiones ocurren en la traducción de decimal a binario. En cambio con BigDecimal, el programador razona en decimal, tendrá que redondear, pero todo ocurre bajo su control.

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

En este ejemplo no hay imprecisiones y parece por tanto que es mucho mejor usar BigDecimal que trabajar con double pero esto no es así necesariamente ya que con BigDecimal también tendremos que hacer redondeos constantemente. Recuerda:

- aritmética de coma fija: para contextos comerciales y financieros. De hecho bigdecimal nos permite imitar las cuentas que se hacían antiguamente a mano en tiendas, bancos etc.
- aritmética de punto flotante: para cálculos científicos. Se pierde un poco de control en el paso de decimal a binario pero los científicos están acostumbrados a trabajar con el concepto de "error" en las medidas y los resultados y tenemos la ventaja que la velocidad a la que se hacen internamente los cálculos es muy superior a la aritmética de coma fija.

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 el resultado de 1 / 3 no se puede representar con decimales de forma exacta, 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 sabe cómo redondear "avisa al programador".

Crear objetos BigDecimal, mejor con parámetro String

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 como double 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());

   }
}
```

SALIDA

```
0.1
0.1000000000000000055511151231257827021181583404541015625
```

LA ESCALA Y REDONDEO DE UN BIGDECIMAL

Los BiGDecimal es utilizado en ciertos contextos de trabajo, como los procesos financieros por dos razones:

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

Ambas habilidades redundan en que aunque hay imprecisiones, están bajo el control del programador de una forma cómoda para cálculos financieros. Si 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());
   }
}
```

```
L:\Programacion>java Unidad2
2.53333
5
```

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

Los objetos BigDecimal son inmutables

Ya vimos que los BigInteger son objetos inmutables. Los objetos BigDecimal también son inmutables. Como ya vimos con BigInteger, el "truco" para simular que un objeto inmutable se modifica, es crear uno nuevo a partir del viejo con las modificaciones que deseemos y reasignarlo a la variable referencia que referenciaba al objeto "viejo", de forma parecida a como se hace constantemente con objetos Strings que también son inmutables.

```
import java.math.BigDecimal;
```

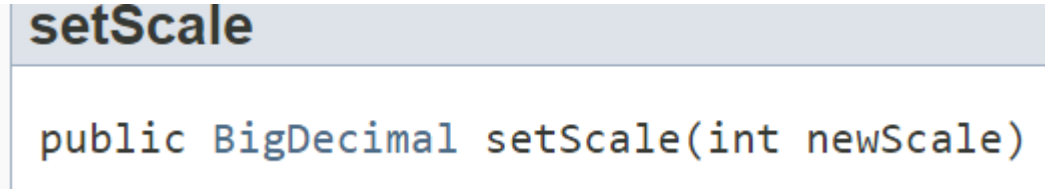
```
class Unidad2 {
    public static void main(String[] args) {

       String nombre="Sonia";
       nombre=nombre+ " Vázquez"; //nombre referencia a un nuevo objeto
        System.out.println(nombre);

       BigDecimal x= new BigDecimal("9");
       x=x.add(new BigDecimal("1"));
       System.out.println(x);
   }
}
```

Modificar la escala de un objeto BigDecimal con setScale()

Si los objetos BigDecimal son inmutables, de forma directa no se les puede modificar la escala, lo demostramos:

```
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());
   }
}
```

Cuando vemos en un método la palabra "set" sabemos que va a modificar algo, pero al trabajar con objetos inmutables pese al nombre "set" realmente no se modifica el objeto original si no que crea uno nuevo con los cambios indicados. si consultamos API, observamos que setScale() realmente devuelve un "nuevo" BigDecimal con la escala actualizada



así que para simular la modificación de la escala lo que puedo hacer es "utilizar" el nuevo objeto que devuelve setScale()

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

O directamente reutilizando la variable *a* para que apunte al nuevo objeto con la nueva escala

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

```
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());
   }
}
```

Estas apreciaciones se aplican no sólo a setScale() si no a gran cantidad de métodos que al no poder modificar el objeto por ser inmutable generan uno nuevo actualizado.

Redondeo

Recordemos que "lo bueno" de BigDecimal es que permite al programador tener bajo su control la imprecisión inevitable al representar números 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. Aunque en los ejemplos anteriores lo fuimos evitando, el redondeo es un parámetro que hace falta para prácticamente cualquier operación con BigDecimal.

En el ejemplo anterior cambiamos la escala. Al aumentar la escala no tuvimos problemas con el redondeo, por el contrario, el siguiente ejemplo que disminuye la escala, producirá un 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);
     }
}
```

setScale va a generar un número de dos decimales, por lógica hay dos candidatos 2.53 y 2.54 ¿Con cuál se queda?, es decir, ¿cómo redondea?

indicar redondeos con la enumeración RoundingMode

La base del redondeo es indicar cómo queremos redondear a través de una constante de la enumeración RoundingMode. No sabemos de momento lo que es una enumeración, nos limitamos simplemente a verlas como un conjunto de constantes. Hay varios tipos de redondeos y para cada uno una constante asociada, por ejemplo, las constantes RoundingMode.DOWN y RoundingMode.UP para los redondeos hacia abajo y hacia arriba tradicionales.

En el siguiente ejemplo, de paso que cambiamos de escala, como segundo parámetro indicamos el redondeo deseado.

