CLASES WRAPPER, INMUTABILIDAD Y AUTOBOXING

Inmutabilidad y autoboxing son dos conceptos importantes cuando se trabaja con colecciones. Trabajaremos con colecciones en próximos boletines.

SOBRE LOS OBJETOS INMUTABLES.

Un objeto inmutable es un objeto que una vez creado no se puede cambiar. Es el equivalente a la variables constantes de los tipos primitivos. También puedo verlo como un objeto de "sólo lectura".

POR QUÉ SE PREFIEREN LOS OBJETOS INMUTABLES.

- Código más seguro. Los objetos mutables son peligrosos si se usan en concurrencia de hilos y muchas otras situaciones ya que el objeto puede acabar fuera de control con un estado no deseado.
- El código es más fácil de entender y de mantener ya que no hay seguir la traza de cambios de estado de un objeto a lo largo de la ejecución del programa.

STRING ES INMUTABLE.

Que un objeto string sea inmutable se considera una ventaja por dos razones:

- Piensa que un String, desde el momento que se crea, tiene un tamaño fijo (igual que ocurría con los arrays) y puede almacenarse en posiciones de memorias contiguas lo que provoca en general un acceso más veloz.
- Como los objetos inmutables son de "sólo lectura" dan seguridad al programa ya que está garantizado que no se van a modificar por error. Es una cuestión importante en programación avanzada (multihilo, concurrencia, ...), que no afecta a nuestros ejemplos, pero que es muy importante.

Para algunos contextos específicos podremos tener una versión mutable de la clase String que se consigue con las clases StringBuilder y StringBuffer.

En el boletín de la primera evaluación "objetos que son cadenas de caracteres". Analizamos varias cuestiones interesantes al respecto de la inmutabilidad de la clase strings que puedes repasar:

- Demostramos con trim() que un objeto String es inmutable.
- Hablamos de la creación de objetos String y del matiz entre objeto literal String y otro String no literal: Los literales string son tan usados que para mejorar la eficiencia de los programas tienen un tratamiento especial, de forma que si usamos un literal como "hola" se crea en memoria pero si ese mismo literal String se vuelve a usar no se crea uno nuevo si no que se utiliza el ya creado.

¿HAY MUCHAS CLASES INMUTABLES EN EL API?

SÍ. En las últimas versiones del JDK hubo muchas incorporaciones de clases inmutables. Por

ejemplo, las clases para el tratamiento de fechas/horas como Calendar, GregorianCalendar etc. que trabajan con objetos mutables se mantienen pero se recomienda escribir el nuevo código con un nuevo API basada en objetos inmutables de clases como LocalDate LocalDateTime y otras.
Las clases Wrapper que veremos en este boletín como Integer, Double etc. también son inmutables.

¿PODEMOS ESCRIBIR NOSOTROS UNA CLASE INMUTABLE?.

En el API son inmutables la clase *String*, las subclases de *Number*(*Integer*, *Float*, ...) y algunas otras. También podemos escribir nosotros una clase inmutable

Como se escribe una clase inmutable:

- La clase será *final*(no permite extends). Así evitamos que se sobrescriban sus métodos.
- Los atributos son *final* y por tanto esto condiciona a que no se pueden modificar una vez inicializados. Típicamente se inicializan en la propia declaración del atributo o bien en el constructor.
- Si tiene como atributos referencias a otros objetos se debe usar el principio de ocultación en constructor/set/get a no ser que el objeto referenciado también sea inmutable (como un String) en cuyo caso ya se cumple automáticamente el principio de ocultación.

Un ejemplo: Un artículo inmutable, no parece una clase muy interesante para ser inmutable pero es clarificadora de este concepto.

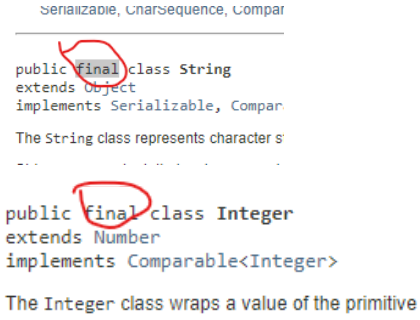
```
final class ArticuloInmutable {
   final private  String codArticulo;
   final private  String descripcion;
   final private  int cantidad;

   ArticuloInmutable(String codArticulo, String descripcion, int cantidad) {
      this.codArticulo = codArticulo;
      this.descripcion = descripcion;
      this.cantidad = cantidad;
   }

   String getCodArticulo() {
      return codArticulo;
   }

   String getDescripcion() {
      return descripcion;
   }

   int getCantidad() {
      return cantidad;
   }

}

class App {
   public static void main(String[] args){
      ArticuloInmutable art = new ArticuloInmutable("1","patata",100);
      //si quiero cambiar la cantidad del objeto anterior
      //tengo que crear uno nuevo con los datos del anterior pero con la cantidad modificada
      art= new ArticuloInmutable(art.getCodArticulo(),art.getDescripcion(),999);

   }
}
```

