

# PEC 2

## Clases y objetos

UOC

### Información relevante:

- Fecha límite de entrega: 7 de noviembre de 2022.
- Peso en la nota de EC: 20%.

# Contenido

<b>Información docente</b>	<b>3</b>
Prerrequisitos	3
Objetivos	3
Resultados de aprendizaje	3
<b>Enunciado</b>	<b>4</b>
Tarea 1 - Proyecto Gradle	4
Creación de un proyecto Gradle en IntelliJ	5
Testing con JUnit + Gradle	9
Ejercicio 1 (5 puntos)	11
Ejercicio 2 (1 punto)	17
Ejercicio 3 (1 punto)	18
Ejercicio 4 (3 puntos)	20
<b>Formato y fecha de entrega</b>	<b>22</b>

## Información docente

Esta PEC está vinculada con el módulo teórico “Abstracción y encapsulación” de los apuntes de la asignatura. Léelo antes de empezar la PEC.

### Prerrequisitos

Para hacer esta PEC necesitas:

- Tener adquiridos los conocimientos de la PEC 1. Para ello te recomendamos que mires la solución que se publicó en el aula y la compares con la tuya.

### Objetivos

Con esta PEC el equipo docente de la asignatura busca que:

- Sepas diferenciar entre objeto y clase, siendo capaz de abstraer la clase a partir de un conjunto de objetos/entidades reales que están relacionados entre sí.
- Veas el potencial de la encapsulación.
- Uses Java como un lenguaje de programación orientado a objetos.
- Crees programas bien documentados que faciliten su mantenimiento posterior.
- Entiendas qué es una excepción y cómo se utiliza en Java.
- Conozcas las diferencias entre las clases `String`, `StringBuilder` y `StringBuffer`.
- Acabes de familiarizarte con la dinámica de la asignatura.

### Resultados de aprendizaje

Con esta PEC debes demostrar que eres capaz de:

- Codificar una clase a partir de unos requisitos y su representación como diagrama de clases.
- Entender la diferencia entre los diferentes modificadores de acceso.
- Comprender qué consecuencias tiene declarar un miembro como `static`.
- Gestionar casos anómalos, incorrectos e indeseados mediante excepciones.
- Documentar con Javadoc un programa.
- Saber utilizar las clases `String` y `StringBuilder`.
- Utilizar test unitarios para determinar que un programa es correcto.
- Usar con cierta soltura un entorno de desarrollo integrado (IDE) como IntelliJ.

## Enunciado

Esta PEC contiene 4 ejercicios evaluables y 1 tarea no evaluable. Debes entregar tu solución de los 4 ejercicios evaluables (ver el último apartado).



Debido a que las actividades están encadenadas (i.e. para hacer una se debe haber comprendido la anterior), **es altamente recomendable hacer las tareas y ejercicios en el orden en que aparecen en este enunciado.**

## Tarea 1 - Proyecto Gradle

Antes de empezar debes:

Leer los apartados 3.1, 3.2 y 3.5 de la Guía de Java.

En la PEC anterior vimos cómo crear un proyecto Java con IntelliJ y cómo añadirle la librería de JUnit 5. En esta PEC vamos a entrar en el mundo de la “automatización de *builds*” (construcciones) y la gestión de dependencias, y lo haremos a través de la herramienta Gradle, la cual nos permite automatizar la construcción de nuestro código.

No vamos a hacer un tutorial completo de Gradle, ya que para eso necesitaríamos un curso entero, y no es el objetivo principal de esta asignatura. No entraremos en todas las tareas que nos permiten automatizar procesos dentro de proyectos de *software* (i.e. compilar, depurar, testing, empaquetado, distribución y *deployment*) o en la creación de tareas propias y complejas para automatizar ciertos procesos. Veremos la ejecución de ciertas tareas básicas y nos centraremos especialmente en la gestión de dependencias.

Cuando hablamos de la gestión de dependencias nos referimos a lo que has hecho en la PEC1 cuando añadías manualmente la librería JUnit. Esto en la actualidad no es una práctica habitual y, en su lugar, se utiliza siempre un gestor (Gradle, Maven, etc.) para construir un proyecto (normalmente llamado *build*) y gestionar sus dependencias. Se puede decir que actualmente ya no podemos vivir sin la ayuda de estas herramientas en proyectos profesionales.

