

Ministerio de Producción y Trabajo

En colaboración con el

Ministerio de Educación Cultura, Ciencia y Tecnología



Programadores

Apunte del Módulo

Desarrollo de Aplicaciones Web con Java

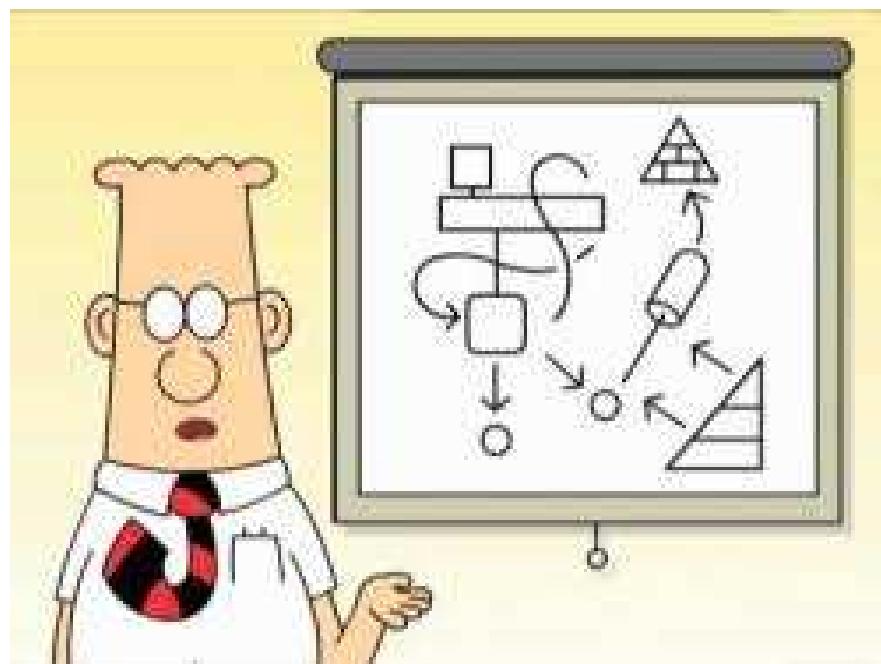


Tabla de Contenido

DEFINICIÓN DEL MÓDULO	5
PRESENTACIÓN	5
PROGRAMACIÓN ORIENTADA A OBJETOS.....	6
SURGIMIENTO DEL LENGUAJE	6
ENTENDIENDO JAVA.....	6
EL ENTORNO DE DESARROLLO DE JAVA.....	7
EL COMPILADOR DE JAVA.....	7
LA JAVA VIRTUAL MACHINE	7
APLICACIONES DE JAVA	9
ANTES DE COMENZAR.....	9
DECLARACIÓN DE CLASES	10
TIPOS DE DATOS	11
TIPOS BÁSICOS DE DATOS.....	11
TIPOS DE DATOS REFERENCIA.....	12
CONVERSIÓN DE TIPOS DE DATOS O CASTING	13
EL RECOLECTOR DE BASURA	13
ÁMBITO DE LAS VARIABLES	14
OPERADORES	14
ARRAYS	18
ESTRUCTURAS DE CONTROL.....	19
ESTRUCTURAS CONDICIONALES.....	19
BUCLAS.....	21
BLOQUES PARA MANEJO DE EXCEPCIONES	22
CLASES Y OBJETOS	23
VARIABLES MIEMBROS DE OBJETO	24
MÉTODOS.....	25
HERENCIA DE CLASES	33
SOBRESCRITURA DE VARIABLES Y MÉTODOS	35
CLASES Y MÉTODOS ABSTRACTOS	36
INTERFACES.....	36
DOCUMENTACIÓN DE CLASES Y MÉTODOS	38
TRABAJANDO CON FECHAS	41
LOCALDATE, LOCALTIME Y LOCALDATETIME	42
OPERACIONES CON FECHAS	44
FORMATOS	44
OPERACIONES MATEMÁTICAS CON BIGDECIMAL	46
OPERACIONES.....	47
PRECISIÓN Y ESCALA	48
REDONDEO	48
COLECCIONES DE TAMAÑO VARIABLE.....	50
LISTAS	50
LA CLASE JAVA.UTILSTACK.....	53
EXCEPCIONES	54

LA API DE JAVA	60
JAVA WEB.....	63
INTRODUCCIÓN A JAVA WEB.....	63
SPRING BOOT.....	63
CREACIÓN DE UN PROYECTO MAVEN.....	64
INICIALIZACIÓN DE UN PROYECTO SPRING BOOT.....	72
CREANDO UN NUEVO PROYECTO SPRING BOOT CON SPRING INITIALIZR	72
CREACIÓN DE CLASES DE CONTROL CON ANOTACIONES	78
PLANTILLAS HTML5 CON EL MOTOR THYMELEAF	79
MAQUETADO DE VISTAS CON BOOTSTRAP	80
JAVASCRIPT Y JQUERY	82
JQUERY	83
INTRODUCCIÓN A TDD	84
EL CICLO DE DESARROLLO DE TDD	85
TESTING UNITARIO CON JUNIT	87
¿QUÉ ES UNA PRUEBA UNITARIA?	87
REFERENCIA DE FIGURAS Y TABLAS.....	106
FIGURAS	106
TABLAS	107
FUENTES DE INFORMACIÓN	108

Definición del Módulo

Denominación de Módulo: **Desarrollo de Aplicaciones Web con Java**

Presentación

El módulo de Desarrollo de Aplicaciones Web con Java tiene como propósito general, contribuir a que los estudiantes desarrollen capacidades técnicas de programación para el desarrollo de aplicaciones en ambiente Web, utilizando JAVA como lenguaje de programación orientado a objetos. Profundiza y amplía las capacidades adquiridas en los módulos de Pensamiento Lógico y de Programación Orientada a Objetos dado que se aplican conceptos y herramientas adquiridas en estos módulos.

Este módulo se constituye así en un espacio de formación que permite a los estudiantes desarrollar saberes propios de la formación específica de la figura profesional de “Programador”.

Para la organización de la enseñanza de esta unidad curricular se han organizado los contenidos en los siguientes ejes temáticos

- Introducción a la Programación con Java
- Introducción al Sprint Boot
- Maquetado con HTML y CSS
- JavaScript y Jquery
- TDD (Test Driven Development)
- Testing Unitario con JUnit

El módulo “Desarrollo de Aplicaciones Web con Java” recupera e integra conocimientos, saberes y habilidades cuyo propósito general es contribuir al desarrollo de los estudiantes de una formación especializada, integrando contenidos, desarrollando prácticas formativas y su vínculo con los problemas característicos de intervención y resolución técnica del Programador.

Este módulo se orienta al desarrollo de las siguientes capacidades profesionales referidas al perfil profesional en su conjunto:

- Construir código de programación de acuerdo con especificaciones.
- Interpretar especificaciones de diseño que le permitan construir el código en el contexto del desarrollo de software en el que participa.
- Dimensionar su trabajo en el contexto del proyecto de desarrollo de software.
- Realizar pruebas unitarias y de sistemas. Verificar el código desarrollado, utilizando revisiones técnicas.
- Analizar Errores de código
- Elaborar documentación técnica de acuerdo con los requerimientos funcionales y técnicos recibidos. Integrar un equipo en el contexto de un Proyecto de Desarrollo de Software.
- Interpretar las especificaciones formales o informales del Líder de proyecto
- Analizar el problema a resolver
- Interpretar el material recibido y clarificar eventuales interpretaciones
- Determinar el alcance del problema y convalidar su interpretación a fin de identificar aspectos faltantes

Programación Orientada a Objetos

A partir de este punto, habiendo analizado las características del paradigma orientado a objetos y el lenguaje estándar que se utiliza para modelar (UML), estudiaremos el lenguaje de programación Java, que está basado en el paradigma orientado a objetos que estudiamos anteriormente.

Surgimiento del Lenguaje

Java surgió en 1991 cuando un grupo de ingenieros de la empresa Sun Microsystems trataron de diseñar un nuevo lenguaje de programación destinado a electrodomésticos. La reducida potencia de cálculo y memoria de los electrodomésticos llevó a desarrollar un lenguaje sencillo capaz de generar código de tamaño muy reducido.

Debido a la existencia de distintos tipos de CPUs y a los continuos cambios, era importante conseguir una herramienta independiente del tipo de CPU utilizada. Es por esto que desarrollan un código “neutro” que no depende del tipo de electrodoméstico, el cual se ejecuta sobre una “máquina hipotética o virtual” denominada Java Virtual Machine (JVM). Es la JVM quien interpreta el código neutro convirtiéndolo a código particular de la CPU utilizada. Esto permitía lo que luego se ha convertido en el principal lema del lenguaje: “Write Once, Run Everywhere”. A pesar de los esfuerzos realizados por sus creadores, ninguna empresa de electrodomésticos se interesó por el nuevo lenguaje. Java, como lenguaje de programación para computadoras, se introdujo a finales de 1995. Si bien su uso se destaca en la Web, sirve para crear todo tipo de aplicaciones (locales, intranet o internet).

Entendiendo Java

Java es un lenguaje compilado e interpretado; esto significa que el programador escribirá líneas de código utilizando un determinado programa que se denomina IDE (Ambiente de Desarrollo Integrado; Integrated Development Environment por sus siglas en inglés). Este programa le provee al programador un espacio de trabajo para crear código que se almacena en archivos de formato .java.

Luego, para poder correr el código y ejecutar el programa creado, se debe realizar primero lo que se denomina Compilación; para eso un programa denominado compilador será el encargado de revisar la sintaxis del código y si está correcto transformará el archivo .java en otro archivo de formato .class que se denominan: “bytecodes”. Una vez realizada la compilación, será la Máquina Virtual (Java Virtual Machine - JVM), la encargada de leer los archivos .class y transformarlos en el código entendido por la CPU de la computadora.

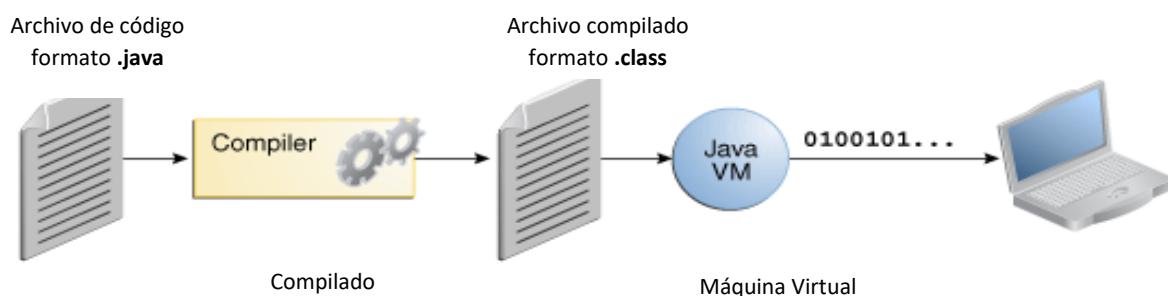


Fig. 1- Funcionamiento general del Lenguaje Java

El entorno de desarrollo de Java

Existen distintos programas comerciales que permiten desarrollar código Java. Oracle, quien compró a Sun (la creadora de Java), distribuye gratuitamente el Java Development Kit (JDK). Se trata de un conjunto de programas y librerías que permiten desarrollar, compilar y ejecutar programas en Java. Incorpora además la posibilidad de ejecutar parcialmente el programa, deteniendo la ejecución en el punto deseado y estudiando en cada momento el valor de cada una de las variables (es el denominado Debugger). Es tarea muy común de un programador realizar validaciones y pruebas del código que construye, el debugger permite realizar una prueba de escritorio automatizada del código, lo cual ayuda a la detección y corrección de errores. En el momento de escribir este trabajo las herramientas de desarrollo: JDK, van por la versión 1.8. Estas herramientas se pueden descargar gratuitamente de <http://www.oracle.com/technetwork/java>.

Los IDEs (Integrated Development Environment), tal y como su nombre indica, son entornos de desarrollo integrados. En un mismo programa es posible escribir el código Java, compilarlo y ejecutarlo sin tener que cambiar de aplicación. Algunos incluyen una herramienta para realizar Debug gráficamente, frente a la versión que incorpora el JDK basada en la utilización de una Consola bastante difícil y pesada de utilizar. Estos entornos integrados permiten desarrollar las aplicaciones de forma mucho más rápida, incorporando en muchos casos librerías con componentes ya desarrollados, los cuales se incorporan al proyecto o programa. Estas herramientas brindan una interfaz gráfica para facilitar y agilizar el proceso de escritura de los programas. El entorno elegido para este módulo, por su relevancia en el mercado y su licencia gratuita se denomina Netbeans. El mismo se encuentra disponible para descargar de forma gratuita en www.netbeans.org. En la Guía Práctica abordaremos la forma de instalación y configuración de estas herramientas.

El compilador de Java

Se trata de una de las herramientas de desarrollo incluidas en el JDK. Realiza un análisis de sintaxis del código escrito en los ficheros fuente de Java (con extensión *.java). Si no encuentra errores en el código genera los ficheros compilados (con extensión *.class). En otro caso muestra la línea o líneas erróneas. En el JDK de Sun dicho compilador se llama javac.exe si es para el sistema operativo Windows.

La Java Virtual Machine

Tal y como se ha comentado al comienzo, la existencia de distintos tipos de procesadores y ordenadores llevó a los ingenieros de Sun a la conclusión de que era muy importante conseguir un software que no dependiera del tipo de procesador utilizado. Se plantea la necesidad de conseguir un código capaz de ejecutarse en cualquier tipo de máquina. Una vez compilado no debería ser necesaria ninguna modificación por el hecho de cambiar de procesador o de ejecutarlo en otra máquina. La clave consistió en desarrollar un código “neutro” el cual estuviera preparado para ser ejecutado sobre una “máquina hipotética o virtual”, denominada Java Virtual Machine (JVM). Es esta JVM quien interpreta este código neutro convirtiéndolo a código particular de la CPU o chip utilizada. Se evita tener que realizar un programa diferente para cada CPU o plataforma.

La JVM es el intérprete de Java. Ejecuta los “bytecodes” (ficheros compilados con extensión *.class) creados por el compilador de Java (javac.exe). Tiene numerosas opciones entre las que destaca la posibilidad de utilizar el denominado JIT (Just-In-Time Compiler), que puede mejorar entre 10 y 20 veces la velocidad de ejecución de un programa.

Características de Java

Java es un lenguaje:

- **Orientado a objetos:** Al contrario de otros lenguajes como C++, Java no es un lenguaje modificado para poder trabajar con objetos, sino que es un lenguaje creado originalmente para trabajar con objetos. De hecho, todo lo que hay en Java son objetos.
- **Independiente de la plataforma:** Debido a que existen máquinas virtuales para diversas plataformas de hardware, el mismo código Java puede funcionar prácticamente en cualquier dispositivo para el que exista una JVM.
- **Compilado e Interpretado:** La principal característica de Java es la de ser un lenguaje compilado e interpretado. Todo programa en Java ha de compilarse y el código que se genera **bytecode** es interpretado por una máquina virtual, como se vio anteriormente. De este modo se consigue la independencia de la máquina: el código compilado se ejecuta en máquinas virtuales que sí son dependientes de la plataforma. Para cada sistema operativo distintos, ya sea Microsoft Windows, Linux, OS X, existe una máquina virtual específica que permite que el mismo programa Java pueda funcionar sin necesidad de ser recompilado.
- **Robusto:** Su diseño contempla el manejo de errores a través del mecanismo de Excepciones y fuerza al desarrollador a considerar casos de mal funcionamiento para reaccionar ante las fallas.
- **Gestiona la memoria automáticamente:** La máquina virtual de Java gestiona la memoria dinámicamente como veremos más adelante. Existe un **recolector de basura** que se encarga de liberar la memoria ocupada por los objetos que ya no están siendo utilizados.
- **No permite el uso de técnicas de programación inadecuadas:** Como veremos más adelante, todo en Java se trata de objetos y clases, por lo que para crear un programa es necesario aplicar correctamente el paradigma de objetos.
- **Multihilos (multithreading):** Soporta la creación de partes de código que podrán ser ejecutadas de forma paralela y comunicarse entre sí.
- **Cliente-servidor:** Java permite la creación de aplicaciones que pueden funcionar tanto como clientes como servidores. Además, provee bibliotecas que permiten la comunicación, el consumo y el envío de datos entre los clientes y servidores.
- **Con mecanismos de seguridad incorporados:** Java posee mecanismos para garantizar la seguridad durante la ejecución comprobando, antes de ejecutar código, que este no viola ninguna restricción de seguridad del sistema donde se va a ejecutar. Además, posee un *gestor de seguridad* con el que puede restringir el acceso a los recursos del sistema.
- **Con herramientas de documentación incorporadas:** Como veremos más adelante, Java contempla la creación automática de documentación asociada al código mediante la herramienta **Javadoc**.



Aplicaciones de java

Java es la base para prácticamente todos los tipos de aplicaciones de red, además del estándar global para desarrollar y distribuir aplicaciones móviles y embebidas, juegos, contenido basado en web y software de empresa. Java¹ se encuentra aplicado en un amplio rango de dispositivos desde portátiles hasta centros de datos, desde consolas para juegos hasta súper computadoras, desde teléfonos móviles hasta Internet.

- El 97% de los escritorios empresariales ejecutan Java.
- Es la primera plataforma de desarrollo.
- 3 mil millones de teléfonos móviles ejecutan Java.
- El 100% de los reproductores de Blu-ray incluyen Java.
- 5 mil millones de Java Cards en uso.
- 125 millones de dispositivos de televisión ejecutan Java.

Antes de comenzar

Como se detalló en el apartado “Entendiendo Java”, los archivos de código escritos por el programador que se denominan: *archivos de código fuente*, en Java son los archivos .java. Éstos se podrían crear utilizando simplemente un editor de texto y guardándolo con la extensión .java. Por lo tanto, esto se puede hacer utilizando cualquier editor de texto como el bloc de notas de Windows. Para facilitar la tarea de desarrollo en la práctica, como vimos, se utilizan IDEs (entornos de desarrollo integrados) que ofrecen herramientas como coloreado de palabras clave, análisis de sintaxis en tiempo real, compilador integrado y muchas otras funciones que usaremos.

Al ser un lenguaje multiplataforma y especialmente pensado para su integración en redes, la codificación de texto utiliza el estándar Unicode, lo que implica que los programadores y programadoras hispanohablantes podemos utilizar sin problemas símbolos de nuestra lengua como la letra “ñ” o las vocales con tildes o diéresis a la hora de poner nombre a nuestras variables.

Algunos detalles importantes son:

- En java (como en C) hay diferencia entre mayúsculas y minúsculas por lo que la variable nombrecompleto es diferente a nombreCompleto.
- Cada línea de código debe terminar con un; (punto y coma)
- Una instrucción puede abarcar más de una línea. Además, se pueden dejar espacios y tabuladores a la izquierda e incluso en el interior de la instrucción para separar elementos de la misma.
- A veces se marcan bloques de código, es decir código agrupado. Cada bloque comienza con llave que abre, "{" y termina con llave que cierra, "}"

¹ Datos obtenidos en <https://www.java.com/es/about/>

Declaración de Clases

En el caso más general, la declaración de una clase puede contener los siguientes elementos:

```
[public] [final | abstract] class NombreDeLaClase [extends ClaseMadre] [implements Interfase1 [, Interfase2 ]...]
```

Donde las porciones encerradas entre corchetes son opcionales a optar entre las posibilidades separadas por la barra vertical.

O bien, para interfaces:

```
[public] interface NombreDeLaInterface [extends InterfaceMadre1 [, InterfaceMadre2 ]...]
```

Como se ve, lo único obligatorio es la palabra reservada `class` y el nombre que queramos asignar a la clase. Las interfaces son un caso de tipo particular que veremos más adelante.

De esta forma, podríamos declarar la clase `Perro` como sigue:

```
public class Perro
{
    // esto es un comentario
    // aquí completaremos luego el cuerpo de la clase
}
```

De esta forma una instancia (objeto) de la clase `Perro` puede declararse utilizando el operador `new` de la siguiente forma, ocupando un espacio en memoria que luego podrá ser accedido mediante el nombre `miPerro`.

```
Perro miPerro = new Perro();
```

En el ejemplo anterior podemos identificar, además de la declaración de la clase `Perro` como de tipo `public` (veremos luego su significado), el uso de llaves (`{ y }`) para encerrar los componentes de la clase y las barras oblicuas para definir comentarios de una sola línea. En caso de querer escribir comentarios más extensos podemos encerrar el bloque entre `/* y */` de la siguiente forma:

```
/* Este es un comentario más extenso que ocupa más de una línea:
Al utilizar los símbolos barra oblicua y asterisco, indicamos que lo que sigue a continuación
es comentario del código. Los comentarios de código se utilizar frecuentemente en el ámbito de
la programación ya que es necesario aclarar distintos aspectos de la codificación, como, por
ejemplo: cuáles son los objetivos de los métodos, qué parámetros espera un método, explicar
una sentencia en particular, etc. Los comentarios son muy útiles para explicar utilizando el
lenguaje natural qué es lo que se está programando y de esta manera, poder en un futuro leer
rápidamente los comentarios y entender y recordar cuál era el objetivo del código. Además, en
esta disciplina realizar comentarios durante el proceso de programación es una muy buena
práctica que se utiliza frecuentemente para que en el caso de que otro programador tenga que
continuar con el código pueda entender cuál fue el pensamiento del que creó originariamente el
mismo.*/
```

Los comentarios enmarcados entre /* y */ no se pueden anidar. Los comentarios tanto sean de una línea o de bloques serán ignorados por el compilador a la hora de generar el **bytecode** a ser ejecutado en la máquina virtual, sólo sirven a los fines de lectura y comprensión del código por parte del equipo desarrollador. Más adelante veremos una forma especial de escritura de comentarios denominada **JavaDoc** que nos permitirá generar la documentación asociada al programa con las definiciones de las clases, métodos y demás de forma automatizada. Sintácticamente los comentarios JavaDoc comienzan con barra y dos asteriscos y terminan con asterisco barra:

```
/**  
...  
*/
```

Tipos de Datos

Los tipos de variables disponibles son básicamente 3:

- Tipos básicos (no son objetos)
- Arreglos (arrays o vectores)
- Clases e Interfaces

Con lo que vemos que cada vez que creamos una clase o interface estamos definiendo un nuevo tipo. Siempre es aconsejable asignar un valor por defecto en el momento de declaración de una variable. En algunos casos, incluso, se producirá un error durante la compilación si hemos olvidado inicializar el valor de alguna variable y tratamos de manipular su contenido.

Tipos básicos de datos

Nombre	Declaración	Rango	Descripción
Booleano	boolean	true - false	Define una bandera que puede tomar dos posibles valores: true o false.
Byte	byte	[-128 .. 127]	Representación del número de menor rango con signo.
Entero pequeño	short	[-32,768 .. 32,767]	Representación de un entero cuyo rango es pequeño.
Entero	int	[-2 ³¹ .. 2 ³¹ -1]	Representación de un entero estándar. Este tipo puede representarse sin signo usando su clase Integer a partir de la Java SE 8.
Entero largo	long	[-2 ⁶³ .. 2 ⁶³ -1]	Representación de un entero de rango ampliado. Este tipo puede representarse sin signo usando su clase Long a partir de la Java SE 8.
Real	float	[±3,4·10 ⁻³⁸ .. ±3,4·10 ³⁸]	Representación de un real estándar. Recordar que al ser real, la precisión del dato contenido varía en función del tamaño del número: la precisión se amplía con números más próximos a 0 y disminuye cuanto más se aleja del mismo.

Real largo	double	$[\pm 1,7 \cdot 10^{-308} .. \pm 1,7 \cdot 10^{308}]$	Representación de un real de mayor precisión. Double tiene el mismo efecto con la precisión que float.
Carácter	char	$['\u0000' .. '\uffff']$ o $[0 .. 65.535]$	Carácter o símbolo. Para componer una cadena es preciso usar la clase String, no se puede hacer como tipo primitivo.

Tabla 1 – Tipos de básicos de datos en JAVA

Tipos de datos referencia

En Java los objetos, instancias de clases, se manejan a través de referencias. Cuando se crea una nueva instancia de una clase con el operador *new*, este devuelve una referencia al tipo de la clase. Para aclararlo veamos un ejemplo:

Si tenemos la clase Punto definida de la siguiente manera:

```
public class Punto {
    private float float x;
    private float y;
    ...
    public void setX(float x) {
        this.x = x;
    }
    ...
}
Punto unPunto = new Punto();
```

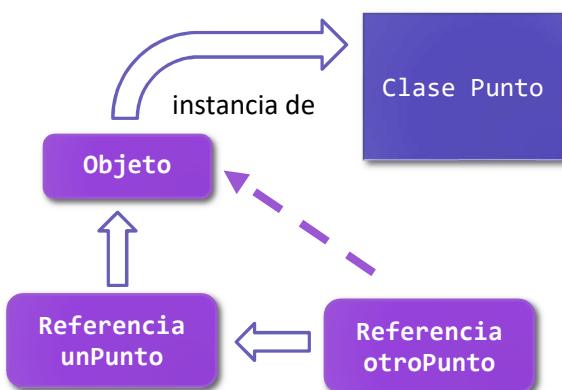
El operador *new* reserva espacio en memoria para contener un objeto del tipo Punto y devuelve una referencia que se asigna a *unPunto*, con los valores de atributos que se definieron en el constructor. A partir de aquí, accedemos al objeto a través de su referencia. Es posible, por tanto, tener varias referencias al mismo objeto. Presta atención al siguiente fragmento de código.

```
Punto unPunto = new Punto();
unPunto.print();

Punto otroPunto = unPunto;
otroPunto.setX(1.0f);
otroPunto.setY(2.0f);
otroPunto.print();
```

La salida por pantalla es:

Coordenadas del punto (0.0f,0.0f)
Coordenadas del punto (1.0f,2.0f)



Como las dos referencias: *unPunto* y *otroPunto*, hacen referencia a la misma instancia, los cambios sobre el objeto se pueden realizar a través de cualquiera de ellas.

Conversión de tipos de datos o Casting

En muchas ocasiones hay que transformar una variable de un tipo a otro, por ejemplo, de *int* a *double*, o de *float* a *long*. En otras ocasiones la conversión debe hacerse entre objetos de clases diferentes, aunque relacionadas mediante la herencia de una clase común o la implementación de una interfaz compatible. Veremos más sobre este último caso cuando hablaremos de Polimorfismo.

La conversión entre tipos primitivos es más sencilla. En Java, se realizan de modo automático conversiones implícitas *de un tipo a otro de más precisión*, por ejemplo, de *int* a *long*, de *float* a *double*, etc. Estas conversiones se hacen al mezclar variables de distintos tipos en expresiones matemáticas o al ejecutar sentencias de asignación en las que el miembro izquierdo tiene un tipo distinto que el resultado de evaluar el miembro derecho.

Por ejemplo:

```
int a = 5;
float b = 1.5;
float c = a * b; // producto de distintos tipos de datos primitivos convertidos
automáticamente
```

Las conversiones de un tipo de mayor a otro de menor precisión requieren una orden explícita del programador, pues son *conversiones inseguras* que pueden dar lugar a errores (por ejemplo, para pasar a *short* un número almacenado como *int*, hay que estar seguro de que puede ser representado con el número de cifras binarias de *short*). A estas conversiones **explícitas** de tipo se les llama *cast*. El *cast* se hace poniendo el tipo al que se desea transformar entre paréntesis, como, por ejemplo,

```
long result;
result = (long) (a/(b+c));
```

El Recolector de basura

Los objetos que dejan de estar referenciados a través de alguna variable no se pueden volver a recuperar. Para que estos objetos “desreferenciados” no ocupen memoria, un recolector de basura se encarga de «destruirlos» y liberar la memoria que estaban ocupando. Por lo tanto, para «destruir» un objeto basta con asignar a su variable de referencia el valor *null* como se puede ver en el siguiente ejemplo.

```
Punto unPunto = new Punto(1.0f, 2.0f);
Punto otroPunto = new Punto(1.0f, -1.0f);
unPunto = new Punto(2.0, 2.0f); // El punto (1.0f, 2.0f) se pierde
otroPunto = null; // El punto (1.0f, -1.0f) se pierde
```

Ámbito de las variables

Toda variable tiene un ámbito. Esto es la parte del código en la que una variable se puede utilizar. De hecho, las variables tienen un ciclo de vida:

1. En la declaración se reserva el espacio necesario para que se puedan comenzar a utilizar (digamos que se avisa de su futura existencia).
2. Se la asigna su primer valor (la variable nace).
3. Se la utiliza en diversas sentencias.
4. Cuando finaliza el bloque en el que fue declarada, la variable muere. Es decir, se libera el espacio que ocupa esa variable en memoria. No se la podrá volver a utilizar.

Una vez que la variable ha sido eliminada, no se puede utilizar. Dicho de otro modo, no se puede utilizar una variable más allá del bloque en el que ha sido definida.

El ámbito de las variables está determinado por el bloque de código donde se declaran y todos los bloques que estén anidados por debajo de este. Presta atención al siguiente fragmento de código:

```
{
    // Aquí tengo el bloque externo
    int entero = 1;
    Punto unPunto = new Punto();
    {
        // Y aquí tengo el bloque interno
        int entero = 2; // Error ya está declarada
        unPunto = new Punto(1.0f, 1.0f); // Correcto
    }
}
```

Operadores

Las variables se manipulan muchas veces utilizando operaciones con ellos. Los datos se suman, se restan, multiplican, etc. y a veces se realizan operaciones más complejas.

Operadores aritméticos

En Java disponemos de los operadores aritméticos habituales en lenguajes de programación como son suma, resta, multiplicación, división y operador que devuelve el resto de una división entre enteros, también llamado módulo:

OPERADOR	DESCRIPCIÓN
+	Suma
-	Resta
*	Multiplicación

/	División
%	Resto de una división entre enteros (en otros lenguajes denominado mod)

Tabla 2 – Tipos de operadores aritméticos en JAVA

Cabe destacar que el operador % es de uso exclusivo entre enteros. $7 \% 3$ devuelve 1 ya que el resto de dividir 7 entre 3 es 1. Al valor obtenido lo denominamos módulo (en otros lenguajes en vez del símbolo % se usa la palabra clave mod) y a este operador a veces se le denomina “operador módulo”.

Las operaciones con operadores siguen un **orden de prelación o de precedencia** que determinan el orden con el que se ejecutan. Si existen expresiones con varios operadores del mismo nivel, la operación se ejecuta de izquierda a derecha. Para evitar resultados no deseados, en casos donde pueda existir duda se recomienda el uso de paréntesis para dejar claro con qué orden deben ejecutarse las operaciones. Por ejemplo, si dudas si la expresión:

$3 * 2 / 7 + 2$

se ejecutará en el orden que esperas, es posible agrupar términos utilizando paréntesis. Por ejemplo:

$3 * ((a / 7) + 2)$

Se podrían mostrar también los resultados de las dos expresiones, para ver las diferencias.

