

Apunte 05 - Programación Orientada a Objetos en Java

Introducción a POO

De la misma forma que hemos dicho que la presente asignatura no pretende ser un curso de Java, decimos ahora que tampoco es nuestro objetivo se un curso de Programación Orientada a Objetos, en el caso del Lenguaje Java ya hemos explicado que proponemos revisar los elementos fundamentales para poder implementar los elementos de backend propuestos. En el caso de Programación Orientada a Objetos, sin embargo, entendemos que es un tema ya abordado en materias anteriores o concurrentes con Backend de Aplicaciones y por lo tanto nos limitaremos a revisar las herramientas de Java para implementar la POO agregando solo algunos comentarios sobre buenas prácticas.

Lo primero que nos gustaría, mostrar en este apunte y es uno de los por qué de la POO en lo que respecta al manejo de la complejidad, para eso vamos a tomar como ejemplo guía del presente apunte el siguiente caso:

Vamos a realizar una comparación elemental de complejidad entre el cálculo del promedio de 10 números enteros y el cálculo del promedio de 10 fracciones tratando de aproximar la complejidad a partir de las líneas de código y sin entrar en mayor detalle como sería por ejemplo un análisis de complejidad ciclomática.

Si pensamos en el código de del cálculo del promedio de 10 números aleatorios tendríamos algo como lo que sigue:

```
11          // Primero con números enteros
12          System.out.println("Primero con enteros");
13          int acumuladorI = 0;
14          for(int i = 0; i < 10; i++)
15          {
16              int aux = (int) (Math.random() * 10);
17              System.out.println("Generado: " + aux);
18              acumuladorI += aux; // acumuladorI = acumuladorI + aux;
19          }
20
21          double mediaI = acumuladorI / 10.0;
22          System.out.println("\n\nLa media es: " + mediaI);
```

Es evidente que un análisis de complejidad del ejemplo anterior indicaría que el problema es trivial y prácticamente no tiene complejidad apreciable. Sin embargo, que pasaría si queremos hacer lo mismo pero con fracciones, es decir obtener la fracción (numerador / denominador) promedio de 10 fracciones (numerador / denominador) generadas con valores aleatorios. Es evidente que la complejidad sería diferente al menos en la versión básica apoyada en las herramientas elementales del lenguaje Java. Desde las simples

operaciones de suma y división requeridas para el cálculo del promedio a la necesidad en el caso de las fracciones de simplificar para no obtener una fracción con números enormes.

La idea entonces es programar una clase fracción, que resuelva que permita crear objetos de tipo fracción y que resuelva las operaciones existentes ya para los números enteros en java y, de esta forma, lograr aproximar la complejidad de los problemas en esencia iguales pero que sin embargo, no son iguales como discutimos previamente.

Primeros pasos

Para hacer eso, los lenguajes orientados a objetos (como Java) usan descriptores de entidades conocidos como *clases*. Básicamente, una *clase* es la descripción de una entidad u objeto de forma que luego esa descripción pueda usarse para crear muchos objetos que respondan a la descripción dada. Para establecer analogías, se puede pensar que una clase se corresponde con el concepto de *tipo de dato* de la programación estructurada tradicional, y los objetos creados a partir de la clase (llamados *instancias* en el mundo de la *POO*) se corresponden con el concepto de *variable* de la programación tradicional. Así como el *tipo* es uno solo y describe la forma que tienen todas las muchas *variables* de ese tipo, la *clase* es única y describe la forma y el comportamiento de los muchos *objetos* de esa clase.

Para describir objetos que responden a las mismas características de forma y comportamiento, se declara una *clase*. La definición de una clase incluye esencialmente dos elementos:

- **Atributos:** Son *variables* que se declaran dentro de la clase, y sirven para indicar la *forma* de cada objeto representado por esa clase. Los atributos, de alguna manera, muestran lo que cada objeto *es*, o también, lo que cada objeto *tiene*.
- **Métodos:** Son funciones, procedimientos o rutinas declaradas dentro de la clase, usados para describir el *comportamiento* de los objetos descriptos por esa clase. Los métodos, de alguna manera, muestran lo que cada objeto *hace*.

Como sabemos, una aplicación o programa Java típico debe incluir un método especial, llamado *main()*. Ese método es el primero que se ejecuta cuando se pide lanzar la aplicación, y desde allí se administra la participación de cada *instancia* u *objeto* creado.

Al desarrollar una aplicación en Java, el programador creará por lo general varias clases que luego se usarán a su vez para crear objetos que trabajarán juntos. Es común en ese sentido declarar una clase cuyo único objetivo sea contener al método *main()*. En el contexto de nuestro curso, eso es lo que hemos hecho y seguiremos haciendo mediante nuestra convención de declarar una clase con el nombre *App* o el que se convenga, para que sea ella la que contenga a *main()*. No obstante, digamos que *main()* podría estar incluido en cualquier clase, y que incluso cada clase del proyecto podría tener un *main()*. No es obligatorio que una clase se llame *App*, y que esta deba contener a *main()*. Es sólo una convención de trabajo en el curso.

La definición de una clase en Java se hace mediante la palabra reservada *class*, seguida de un par de llaves que delimitan su contenido. Es común (pero no obligatorio) que los atributos de la clase se declaren antes que los métodos. El conjunto de atributos y métodos de una clase se conoce como el conjunto de *miembros* de la clase. En nuestro ejemplo vamos a requerir la clase Fracción para resolver en ella los problemas asociados a una fracción, pero, que luego permita la creación de tantas fracciones concretas como haga falta en el programa:

```
3 public class Fraccion
4 {
5     // Atributos (Datos miembro)
6     private int numerador;
7     private int denominador;
8 }
```

Los atributos de la clase son variables, que pueden ser de tipo primitivo o pueden ser objetos de otras clases. De hecho, *String* es una clase de Java y no un tipo primitivo, por lo cual si algún atributo de la clase *Fraccion* fuera un *String*, tendríamos un atributo que es un objeto de otra clase, y así los objetos comienzan a colaborar entre ellos... Para la clase *Fraccion*, mínimamente sería necesario contar con un atributo que indique el numerador y otro que indique el denominador. La se vería como la que mostramos en la imagen anterior.

Uno de los principios de la POO es el llamado *Principio de Ocultamiento*, por el cual se sugiere que al definir una clase, no se permita que los atributos sean accesibles en forma directa desde el exterior de la clase, sino que se usen métodos de la misma para consultar sus valores o modificarlos. La idea detrás del *Principio de Ocultamiento* es que el programador que use una clase predefinida no deba preocuparse por los detalles de implementación interna de la clase, sino que simplemente use los métodos que la misma provee y se maneje en un nivel de abstracción más alto. Así, si usamos objetos de la clase *String*, no debemos preocuparnos por cómo está representada esa cadena dentro del objeto. Sabemos que dentro hay una cadena implementada de alguna forma convincente, y la clase brinda todos los métodos para hacer lo que necesitemos hacer. Además, el Principio de Ocultamiento permite que la clase controle qué valores tiene cada atributo y de qué forma esos valores deberían cambiar. Si se permite el acceso a un atributo de la clase desde fuera de ella, podría provocarse que un valor incorrecto, no validado, sea asignado en ese atributo haciendo que desde allí en adelante algún proceso interno de la clase falle, por ejemplo si permitimos que el denominador de la fracción sea cero...

Modificadores de Acceso

La forma que Java provee para que un programador obligue a respetar el Principio de Ocultamiento son los llamados *modificadores de acceso*. Se trata de ciertas palabras reservadas que colocadas delante de la declaración de un atributo o de un método de una clase, hacen que ese atributo o ese método tengan accesibilidad más amplia o menos amplia desde algún método que no esté en la clase. Así, los calificadores de acceso en Java pueden ser cuatro: *public*, *private*, *protected*, y "*default*" (este último no es una palabra reservada: es el estado en el que queda un miembro si no se antepone ninguno de los otros tres calificadores anteriores):

- **public**: un miembro público es accesible tanto desde el interior de la clase (por sus propios métodos), como desde el exterior de la misma (por métodos de otras clases).
- **private**: sólo es accesible desde el interior de la propia clase, usando sus propios métodos.
- **protected**: aplicable en contextos de herencia (tema que veremos más adelante), hace que un miembro sea público para sus clases derivadas y para clases en el mismo paquete, pero los hace privados para el resto. ☐!!!

- "**default**" : un miembro que no sea marcado con ningún calificador de acceso, asume estatus de acceso *por defecto*, lo cual significa que será público para clases en el mismo paquete, pero privado para el resto. Notar (otra vez) que la palabra "default" en realidad no es una palabra reservada, sino sólo el nombre del estado en que queda el miembro.

Por todo lo expresado, es que en la clase *Fraccion* se han declarado los atributos como *private*. Entre los métodos de una clase, el más característico es el llamado método *constructor*. Un constructor es un método cuyo objetivo básico es el de inicializar los atributos de un objeto en el momento en que ese objeto o instancia se crea. Como veremos en breve, un objeto se crea en Java usando un operador del lenguaje llamado *new*, y el constructor se invoca *automáticamente* al crear con *new* una instancia de la clase.

Un constructor lleva el mismo nombre que la clase que lo contiene. No se debe indicar ningún tipo devuelto para él (ni siquiera *void*), y puede recibir parámetros como un método normal. Por otra parte, tanto los constructores como cualquier otro método de la clase pueden ser *sobrecargados*. Eso significa que se pueden definir *varias versiones del mismo método*. El compilador distingue entre las diversas sobrecargas de un método *por la forma de su lista de parámetros*. Por lo tanto, si un método tiene varias versiones definidas en una clase, las distintas versiones deben diferir en la cantidad de parámetros, en el tipo de los parámetros, o en ambas características. Notar que el cambio en el tipo devuelto por el método no define una nueva sobrecarga: solo las formas diversas en la lista de parámetros. En base a esto, la clase *Fracción* podría tener este aspecto si agregamos algunos constructores:

```
3  public class Fraccion
4  {
5      // Atributos (Datos miembro)
6      private int numerador;
7      private int denominador;
8
9      // Constructores
10     new *
11     public Fraccion(int num, int den)
12     {
13         numerador = num;
14         setDenominador(den);
15     }
16
17     new *
18     public Fraccion(int num)
19     {
20         this(num, den: 1);
21     }
22
23     new *
24     public Fraccion(Fraccion aCopiar) {
25         this(aCopiar.numerador, aCopiar.denominador);
26     }
27
```

Notar que en primero de los constructores la lista de parámetros está compuesta de dos números enteros, que son utilizados para inicializar el numerador y el denominador, también se puede observar que para el caso del denominador estamos usando un comportamiento propio de la clase que revisaremos en el siguiente apartado.

En el segundo caso tenemos un constructor que recibe un solo número entero y entienden en este caso la inicialización de una fracción impropia asignando en el denominador un 1. En este ejemplo también descubrimos el uso de la palabra reservada *this* que es la referencia implícita al propio objetos para invocación de métodos y atributos, y en el caso que aquí se puede observar el llamado explícito a otro constructor. La restricción para hacer este llamado es que debe realizarse si o sí como primera línea de código de un constructor y no puede ser utilizada en ningún otro lado para invocación de constructores.

Finalmente la tercera sobrecarga del constructor recibe como parámetro un objeto de la clase Fracción y provoca una nueva facción con los mismos valores de numerador y denominador que la fracción dada. En el apartado siguiente veremos la utilidad de este constructor y su necesidad para sortear la situación de la asignación de referencias, más allá de que en java el proceso correcto para provocar esta copia se denomina clonación y será analizado en clases posteriores.

Ahora bien, ahora tenemos una clase que es capaz de contener los datos del numerador y denominador de una fracción pero no tenemos aún una razón de ser para la misma, es decir no tenemos ningún método que provoque un comportamiento real, es decir que sea capaz de responder a un mensaje propio del concepto de una fracción.