Por ejemplo, antes, si una persona entraba a trabajar en una empresa, ésta se descargaba el proyecto Java en su IDE, y tenía que importar todas las dependencias a mano (como hemos hecho en la PEC1), y esto suponía perder una cantidad nada despreciable de tiempo en configurar el entorno de trabajo antes de poder empezar a trabajar realmente con el código de la aplicación. En la actualidad, al trabajar con herramientas del estilo de Gradle, el nuevo programador tan sólo descargará el proyecto, ejecutará la tarea de compilación, y Gradle ya se encargará de resolver y descargar todas las dependencias necesarias (previamente indicadas en el fichero de configuración de Gradle) para que el proyecto funcione correctamente. En cuestión de segundos, tenemos el entorno montado y listo para

empezar a desarrollar, sin arduas tareas de configuración. Piensa que a menudo una dependencia (i.e. una librería) de nuestro proyecto puede depender de otra librería que también hay que incluir. Gradle (o Maven, etc.) también se encarga de realizar esta tarea de rastreo y descarga de dependencias de dependencias.

Como características principales de Gradle podemos destacar:

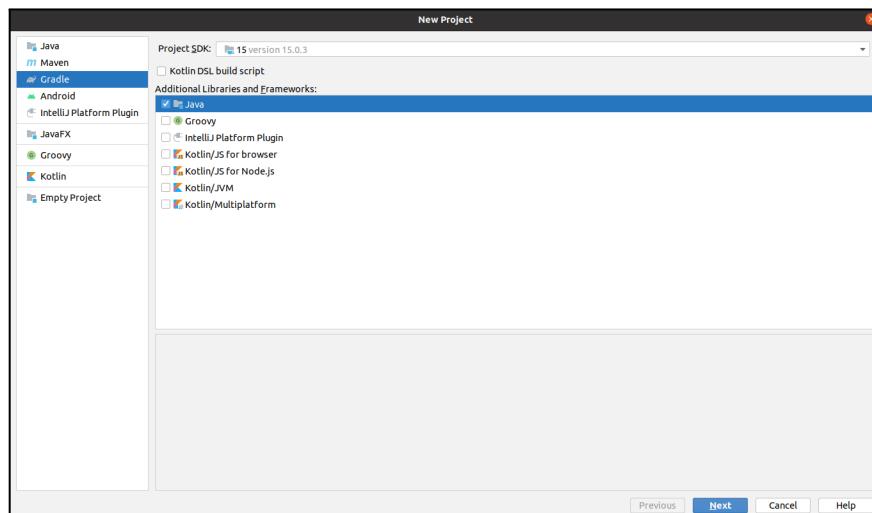
- Flexibilidad: soporta varios lenguajes de programación, a diferencia de Maven que es solo para aplicaciones Java.
- Utiliza DSL (*Domain Specific Language*), lo cual proporciona un lenguaje sencillo y claro a la hora de construir el fichero de configuración, a diferencia de Maven que está basado en XML.
- Gestión del ciclo de vida: añade la capacidad de soportar todo el proceso de vida del *software* (desde la compilación, pruebas, análisis estadístico e implementación).
- Sistema de gestión de dependencias muy estable.
- Basado en “tareas” (en inglés, *tasks*).

Aquí hay una serie de videos cortos tipo “tutoriales” sobre el funcionamiento de Gradle:  
<https://www.makigas.es/series/automatizando-con-gradle>