Operadores lógicos

En Java disponemos de los operadores lógicos habituales en lenguajes de programación como son “es igual”, “es distinto”, menor, menor o igual, mayor, mayor o igual, and (y), or (o) y not (no). La sintaxis se basa en símbolos como veremos a continuación y cabe destacar que hay que prestar atención a no confundir == con = porque implican distintas cosas.

OPERADOR	DESCRIPCIÓN
==	Es igual
!=	Es distinto
<, <=, >, >=	Menor, menor o igual, mayor, mayor o igual
&&	Operador and (y)
	Operador or (o)
!	Operador not (no)

Tabla 3 – Tipos de operadores lógicos en JAVA

Los conectivos lógicos (AND, OR y NOT), sirven para evaluar condiciones complejas. NOT sirve para negar una condición. Ejemplo:

```
boolean mayorDeEdad, menorDeEdad;
int edad = 21;
mayorDeEdad = edad >= 18; //mayorDeEdad será true menorDeEdad = !mayorDeEdad;
//menorDeEdad será false
```

El operador `&&` (AND) sirve para evaluar dos expresiones de modo que, si ambas son ciertas, el resultado será true sino el resultado será false. Ejemplo:

```
boolean carnetConducir=true;
int edad=20;
boolean puedeConducir= (edad>=18) && carnetConducir; //Si la edad es de al
menos 18 años y carnetConducir es //true, puedeConducir es true
```

El operador `||` (OR) sirve también para evaluar dos expresiones. El resultado será true si al menos una de las expresiones es true. Ejemplo:

```
boolean nieva =true, llueve=false, graniza=false;
malTiempo= nieva || llueve || graniza;
```

Los operadores `&&` y `||` se llaman operadores en cortocircuito porque si no se cumple la condición de un término no se evalúa el resto de la operación. Por ejemplo:

$$(a == b \&\& c != d \&\& h >= k)$$

tiene tres evaluaciones: la primera comprueba si la variable `a` es igual a `b`. Si no se cumple esta condición, el resultado de la expresión es falso y no se evalúan las otras dos condiciones posteriores.

En un caso como:

$$(a < b \mid\mid c != d \mid\mid h <= k)$$

se evalúa si `a` es menor que `b`. Si se cumple esta condición el resultado de la expresión es verdadero y no se evalúan las otras dos condiciones posteriores.

Operadores de asignación

Permiten asignar valores a una variable. El fundamental es `=`. Pero sin embargo se pueden usar expresiones más complejas como:

```
x = 5;
x += 3; // x vale 8 ahora
```

En el ejemplo anterior lo que se hace es sumar 3 a la `x` (es lo mismo `x+=3`, que `x=x+3`).

Eso se puede hacer también con todos estos operadores:

Nombre	Operador	Ejemplo	Equivalencia
Operadores aritméticos			
Suma y asignación	$+=$	$a += b$	$a = a+b$
Resta y asignación	$-=$	$a -= b$	$a = a-b$
Multiplicación y asignación	$*=$	$a *= b$	$a = a*b$
División y asignación	$/=$	$a /= b$	$a = a/b$
Resto de la división y asignación	$%=$	$a %= b$	$a = a \% b$
Operadores a nivel de bits			
AND binario y asignación	$\&=$	$a \&= b$	$a = a \& b$
OR binario y asignación	$ =$	$a = b$	$a = a b$
XOR binario y asignación	$^=$	$a ^= b$	$a = a ^ b$
Desplazamiento de bits hacia la izquierda en b posiciones y asignación	$<<=$	$a <<= b$	$a = a << b$
Desplazamiento de bits hacia la derecha en b posiciones y asignación	$>>=$	$a >>= b$	$a = a >> b$

Tabla 4 – Operadores de Asignación en JAVA

También se pueden concatenar asignaciones:

```
x1 = x2 = x3 = 5; // todas valen 5
```

Otros operadores de asignación son “**++**” (incremento) y “**--**”(decremento), que incrementan o decrementan en una unidad el valor de la variable. Son muy utilizados en algoritmos de recorrido o dentro de estructuras de control como el ciclo for que vimos en el módulo de Técnicas de Programación.

Pero hay dos formas de utilizar el incremento y el decremento. Se puede usar por ejemplo **x++** o **++x**, la diferencia estriba en el modo en el que se comporta la asignación en cuanto al orden en que es evaluada.

Ejemplo:

```
int x=5, y=5, z;
z=x++; // z vale 5, x vale 6
z=++y; // z vale 6, y vale 6
```

Operador ternario

Este operador (conocido como if de una línea) permite devolver un valor u otro según el valor de la expresión analizada. Su sintaxis es la siguiente:

Expresionlogica ? valorSiVerdadero: valorSiFalso;

Como, por ejemplo:

```
float precioDeLista = aplicaDescuento == true ? 123 : 234;
```

En el caso de que la variable `aplicaDescuento` tenga valor verdadero, el precio de lista del producto que estamos analizando será menor; es importante destacar que la asignación anterior es equivalente a utilizar condicionales como veremos en las siguientes secciones.

Arrays

En Java los arrays son objetos, instancias de la clase `Array`, la cual dispone de ciertos métodos útiles. La declaración sigue la siguiente sintaxis: se debe declarar el tipo base de los elementos del array. El tipo base puede ser un tipo primitivo o un tipo de referencia:

```
int arrayDeEnteros[] = null; // Declara un array de enteros
Punto arrayDePuntos[] = null; /* Declara un array de referencias a Puntos */
```

La creación del array se hace, como con cualquier otro objeto, mediante el uso del operador `new()`:

```
arrayDeEnteros = new int[100]; /* Crea el array con espacio para 100 enteros */
arrayDePuntos = new Punto[100]; /* Crea el array con espacio para 100 referencias
a Punto */
```

En el primer caso se reserva espacio para contener 100 enteros. En el segundo caso se crea espacio para contener 100 referencias a objetos de la clase `Punto`, pero no se crea cada uno de esos 100 objetos. En el siguiente ejemplo se muestra como se crea cada uno de esos 100 objetos de la clase `Punto` y se asignan a las referencias del array.

```
for(int i = 0; i < 100; i++)
    arrayDePuntos[i] = new Punto();
```

Los arrays se pueden iniciar en el momento de la creación, como en el siguiente ejemplo:

```
int arrayDeEnteros[] = {1, 2, 3, 4, 5};
Punto arrayDePuntos[] = {new Punto(), new Punto(1.0f, 1.0f)};
```

Los arrays disponen de un atributo llamado `length`, que significa longitud, indica el número de elementos que contiene, al que se puede acceder como sigue:

```
int arrayDeEnteros[] = {1, 2, 3, 4, 5};
int tamaño = arrayDeEnteros.length; // el valor es 5
```

Cadenas de caracteres

En Java existe una clase para representar y manipular cadenas, la clase `String`. Una vez creado un `String` no se puede modificar. Se pueden crear instancias de una manera abreviada y sobre ellas se puede utilizar el operador de concatenación `+`:

```
String frase = "Esta cadena es una frase "
String larga = frase + "que se puede convertir en una frase larga."
```

Constantes

Una constante es una “variable” de solo lectura. Dicho de otro modo, más correcto, es un valor que no puede variar (por lo tanto, no es una variable en sí).

La forma de declarar constantes es la misma que la de crear variables, sólo que hay que anteponer la palabra final que es la que indica que estamos declarando una constante y por tanto no podremos variar su valor inicial durante la ejecución del programa:

```
final double PI = 3.141591;
```

Es una buena práctica comúnmente aceptada el asignar nombres en mayúscula sostenida para las constantes a fin de diferenciarlas luego en el código del programa.

Estructuras de control

Estructuras condicionales

Dos son las estructuras de control condicionales en Java: bifurcación y selección múltiple.

Bifurcación: if-else, if-else-if

Su sintaxis es:

```
if (condicion)
{
    instruccion1();
    instruccion2();
    // etc
} else
{
    instruccion1();
    instruccion2();
    // etc
}
```

Es necesario que la condición sea una variable o expresión booleana. Si sólo existe una instrucción en el bloque, las llaves no son necesarias. No es necesario que exista un bloque else, como en el siguiente ejemplo:

```
if (condicion)
{
    bloqueDeInstrucciones();
}

// continua la ejecución del programa
```

```

    }
else
{
    if(condicion2)
    {
        bloqueDeInstrucciones();
    }
else
{
    bloqueDeInstrucciones();
}
}

```

Cada cláusula else corresponde al último if inmediato anterior que se haya ejecutado, es por eso que debemos tener especial consideración de encerrar correctamente entre llaves los bloques para determinar exactamente a qué cláusula corresponde. En este caso, es de especial utilidad indentar nuestro código utilizando espacios o tabulaciones como se muestra en el ejemplo anterior."

Un ejemplo del uso para un caso en particular es el siguiente:

```

final int totalMaterias = 4;
int materiasAprobadas = 2;

if (materiasAprobadas == totalMaterias)
{
    otorgarCertificado();
}
else
{
    continuarCapacitacion();
}

```

Selección múltiple: switch

Su sintaxis es la siguiente:

```

switch (expresion)
{
    case valor1:
        instrucciones();
        break;
    case valor2:
        instrucciones();
        break;
    default:
        instrucciones();
}

```

La expresión ha de ser una variable de tipo int o una expresión que devuelva un valor entero. Cuando se encuentra coincidencia con un **case** se ejecutan las instrucciones a él asociadas hasta encontrar el primer **break**. Si no se encuentra ninguna coincidencia se ejecutan las instrucciones del bloque **default**, la cual es opcional. El **break** no es exigido en la sintaxis y entonces si no se pone, el código se ejecuta atravesando todos los cases hasta que encuentra uno.

Bucles

Un bucle se utiliza para realizar un proceso repetidas veces. Se denomina también lazo o loop. El código incluido entre las llaves {} (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumpla unas determinadas condiciones. Hay que prestar especial atención a los bucles infinitos, hecho que ocurre cuando la condición de finalizar el bucle (expresión booleana) no se llega a cumplir nunca.

Bucle While

Las sentencias encerradas por llaves se ejecutan mientras **condición** tenga un valor verdadero:

```
while (condicion)
{
    accion();
}
```

Bucle For

La forma general del bucle for es la siguiente:

```
for (inicio; condicion; incremento)
{
    accion();
}
```

que es equivalente a utilizar while en la siguiente forma,

```
inicio();
while (condicion)
{
    accion();
    incremento();
}
```

Bucle Do While

Es similar al bucle while pero con la particularidad de que el control está al final del bucle lo que hace que el cuerpo del bucle se ejecute al menos una vez, independientemente de que la condición se cumpla o no. Una vez ejecutados el cuerpo, se evalúa la condición: si resulta true se vuelven a ejecutar las sentencias incluidas en el bucle, mientras que si la condición se evalúa a false finaliza el bucle.

```
do
{
    acciones();
}
while (condicion);
```

Sentencias break y continue

La sentencia **break** es válida tanto para las bifurcaciones como para los bucles. Hace que se salga inmediatamente del bucle o bloque que se está ejecutando sin finalizar el resto de las sentencias.

La sentencia **continue** se utiliza en los bucles (no en bifurcaciones). Finaliza la iteración que en ese momento se está ejecutando (no ejecuta el resto de las sentencias que hubiera hasta el final del cuerpo del bucle). Vuelve al comienzo del bucle y comienza la siguiente iteración si existiera.

Bloques para manejo de excepciones

Java incorpora en el propio lenguaje la gestión de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo, prácticamente solo los errores de sintaxis son detectados en esta operación. El resto de problemas surgen durante la ejecución de los programas.

En el lenguaje *Java*, una *Exception* es una clase que representa un tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas *excepciones* son *fatales* y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras *excepciones*, como por ejemplo no encontrar un archivo en el que hay que leer o escribir algo, pueden ser *recuperables*. En este caso el programa debe dar al usuario la oportunidad de corregir el error (dando por ejemplo la opción de seleccionar un nuevo archivo).

Existen algunos tipos de excepciones que *Java* obliga a tener en cuenta. Esto se hace mediante el uso de bloques *try*, *catch* y *finally*.

```
try {
    // código que puede arrojar una excepción
}
catch (Exception ex) { // atrapamos el error producido
    // lo procesamos
}
finally {
    // y finalmente lo post procesamos
}
```

El código dentro del bloque *try* está *vigilado*: si se produce una situación anormal y se lanza como consecuencia una excepción, el control pasa al bloque *catch* que se hace cargo de la situación y decide lo que hay que hacer. Se pueden incluir tantos bloques *catch* como se desee, cada uno de los cuales tratará un tipo de excepción. Finalmente, si está presente, se ejecuta el bloque *finally*, que es opcional, pero que en caso de existir se ejecuta siempre, sea cual sea el tipo de error. Es importante destacar que el parámetro del bloque *catch* es un objeto de una clase que hereda de *Exception* (veremos esto más adelante), pero por ahora debemos saber que este objeto puede contener información importante acerca de la causa del error.

En el caso en que el código de un método pueda generar una *Exception* y no se desee incluir en dicho método la gestión del error (es decir los bloques *try/catch* correspondientes), es necesario que el método pase la *Exception* al método desde el que ha sido llamado. Esto se consigue mediante la adición de la palabra *throws* seguida del nombre de la *Exception* concreta, después de la lista de argumentos del método. A su vez el método superior deberá incluir los bloques *try/catch* o volver a pasar la *Exception*. De esta forma se puede ir pasando la *Exception* de un método a otro hasta llegar al último método del programa, el método *main()*.

En el peor de los casos que nadie se haga cargo de la excepción ésta llegará al usuario en forma de un mensaje de “Excepción no capturada” lo cual no es una buena práctica.

Como veremos más adelante para algunos tipos de excepciones es obligatorio el uso de los bloques try-catch para manejar los posibles resultados, o en el último de los casos, el método de ejecución main(String[] args) deberá arrojarla explícitamente usando la palabra reservada throws y llegará al usuario final, lo cual no es en absoluto recomendable. En cambio, algunas excepciones propias del lenguaje Java heredan de la clase RuntimeException y se utilizan para tratar errores que pueden ocurrir al momento de ejecutar el programa posiblemente derivados del manejo de datos que no pudieron ser previstos en el momento de la programación. Estas excepciones no necesitan ser capturadas en un bloque try-catch de forma explícita.

Clases y Objetos

Como vimos anteriormente, las *clases* son el centro del paradigma de *Programación Orientada a Objetos (POO)*. Algunos conceptos importantes de la POO son los siguientes:

1. **Encapsulamiento:** Las clases pueden ser declaradas como públicas (*public*) y como paquete (*package*) (accesibles sólo para otras clases del mismo paquete). Las variables miembros y los métodos pueden ser *public*, *private*, *protected* y *package*. De esta forma se puede controlar el acceso entre objetos y evitar un uso inadecuado.
2. **Herencia:** Una clase puede衍生 de otra (*extends*), y en ese caso hereda todas sus variables y métodos. Una clase derivada puede *añadir* nuevas variables y métodos y/o *redefinir* las variables y métodos heredados.
3. **Polimorfismo:** Los objetos de distintas clases pertenecientes a una misma jerarquía o que implementan una misma interface, pueden responder de forma indistinta a un mismo método. Esto, como se ha visto anteriormente, facilita la programación y el mantenimiento del código.

A continuación, veremos cómo se declaran tanto clases como interfaces, y cuál es el proceso para crear sus instancias.

Intuitivamente, una clase es una agrupación de *datos* (variables o campos) y de *funciones* (métodos) que operan sobre esos datos. Todos los métodos y variables deben ser definidos dentro del *bloque* {...} de la clase.

Un *objeto* (en inglés *instance*) es un ejemplar concreto de una clase. Las *clases* son como tipos de variables, mientras que los *objetos* son como variables concretas de un tipo determinado.

```
NombreDeLaClase unObjeto;
NombreDeLaClase otroObjeto;
```

A continuación, se enumeran algunas características importantes de las clases:

1. Todas las variables y métodos de *Java* deben pertenecer a una clase. No hay variables y funciones globales.
2. Si una clase deriva de otra (*extends*), hereda todas sus variables y métodos.
3. *Java* tiene una jerarquía de clases estándar de la que pueden derivar las clases que crean los usuarios. Es decir que toda clase definida por el programador es heredada de la clase Object definida por el lenguaje de programación.

4. Una clase sólo puede heredar de una única clase (en *Java* no hay herencia múltiple). Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de *Object*. La clase *Object* es la base de toda la jerarquía de clases de *Java*.
5. En un archivo de código fuente se pueden definir varias clases, pero en un mismo archivo no puede haber más que una clase definida como *public*. Este archivo se debe llamar como la clase *public* que debe tener extensión *.java*. Con algunas excepciones, lo habitual es escribir una sola clase por archivo.
6. Si una clase contenida en un fichero no es *public*, no es necesario que el fichero se llame como la clase.
7. Los métodos y variables de una clase pueden referirse de modo global a un *objeto* de esa clase a la que se aplican por medio de la referencia *this*. Al utilizar la palabra reservada 'this' para referirse tanto a métodos como atributos se restringe el ámbito al objeto que hace la declaración.
8. Las clases se pueden agrupar en *packages que significa paquetes*, introduciendo una línea al comienzo del fichero (*package packageName;*). Esta agrupación en *packages* está relacionada con la jerarquía de carpetas y archivos en la que se guardan las clases.
En la práctica usamos paquetes para agrupar clases con un mismo propósito usando jerarquía de paquetes; esta decisión es muy importante a la hora de diseñar la estructura de nuestro programa.

Variables miembros de objeto

La programación orientada a objetos está *centrada en los datos*. Una clase son *datos y métodos* que operan sobre esos datos.

Cada objeto, es decir cada instancia concreta de una clase, tiene su propia copia de las variables miembro. Las variables miembros de una clase (también llamadas *atributos*) pueden ser de *tipos primitivos* (*boolean*, *int*, *long*, *double*, ...) o referencias a *objetos* de otra clase (*agregación* y *composición*).

Un aspecto muy importante para el correcto funcionamiento de los programas es que no haya datos sin inicializar. Por eso las variables miembros de *tipos primitivos* se inicializan siempre de modo automático, incluso antes de llamar al *constructor* (*false* para *boolean*, el carácter nulo para *char* (código Unicode '\u0000') y cero para los tipos numéricos). De todas formas, lo más adecuado es inicializarlas en el constructor.

También pueden inicializarse explícitamente en la *declaración*, como las variables locales, por medio de constantes o llamadas a métodos. Por ejemplo,

```
public class Alumno
{
    int edad = 18;
}
```

Las variables miembros se inicializan en el mismo orden en que aparecen en el código de la clase. Esto es importante porque unas variables pueden apoyarse en otras previamente definidas.

Cada *objeto* que se crea de una clase tiene *su propia copia* de las variables miembro. Por ejemplo, cada objeto de la clase *Circulo* tiene sus propias coordenadas del centro *x* e *y*, y su propio valor del radio *r*. Se puede aplicar un método a un objeto concreto poniendo el nombre del objeto y luego el nombre del

método separados por un punto. Por ejemplo, para calcular el área de un objeto de la clase *Circulo* llamado *c1* se escribe: *c1.area()*;

La definición de cada atributo debe empezar con un modificador de acceso. Los modificadores de acceso indican la visibilidad, es decir, si se puede tener acceso sólo desde la clase (*private*), desde la clase y las clases que heredan de ella (*protected*) y desde cualquier clase definida en el mismo paquete o desde cualquier clase (*public*).

Tras el modificador de acceso se escribe el tipo del argumento, este puede ser un tipo primitivo o de tipo referencia. Tras el modificador de acceso, un atributo se puede declarar como *static*. Esto implica que no existe una copia de este atributo en cada instancia de la clase, si no que existe uno único común a todas las instancias. A los atributos *static* también se les llama atributos de clase.

Otro modificador que puede afectar al comportamiento de los atributos de una clase es *final*. Si un atributo se declara como *final*, implica que no se puede cambiar su valor una vez definido. Un ejemplo de uso de este modificador son las constantes de clase:

```
public static final float E = 2.8182f;
```

Una clase puede tener variables propias de la clase y no de cada objeto. A estas variables se les llama *variables de clase* o *variables static*. Las variables *static* se suelen utilizar para definir constantes comunes para todos los objetos de la clase (por ejemplo, *PI* en la clase *Circulo*) o variables que sólo tienen sentido para toda la clase (por ejemplo, un contador de objetos creados como *numCirculos* en la clase *Circulo*).

Las variables de clase se crean anteponiendo la palabra *static* a su declaración como en el ejemplo anterior. Para llamarlas se suele utilizar el nombre de la clase (no es imprescindible, pues se puede utilizar también el nombre de cualquier objeto), porque de esta forma su sentido queda más claro. Por ejemplo, *Circulo.numCirculos* es una variable de clase que cuenta el número de círculos creados.

Si no se les da valor en la declaración, las variables miembros *static* se inicializan con los valores por defecto (*false* para *boolean*, el carácter nulo para *char* (código Unicode '\u0000') y cero para los tipos numéricos) para los tipos primitivos, y con *null* si es una referencia.

Las variables miembros *static* se crean en el momento en que pueden ser necesarias: cuando se va a crear el primer objeto de la clase, en cuanto se llama a un método *static* o en cuanto se utiliza una variable *static* de dicha clase. Lo importante es que las variables miembros *static* se inicializan siempre antes que cualquier objeto de la clase.

Métodos

Los métodos especifican el comportamiento de la clase y sus instancias. Los modificadores de acceso y su significado son los mismos que al operar sobre atributos. En particular, al declarar un método estático implica que es un método de la clase, y por lo tanto no es necesario crear ninguna instancia de la clase para poder llamarlo. El conjunto de los métodos públicos de una clase forma su interface.

Un método declarado *final* implica que no se puede redefinir en ninguna clase que herede de esta, veremos el tema de Herencia más adelante, pero es importante destacar que el cuerpo de un método definido como *final* no podrá ser modificado por ninguna clase hijo.

En el momento de la declaración hay que indicar cuál es el tipo del parámetro que devolverá el método o *void* en caso de que no devuelva nada. En otros lenguajes, estos tipos de métodos o funciones se denominan procedimientos.

Los métodos tienen *visibilidad directa* de las variables miembro del objeto que es su *argumento implícito*; es decir, pueden acceder a ellas sin cualificarlas con un nombre de objeto y el operador punto (.). De todas formas, también se puede acceder a ellas mediante la referencia *this*, de modo discrecional (como en el ejemplo anterior con *this.r*) o si alguna variable local o argumento las oculta.

Los métodos pueden definir *variables locales*. Su visibilidad llega desde la definición al final del bloque en el que han sido definidas. No hace falta inicializar las variables locales en el punto en que se definen, pero el compilador no permite utilizarlas sin haberles dado un valor. A diferencia de las variables miembro, las variables locales no se inicializan por defecto.

También se ha de especificar el tipo y nombre de cada uno de los argumentos del método entre paréntesis. Si un método no tiene argumentos el paréntesis queda vacío, no es necesario escribir void. El tipo y número de estos argumentos identifican al método, ya que varios métodos pueden tener el mismo nombre, con independencia del tipo devuelto, y se distinguirán entre sí por el número y tipo de sus argumentos, como veremos a continuación.

Ejemplo:

```
public class Alumno
{
    private String nombre;
    private String apellido;

    public String getNombreCompleto ()
    {
        return nombre + " " + this.apellido;
    }
}
```

En el ejemplo anterior hemos definido la clase Alumno con dos atributos de tipo cadena de texto: nombre y apellido. Es importante destacar que estos atributos son definidos como privados para poder respetar el principio de encapsulamiento del paradigma orientado a objetos y lo tomaremos como una buena práctica: los atributos de un objeto deberían ser accesibles sólo a través de métodos públicos. Además hemos definido el método público *getNombreCompleto()* que no recibe parámetros y devuelve una cadena de texto formada por el nombre y el apellido del alumno separados por un espacio. Es importante destacar que para acceder a los atributos de la instancia de esta clase puede usarse o no la palabra reservada *this* ya que en el cuerpo del método no existe otra variable local con el mismo nombre que pueda resultar en ambigüedad.

Sobre el paso de parámetros a un método

En Java los argumentos de los *tipos primitivos* se pasan siempre *por valor*. El método recibe una copia del argumento actual; si se modifica esta copia, el argumento original que se incluyó en la llamada no queda modificado. Para modificar un argumento de un tipo primitivo dentro del cuerpo del método puede incluirse como variable miembro o ser retornado para luego realizar la asignación en el momento de la llamada. Las *referencias* se pasan también *por valor*, pero a través de ellas se pueden modificar los objetos referenciados.

Sobre los métodos de seteo (get y set)

En el ejemplo anterior hemos visto que el prefijo del nombre es get, y en la práctica veremos que es de uso común nombrar métodos de obtención y modificación con los prefijos get y set respectivamente. Para poder respetar la buena práctica antes mencionada relacionada con el encapsulamiento de la orientación a objetos cada atributo de la clase debería ser definido como privado y existir métodos get y set para poder obtener y modificar sus valores.

Así, la definición de un método get para obtener el valor del atributo de un objeto tendría la siguiente forma:

```
public TipoDeDatos getAtributo ()
{
    return this.atributo;
}
```

Donde TipoDeDatos puede ser tanto un tipo primitivo como una clase o interfaz si estamos referenciando a otros objetos. Otro punto importante es que por convención el nombre del método luego del prefijo get o set normalmente se escribe en CamelCase como mencionamos en el módulo anterior a la hora de nombrar variables.

De esta forma, los métodos set reciben como parámetro el nuevo valor y no retornan nada, por lo que el tipo especificado es void.

```
public void setAtributo (TipoDeDatos nuevoValor)
{
    this.atributo = nuevoValor;
}
```

Ejemplo:

```
public class Alumno
{
    private String nombre;
    private String apellido;

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

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

    public String getApellido ()
    {
        return this.apellido;
    }

    public void setApellido (String apellido)
    {
        this.apellido = apellido;
    }
}
```

```

public String getNombreCompleto ()
{
    return this.nombre + " " + this.apellido;
}
}

```

Entonces ahora podríamos crear un par de instancias de alumnos y mostrar en pantalla sus nombres completos de la siguiente forma:

```

Alumno alumno1 = new Alumno();
alumno1.setNombre("Pablo");
alumno1.setApellido("Filippo");

Alumno alumno2 = new Alumno();
alumno2.setNombre("Florencia");
alumno2.setApellido("Venne");

System.out.println("Nombre del alumno 1: " + alumno1.getNombreCompleto());
System.out.println("Nombre del alumno 2: " + alumno2.getNombreCompleto());

```

En programación orientada a objetos las llamadas a los métodos se les llama paso de mensajes, llamar a un método es análogo a pasarle un mensaje.

El método main()

Existe un nombre de método que está reservado en Java y otros lenguajes: el método main. Este método es especial ya que es el que da lugar al inicio del programa y será llamado por la máquina virtual al momento de la ejecución.

Es importante tener claro que el método main() no es el elemento principal en el desarrollo del programa. El programa, de acuerdo con el paradigma de programación orientada a objetos, se desarrolla mediante la interacción entre objetos por lo que el objetivo de este método normalmente es iniciar el programa delegar el comportamiento a los distintos los objetos correspondientes.

Este método debe pertenecer a una clase pública y su definición es la siguiente:

```

public static void main(String[] args)
{
    // cuerpo de nuestro programa
}

```

Es estático ya que no depende de una instancia en particular de la clase en la que se declara y no tiene ningún valor de retorno. Podemos ver que recibe un array de parámetros de tipo String que representan los argumentos pasados a la hora de ejecutar el programa. En el caso de realizar la ejecución mediante la línea de comando, cada elemento del array será una cada cadena de texto luego de la llamada a nuestro programa, separadas por espacios, ejemplo:

>> java ejemplo1 paso parámetros a mi programa

En este caso, al ejecutar el programa llamada ejemplo1, el método main() recibirá el array args[] compuesto de la siguiente forma:

```
args[0] => "paso"
args[1] => "parámetros"
args[2] => "a"
args[3] => "mi"
args[4] => "programa"
```

Métodos sobrecargados y redefinición (overload y override)

Java permite métodos *sobrecargados (overloaded)*, es decir métodos distintos con *el mismo nombre* que se diferencian por el número y/o tipo de datos de los argumentos.

A la hora de llamar a un método sobrecargado, Java sigue unas reglas para determinar el método concreto que debe llamar:

1. Si existe el método cuyos argumentos se ajustan exactamente al tipo de los argumentos de la llamada (argumentos actuales), se llama ese método.
2. Si no existe un método que se ajuste exactamente, se intenta promover los argumentos actuales al tipo inmediatamente superior (por ejemplo *char* a *int*, *int* a *long*, *float* a *double*, etc.) y se llama el método correspondiente.
3. Si sólo existen métodos con argumentos de un tipo más amplio (por ejemplo, *long* en vez de *int*), el programador debe hacer un *cast* explícito en la llamada, responsabilizándose de esta manera de lo que pueda ocurrir.
4. El valor de retorno no influye en la elección del método sobrecargado. En realidad, es imposible saber desde el propio método lo que se va a hacer con él. No es posible crear dos métodos sobrecargados, es decir con el mismo nombre, que sólo difieran en el valor de retorno.

Diferente de la *sobre carga* de métodos es la *redefinición*. Una clase puede *redefinir (override)* un método heredado de una superclase. *Redefinir* un método es dar una nueva definición. En este caso el método debe tener exactamente los mismos argumentos en tipo y número que el método redefinido. Este tema se verá de nuevo al hablar de la *Herencia*.

Ejemplo:

```
public class Alumno
{
    private String nombre;
    private String apellido;

    // se omiten los métodos get y set por simplicidad

    public String getNombreCompleto (String titulo)
    {
        return titulo + " " + this.nombre + " " + this.apellido;
    }
    public String getNombreCompleto ()
    {
```

```

        return this.getNombreCompleto("Sr/a.");
    // usamos un título por defecto para cuando no se especifica por parámetros
}
}

```

Entonces para la ejecución del ejemplo anterior que coloca como prefijo del nombre del alumno el título en caso de tenerlo o "Sr/a." en su defecto tenemos lo siguiente:

```
System.out.println("Nombre del alumno 1: " + alumno1.getNombreCompleto());
// muestra en pantalla "Sr/a. Pablo Filippo"
```

```
System.out.println("Nombre del alumno 2: " +
alumno2.getNombreCompleto("Contadora."));
// muestra en pantalla "Contadora. Paula Filippi"
```

Constructores de Objetos

Los métodos que tienen el mismo nombre que la clase tienen un comportamiento especial, sirven para crear las instancias de la clase y se les denomina constructores. Un *constructor* es un operador que se llama automáticamente cada vez que se crea un objeto de una clase. La principal misión del *constructor* es reservar memoria e inicializar las variables miembros de la clase. A continuación, se listan los principales aspectos de los constructores en Java que es necesario entender:

- Cuando se llama al constructor de una clase para instanciarla y crear el objeto, se invoca al constructor.
- Los constructores no tienen valor de retorno (ni siquiera void) y su nombre es el mismo que el de la clase.
- Su argumento implícito es el objeto que se está creando.
- Si no se define explícitamente un constructor, Java lo hará por nosotros ya que siempre es necesario que exista. Se creará un constructor sin argumentos.
- Una clase puede tener *varios constructores*, que se diferencian por el tipo y número de sus argumentos (los constructores justamente son un ejemplo típico de métodos *sobrecargados* que vimos anteriormente).