```
import java.math.BigDecimal;
import java.math.RoundingMode;
public class Unidad2 {
   public static void main(String[] args) {
     BigDecimal a = new BigDecimal("2.5333");
     a=a.setScale(2,RoundingMode.DOWN);
```

```
    System.out.println(a);
    BigDecimal b = new BigDecimal("2.5333");
    b=b.setScale(2,RoundingMode.UP);
    System.out.println(b);
    //y hay más tipos de redondeo ...
   }
}
```

puedes consultar los diversos tipos de redondeos existentes en el api de BigDecimal

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. Echa un vistazo al API de BigDecimal y encuentra la justificación de los resultados anteriores.

Ejercicio U2_B10_E1

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.

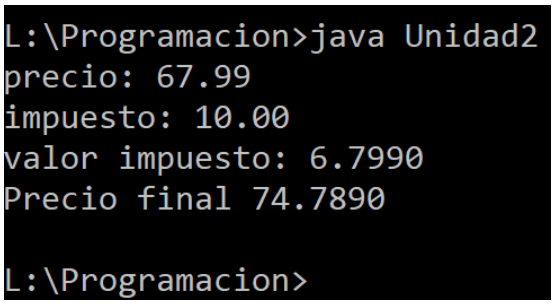
La operación de división genera una excepción, intenta razonar por qué y consultando el API escoge la versión del método *divide()* que te permite indicar redondeo para solucionar el problema.

Ejercicio U2_B10_E2:

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



OJO nextBigDecimal() depende de la configuración del idioma y teclado del ordenador en que se ejecuta. Puede ocurrir que el ejemplo anterior hubiera que teclearlo con coma decimal en lugar de punto decimal(Recuerda que esto mismo ocurría con nextFloat() y nextDouble())

Ejercicio U2_B10_E3:
Repite el ejercicio anterior sin nextBigDecimal().

Ejercicio U2_B10_E4:
Repetimos el ejercicio anterior pero ahora con la siguiente consideración. Suponemos que somos el administrativo de una empresa y que hacemos los cálculos a boli y papel de la siguiente forma:

1. Tomamos el precio y nos quedamos con dos decimales. Si hiciera falta redondeo, redondeamos hacia arriba.
   por ejemplo 10.542 =>10.55

2. lo mismo para el valor del impuesto en tanto por cien.
   por ejemplo
   10.0=>10.00
3. paso el impuesto a tanto por uno con 2 decimales, redondeo si fuera necesario
   10.00/100=>0.10

4. calculo el valor del impuesto para el precio. Resultado con 2 decimales y consiguiente redondeo si hiciera falta
   10.55*0.10=>1.0550=>1.06
5. obtengo el pvp
   10,55+1,06=11,61

Creo que legalmente, en una factura, es suficiente con redondear el total final (el pvp) pero lo que nos interesa ahora es ver que desde java podemos hacer un proceso como el de arriba, es decir, que tenemos el control deseado sobre escala y redondeo.

Ahora se pide escribir este proceso en java, tomando el valor del precio y el impuesto en tanto por cien desde el teclado. Un ejemplo de ejecución podría ser:

```
C:\Users\Pilt>java  Unidad2
precio en euros: 10.543
precio redondeado: 10.55
impuesto en %: 10.0
impuesto en % redondeado: 10.00
impuesto en tanto por 1 y redondeado: 0.10
impuesto en euros sobre el precio: 1.0550
impuesto en euros sobre el precio redondeado: 1.06
pvp: 11.61

C:\Users\Pilt>
```

```
C:\Users\Pilt>java  Unidad2
precio en euros: 10.123456
precio redondeado: 10.13
impuesto en %: 9.512
impuesto en % redondeado: 9.52
impuesto en tanto por 1 y redondeado: 0.10
impuesto en euros sobre el precio: 1.0130
impuesto en euros sobre el precio redondeado: 1.02
pvp: 11.15
```

Ejercicio U2_B8_E5:

Recuerda el siguiente ejemplo

```
class Cuenta {
    private String numeroCuenta;
    private String titular;
    private double saldo;

    Cuenta(String numeroCuenta, String titular, double saldo) {
        this.numeroCuenta = numeroCuenta;
        this.titular = titular;
        this.saldo = saldo;
    }

    Cuenta() {
        this("sin numero", "sin titular", 0.0);
    }

    void setSaldo(double saldo) {
        this.saldo = saldo;
    }

    double getSaldo() {
        return saldo;
    }

    public String toString() {
        return "("+numeroCuenta + ", " + titular + ", " + saldo+")";
    }

}

public class Unidad2 {
    public static void main(String[] args) {
        Cuenta c1 = new Cuenta("111-222", "Epi", 50.0);
        System.out.println("el saldo inicial de Epi es: " + c1.getSaldo());

        Cuenta c2 = new Cuenta("999-888", "Blas", 100.0);
        System.out.println("Datos de la cuenta c2: " + c2);

        c1.setSaldo(250.0);
        System.out.println("el nuevo saldo de Epi es: " + c1.getSaldo());

        Cuenta c3 = new Cuenta();
        System.out.println("datos de cuenta creada sin parámetros: " + c3);
    }
}
```

Actualiza el ejemplo anterior sustituyendo

```
private double saldo;
```

Por

```
private BigDecimal saldo;
```

lo que implica diversos cambios en el código. Como el ejemplo no hace operaciones con el saldo no te preocupes en este caso por escala y redondeo.