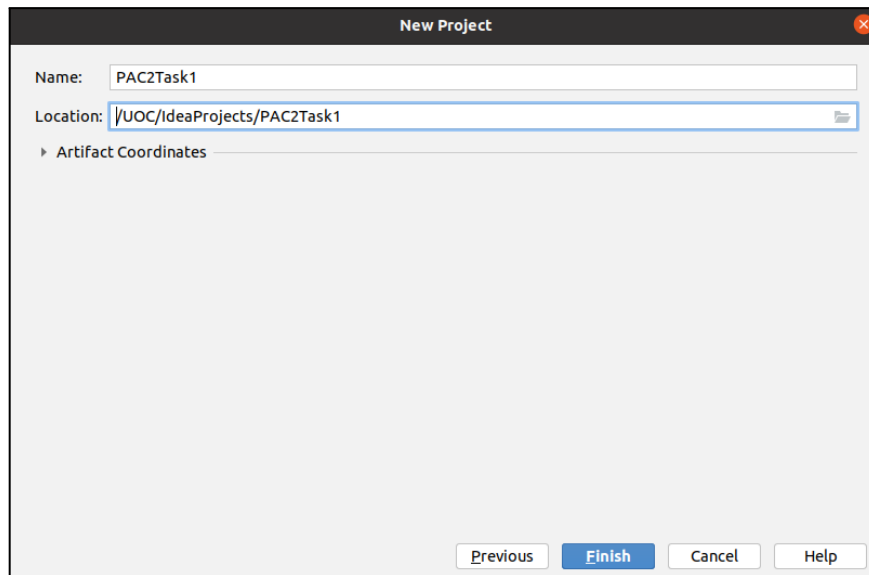
Como hemos comentado, no nos vamos a centrar en hacer un curso de Gradle, y vamos a ir directamente a cómo crear un proyecto Gradle desde el IDE IntelliJ IDEA.

## Creación de un proyecto Gradle en IntelliJ

**Crea un nuevo proyecto** con el botón `New Project` de la pantalla de bienvenida de IntelliJ o desde `File→New→Project` una vez abierto IntelliJ. Una vez clicado el botón para crear un nuevo proyecto, IntelliJ te mostrará una ventana de configuración del proyecto (como la que se muestra en la imagen siguiente). Por defecto IntelliJ selecciona como tipo de proyecto `Java`. Ahora debemos cambiar y seleccionar `Gradle` en la barra lateral, y como lenguaje de programación seleccionamos `Java`. Cliquea en el botón `Next`.

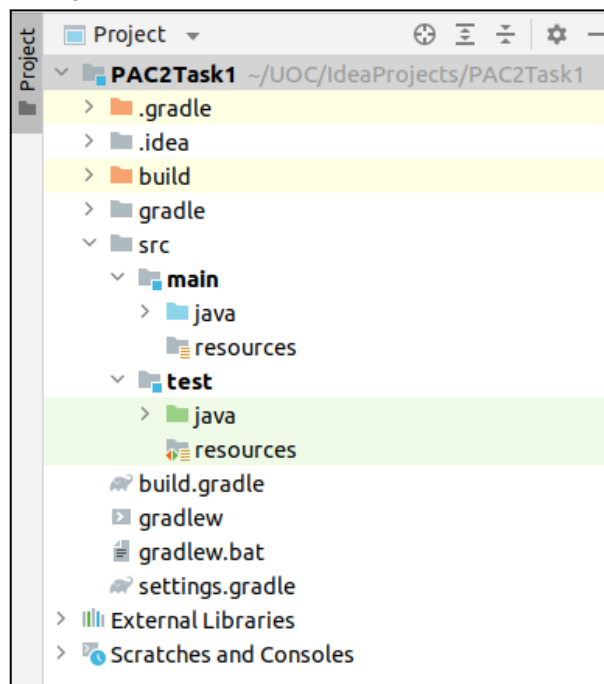


En la siguiente pantalla, introduce el nombre del proyecto, en este caso, **PAC2Task1**. Verás que ya te selecciona automáticamente tu carpeta *workspace* donde tienes el resto de proyectos (i.e. *IdeaProjects*). Clica en el botón **Finish** para crear este proyecto Gradle-Java.



Una vez creado, verás que, en la parte inferior derecha, IntelliJ está ejecutando tareas: descargando el código de Gradle e integrándolo en el proyecto, así como configurando el proyecto Gradle.