Vamos a agregar 2 métodos extra en la clase *Fraccion* antes de comenzar a utilizar objetos de tipo fracción, por un lado el comportamiento más elemental de cualquier fracción que es devolver el valor real de la relación y en segundo lugar un comportamiento que como veremos más adelante toda clase java puede implementar para transformar la instancia de un objeto de la clase en una cadena de caracteres. Con estos métodos nuestra clase Fracción agregaría:

```
new *  
84     public double valorReal()  
85     {  
86         double resp = numerador / (double) denominador;  
87         return resp;  
88     }  
89  
90     new *  
91 ⚡↑  public String toString()  
92     {  
93         return "[" +this.numerador + "/" + this.denominador + "]";  
94     }  
95
```

Referencias y Creación de Objetos

Así como está definida, la clase *Fraccion* está todavía muy incompleta pero ya puede usarse para crear objetos y mostrar la forma básica de usar un constructor. Supongamos entonces que se quieren crear dos o tres objetos de la clase *Fraccion* para comenzar a trabajar con ellos. El método *main()* de nuestra clase *App* podría hacerlo:

```

5 ► public static void main(String[] args)
6 {
7     //Declaro una referencia a Fraccion
8     Fraccion f1 = null;
9
10    // Instancio el objeto.
11    // f1 = new Fraccion(); // Utiliza el constructor por defecto y no compila porque al agregar constructor con
12    // parámetros el constructor por defecto deja de existir
13    System.out.println(f1);
14
15    // f1.numerador = 3; // compila porque los atributos están definidos com privados.
16    // f1.denominador = 5;
17
18    // System.out.println("\tValor real: " + f1.valorReal()); // provocaría un error ya que aún no creamos el objeto.
19
20    f1 = new Fraccion( num: 2, den: 3); // creamos la fracción 2 tercios
21    System.out.println(f1); // al imprimir invocamos el método toString, notar que se invoca de manera implícita
22    System.out.println(f1.valorReal()); // mostramos el valor real de la fracción al usar el comportamiento para eso
23
24    System.out.println("Fin!");
25
26 }

```

Note el uso del operador *new* para crear los objetos: ese operador crea un objeto de la clase que se le indique, cada vez que se lo usa. Pero en cada creación, *new* llama a algún constructor que figure en la clase del objeto que se pide crear, y la decisión de cuál constructor llamar depende de la lista de parámetros actuales que se pase a ese constructor al invocarlo: puede verse que al crear el objeto *f1*, se está invocando al primero de los constructores pues la lista de parámetros actuales es de dos números enteros.

En síntesis: el primer *new* crea un objeto que será manejado por la variable *f1*, y en ese objeto los atributos *numerador* y *denominador* quedan valiendo los valores { 2, 3} respectivamente.

Una vez que se han creado objetos de una clase (por ejemplo, lo que vimos en el método *main()*), estos objetos pueden comenzar a *invocar métodos* para que se apliquen sobre sus atributos. La forma de hacerlo, consiste en usar la variable que maneja al objeto y *colocar luego de ella un punto, seguido del nombre del método* que se quiere invocar para ese objeto como podemos observar al utilizar el método *valorReal* de la fracción referenciada por *f1*.

Se dice que el objeto desde el cual se invoca al método está *llamando* al método, o también se dice que a ese objeto se le está *pasando un mensaje*. En el ejemplo, el objeto *f1* está llamando al método *valorReal()* (o también decimos que *f1* recibe el mensaje de retornar su valor real). La salida del ejemplo anterior sería:

```

↑ "C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBra
↓ null
→ [2/3]
→ 0.6666666666666666
→ Fin!

```

Métodos de acceso de lectura y escritura

Sigamos entonces con la clase *Fraccion*. Si los atributos de una clase se definen privados, entonces la clase puede proveer métodos que permitan el acceso al valor de esos atributos, tanto para consultar su valor como para modificarlo (si es decisión del programador de la clase permitir esas operaciones) notar que al implementar estos métodos estamos diseñando el comportamiento de la clase. Por ejemplo en la clase *fracción* no podemos permitir que se modifiquen de manera independiente el numerador o el denominador

puesto que al modificar uno y solo uno de los atributos en realidad estaríamos en presencia de una nueva fracción y por lo tanto de un nuevo objeto aunque estos es conceptual.

La especificación de Java Beans en Java define que para esos métodos intervengan las palabras "get" (para los métodos de consulta) y "set" para los métodos modificadores, junto al nombre del atributo al que se quiere proveer acceso con la primera letra en mayúscula. Así, si un atributo se llama "clave", el método de consulta para el mismo podría llamarse "getClave()" y el modificador "setClave()". En la clase *Fraccion*, incluimos ahora este conjunto de *métodos de acceso*.

```
new ^  
27     public int getNumerador() {  
28         return numerador;  
29     }  
30  
31     new *  
32         public int getDenominador() {  
33             return denominador;  
34         }  
35  
36     new *  
37         private void setDenominador(int den) {  
38             if (den != 0)  
39                 denominador = den;  
40             else  
41                 denominador = 1;  
42         }  
43
```

Note que los métodos de consulta de la clase se ha marcado como *public* (incluso los constructores). Esto es consecuencia directa de respetar el *Principio de Ocultamiento*, que en última instancia aconseja declarar privados a los atributos y públicos a los métodos de una clase. Los métodos de acceso son muy sencillos: cada uno de los métodos tipo *get* retorna el valor del atributo con el cual se asocia, el método tipo *setDenominador* cambia el valor de ese atributo teniendo en cuenta el control a realizar. No hemos implementado el método *setNumerador* ya sería privado al igual que *setDenominador* y que no hay control necesario y podemos desde dentro de la clase acceder directamente al atributo. Cabe aclarar que en algunos equipos se conviene en programar siempre todos los métodos *set* aunque no cumplan función alguna y solo para mantener coherencia.

Finalmente, digamos que toda clase Java en última instancia hereda o se define a partir de otra muy general llamada *Object*, la cual provee ya definidos una serie de métodos elementales. Varios de esos métodos se

usan tal como vienen desde *Object*, pero algunos deberían ser redefinidos por cada clase. El método *toString()* es uno de ellos, y se usa para retornar una cadena de caracteres con el contenido del objeto invocante, de forma que sea adecuadamente visualizable en un dispositivo de salida. Si no se redefine, el método *toString()* retorna una cadena con el nombre de la clase a la cual pertenece el objeto, más la dirección de memoria de ese objeto en formato hexadecimal. En general, nuestras clases deberían contar con una versión propia del método *toString()*, lo cual es normalmente fácil de hacer. Mostramos la clase *Fraccion* completa con ese método incluido al final:

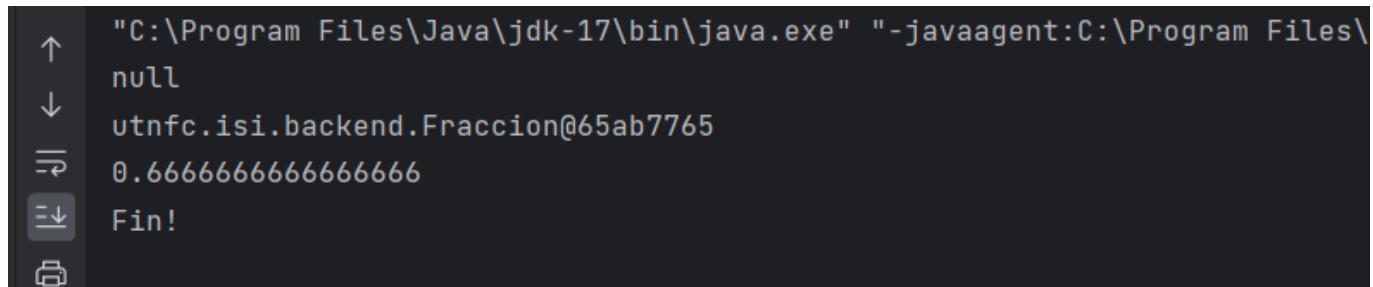
El método *toString()* es muy especial en Java. El lenguaje asume *automáticamente* que se está invocando a ese método en cualquier contexto en el cual se use un objeto pero se requiera una conversión a *String* del mismo. Eso significa que la línea:

```
System.out.println("Fracción f1: " + f1.toString());
```

también puede escribirse así, y el resultado es exactamente el mismo:

```
System.out.println("Fracción f1: " + f1);
```

Pero aún menos intuitivo es lo que pasa si en la clase fracción comentamos el método *toString* o le cambiamos el nombre de alguna forma. La salida anterior pasaría a ser como sigue:



```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\utnfc.isi.backend.Fraccion@65ab7765
null
utnfc.isi.backend.Fraccion@65ab7765
0.6666666666666666
Fin!
```

Lo que está pasando aquí es que al cambiar el nombre del método en la clase *Fracción* el método *toString* que está usando java para obtener la versión de cadena de caracteres del objeto es el existente en la clase *Object* y por lo tanto muestra solo los elementos disponibles dentro de la clase *object* que son el nombre de la clase y la dirección de memoria de la referencia.

Volviendo al promedio de 10 fracciones

Bueno, habiendo llegado hasta aquí tenemos nuestra clase *Fracción* con los métodos que ya hemos analizado y solo nos resta agregar los demás métodos a la clase para permitir que una fracción se pueda sumar, restar, multiplicar o dividir por otra fracción además de la posibilidad de simplificarse. La versión de fracción que acompaña el presente material ya tiene estos métodos implementados.

Ahora nos proponemos analizar la complejidad del problema planteado al inicio pero contando con la clase *Fracción* creada:

```

25
26     // Ahora con Fracciones
27     System.out.println("Ahora con fracciones");
28     Fraccion acumuladorF = new Fraccion( num: 0 );
29     for(int i = 0; i < 10; i++)
30     {
31 >     // ...
32         Fraccion aux = new Fraccion((int)(Math.random() * 7 + 1), (int)(Math.random() * 9 + 1));
33
34         System.out.println("Generado: " + aux);
35         acumuladorF = acumuladorF.sumarA(aux);
36     }
37
38     Fraccion mediaF = acumuladorF.dividirPor( num: 10 );
39     System.out.println("\n\nLa media es: " + mediaF);
40
41 }
42
43 }
44

```

Como podemos observar fuera de la necesidad de 2 números aleatorios en lugar de uno para la creación de cada fracción la complejidad entre este fragmento y el inicial asociado números enteros es similar puesto que ya hemos resuelto las problemáticas específicas de las fracciones dentro de la clase fracción.

[!Note]

Objetos en base a Objetos

Acompañando este apunte además del ejemplo de fracciones que hemos analizado, agregamos un ejemplo más donde unos > objetos se crean en base a otros objetos más simples, vale su análisis para comprender el manejo de la complejidad por > capas que es lo que en general se persigue cuando realizamos diseños orientados a objetos.

❖ Reducción de Código Repetitivo con Lombok (Boilerplate Reduction)

En el desarrollo de aplicaciones con Java, uno de los problemas más comunes es la cantidad de código repetitivo que se debe escribir en las clases, incluso para tareas simples como declarar atributos, generar constructores, **getters**, **setters**, o redefinir **toString()** y **equals()**.

A este tipo de código, que es necesario pero no aporta lógica de negocio, se lo llama **boilerplate**. No es exclusivo de Java, pero en Java es especialmente visible.

En este contexto, la comunidad de Java ha desarrollado herramientas para reducir ese tipo de código repetitivo, y una de las más populares y aceptadas es **Lombok**.

¿Qué es Lombok?

Lombok es una biblioteca para Java que permite eliminar la necesidad de escribir código repetitivo mediante **anotaciones**. Con solo agregar una anotación sobre una clase (**@Getter**, **@Setter**, **@AllArgsConstructor**, **@ToString**, etc.), se genera automáticamente el código correspondiente en tiempo de compilación.

Este comportamiento es transparente: el código generado no aparece en los archivos **.java**, pero sí queda disponible en los archivos **.class**, y por lo tanto puede usarse como si estuviera escrito a mano.

¿Cómo funciona Lombok internamente?

Lombok se basa en un mecanismo conocido como **procesamiento de anotaciones (annotation processing)**, un estándar de Java que permite analizar y modificar el código durante el proceso de compilación. Para eso:

1. Utiliza una librería interna que intercepta el compilador (**javac**) y modifica el árbol de sintaxis abstracta (AST) del código fuente.
2. Inyecta automáticamente el código necesario según las anotaciones utilizadas.
3. El bytecode generado por el compilador incluirá getters, setters, constructores, etc., aunque no estén escritos explícitamente.

Este proceso ocurre **en tiempo de compilación**, por esto antes del tiempo de compilación el código que escribimos va a estar haciendo referencia a porciones de código inexistentes esto hace que también necesitemos extensiones de desarrollo y requiere que el IDE o entorno de compilación esté correctamente configurado para reconocer y aplicar las anotaciones de Lombok.

¿Cómo se configura Lombok?