Si es necesario que un constructor llame a otro constructor lo debe hacer antes que cualquier otra cosa. Se llama *constructor por defecto* al constructor que no tiene argumentos. El programador debe proporcionar en el código, valores iniciales adecuados para todas las variables miembro. En caso de que sólo definamos un constructor con parámetros el constructor por defecto no será creado por Java, por lo que deberemos definirlo explícitamente en caso de ser necesario.

Por ejemplo:

```
public class Alumno
{
    private String nombre;
    private String apellido;

    public Alumno ()
    {

    }

    public Alumno (String nombre, String apellido)
```

```

{
    this.nombre = nombre;
    this.apellido = apellido;
}

// se omiten los métodos get y set por simplicidad
}

```

De esta forma podemos crear los mismos alumnos anteriores de la forma:

```

Alumno alumno1 = new Alumno("Pablo", "Filippo");
Alumno alumno2 = new Alumno("Florencia", "Venne");

```

Al igual que los demás métodos de una clase, los *constructores* pueden tener también los modificadores de acceso *public*, *private*, *protected* y *package*. Si un *constructor* es *private*, ninguna otra clase puede crear un objeto de esa clase. En este caso, puede haber métodos *public* y *static* (*factory methods*) que llamen al *constructor* y devuelvan un objeto de esa clase.

Dentro de una clase, los *constructores* sólo pueden ser llamados por otros *constructores* o por métodos *static*. No pueden ser llamados por los *métodos de objeto* de la clase.

Finalización y Destrucción de Objetos

Como mencionamos anteriormente, en *Java* el sistema se ocupa automáticamente de liberar la memoria de los objetos que ya han *perdido la referencia*, esto es, objetos que ya no tienen ningún nombre que permita acceder a ellos, por ejemplo, por haber llegado al final del bloque en el que habían sido definidos (se acabó su *scope*), porque a la *referencia* se le ha asignado el valor *null* o porque a la *referencia* se le ha asignado la dirección de otro objeto. A esta característica de *Java* se le llama *garbage collection* (recogida de basura).

Antes de que un objeto sea completamente eliminado de la memoria por el *recolector de basura*, se llama a su método *finalize()*. Este método está definido en la clase *Object* de la que hereda implícitamente cualquier nueva clase.

Un *finalizador* es un método de objeto (no *static*), sin valor de retorno (*void*), sin argumentos y que siempre se llama *finalize()*. Los *finalizadores* se llaman de modo automático siempre que hayan sido definidos por el programador de la clase. Para realizar su tarea correctamente, un *finalizador* debería terminar siempre llamando al *finalizador* de su *super-clase*.

Agrupando Clases en Paquetes

Un *package* es una agrupación de clases que sirve para establecer una jerarquía lógica en la organización de las clases. Además, tiene una relación directa con la organización física de nuestro código ya que también se representa en la estructura de archivos y carpetas que conforman nuestro programa. Al estructurar de este modo las clases, estamos estableciendo un dominio de nombres que la máquina virtual de *Java* utiliza, por ejemplo, cuando nuestra aplicación utiliza clases distribuidas en una red de computadores.

Todas las clases dentro de un mismo paquete tienen acceso al resto de clases declaradas como públicas. Para poder acceder a una clase de otro paquete se ha de importar anteriormente mediante la sentencia *import*.

Para que una clase pase a formar parte de un *package* llamado *nombreDelPaquete*, hay que introducir en ella la sentencia:

```
package nombreDePaquete;
```

Debe ser la primera sentencia del archivo sin contar comentarios y líneas en blanco.

Los nombres de los *packages* se suelen escribir con minúsculas, para distinguirlos de las clases, que empiezan por mayúsculo. El nombre de un *package* puede constar de varios nombres unidos por puntos (los propios *packages* de Java siguen esta norma, como por ejemplo *java.awt.event*).

Todas las clases que forman parte de un *package* deben estar en la misma carpeta. Los nombres compuestos de los *packages* están relacionados con la jerarquía de carpetas en que se guardan las clases. Es recomendable que los *nombres de las clases* de Java sean únicos, es el nombre del *package* lo que permite obtener esta característica.

En un programa de Java, una clase puede ser referida con su nombre completo (el nombre del *package* más el de la clase, separados por un punto). También se pueden referir con el nombre completo las variables y los métodos de las clases. Esto se puede hacer siempre de modo opcional, pero es incómodo y hace más difícil el reutilizar el código y portarlo a otras máquinas.

La sentencia *import* permite abreviar los nombres de las clases, variables y métodos, evitando el tener que escribir continuamente el nombre del *package* importado. Se importan por defecto el *package* *java.lang* y el *package* actual o por defecto (las clases del directorio actual).

Existen dos formas de utilizar *import*: para *una clase* y para *todo un package*:

```
import poo.cine.Actor;
import poo.cine.*;
```

El importar un *package* no hace que se carguen todas las clases del *package*: sólo se cargarán las clases *public* que se vayan a utilizar. Al importar un *package* no se importan los *sub-packages*. Éstos deben ser importados explícitamente, pues en realidad son *packages* distintos. Por ejemplo, al importar *java.awt* no se importa *java.awt.event*.

Imaginemos la siguiente estructura de paquetes y clases:

```
futsal
- torneos
  -- Torneo.java
  -- Fecha.java
  -- Encuentro.java
- equipos
  -- Equipo.java
  -- Jugador.java
- predio
  -- Cancha.java
```

Podríamos definir la clase Equipo de la siguiente forma:

```
package futsal.equipos;

public class Equipo
{
    private String nombre;
    private Jugador[] jugadores;
    // Podríamos seguir listando atributos, pero por simplificación dejamos los establecidos anteriormente.
}
```

Es importante destacar la primera línea "package" en el ejemplo anterior, que es la que indica cuál es el paquete en el que está ubicada la clase Equipo. Además, vemos que nuestra clase tiene relación con la clase Jugador, que al estar en el mismo paquete no debe ser importada ya que comparte la visibilidad. Otro dato importante es que la definición de visibilidad de la clase Equipo es public, lo que nos permitirá utilizarla en otros paquetes.

Por otro lado, podríamos escribir la clase Encuentro entre 2 equipos como sigue:

```
package futsal.torneos;

import futsal.equipos.Equipo;
import futsal.predio.Cancha;

public class Encuentro
{
    private Date fechaHora;
    private Cancha cancha;
    private Equipo equipo1;
    private Equipo equipo2;
    private int golesEquipo1 = 0;
    private int golesEquipo2 = 0;
    private boolean jugado = false;
    ...
}
```

En este caso es necesaria la sentencia import para poder utilizar la clase Equipo ya que se encuentra en otro paquete sobre el que no tenemos visibilidad. Como práctica podrías definir el método obtenerEquipoGanador() que nos retorne el Equipo ganador en caso de que el partido ya se haya jugado?

Herencia de Clases

Como mencionamos en la sección anterior Modelo de Objetos anterior, el concepto de Herencia es de suma importancia en el Paradigma Orientado a Objetos y es una herramienta muy útil a la hora de diseñar nuestros programas. La Herencia define relaciones del tipo "es un" como, por ejemplo "un Alumno es una Persona" por lo que comparte tanto sus atributos como nombre, apellido, dni y su comportamiento, así como tiene sus propios atributos como las materias a las que está inscripto, cuáles ha rendido, entre otras.

Es posible construir una nueva clase a partir de otra mediante el mecanismo de la *herencia*. Mediante el uso de la herencia las clases pueden extender el comportamiento de otra clase permitiendo con ello un gran aprovechamiento del código. Aunque su principal virtud es el Polimorfismo, que veremos luego. Cuando una clase deriva de otra, hereda todas sus variables y métodos. Estas funciones y variables miembro pueden ser *redefinidas (overridden)* en la clase derivada, que puede también definir o añadir nuevas variables y métodos. En cierta forma es como si la *sub-clase* (la clase derivada) “contuviera” un objeto de la *super-clase*; en realidad lo “amplía” con nuevas variables y métodos.

Java permite múltiples niveles de herencia, pero no permite que una clase derive de varias (no es posible la herencia múltiple). Se pueden crear tantas clases derivadas de una misma clase como se quiera.

La herencia induce una jerarquía en forma de árbol sobre las clases con raíz en la clase *Object*. Una clase se dice que hereda o extiende a otra clase antecesora. La palabra reservada *extends* sirve para indicar que una clase extiende a otra. La clase que extiende a otra, hereda todos los atributos y métodos de la clase antecesora, los atributos y métodos privados no tienen visibilidad. La clase antecesora puede extender a su vez otra clase.

Todas las clases de *Java* creadas por el programador tienen una *superclase*. Cuando no se indica explícitamente una *superclase* con la palabra *extends*, la clase deriva de *java.lang.Object*, que es la clase raíz de toda la jerarquía de clases de *Java*. Como consecuencia, todas las clases tienen algunos métodos que han heredado de *Object*.

Un ejemplo de herencia puede ser el siguiente:

```
Persona.java
public class Persona {
    private String nombre;
    private String apellido;
    private String dni;

    public String getNombreCompleto () {
        return this.nombre + " " + this.apellido;
    }

    // ...
}
```

```
Alumno.java
public class Alumno extends Persona {
    private int legajo;
    private Materia[] materiasInscriptas;
    private Materia[] materiasRendidas;

    // ...
}
```

De esta forma cualquier objeto de la clase *Alumno* también tendrá los atributos *nombre*, *apellido* y *dni* y podrá hacer uso del método público *getNombreCompleto()*.

Sobrescritura de variables y métodos

Una clase puede *redefinir* (volver a definir) cualquiera de los métodos heredados de su *superclase* que no sean *final*. El nuevo método sustituye al heredado para todos los efectos en la clase que lo ha redefinido. Además, una clase que extiende a otra puede declarar atributos con el mismo nombre que algún atributo de la clase a la que extiende; se dice que el atributo se ha sobreescrito u ocultado. Un uso de sobre escritura de atributos es la extensión del tipo.

Los métodos de la *super-clase* que han sido redefinidos pueden ser todavía accedidos por medio de la palabra *super* desde los métodos de la clase derivada, aunque con este sistema sólo se puede subir un nivel en la jerarquía de clases.

Finalmente, un método declarado *static* en una clase antecesora puede sobreescibirse en una clase que la extienda, pero también se debe declarar *static*; de lo contrario se producirá un error en tiempo de compilación.

En el ejemplo anterior podríamos redefinir el método *getNombreCompleto()* para que incluya el número de legajo del Alumno a la hora de mostrar su nombre en pantalla.

```
public class Alumno extends Persona {
    private int legajo;
    private Materia[] materiasInscriptas;
    private Materia[] materiasRendidas;

    public String getNombreCompleto () {
        return this.legajo + ":" + super.getNombreCompleto();
    }

    // ...
}
```

Otro ejemplo puede ser el de las Cuentas Bancarias, donde nuestra clase *CuentaBancaria* es heredada por *CuentaCorriente* y *CajaDeAhorro*. De esta forma la clase heredad contiene el atributo *saldo* que es común a ambos tipos de cuenta y define ciertos métodos también comunes como *depositar()* y *extraer()* que reciben un monto como parámetros. Al ser conocida por nosotros la interfaz de la clase *CuentaBancaria* podríamos acceder a estos métodos comunes sin conocer explícitamente la clase que la hereda, esto se conoce como Polimorfismo.

```
CuentaBancaria miCuenta = new CajaDeAhorros();
// realizamos una extracción usando el objeto miCuenta
// cuyo tipo de datos fue definido como CuentaBancaria
miCuenta.extraer(100);
CuentaBancaria miOtraCuenta = new CuentaCorriente();
// realizamos una extracción usando el objeto miOtraCuenta
// cuyo tipo de datos fue definido como CuentaBancaria
// aunque es una instancia de CuentaCorriente
miOtraCuenta.extraer(100);
```

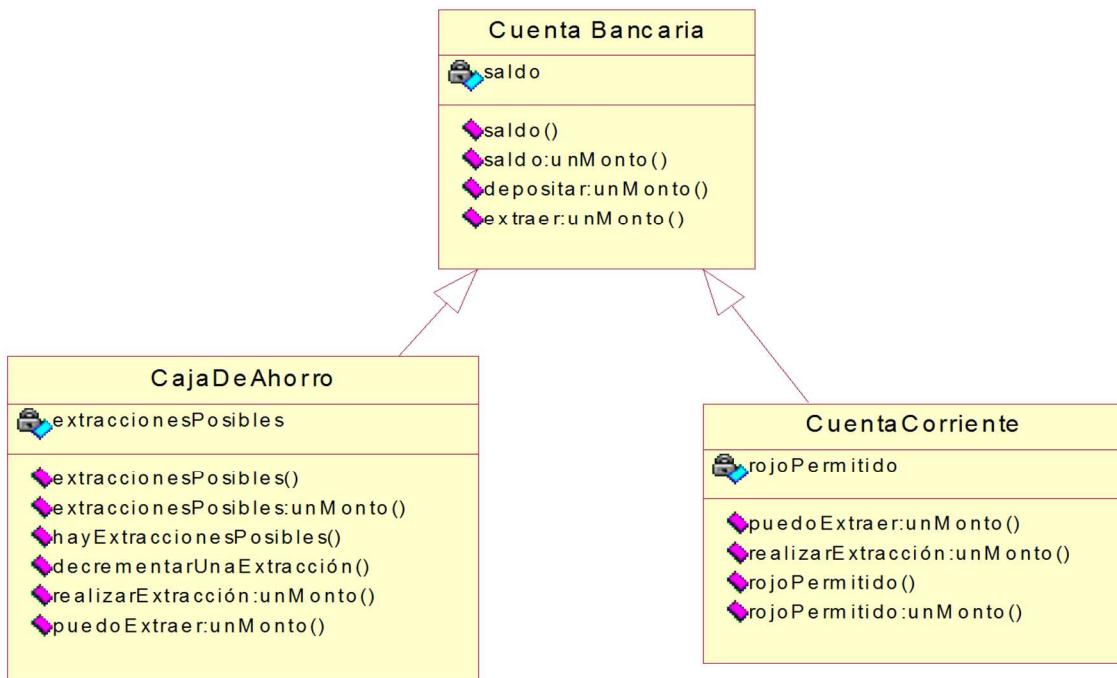


Fig. 2- Ejemplo de herencia y polimorfismo con la Jerarquía de clases de CuentaBancaria

Clases y métodos abstractos

Las clases abstract funcionan como plantillas para la creación de otras clases, son clases de las que no se pueden crear objetos. Su utilidad es permitir que otras clases deriven de ella, proporcionándoles un marco o modelo que deben seguir y algunos métodos de utilidad general. Las clases abstractas se declaran anteponiéndoles la palabra *abstract*, como, por ejemplo

```
public abstract class Geometria { ... }
```

Un método se puede declarar como *abstract*. Un método declarado de esta manera no implementa nada. Si una clase contiene uno o más métodos declarados como *abstract*, ella a su vez debe ser declarada como *abstract*. Las clases que la extiendan deben *obligatoriamente* sobrescribir los métodos declarados como *abstract* o la clase también debe declararse *abstract*.

Interfaces

Así como la Herencia representa relaciones del tipo “es un”, las interfaces nos permiten representar relaciones “se comporta como un”. El concepto de la implementación de interfaces garantiza que objetos de una Clase con esa interfaz tendrán disponibles ciertos métodos definidos. Es en este punto que entra el concepto de Polimorfismo, ya que cualquier objeto independientemente de su clase que implemente una interfaz determinada podrá responder ante la llamada a esos métodos definidos. Las *interfaces* permiten “publicar” el comportamiento de una clase develando un mínimo de información. El nombre de una *interface* se puede utilizar como un *nuevo tipo de referencia*. En este sentido, el nombre de una interface puede ser utilizado en lugar del nombre de cualquier clase que la implemente, aunque su uso estará restringido a los métodos de la *interface*. Un objeto de ese tipo puede también ser utilizado como valor de retorno o como argumento de un método.

Si queremos que una determinada clase vaya a tener cierto comportamiento, hacemos que implemente una determinada interface. En la interface no se implementa el comportamiento, únicamente se especifica cuál va a ser, es decir, se definirán los métodos, pero no se implementan. No se pueden crear instancias de una interface. Todos los métodos declarados en una interface son públicos, así como sus atributos y la misma interface.

Una *clase* puede *implementar* una o varias *interfaces*. Para indicar que una clase implementa una o más interfaces se ponen los nombres de las *interfaces*, separados por comas, detrás de la palabra *implements*, que a su vez va siempre a la derecha del nombre de la clase o del nombre de la superclase en el caso de herencia.

Veamos un ejemplo:

```
public interface DiagramadorDeTorneo {
    public List<Partidos> diagramar (List<Equipos> equipos, Date fechaInicio);
}
```

La interfaz anterior define un método para poder diagramar un Torneo para el cual tenemos diferentes formas de hacerlo, puede ser por eliminatorias, todos contra todos, entre otras formas. Este método debe recibir una lista con los Equipos que se enfrentarán y una fecha que representa el inicio del Torneo. Por otro lado, retorna una lista con los partidos a jugar para la forma de diagramación correspondiente. Así podríamos tener dos clases que implementen esta interfaz:

```
public class TodosContraTodos implements DiagramadorDeTorneo {
    public List<Partidos> diagramar (List<Equipos> equipos, Date fechaInicio) {
        // acá irá el código que genera los partidos con la combinación de todos los
        equipos
    }
}

public class PorEliminatorias implements DiagramadorDeTorneo {
    public List<Partidos> diagramar (List<Equipos> equipos, Date fechaInicio) {
        // acá irá el código que genera los partidos con llaves de eliminatorias
    }
}
```

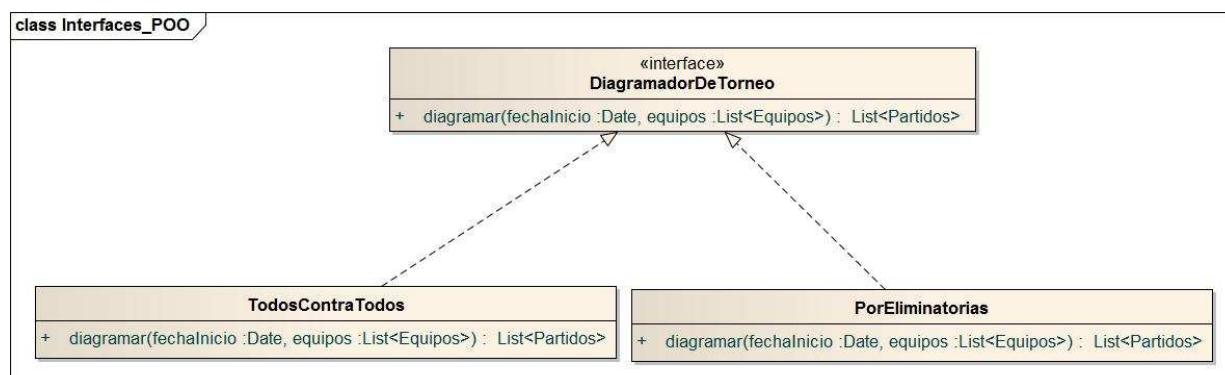


Fig. 3- Ejemplo de interfaces – Diagramación de Torneos

Documentación de Clases y Métodos

Documentar un proyecto es algo fundamental de cara a su futuro mantenimiento. Cuando programamos una clase, debemos generar documentación lo suficientemente detallada sobre ella como para que otros programadores sean capaces de usarla sólo con su interfaz. No debe existir necesidad de leer o estudiar su implementación, lo mismo que nosotros para usar una clase la biblioteca de Java no leemos ni estudiamos su código fuente.

Javadoc es una utilidad de Oracle para la generación de documentación en formato de página web a partir de código fuente Java. Javadoc es el estándar para documentar clases de Java. La mayoría de los IDEs utilizan javadoc para generar de forma automática documentación de clases.

La documentación a ser utilizada por javadoc se escribe en comentarios que comienzan con `/**` (notar el doble `*`) y que terminan con `*/`. A la vez, dentro de estos comentarios se puede escribir código HTML y operadores para que interprete javadoc (generalmente precedidos por `@`).

Tag	Descripción	Uso
<code>@author</code>	Nombre del desarrollador creador de la clase o interfaz	nombre
<code>@deprecated</code>	Indica que el método o clase es antigua y que no se recomienda su uso porque posiblemente desaparecerá en versiones posteriores	descripción
<code>@param</code>	Definición de un parámetro de un método, es requerido para todos los parámetros del método	nombre_parametro descripción
<code>@return</code>	Informa lo que devuelve el método, no se puede usar en constructores o métodos "void"	descripción
<code>@see</code>	Asocia con otro método o clase, brinda más información o elementos relacionados	referencia (<code>#método()</code> ; <code>clase#método();</code> <code>paquete.clase;</code> <code>paquete.clase#método()</code>).
<code>@throws</code>	Excepción lanzada por el método (lo veremos más adelante)	nombre_clase descripción
<code>@version</code>	Versión del método o clase	versión

Tabla 5 – Tags utilizados en Javadoc

Las etiquetas `@author` y `@version` se usan para documentar clases e interfaces. Por tanto no son válidas en cabecera de constructores ni métodos. La etiqueta `@param` se usa para documentar constructores y métodos. La etiqueta `@return` se usa solo en métodos de tipo función.

Dentro de los comentarios se admiten etiquetas HTML, por ejemplo con @see se puede referenciar una página web como link para recomendar su visita de cara a ampliar información.

Un ejemplo de su uso en un archivo de código puede ser:

```
package futsal.equipos;

import java.util.Date;

/**
 * Clase que representa a cualquier Jugador registrado en
 * el Torneo. Es importante considerar que sólo puede pertenecer
 * a un Equipo inscripto.
 *
 * @author joaquinleonelrobles
 * @see futsal.equipos.Equipo
 */
public class Jugador {

    private String nombre;
    private String apellido;
    private Date fechaNacimiento;

    /**
     * Calculamos la edad del jugador en base a su fecha de nacimiento
     *
     * @return Edad
     */
    public int calcularEdad () {
        // TODO recordar implementar este método
        return 0;
    }

    /**
     * Comprobamos si el Jugador puede participar en un torneo en
     * base a la edad mínima pasada por parámetros.
     *
     * @param edadMinima Edad mínima (inclusive) en años
     * @return Verdadero si puede participar
     */
    public boolean puedeParticiparPorSuEdad (int edadMinima) {
        return edadMinima >= this.calcularEdad();
    }
}
```

En el ejemplo anterior puede notarse un comentario que inicia con las letras “TODO”, se trata de una convención utilizada por los desarrolladores para denotar código que aún debe implementarse, del inglés “To Do” o “Pendiente”. Algunos IDE agregan automáticamente estos comentarios y además permiten resaltarlos para recordarnos de las porciones de código que nos faltan escribir.

```

24 public int calcularEdad () {
25     // TODO recordar implementar este metodo
26     return 0;
27 }
```

Fig. 4- Resaltado de comentarios TODO en la IDE Eclipse

La documentación generada por JavaDoc tendrá una apariencia similar a la siguiente captura de pantalla:

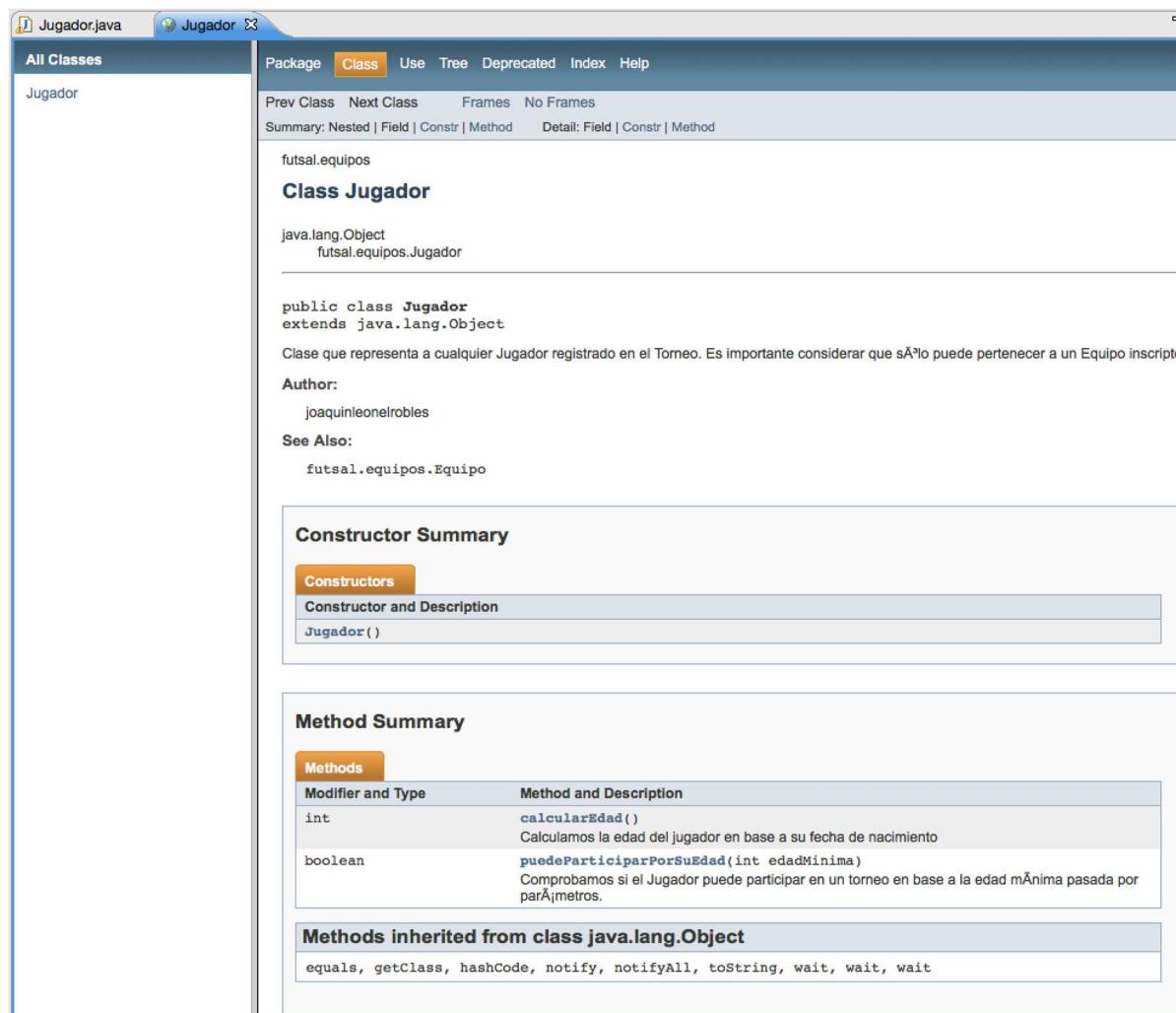


Fig. 5-. Documentación JavaDoc generada automáticamente en base a comentarios

Trabajando con Fechas

A partir de la versión 8 de Java se incorporó un nuevo paquete para el tratamiento de fechas y horas que extiende la funcionalidad antes provista por las clases `java.util.Date` y `java.util.Calendar` para paliar algunos de sus problemas y dificultades de implementación. La documentación oficial del paquete `java.time` se encuentra disponible en <https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html>.

Las clases definidas en este paquete representan los principales conceptos de fecha - hora, incluyendo instantes, fechas, horas, períodos, zonas de tiempo, etc. Están basados en el sistema de calendario ISO, el cual es el calendario mundial de-facto que sigue las reglas del calendario Gregoriano.

Entre las clases disponibles en el paquete encontramos tres de particular importancia:

`java.time.LocalDate` que representa fechas como 11/04/2019

`java.time.LocalTime` que representa horarios como 12:03:23 PM

`java.time.LocalDateTime` que representa fechas y horas como 11/04/2019 a las 12:03:23 PM

Anteriormente la clase `java.util.Date` estaba destinada para almacenar tanto tipos de datos relacionados con fechas (como el 31 de Diciembre de 2018) como aquellos que representaban tanto fechas como hora (como el 31 de Diciembre de 2018 a las 13:54:00), lo cual podía ocasionar confusión y necesidad de aclaraciones adicionales a la hora de conocer la semántica de la variable que queríamos representar, por ejemplo:

```
import java.util.Date;

class Viaje {
    private Date fechaLlegada;
    private Date fechaHoraCreacion;
}
```

En el ejemplo anterior podemos ver una representación parcial de la clase `Viaje`, en la que encontramos dos atributos cuyo tipo de datos seleccionado es `java.util.Date` pero que tienen una semántica diferente: la `fechaLlegada` corresponde a la fecha en la que el viajero llegará a destino sin importar el momento del día, es decir sólo la porción de la fecha correspondiente al día/mes/año, mientras que para la fecha y hora de creación de la entidad efectivamente necesitamos representar el momento exacto, considerando la hora en la que se actualizó el atributo.

Para poder representar sin ambigüedad esta diferencia es que disponemos de LocalDate y LocalDateTime, que en el ejemplo anterior podrían ser usados como:

```
import java.time.LocalDate;  
import java.time.LocalDateTime;  
  
class Viaje {  
    private LocalDate fechaLlegada;  
    private LocalDateTime fechaHoraCreacion;  
}
```

Veremos más adelante que estas clases tienen relación directa con la selección del tipo de datos más adecuado para las columnas correspondientes en una tabla de una base de datos relacional a la hora de implementar un *Esquema de Persistencia*.

LocalDate, LocalTime y LocalDateTime

Antes de ver cada una de estas clases de forma independiente es importante aclarar que todas son agnósticas a la zona horaria, es decir que a la hora de utilizarlas para representar fechas que corresponden a diferentes husos horarios (como GMT-3 para Argentina o GMT-5 para Nueva York) deberemos utilizar otros tipos de datos con mayor precisión, tales como ZonedDateTime que también pertenece al paquete java.time.

LocalDate

LocalDate representa una fecha sin tener en cuenta el tiempo. Haciendo uso del método of(int year, int month, int dayOfMonth), podemos obtener una representación de una fecha determinada:

```
LocalDate date = LocalDate.of(1989, 11, 11);  
System.out.println(date.getYear()); //1989  
System.out.println(date.getMonth()); //NOVEMBER  
System.out.println(date.getDayOfMonth()); //11
```

En el ejemplo anterior también podemos ver en acción algunos métodos auxiliares disponibles para obtener los componentes individuales de nuestra fecha.