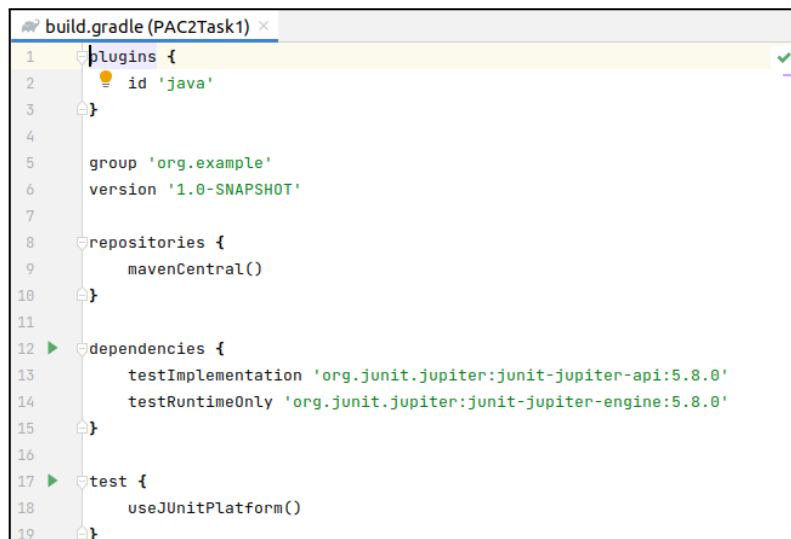
Tras configurar y crear el proyecto **PAC2Task1**, puedes ver en el *project tool window* (lateral izquierdo) que la estructura o árbol de directorios de un proyecto Gradle (ver siguiente imagen) difiere de la del proyecto Java estándar creado en la PEC anterior.



En él verás la carpeta `src`, la cual ahora contiene dos subcarpetas: (1) `main`, en cuya subcarpeta `main/java` será donde pondrás el código fuente de tu programa, y (2) `test`, donde en la subcarpeta `test/java` será donde pondrás los test unitarios JUnit. Fíjate que ahora cada carpeta ya tiene asignado el color correspondiente a `Sources Root` (azul) y `Test Sources Root` (verde). Esta manera de estructurar un proyecto es heredada de Maven. De hecho, Gradle no obliga ni crea esta estructura, es IntelliJ quien la crea. Maven ha sido (y sigue siendo) tan utilizado por tantos profesionales que su estructura se ha convertido en un estándar *de facto* (por eso IntelliJ la adopta para los proyectos Gradle).

Por otro lado, ves una carpeta de Gradle, donde está el código de esta herramienta, así como el fichero `gradlew` (versión Windows `-.bat-` y Linux/MacOS) que es el ejecutable de Gradle y el fichero de configuración de Gradle `build.gradle` (con el icono de un elefante).

El fichero `build.gradle` (se podría llamar diferente, pero éste es el nombre que se utiliza por convenio, así pues, es mejor respetarlo) es el que vamos a utilizar para configurar y gestionar los *builds* (i.e. proyectos) y sus dependencias. Analicémoslo brevemente:



```

1  plugins {
2      id 'java'
3  }
4
5  group 'org.example'
6  version '1.0-SNAPSHOT'
7
8  repositories {
9      mavenCentral()
10 }
11
12 dependencies {
13     testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.0'
14     testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.0'
15 }
16
17 test {
18     useJUnitPlatform()
19 }

