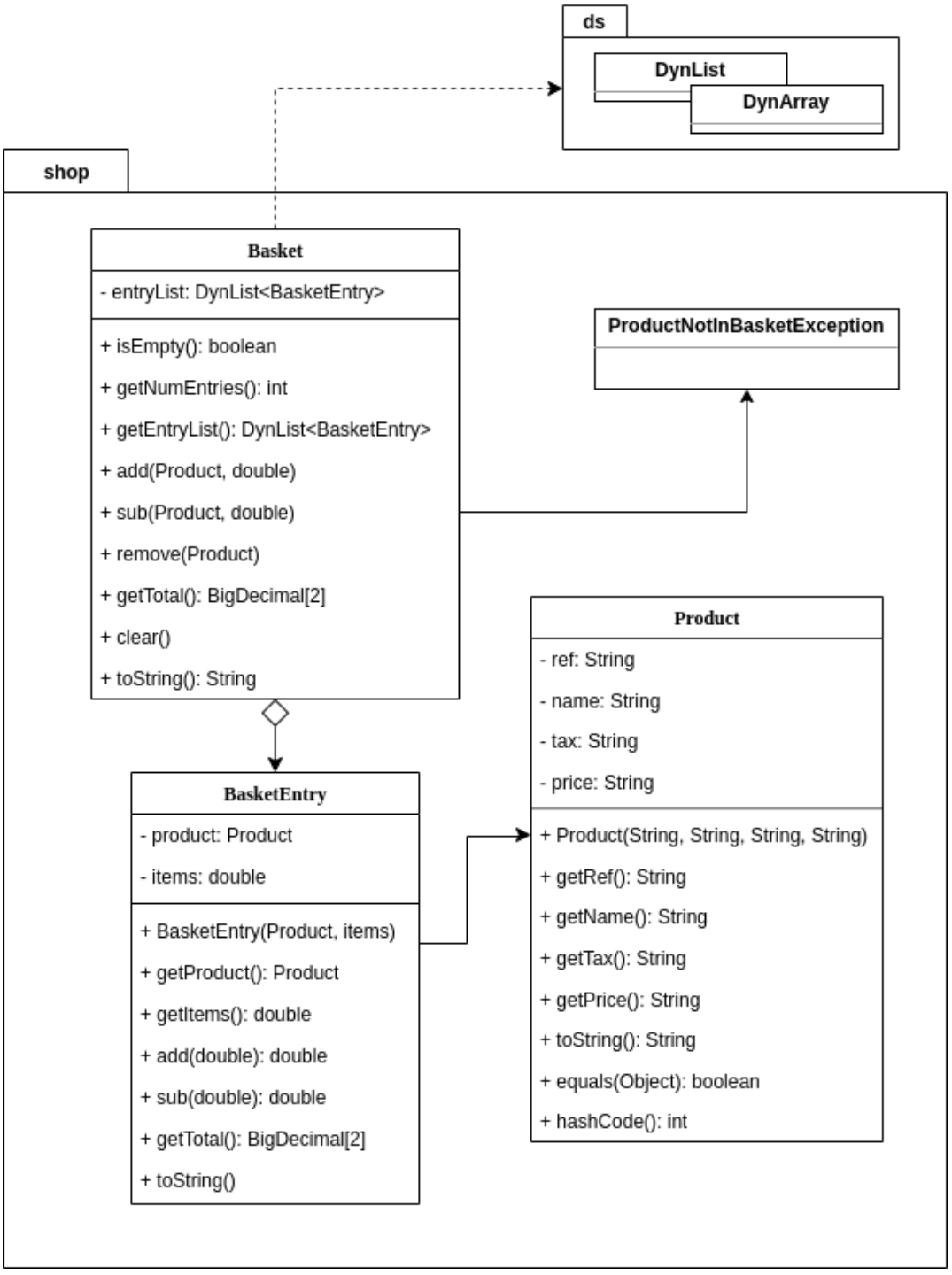
DAM/DAW Programación

UNIDAD 8 - Generic DynArray - Parte II

1.

La segunda parte del ejercicio consistirá en la implementación de un carrito de la compra simplificado.

Para ello, implementaremos el siguiente diagrama de clases:



CLASES

shop.Basket

Reperesenta un carrito de la compra. Dispondrá de diversos métodos que nos permitirán añadir o eliminar Productos del mismo. Además, nos devolverá el importe total de los productos incluidos así como cuál es la cantidad correspondiente a impuestos.