Para poder utilizar Lombok:

1. **Agregar la dependencia** en el proyecto (por ejemplo en **pom.xml** si usás Maven):

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.30</version>
    <scope>provided</scope>
</dependency>
```

2. **Configurar el IDE** (IntelliJ, VS Code, Eclipse, etc.) para que reconozca Lombok:

- Por ejemplo, en IntelliJ: instalar el plugin Lombok desde el Marketplace.
 - Asegurarse de que “Enable annotation processing” esté activado en los ajustes del proyecto.
- O en VS Code no hace falta nada especial porque el soporte para lombok ya está incluido en el Java Extension Pack

Explorando las principales anotaciones de Lombok

A continuación, presentamos un conjunto de anotaciones que ofrece Lombok para reducir el código repetitivo (boilerplate) en Java. Para cada una mostramos su utilidad, un ejemplo básico y cuál sería el equivalente manual.

@Getter y @Setter

Estas anotaciones generan automáticamente los métodos **get...()** y **set...()** para los atributos de una clase.

```
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class Persona {
    private String nombre;
    private int edad;
}
```

👉 Equivalente manual:

```
public class Persona {
    private String nombre;
    private int edad;

    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }

    public int getEdad() { return edad; }
    public void setEdad(int edad) { this.edad = edad; }
}
```

También se pueden aplicar a nivel de atributo para más control.

1. ¿Se pueden usar @Getter y @Setter por atributo en lugar de toda la clase?

Lombok permite aplicar sus anotaciones tanto a nivel de clase como a nivel de atributo, y esto es útil cuando:

- Si queremos exponer sólo algunos atributos (e.g., exponer getId() pero ocultar el resto).
- Si queremos que un atributo sea de sólo lectura o escritura.
- Si queremos aplicar una anotación específica sin que afecte toda la clase.

❖ Ejemplo:

```
public class Persona {

    @Getter
    private String nombre;

    @Getter @Setter
    private int edad;

    private String clave; // sin getter ni setter
}
```

⌚ Resultado:

- Se genera `getNombre()`
- Se genera `getEdad()` y `setEdad(...)`
- No se genera ningún método para clave

```
public class Persona {  
    private String nombre;  
    private int edad;  
  
    public String getNombre() { return nombre; }  
    public void setNombre(String nombre) { this.nombre = nombre; }  
  
    public int getEdad() { return edad; }  
    public void setEdad(int edad) { this.edad = edad; }  
}
```

@ToString

Genera automáticamente el método `toString()` incluyendo todos los atributos no `static` ni `transient`.

```
import lombok.ToString;  
  
@ToString  
public class Producto {  
    private String nombre;  
    private double precio;  
}
```

⌚ Equivalente manual:

```
public class Producto {  
    private String nombre;  
    private double precio;  
  
    public String toString() {  
        return "Producto(nombre=" + nombre + ", precio=" + precio + ")";  
    }  
}
```

@NoArgsConstructor y @AllArgsConstructor

Generan automáticamente:

- Un constructor vacío (`@NoArgsConstructor`)
- Un constructor con todos los atributos (`@AllArgsConstructor`)

```
import lombok.NoArgsConstructor;
import lombok.AllArgsConstructor;

@NoArgsConstructor
@AllArgsConstructor
public class Usuario {
    private String email;
    private String clave;
}
```

☞ Equivalente manual:

```
public class Usuario {
    private String email;
    private String clave;

    public Usuario() {}

    public Usuario(String email, String clave) {
        this.email = email;
        this.clave = clave;
    }
}
```

@EqualsAndHashCode

Genera `equals()` y `hashCode()` considerando todos los atributos.

Este comportamiento puede modificarse agregando el parámetro (`of = "nombreAtributo"`) a la anotación

```
import lombok.EqualsAndHashCode;

@EqualsAndHashCode
public class Documento {
    private String tipo;
    private int numero;
}
```

☞ Equivalente manual (simplificado):

```
public class Documento {  
    private String tipo;  
    private int numero;  
  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;  
        Documento doc = (Documento) o;  
        return numero == doc.numero && tipo.equals(doc.tipo);  
    }  
  
    public int hashCode() {  
        return Objects.hash(tipo, numero);  
    }  
}
```

@Data

Ahora la cosa se pone interesante, hasta acá vimos configuraciones independientes de cada aspecto y eso es completamente viable, sin embargo, ahora veremos configuraciones que incluyen a las anteriores, en este caso `@Data` genera automáticamente:

- `@Getter` y `@Setter`
- `@ToString`
- `@EqualsAndHashCode`
- `@RequiredArgsConstructor` (Incluye solo los atributos `final` o marcados como `@NonNull`)

Ideal para POJOs donde queremos todos los métodos básicos, por ejemplo:

```
import lombok.Data;  
  
@Data  
public class Alumno {  
  
    private final int legajo;  
  
    @NonNull  
    private String nombre;  
  
    private int idCurso;  
}
```

👉 Equivalente manual (simplificado):

```
import java.util.Objects;

public class Alumno {

    private final int legajo;
    private String nombre;
    private int idCurso;

    // Constructor requerido (legajo y nombre)
    public Alumno(int legajo, String nombre) {
        if (nombre == null) {
            throw new NullPointerException("nombre is marked non-null but is null");
        }
        this.legajo = legajo;
        this.nombre = nombre;
    }

    // Getter para legajo (no hay setter porque es final)
    public int getLegajo() {
        return legajo;
    }

    // Getter y Setter para nombre
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        if (nombre == null) {
            throw new NullPointerException("nombre is marked non-null but is null");
        }
        this.nombre = nombre;
    }

    // Getter y Setter para idCurso
    public int getIdCurso() {
        return idCurso;
    }

    public void setIdCurso(int idCurso) {
        this.idCurso = idCurso;
    }

    // equals() y hashCode() considerando todos los campos
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Alumno)) return false;
        Alumno alumno = (Alumno) o;
        return legajo == alumno.legajo &&
```

```

        idCurso == alumno.idCurso &&
        Objects.equals(nombre, alumno.nombre);
    }

    @Override
    public int hashCode() {
        return Objects.hash(legajo, nombre, idCurso);
    }

    // toString()
    @Override
    public String toString() {
        return "Alumno(legajo=" + legajo +
            ", nombre=" + nombre +
            ", idCurso=" + idCurso + ")";
    }
}

```

@Builder

Permite generar un patrón Builder automáticamente.

```

import lombok.Builder;

@Builder
public class Cliente {
    private String nombre;
    private String direccion;
    private int edad;
}

```

👉 Uso:

```

Cliente c = Cliente.builder()
    .nombre("Juan")
    .direccion("Av. Siempreviva 123")
    .edad(40)
    .build();

```

Este patrón es especialmente útil cuando hay múltiples campos opcionales o con valores por defecto debido a la clase builder que se genera, en este punto no profundizamos el tema pero lo podemos sumar en ejemplos más adelante.

@Value

Anotación para clases inmutables:

- Hace los atributos final
- Clase final
- Genera constructor, getters, `equals()`, `hashCode()` y `toString()`

```
import lombok.Value;

@Value
public class Punto {
    int x;
    int y;
}
```

Este último ejemplo agrega un concepto que hoy java introdujo en el lenguaje de programación a través de la declaración de `record`

📋 Anexo I: `@Value` de Lombok vs `record` de Java

¿Qué es un `record` en Java?

Un `record` es una **nueva estructura de clase inmutable**, pensada específicamente para modelar **objetos portadores de datos (data carriers)** de manera **concisa, segura, y legible**.

💡 Un `record`:

- Genera automáticamente:
 - `constructor`
 - `getters` (sin el prefijo `get`)
 - `equals()`, `hashCode()`, `toString()`
- Hace los campos `final` e inmutables
- No permite herencia (pero sí puede implementar interfaces)

❖ **Sintaxis:**

```
public record Alumno(int legajo, String nombre, int idCurso) {}
```

¿Qué hace `@Value` en Lombok?

```
import lombok.Value;

@Value
public class Alumno {
    int legajo;
    String nombre;
    int idCurso;
}
```

@Value en Lombok:

- Declara automáticamente todos los atributos como `private final`
- Genera:
 - Constructor con todos los campos
 - Getters (`getNombre()`, etc.)
 - `equals()`, `hashCode()`, `toString()`
- Hace la clase `final` (no se puede extender)
- Requiere Lombok y configuración de anotaciones en el proyecto

Comparativa detallada

Característica	@Value de Lombok	record de Java
Inmutabilidad	<input checked="" type="checkbox"/> (con <code>final</code> y sin setters)	<input checked="" type="checkbox"/> (todos los campos <code>final</code> por diseño)
Generación de métodos	Constructor, getters, equals, etc.	Constructor, <i>accessors</i> , equals, etc.
Herencia	No se puede extender (es <code>final</code>)	No permite herencia
Dependencia externa	Requiere Lombok	Parte del JDK desde Java 16
Accesores (<code>getX()</code>)	Sí (estilo JavaBeans)	No (<code>nombre()</code> en lugar de <code>getNombre()</code>)
Campos <code>private</code>	Sí	No → campos son públicos de solo lectura
Anidamiento	Admite clases internas normales	Requiere clases <code>static</code> anidadas
Personalización parcial	Sí (constructores, validaciones)	Sí (compact constructors, validations)
Patrón DTO / POJO	Muy usado	Ideal reemplazo moderno

¿Cuándo usar uno u otro?

Necesidad	Recomendación
Proyecto con Lombok ya integrado	Usá <code>@Value</code>
Proyecto Java moderno (16+) sin deps	Usá <code>record</code>
Necesitás compatibilidad anterior	<code>@Value</code> (Java 8+)
APIs públicas estilo JavaBeans	<code>@Value</code> (con getters)
Modelado de datos puro, simple	<code>record</code>

Ejemplo completo comparado

Con `@Value`

```
@Value
public class Alumno {
    int legajo;
```

```
String nombre;
int idCurso;
}
```

Con record

```
public record Alumno(int legajo, String nombre, int idCurso) {}
```

Uso

```
Alumno a = new Alumno(123, "Sofía", 4);
System.out.println(a.nombre()); // en record
System.out.println(a.getNombre()); // en @Value
```

Conclusión

Si iniciamos de cero una aplicación con las últimas versiones de java, los **record** son, quizás, la **alternativa oficial** a las clases inmutables típicas. Son más livianos, integrados y expresivos.

Si trabajás con **versiones anteriores** o necesitás **estilo JavaBeans (get/set)** por herencia de dependencias o implementación de frameworks existentes, **@Value** sigue siendo una excelente opción, especialmente en proyectos con Lombok y Spring.

En la cátedra Backend de Aplicaciones hemos decidido optar por usar lombok para la reducción del boilerplate por lo que los ejemplos de la cátedra se implementarán en base a esa estructura.

📝 Refactorización completa de la clase Fracción usando Lombok

A modo de cierre de este apunte, proponemos volver a construir la clase Fracción que utilizamos para hacer el seguimiento de los conceptos básicos de la implementación del paradigma de programación orientado a objetos en Java. En esta oportunidad, la reimplementaremos utilizando Lombok, con el objetivo de ilustrar cómo esta herramienta permite reducir el boilerplate sin perder claridad en el diseño, y al mismo tiempo reforzar los conceptos fundamentales de encapsulamiento, constructores, validaciones y comportamiento orientado a objetos.

En este documento se presenta una **implementación completa y equivalente** de la clase **Fraccion** construida inicialmente de forma manual, pero ahora **aprovechando las anotaciones de Lombok**. Se busca demostrar cómo se puede lograr el mismo resultado funcional con menos código repetitivo, conservando legibilidad, buenas prácticas y validaciones.

Parte 1 - Reducción básica con **@Data**

```
import lombok.Data;

@Data
public class Fraccion {
    private int numerador;
    private int denominador;
}
```

Esta versión básica con `@Data` reemplaza la necesidad de:

- `getNumerador()`, `getDenominador()`
- `setNumerador()`, `setDenominador()`
- `toString()`, `equals()`, `hashCode()`

Pero **todavía le faltan**:

- Validaciones del constructor
- Cálculo de MCD y simplificación
- Métodos de operaciones (suma, resta, etc.)

Parte 2 - Versión completa con comportamiento equivalente

```
import lombok.Data;
import lombok.NonNull;

@Data
public class Fraccion {
    private int numerador;
    private int denominador;

    public Fraccion(int numerador, @NonNull int denominador) {
        if (denominador == 0) {
            throw new ArithmeticException("El denominador no puede ser cero");
        }
        this.numerador = numerador;
        this.denominador = denominador;
        simplificar();
    }

    public double valorReal() {
        return (double) numerador / denominador;
    }

    private int mcd(int a, int b) {
        return b == 0 ? a : mcd(b, a % b);
    }

    private void simplificar() {
        int divisor = mcd(Math.abs(numerador), Math.abs(denominador));
    }
}
```