También, se puede hacer uso de la enumeración Month para dar mayor legibilidad al código:

```
LocalDate date = LocalDate.of(1989, Month.NOVEMBER, 11);
```

Finalmente, para obtener la representación de la fecha actual según el reloj del sistema se puede usar el método now():

```
LocalDate date = LocalDate.now();
```

LocalTime

De manera similar encontramos la clase LocalTime, la cual representa un tiempo determinado. Haciendo uso del método of() esta clase puede crear un LocalTime teniendo en cuenta: la hora y minuto; hora, minuto y segundo o finalmente hora, minuto, segundo y nanosegundo.

```
LocalTime time = LocalTime.of(5, 30, 45, 35); //05:30:45:35
System.out.println(time.getHour()); //5
System.out.println(time.getMinute()); //30
System.out.println(time.getSecond()); //45
System.out.println(time.getNano()); //35
```

Análogamente a la clase anterior para obtener el LocalTime actual se puede usar el método now():

```
LocalTime time = LocalTime.now();
```

LocalDateTime

LocalDateTime es una clase compuesta que combina las clases anteriormente mencionadas LocalDate y LocalTime en una única representación de fecha y hora.

En el siguiente ejemplo podemos ver la construcción de un LocalDateTime haciendo uso de todos los campos (año, mes, día, hora, minuto, segundo, nanosegundo):

```
LocalDateTime dateTime = LocalDateTime.of(1989, 11, 11, 5, 30, 45, 35); //1989-11-11T05:30:45.0000000035
```

También se puede crear un objeto LocalDateTime basado en dos objetos LocalDate y LocalTime previamente instanciados usando el método of(LocalDate date, LocalTime time):

```
LocalDate date = LocalDate.of(1989, 11, 11);
LocalTime time = LocalTime.of(5, 30, 45, 35);
LocalDateTime dateTime = LocalDateTime.of(date, time);
```

Finalmente podemos obtener el LocalDateTime exacto correspondiente al momento de la ejecución usando el método now(), como se muestra a continuación:

```
LocalDateTime dateTime = LocalDateTime.now();
```

Operaciones con fechas

De forma general la API de java.time tiene una cantidad relativamente grande de métodos para cada una de sus clases. Esta gran cantidad de métodos es fácilmente manejable gracias a un esquema de nombrado uniformemente aplicado a cada una de sus clases, donde encontramos los siguientes prefijos para las signaturas de los métodos:

- of - método estático para fabricación
- parse - método estático para fabricación en base a interpretación (parsing)
- get - devuelve el valor de algo
- is - comprueba que algo sea true
- with - el equivalente inmutable a un método de seteo
- plus - agrega una cantidad a un objeto
- minus - disminuye una cantidad a un objeto
- to - convierte este objeto a algún otro tipo
- at - combina este objeto con otro, como por ejemplo fecha.atTime(hora)

Veamos un ejemplo:

```
LocalDate date = LocalDate.of(2016, Month.JULY, 18);
LocalDate datePlusOneDay = date.plus(1, ChronoUnit.DAYS);
LocalDate dateMinusOneDay = date.minus(1, ChronoUnit.DAYS);
```

Asimismo puede hacerse uso de las siguientes unidades ChronoUnit.YEARS, ChronoUnit.MONTHS. Si en los ejemplos usamos ChronoUnit.HOURS la siguiente excepción será lanzada java.time.temporal.UnsupportedTemporalTypeException: Unsupported unit: Hours.

Formatos

Cuando trabajamos con fechas ocasionalmente se requiere mostrarlas en formato de texto respetando un formato personalizado. Java 8 ofrece la clase java.time.format.DateTimeFormatter la cual provee algunos formatos predeterminados que podemos aplicar como parámetros del método format() para obtener un String:

```
LocalDate date = LocalDate.of(2016, Month.JULY, 25);
String date1 = date.format(DateTimeFormatter.BASIC_ISO_DATE); //20160725
String date2 = date.format(DateTimeFormatter.ISO_DATE); //2016-07-25
```

Es importante destacar que el formato que seleccionemos corresponda con el contenido de nuestra fecha/hora a mostrar, por ejemplo si en el último ejemplo usamos DateTimeFormatter.ISO_DATE_TIME la siguiente excepción será lanzada java.time.temporal.UnsupportedTemporalTypeException: Unsupported field: HourOfDay debido a que intentamos dar formato a una porción de hora no disponible en nuestro LocalDate.

También se puede usar el método `ofPattern(String pattern)` para definir un formato personalizado.

```
LocalDate date = LocalDate.of(2016, Month.JULY, 25);
String date1 = date.format(DateTimeFormatter.ofPattern("yyyy/MM/dd")); //2016/07/25
```

Debajo mencionamos algunas de posibilidades de formatos personalizados:

Symbol	Meaning	Presentation	Examples
G	era	text	AD; Anno
Domini;			A
u	year	year	2004; 04
y	year-of-era	year	2004; 04
D	day-of-year	number	189
M/L	month-of-year	number/text	7; 07; Jul;
July;			J
d	day-of-month	number	10
Q/q	quarter-of-year	number/text	3; 03; Q3; 3rd
quarter			
Y	week-based-year	year	1996; 96
w	week-of-week-based-year	number	27
W	week-of-month	number	4
E	day-of-week	text	Tue; Tuesday;
T			
e/c	localized day-of-week	number/text	2; 02; Tue;
Tuesday;			T
F	week-of-month	number	3
a	am-pm-of-day	text	PM
h	clock-hour-of-am-pm (1-12)	number	12
K	hour-of-am-pm (0-11)	number	0
k	clock-hour-of-am-pm (1-24)	number	0
H	hour-of-day (0-23)	number	0
m	minute-of-hour	number	30
s	second-of-minute	number	55
S	fraction-of-second	fraction	978
A	milli-of-day	number	1234
n	nano-of-second	number	987654321
N	nano-of-day	number	1234000000

Por último para construir nuestro formato personalizados podemos hacer uso de la clase `java.time.format.DateTimeFormatterBuilder`:

```

LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 25, 15, 30);
OffsetDateTime offsetDateTime = dateTime.atOffset(ZoneOffset.ofHours(-5));
DateTimeFormatter formatter = new DateTimeFormatterBuilder()
    .appendText(ChronoField.DAY_OF_MONTH)
    .appendLiteral(" ")
    .appendText(ChronoField.MONTH_OF_YEAR)
    .appendLiteral(" ")
    .appendText(ChronoField.YEAR)
    .appendOffsetId()
    .toFormatter();
String dateTime1 = offsetDateTime.format(formatter); //25 July 2016-05:00

```

Es posible encontrar más información sobre las opciones de formato en la documentación oficial: <https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html>.

Operaciones matemáticas con BigDecimal

A la hora de desarrollar aplicaciones donde necesitemos representar valores monetarios generalmente nos encontramos con situaciones donde las operaciones con decimales pueden dar resultados incorrectos, específicamente en los decimales con resultados quedando centésimas arriba o abajo del resultado esperado.

Las computadoras no cuentan como nosotros, las computadoras "cuentan en binario", nosotros, contamos en decimal. Cuando usamos una variable float o double, estamos dejando al microprocesador de la computadora el trabajo de efectuar los cálculos con su coprocesador matemático pero, como las computadoras "piensan" en binario, cometan "errores" de precisión diferentes a los nuestros. Por ejemplo, nosotros los humanos, tenemos un problema para representar a la tercera parte de "1" (1/3), y la escribimos: 0.333333333333333... (hasta el infinito), en nuestro sistema decimal (base 10), no hay modo de representar a un tercio de un entero. Si por otro lado usamos un sistema base "9", entonces la representación de la tercera parte de un entero es simplemente: "0.3" con la correspondiente precisión que lleva.

En binario (base 2), no se puede representar a 1/10 (la décima parte de 1) ni a 1/100 (la centésima parte de 1) y por eso, cuando nosotros escribimos "0.01" la computadora lo entiende como 110011001100110011001100110011... (hasta el infinito) y efectúa el cálculo incorrectamente (desde nuestro punto de vista "decimal").

Si trabajamos con sistemas financieros, los pequeños errores de cálculo acumulados con varios centavos pueden resultar en reportes cuyos valores difieren de las operaciones que normalmente realizamos los humanos. Para resolver este problema, en Java se incluye la clase BigDecimal del paquete java.math que realiza los cálculos del mismo modo que los humanos (en base 10) y así evitamos el problema.

Para poder hacer frente a este problema la clase BigDecimal del paquete java.math nos permite representar un número decimal con las siguientes características

- Operaciones decimales: Las operaciones sobre este tipo se realizan internamente en base 10, esto implica una pérdida de rendimiento pero a cambio los errores de aritmética binaria se previenen por completo.
- Precisión arbitraria: No se impone un límite en la cantidad de decimales, la cantidad de decimales a usar esta limitada únicamente por la memoria del sistema
- Inmutable: Una vez creado un objeto BigDecimal el valor de este no puede ser modificado, cada operación que realicemos devuelve una nueva instancia con el resultado.

Operaciones

Las operaciones entre instancias de la clase BigDecimal, a diferencia de lo esperado, no utilizan los operadores aritméticos como + - * sino que implementan estas operaciones mediante métodos:

add suma un valor

subtract resta un valor

multiply multiplica por un valor

divide divide por un valor

En todos los casos el primer parámetro será la instancia con la cual queremos aplicar la operación. Es muy importante destacar que como las operaciones con BigDecimal son **inmutables** la invocación del método no producirá cambios en el objeto que la invoca, sino que devolverá una nueva instancia de BigDecimal con el resultado de la operación.

De esta forma si quisiéramos incrementar el valor de una variable tendríamos que recurrir a una asignación de la forma:

```
BigDecimal incremento = new BigDecimal(10);
BigDecimal subtotal = new BigDecimal(5);
subtotal = subtotal.add(incremento);
```

Olvidar la inmutabilidad de BigDecimal y la necesidad de asignación es el error más común a la hora de trabajar con este tipo de datos.

Para un listado completo de los métodos disponibles en la clase BigDecimal podemos acudir a su JavaDoc disponible en: <https://docs.oracle.com/javase/10/docs/api/java/math/BigDecimal.html>

Precisión y escala

Como comentamos anteriormente `BigDecimal` es un tipo de datos de precisión fija, es decir que tenemos control absoluto de la cantidad de dígitos antes y después de la coma. En este contexto definimos a la **precisión** como la cantidad total de dígitos (enteros y decimales) mientras que la **escala** es la cantidad de dígitos decimales.

Veamos algunos ejemplos:

Valor	Precisión	Escala
$9999/10000 = 0.9999$	4	4
$999/1000 = 0.999$	3	3
$999/10 = 99.9$	3	1

Más adelante veremos que esta precisión y escala tienen directa relación con el tamaño asociado a las columnas correspondientes a una tabla de una base de datos relacional al implementar un Esquema de Persistencia.

Redondeo

La clase `BigDecimal` nos da control total sobre el comportamiento del redondeo. Si no especificamos de forma explícita un modo de redondeo y el resultado exacto de la operación no puede ser representado recibiremos una excepción. Para poder especificar el modo de redondeo y evitar estos problemas `BigDecimal` nos ofrece 8 tipos diferentes de formas de redondear mediante la enumeración `RoundingMode`:

Resultado de cada modo al redondear la entrada a un sólo dígito								
Input	UP	DOWN	CEILING	FLOOR	HALF_UP	HALF_DOWN	HALF_EVEN	
5.5	6	5	6	5	6	5	6	
2.5	3	2	3	2	3	2	2	
1.6	2	1	2	1	2	2	2	
1.1	2	1	2	1	1	1	1	
1.0	1	1	1	1	1	1	1	
-1.0	-1	-1	-1	-1	-1	-1	-1	

-1.1	-2	-1	-1	-2	-1	-1	-1
-1.6	-2	-1	-1	-2	-2	-2	-2
-2.5	-3	-2	-2	-3	-3	-2	-2
-5.5	-6	-5	-5	-6	-6	-5	-6

Para todos los casos anteriores el modo ROUND_UNNECESSARY arrojará una excepción del tipo ArithmeticException excepto para 1.0 y -1.0, casos en los que devolverá exactamente los mismos valores sin modificar.

Veamos un ejemplo en aplicación:

```
BigDecimal bigDecimal = new BigDecimal("23323323.3533");
System.out.println(bigDecimal);
System.out.println("CEILING: "+bigDecimal.setScale(2,RoundingMode.CEILING));
System.out.println("DOWN: "+bigDecimal.setScale(2,RoundingMode.DOWN));
System.out.println("FLOOR: "+bigDecimal.setScale(2,RoundingMode.FLOOR));
```

Tendrá la siguiente salida:

23323323.3533	23323323.36
CEILING:	23323323.35
DOWN:	
FLOOR: 23323323.35	

Colecciones de tamaño variable

Anteriormente hemos mencionado los arrays de datos, los cuales son inicializados con un tamaño fijo que no podremos modificar a lo largo del desarrollo del programa. En algunos casos es útil poder tener colecciones de tamaño variable, a las cuales podamos aplicar operaciones de agregado o remoción de elementos sin tener que preocuparnos si el espacio en memoria está o no reservado. Java incorpora en su biblioteca de clases algunas alternativas muy útiles para trabajar con este tipo de colecciones, a continuación, veremos algunas de ellas.

Listas

Las listas son estructuras de datos que permiten tener cierta flexibilidad en su manejo, pueden crecer o acortarse según se lo requiera. Existen varias formas de implementar una lista en Java: en este caso se presenta un ejemplo en código utilizando punteros mediante la referencia a objetos. Una lista es una secuencia de elementos dispuesto en un cierto orden, en la que cada elemento tiene como mucho un predecesor y un sucesor. El número de elementos de la lista no suele estar fijado, ni suele estar limitado por anticipado.

Representaremos la estructura de datos de forma gráfica con cajas y flechas. Las cajas son los elementos y las flechas simbolizan el orden de los elementos.



Fig. 6- Representación de nodos en una lista ordenada

La estructura de datos deberá permitirnos determinar cuál es el primer elemento y el último de la estructura, cuál es su predecesor y su sucesor (si existen de cualquier elemento dado). Cada uno de los elementos de información suele denominarse **nodo**.

La interfaz java.util.List

Esta interfaz normalmente acepta elementos repetidos o duplicados y al igual que los arrays es lo que se llama “basada en 0”, esto quiere decir que el primer elemento no es el que está en la posición “1”, sino en la posición “0”.

Esta interfaz proporciona iterador especial que nos permite recorrer los distintos elementos de la lista. Este iterador permite además de los métodos definidos por cualquier iterador (estos métodos son `hasNext`, `next` y `remove`) métodos para inserción de elementos y reemplazo, acceso bidireccional para recorrer la lista y un método proporcionado para obtener un iterador empezando en una posición específica de la lista.

Debido a la gran variedad y tipo de listas que puede haber con distintas características como permitir que contengan o no elementos nulos, o que tengan restricciones en los tipos de sus elementos, hay una gran cantidad de clases que implementan esta interfaz.

Es posible restringir de qué clase serán instancia los objetos que pertenezcan a la lista utilizando los símbolos `<` y `>` en la definición de la variable, como, por ejemplo:

```
List<Jugador> jugadores;
```

De esta forma la lista jugadores sólo podrá estar compuesta por elementos que sean instancias de la clase Jugador.

A continuación, veremos las siguientes clases que implementan esta interfaz:

- ArrayList
- Stack

La Clase java.util.ArrayList

ArrayList, como su nombre lo indica, basa la implementación de la lista en un array. Eso sí, un array dinámico en tamaño (es decir, de tamaño variable), pudiendo agrandarse o disminuirse el número de elementos. Implementa todos los métodos de la interfaz List y permite incluir elementos null.

Un beneficio de usar esta implementación de List es que las operaciones de acceso a elementos, capacidad y saber si es vacía o no se realizan de forma eficiente y rápida. Todo arraylist tiene una propiedad de capacidad, aunque cuando se añade un elemento esta capacidad puede incrementarse. Java amplía automáticamente la capacidad de un arraylist cuando sea necesario. A través del código podemos incrementar la capacidad del arraylist antes de que este llegue a llenarse, usando el método ensureCapacity.

A continuación, veremos un ejemplo en el cual calculamos la edad promedio de un grupo de Jugadores.

```
package futsal.equipos;

public class Jugador {

    private String nombre;
    private String apellido;
    private int edad;

    public Jugador (nombre, apellido, edad) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
    }
}
```

En el cuerpo de nuestro programa creamos algunos jugadores con sus edades, los incluimos en un ArrayList y luego lo recorremos para obtener el promedio de edades.

```
// creamos los jugadores con sus edades
Jugador j1 = new Jugador("Miguel", "Houellebecq", 21);
Jugador j2 = new Jugador("Pablo", "Auster", 19);
Jugador j3 = new Jugador("Aldo", "Huxley", 23);
Jugador j4 = new Jugador("Alejandro", "Baricco", 21);

// inicializamos la variable acumuladora de promedios
float promedio = 0;

// inicializamos la lista de jugadores
```

```

// se especifica el tipo de datos de los elementos de la lista
// entre cocodrilos (< y >) para evitar insertar otro tipo de objetos
List<Jugador> jugadores = new ArrayList<Jugador>();

// agregamos un elemento a la lista con el método add()
jugadores.add (j1);
jugadores.add (j2);
jugadores.add (j3);
jugadores.add (j4);

// obtenemos un objeto Iterador que nos permita recorrer la lista
Iterator<Jugador> iter = jugadores.iterator();

// mientras exista un elemento siguiente por recorrer
while (iter.hasNext()) {
    // obtenemos el Jugador siendo recorrido
    Jugador j = iter.next();

    // acumulamos su edad promedio sobre el total de jugadores
    // que obtenemos con el método size() de la lista
    promedio += j.getEdad() / jugadores.size();
}

// mostramos el resultado en pantalla
System.out.println("Promedio: " + promedio);

```

A continuación, se listan algunos de los métodos más relevantes de la clase ArrayList:

Retorno	Método y descripción
boolean	add(E e) Agrega el elemento al final de la lista
void	add(int index, E element) Agrega el elemento en la posición especificada
boolean	addAll(Collection<? extends E> c) Agrega todos los elementos de otra colección al final de esta lista
boolean	addAll(int index, Collection<? extends E> c) Agrega todos los elementos de otra colección en la posición especificada
void	clear() Elimina todos los elementos de la lista
boolean	contains(Object o) Devuelve true si el objeto o existe en la lista
void	ensureCapacity(int minCapacity) Incrementa la capacidad del array subyacente para asegurar la capacidad especificada
E	get(int index) Devuelve el elemento de la posición solicitada
int	indexOf(Object o) Devuelve la posición del objeto solicitado
boolean	isEmpty()

	Devuelve true si la lista no contiene elementos
Iterator<E>	iterator()
	Devuelve un objeto iterador para recorrer la lista
E	remove(int index)
	Elimina el elemento de la posición especificada
boolean	remove(Object o)
	Elimina la primera ocurrencia del objeto en la lista
E	set(int index, E element)
	Reemplaza el elemento en la posición especificada
int	size()
	Devuelve el tamaño de la lista

Tabla 6. Extracto del JavaDoc para la clase ArrayList

La clase java.util.Stack

Esta clase es una implementación de la estructura de datos que vimos en el módulo de Técnicas de Programación denominada Pila, una estructura de tipo LIFO (Last In First Out, último en entrar primero en salir). Provee las operaciones de push (colocar) y pop (extraer) así como otros métodos accesorios como el size(), peek (consulta el primer elemento de la cima de la pila), empty (que comprueba si la pila está vacía) y search (que busca un determinado elemento dentro de la pila y devuelve su posición dentro de ella).

Veamos un ejemplo apilando Jugadores:

```
// creamos los jugadores con sus edades
Jugador j1 = new Jugador("Miguel", "Houellebecq", 21);
Jugador j2 = new Jugador("Pablo", "Auster", 19);
Jugador j3 = new Jugador("Aldo", "Huxley", 23);
Jugador j4 = new Jugador("Alejandro", "Baricco", 21);

// inicializamos la pila de jugadores
Stack<Jugador> pila = new Stack<Jugador>();

// agregamos un elemento a la lista con el método add()
pila.push (j1);
pila.push (j2);
pila.push (j3);
pila.push (j4);

// mientras existan elementos por extraer
while (!pila.empty()) {
    // mostramos el nombre del jugador en pantalla
    System.out.println (pila.pop().getNombreCompleto());
}
```

La salida de nuestro programa será la siguiente, respetando el orden de extracción Último en Entrar Primero en Salir:

Alejandro Baricco
 Aldo Huxley
 Pablo Auster
 Miguel Houllebecq

A continuación, presentamos un extracto del JavaDoc de la clase Stack con los métodos más relevantes:

Retorno	Método y Descripción
boolean	empty() Devuelve true si la pila está vacía
E	peek() Consultamos el primer elemento de la pila sin extraerlo
E	pop() Extraemos el primer elemento de la pila
E	push(E item) Colocamos un elemento en el tope de la pila
int	search(Object o) Devuelve la posición del elemento en la pila (basado en 1)

Tabla 7. Extracto del JavaDoc para la clase Stack

Excepciones

Como mencionamos anteriormente, en Java existe un tipo especial de errores denominado Excepción. El mejor momento para detectar los errores es durante la compilación. Sin embargo, prácticamente sólo los errores de sintaxis son detectados durante este periodo. El resto de los problemas surgen durante la ejecución de los programas.

Una “*Exception*” es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas *excepciones* son fatales y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras, como por ejemplo no encontrar un archivo en el que hay que leer o escribir algo, pueden ser recuperables. En este caso el programa debe dar al usuario la oportunidad de corregir el error (indicando una nueva localización del fichero no encontrado).

Para poder trabajar con excepciones y capturarlas para poder desarrollar un programa que pueda ser tolerante a errores se utilizan los bloques try-catch-finally que enunciamos anteriormente cuando hablamos de Estructuras de Control. Ahora veremos con mayor profundidad cómo se utilizan.

Los errores se representan mediante dos tipos de clases derivadas de la clase *Throwable*: *Error* y *Exception*, sobre estas últimas es que trabajaremos en nuestros programas.

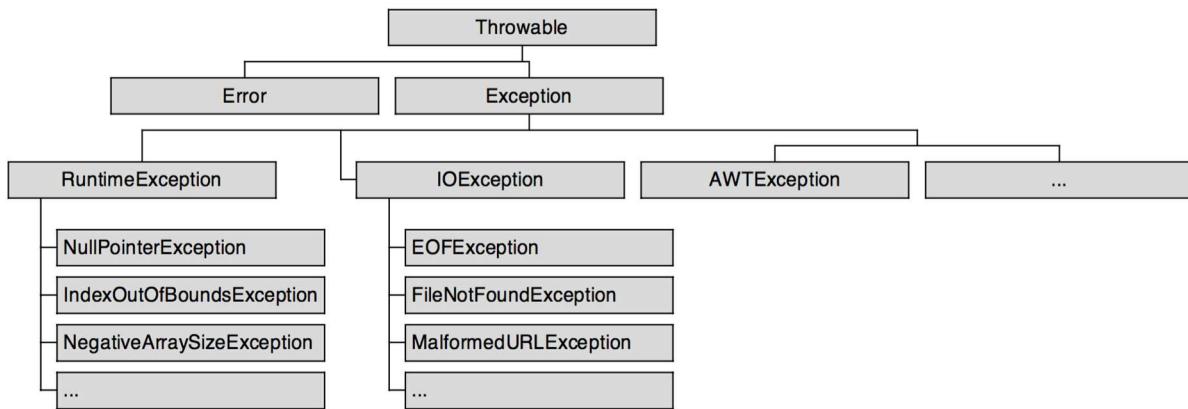


Fig. 7- Jerarquía de Excepciones en el lenguaje Java

Las clases derivadas de *Exception* pueden pertenecer a distintos packages de *Java*. Algunas pertenecen a *java.lang* (*Throwable*, *Exception*, *RuntimeException*, ...); otras a *java.io* (*EOFException*, *FileNotFoundException*, ...) o a otros packages. Por heredar de *Throwable* todos los tipos de excepciones pueden usar los siguientes métodos:

1. `String getMessage()`
Extrae el mensaje asociado con la excepción
2. `String toString()`
Devuelve un String que describe la excepción
3. `void printStackTrace()`
Indica el método donde se lanzó la excepción

Además disponemos de los siguientes constructores propios de la clase *java.lang.Exception* que podemos invocar en nuestra clase:

1. `Exception()`
2. `Exception(String message)`

El primero determina un mensaje nulo para la causa de la excepción mientras que el segundo nos permite especificar el mensaje en la invocación al constructor de la clase padre.

En base a lo anterior, podemos declarar nuestras propias excepciones a la hora de desarrollar nuestros programas para manejar los errores propios de la funcionalidad que implementemos; por ejemplo, para el caso de un Encuentro entre dos Equipos, a la hora de intentar obtener el equipo ganador podríamos lanzar una excepción en el caso de que el partido aún no se haya jugado. Para crear nuestra propia excepción sólo tenemos que hacer lo siguiente:

```

public class PartidoNoJugadoException extends Exception
{
    // acá podríamos declarar y redefinir los métodos enunciados más arriba
    // por ahora sólo nos ocuparemos del mensaje de error

    public String getMessage ()
    {
        return "El partido aún no se ha jugado";
    }
}
  
```

```

    }
}
```

De la misma forma podemos especificar el mensaje de la causa declarando el constructor de nuestra clase invocando al constructor de la clase Exception:

```

public class PartidoNoJugadoException extends Exception
{
    public PartidoNoJugadoException()
    {
        super("El partido aún no se ha jugado");
    }
}
```

Es importante notar que es una buena práctica agregar como sufijo al nombre del método la palabra Exception para saber de antemano la naturaleza de esta clase.

En la implementación del método obtenerEquipoGanador() deberíamos agregar la cláusula throws para indicar que el método puede arrojar una excepción de la que explícitamente deberemos hacernos cargo:

```

public Equipo obtenerEquipoGanador() throws PartidoNoJugadoException
{
    // comprobamos si el partido ya se ha jugado
    if (jugado)
    {
        // comparamos la cantidad de goles
        if (golesEquipo1 > golesEquipo2)
            return equipo1; //Cuando el bloque if tiene una única sentencia se
                           //puede evitar el uso de {}
        else
        {
            if (golesEquipo1 < golesEquipo2)
                return equipo2;
            else
                return null;// hubo un empate por lo que devolvemos nulo
        }
    }
    // el partido aún no se ha jugado por lo que lanzamos una excepcion
    else
    {
        throw new PartidoNoJugadoException();
    }
}
```

Por otro lado, también podríamos redefinir el constructor de la clase de excepción y agregar parámetros propios del dominio de nuestro problema. En este caso podríamos, por ejemplo, agregar cuál fue el partido del cual intentamos obtener el resultado.

Durante la ejecución de nuestro programa podríamos tratar de obtener el equipo ganador de la siguiente forma:

```

Encuentro encuentro = new Encuentro();
encuentro.setEquipo1(losQuirquinchosVerdes);
encuentro.setEquipo2(losPadecientesFC);

// como la llamada a nuestro método puede arrojar una excepción debemos
// encerrarla en un bloque try-catch
try
{
    Equipo ganador = encuentro.obtenerEquipoGanador();
    System.out.println("El equipo ganador fue: " + ganador);
}
catch (PartidoNoJugadoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque: " +
ex.getMessage());
}

```

En el ejemplo anterior podemos notar que al capturar la excepción con el bloque catch() además obtenemos una instancia, un objeto de la clase PartidoNoJugadoException, lo que nos permite llamar al método getMessage() para conocer la causa del problema.

La salida del ejemplo anterior sería:

No se puede obtener el equipo ganador porque: El Partido aún no se ha jugado

Ahora que tenemos nuestras excepciones definidas y creadas es necesario arrojarlas en el momento en que se producen durante la ejecución de nuestro programa, para ello disponemos de la palabra reservada throw cuya sintaxis es la siguiente:

```
// alguna condición que genera nuestra excepción
throw objetoDeExcepcion;
```

A los fines de abreviar el código es frecuente ver la creación de los objetos instancia de la clase de excepción en la misma línea en que son arrojados; como, por ejemplo:

```
// alguna condición que genera nuestra excepción
throw new PartidoNoJugadoException();
```

Es importante destacar la diferencia con la palabra reservada throws que informa que un método en particular puede arrojar una excepción de un tipo determinado y deberá ser manejada por el método que lo invoque.

Por otro lado, al reformular el ejemplo agregando el siguiente bloque antes del try-catch la ejecución tomaría el camino feliz mostrándonos al flamante ganador del encuentro: Los Padecientes FC.

```
encuentro.setJugado (true);
encuentro.setGolesEquipo1 (1);
encuentro.setGolesEquipo2 (9); // <-- goleada!
```

Jerarquía de Excepciones

Como vimos anteriormente todas las excepciones heredan de la clase `java.lang.Exception`, pero demás podemos heredar de otras excepciones que finalmente hereden ésta clase. De esta forma podemos diseñar una jerarquía de excepciones para ir del caso más general al más particular en la definición de nuestros errores, veamos un ejemplo expandiendo la jerarquía de la excepción `PartidoNoJugadoException`:

```
public class PartidoNoJugadoException extends ResultadoDePartidoException
{
    // el código de nuestra excepción va aquí
}

public class PartidoCanceladoException extends ResultadoDePartidoException
{
    // el código de nuestra excepción va aquí
}

public class ResultadoDePartidoException extends Exception
{}
```

Entonces ahora tenemos dos excepciones para tratar los posibles errores a la hora de intentar obtener el resultado de un partido en base su estado. Además, a la hora de capturarlas podemos hacer uso de múltiples bloques `catch`, de la siguiente forma:

```
try
{
    Equipo ganador = encuentro.obtenerEquipoGanador();
    System.out.println("El equipo ganador fue: " + ganador);
}
catch (PartidoNoJugadoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque el partido aún no se ha jugado");
}
catch (PartidoCanceladoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque el partido fue cancelado");
}
```

O utilizando la propiedad del polimorfismo propia del lenguaje de programación orientado a objetos podemos capturar todas las excepciones que heredan de `ResultadoDePartidoException` con sólo un bloque `catch`, usando:

```
try
{
    Equipo ganador = encuentro.obtenerEquipoGanador();
    System.out.println("El equipo ganador fue: " + ganador);
}
```

```
catch (ResultadoDePartidoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque" + ex.getMessage());
}
```

Y usando una combinación de ambas estrategias podemos tomar acciones para algunas excepciones en particular y, en los demás casos, realizar una operación para todos los demás casos como, por ejemplo:

```
try
{
    Equipo ganador = encuentro.obtenerEquipoGanador();
    System.out.println("El equipo ganador fue: " + ganador);
}

catch (PartidoNoJugadoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque el partido aún no se ha jugado");
}

catch (ResultadoDePartidoException ex)
{
    System.out.println("No se puede obtener el equipo ganador porque" + ex.getMessage());
}
```

Es importante destacar que cada bloque catch() es evaluado en orden y sólo se ejecutará el primero que corresponda a la excepción arrojada, por lo cual deberemos declararlos en orden de las excepciones más particulares a las más generales. Si en el ejemplo anterior hubiésemos capturado primero ResultadoDePartidoException antes de PartidoNoJugadoException, el último bloque catch nunca sería evaluado ya que todos los casos serían capturados con el primer bloque más general.