- Atributos:

Nombre	Descripción
entryList	Colección de entradas del carrito

- Métodos:

Nombre	Descripción
getEntryList	Devuelve una **referencia** a la colección de entradas del carrito
isEmpty	Indica si el carrito está vacío o no
getNumEntries	Devuelve el número de entradas del carrito
clear	Vacía el carrito
add	Crea una nueva entrada en el carrito para el Producto y número de unidades indicadas. Si el Producto ya se encuentra en el carrito, se actualiza el número de unidades de la entrada correspondiente
sub	Resta el número de unidades indicado del Producto del carrito. Si se eliminan todas las unidades de un Producto, la entrada debe eliminarse Si no hay en el carrito ninguna ocurrencia del Producto indicado, lanzará la excepción ProductNotInBasketException
remove	Elimina del carrito la entrada del Productoindicado. Si no hay en el carrito ninguna ocurrencia del Producto indicado, lanzará la excepción ProductNotInBasketException
getTotal	Devuelve un array con el total de impuestos y el total del carrito (ver NOTA (††))
toString	Devolverá un representación en texto del carrito según el formato indicado (ver NOTA (†))

shop.BasketEntry

Reperesenta una entrada del carrito de la compra. Cada entrada constará de un Producto y un número (positivo) de unidades solicitadas. Dispondrá de diversos métodos que nos permitirán añadir o eliminar unidades del Prodcuto. Además, nos permitirá obtener el importe total y el total de impuestos de la entrada según el siguiente cálculo:

```
base = num_unidades x precio_producto
impuestos = base x tasa_producto
total = base + impuestos
```

- Atributos:

Nombre	Descripción
Producto	Producto de la entrada de carrito
items	Número de unidades (real positivo)

- Métodos:

Nombre	Descripción
(Constructor)	Una entrada de carrito se inicializará con un Producto y un número inicial de unidades
getProduct	Devuelve el Producto de la entrada
getItems	Devuelve el número de unidades de la entrada
add	Añade el número de unidades indicado y devuelve el valor actualizado
sub	Resta el número de unidades indicado y devuelve el valor actualizado
getTotal	Devuelve un array con el total de impuestos y el total de la entrada de carrito (ver NOTA (††))
toString	Devolverá un representación en texto del carrito según el formato indicado (ver NOTA (†))

shop.Producto

Representa los diferentes productos que pueden ser añadidos al carrito. Cada Producto está identificado por su **referencia**.

- Atributos:

Nombre	Descripción
ref	Referencia única del Producto
name	Descripción del Producto
tax	Porcentaje de impuestos sobre el Producto (ver NOTA (††))
price	Precio (base imponible) del Producto (ver NOTA (††))

- Métodos:

Nombre	Descripción
(Constructor)	Una Producto se inicializará con su referencia, descripción, tasa de impuestos y precio
getRef	Devuelve la referencia del Producto
getName	Devuelve la descripción del Producto
getTax	Devuelve la tasa de impuestos aplicada al Producto
getPrice	Devuelve el precio de Producto
equals	Comprobación de igualdad (ver NOTA (†††))
hashCode	Valor hash del Producto (ver NOTA (†††))
toString	Devolverá un representación en texto del Producto según el formato indicado (ver NOTA (†))

shop.NoProductInBasketException

Excepción generada al intentar realizar una operación sobre un producto que no se encuentra en el carrito.

Por defecto, se inicializará con el mensaje: “El producto no se encuentra en el carrito”

NOTAS

(†)

Formatos de Salida

Los formatos de salida de las clases serán los siguientes:

- **shop.Producto:**

```
[ref; name; tax; price]
```

Ejemplo:

```
[R001; Producto 1; 0.04; 21.25]
```

- **shop.BasketEntry:**

```
product[items][totalTax, totalAmount]
```

Ejemplo:

```
[R003; Producto 3; 0.21%; 112.18][2.0][47.12, 271.48]
```

- **shop.Basket:**

```
1:basketEntry 1
2:basketEntry 2
…
#[totalTax, totalAmount]
```

Ejemplo:

```
1:[R001; Producto 1; 0.04%; 21.25€][3.0][2.55, 66.30]
2:[R002; Producto 2; 0.10%; 32.50€][1.0][3.25, 35.75]
3:[R003; Producto 3; 0.21%; 112.18€][2.0][47.12, 271.48]
#:[52.92, 373.53]
```

(††)

La clase BigDecimal y los tipos numéricos de punto flotante

Los tipos primitivos para números de coma flotante **float** y **double**, basados en el estandar IEEE754, permiten obtener una representación binaria de los números reales en notación científica mediante el almacenamiento de su exponente y mantisa.

