21

La diferencia sustancial es que, en JAVA, `int` es un tipo primitivo, no un objeto, mientras que `Integer` es un objeto o una Clase.

Dicho en lenguaje coloquial: un `int` es un número, y un `Integer` es un puntero que hace referencia a una clase que contiene un entero. O... más coloquial todavía: un `Integer` es una caja, y un `int` es lo que hay dentro de esa caja.

¿Esto qué significa?

Un `int` es mucho más rápido cuando se trata de calcular números en el rango -2.147.483.648 [-2^{31}], es decir, `Integer.MIN_VALUE` y +2.147.483.647 [$2^{31}-1$] es decir, `Integer.MAX_VALUE` . O sea, un `int` tiene a nuestra disposición 32 bits de información para ser usados directamente. Ver especificaciones.

Las variables `int` son mutables. A menos que se les marque como `final` , pueden cambiar su valor en cualquier momento. Un ejemplo típico de uso de `int` para cambiar el valor del contador dentro de los bucles `for` , `while` , etc.

Un `Integer` , es un objeto que contiene un único campo `int` . Un `Integer` es mucho más voluminoso que un `int` . Los objetos `Integer` son inmutables. Si se desea afectar el valor de una variable `Integer` , la única manera es crear un nuevo objeto `Integer` y descartar el antiguo.

A. El primitivo `int`

Las variables de tipo int almacenan el valor binario real para el entero que representan.

Por eso, el siguiente código es erróneo en Java:

```
int.parseInt("1");
```

porque `int` no tiene métodos, sólo puede ser declarado para almacenar un valor.

B. La Clase `Integer`

`Integer` , como decía, es una Clase, como cualquier otra clase de Java, con sus métodos.

Este código sí es correcto en Java:

```
Integer.parseInt ("1");
```

Se trata de una llamada al método estático `parseInt` de la clase `Integer` , el cual devuelve un `int` , no un `Integer` .

Podríamos decir que `Integer` es una clase con un solo campo de tipo `int` . Esta clase se utiliza donde se necesita un `int` para ser tratado como cualquier otro objeto, como en tipos genéricos

o situaciones en las que se necesitan valores nulos.

¿Cuándo conviene usar uno u otro?

He aquí una pequeña tabla con algunos indicadores:

Uso	int	Integer
Cálculos con + - * / % ^ etc.	sí	no
Pasar como parámetro	sí	sí
Retornar como un valor	sí	sí
Usar métodos desde java.lang.Integer	no	sí
Almacenarlo en un Vector o en otra Colección	no	sí
Usarlo como una llave de HashMap	no	sí
Serializarlo	no	sí
Pasarlo como un objeto genérico (TableCellRenderer)	no	sí
Admitir como un valor nulo para significar que no hay valor	no	sí
Enviarse a sí mismo por RMI (Remote Method Invocation)	no	sí
Enviarlo como parte de otro Objeto a través de RMI	sí	sí

EDIT: Autoboxing y UnBoxing

No todo lo que brilla es oro, ¡ciudado!

Dado que las indicaciones de la tabla anterior fueron puestas en cuestión por @LuiggiMendoza en los comentarios, he querido agregar este apartado porque se puede difundir un falso concepto sobre el uso de **autoboxing y unboxing**

Desde Java 1.5 se permite convertir primitivos a objetos (wrappers) o viceversa de forma automática. Esto se conoce como AutoBoxing y UnBoxing.

Gran parte de lo que sigue está tomado de la documentación de Java:

A. Autoboxing

Autoboxing es la conversión automática que hace el compilador Java entre los tipos primitivos y sus correspondientes clases de contenedor de objetos. Por ejemplo, convertir un `int` a un `Integer`, un `double` a un `Duoble`, y así sucesivamente. Si la conversión se realiza de otra forma, se denomina unboxing.

El compilador de Java aplica autoboxing cuando un valor primitivo es:

- Pasado como parámetro a un método que espera un objeto de la clase de contenedor correspondiente.
- Asignado a una variable de la clase de contenedor correspondiente.

Aquí está el ejemplo más simple de autoboxing:

```
Character ch = 'a';
```

El resto de los ejemplos de esta sección usan genéricos...

Consideremos el siguiente código:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
     li.add(i);
```

Aunque se agregue a `li` los valores `int` como tipos primitivos, en lugar de objetos Integer, el código se compila. Porque `li` es una lista de objetos `Integer`, no una lista de valores `int`. Uno puede preguntarse por qué el compilador Java no emite un **error en tiempo de compilación**. El compilador no genera un error porque **crea un objeto `Integer` desde `i` y añade el objeto a `li`**. Por lo tanto, **el compilador convierte el código anterior** a lo siguiente en tiempo de ejecución:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(Integer.valueOf(i));
```

> O sea, el compilador tiene que hacer un trabajo suplementario que debió hacer el programador al enviar el dato del tipo correcto. Es más, podríamos decir que el compilador corrige un **error** que le hemos enviado.

