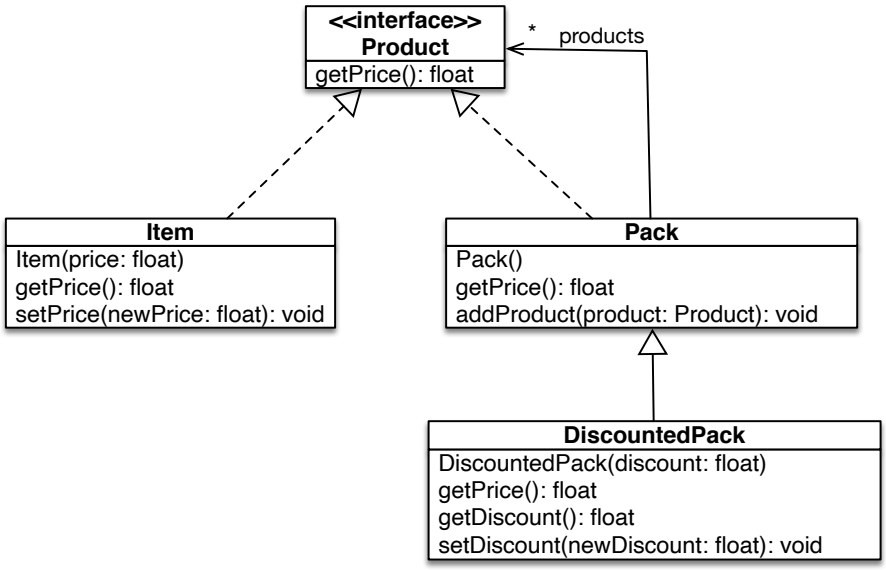
Dado el siguiente diseño de partida:



La clase `Item`:

- representa un producto individual que, en su constructor, recibe el valor del precio que, para simplificar, será un `float`[1].
- tiene el método `getPrice` que retorna el precio del producto
- tiene el método `setPrice` per cambiar el precio del producto

La clase `Pack`

- representa un grupo de productos (que pueden ser tanto `Items` como `Packs`)
- tiene el método `addProduct` para añadir un nuevo producto al pack
- tiene el método getPrice para calcular el precio del pack, que es la suma de los precios de los productos que contiene. Por ejemplo: si los productos del pack tienen precios 10,0 i 20,0, el precio del pack es 30,0

La clase `DiscountedPack`

- tiene un constructor que recibe el tanto por uno de descuento a aplicar
- tiene el método `getDiscount` que retorna el tanto por uno de descuento
- tiene el método `setDiscount` para cambiar el tanto por uno de descuento
- tiene el método `getPrice` que retorna el precio del pack con descuento: suma de los precios de los productos descontada según el tanto por uno de descuento. Por ejemplo: si los productos tienen precios 10.0 i 20.0 i el tanto por uno es 0.2, el precio es (1,0 – 0,2) * (10,0 + 20,0) = 0,8 * 30,0 = 24,0

Consideraciones generales:

- si en algún método se recibe un **precio negativo o cero** se lanza la excepción no comprobada `IllegalArgumentException`
- si el **descuento no està entre 0,0 i 1,0** (sin incluir los límites) se lanza también `IllegalArgumentException`. NOTA: 0,0 y 1.0 no son valores válidos para el descuento.

---

[1] Nunca, nunca, nunca modeléis precios con `floats`. Usad `BigDecimal` o una clase monetaria cuya aritmética permita fijar el nivel de precisión de los cálculos. NOTA: El examen, que no es la vida real, se ha de resolver con floats.

- **podéis suponer que los productos forman un árbol y no debéis hacer nada que lo garantice**

a) (1,5 punto) Implementad el método `getPrice` de la clase `Pack` (deberéis indicar también la representación de la clase[2]).
b) (1,5 punto) Implementad el método `getPrice` de la clase `DiscountedPack` (deberéis indicar también la representación de la clase). **No dupliquéis código.**

Se desea poder observar los **cambios de precios de los productos** usando el **patrón observador**, en su versión **push**, pero **nunca notificar si el precio no ha cambiado de valor**.

Para indicar el cambio de precios producido, se dispone de la clase `PriceChanged`

```
1 public final class PriceChanged {
2   private final float oldPrice;
3   private final float newPrice;
4
5   public PriceChanged(float oldPrice, float newPrice) {
6     this.oldPrice = oldPrice;
7     this.newPrice = newPrice;
8   }
9   public float getOldPrice() { return oldPrice; }
10  public float getNewPrice() { return newPrice; }
11 }
```

c) (1 punto) Modificad el diagrama para poder hacer que los Productos sean Observables y que **notifiquen cambios de precio sólo cuando éste se ha producido**. Podéis esperar a tener los apartados *d* a *g* acabados para hacer el diagrama definitivo.
d) (1,5 punto) Implementad el método `setPrice` de la clase `Item` (deberéis indicar también la representación de la clase)
e) (1,5 puntos) Implementad el método `addProduct` de la clase `Pack` (deberéis indicar también la representación de la clase, si ha cambiado respecto el apartado *a*)
f) (1,5 puntos) Implementad el **método que propaga los cambios de precio** de los `Packs` cuando alguno de los productos que contiene ha cambiado de precio (deberéis indicar también la representación de la clase si ha cambiado respecto el apartado *e*)
g) (1,5 puntos) Cuando cambia el descuento de un `DiscountedPack` su precio también cambia. Implementad el método `setDiscount` para que esto sea así (deberéis indicar también la representación de la clase si ha cambiado respecto el apartado *b*)

Clase `Observable`:

- `public void addObserver(Observer o)`
- `public void deleteObserver(Observer o)`
- `public void deleteObservers()`
- `public int countObservers()`
- `public void notifyObservers()`
- `public void notifyObservers(Object arg)`
- `protected void setChanged()`
- `protected void clearChanged()`
- `public boolean hasChanged()`

Interfície `Observer`:

- `public void update(Observable o, Object arg)`

[2] Cabecera (con `extends` e `implements`) y las variables de instancia y estáticas de la clase. NOTA: No pido los constructores de la clase.