Sin embargo, dichos formatos, de 32 y 64 bits, no permiten representar el rango infinito de valores del conjunto de números reales y, en la mayoría de los casos, serán aproximaciones. Esto es debido a que el espacio de almacenamiento de exponente y mantisa es finito, lo que determina el denominado **espaciado de números**, es decir, la separación entre números representables que, además, será proporcional al exponente.

Así, por ejemplo, el siguiente número a **1** que podemos representar será 1.0000000000000002, pero no ningún otro intermedio. De igual modo que no podemos representar ningún valor decimal entre 2251799813685248.0 y 2251799813685249.0

Todo esto puede tener repercusiones en nuestros cálculos aritméticos. Especialemte en el caso de trabajar con cantidades monetarias. Esto lo podemos ver en las siguientes operaciones con double:

```
jshell> 1.75 - 1.15
$1 ==> 0.6000000000000001

jshell> 1.70 - 1.10
$2 ==> 0.5999999999999999
```

Aún cuando ambas restas deberían darnos el mismo resultado (0.60) vemos que, en la práctica, no es así. El valor 1.75 puede representarse de forma exacta en binario pero 1.25, 1.70 y 1.10 no, con lo que sus restas binarias no serán exactamente 0.60 (que, además, tampoco tiene representación exacta)

Para evitar estas situaciones al trabajar con valores monetarios con **coma fija** (como es el caso del euro), tenemos dos opciones:

1) Trabajar con números enteros y cantidades en céntimos. No tendremos problemas de "exactitud" en la representación de las cantidades pero nos obliga a gestionar la posición decimal (en multiplicaciones y divisiones) y a hacer el redondeo correspondiente

2) Emplear la clase BigDecimal de Java, que nos facilita la representación de números decimales de precisión (escala) fija. En nuestro caso, esta será la opción que adoptaremos

java.Math.BigDecimal

https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html

BigDecimal representa números decimales con signo de precisión arbitraria e **inmutables.** Constan de dos partes:

- Valor sin escala, que será un número entero
- Escala, número de dígitos a la derecha del punto decimal

Por ejemplo, el BigDecimal 5.73 tendrá de valor 573 y de escala 2

Usamos BigDecimal para aritmética de alta precisión o en cálculos que requieran control sobre la escala o el tipo de redondeo (por ejemplo, transacciones financieras)

BigDecimal proporciona constructores para crear sus objetos a partir de Strings, arrays de caracteres, enteros,...

```
BigDecimal b1 = new BigDecimal(“0.1”);    // creado desde String
BigDecimal b2 = new BigDecimal(new char[]{'0','.','2'});    // desde array

// b1 + b1 = b2 ? (0.1 + 0.1 == 0.2)?
b1.add(b1).equals(b2);  // ==> true
```

También a partir de double pero, por lo comentado anteriormente, no es apropiado. Por ejemplo:

```
BigDecimal b3 = new BigDecimal(0.1);      // creado desde Double
BigDecimal b4 = new BigDecimal(0.2);      // creado desde Double

// b3 + b3 = b4 ? (0.1 + 0.1 == 0.2)?
b3.add(b3).equals(b4);  // ==> false
```

Operaciones

Una característica de los objetos BigDecimal es que son **inmutables** (como las instancias de String). Es decir, una vez creados no se pueden modificar

Cualquier operación que hagamos con ellos deberá ser a través de sus diferentes métodos (add, sub, multiply,...) Estos métodos devolverán una nueva instancia BigDecimal con el resultado de la operación. Por ejemplo:

```
BigDecimal b1 = new BigDecimal("10.5");
BigDecimal b2 = new BigDecimal("22.3");
BigDecimal res = b1.multiply(b2);   // res = 10.5 * 22.3 = 235.265
```

Escala y Redondeo

Para establecer la precisión decimal la clase BigDecimal nos proporciona el método **setScale( )**. De las diferentes versiones sobrecargadas de este método, nos interesa especialmente:

```
setScale(int newScale,  RoundingMode roundingMode)
```

Este método permite establecer la precisión decimal e indicar el tipo de redondeo que se aplicará en caso de ser necesario. Recuerda que, al ser inmutables, este método no modifica el objeto BigDecimal sobre el que se aplica, sino que nos devolverá un nuevo objeto BigDecimal con el resultado.