Ejemplo: intenta añadir a la clase anterior el típico setCantidad()

```
void setCantidad(int nuevaCantidad){
      this.cantidad=nuevaCantidad;
}
```

¿Que ocurre?

Ejemplo: String e Integer son inmutables, por tanto tienen que ser final. Compruébalo en el API



CLASE ENVOLTORIO, ENVOLVENTE O WRAPPER.

En general cuando una clase "toma como parámetro en su constructor a otra clase" para añadirle funcionalidad u ocultar algún detalle decimos que "envuelve" a esa clase como, por ejemplo

```
BufferedReader in = new BufferedReader(new FileReader(fichero));
```

Decimos que en el ejemplo anterior BufferedReader "envuelve" a FileReader

O un ejemplo más familiar

```
Scanner sc = new Scanner(new String("hola"))
```

Hay además clases que en lugar de envolver a otras clases *envuelven* a los tipos primitivos, son las clases Integer, Double, ... Cuando nos referimos a clases "Envolventes" o "Wrapper", normalmente nos referimos a estas clases que envuelven los tipos primitivos, aunque como indicamos el término "Envolver" es más amplio y hay que estar atento al contexto.

Las clases Wrapper de tipos primitivos son inmutables.

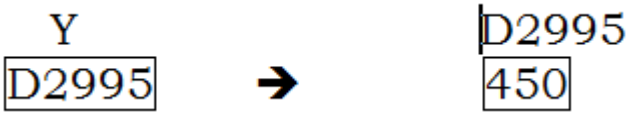
Veremos ejemplos con la clase Integer pero son extensibles a Double, Float,etc.

```
Integer i= new Integer(3);
```

Al crear un objeto Integer se le indica el valor entero que queremos que almacene, 3 en el ejemplo, y no hay forma de cambiar el valor del Integer creado, puedes consultar la API y comprobar que no existe ningún método que modifique un objeto Integer. INTEGER ES UN OBJETO INMUTABLE.

Observa la diferencia entre Integer e int en el siguiente ejemplo.

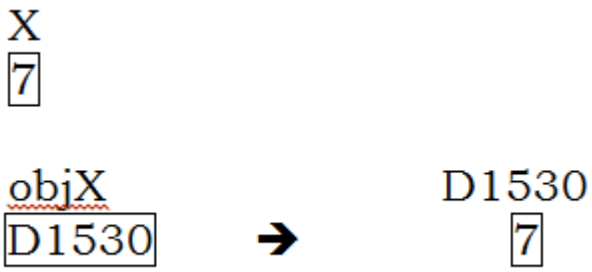
Integer y = new Integer(450);



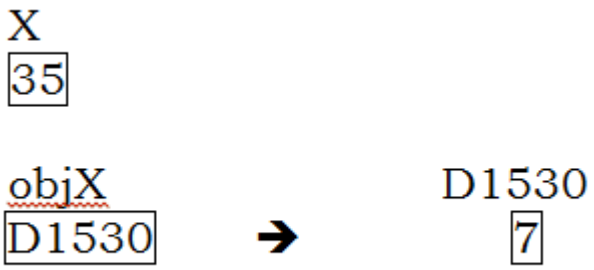
Con el valor D2995 representamos una referencia (dirección de memoria). Lógicamente, el valor de la referencias anterior nos la inventamos en el ejemplo ya que la gestión de referencias la realiza automáticamente java. En la posición de memoria D2995 hay un objeto Integer. Una de las propiedades de este objeto es su valor, que es un atributo int que vale 450.

Seguimos "observando" la memoria con nuevas intrucciones ….

int x = 7;
Integer objX = new Integer(x);



x = 35;



- **Para modificar un Integer inmutable, realmente, creo uno nuevo(no puedo modificar "el viejo").**

Si con Strings hacía

String s="primero";
s= s + " y segundo";

Con Integer ocurre algo similar

```
Integer i=new Integer(1);
Integer j= new Integer(1);
i=new Integer(i.intValue()+j.intValue());
```

AUTOBOXING.