La API de Java

Como el lenguaje Java es un lenguaje orientado a objetos, la API de Java provee de un conjunto de clases utilitarias para efectuar toda clase de tareas necesarias dentro de un programa. Se trata de un conjunto de clases de propósito general ya desarrolladas y probadas, que nos permiten ahorrar tiempo a la hora de programar nuestras aplicaciones. Algunas de ellas ya las hemos mencionado y hasta utilizado en ejemplos anterior, como ArrayList, Stack y Exception. La API es parte integral del lenguaje de programación y está implementada para cada máquina virtual independientemente de la plataforma donde ejecutemos nuestro código, por lo cual podemos confiar en su portabilidad.

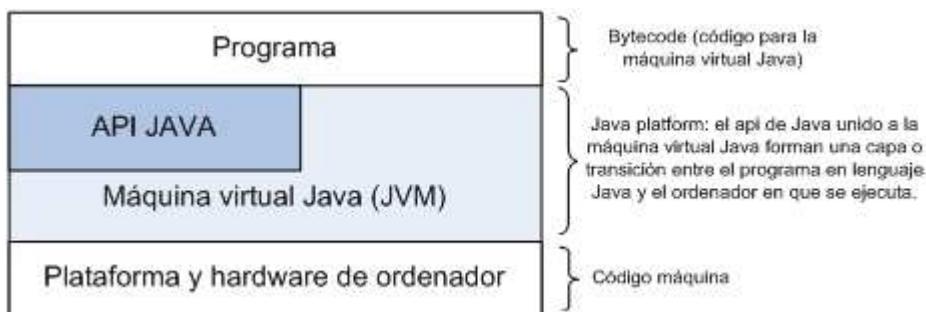


Fig. 8- La API de Java en contexto

La API Java está organizada en paquetes lógicos, donde cada paquete contiene un conjunto de clases relacionadas semánticamente. A continuación, se muestra un esquema que describe en términos generales la estructura de la API.

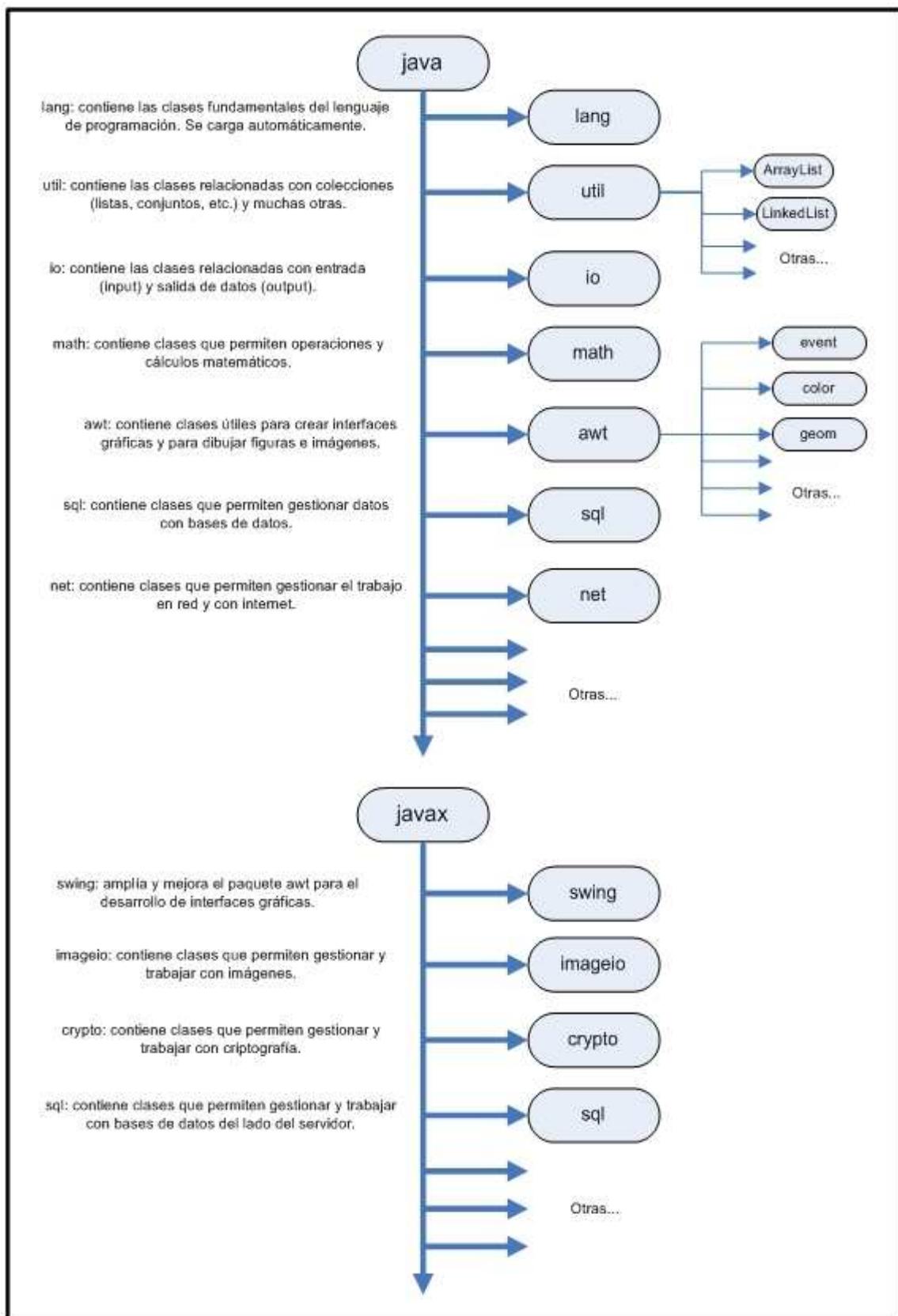


Fig. 9- Estructura de paquetes general para la API de Java 8.

Las librerías podemos decir que se organizan en ramas como si fueran las ramas de un árbol. Vamos a fijarnos en dos grandes ramas: la rama “java” y la rama “javax”. La rama java parte de los orígenes de Java y contiene clases más bien generales, mientras que la rama javax es más moderna y tiene paquetes con fines mucho más específicos.

Encontrar un listado de librerías o clases más usadas es una tarea casi imposible. Cada programador, dependiendo de su actividad, utiliza ciertas librerías que posiblemente no usen otros programadores. Los programadores centrados en programación de escritorio usarán clases diferentes a las que usan programadores web o de gestión de bases de datos.

La especificación de la API de forma completa (en JavaDoc por supuesto) está disponible en <http://www.oracle.com/technetwork/java/api-141528.html> y es propia de cada versión del lenguaje.

Java Web

Introducción a Java Web

Spring Framework es una plataforma que nos proporciona soporte para desarrollar aplicaciones Java. Spring maneja toda la infraestructura y así nos podemos concentrar en la funcionalidad específica de nuestra aplicación. En otras palabras, Spring es el “pegamento” que une todos los componentes de la aplicación, maneja su ciclo de vida y la interacción entre ellos.

Spring no solo se puede usar para crear aplicaciones web, se podría usar para cualquier aplicación java, aunque su uso habitual sea en entornos web nada te impide utilizarlo para cualquier tipo de aplicación.

Nació en una época en la que las tecnologías estándar Java EE y la visión "oficial" de lo que debía ser una aplicación Java Empresarial tenían todavía muchas aristas por pulir. Los servidores de aplicaciones eran muy grandes y consumían muchísimos recursos y las aplicaciones web en Java eran pesadas, inflexibles; y era demasiado complejo trabajar con ellas. En ese contexto, Spring hizo popular ideas ya existentes como la inyección de dependencias o el uso objetos de negocio que permitían un desarrollo más sencillo y rápido con aplicaciones más livianas. De esta forma Spring pasó de ser un framework inicialmente diseñado para la capa de negocio a un completo stack de tecnologías para todas las capas de la aplicación.

Spring puede gestionar el ciclo de vida de nuestros objetos. Los objetos gestionados por el framework se denominan genéricamente *beans de Spring*. Esto significa que a los objetos no los instanciamos manualmente sino que lo hace el contenedor web cuando son necesarios. Spring extiende esta idea permitiéndonos gestionar el ciclo de vida de cualquier objeto y, para ello, sólo tendremos que *anotarlo*.

Spring ofrece una serie de anotaciones estándar para los objetos de nuestra aplicación: por ejemplo, `@Service` indica que la clase es un bean de la capa de negocio, mientras que `@Repository` indica que es un objeto de capa de acceso a datos. Si simplemente queremos especificar que algo es un bean sin decir de qué tipo es podemos usar la anotación `@Component`.

Spring Boot

Spring Boot es el proyecto más reciente de Spring que nos ayuda a iniciar nuestro proyecto utilizando los diferentes proyectos de spring de una manera rápida y evitar el exceso de configuración. Toda esta magia se debe a la configuración por defecto que trae dentro y que puede ser configurada mediante propiedades. Spring Boot nos permite poner foco en agregar valor y mejorar la experiencia del desarrollador.

Spring Boot también provee lo siguiente:

- **Convención sobre configuración**

En lugar de estar escribiendo la configuración necesaria y validar si es correcta, Spring Boot provee las configuraciones necesarias bajo diferentes escenarios. De esta manera evitamos la tarea repetitiva de estar agregando las configuraciones.

- **Gestión de dependencias**

Spring Boot provee un análisis de las dependencias que los proyectos alrededor de spring utilizan de manera que nosotros solo tenemos que indicar que dependencia necesitamos sin necesidad de indicar la versión.

- **Auto-configuration**

Como se mencionó en un inicio, spring y sus diferentes proyectos necesitan ser configurados para que funcionen de manera adecuada y podamos continuar agregando valor a nuestra aplicación. Pero la realidad es que muchas veces invertimos mucho tiempo en esas configuraciones. Spring Boot es lo suficientemente inteligente para poder activar las configuraciones necesarias si cumple con ciertas condiciones.

- **Starters**

Spring Boot provee dependencias que traen consigo las dependencias necesarias para empezar nuestro proyecto y ahorrarnos el trabajo de ir a buscar nuestra plantilla con dependencias a otro lado.

- **Servidor Incorporado**

Spring Boot da soporte para [Tomcat](#), [Jetty](#) y [Undertow](#) ya incorporados por lo que la ejecución y despliegue son mucho más rápidos.

Creación de un proyecto Maven

Una de las herramientas más útiles a la hora de utilizar librerías de terceros es Maven. Maven se utiliza en la gestión y construcción de software. Posee la capacidad de realizar ciertas tareas claramente definidas, como la compilación del código y su empaquetado. Es decir, hace posible la creación de software con dependencias incluidas dentro de la estructura del JAR. Es necesario definir todas las dependencias del proyecto (librerías externas utilizadas) en un fichero propio de todo proyecto Maven, el POM (Project Object Model). Este es un archivo en formato XML que contiene todo lo necesario para que a la hora de generar el fichero ejecutable de nuestra aplicación este contenga todo lo que necesita para su ejecución en su interior.

Sin embargo, la característica más importante de Maven es su capacidad de trabajar en red. Cuando definimos las dependencias de Maven, este sistema se encargará de ubicar las librerías que deseamos utilizar en Maven Central, el cual es un repositorio que contiene cientos de librerías constantemente actualizadas por sus creadores. Maven permite incluso buscar versiones más recientes o más antiguas de un código dado y agregarlas a nuestro proyecto. Todo se hará de forma automática sin que el usuario tenga que hacer nada más que definir las dependencias.

La documentación oficial de Maven se encuentra disponible en su web <http://maven.apache.org/>.

Para crear un nuevo proyecto Maven con Spring Tool Suite debemos seleccionar la opción Nuevo Proyecto y luego seleccionar Proyecto de Maven en la lista de opciones:

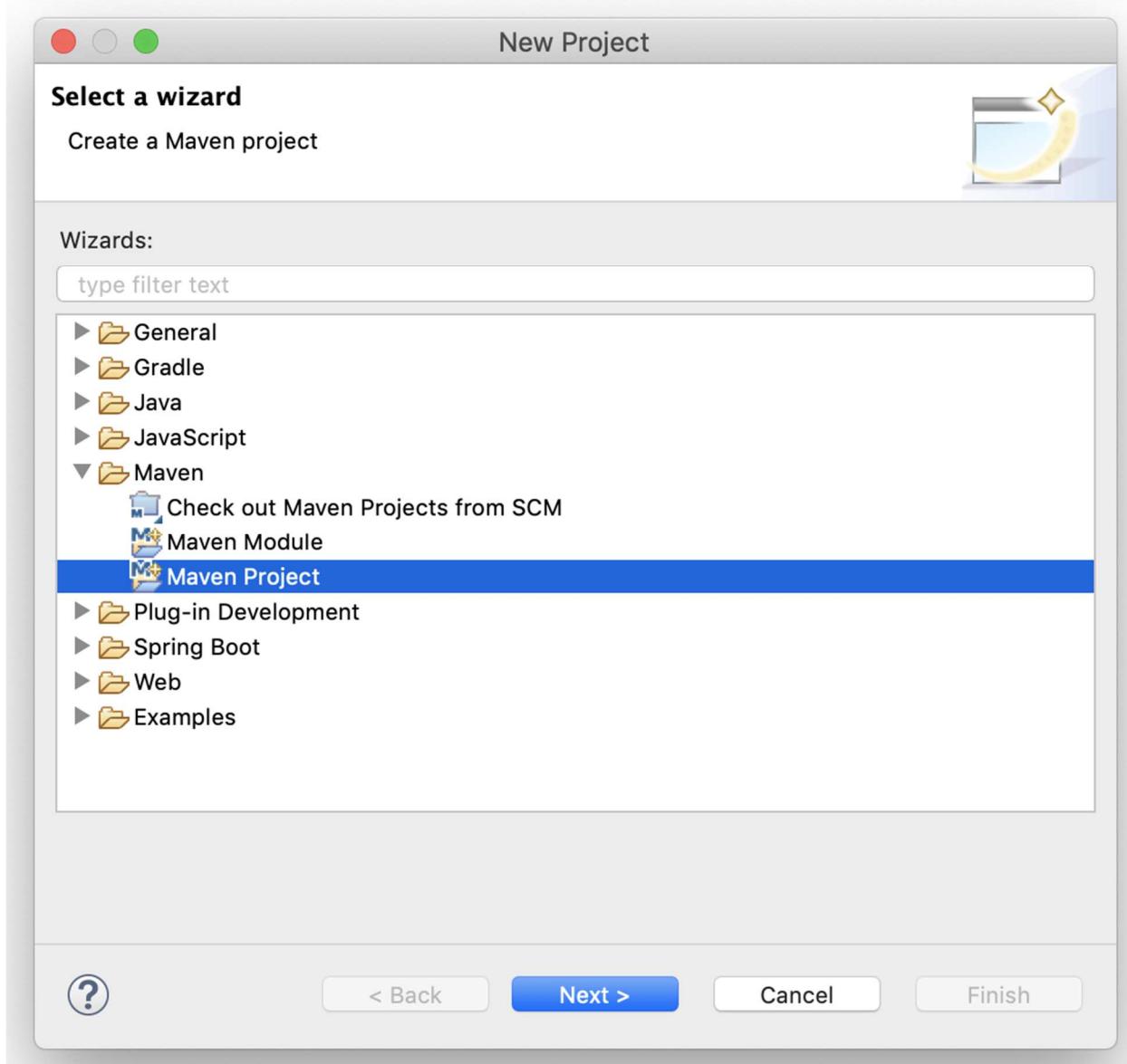


Fig. 10- Creación de un nuevo proyecto Maven con Spring Tool

A continuación podremos elegir la ubicación del proyecto en disco:

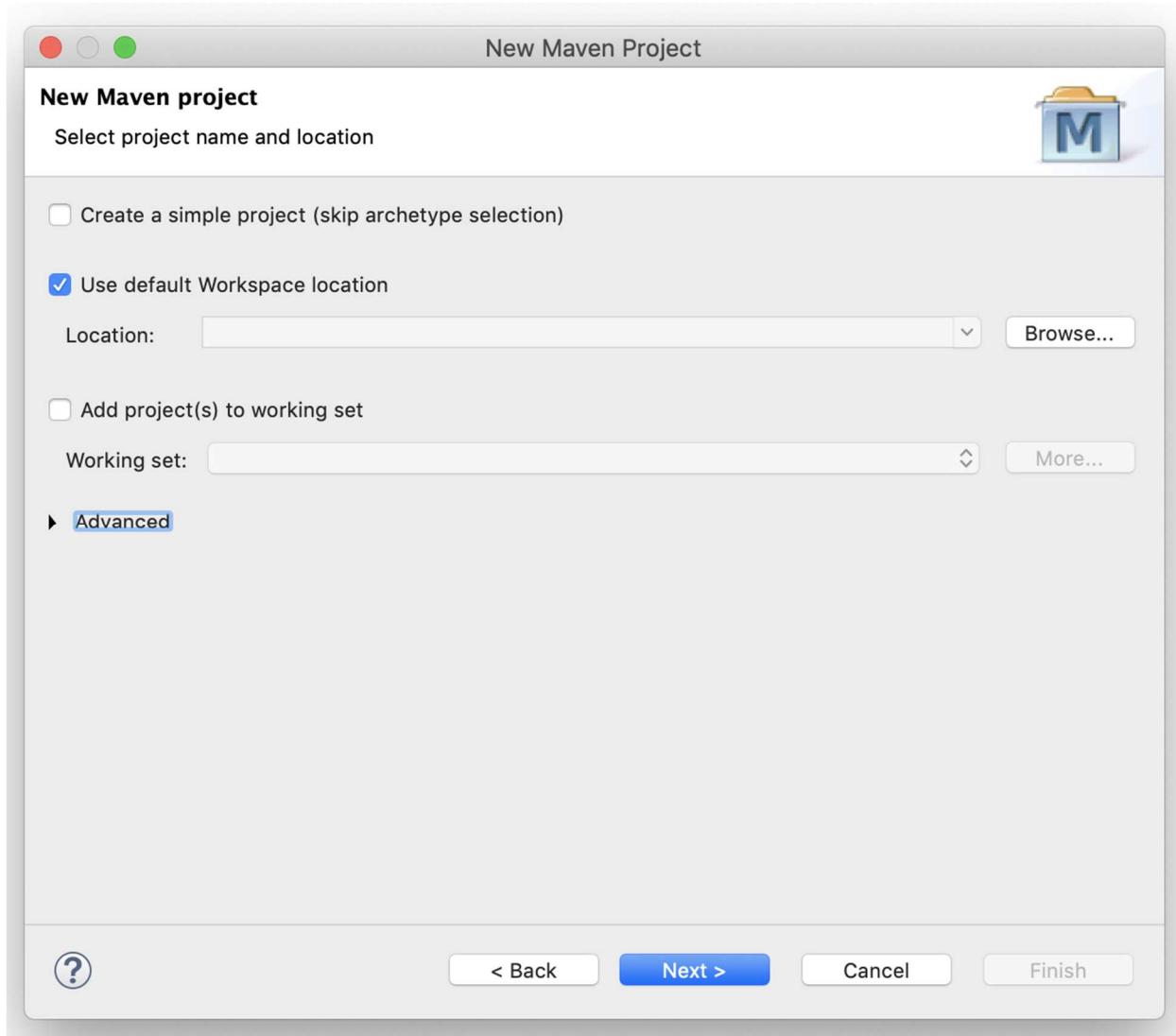


Fig. 11- Elección de la ubicación del proyecto

A continuación deberemos seleccionar el “Arquetipo”, que es nada más y nada menos que una plantilla preconfigurada para iniciar un nuevo proyecto, en este caso seleccionaremos “Quickstart”. Para más información sobre los arquetipos de Maven puedes consultar la documentación en <https://maven.apache.org/guides/introduction/introduction-to-archetypes.html>.

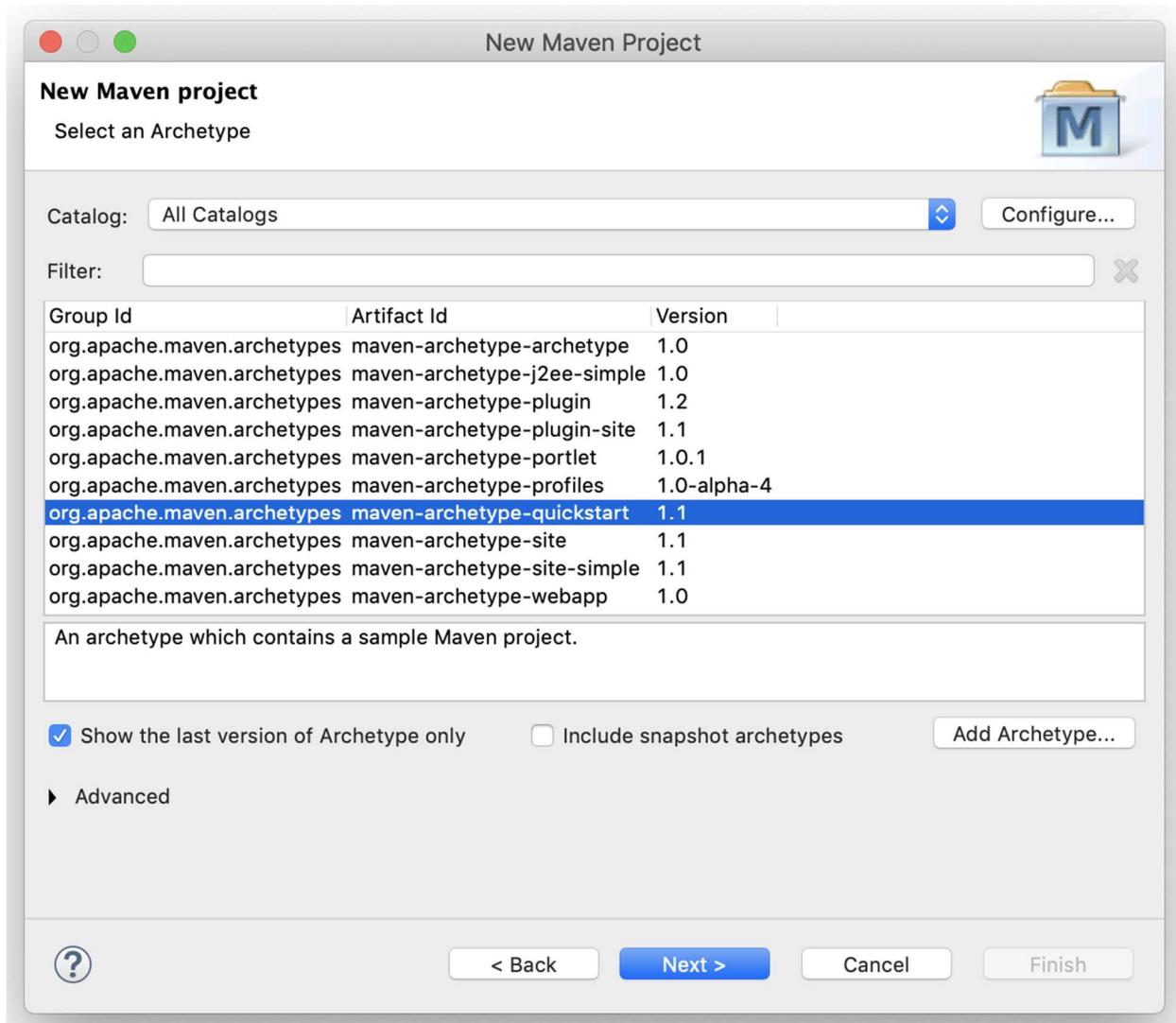


Fig. 12- Selección de un arquetipo para el proyecto

Luego deberemos especificar los datos de identificación de nuestro proyecto que, al tratarse de ser un artefacto de Maven, debe contener un ID (nombre identificador) y un grupo:

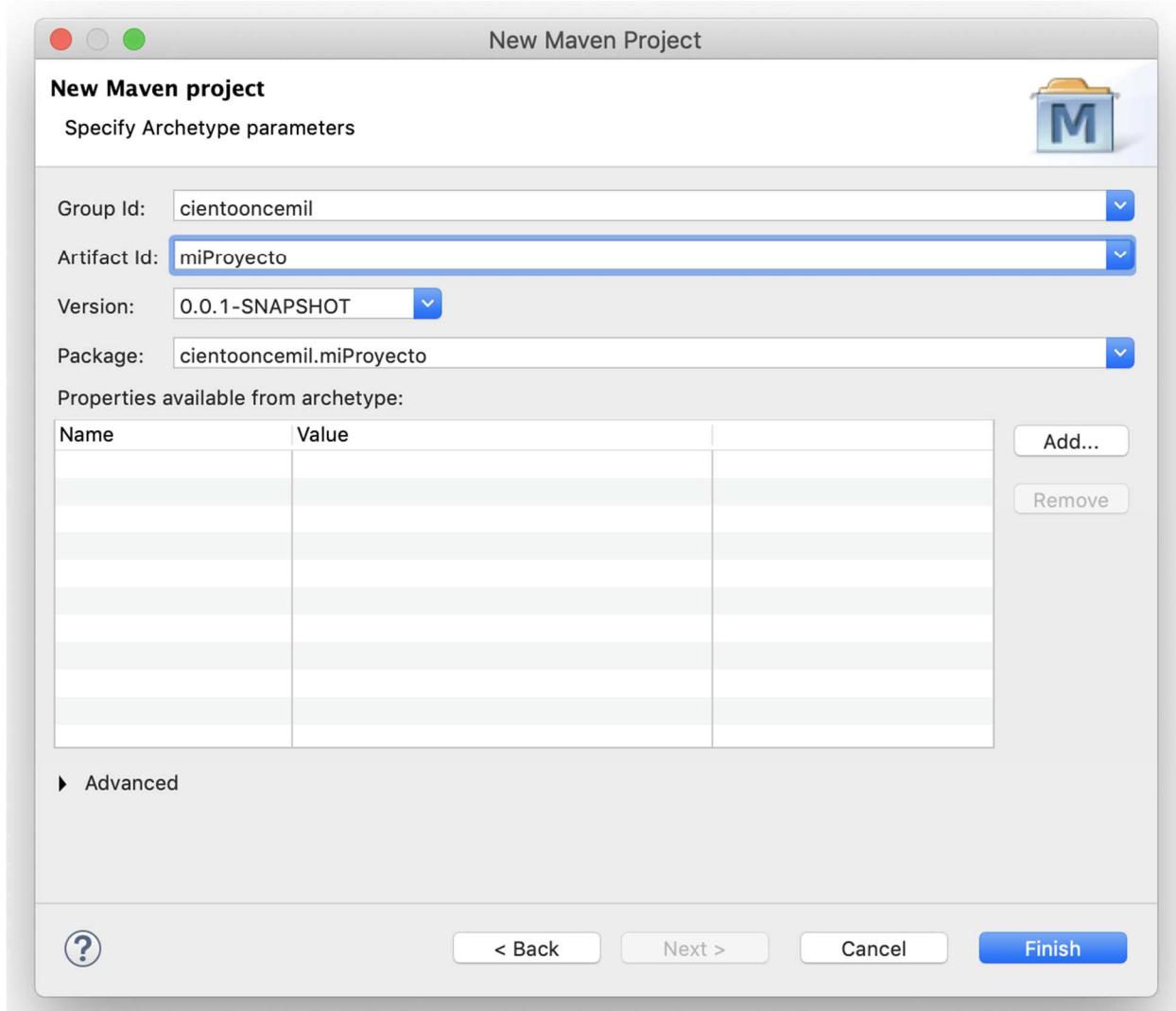


Fig. 13- Especificación de los datos de identificación de nuestro proyecto el proyecto

Al finalizar tendremos nuestro proyecto ya inicializado en base a la plantilla con la siguiente estructura:

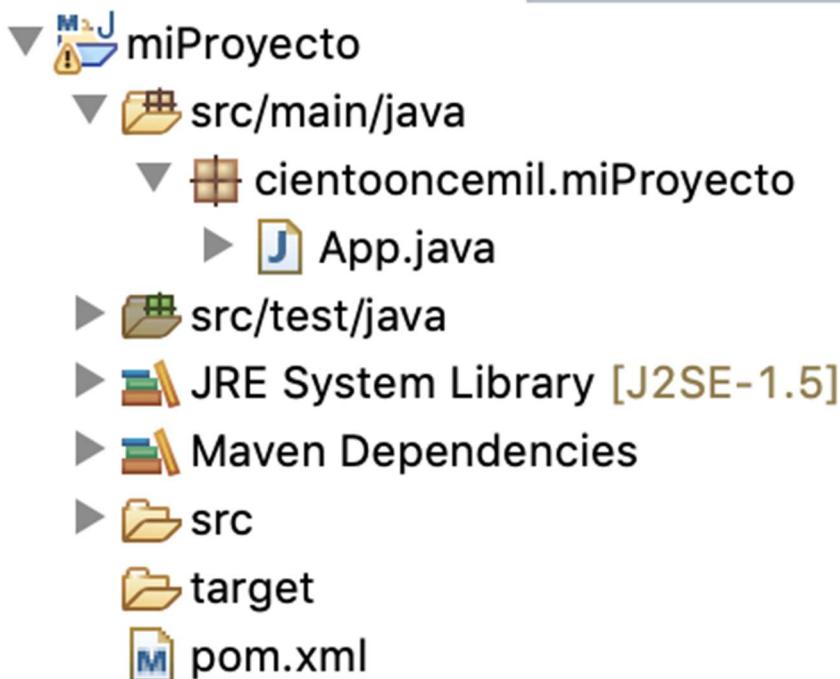


Fig. 14- Estructura de inicialización del proyecto

Y podemos ver que el archivo POM de configuración ya ha sido creado para nosotros con el siguiente contenido:

```

<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>cientooncemil</groupId>
    <artifactId>miProyecto</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>miProyecto</name>
    <url>http://maven.apache.org</url>

    <properties>
```

```
<project.build.sourceEncoding>UTF-  
8</project.build.sourceEncoding>  
</properties>  
  
<dependencies>  
  <dependency>  
    <groupId>junit</groupId>  
    <artifactId>junit</artifactId>  
    <version>3.8.1</version>  
    <scope>test</scope>  
  </dependency>  
</dependencies>  
</project>
```

Sólo con editar lo podremos agregar nuevas dependencias o modificar opciones de configuración de nuestro proyecto. Para poder actualizar las dependencias descriptas en el POM sólo tenemos que seleccionar la opción “Update Project” dentro del menú contextual de nuestro proyecto:

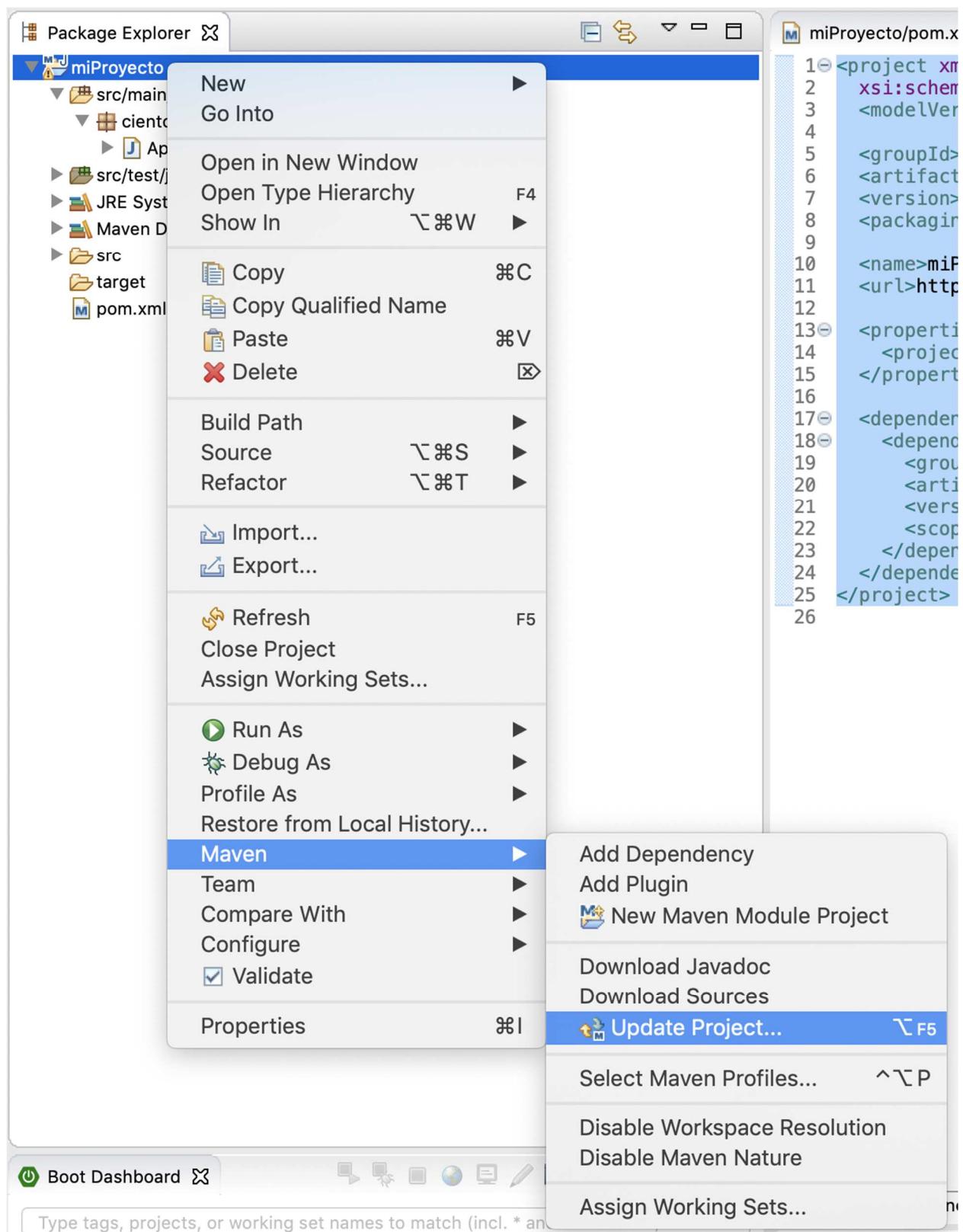


Fig. 15- Actualización del proyecto con el menú contextual

Inicialización de un proyecto Spring Boot

Para poder crear un nuevo proyecto de Spring Boot tenemos dos alternativas: desde Spring Tool Suite con el asistente de creación de un nuevo proyecto o utilizando Spring Initializr en <https://start.spring.io/>.