```

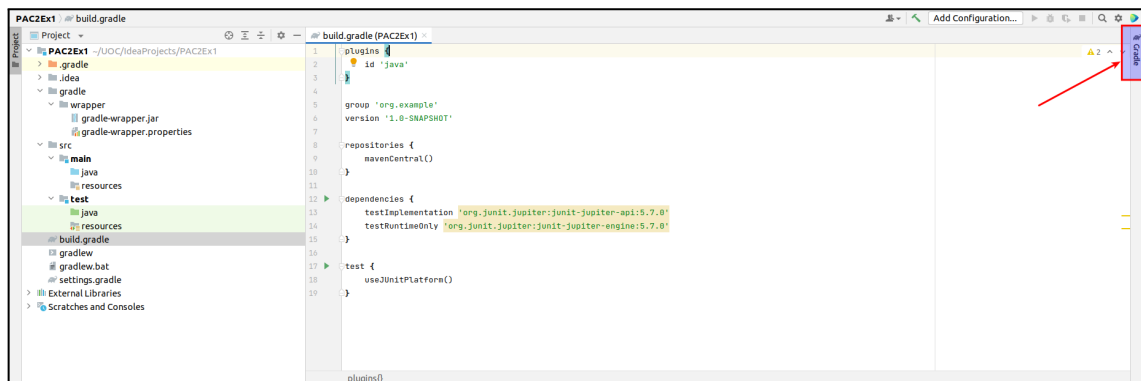
- `plugins`: con el valor `id 'java'` está indicando que estamos utilizando Java, lo que hace que añada tareas específicas de la JVM (compilación, compilación de test, generación de JAR, *clean*, etc.)
- `group`: es el espacio de nombres (*namespace*) para identificar el artefacto o los artefactos producidos por una construcción (*build*). Suele ser el dominio de la empresa en la que trabajamos, p.ej. `edu.uoc`. Su valor puede indicarse al crear el proyecto si se despliega la opción `Artifact Coordinates`.
- `version`: versión del artefacto. También puede indicarse mediante `Artifact Coordinates`.
- `repositories`: por defecto se pone el valor `mavenCentral()`. Este valor indica que las dependencias se obtienen del repositorio central de Maven 2. Puedes ver las librerías incluidas en Maven usando el siguiente buscador: <https://search.maven.org/>.

- **dependencies:** en este apartado se añaden las librerías que el proyecto necesita para funcionar. En este caso se ha añadido únicamente la librería de test JUnit.
- **test:** con valor `useJUnitPlatform()`. Con el valor anterior se habilita JUnit Platform para ejecutar los test (pruebas). Podríamos tener varias librerías de test descargadas como dependencias y aquí diríamos cuál utilizar.

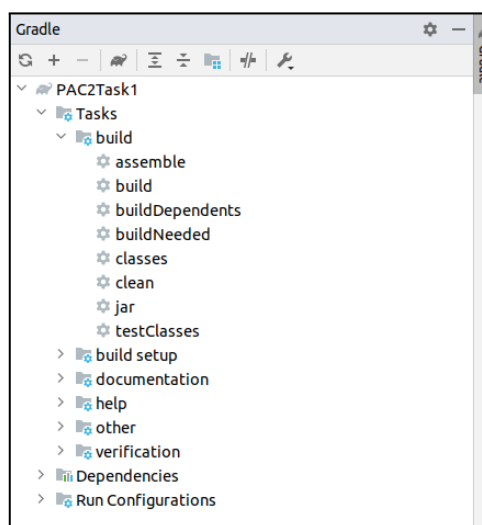


**Nota:** En la asignatura nos centraremos en las dependencias y en ejecutar las tareas (*task*) de *build* cuando añadamos nuevas dependencias al proyecto. Pero es interesante conocer mínimamente el porqué del resto de entradas del fichero de configuración.

Por último, ahora desde IntelliJ, al tener un proyecto Gradle, te aparece una nueva pestaña para poder ejecutar las diferentes tareas de Gradle (no hace falta ejecutar Gradle mediante línea de comandos, ya que lo tenemos integrado en IntelliJ).



Al desplegar esta pestaña, puedes ver las tareas Gradle que puedes realizar (ver siguiente imagen). Las tareas disponibles dependen, entre otras cosas, de los *plugins* que se indiquen en el fichero `build.gradle`.



Ya tenemos configurado un proyecto con Gradle listo para empezar a trabajar. Aquí te dejamos un vídeo-tutorial oficial de JetBrains sobre Gradle+IntelliJ que explica más detalladamente lo que acabamos de hacer en esta guía: <https://youtu.be/6V6G3RyxEMk>



## Testing con JUnit + Gradle

Vamos a ver ahora cómo ejecutar JUnit desde Gradle. Evidentemente, como estamos usando un IDE, podemos ejecutar los JUnit como hemos hecho en la PEC1. Pero ya que hemos añadido Gradle, vamos a ver un ejemplo de cómo ejecutar la tarea de `test` para lanzar nuestros test unitarios (y así no centramos únicamente en la gestión de dependencias). Para ello, sigue los pasos que se detallan a continuación:

1. En el directorio `src/main/java`, crea el *package* `edu.uoc.pac2` donde codificarás tus clases Java. Para ello haz botón derecho sobre el directorio `java` y escoge `New→Package`.
2. En el *package* que has creado, crea la clase `PAC2Task1` (botón derecho sobre el *package* y `New→Java Class`) donde tendrás que copiar el siguiente código:

```
package edu.uoc.pac2;

public class PAC2Task1 {
    public static int add(int a, int b){
        return a + b;
    }
}
```

3. Una vez tienes la implementación anterior realizada, vamos ahora a crear la clase que hará de *test case* y que incluirá los *unit tests* de la clase `PAC2Task1`. En el directorio `src/test/java`, crea el *package* `edu.uoc.pac2` donde tendremos nuestras clases Java de test.
4. En el *package* que acabas de crear, crea la clase `PAC2Task1Test` donde testearás la implementación del método `add` de la clase `PAC2Task1`. Copia el siguiente código en el fichero `PAC2Task1Test.java`:


```
package edu.uoc.pac2;

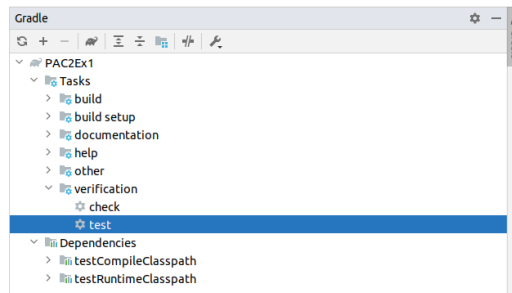
import org.junit.jupiter.api.Test;

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

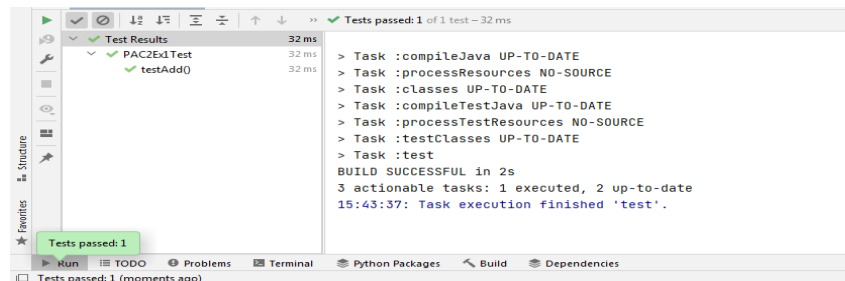
class PAC2Task1Test {

    @Test
    public void testAdd() {
        assertEquals(5, PAC2Task1.add(2, 3));
    }
}
```

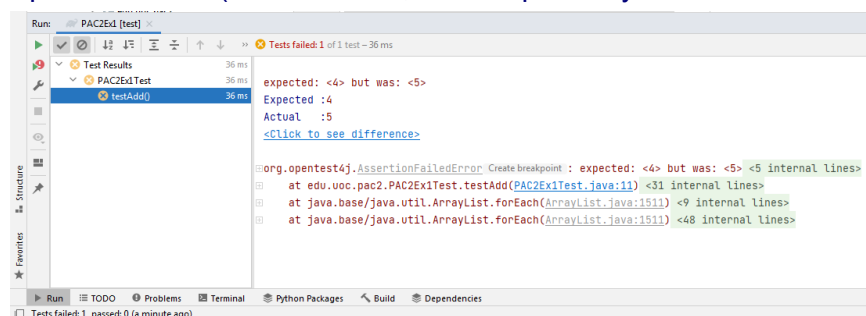
5. Ahora ya puedes ejecutar el test implementado en la clase `PAC2Task1Test`. Ves a la lista de tareas de Gradle (lateral derecho) y haz doble click en `Tasks`→`verification`→`test`, e IntelliJ ejecutará mediante la tarea Gradle los test unitarios (en este caso, tanto la tarea `test` como `check` hacen lo mismo. `Check` ejecuta todos los test y `test` únicamente los test unitarios). Como hemos comentado, puedes ejecutarlo como hemos explicado anteriormente en la PEC1, clicando en el botón derecho sobre la clase y accediendo a `Run`, o con los botones  que verás delante de la clase y los métodos que implementan los test unitarios. Pero de esta manera ya no estarías utilizando Gradle, estarías utilizando el IDE.