En los ejemplos anteriores se observa que:

- para meter un valor entero en un Integer tengo que crear un nuevo objeto Integer y pasarle un valor primitivo int al constructor

new Integer(valor)

- para leer un valor entero de un Integer usamos el método intValue().

A la operación de meter un entero en un Integer se le llama ***Box*** (encajar/ empaquetar / envolver) y leerlo ***UnBox***(desencajar).

En las instrucciones anteriores hay un new Integer() implícitos. **Para entender esto es necesario entender el mecanismo de autoboxing.**

A partir de Java 5 existe el mecanismo *AutoBoxing* que deduce automáticamente en determinados contextos cuando tiene que hacer un Box y un UnBox. ¿Para qué vale? Para trabajar más cómodamente ya que los objetos como Integer son muy habituales y es una lata estar haciendo constantemente new Integer() e invocando al método intValue(),

Ejemplo: Observa este ejemplo donde i es tipo primitivo int pero J es objeto Integer

Con Autoboxing	Sin Autoboxing
int i;	int i;
Integer j;	Integer j;
i = 1;	i = 1;
j = 2;	j = new Integer(2);
i = j;	i = j.intValue();
j = i;	j = new Integer(i);

Si te fijas en la secuencia de instrucciones anteriores con AutoBoxing, podemos percibir falsamente que los Integer son mutables ya que no se hace ningún new y nos da la impresión que el objeto referenciado por J se puede modificar, pero observa atentamente la columna "sin autoboxing" porque es lo que realmente ocurre aunque escribamos las instrucciones de la columna "con autoBoxing"

Otro ejemplo, al hacer

Integer i= new Integer(3);
i=i+1; //aquí ahora i apunta a un nuevo objeto Integer, es decir tiene otra dirección.

Es decir, java está haciendo por mi algo similar a

Integer i= new Integer(3);
i = new Integer(i.intValue()+1); //ahora i contiene una referencia de memoria nueva, es decir, a otro objeto

Recuerda lo que indicamos para la creación de objetos literales String, por ejemplo, el literal "hola" se crea una sóla vez aunque se utilice muchas. Con Integer como también es inmutable le ocurre algo parecido aunque más confuso Observa como ahora sin usar el new para crear objetos h==z da true

```
class App {
   public static void main(String[] args) {
      //Integer z = new Integer(44);
      //Integer h = new Integer(44);
      Integer z = 44;
```

```
        Integer h = 44;

        System.out.println("z == h -> " + (h == z));
        System.out.println("Referencia que contiene z: "+System.identityHashCode(z));
        System.out.println("Referencia que contiene h: "+System.identityHashCode(h));
    }
}
```

Es decir que
Integer z = 44;
Genera un Integer que envuelve a 44 pero se reutiliza al hacer
Integer h = 44;

Sí ejecutas el código anterior ahora cambiando los comentarios

```
        Integer z = new Integer(44);
        Integer h = new Integer(44);
        //Integer z = 44;
        //Integer h = 44;
```

Observas que efectivamente ahora z y h referencian a objetos diferentes.

Ejercicio U6_B1_E1: Escribe el siguiente código sin autoboxing.

```
class App {
    public static void main(String[] args) {
        Integer z = 44;
        z++;
        System.out.println(z);
    }
}
```

Ejercicio U6_B1_E2: Observa el siguiente ejemplo y consultando en el API la clase Double, explica la diferencia entre valueOf y parseDouble.

```
class App {
    public static void main(String[] args) {
        String s="1234.5678";
        double d;
        d=Double.valueOf(s).doubleValue();
        System.out.println("d con valueOf: "+ d);
        d=Double.parseDouble(s);
        System.out.println("d con parseDouble: "+ d);
    }
}
```

Ejercicio U6_B1_E3:
Observa el siguiente código, ojo que no usa parseDouble como el de arriba:

```
class App {
    public static void main(String[] args) {
        String s="1234.5678";
        double d;
        d=Double.valueOf(s).doubleValue();
        System.out.println("d con Double.valueOf(s).doubleValue(): "+ d);
```

```
        d=Double.valueOf(s);
        System.out.println("d con Double.valueOf(s): "+ d);
    }
}
```

¿Por qué producen el mismo resultado las instrucciones?

```
        d=Double.valueOf(s).doubleValue();
         d=Double.valueOf(s);
```