Por ejemplo:

```
BigDecimal b1 = new BigDecimal("10.5");
BigDecimal b2 = new BigDecimal("22.3");
BigDecimal res = b1.multiply(b2);   // res = 10.5 * 22.3 = 235.265
BigDecimal rnd = res.setScale(2, RoundingMode.HALF_EVEN); // rnd = 235.26
```

Existen 8 tipos de redondeo:

Tipo	**Redondeo**	**Ejemplo** (scale=2)
CEILING	Hacia arriba (⇒ +∞)	3.521 >> 3.53; -3.529 >> -3.52
FLOOR	Hacia abajo (⇒ -∞)	3.529 >> 3.52; -3.521 >> -3.53
UP	Hacia arriba (0 ⇒)	3.521 >> 3.53; -3.529 >> -3.53
DOWN	Hacia abajo (⇒ 0)	3.529 >> 3.52; -3.521 >> -3.52
HALF_UP	Al vecino más cercano. Si están equidistantes, hacia arriba	3.525 >> 3.53; -3.525 >> -3.53
HALF_DOWN	Al vecino más cercano. Si están equidistantes, hacia arriba	3.525 >> 3.52; -3.525 >> -3.52
HALF_EVEN	**Al vecino más cerca. Si están equidistantes, al vecino par**	3.515 >> 3.52; 3.525 >> 3.52
UNNECESSARY	Si el resultado no es exacto y se necesitara redondear, lanza una excepción ArithmeticExcepction	

Más ejemplos:
https://docs.oracle.com/javase/8/docs/api/java/math/RoundingMode.html

El redondeo **HALF_EVEN** minimiza los errores acumulados de redondeo. Es el **más usado**. Se conoce también como *convergent rounding*, *statistician's rounding*, odd–even rounding, o **bankers' rounding**.

(†††)

Comparación de Productos: el método equals( )

Cuando queremos comparar dos instancias de la misma clase, debemos distinguir entre aquellas clases donde cada instancia representa un **"objeto del problema"** único en si mismo (como, por ejemplo, un carrito que será siempre diferente de otro carrito aunque tengan el mismo contenido), de las llamadas **"clases valor"**, donde las clases simplemente representan un valor (por ejemplo, un String, donde dos instancias con la misma cadena de caracteres las solemos considerar **"iguales"**)

En el primer caso, la simple comparación del valor de la referencia del objeto es suficiente para determinar si dos instancias son iguales o no (carrito1 == carrito2). En el segundo caso, donde la igualdad se determina **"a nivel lógico"**, empleamos el método **equals( )** para evaluar esa igualdad.

En nuestro caso, la clase Producto es una de esas clase-valor donde la igualdad se determina en base al "contenido" de la instancia y no por la referencia de la instancia. Por ejemplo:

```
Producto p1 = new Producto("A0001", "Manzanas Golden", "0.04", "1.12");
Producto p2 = new Producto("A0001", "Manzanas Golden", "0.04", "1.12");

p1 == p2          // ==> false (diferentes referencias de instancia)
p1.equals(p2)     // ==> true (mismo atributo ref de Producto)
```

Si bien p1 y p2 referencian dos instancias diferentes de la clase (`p1 != p2`), a nivel lógico "representan" dos valores iguales del problema (Manzanas Golden). Por supuesto, al ser una igualdad a nivel lógico, depende del problema en cuestión (en otro programa diferente podría considerarse que son "objetos" diferentes). En nuestro caso concreto, la clase Producto es una clase-valor de forma que la igualdad entre dos productos queda determinada por su **abributo ref** (referencia del producto) y no las referencias de las instancias. De hecho, no podemos tener más de una entrada en el carrito para el mismo producto.