Algunos consideran que el autoboxing - unboxing son algo mágico, algo estupendo. Considero en lo personal que no lo es. Y aquí cabe preguntarse, **¿por qué delegar en el compilador cosas que puede hacer el programador?** Cuidado, porque podríamos tener un código bonito, de agradable lectura, pero que podría tener un fallo grave en cualquier momento.

Eso bastaría pero... además de eso, el uso inadecuado de autoboxing-unboxing tendría un costo de rendimiento que puede ser ciertamente leve en operaciones pequeñas, pero que podría ser consecuencia de un error en la ejecución del programa sin pensamos en operaciones que tengan que manejar gran cantidad de datos o en dispositivos con poca capacidad de memoria.

Esto, entre otras cosas, son indicadas con toda claridad en la documentación de Java:

> A boxing conversion may result in an **OutOfMemoryError** if a new instance of one of the wrapper classes (Boolean, Byte, Character, Short, Integer, Long, Float, or Double) needs to be allocated and insufficient storage is available.

B. Unboxing

La conversión de un objeto de un tipo de contenedor (`Integer`) en su valor primitivo (`int`) correspondiente se denomina unboxing.

El compilador Java aplica unboxing cuando un objeto de una clase wrapper es:

- Pasado como parámetro a un método que espera un valor del tipo primitivo correspondiente.
- Asignado a una variable del tipo primitivo correspondiente.

Consideremos el siguiente método:

```
public static int sumEven(List<Integer> li) {
    int sum = 0;
    for (Integer i: li)
        if (i % 2 == 0)
            sum += i;
        return sum;
}
```

Dado que los operadores (%) y (+ =) no se aplican a objetos `Integer` , uno se podría preguntar por qué el compilador de Java compila el método sin emitir ningún error. **El compilador no genera un error porque invoca el método intValue para convertir un `Integer` en un `int` en tiempo de ejecución**:

> Aquí tenemos de nuevo al compilador haciendo un trabajo sumplementario porque el programador no usó el tipo de dato que esperaba el elemento.

```
public static int sumEven(List<Integer> li) {
    int sum = 0;
    for (Integer i : li)
        if (i.intValue() % 2 == 0)
            sum += i.intValue();
        return sum;
}
```

> Me pregunto de nuevo: **¿por qué motivo, si el compilador espera un tipo de dato tengo que enviarle otro?**

Según Java, es una cuestión de **estética**, no de rendimiento:

> Autoboxing and unboxing lets developers write cleaner code, making it easier to read.
>
> Autoboxing y unboxing permiten a los desarrolladores escribir un código más claro, haciéndolo más lisible.

Por eso dice Java:

> So when should you use autoboxing and unboxing? Use them only when there is an "impedance mismatch" between reference types and primitives, for example, when you have to put numerical values into a collection. It is not appropriate to use autoboxing and unboxing for scientific computing, or other performance-sensitive numerical code. An Integer is not a substitute for an int; autoboxing and unboxing blur the distinction between primitive types and reference types, but they do not eliminate it.
>
> Entonces, ¿cuándo debería usar autoboxing y unboxing? Úselos sólo cuando hay un "fallo de impedancia(*)" (impedance mismatch) entre los tipos de referencia y los primitivas, por ejemplo, cuando hay que poner valores numéricos en una colección. No es apropiado usar autoboxing y unboxing para la computación científica **u otro código numérico sensible al**

> **rendimiento**. **Un entero no es un sustituto de un int; Autoboxing y unboxing difuminan la distinción entre tipos primitivos y tipos de referencia, pero no la eliminan**.

(*)Fallo de impedancia (impedance mismatch). Para entender este concepto pondré un ejemplo: Si estamos esperando valores de un columna de una base de datos que admite nulos, y asignamos ese valor en una variable del tipo `int` estamos ante un caso de `impedance mismatch` ya que como `int` no admite nulos, el programa podría dar error, o a lo sumo asignará el valor `0` en vez de `NULL`. O viceversa, si enviamos los valores desde Java a la BD, si usamos un `int` para INSERT o UPDATE en la BD, podríamos estar enviando a algunas columnas en vez del valor `NULL` el valor `0`. Esto, que puede parecer una **tontería** podría ser algo muy grave en algunos casos.

> **En resumen, no delegues en el compilador cosas que deberías hacer tú. Los resultados pueden ser catastróficos. Perder de la mano el control del tipo de dato que se ha de usar delegándolo, es una mala práctica de programación.**

Y no digamos nada si hay que hacer **comparaciones**. Podríamos tener resultados inesperados, ya que autoboxing - unboxing enmascaran los valores reales, porque jamás un `Integer` será un sustituto de un `int`. **Una cosa es la caja, y otra lo que hay dentro de la caja.** Si confundimos las dos cosas podemos cometer errores muy graves.

Hay más motivos por los que no es bueno hacer que Java haga lo que uno debería hacer. Pero sería extenderse demasiado...

Es notorio señalar que en Java cada tipo primitivo tiene una clase de contenedor equivalente:

- `byte` tiene `byte`
- `short` tiene `Short`
- `int` tiene `Integer`
- `long` tiene `Long`
- `boolean` tiene `Boolean`
- `char` tiene `Char`
- `float` tiene `Float`
- `double` tiene `Double`