```
        numerador /= divisor;
        denominador /= divisor;
    }

    public Fraccion sumar(Fraccion otra) {
        return new Fraccion(
            this.numerador * otra.denominador + otra.numerador * this.denominador,
            this.denominador * otra.denominador);
    }

    public Fraccion restar(Fraccion otra) {
        return new Fraccion(
            this.numerador * otra.denominador - otra.numerador * this.denominador,
            this.denominador * otra.denominador);
    }

    public Fraccion multiplicar(Fraccion otra) {
        return new Fraccion(
            this.numerador * otra.numerador,
            this.denominador * otra.denominador);
    }

    public Fraccion dividir(Fraccion otra) {
        if (otra.numerador == 0) {
            throw new ArithmeticException("No se puede dividir por una fracción
con numerador cero");
        }
        return new Fraccion(
            this.numerador * otra.denominador,
            this.denominador * otra.numerador);
    }
}
```

⌚ Análisis y comparación

- `@Data` se encarga de los métodos de acceso, comparación y representación textual.
- El constructor se define manualmente para incluir validación del denominador.
- Se mantiene la lógica de simplificación como parte del constructor.
- Se agregan los métodos de operaciones: `sumar`, `restar`, `multiplicar` y `dividir`.

🎓 Esta clase puede ahora utilizarse en tests, estructuras de colecciones, o para ejemplificar refactorización con herramientas modernas del ecosistema Java.

Esto permite **cerrar el ciclo de aprendizaje** desde lo manual y detallado hacia lo productivo y moderno.

Apunte 06 - Vectores y Manejo de Excepciones

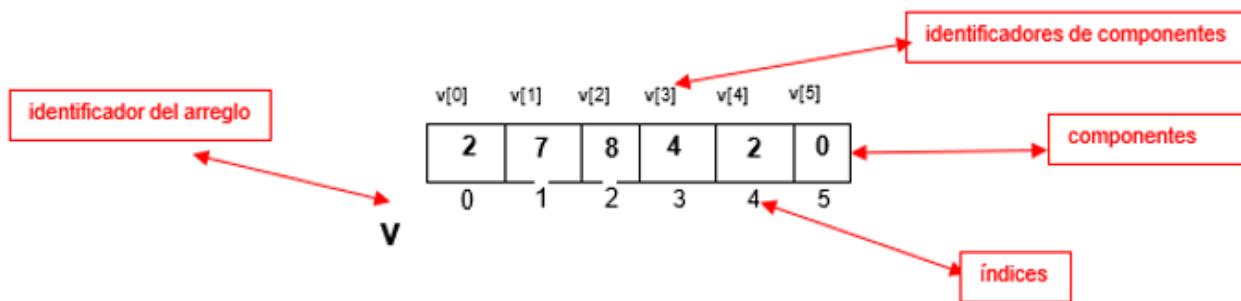
Arreglos en Java

La estructura de datos nativa de Java es la que se conoce como arreglo unidimensional o vector. Se trata de una colección de valores que deben ser del mismo tipo y que se organiza de tal forma que cada valor o componente individual es identificado automáticamente por un número designado como índice. El uso de los índices permite el acceso y posterior uso de cada componente en forma individual.

La cantidad de índices que se requieren para acceder a un elemento individual, se llama dimensión del arreglo. Los arreglos unidimensionales se denominan así porque solo requieren un índice para acceder a un componente. Por otra parte, dada la similitud que existe entre el concepto de arreglo unidimensional y el concepto de vector en Algebra, se suele llamar también vectores a los arreglos unidimensionales.

El siguiente gráfico muestra la forma conceptual de entender un arreglo unidimensional. Se supone que la variable arreglo se denomina *v* y que la misma está dividida en seis casilleros, de forma que en cada casillero puede guardarse un valor. Se supone también que el tipo de valor que puede guardarse en el arreglo *v* del ejemplo, es int.

Observar que cada casillero es automáticamente numerado con índices, los cuales en Java comienzan siempre a partir del cero: la primera casilla del arreglo siempre está relacionada con el subíndice con el valor cero, en forma automática. A partir del índice, cada elemento del arreglo *v* puede accederse en forma individual usando el identificador del componente: se escribe el nombre del arreglo, luego un par de corchetes, y entre los corchetes el valor del índice de la casilla que se quiere acceder. En ese sentido, el identificador del componente cuyo índice es 2, resulta ser *v[2]*:



Para declarar una variable de tipo arreglo en Java, hay que recordar primero que en Java los arreglos de cualquier dimensión son objetos, y por lo tanto deben ser creados con el operador new. Lo primero es declarar entonces una referencia con la cual se va a apuntar al arreglo que se quiere crear. En Java, esto se hace escribiendo el tipo de valor que se almacenará en el arreglo (ese tipo se conoce como el tipo base del arreglo), luego el nombre del arreglo y finalmente un par de corchetes vacíos:

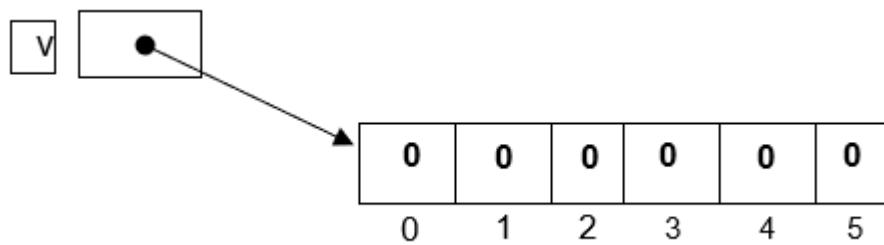
```
int[] v;
```

Al declarar una referencia de tipo arreglo, lo que se está haciendo es declarar una variable que luego será capaz de contener la dirección de memoria de un arreglo. El valor inicial de esa referencia es null, y el arreglo aún no existe en memoria:

Luego se usa el operador new para crear el objeto arreglo: se escribe new, seguido nuevamente del tipo base del arreglo, y otra vez el par de corchetes pero de forma que esta vez, se escribe dentro de ellos el tamaño del arreglo que se quiere crear:

```
int[] v = new int[6];
```

La instrucción anterior crea un arreglo de seis componentes capaces de almacenar cada uno un valor int, inicializa en cero cada casilla de ese arreglo, y retorna la dirección del mismo (que en este caso se almacena en la referencia v que declaramos antes):



Observar que si el tamaño de un arreglo es 6, entonces la última casilla del mismo lleva el índice 5 debido a que los índices comienzan siempre desde el 0. En un arreglo en Java, no existe una casilla cuyo índice coincida con el tamaño del arreglo.

Una vez que se creó el arreglo con new, se usa la referencia que lo apunta para acceder a sus componentes, colocando a la derecha de ella un par de corchetes y el índice del valor que se quiere acceder. Los siguientes son ejemplos de las operaciones que pueden hacerse con los componentes de un arreglo (tomamos como modelo el arreglo v anteriormente creado):

```

7  import java.util.Scanner;
8
9  public class Ejemplo01 {
10
11
12     public static void main(String[] args) {
13         int v[];
14         v = new int[6];
15         Scanner miScanner = new Scanner(System.in);
16         v[3] = 4;
17         v[1]++;
18         System.out.println( v[0] );
19         v[4] = v[1] - v[0];
20         v[5] = miScanner.nextInt();
21         v[2] = v[2] - 8;
22     }
23
24 }
25

```

Si se desea procesar un arreglo de forma que la misma operación se efectúe sobre cada uno de sus componentes, es normal usar un ciclo for, de forma se aproveche la variable de control del ciclo como índice para entrar a cada componente. Los siguientes esquemas muestran la forma de hacer una carga por teclado y una visualización por pantalla de un arreglo de seis componentes de tipo int:

```

import java.util.Scanner;

public class Ejemplo02 {
    public static void main(String[] args) {
        Scanner miScanner = new Scanner(System.in);
        int v[];
        v = new int[6];
        for( int i=0; i<6; i++ )
        {
            System.out.print("Ingrese v["+i+"]: ");
            v[i] = miScanner.nextInt();
        }
    }
}
for( int i=0; i<6; i++ )
{
    System.out.print(v[i]);
}

```

Un detalle interesante es que todo objeto arreglo en Java provee un atributo llamado length, que contiene el tamaño del arreglo tal como fue declarado al crear ese arreglo con new. Ese atributo es de naturaleza pública (public), por lo que puede accederse directamente mediante el identificador de la variable referencia que apunta al arreglo. Los dos ciclos anteriores, podrían escribirse así:

```

int v[];
v = new int[6];
for( int i=0; i<v.length; i++ )
{
    System.out.print("Ingrese v["+i+"]: ");
    v[i] = miScanner.nextInt();
}
for( int i=0; i<v.length; i++ )
{
    System.out.print(v[i]);
}

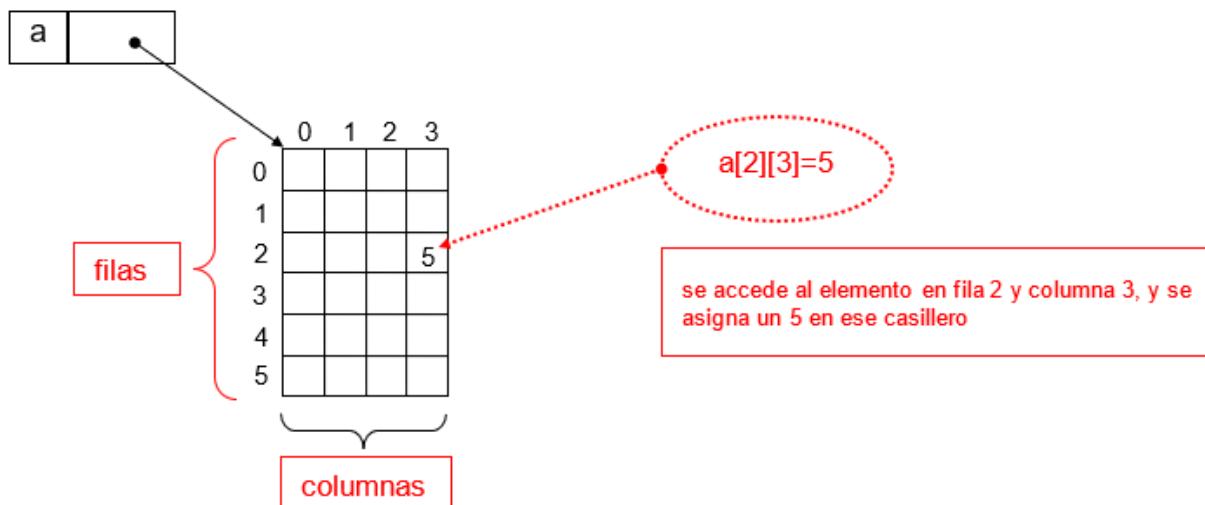
```

Creación y uso de matrices

Básicamente, un arreglo bidimensional o matriz es un arreglo cuyos elementos están dispuestos en forma de tabla, con varias filas y columnas. Aquí llamamos filas a las disposiciones horizontales del arreglo, y columnas a las disposiciones verticales.

Para entrar a un componente, debe darse el índice de la fila del mismo y también el índice de la columna. Como los índices requeridos son dos, el arreglo es de dimensión dos. El siguiente esquema ilustra la manera de declarar y crear un arreglo bidimensional de componentes int en Java, la forma conceptual de representarlo, y la manera de acceder a sus componentes:

```
int a [][]; // se declara una referencia al arreglo, con valor inicial
null.
a = new int [ 6 ][ 4 ]; // se crea el arreglo, con 6 filas y 4 columnas.
a[2][3] = 5; // se accede a una casilla y se asigna un valor en ella.
```



Para declarar la referencia al arreglo se usan ahora dos pares de corchetes vacíos. Y para crear el arreglo con `new`, en el primer par de corchetes se escribe la cantidad de filas que se necesitan y en el segundo par se escribe la cantidad de columnas.

Observar que para acceder a un elemento, se coloca el nombre de la referencia al arreglo, luego el número de la fila del elemento que se quiere acceder, pero encerrado entre corchetes, y por último el número de la columna de ese elemento, también encerrado entre corchetes. Notar además, que en el lenguaje Java los arreglos de cualquier dimensión están basados en cero, lo cual significa que el primer índice de cada dimensión es siempre cero. En la figura anterior, puede verse que el arreglo tiene seis filas, pero numeradas del 0 (cero) al 5 (cinco), y cuatro columnas, numeradas del 0 (cero) al 3 (tres). No hay excepciones a esta regla, por lo cual debe tenerse cuidado de ajustar correctamente los ciclos para procesamiento de arreglos.

Para cargar por teclado una matriz `a` de n filas y m columnas (y en general, para procesar de forma secuencial una matriz), se pueden usar dos ciclos `for` anidados, de forma que el primero recorra las filas de la matriz, y el segundo las columnas:

```

int i, j;
System.out.println("Cargue los números del arreglo: ");
for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
    {
        System.out.print("mat[" + i + "][" + j + "]:");
        mat[i][j] = miScanner.nextInt();
    }
}

```

La idea básica del proceso aquí definido es que la variable *i* del ciclo más externo se usa para indicar qué fila se está procesando en cada vuelta. Dado un valor de *i*, se dispara otro ciclo controlado por *j*, cuyo objetivo es el de recorrer todas las columnas de la fila indicada por *i*. Notar que mientras avanza el ciclo controlado por *j* permanece fijo el valor de *i*. Sólo cuando corta el ciclo en *j*, se retorna al ciclo en *i*, cambiando ésta de valor y comenzando por ello con una nueva fila. El proceso de recorrer de forma secuencial una matriz avanzando fila por fila empezando desde la cero, como aquí se describe, se denomina recorrido en orden de fila creciente.

Se pueden hacer recorridos de otros tipos si fuera necesario, simplemente cambiando el orden de los ciclos. Por ejemplo, el siguiente esquema realiza un recorrido en orden de fila decreciente: comienza con la última fila, y barre cada fila hacia atrás hasta llegar a la fila cero.

Si se desea un recorrido en orden de columna creciente (o decreciente), sólo deben invertirse los ciclos: el ciclo en *j* debe ir por fuera, y el ciclo en *i* debe ir por dentro. De esta forma, el valor de *j* no cambia hasta que el ciclo en *i* termine todo su recorrido. Sin embargo, no debe olvidarse que si queremos que *j* indique una columna, entonces *j* debe usarse en el segundo par de corchetes al acceder a la matriz. Y si la variable *i* va a indicar filas, entonces debe usarse en el primer par de corchetes. Esto es independiente del orden en que se presenten los ciclos para hacer cada recorrido:

```

int i, j;
System.out.println("Cargue los números del arreglo: ");
for (j = 0; j < m; j++)
    for (i = 0; i < n; i++)
    {
        System.out.print("mat[" + i + "][" + j + "]:");
        mat[i][j] = miScanner.nextInt();
    }
}

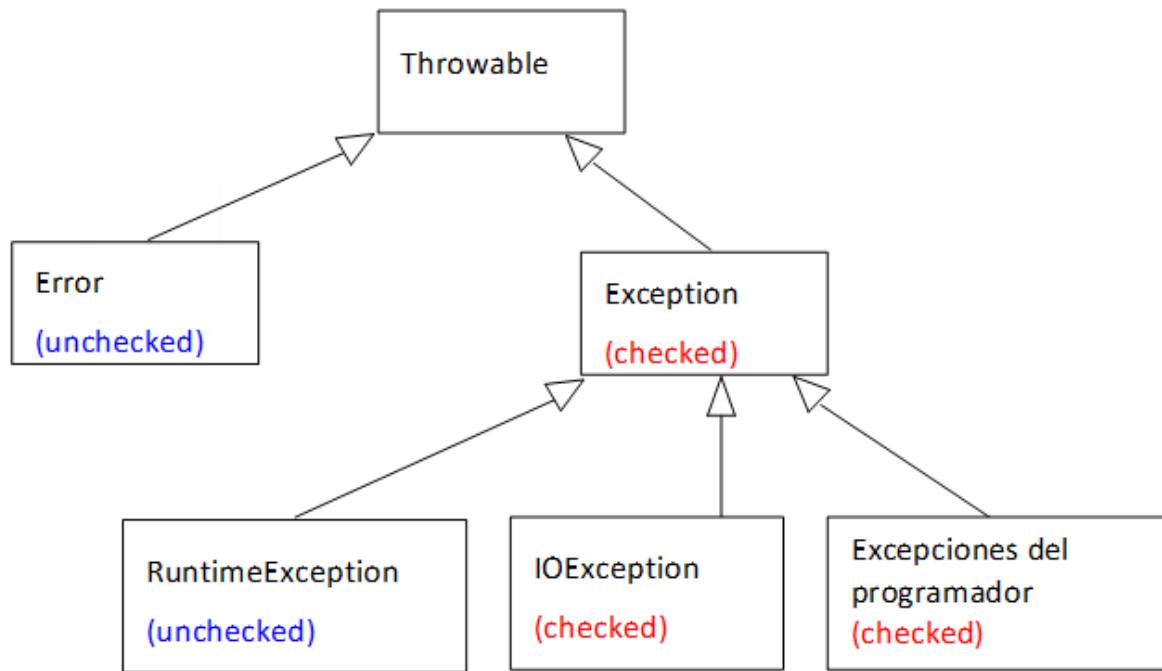
```

La variable que indica la fila va en el primer par de corchetes, y la variable que indica la columna va en el segundo, sin importar cuál ciclo va por fuera y cuál por dentro.

Manejo de Excepciones en Java

En Java, un error en tiempo de ejecución provocado por circunstancias anormales (errores matemáticos, índices fuera de rango, casting imposible de realizar, puntero nulo invocando a un método, etc.) se representa como un objeto. Así como la JVM crea automáticamente objetos que representan a los eventos producidos por el usuario sobre los componentes de la interfaz visual de usuario, también genera automáticamente objetos que representan a un error en tiempo de ejecución, permitiendo que el programador pueda (si lo desea) intervenir en esa situación y eventualmente recuperarse de ella, incluso sin que la aplicación finalice de forma abrupta. Esos objetos que representan errores de ejecución se llaman objetos de excepción o simplemente excepciones y forman parte de una jerarquía de clases cuya clase base es la clase `Throwable` (o sea: "que puede ser lanzada").

Si bien aún no hemos tratado aquí la forma de implementar Herencia en Java el principio de Herencia de la POO es conocido por los alumnos de materias anteriores y por lo tanto es podemos revisar esta jerarquía:



Los errores representados por la clase `Error` son errores graves de hardware o de sistema frente a los cuales no se espera que el programador pueda hacer nada más que darse por notificado del hecho. Son manejados automáticamente por la JVM. Los errores representados por la clase `Exception` son errores comunes de programación, algunos más graves que otros, que pueden requerir que el programador se vea obligado a escribir código de respuesta para esas excepciones, pues de lo contrario el programa no compilará. En ese sentido, las excepciones pueden clasificarse en chequeadas (`checked`) y en no chequeadas (`unchecked`). El gráfico anterior muestra cuáles son de cada tipo.

Si un segmento de código puede llegar a lanzar una excepción del tipo `checked`, entonces el compilador obliga a que el programador tenga eso en cuenta, escribiendo código para tratar esa posible excepción, aunque luego en la práctica la misma no llegue a lanzarse.

En general, todas las clases de excepción que derivan de la clase `Exception` son chequeadas (salvo las que derivan a su vez de la clase `RuntimeException`). Por ejemplo, las excepciones de IO que pueden producirse al trabajar con entrada de datos desde distintos dispositivos, derivan todas de la clase `IOException`, que a su vez baja desde `Exception`, y todas ellas son chequeadas: debemos escribir código para tratar las posibles excepciones de IO provocadas por nuestro código.

Si la excepción es no chequeada, el compilador no obliga a escribir código alguno de respuesta, y es decisión del programador el hacerlo o no. Todas las clases de excepción derivadas desde `RuntimeException` son no chequeadas, y también la clase `Error`. Si el programador no incluye ningún código de tratamiento para estas clases de excepciones y alguna llega a producirse, simplemente el programa o el método finalizará mostrando un mensaje de error acorde a la excepción producida, pero no habrá problemas de compilación previa. Algunas de las clases de excepción más comunes derivadas de `RuntimeException` y de `IOException` son las siguientes:

- Derivadas más comunes de `RuntimeException`:

- NullPointerException
 - ArithmeticException
 - IndexOutOfBoundsException
 - NegativeArraySizeException
- Derivadas más comunes de IOException:
 - EOFException
 - FileNotFoundException
 - InterruptedIOException
 - ObjectStreamException

Existen dos formas básicas de responder a una excepción (o sea: hay dos formas básicas de código preventivo contra esa excepción). Si la excepción es chequeada, el programador está obligado a decidirse por alguna de las dos que indicaremos. Si la excepción es no chequeada, el programador puede optar por no usar ninguna y simplemente ignorarla, o puede tratarla con cualquiera de las dos formas, tratar la excepción o informar su existencia posible.

Tratamiento de Excepciones

La forma más natural de tratar una excepción (sobre todo si es chequeada), consiste en capturarla con un bloque try – catch, en dicho bloque se comprueba si una excepción se ha producido, y actuar en consecuencia. Para ello se utilizan las palabras reservadas try, catch y finally.

El bloque try tiene que ir seguido, al menos, por una cláusula catch o una cláusula finally. La sintaxis general del bloque try consiste en la palabra clave try y una o más sentencias entre llaves.

```
try {
    // Sentencias Java
}
catch (UnTipoTrhowable nombreVariable) {
    // Sentencias Java
}
finally {
    // Sentencias Java
}
```

Puede haber más de una sentencia que genere excepciones, en cuyo caso habría que proporcionar un bloque try para cada una de ellas. Algunas sentencias, en especial aquellas que invocan a otros métodos, pueden lanzar, potencialmente, muchos tipos diferentes de excepciones, por lo que un bloque try consistente en una sola sentencia requeriría varios controladores de excepciones.

También se puede dar el caso contrario, en que todas las sentencias, o varias de ellas, que puedan lanzar excepciones, se encuentren en un único bloque try, con lo que habría que asociar múltiples controladores a ese bloque. Aquí la experiencia del programador es la que cuenta y es el propio programador el que debe decidir qué opción tomar en cada caso. Los controladores de excepciones deben colocarse inmediatamente después del bloque try. Si se produce una excepción dentro del bloque try, esa excepción será manejada por el controlador que esté asociado con el bloque try.

Cuando se produce la excepción, el código que se ejecuta es el que está en el bloque de catch. Es como si dijésemos "controla cualquier excepción que coincida con mi argumento". El argumento de la sentencia declara el tipo de excepción que el controlador, el bloque catch, va a manejar.

En este bloque tendremos que asegurarnos de colocar código que no genere excepciones. Se pueden colocar sentencias catch sucesivas, cada una controlando una excepción diferente. No debería intentarse capturar todas las excepciones con una sola cláusula, como esta:

```
catch( Expcion e ) { //...
```

Esto representaría un uso demasiado general, podrían llegar muchas más excepciones de las esperadas. En este caso es mejor dejar que la excepción se propague hacia arriba y dar un mensaje de error al usuario. La cláusula catch comprueba los argumentos en el mismo orden en que aparezcan en el programa. Si hay alguno que coincide, se ejecuta el bloque.

Cuando se colocan varios controladores de excepción, es decir, varias sentencias catch, el orden en que aparecen en el programa es importante, especialmente si alguno de los controladores engloba a otros en el árbol de jerarquía. Se deben colocar primero los controladores que manejen las excepciones más alejadas en el árbol de jerarquía, porque de otro modo, estas excepciones podrían no llegar a tratarse si son recogidas por un controlador más general colocado anteriormente.

Por lo tanto, los controladores de excepciones que se pueden escribir en Java son más o menos especializados, dependiendo del tipo de excepciones que traten. Es decir, se puede escribir un controlador que maneje cualquier clase que herede de Throwable; si se escribe para una clase que no tiene subclases, se estará implementando un controlador especializado, ya que solamente podrá manejar excepciones de ese tipo; pero, si se escribe un controlador para una clase nodo, que tiene más subclases, se estará implementando un controlador más general, ya que podrá manejar excepciones del tipo de la clase nodo y de sus subclases.

Por último el bloque finally, es el bloque de código que se ejecuta siempre, haya o no excepción. Por ejemplo, podría servir para hacer un log o un seguimiento de lo que está pasando, porque como se ejecuta siempre puede dejar grabado si se producen excepciones y si el programa se ha recuperado de ellas o no. Este bloque finally puede ser útil cuando no hay ninguna excepción. Es un trozo de código que se ejecuta independientemente de lo que se haga en el bloque try.

A la hora de tratar una excepción, se plantea el problema de qué acciones se van a tomar. En la mayoría de los casos, bastará con presentar una indicación de error al usuario y un mensaje avisándole de que se ha producido un error y que decida si quiere o no continuar con la ejecución del programa.

A continuación mostramos el uso del bloque try-catch con un ejemplo que permite leer un carácter desde teclado y comprueba que se ingrese el tipo correcto, utilizando el método read del paquete System.in para leer el carácter:

```
6  public char leer() {  
7      char r = ' ';  
8      try {  
9          System.out.print("Cargue un caracter: ");  
10         r = (char) System.in.read();  
11     } catch (IOException ex) {  
12         System.out.println("\nError: " + ex.getMessage());  
13     }  
14     return r;  
15 }  
16 }
```

Si el bloque encerrado entre las llaves de try llega a disparar una excepción de tipo IOException, automáticamente la JVM creará un objeto de esa clase y buscará el bloque catch que tenga un parámetro IOException. Si lo encuentra, pasará el objeto creado a ese bloque, y se ejecutará el código encerrado entre las llaves de catch. Luego de ello, el programa seguirá ejecutando las instrucciones que se encuentran debajo del bloque catch (a menos que dentro del catch se haya lanzado un System.exit()). Como se ve, no necesariamente el programa termina si una excepción se produce y la misma es capturada. La decisión de terminarlo es del programador.

El objeto generado por la JVM para representar la excepción dispone de una serie de métodos que permiten que el programador tenga mayor conocimiento del error producido. Uno de esos métodos es getMessage() (heredado desde Throwable) que retorna un String con una descripción del error que provocó la excepción. El mismo puede usarse para mostrar un mensaje más claro.

Como mencionamos anteriormente, si el bloque try puede llegar a lanzar excepciones de varias clases diferentes (aunque obviamente, sólo una en un momento dado), se pueden escribir varios bloques catch, cada uno con un parámetro que represente a la excepción esperada. Sólo debe tenerse en cuenta que la JVM recorre las definiciones catch por orden de escritura en el código, y al primero cuyo parámetro coincida con la excepción generada (incluidas las referencias polimórficas), lo aceptará. Por lo tanto, escriba los diversos catch comenzando por los menos polimórficos, y siga con los más polimórficos a continuación (de otro modo, no compilará...) El siguiente ejemplo muestra la forma correcta de hacerlo: El primer catch tiene un parámetro IOException, y el segundo uno de tipo Exception. Por lo tanto, este último debe escribirse al final:

```
18  public char leer() {  
19      char r = ' ';  
20      try {  
21          System.out.print("Cargue un caracter: ");  
22          r = (char) System.in.read();  
23      } catch (IOException ex) {  
24          System.out.println("\nError: " + ex.getMessage());  
25      }  
26      /* Si este bloque estuviera antes que el anterior,  
27      no dejaría pasar ninguna excepción derivada de ella...  
28      y no compilaría...*/  
29      catch (Exception ex) {  
30          System.out.println("\nError: " + ex.getMessage());  
31          System.exit(0);  
32      }  
33      return r;  
34 }
```

Note que cuando una excepción se captura y procesa con un bloque try – catch, entonces el lanzamiento de una excepción en el bloque try provocará que la ejecución de la secuencia dentro del try se interrumpa, y se traslade el flujo de ejecución al catch que corresponda. Por el contrario, si el try se ejecuta en forma normal, entonces ningún bloque catch se ejecutará. Esto puede provocar que en determinadas circunstancias el programa no ejecute ciertas instrucciones críticas (por ejemplo, el cierre de un archivo o de una base de datos, o la liberación de recursos gráficos): si esas instrucciones están en el bloque try y se dispara una excepción antes de ejecutarlas, o están en un catch pero no se produce una excepción, entonces las instrucciones críticas no serán ejecutadas nunca.

Se pueden producir casos donde se estén manipulando archivos, los cuales necesitan cerrarse para cerrar el flujo con el archivo físico, si el cierre del mismo se coloca dentro del bloque try y durante la ejecución del try se produce una excepción, se interrumpirá el try y el archivo podría llegar a no cerrarse. Esto puede ser un gran problema si se supone que el programa continúa en ejecución.

En casos así, se puede usar un bloque finally para complementar el try – catch. Un bloque finally se asemeja a un catch(), salvo por el hecho de que no se particulariza para un tipo específico de excepción, y por el hecho de que un bloque finally siempre se ejecuta: si el bloque try se ejecuta sin problemas, entonces al finalizar el mismo se ejecutará el bloque finally, y si el try se interrumpe se activará un catch(), pero al terminar de ejecutarse ese catch() se ejecutará de todos modos el finally. El siguiente esquema muestra una forma correcta de solucionar nuestro problema del cierre del archivo:

```
RandomAccessFile raf = null;
37   try {
38     raf = new RandomAccessFile("prueba.dat", "rw");
39     raf.writeInt(120);
40     raf.writeFloat(2000);
41     raf.writeUTF("Juan Perez");
42   } catch (IOException e) {
43     System.out.println("Error:: " + e.getMessage());
44   } finally {
45     if (raf != null) {
46       raf.close();
47     }
48   }
```

Debe notar que el bloque finally será ejecutado prácticamente en toda combinación de situaciones que afecten al bloque try – catch. La única situación en la que finally no se ejecutará es aquella en la cual se ejecuta un System.exit() en el try o en el catch() (en casos así, la invocación a exit() provoca que el programa se interrumpa de inmediato, sin llegar a ejecutar ninguna otra instrucción).

Try with resources (Try con recursos)

Para terminar esta sección, digamos brevemente que a partir de la versión 7 de Java se introdujo una nueva variante en cuanto a la posibilidad de escribir un bloque try, que se conoce como "try con recursos" (o "try with resources") Se trata de un bloque try en el cual se declaran en forma especial ciertos recursos. Esos recursos son objetos que deben ser cerrados al terminar el programa (como en el caso del ejemplo que hemos mostrado hasta aquí). El bloque try with resources garantiza que esos recursos serán efectivamente cerrados (en forma automática) al terminar de ejecutarse el bloque try. Técnicamente, los recursos declarados en un try with resources son objetos de cualquier clase que implemente la interface java.lang.AutoCloseable o su derivada java.lang.Closeable y el único método que estas declaran es justamente el método close().

A modo de ejemplo de uso, volvamos sobre el ejemplo que hemos mostrado para abrir, grabar y cerrar un RandomAccessFile. En el último esquema hemos visto que toda la operación podría escribirse con un try – catch – finally sin problemas. Pero alternativamente, a partir de Java 7, podemos hacer en forma más compacta la misma operación mediante un try with resources:

```

9   try (RandomAccessFile raf = new RandomAccessFile("prueba.dat", "rw")) {
10      raf.writeInt(120);
11      raf.writeFloat(2000);
12      raf.writeUTF("Juan Perez");
13  } catch (IOException e) {
14      System.out.println("Error:: " + e.getMessage());
15  }

```

En el ejemplo anterior mostramos el uso de un try with resources. El recurso asociado al bloque try se declara entre paréntesis inmediatamente luego de la palabra try. En nuestro caso se trata de un objeto de la clase RandomAccessFile (que implementa AutoCloseable, como todas las clases de gestión de archivos y flujos externos en Java). Tanto si el try se ejecuta sin problemas, como si el try provoca una excepción, el archivo representado por raf será cerrado automáticamente al terminar de ejecutarse el try o el catch (lo cual como primera ventaja, simplifica el bloque al no tener que escribir un finally explícito sólo para el cierre del archivo).

Note que un try with resources puede incluir bloques catch y finally en forma normal y cualquier bloque catch o finally será ejecutado después que el recurso asociado haya sido automáticamente cerrado.

Throws (avisando posible ocurrencia de excepciones)

La palabra clave throws se utiliza para identificar la lista posible de excepciones que un método puede lanzar. Si un método es capaz de provocar una excepción que no maneja él mismo, debería especificar este comportamiento, para que todos los métodos que lo llamen puedan colocar protecciones frente a esa excepción. Para la mayoría de las subclases de la clase Exception, el compilador Java obliga a declarar qué tipos podrá lanzar un método. Si el tipo de excepción es Error o RuntimeException, o cualquiera de sus subclases, no se aplica esta regla, dado que no se espera que se produzcan como resultado del funcionamiento normal del programa. Si un método lanza explícitamente una instancia de Exception o de sus subclases, a excepción de la excepción de runtime, se debe declarar su tipo con la sentencia throws. La declaración del método sigue ahora la sintaxis siguiente:

```
tipo nombreMetodo( argumentos ) throws excepciones { }
```

Lo más sencillo, es simplemente declarar la posibilidad de lanzamiento de la excepción en la cabecera del método que posea el bloque de código inseguro. A continuación mostramos el mismo ejemplo del punto anterior, pero haciendo que el método leer explice la excepción a manejar:

```

5   public char leer() throws IOException
6   {
7       System.out.print("Cargue un caracter: ");
8       char r = (char) System.in.read(); // puede lanzar una IOException
9       return r;
10  }

```

Si el código inseguro pudiera lanzar más de un tipo de excepción, se pueden declarar todas en la cabecera:

```
11  public char leer() throws IOException, Exception
12  {
13      System.out.print("Cargue un caracter: ");
14      char r = (char) System.in.read(); // puede lanzar una IOException
15      return r;
16 }
```

Lo anterior hace que el programador obligue a quien invoque al método leer() a tratar a su vez la excepción lanzada por él. En otras palabras, el programador de leer() se "saca el problema de encima" y lo transfiere a su "cliente".

En el ejemplo, para poder usar el método leer hay que escribir el siguiente código:

```
20  public static void main(String args[]) {
21      char res;
22      try {
23          res = leer();
24          System.out.println("\nEl caracter cargado es : " + res);
25      } catch (IOException ex) {
26          System.out.println("\nError: " + ex.getMessage());
27      }
28 }
```

Si no se encierra la invocación del método leer() en un bloque try catch, el compilador informará del error.

Provocar excepciones **throw**

La sentencia throw se utiliza para lanzar explícitamente una excepción. En primer lugar se debe obtener un descriptor de un objeto Throwable, bien mediante un parámetro en una cláusula catch o, se puede crear utilizando el operador new. La forma general de la sentencia throw es:

```
throw objetoThrowable;
```

El flujo de la ejecución se detiene inmediatamente después de la sentencia throw, y nunca se llega a la sentencia siguiente. Se inspecciona el bloque try que la engloba más cercano, para ver si tiene la cláusula catch cuyo tipo coincide con el del objeto o instancia Thorwable. Si se encuentra, el control se transfiere a esa sentencia. Si no, se inspecciona el siguiente bloque try que la engloba, y así sucesivamente, hasta que el gestor de excepciones más externo detiene el programa y saca por pantalla el trazado de lo que hay en la pila hasta que se alcanzó la sentencia throw.

En el programa siguiente, se muestra como se hace el lanzamiento de una nueva instancia de una excepción, en este caso una excepción de ArithmeticException, cuando encuentra que el posible divisor tiene el valor de 0:

```
13     int res = 0;
14     int valor;
15     try {
16         for(int x = 0; x < 100; x ++ )
17         {
18             valor = (int) Math.random();
19             if (valor == 0)
20                 throw new ArithmeticException();
21             res += res/valor;
22         }
23     }
24     catch( ArithmeticException e ) {
25         System.out.println("Error division por cero: " + e.getMessage());
26     }
27
28     System.out.println("El resultado obtenido es : " + res);
29 }
```

Por donde seguimos

Si bien en este apunte hemos revisado algunos conceptos del lenguaje de programación java como el uso de estructuras de datos nativas como los vectores y las matrices o las herramientas para manejar las excepciones o las situaciones no deseadas, lejos estamos aún de lograr componer con estas herramientas lógica de backend.

A priori los próximos pasos serían lograr procesar archivos de datos con estructuras puntuales manejando esos datos en objetos para lograr disponer de ellos en los procesos necesarios.

Por otro lado si bien los vectores son estructuras de datos potentes en general, en Java existe todo un Framework de colecciones que implementan herramientas más potentes y flexibles para construir nuestros programas con estructuras de datos y finalmente revisar los demás elementos de la programación orientada a objetos y su implementación en Java, ambos temas que abordaremos en la próxima semana.

Apunte 7 - Testing

Introducción General

Una vez desarrollada una parte de un programa, o a veces inclusive antes del desarrollo, llega el momento de verificar que todo funciona como se supone que debería funcionar. Para ello se realizan una serie de pruebas, donde el programador ejecuta una parte del programa, conociendo de antemano los resultados que deberían darse y verifica que efectivamente se estén entregando esos resultados. Estas pruebas de partes del programa se conocen como pruebas unitarias, o tests unitarios.

Si bien esto puede hacerse en forma manual para programas chicos, llega el punto en el que uno quiere automatizar dichos tests, para poder ejecutarlos múltiples veces al ir haciendo cambios al programa. Eso garantiza, o ayuda a evitar que de forma accidental el programador introduzca un cambio que termine rompiendo o modificando la funcionalidad existente de forma no esperada.

El ecosistema Java posee varias herramientas pensadas para realizar este tipo de tests, y la mas común es la utilización de una librería llamada *JUnit*. Dicha librería nos provee una serie de anotaciones y clases que uno puede usar para generar las pruebas unitarias. Esta librería fue una de las primeras dedicadas a la automatización de pruebas, y fue creada por dos proponentes del uso de testing automatizado como son *Kent Beck* y *Erich Gamma*.

Una vez creados los casos de prueba estos se pueden ejecutar desde el propio entorno de desarrollo, sea IntelliJ u otro, ya que casi todos tienen soporte para la ejecución de pruebas unitarias. Además se pueden correr desde el propio *Maven* como parte del proceso de empaquetado del programa.

Introducción a JUnit

JUnit es una librería que nos provee herramientas para la ejecución y creación de pruebas unitarias. Esta librería es una de las mas usadas para la generación de pruebas unitarias y actualmente se encuentra en la versión 5 (Al momento de escribir esto la versión 5.10)

Particularmente la versión 5 de JUnit se compone de tres componentes:

- *JUnit Platform*: Que es la plataforma que permite el descubrimiento y ejecución de las pruebas unitarias. Esta plataforma se encuentra integrada en casi todos los IDE de java y en las herramientas de construcción como *Maven* o *Gradle*
- *JUnit Jupiter*: Es un modelo de programación que nos permite escribir los tests y provee una serie de anotaciones y extensiones que ayudan a la escritura de los tests.
- *JUnit Vintage*: Es una capa de compatibilidad para poder seguir usando tests escritos para JUnit4 en proyectos que usan JUnit 5.

Para el caso de un proyecto que arranca con JUnit 5 no es necesario usar el JUnit Vintage, y la mayor parte del esfuerzo se va a centrar en la escritura de los tests usando el Junit Jupiter, que no es mas que un conjunto de anotaciones y clases a usar al momento de escribir una prueba unitaria.

Como importar JUnit en un proyecto Maven

Para poder utilizar JUnit en un proyecto maven, basta con agregar una dependencia al artefacto con groupId *org.junit.jupiter* y artifactId *junit-jupiter*. En general al referirse a una dependencia de maven se suele usar la forma groupId:artifactId, o sea para este caso la dependencia necesaria sería *org.junit.jupiter:junit-jupiter*

Dicha dependencia se agrega en la sección *<dependencies>* del archivo pom.xml de la siguiente forma:

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.10.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Agregando esta dependencia se agrega transitivamente la dependencia de junit-engine necesaria para la ejecución de los tests.

Nótese la inclusión del tag **scope** dentro de la definición de la dependencia. Esto le indica a Maven que esta dependencia será usada durante el proceso de testing, pero no será incluida dentro del artefacto generado en el packaging del proyecto.

Creación de un unit test

Lo primero que hay que ver al momento de crear un unit test, es donde se deben ubicar los archivos del test. Por convención de Maven, dentro de la carpeta src hay dos carpetas, **main** y **test**. La carpeta main es donde van a estar todos los archivos de código fuente que forman parte del código de producción, o sea el que se va a terminar empaquetando con el proyecto. La carpeta test está para que en ella se ubiquen todos los test unitarios, estos fuentes sólo se compilan durante la ejecución de los tests, pero no forman parte del producto.

Dentro de JUnit un unit test es básicamente un método anotado con la anotación **@Test**, y los tests se agrupan en Suites, que son clases cuyo nombre generalmente termina en *Test* y contienen uno o más métodos anotados con la anotación **@Test**.

```
public class EjemploTest {

    public int suma(int a, int b) {
        return a+b;
    }

    @Test
    public void testSuma() {
        int suma = suma(1, 3);
        Assertions.assertEquals(4, suma);
    }
}
```

En el ejemplo previo, el método testSuma va a invocar al método suma y comprobar el resultado devuelto por la función suma contra un valor esperado.

Además de @Test hay varias anotaciones mas que se pueden usar para marcar métodos, algunas de ellas son:

- **@BeforeEach**: Marca un método que se va a ejecutar antes de la ejecución de cada uno de los tests de la suite
- **@BeforeAll**: Marca un método que se va a ejecutar antes de la ejecución de todos los tests de la suite (sólo se ejecuta una vez por suite)
- **@AfterEach**: Análogo a BeforeEach, pero se ejecuta luego de cada test
- **@AfterAll**: Igualmente análogo a BeforeAll, sólo se va a ejecutar una vez al terminar de ejecutar todos los tests de la suite
- **@Disabled**: Permite desactivar un test, para que no se ejecute
- **@Timeout**: Permite establecer un tiempo máximo de ejecución para un test, si este tiempo se excede el test falla.

Comprobaciones

Dentro de un test, se espera que además de ejecutar algo y ver que no se produzcan excepciones, se compruebe que los resultados provistos por el código sean los correctos. Esto se hace mediante comprobaciones, conocidas como *assertions*. Si la comprobación es correcta, sigue la ejecución, pero si no es correcta se interrumpe la ejecución del test con un error.

Dentro de JUnit 5 las comprobaciones están dentro de la clase Assertions. Algunos de los métodos provistos por esta clase son:

- **assertTrue(boolean)***: Comprueba que el boolean sea true, o falla
- **assertTrue(boolean, mensaje)**: Comprueba que el boolean sea true o falla informando el mensaje indicado
- **assertFalse(boolean)**: Comprueba que el boolean sea false, o falla
- **assertFalse(boolean, mensaje)**: Comprueba que el boolean sea false o falla informando el mensaje indicado
- **assertEquals(esperado, valor)**: Comprueba que valor sea igual a esperado o falla. Este método tiene muchas variantes con diferentes tipos, notable de destacar es la variante de float que admite un tercer parámetro indicando un delta que tiene que superarse para que se considere que los valores no son iguales.
- **assertEquals(esperado, valor, mensaje)**: Igual que el anterior, pero agregando el mensaje a mostrar en caso de falla.
- **assertNotEquals(esperado, valor)**: Contrario a assertEquals
- **assertNotEquals(esperado, valor, mensaje)**: Contrario a assertEquals
- **fail()****: hace fallar el test
- **fail(mensaje)****: hace fallar el test informando un mensaje
- **assertThrows(clase, ejecutable)**: Comprueba que el código ejecutado en *ejecutable*, que es una interfaz funcional donde se puede usar un lambda, lance una Exception del tipo *clase*
- **assertDoesNotThrow(ejecutable)**: Comprueba que el código ejecutado en *ejecutable* NO lance una exception.

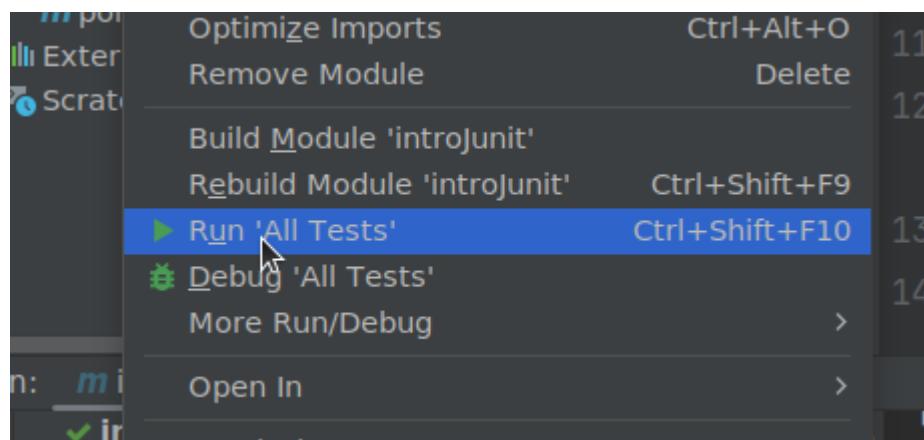
Todos los métodos definidos en Assertions son estáticos, con lo cual se pueden usar de la forma `Assertions.assertEquals(a, b)`, o también se suele hacer de forma habitual un import static para que todos los métodos assert estén disponibles para llamar directamente.

```
import static org.junit.jupiter.api.Assertions.assertTrue;
```

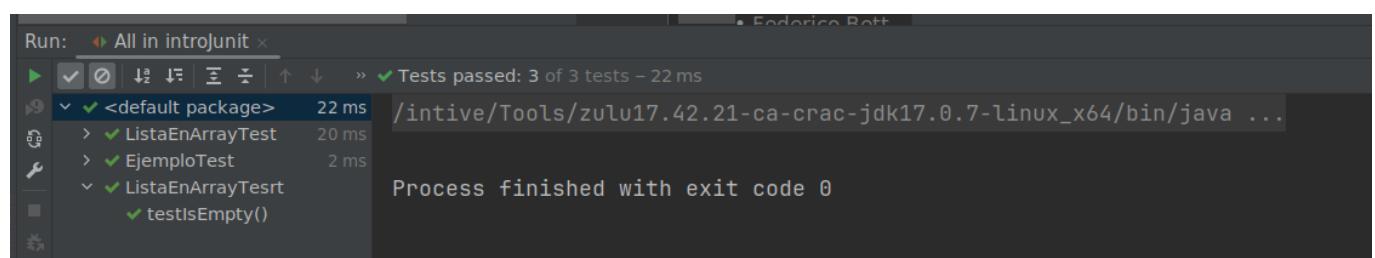
Haciendo este import se puede usar por ejemplo `assertEquals` como si estuviera definida dentro del test.

Ejecución de tests

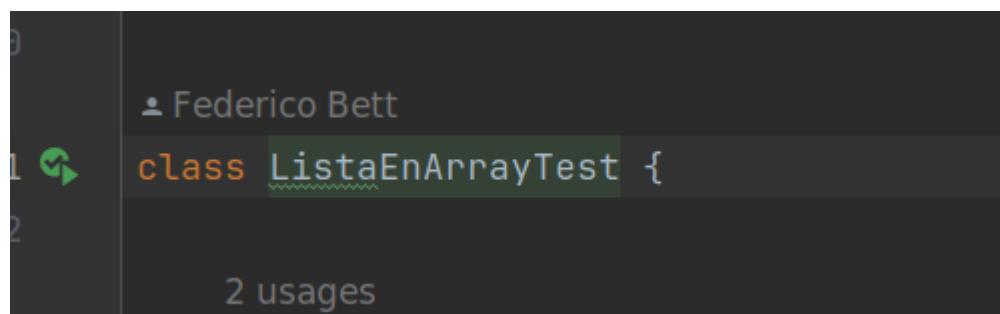
Para ejecutar los tests desde IntelliJ la forma mas simple es mediante el uso del botón derecho sobre el arbol de proyecto, y seleccionando la opción **Run 'All Tests'**



Esto va a ejecutar todas las suites de tests presentes en el proyecto y luego se va a mostrar el resultado de dicha ejecución, marcando los test que se ejecutaron exitosamente y los que no.



Alternativamente se puede ejecutar una sola suite de test, o un test individual haciendo click en la flecha verde que presenta IntelliJ al abrir el código de dicha suite de test.



Otra forma de ejecutar los tests, es mediante el uso de maven. Dentro de los diferentes pasos del ciclo de vida que provee maven, existe uno llamado **test** que se puede invocar para ejecutar los test

Este paso del ciclo de vida también se termina ejecutando al realizar otras acciones con maven como pueden ser **deploy**, **package** o **install**.