Creando un nuevo proyecto Spring Boot con Spring Initializr

Utilizando la interfaz web de generación de proyectos Spring Initializr podemos indicar los datos de identificación de nuestro proyecto (un proyecto Maven!):

The screenshot shows the Spring Initializr interface for creating a Maven Project. The configuration includes:

- Project:** Maven Project (selected)
- Language:** Java (selected)
- Spring Boot:** 2.1.5 (selected)
- Project Metadata:**
 - Group: com.cientoencemil
 - Artifact: demo-web
- Options:**
 - Name: demo-web
 - Description: Demo project for Spring Boot
 - Package Name: com.cientoencemil.demo-web
 - Packaging: Jar (selected)
 - Java: 8 (selected)

Fig. 16- Interfaz web de generación de proyectos

Además especificar adicionalmente qué dependencias queremos incluir, en nuestro caso:

Selected dependencies

Spring Web Starter

Build web, including RESTful, applications using Spring MVC.



Uses Tomcat as the default embedded container.

Spring Boot DevTools

Provides fast application restarts, LiveReload, and configurations for enhanced development experience.



Spring Data JPA

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.



Fig. 17- Selección de Dependencias

Y finalmente seleccionamos la opción “Generate the project”, lo cual nos generará una versión ZIP para descargar y descomprimir en nuestra pc. A continuación deberemos importar el proyecto descargado a Spring Tool Suite utilizando la opción Archivo, Importar proyecto desde el sistema de archivos:

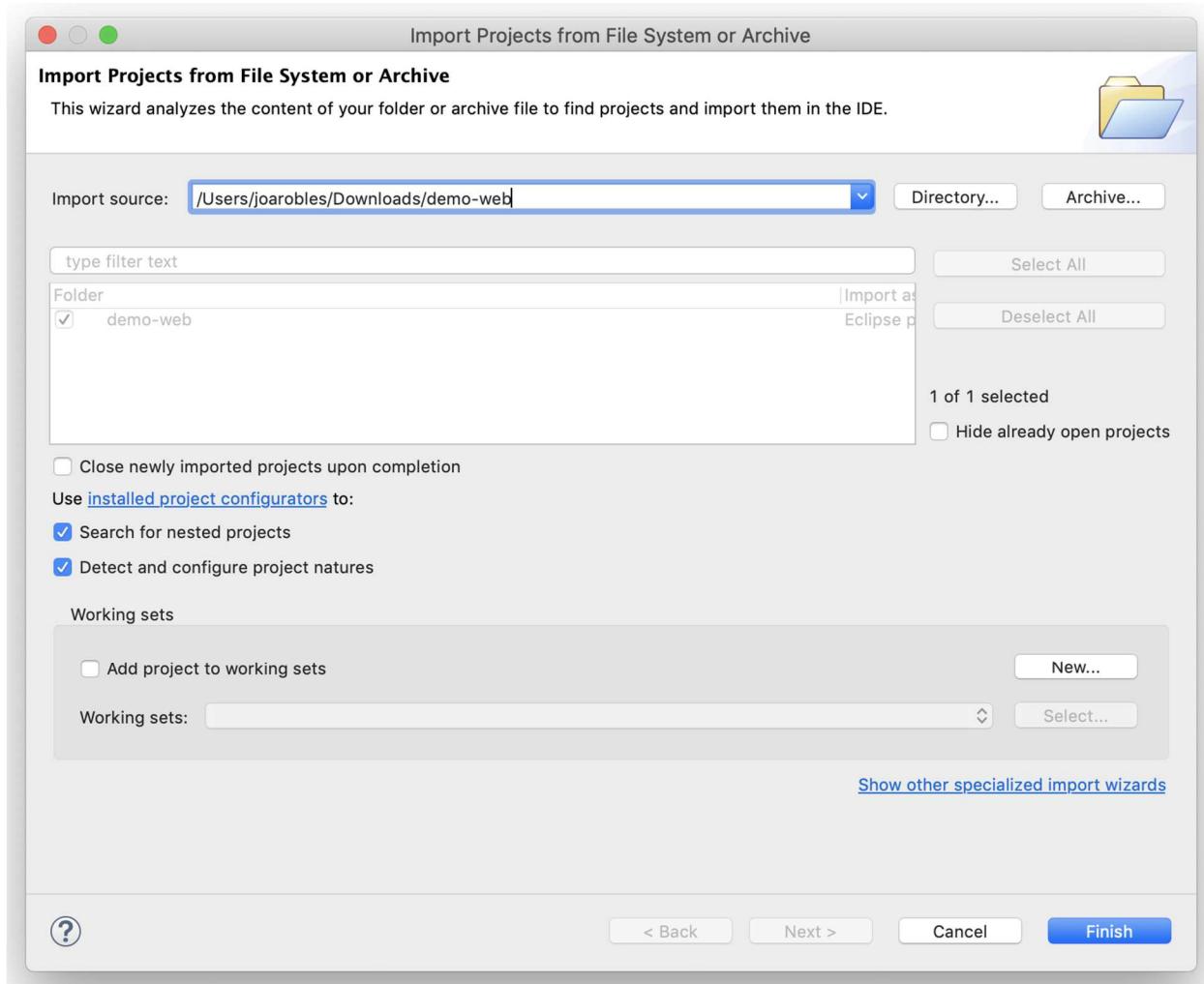


Fig. 18- Importar proyecto desde el sistema de archivos

Y luego seleccionar la opción Finalizar. A continuación Maven se encargará de descargar todas las dependencias necesarias indicadas en el archivo POM y configurar nuestro proyecto inicial, al finalizar tendremos una estructura de proyecto similar a la siguiente:

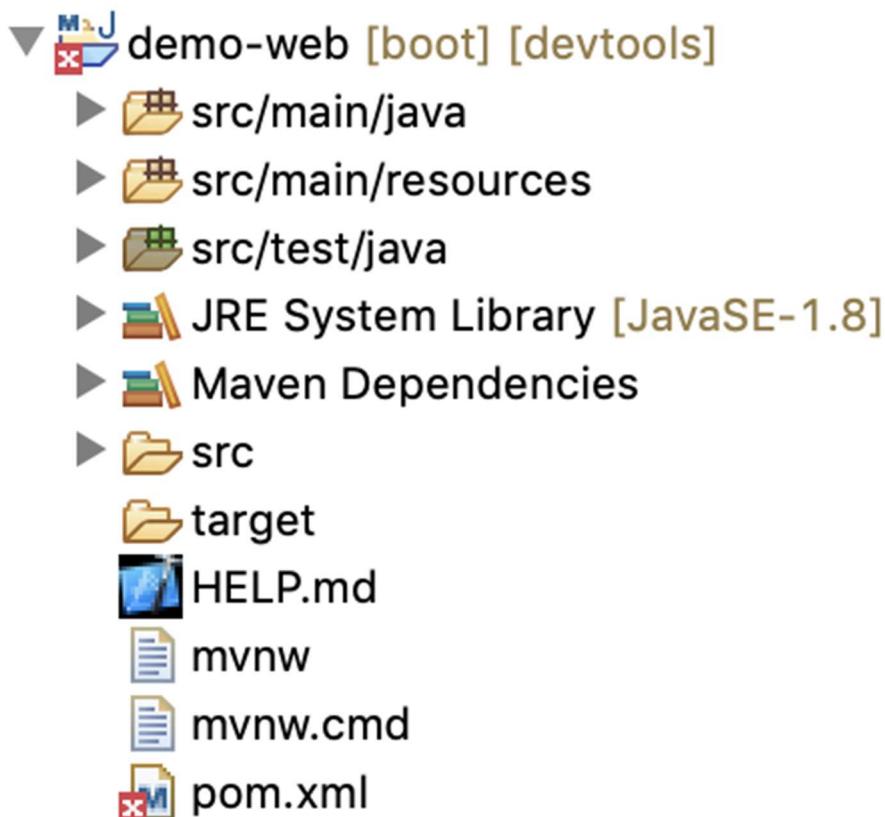


Fig. 19- Estructura del proyecto

Ahora bien, para poder conectarnos a una base de datos MariaDB debemos agregar la siguiente dependencia a nuestro archivo POM:

```
<dependency>
    <groupId>org.mariadb.jdbc</groupId>
    <artifactId>mariadb-java-client</artifactId>
</dependency>
```

Para poder ejecutar nuestro proyecto sólo tenemos que seleccionar la opción “Start/Restart” de la vista “Boot Dashboard”:

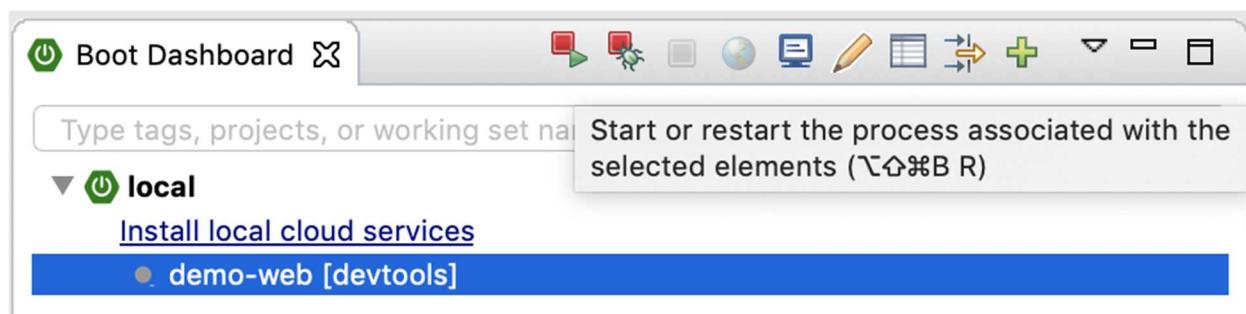


Fig. 20- Ejecución del proyecto desde Boot Dashboard

Al iniciar nuestra aplicación de Spring Boot veremos una salida como la siguiente:



```

  \_\_\_\_\_  _\_\_\_\_\_
  \_\_\_\_\_|\_\_\_\_\_|\_\_\_\_\_
  \_\_\_\_\_|_\_\_\_\_|_\_\_\_\_
  :: Spring Boot ::      (v2.1.5.RELEASE)

2019-06-14 11:57:18.540 INFO 47446 --- [ restartedMain] c.c.demoweb.DemoWebApplication      : Starting DemoWebApplication on MacBook-Pro-de-Joaquin-2.local with PID 47446 (/Users/joarobles/C
2019-06-14 11:57:18.542 INFO 47446 --- [ restartedMain] c.c.demoweb.DemoWebApplication      : No active profile set, falling back to default profiles: default
2019-06-14 11:57:18.582 INFO 47446 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : Devtools property defaults active! Set 'spring.devtools.add-properties' to 'false' to disable
2019-06-14 11:57:18.584 INFO 47446 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging.level.web' property to DEBUG
2019-06-14 11:57:19.039 INFO 47446 --- [ restartedMain] .e.DevToolsPropertyDefaultsPostProcessor : For additional web related logging consider setting the 'logging.level.web' property to DEBUG
2019-06-14 11:57:19.046 INFO 47446 --- [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 9ms. Found 0 repository interfaces.
2019-06-14 11:57:19.273 INFO 47446 --- [ restartedMain] trationDelegates$eanPostProcessorChecker : Bean 'org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration' of typ
2019-06-14 11:57:19.474 INFO 47446 --- [ restartedMain] o.s.d.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2019-06-14 11:57:19.484 INFO 47446 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2019-06-14 11:57:19.485 INFO 47446 --- [ restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.19]
2019-06-14 11:57:19.541 INFO 47446 --- [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[] : Initializing Spring embedded WebApplicationContext
2019-06-14 11:57:19.542 INFO 47446 --- [ restartedMain] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 959 ms
2019-06-14 11:57:19.585 INFO 47446 --- [ restartedMain] ConfigServletWebServerApplicationContext : Exception encountered during context initialization - cancelling refresh attempt: org.springfram
2019-06-14 11:57:19.588 INFO 47446 --- [ restartedMain] o.apache.catalina.core.StandardService : Stopping service [Tomcat]
2019-06-14 11:57:19.600 INFO 47446 --- [ restartedMain] ConditionEvaluationReportLoggingListener : Stopping service [Tomcat]

Error starting ApplicationContext. To display the conditions report re-run your application with 'debug' enabled.
2019-06-14 11:57:19.605 ERROR 47446 --- [ restartedMain] o.s.b.d.LoggingFailureAnalysisReporter : ****
APPLICATION FAILED TO START
****

Description:
Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class

Action:
Consider the following:
If you want an embedded database (H2, HSQL or Derby), please put it on the classpath.
If you have database settings to be loaded from a particular profile you may need to activate it (no profiles are currently active).


```

Fig. 21- Visualización de la salida que se muestra al iniciar la aplicación de Spring Boot

En donde podemos ver que nuestra aplicación no finalizó el inicio debido a que falta especificar los datos de conexión a la base de datos MariaDB, lo cual podremos hacer utilizando el archivo principal de configuración application.properties disponible dentro de la carpeta de recursos:



Fig. 22- Especificación de los datos de conexión de la Base de Datos MariaDB

Donde ingresaremos el siguiente contenido:

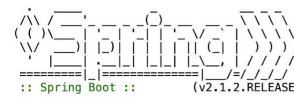
```

spring.datasource.url=jdbc:mariadb://localhost:3306/BASE_DE_DATOS
spring.datasource.username=USUARIO
spring.datasource.password=PASSWORD
spring.jpa.hibernate.ddl-auto=update

```

De esta forma tendremos ya configurada la conexión con nuestra base de datos MariaDB y además dispondremos de generación automática del schema de la base de datos en base a las directivas de mapeo que especifiquemos en nuestras clases de entidad.

Ahora al iniciar nuevamente nuestro proyecto podremos ver la siguiente salida:



```

:: Spring Boot :: (v2.1.2.RELEASE)

2019-06-14 12:14:41.483 INFO 47554 --- [ restartedMain] c.m.d.DeliveryfastApplication      : Starting DeliveryfastApplication on MacBook-Pro-de-Joaquin-2.local with PID 4755
2019-06-14 12:14:41.483 INFO 47554 --- [ restartedMain] c.m.d.DeliveryfastApplication      : No active profile set, falling back to default profiles: default
2019-06-14 12:14:41.663 INFO 47554 --- [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data repositories in DEFAULT mode.
2019-06-14 12:14:41.670 INFO 47554 --- [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 7ms. Found 3 repository interfaces.
2019-06-14 12:14:41.723 INFO 47554 --- [ restartedMain] trationDelegates$BeanPostProcessorChecker : Bean 'org.springframework.transaction.annotation.ProxyTransactionManagementConfi
2019-06-14 12:14:41.788 INFO 47554 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2019-06-14 12:14:41.787 INFO 47554 --- [ restartedMain] org.apache.catalina.core.StandardService : Starting service [Tomcat]
2019-06-14 12:14:41.787 INFO 47554 --- [ restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.14]
2019-06-14 12:14:41.797 INFO 47554 --- [ restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2019-06-14 12:14:41.797 INFO 47554 --- [ restartedMain] o.s.web.context.ContextLoader      : Root WebApplicationContext: initialization completed in 312 ms
2019-06-14 12:14:41.849 INFO 47554 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource   : HikariPool-2 - Starting...
2019-06-14 12:14:41.852 INFO 47554 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource   : HikariPool-2 - Start completed.
2019-06-14 12:14:41.856 INFO 47554 --- [ restartedMain] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [
    name: default
    ...
2019-06-14 12:14:41.864 INFO 47554 --- [ restartedMain] org.hibernate.dialect.Dialect      : HHH000400: Using dialect: org.hibernate.dialect.MariaDB103Dialect
2019-06-14 12:14:42.077 INFO 47554 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2019-06-14 12:14:42.087 INFO 47554 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer     : LiveReload server is running on port 35729
2019-06-14 12:14:42.179 INFO 47554 --- [ restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2019-06-14 12:14:42.186 WARN 47554 --- [ restartedMain] a.webConfigurations$JpaWebMvcConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may b
2019-06-14 12:14:42.253 INFO 47554 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2019-06-14 12:14:42.254 INFO 47554 --- [ restartedMain] c.m.d.DeliveryfastApplication      : Started DeliveryfastApplication in 0.787 seconds (JVM running for 25.893)
2019-06-14 12:14:42.258 INFO 47554 --- [ restartedMain] ConditionEvaluationDeltaLoggingListener : Condition evaluation unchanged

```

Fig. 23- Visualización de la salida que se muestra al iniciar exitosamente la aplicación de Spring Boot

Y acceder a nuestra aplicación web mediante la siguiente URL <http://localhost:8080>.

Creación de clases de Control con anotaciones

La creación de controladores con Spring Boot es muy sencilla, sólo debemos agregar a nuestras clases de control una anotación en particular @Controller. De esta forma Spring Boot creará automáticamente las instancias necesarias de nuestros controladores en la medida en que sean necesarios para lograr la comunicación con la interfaz:

```
@Controller
@RequestMapping("pedidos")
public class PedidosController {

    ...
}
```

Además deberemos agregar una anotación de @RequestMapping que nos permitirá especificar el prefijo de URLs a las que responderá este controlador, en el caso anterior “pedidos”, de la forma siguiente:

[http://localhost:8080/pedidos/...](http://localhost:8080/pedidos/)

Luego podemos crear los métodos específicos que responderán a cada solicitud anotándolos con @GetMapping o @PostMapping, si se tratan de solicitudes HTTP de tipo GET o POST respectivamente. Para más información sobre los métodos de solicitudes de HTTP puedes consultar <https://developer.mozilla.org/es/docs/Web/HTTP/Methods>.

De esta forma si creamos el método pedirLoQueSea lo anotamos de la siguiente forma:

```
@GetMapping("nuevo")
public void pedirLoQueSea(Model model) {
```

Así al ingresar a la URL <http://localhost:8080/pedidos/nuevo> ejecutaremos el método pedirLoQueSea que recibirá por parámetros una instancia de la interfaz Model de Spring, que nos permitirá intercambiar datos con nuestra vista de la siguiente forma:

```
Float unMonto = 20;
model.addAttribute("montoDeComision", unMonto);
```

De esta forma dispondremos de la variable unMonto bajo el nombre montoDeComision disponible en nuestra vista para poder mostrarla.

Para más información sobre los controladores en Spring Boot puedes consultar la siguiente guía oficial: <https://spring.io/guides/gs/serving-web-content/>.

Plantillas HTML5 con el motor Thymeleaf

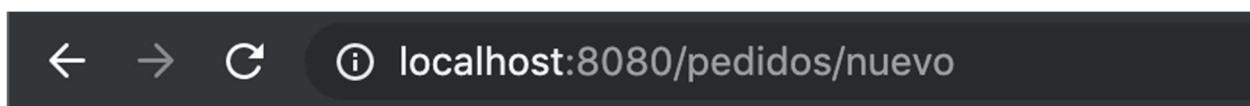
Para poder mostrar nuestra interfaz mediante una página web deberemos crear plantillas utilizando el lenguaje de maquetado HTML5 y el motor de plantillas Thymeleaf. Puedes consultar la documentación oficial de Thymeleaf en <https://www.thymeleaf.org/> y la guía de desarrollo de Mozilla para HTML5 en <https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5>.

Crear una nueva plantilla es tan sencillo como crear un nuevo archivo con extensión .html cuyo nombre, para facilitar la resolución automática, debe coincidir con el nombre del mapping (especificado en el value de la anotación GetMapping o PostMapping correspondiente) dentro de nuestra clase de control que la mostrará.

Para el ejemplo anterior crearemos el archivo nuevo.html dentro de la carpeta templates ubicada en nuestro paquete de recursos:

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Nuestra primera plantilla</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
    <p th:text="'El monto de comisión es de ' + ${montoDeComision}" />
</body>
</html>
```

Lo que producirá la siguiente salida al ingresar a <http://localhost:8080/pedidos/nuevo>:



El monto de comisión es de 35

Fig. 26- Visualización de la salida al ingresar la dirección <http://localhost:8080/pedidos/nuevo>

De forma análoga, podemos crear formularios de HTML para poder enviar datos a nuestro controlador.

Para más información sobre formularios de HTML puedes consultar
<https://www.thymeleaf.org/doc/tutorials/2.1/thymeleafspring.html#creating-a-form>.

Maquetado de vistas con Bootstrap

Bootstrap es un kit de desarrollo HTML, CSS y Javascript orientado al desarrollo de aplicaciones web responsivas, es decir, que pueden funcionar en diferentes dispositivos como computadoras, smartphones y tablets, adaptando la disposición del contenido a la pantalla que lo muestra.

Para incluir Bootstrap en nuestras plantillas HTML sólo debemos agregar la referencia al archivo de estilos dentro de la sección `<head>` de la siguiente forma:

```
<link rel="stylesheet"  
      href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css">
```

Además, para poder incluir el soporte a funciones adicionales debemos incluir las librerías de Javascript necesarias, ubicando el siguiente contenido al final de nuestras plantillas HTML, justo antes de cerrar la etiqueta `<body>`:

```
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"></script>  
<script  
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"></script>  
<script  
src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"></script>
```

Utilizando clases de CSS y un sistema de grilla de 12 columnas, Bootstrap nos permite distribuir el contenido de nuestra plantilla HTML en función del tamaño de pantalla del dispositivo mediante el cual accedemos a nuestra aplicación web. En su documentación oficial puedes conocer un poco más acerca de su funcionamiento, <https://getbootstrap.com/>.

Además, podemos definir nuestras propias hojas de estilos CSS para cambiar la manera en la que nuestras plantillas HTML son mostradas por los diferentes navegadores. De esta forma es posible no sólo cambiar colores, tamaños o posiciones, sino también modificar parámetros muy específicos que hacen que cada aplicación web pueda implementar un diseño gráfico diferenciado.

Las hojas de estilo CSS no son más que archivos de texto con la extensión .css con una estructura similar a la siguiente:

```
etiqueta,.nombre-de-clase, #identificador-unico {  
    propiedad: valor;  
}
```

Por ejemplo, para asignar un color de texto rojo a un título podríamos escribir el siguiente código HTML:

```
<h1>Este es un título rojo</h1>
```

Y agregar a nuestra hoja de estilos lo siguiente:

```
h1 {  
    color: #ff0000;  
}
```

Lo que producirá la siguiente salida en el navegador:

Este es un título rojo

Lo que indica que todas las etiquetas HTML denominadas h1, es decir todos los títulos, deben tener como color de tipografía aquel con código hexadecimal #FF0000 que es el rojo. Puedes encontrar un listado exhaustivo de etiquetas HTML disponibles en la referencia <https://www.w3schools.com/tags/>.

Javascript y jQuery

JavaScript es un lenguaje de *scripting* multiplataforma, orientado a objetos, pequeño y liviano. JavaScript contiene una librería estándar de objetos, tales como Array, Date, y Math, y un conjunto central de elementos del lenguaje, tales como operadores, estructuras de control, y sentencias. El núcleo de JavaScript puede extenderse para varios propósitos, complementándolo con objetos adicionales, por ejemplo:

- JavaScript **del lado del cliente** extiende el núcleo del lenguaje proporcionando objetos para controlar un navegador. Por ejemplo, las extensiones del lado del cliente permiten que una aplicación coloque elementos en un formulario HTML y responda a eventos del usuario, tales como clicks del mouse, ingreso de datos al formulario y navegación de páginas.
- JavaScript **del lado del servidor** extiende el núcleo del lenguaje proporcionando objetos relevantes a la ejecución de JavaScript en un servidor. Por ejemplo, las extensiones del lado del servidor permiten que una aplicación se comunique con una base de datos, trabajo con archivos en un servidor, entre otras cosas.

JavaScript y Java son similares en algunos aspectos, pero fundamentalmente diferentes en otros. El lenguaje JavaScript se parece a Java pero no tiene el tipo estático (static) de Java, ni tiene un chequeo de tipos fuerte. Además usa la mayoría de la sintaxis de expresiones de Java, convenciones de nombrado, y las construcciones básicas de control de flujo.

A diferencia de Java que cuenta con un sistema de clases construidas por declaraciones que se usa en tiempo de compilación, JavaScript soporta un sistema de tiempo de ejecución basado en un pequeño número de tipos de datos que representan valores numéricos, lógicos, y de cadena de caracteres (string). Tiene un modelo de objetos basado en prototipos en lugar del modelo de objetos basado en clases, que es más común. Este modelo proporciona herencia dinámica; esto es, que lo que se hereda puede variar entre objetos individuales. JavaScript también soporta funciones sin ningún requerimiento declarativo especial. Las funciones pueden ser propiedades de los objetos, ejecutándose como métodos débilmente tipados.

JavaScript funciona muy bien integrado con HTML para darle funcionalidad a la capa de presentación de nuestra aplicación, así, una porción de código Javascript puede ser incorporada a una plantilla HTML5 con la siguiente etiqueta:

```
<script>
// código Javascript
</script>
```

Puedes encontrar más información sobre JavaScript en el enlace:
<https://developer.mozilla.org/es/docs/Web/JavaScript/Guide>.

jQuery

JQuery es una librería de JavaScript de código abierto que simplifica la tarea de programar en JavaScript y permite agregar interactividad a un sitio web con mayor facilidad. jQuery es liviano, de bajo tamaño y posee múltiples aplicaciones que nos permiten lograr gran cantidad de funcionalidad con menos código (y más legible). Nos permite facilitar cosas como recorrer y manipular nuestro HTML, manejar eventos, aplicar animaciones, entre muchas otras cosas.

Para utilizar jQuery en nuestra plantilla HTML primero debemos agregar su referencia ya sea mediante la CDN provista o descargando una versión desde la web y guardándola de manera local. Puedes encontrar todas las formas disponibles en <https://jquery.com/download/>.

Una vez incorporada manipular nuestra vista HTML como por ejemplo para modificar el texto de un botón es tan sencillo como esto:

```
<script>
$( "button.continuar" ).html( "Cargando..." )
</script>
```

Existen muchísimos plugins tanto oficiales como de otros desarrolladores para jQuery que permiten reutilizar mucha funcionalidad para agregarla a tus plantillas. Puedes encontrar más información en <https://www.npmjs.com/search?q=keywords.jquery-plugin>.

Introducción a TDD

TDD, del inglés *Test Driven Development*, es una técnica de diseño e implementación de software incluida dentro de la metodología XP que se centra en tres pilares fundamentales:

1. La implementación de las funciones justas que el cliente necesita y no más
2. La minimización del número de defectos que llegan al software en fase de producción
3. La producción de software modular, altamente reutilizable y preparado para el cambio

Pasamos, de pensar en implementar User Stories, a pensar en ejemplos certeros que eliminan la ambigüedad creada por el lenguaje natural (nuestro idioma). Hasta ahora estábamos acostumbrados a que las User Stories eran las unidades de trabajo más pequeñas sobre las que ponerse a desarrollar código. Con TDD intentamos traducir la User Story en cierta cantidad de ejemplos, hasta que el número de ejemplos sea suficiente como para describir la funcionalidad sin lugar a malinterpretaciones de ningún tipo.

En TDD dejamos que la propia implementación de pequeños ejemplos, en constantes iteraciones, hagan *emergir la arquitectura* que necesitamos usar. Lógicamente tendremos que saber si el desarrollo será para un smartphone, para una web o para una pc de escritorio porque tenemos que elegir unas herramientas de desarrollo conformes a las exigencias, sin embargo nos limitamos a escoger el framework correspondiente y a usar su arquitectura como base.

Kent Beck, uno de los padres de la metodología XP, da unos argumentos muy claros y directos sobre por qué es beneficioso convertir a TDD en nuestra herramienta de diseño principal, entre ellas:

- La calidad del software aumenta
- Conseguimos código altamente reutilizable
- El trabajo en equipo se hace más fácil, une a las personas
- Nos permite confiar en nuestros compañeros aunque tengan menos experiencia
- Multiplica la comunicación entre los miembros del equipo
- Las personas encargadas de la garantía de calidad adquieren un rol más inteligente e interesante
- Escribir el ejemplo (test) antes que el código nos obliga a escribir el mínimo de funcionalidad necesaria, evitando sobre diseñar.
- Cuando revisamos un proyecto desarrollado mediante TDD, nos damos cuenta de que los tests son la mejor documentación técnica que podemos consultar

El Ciclo de Desarrollo de TDD

Implementar TDD en la práctica es muy sencillo, sólo hay que seguir los siguientes pasos:

1. Escribir una Prueba

En base a la especificación del requerimiento, o nuestra User Story, deberemos comprender la funcionalidad a implementar tanto en su camino feliz como sus alternativas, excepciones y condiciones de error. En base a esta comprensión escribimos una prueba en algún framework de Testing Unitario para el stack de tecnologías que utilice nuestro proyecto, reutilizando una prueba anterior si estuviera disponible. Este paso hace explícita la escritura de la prueba antes de la implementación, a diferencia de la aproximación tradicional de escribir las pruebas unitarias luego de la implementación.

2. Ejecutar todas las pruebas y comprobar si la nueva prueba pasa

Antes de pasar a la implementación deberemos ejecutar todo el conjunto de pruebas y comprobar que la nueva prueba falla, y por la razón esperada. Esto es importante ya que nos permite validar que la funcionalidad efectivamente aún no ha sido implementada y que nuestra prueba está correctamente diseñada.

3. Escribir el Código

En este paso programaremos la funcionalidad y sólo la funcionalidad necesaria para que la prueba anteriormente diseñada pase. Limitarnos a codificar lo necesario para pasar el test nos permite concentrarnos en la funcionalidad requerida y que aporta valor, y no agregar funcionalidad innecesaria.

4. Ejecutar las pruebas

Al ejecutar nuevamente las pruebas obtendremos un resultado positivo, en caso de que alguna prueba falle será necesario volver al paso anterior para aplicar los cambios necesarios que permitan a nuestro código pasar.

5. Refactorizar

Debido a que este ciclo es posible de incorporar código no eficiente, duplicado o centrado localmente, es importante que con cierta frecuencia apliquemos una refactorización para optimizar la funcionalidad implementada y mejorar la calidad de nuestra implementación.

6. REPETIR!

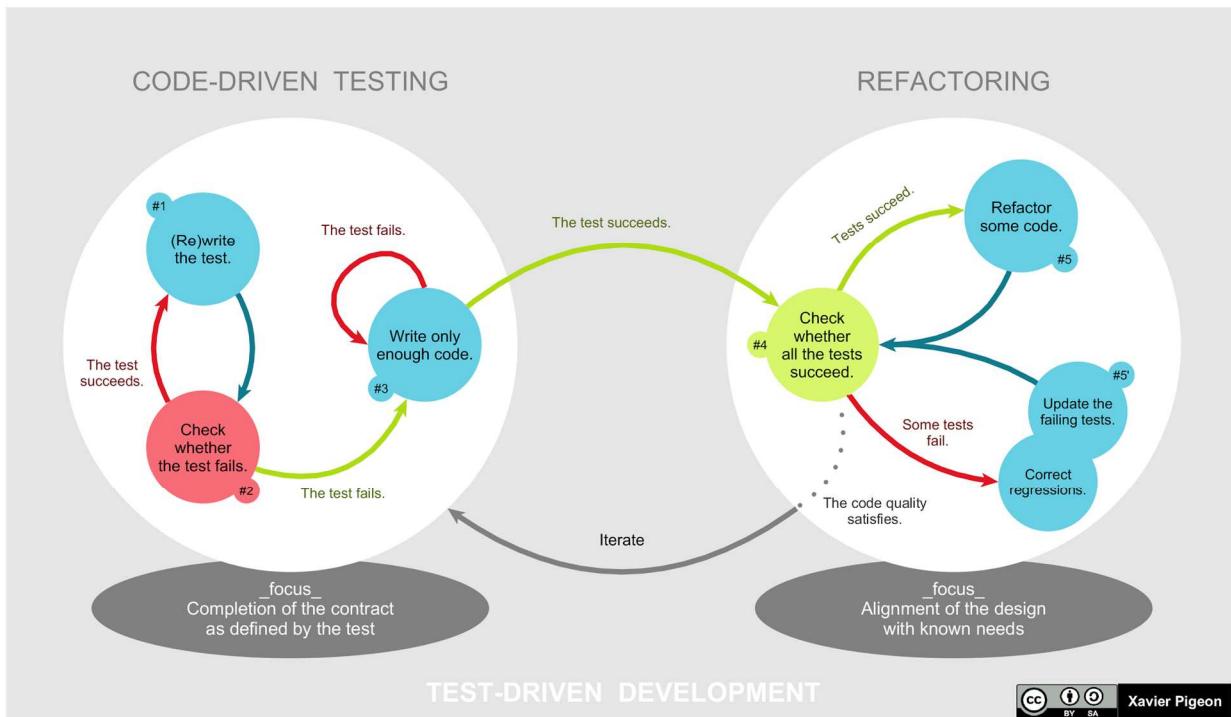


Fig. 27- Desarrollo Conducido por Pruebas (TDD)

En la siguiente sección veremos cómo escribir nuestras Pruebas Unitarias en Java con el popular framework JUnit.

Testing Unitario con JUnit

El framework JUnit, es un framework que nos permite realizar pruebas unitarias de nuestro código de manera controlada para verificar que todo esté funcionando correctamente.

¿Qué es una prueba unitaria?

Una prueba unitaria es una pieza de código escrito por un desarrollador que ejecuta una funcionalidad específica en el código que se prueba. Una prueba unitaria se dirige a una pequeña unidad de código, por ejemplo, un método o una clase. El porcentaje de código que está probado por las pruebas normalmente se llama cobertura de la prueba.

Las pruebas unitarias aseguran que el código funciona como está previsto. Tener una alta cobertura de la prueba de su código le permite continuar con el desarrollo de las funciones sin tener que realizar un montón de pruebas manuales.

Para comenzar debemos agregar a la librería de JUnit como una dependencia de nuestro proyecto, dentro de la sección <dependencies> de nuestro pom.xml de Maven agregamos la siguiente dependencia:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
```

Desde el repositorio central de Maven podemos consultar cuál es la última versión estable del framework, así como también desde la documentación oficial disponible en <https://junit.org>.

Veamos el siguiente ejemplo de código para la clase Programación del Complejo de Cines:

```
public class Programacion {

    private Date fechaInicio;
    private Date fechaFin;
    private Date fechaHoraCreada;

    // ...
}
```

```

public boolean estaVigente(Date fecha) {
    // TODO: Implementar esta funcionalidad
    throw new UnsupportedOperationException("Aún no implementado.");
}

```

Como su JavaDoc lo indica, el método `estaVigente` pregunta a la Programación si se encuentra vigente para una fecha pasada por parámetros, es decir si la fecha pasada por parámetros se encuentra dentro del rango entre la fecha de inicio de la programación y la fecha de fin. Es importante destacar que debemos considerar también los extremos, ya que la programación se considerará vigente también el mismo día de inicio y de fin.

De esta forma podríamos armar una tabla para nuestras pruebas de escritorio para validar esta funcionalidad:

Fecha de inicio: 17/10/2016

Fecha de fin: 23/10/2016

Prueba	Fecha	¿Está Vigente?
1	16/10/2016 23:59:59	NO
2	17/10/2016 00:00:00	SÍ
3	20/10/2016 00:00:00	SÍ
4	23/10/2016 23:59:59	SI
5	24/10/2016 00:00:00	NO

Este método es un candidato ideal para la aplicación de pruebas unitarias con JUnit y, en el caso de escribir antes las pruebas que la implementación, podemos utilizar la metodología TDD para orientar la implementación en base a la retroalimentación de las pruebas.

Veamos cómo se escribe un caso de prueba unitario para probar la funcionalidad anterior:

El primer paso es crear una nueva clase para el caso de pruebas unitario, para ello podemos utilizar la opción “New JUnit Test Case”:

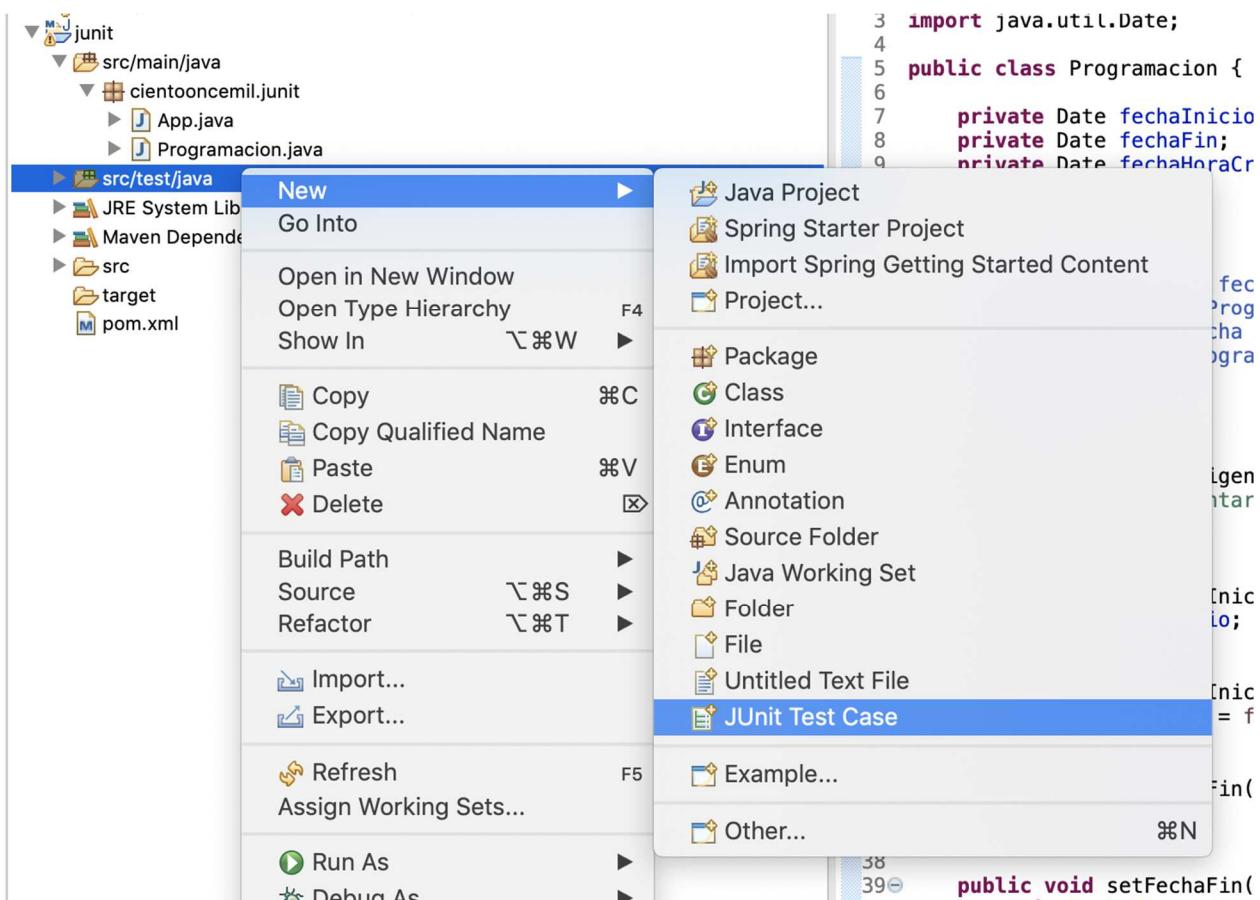


Fig. 28- Creación de un nuevo caso de prueba con JUnit

A continuación indicamos que se trata de un caso de pruebas escrito para la versión 4 de la librería JUnit y le asignamos un nombre, en este caso `ProgramacionTest`. Opcionalmente podemos seleccionar la clase que se encuentra bajo pruebas para que el IDE genere automáticamente un esqueleto de nuestra clase. Además, seleccionamos el método `setUp()` para que sea generado sin contenido, de forma tal que ahorraremos código a escribir.

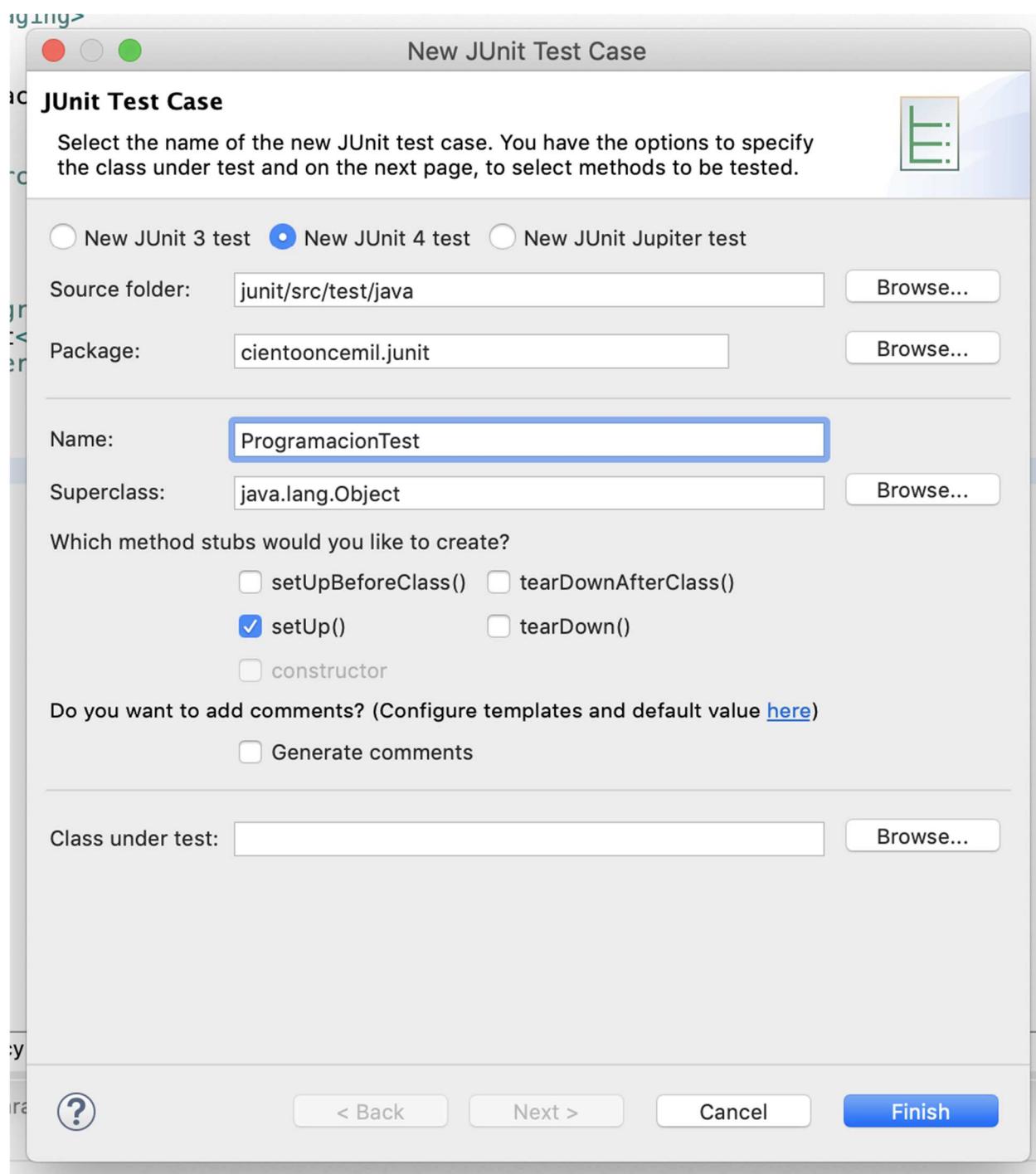


Fig. 29- Asignación de un nombre para el caso de prueba con JUnit

Veamos el contenido de la clase generada:

```
package cientooncemil.junit;

import static org.junit.Assert.*;
```

```
import org.junit.Before;
import org.junit.Test;

public class ProgramacionTest {

    @Before
    public void setUp() throws Exception {
    }

    @Test
    public void test() {
        fail("Not yet implemented");
    }
}
```

No es nada más y nada menos que una clase de Java con algunos métodos vacíos y anotaciones. La anotación `@Before` nos permite indicar que el método debe ser ejecutado antes de comenzar las pruebas, esta anotación se reserva para crear las condiciones iniciales de nuestras pruebas. En este caso, crearemos nuestra Programación indicando las fechas de inicio y fin que serán las base de pruebas.

Cada método que ejecutará una prueba en particular está anotado con `@Test`. Es muy importante que el nombre de cada método de pruebas describa de forma sintética y clara cuál es el escenario que se está probando, como por ejemplo `probarEstaVigenteAntes()` o `probarEstaVigenteDespues()`. Además, como consejo general, cada método debe tener sólo una razón por la que puede fallar, esto nos permite identificar rápidamente y sin ambigüedad cuál es la funcionalidad afectada y que debe ser modificada para ser corregida.

Antes de ejecutar esta prueba piloto, vamos a desplegar la vista de JUnit dentro del IDE que nos permitirá obtener un reporte visual de los resultados de la ejecución de cada ciclo de pruebas. Para ello seleccionamos la opción “Window”, luego “Show View” y luego “Other” para poder buscar la vista “JUnit”:

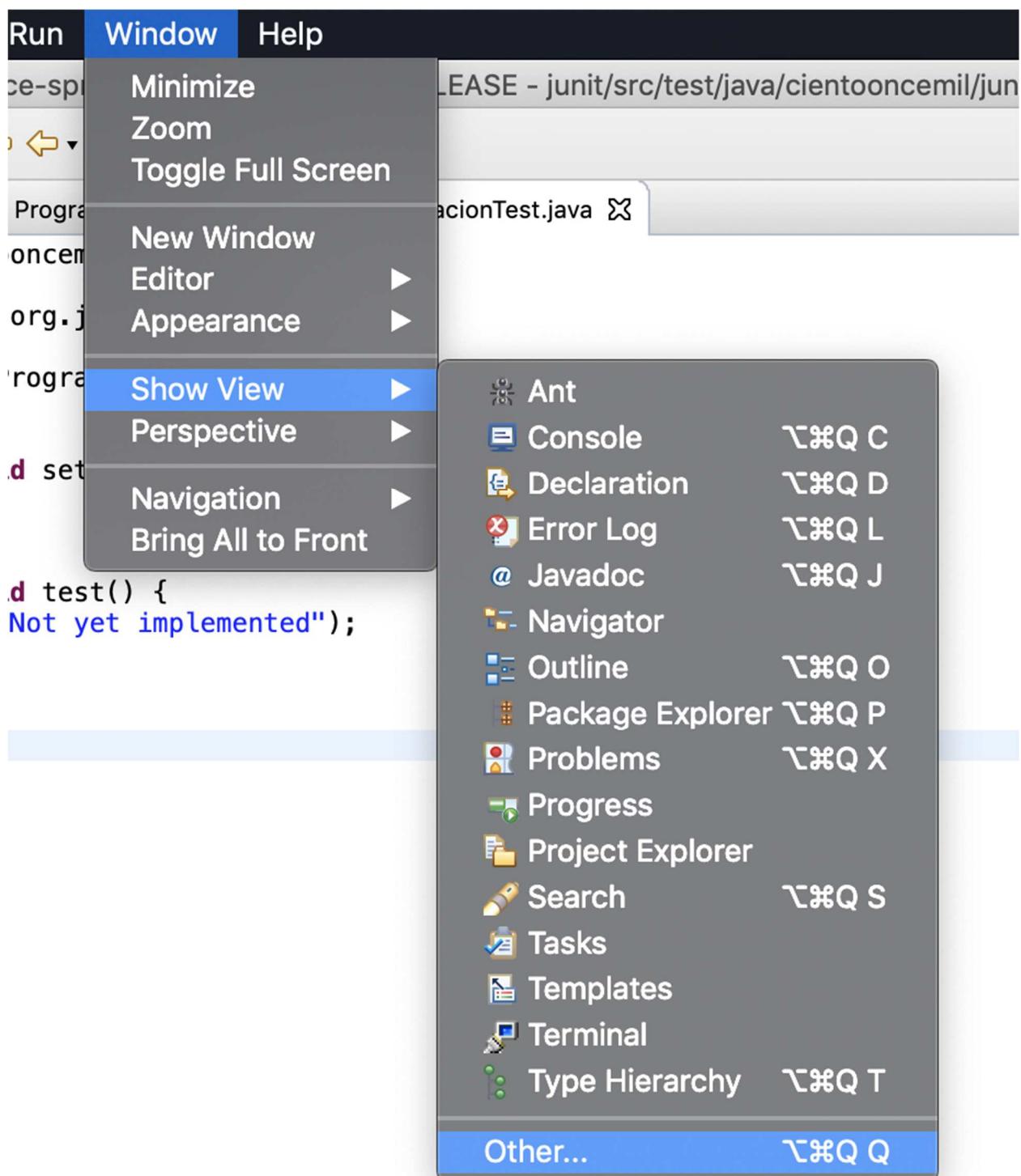


Fig. 30- Vista de JUnit dentro del IDE que nos permitirá obtener un reporte visual de los resultados

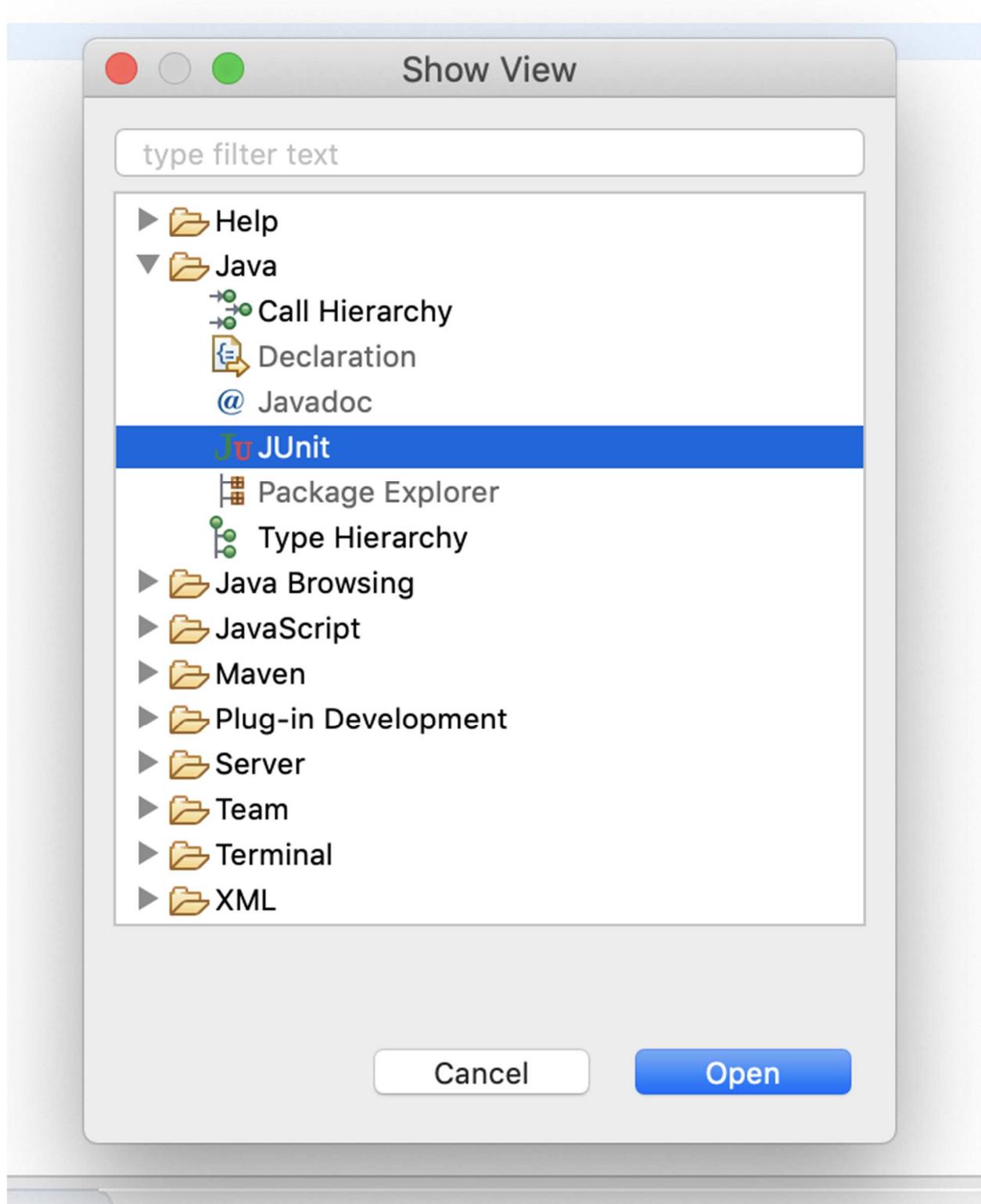


Fig. 31- Vista de JUnit

Esto nos mostrará una ventana similar a la siguiente:

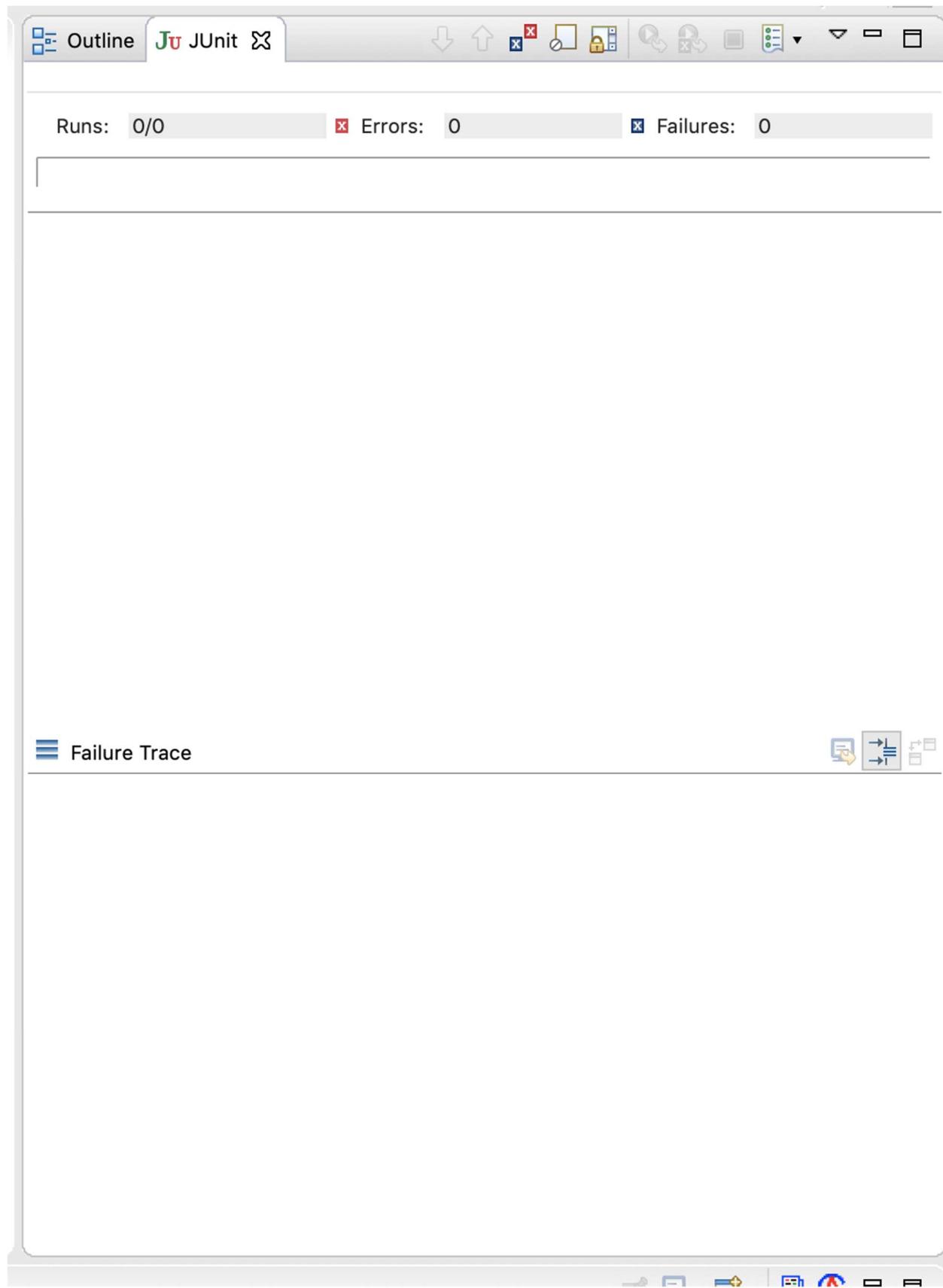


Fig. 32: Ventana que se muestra

A continuación vamos a crear una “Run Configuration” para ejecutar todas nuestras pruebas utilizando el botón “Run...” de una sola vez. Para ello desplegamos el menú “Run..” y seleccionamos la opción “Run Configurations”:

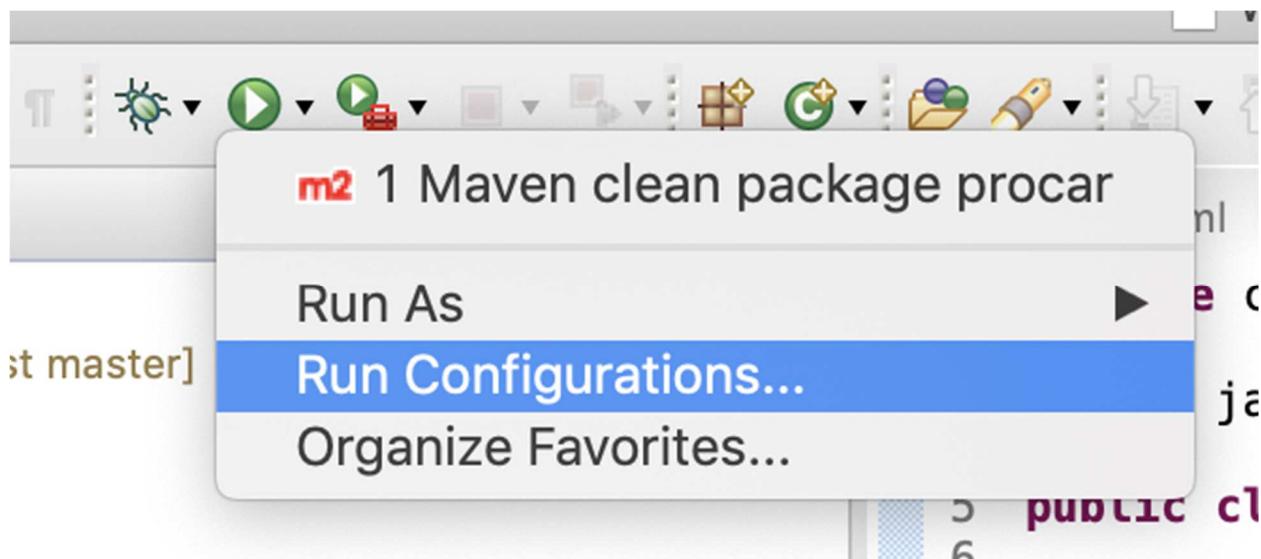


Fig. 33: Creación de una configuración para ejecutar pruebas

Y dentro de la opción “JUnit” creamos una nueva configuración, asignándole como nombre “Correr todas las pruebas con JUnit” o similar. Además debemos indicar que se ejecutarán todas las clases de prueba disponibles utilizando la versión 4 del framework:

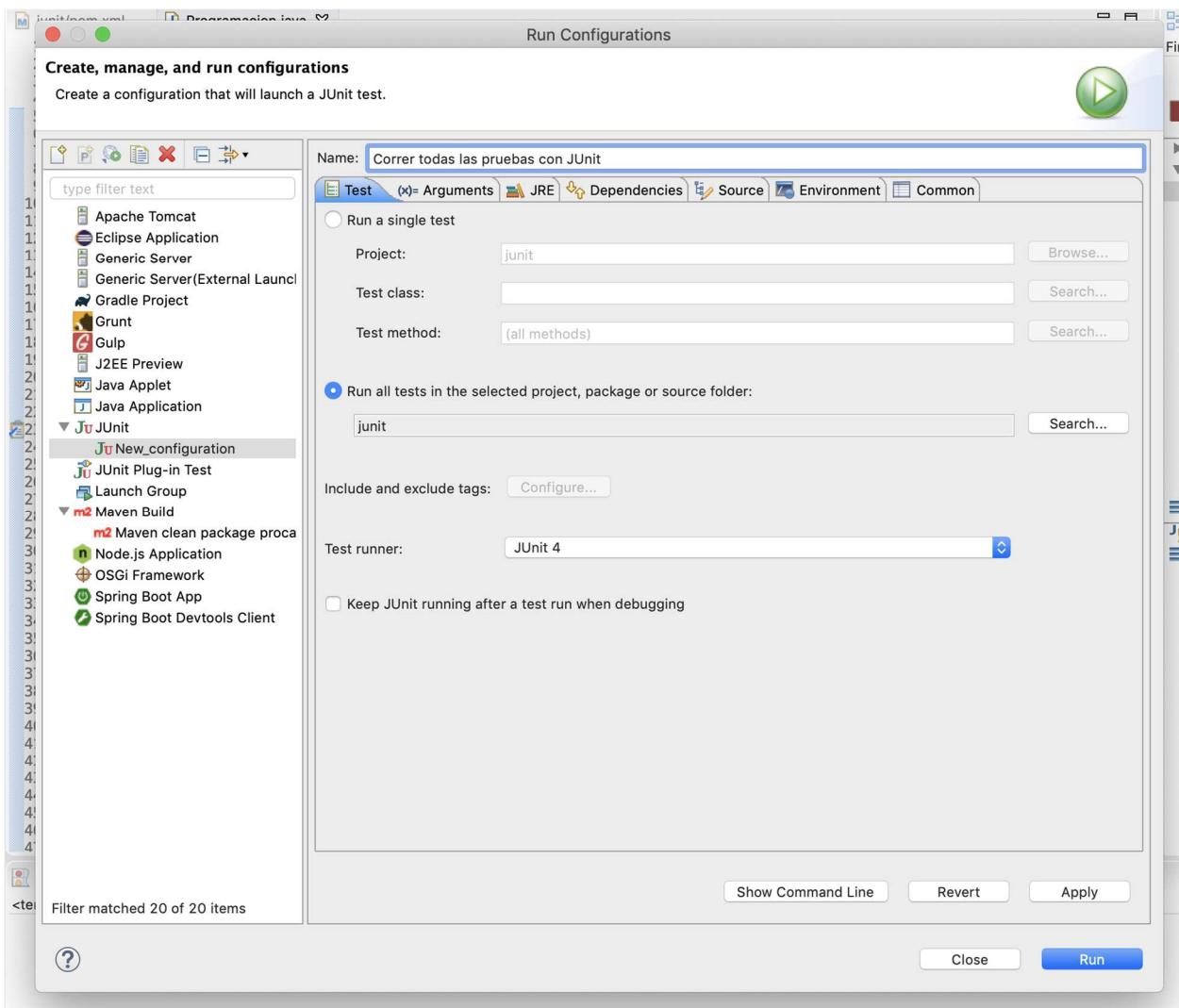


Fig. 34: Ejecución de todas las pruebas disponibles

Al seleccionar la opción “Run” nuestra configuración quedará guardada y además se mostrarán los resultados de la ejecución dentro de la vista “JUnit” que hemos abierto anteriormente, con la falla esperada:

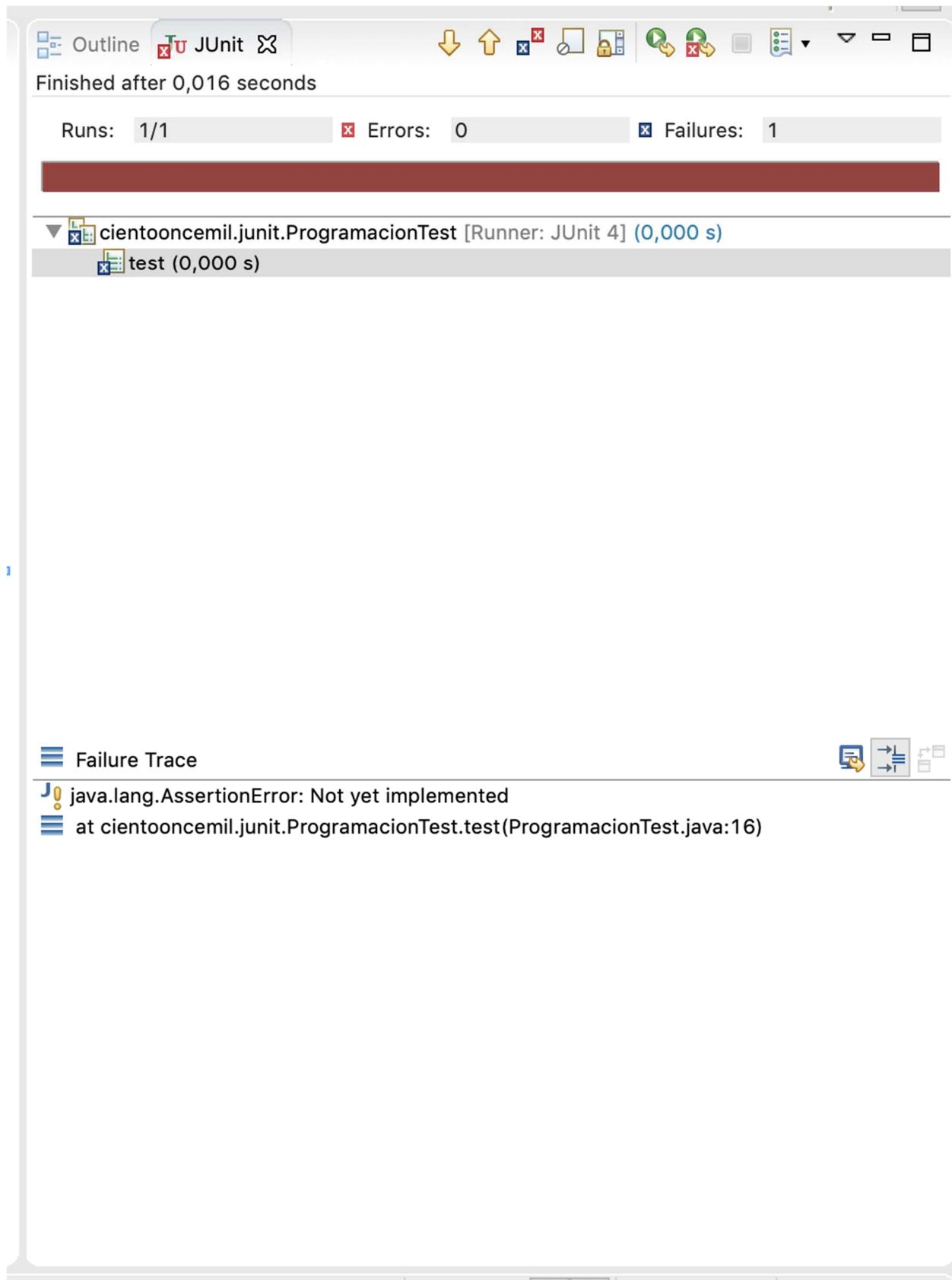


Fig. 35: Resultado de la Ejecución con las fallas esperadas

De esta forma ingresamos directo al ciclo “Red, Green, Refactor” que propone TDD, con nuestro primer RED bien grande.

Comencemos entonces a escribir nuestras pruebas, las que deberá luego pasar nuestra funcionalidad, una vez implementada.

Como primer paso completamos el método `setUp()` para inicializar una Programación con fechas controladas, las que hemos definido antes en nuestra tabla modelo:

```
private Programacion programacion;

@Before

public void setUp() {
    // creamos las fechas de inicio y fin
    // INICIO 17/10/2016 00:00:00
    Calendar inicio = Calendar.getInstance();
    inicio.set(2016, 10, 17, 0, 0, 0);
    inicio.set(Calendar.MILLISECOND, 0);

    // FIN 23/10/2016 23:59:59
    Calendar fin = Calendar.getInstance();
    fin.set(2016, 10, 23, 23, 59, 59);
    fin.set(Calendar.MILLISECOND, 999);

    Calendar hoy = Calendar.getInstance();
    hoy.set(2016, 10, 10, 18, 0, 0);
    hoy.set(Calendar.MILLISECOND, 0);

    programacion = new Programacion(inicio.getTime(),
fin.getTime(), hoy.getTime());
}
```

Ahora bien debemos escribir nuestra primera prueba, en la que vamos a verificar que la programación se encuentre vigente en una fecha anterior al inicio. Si consideramos los valores extremos como en nuestra tabla modelo, debemos probar con la fecha 16/10/2016 a las 23:59:59, el último segundo en el que la programación no está vigente antes de su fecha de inicio:

```

@Test
public void probarEstaVigenteAntes () {
    // probamos en una fecha anterior al inicio
    Calendar antes = Calendar.getInstance();
    antes.set(2016, 10, 16, 23, 59, 59);
    assertFalse(programacion.estavigente(antes.getTime()));
}

```

El componente principal de las pruebas son las aserciones, que se definen como condiciones que nuestra funcionalidad debe cumplir. En este caso estamos indicando a JUnit que debe verificar que la invocación al método `estaVigente()` con la fecha indicada **debe** retornar **false**, que es nuestro valor esperado.

Haremos lo mismo para el resto de los métodos de prueba en base a los casos que planteamos al comienzo de esta sección:

```

@Test
public void probarEstaVigenteAlInicio () {
    // probamos el mismo momento de inicio
    Calendar inicio = Calendar.getInstance();
    inicio.set(2016, 10, 17, 0, 0, 0);
    inicio.set(Calendar.MILLISECOND, 0);
    assertTrue(programacion.estavigente(inicio.getTime()));
}

@Test
public void probarEstaVigenteDurante () {
    // probamos entre las fechas de inicio y fin
    Calendar durante = Calendar.getInstance();
    durante.set(2016, 10, 20, 0, 0, 0);
    assertTrue(programacion.estavigente(durante.getTime()));
}

@Test
public void probarEstaVigenteAlFin () {
    // probamos el mismo momento de fin
    Calendar fin = Calendar.getInstance();

```

```
fin.set(2016, 10, 23, 23, 59, 59);
fin.set(Calendar.MILLISECOND, 999);
assertTrue(programacion.estavigente(fin.getTime()));
}

@Test
public void probarEstavigenteDespues () {
    // probamos despues del fin
    Calendar despues = Calendar.getInstance();
    despues.set(2016, 10, 24, 0, 0, 0);
    assertFalse(programacion.estavigente(despues.getTime()));
}
```

Al ejecutar nuestras pruebas obtenemos el siguiente resultado:

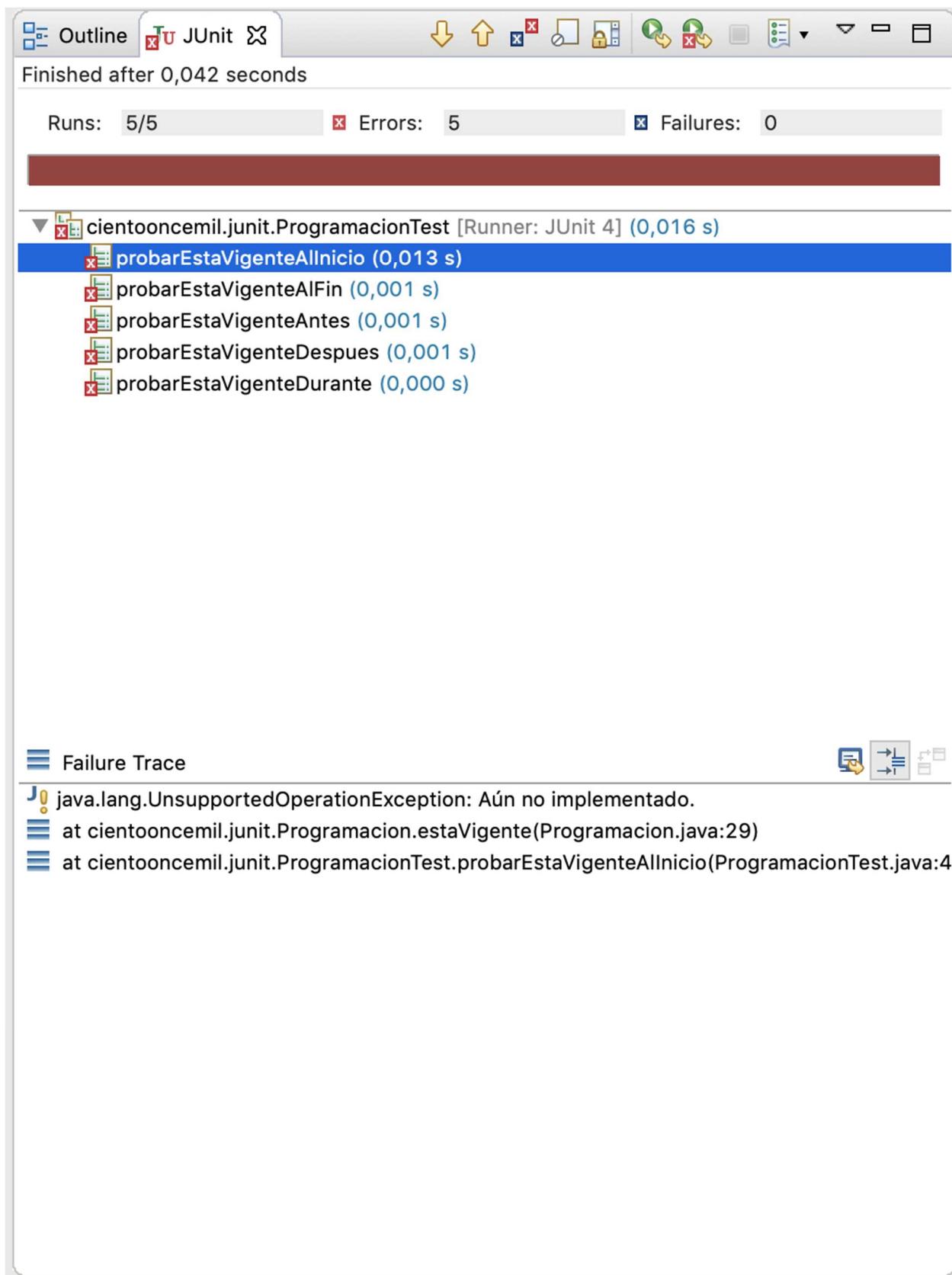


Fig. 36: Resultado de la Ejecución de las pruebas

¡Ninguna de nuestras pruebas pasa! Esto es completamente esperable, considerando que aún no hemos escrito una sola línea de código de la implementación de nuestro método `estaVigente()`.

Como siguiente paso debemos comenzar su implementación, por ejemplo con el siguiente código:

```
public boolean estaVigente(Date fecha) {  
    Calendar cuando = Calendar.getInstance();  
    cuando.setTime(fecha);  
  
    Calendar inicio = Calendar.getInstance();  
    inicio.setTime(fechaInicio);  
  
    Calendar fin = Calendar.getInstance();  
    fin.setTime(fechaFin);  
  
    return inicio.compareTo(cuando) < 0 && fin.compareTo(cuando) > 0;  
}
```

Y ejecutamos nuestras pruebas:

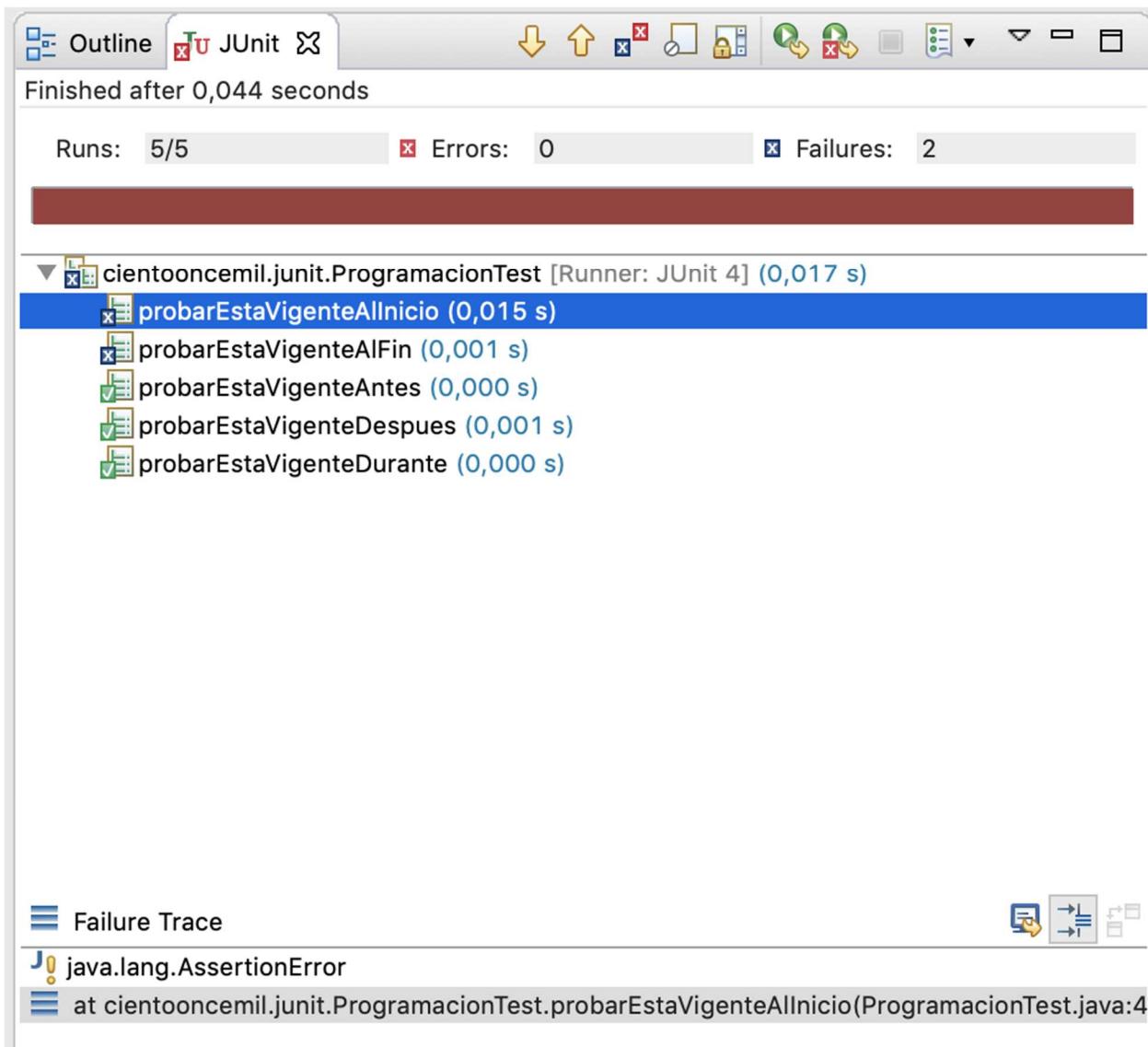


Fig. 37: Resultado de la Ejecución de las pruebas

Nuestras pruebas para los momentos antes, durante y después pasan flamantemente, pero ¿qué sucede en los extremos? Podemos ver que en el momento del inicio y del fin de la programación, que deberían estar incluidos en la vigencia, no están devolviendo el valor esperado.

Analizando nuevamente nuestra implementación, podemos ver que en la comparación de fechas hemos utilizado los operadores `< y >`, los cuales no incluyen los casos exactamente iguales. Probemos entonces reemplazar los operadores:

```
public boolean estaVigente(Date fecha) {
    // ...
    return inicio.compareTo(cuando) <= 0 && fin.compareTo(cuando) >= 0;
}
```

Y ejecutemos nuestras pruebas nuevamente:

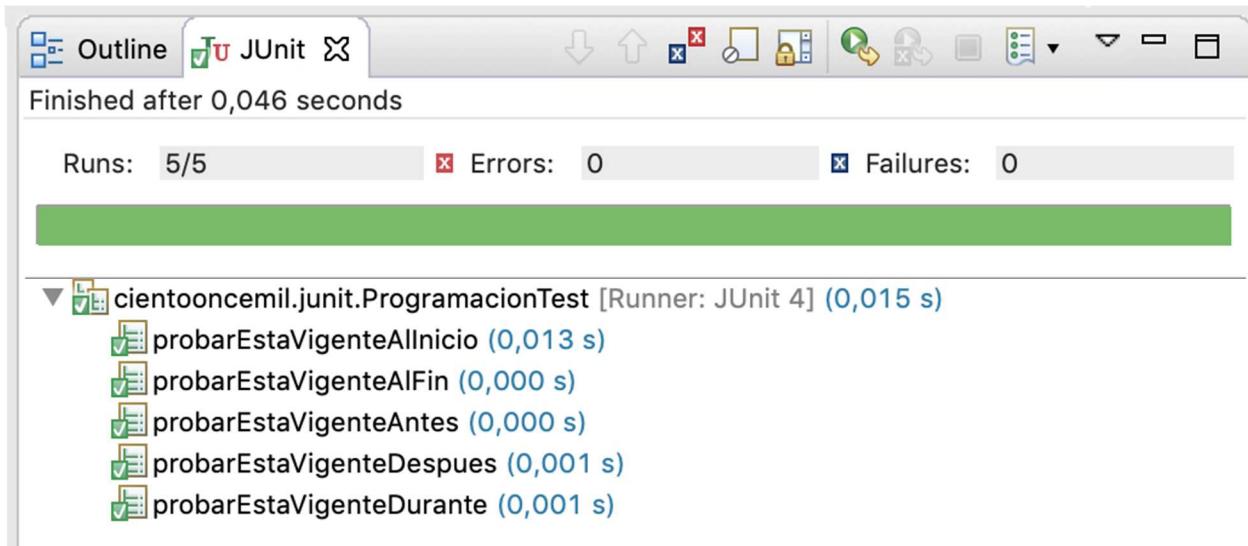


Fig. 38: Resultado de la Ejecución de las pruebas, con todas las pruebas corridas exitosamente

¡Un éxito rotundo! Ahora que hemos conseguido el color verde, debemos aplicar un ciclo de “Refactor” a nuestro código, es decir, realizar algunas mejoras para incrementar la calidad de nuestro producto. Por ejemplo podríamos agregar comentarios y el JavaDoc correspondiente, obteniendo el siguiente resultado final:

```
/**
 * Comprobamos si la fecha actual se encuentra dentro del periodo de
 * vigencia de esta Programacion (fecha de inicio y fin), consideramos
 * tambien que la fecha de inicio y fin tambien estan en el periodo de
 * vigencia de la Programacion
 *
 * @param fecha
 * @return
 */
public boolean estaVigente(Date fecha) {
    Calendar cuando = Calendar.getInstance();
    cuando.setTime(fecha);

    Calendar inicio = Calendar.getInstance();
    inicio.setTime(fechaInicio);

    Calendar fin = Calendar.getInstance();
    fin.setTime(fechaFin);
```

```
// comparamos contra la fecha de inicio y fin de la programacion  
// utilizamos los operadores <= y >= para asegurarnos de que  
// ambos extremos de fecha (inicio y fin) sean incluidos en el chequeo  
return inicio.compareTo(cuando) <= 0 && fin.compareTo(cuando) >= 0;  
}
```

¡Pero no tan rápido! Antes de continuar con otra porción de funcionalidad debemos comprobar que el ciclo de “Refactor” que hemos aplicado no haya interferido con nuestra funcionalidad, es decir que no haya incorporado algún defecto. Para ello sólo necesitamos volver a ejecutar nuestro ciclo de pruebas y, en caso de que alguna prueba falle, volver al inicio del “Red, Green, Refactor”.

Referencia de Figuras y Tablas

Figuras

Fig. 1- Funcionamiento general del Lenguaje Java	6
Fig. 2- Ejemplo de herencia y polimorfismo con la Jerarquía de clases de CuentaBancaria.....	36
Fig. 3- Ejemplo de interfaces – Diagramación de Torneos	37
Fig. 4- Resaltado de comentarios TODO en la IDE Eclipse.....	40
Fig. 5-. Documentación JavaDoc generada automáticamente en base a comentarios	40
Fig. 6- Representación de nodos en una lista ordenada	50
Fig. 7- Jerarquía de Excepciones en el lenguaje Java	55
Fig. 8- La API de Java en contexto	60
Fig. 9- Estructura de paquetes general para la API de Java 8.....	61
Fig. 10- Creación de un nuevo proyecto Maven con Spring Tool.....	65
Fig. 11- Elección de la ubicación del proyecto.....	66
Fig. 12- Selección de un arquetipo para el proyecto.....	67
Fig. 13- Especificación de los datos de identificación de nuestro proyecto el proyecto	68
Fig. 14- Estructura de inicialización del proyecto	69
Fig. 15- Actualización del proyecto con el menú contextual.....	71
Fig. 16- Interfaz web de generación de proyectos	72
Fig. 17- Selección de Dependencias	73
Fig. 18- Importar proyecto desde el sistema de archivos	74
Fig. 19- Estructura del proyecto	75
Fig. 20- Ejecución del proyecto desde Boot Dashboard	75
Fig. 21- Visualización de la salida que se muestra al iniciar la aplicación de Spring Boot.....	76
Fig. 22- Especificación de los datos de conexión de la Base de Datos MariaDB	76
Fig. 23- Visualización de la salida que se muestra al iniciar exitosamente la aplicación de Spring Boot.	77
Fig. 26- Visualización de la salida al ingresar la dirección http://localhost:8080/pedidos/nuevo.....	79
Fig. 27- Desarrollo Conducido por Pruebas (TDD).....	86
Fig. 28- Creación de un nuevo caso de prueba con JUnit.....	89
Fig. 29- Asignación de un nombre para el caso de prueba con JUnit.....	90
Fig. 30- Vista de JUnit dentro del IDE que nos permitirá obtener un reporte visual de los resultados...	92
Fig. 31- Vista de JUnit	93
Fig. 32: Ventana que se muestra	94
Fig. 33: Creación de una configuración para ejecutar pruebas.....	95

Fig. 34: Ejecución de todas las pruebas disponibles.....	96
Fig. 35: Resultado de la Ejecución con las fallas esperadas	97
Fig. 36: Resultado de la Ejecución de las pruebas.....	101
Fig. 37: Resultado de la Ejecución de las pruebas.....	103
Fig. 38: Resultado de la Ejecución de las pruebas, con todas las pruebas corridas exitosamente	104

Tablas

Tabla 1 – Tipos de básicos de datos en JAVA	12
Tabla 2 – Tipos de operadores aritméticos en JAVA	15
Tabla 3 – Tipos de operadores lógicos en JAVA	15
Tabla 4 – Operadores de Asignación en JAVA.....	17
Tabla 5 – Tags utilizados en Javadoc	38
Tabla 6. Extracto del JavaDoc para la clase ArrayList.....	53
Tabla 7. Extracto del JavaDoc para la clase Stack.....	54

Fuentes de Información

- **Belmonte Fernández Oscar** - Introducción al lenguaje de programación Java. Una guía básica
- **Sánchez Asenjo Jorge** – Programación Básica en Lenguaje Java. (<http://www.jorgesanchez.net> Año 2009)
- **García de Jalón Javier, Rodríguez SEP, José Ignacio, Mingo Iñigo, Imaz Aitor, Brazález Alfonso, Larzabal Alberto, Calleja Jesús, García Jon** – Aprenda Java como si estuviera en primero (Escuela Superior de Ingenieros Industriales, Universidad de Navarra Año 1999)
- **Jurado Carlos Blé** - Diseño Ágil con TDD Primera Edición, www.iExpertos.com (Año 2010)
- <http://java-white-box.blogspot.com.ar/2012/08/javadoc-que-es-el-javadoc-como-utilizar.html>
- <http://www.ecured.cu/Javadoc>
- http://www.ciberaula.com/articulo/listas_en_java
- http://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=603:interface-list-del-api-java-clases-arraylist-linkedlist-stack-vector-ejemplo-con-arraylist-cu00917c&catid=58:curso-lenguaje-programacion-java-nivel-avanzado-i&Itemid=180
- <https://geekytheory.com/tutorial-14-java-swing-interfaces-graficas/>
- <https://www.oracle.com/technetwork/es/articles/java/paquete-java-time-2390472-esa.html>
- <https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html>
- <http://blog.eddumelendez.me/2016/07/conociendo-la-nueva-date-api-en-java-8-parte-i/>
- <http://blog.eddumelendez.me/2016/07/conociendo-la-nueva-date-api-en-java-8-parte-ii/>
- <https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html>
- <https://medium.com/el-acordeon-del-programador/bigdecimal-no-mas-errores-de-redondeo-31954f276f8d>
- http://www.javamexico.org/blogs/luxspes/por_que_usar_bigdecimal_y_no_double_para_calculos_aritmeticos_financieros
- <https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html>
- <https://www.journaldev.com/16409/java-bigdecimal>
- <https://docs.oracle.com/javase/7/docs/api/java/math/RoundingMode.html>
- <http://java-white-box.blogspot.com/2014/05/junit-introduccion-primeros-pasos-con.html>
- <https://www.genbeta.com/desarrollo/spring-framework-introduccion>
- <http://www.itech.ua.es/j2ee/publico/spring-2012-13/sesion01-apuntes.html>
- <http://blog.eddumelendez.me/2016/09/introduccion-a-spring-boot/>
- <http://panamahitek.com/que-es-maven-y-para-que-se-utiliza/>
- <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Introducci%C3%B3n>
- <https://www.laurachuburu.com.ar/tutoriales/que-es-jquery-y-como-implementarlo.php>
- <https://www.jquery.com/>
- <https://www.npmjs.com/search?q=keywords:jquery-plugin>
- <https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html>
- https://en.wikipedia.org/wiki/Test-driven_development