Así, en distintas operaciones realizadas sobre el carrito (add, sub, remove) nos veremos en la obligación de hacer búsquedas sobre las entradas del carrito para determinar si el Producto indicado ya se encuentra en el mismo.

Para poder hacer estas comprobaciones de forma adecuada, deberemos sobreescribir el método **equals( )** de la clase Producto para comprobar si dos instancias de esta clase se corresponden con el mismo Producto

```
p1.equals(p2)     si y sólo si      p1.ref == p2.ref
```

Si nos fijamos en la firma del método **equals( )** heredado de **java.lang.Object**:

```
public boolean equals(Object obj)
```

vemos que define como parámetro una variable de tipo Object. Para poder hacer las operaciones de comprobación correspondientes, deberemos hacer un **cast** de ese Object a nuestra clase Producto

Comparación de Productos: el método hashCode( )

Cada vez que sobreescribimos el método **equals( )**, debemos sobreescribir el método **hashCode( )**

El método **hashCode( )** devuelve un identificador del objeto que es empleado en colecciones del tipo **HashMap** o **HashSet** donde las búsquedas u ordenaciones se realizan en base al código hash devuelto por dicho método.

La especificación de Object establece el siguiente contrato respecto al método **hashCode( )**:

- Durante la ejecución de una aplicación, la invocación de hashCode( ) sobre un objeto devolverá siempre el mismo valor, siempre y cuando ninguno de los atributos empleados en equals( ) se hayan modificado. Dicho valor podrá variar entre diferentes ejecuciones del programa

- **Si dos objetos son iguales según el método equals( ), ambos objetos deben devolver el mismo valor al invocar su método hashCode( )**

- Si dos objetos son diferentes según el método equals( ), **no están obligados** a devolver un valor diferente al invocar su método hashCode()

De los puntos indicados, nos interesa especialmente la segunda: **objetos iguales deben tener código hash iguales.**

Sin embargo, la implementación por defecto de hashCode( ), devuelve valores diferentes **para cada objeto**, lo cual no es válido en nuestro caso:

```
Producto p1 = new Producto("A0001", "Manzanas Golden", "0.04", "1.12");
Producto p2 = new Producto("A0001", "Manzanas Golden", "0.04", "1.12");

p1.hashCode();    // ==> 225534817
p1.hashCode();    // ==> 1878246837
```

Por tanto, nos vemos en la obligación de sobreescribir el método hashCode( ) para proporcionar el **mismo número hash** para instancias de Producto que representen el **mismo "objeto lógico"**

En realidad, no existe ningún requerimiento específico para la implementación de dicho método, más allá de que debe devolver un número entero y que ese número **debería** ser igual para aquellas instancias consideradas iguales por el método equals( ).

Aunque ya vimos que **no es obligatorio**, es muy conveniente que nuestro método genere un **valor único** para cada instancia diferente de la clase

Existen diferentes métodos para generar estos valores únicos. En general, se trata de generar dicho valor a partir de los **atributos significativos** de la clase, es decir aquellos atributos que representan "unívocamente" un objeto de la clase (por ejemplo, el atributo ref de Producto) y aplicarles una función hash (MD5, SHA1, CRC,...). Por ejemplo, dado que el atributo ref de Producto es único para cada uno de ellos, y lo empleamos en la comparación, usando los propios métodos hashCode( ) de las clases "wrapper" de los tipos Java, podríamos simplemente hacer:

```
public int hashCode() {
    return String.hashCode(this.ref);
}
```

Normalmente, se emplean métodos "más elaborados" que amplíen el espacio de claves y eviten colisiones (hash iguales para objetos diferentes)

La siguiente es una implementación típica en la que se calcula el hash para uno de los campos significativos y se van acumulando los hashes del resto de campos (normalmente, significativos):

```
public int hashCode() {
    int result = Tipo.hashCode(atributo1);
    result = 31*result + Tipo.hashCode(atributo2);
    result = 31*result + Tipo.hashCode(atributo3);
    ...
    return result
}
```

A modo de comentario y fuera del ámbito de nuestro problema, el hecho de usar el número 31 tiene que ver con que se trata de un número primo impar (lo que aumenta la varianza en la distribución resultante) y que esa multiplicación es trivial implementarla con desplazamiento de bits (lo que redunda en el rendimiento). Por supuesto, cuantos más campos se añadan, más "costoso" será el cálculo