```
18:50 $ mvn test
[INFO] Scanning for projects...
[INFO]
[INFO] -----< ar.edu.utn.frc.bso:introJunit >-----
[INFO] Building introJunit 1.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- resources:3.3.0:resources (default-resources) @ introJunit ---
[INFO] Copying 0 resource
[INFO]
[INFO] --- compiler:3.10.1:compile (default-compile) @ introJunit ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- resources:3.3.0:testResources (default-testResources) @ introJunit ---
[INFO] skip non existing resourceDirectory
/home/fbett/UTN/2023/Backend/material/semana-
07/testing/introJunit/introJunit/src/test/resources
[INFO]
[INFO] --- compiler:3.10.1:testCompile (default-testCompile) @ introJunit ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- surefire:3.0.0:test (default-test) @ introJunit ---
[INFO] Using auto detected provider
org.apache.maven.surefire.junitplatform.JUnitPlatformProvider
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running ar.edu.utn.frc.bso.EjemploTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.043 s -
in ar.edu.utn.frc.bso.EjemploTest
[INFO] Running ar.edu.utn.frc.bso.ListaEnArrayList
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.006 s -
in ar.edu.utn.frc.bso.ListaEnArrayList
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.300 s
[INFO] Finished at: 2023-10-01T18:50:48-03:00
[INFO]
[INFO] -----
[WARNING]
[WARNING] Plugin validation issues were detected in 2 plugin(s)
[WARNING]
```

```
[WARNING] * org.apache.maven.plugins:maven-compiler-plugin:3.10.1
[WARNING] * org.apache.maven.plugins:maven-resources-plugin:3.3.0
[WARNING]
[WARNING] For more or less details, use 'maven.plugin.validation' property with
one of the values (case insensitive): [BRIEF, DEFAULT, VERBOSE]
[WARNING]
```

Mocking

Durante el desarrollo de pruebas unitarias, es normal que en algún punto tengamos que hacer que algún objeto devuelva un valor predefinido, esperado por el test.

Por ejemplo, al hacer tests a un servicio que usa un repositorio, va a ser necesario, para poder probar algunas cosas, hacer que ese repositorio devuelva cierto valor en concreto. Esto se logra implementando una versión del repositorio que devuelve valores fijos, o configurables, y usando dicha versión al instanciar el servicio en el test.

Por ejemplo dado este servicio de ejemplo:

```
package ar.edu.utn.frc.bso;

import java.util.List;
import java.util.Optional;

public class ServicioAlumnos {

    private RepositorioAlumnos repositorio;

    public ServicioAlumnos(RepositorioAlumnos repositorio) {
        this.repositorio = repositorio;
    }

    public Alumno obtenerAlumno(int legajo) {
        List<Alumno> lista = repositorio.listar();
        for(Alumno a: lista) {
            if (a.getLegajo() == legajo) {
                return a;
            }
        }
        return null;
    }
}
```

Uno podría intentar hacer dos tests para el método *obtenerAlumno*. Uno podría probar el caso en el que se encuentra el alumno, y uno el caso de que no se encuentre. Entonces, para ello se podría hacer una clase que herede de *RepositorioAlumnos* y agregar métodos para definir la lista de alumnos a devolver, para que cada test defina de antemano lo que debería devolver la llamada a *listar()* del repositorio.

Esto quedaría así:

```
package ar.edu.utn.frc.bso;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

class RepositorioTest extends RepositorioAlumnos {
    private List<Alumno> listaADevolver;

    public void setListaADevolver(List<Alumno> l) {
        this.listaADevolver = l;
    }

    @Override
    public List<Alumno> listar() {
        return listaADevolver;
    }
}

public class ServicioAlumnosTest {

    private ServicioAlumnos servicio;
    private RepositorioTest repositorioTest;

    @BeforeEach
    public void setup() {
        repositorioTest = new RepositorioTest();
        servicio = new ServicioAlumnos(repositorioTest);
    }

    @Test
    public void testAlumnoExistente() {
        Alumno alumnoEsperado = new Alumno("Pepe", 123);
        repositorioTest.setListaADevolver(List.of(alumnoEsperado));
        Alumno x = servicio.obtenerAlumno(123);
        Assertions.assertEquals(alumnoEsperado, x);
    }

    @Test
    public void testAlumnoNoExistente() {
        Alumno alumnoEsperado = new Alumno("Laura", 234);
        repositorioTest.setListaADevolver(List.of(alumnoEsperado));
        Alumno x = servicio.obtenerAlumno(123);
        Assertions.assertNull(x);
    }
}
```

La clase de test del repositorio hereda de la clase real (o implementa la interfaz si hubiera una), y permite establecer la lista de alumnos a devolver. Con esta funcionalidad cada test define la lista previamente a llamar al servicio, con lo cual se puede predecir que debería devolverse en cada caso.

Proyectos de Ejemplo

- [IntroJUnit](#) Proyecto con ejemplos de unit tests

📝 Ejemplo completo: Tests unitarios sobre la clase **Fracción**

A modo de cierre de este apunte, presentamos un ejemplo completo de test unitario utilizando la clase **Fracción**. Este ejemplo tiene por objetivo consolidar el uso de JUnit 5, incluyendo anotaciones como `@BeforeEach`, `@DisplayName`, `assertEquals`, `assertAll` y `assertThrows`, con un enfoque didáctico paso a paso.

▀ 1. Contexto: clase **Fracción**

La clase **Fracción** representa un número racional con numerador y denominador. Incluye:

- Constructor con validación para evitar denominadores cero
- Método `simplificar()` que reduce la fracción dividiendo ambos valores por su MCD
- Método `valorReal()` que retorna el valor decimal equivalente de la fracción

A continuación, desarrollaremos una clase de test para validar su comportamiento.

⌚ 2. Estructura del test con `@BeforeEach`

Antes de cada test se inicializan dos fracciones, una que se simplifica a 1/2 y otra a 1/3. Esto permite reutilizar las mismas instancias en múltiples pruebas.

```
import org.junit.jupiter.api.*; // Importa anotaciones y clases de JUnit 5
import static org.junit.jupiter.api.Assertions.*; // Importa métodos de aserción

@DisplayName("Test unitarios sobre la clase Fraccion")
class FraccionTest {

    Fraccion f1;
    Fraccion f2;

    @BeforeEach
    void init() {
        // Se ejecuta antes de cada método de test
        f1 = new Fraccion(2, 4); // Se espera que se simplifique a 1/2
        f2 = new Fraccion(3, 9); // Se espera que se simplifique a 1/3
    }
}
```

Al correr estos tests, la consola mostrará los nombres descriptivos definidos por `@DisplayName`, lo cual mejora la legibilidad de los reportes.

3. Validación de simplificación con `assertAll`

En esta prueba se agrupan múltiples validaciones con `assertAll`, lo cual permite validar todos los atributos relevantes de las fracciones sin detener la ejecución ante el primer fallo.

```
@Test
@DisplayName("Simplificación correcta de fracciones")
void testSimplificar() {
    // Invoca al método simplificar()
    f1.simplificar();
    // Valida numerador y denominador simplificados
    assertAll("Fracción f1 simplificada",
        () -> assertEquals(1, f1.getNumerador(), "Numerador esperado: 1"),
        () -> assertEquals(2, f1.getDenominador(), "Denominador esperado:
2")
    );
    // Invoca al método simplificar()
    f2.simplificar();
    // Valida numerador y denominador simplificados
    assertAll("Fracción f2 simplificada",
        () -> assertEquals(1, f2.getNumerador(), "Numerador esperado: 1"),
        () -> assertEquals(3, f2.getDenominador(), "Denominador esperado:
3")
    );
}
```

► *Ejecución esperada:* los valores de `f1` y `f2` deben estar simplificados correctamente. Si alguno no lo está, JUnit reportará qué parte falló.

4. Verificación de excepciones con `assertThrows`

Aquí se prueba que la clase arroje una excepción si se intenta crear una fracción con denominador igual a cero.

```
@Test
@DisplayName("Excepción si el denominador es cero")
void testDenominadorCero() {
    // Verifica que el constructor lanza la excepción esperada
    ArithmeticException ex = assertThrows(ArithmeticException.class,
        () -> new Fraccion(5, 0),
        "Se esperaba excepción por denominador cero"
    );

    // Valida el mensaje de la excepción para asegurar que sea informativo
    assertEquals("El denominador no puede ser cero", ex.getMessage());
}
```

► *Ejecución esperada:* se lanza una `ArithmeticException` y se verifica también su mensaje.

5. Cálculo del valor real con `assertEquals`

Esta prueba verifica que el método `valorReal()` retorne el valor decimal correcto para cada fracción. Se utiliza una tolerancia (`delta`) para comparar números de punto flotante.

```
@Test
@DisplayName("Cálculo correcto del valor real")
void testValorReal() {
    // Valida que 1/2 sea igual a 0.5 con una tolerancia de 0.0001
    assertEquals(0.5, f1.valorReal(), 0.0001);

    // Valida que 1/3 sea aproximadamente 0.3333
    assertEquals(0.3333, f2.valorReal(), 0.0001);
}
```

► *Ejecución esperada:* si el método `valorReal()` está bien implementado, los resultados coincidirán dentro de la tolerancia indicada.

6. Test parametrizado: cálculo del promedio de fracciones

Para demostrar el uso de `@ParameterizedTest` y validaciones más complejas, presentamos un caso donde se construye una lista de fracciones, se calcula la suma de sus valores reales y el promedio, y se contrasta contra los valores esperados.

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

...
@ParameterizedTest
@CsvSource({
    "1,2,0.5",
    "3,6,0.5",
    "10,20,0.5",
    "0,7,0.0",
    "7,4,1.75",
    "9,3,3.0",
    "8,2,4.0",
    "13,13,1.0"
})
@DisplayName("Cálculo del valor real para múltiples fracciones")
void testFraccionesParametrizadas(int numerador, int denominador, double esperado) {
    Fraccion f = new Fraccion(numerador, denominador);
    assertEquals(esperado, f.valorReal(), 0.0001,
        () -> "Esperado: " + esperado + ", pero fue: " + f.valorReal());
}
```

► *Ejecución esperada:* el test validará que diez fracciones 1/2 suman 5.0 y su promedio es 0.5.

Esta técnica permite testear múltiples valores dinámicamente y validar comportamientos agregados.

█ 7. Cálculo de suma y promedio con lista de fracciones

Este test no es parametrizado, pero exemplifica cómo utilizar una lista de objetos `Fraccion` para calcular operaciones agregadas como la suma total y el promedio. Es útil para mostrar cómo integrar streams y validaciones más avanzadas.

Cabe aclarar que más adelante vamos a ver colecciones y herramientas para hacer esto de una manera mucho más prolífica y organizada, sin embargo por ahora podemos implementar la idea con las herramientas que hemos documentado hasta aquí.

```
@Test
@DisplayName("Suma y promedio de una lista de fracciones")
void testSumaYPromedioFracciones() {
    Fraccion[] fracciones = new Fraccion[]{
        new Fraccion(1, 3), // ≈ 0.333...
        new Fraccion(2, 7), // ≈ 0.2857...
        new Fraccion(5, 6), // ≈ 0.8333...
        new Fraccion(7, 9), // ≈ 0.777...
        new Fraccion(4, 11), // ≈ 0.3636...
        new Fraccion(8, 13), // ≈ 0.6154...
        new Fraccion(3, 8), // = 0.375
        new Fraccion(9, 14), // ≈ 0.6428...
        new Fraccion(11, 16), // = 0.6875
        new Fraccion(6, 7) // ≈ 0.8571...
    };

    Fraccion suma = new Fraccion(0);
    for (Fraccion f : fracciones) {
        suma = suma.sumarA(f);
    }

    suma.simplificar();
    final Fraccion promedio = suma.dividirPor(new
Fraccion(fracciones.length));
    promedio.simplificar();
    final Fraccion sumaResult = suma;

    assertAll("Suma y promedio",
        () -> assertTrue(new Fraccion(831953, 144144).equals(sumaResult),
        "Suma esperada: [1/5] " + sumaResult),
        () -> assertTrue(new Fraccion(831953, 1441440).equals(promedio),
        "Promedio esperado: 0.5" + promedio)
    );
}
}
```

- *Ejecución esperada:* la suma de las fracciones resulta en la fracción [831953/144144] ya simplificada, y su promedio en la fracción [831953/1441440].

Esta prueba es útil como ejercicio integrador para aplicar conocimientos de POO, vectores y testing con JUnit.

Puntos de continuación

Una cosa que es interesante ver, teniendo en mente la idea de testing unitario, es una metodología de desarrollo que se basa en realizar los test antes que el código de producción. A esta metodología se la conoce como TDD (Test Driven Development)

Bibliografía

- [Junit User Guide](#)