6. En la parte inferior del IDE, puedes ver el resultado de los test. Si pulsas en el botón con forma de *tick* (al lado del *play* verde), entonces se muestran todos los test que han sido ejecutados. Hay 2 posibles estados:
- **Test superado:** muestra un *tick verde* conforme el test ha sido superado satisfactoriamente.



- **Test no superado:** si modificas el valor esperado del test, por ejemplo, en vez de un 5, pones un 4, se mostrará que el test no ha sido superado junto con el detalle del error en la parte derecha (traza de error, valor esperado y valor actual del test).



## Ejercicio 1 (5 puntos)

Antes de empezar debes:

Leer los apartados del 3.1 al 3.6 de la Guía de Java y ver los 3 primeros videotutoriales del apartado **Java** del [sitio UOCoders](http://uocoders.com).

Leer el apartado 4.1 de la Guía de Java que habla sobre la clase `String`.

**Abre el proyecto PAC2Ex1 desde IntelliJ.** En el *package* `edu.uoc.pac2` del directorio `/src/test/java` está el fichero con los test unitarios que te proporcionamos. En el *package* `edu.uoc.pac2` del directorio `/src/main/java` tienes que crear y codificar la clase `Contact`, cuya representación en diagrama de clases UML es:



En este ejercicio vamos a codificar la clase `Contact`, la cual representa un contacto que podemos tener en la agenda de nuestro *smartphone*. Para la codificación de la clase `Contact` debes tener en cuenta las siguientes especificaciones/consideraciones:

- Los constructores deben inicializar los atributos cuyos valores no son indicados mediante parámetros con el valor por defecto que se indica en el diagrama de clases UML. El atributo `birthday` será de tipo `LocalDate` (en UML ponemos `Date` para referirnos a una fecha, luego cada lenguaje lo codifica a su manera).

- El valor del atributo `id` se obtiene asignándole el valor que en ese momento tiene `nextId`. Por ejemplo, si `nextId` es igual a 5, entonces el contacto que estamos instanciando (i.e. el objeto de tipo `Contact` que estamos creando) tendrá un `id` igual a 5. Una vez asignado el valor al atributo `id`, el valor de `nextId` se tiene que incrementar una unidad. La primera instancia de `Contact` tendrá `id = 0`. La inicialización del atributo `id` se debe hacer en el constructor y, siempre y cuando, no haya ningún error en los valores de los valores asignados al resto de los atributos de la clase.
- **Siempre que exista conflicto entre nombres**, sobre todo en los métodos *setter* (i.e. `setXXX`), **debes usar la palabra reservada `this`**. Si te fijas, el nombre del parámetro de todos los métodos *setter* se llama exactamente igual que el atributo de instancia que modifican. Si, por ejemplo, hacemos esto:

```
public void setPhone(String phone){
    phone = phone;
}
```

¿Cómo sabe el compilador (y nosotros) si el `phone` de la izquierda de la asignación es el atributo de la clase (o instancia/objeto) o es el parámetro del método `setPhone`? Y lo mismo ocurre con el `phone` del lado derecho. Así pues, para evitar problemas muchos programadores llaman al parámetro del método utilizando el nombre del atributo a modificar y añadiéndole una coletilla, p.ej. `New`:

```
private void setPhone(String phoneNew){
    phone = phoneNew;
}
```

Con esta solución no hay duda ni para el compilador ni para nosotros de qué es qué. A pesar de que podemos usar la solución anterior, lo más habitual (y mejor) es utilizar como nombres de los parámetros de los métodos exactamente los mismos que los de los atributos de la clase. Para resolver el conflicto de qué es qué se usa la palabra reservada `this`:

```
private void setPhone(String phone){
    this.phone = phone;
}
```

La palabra reservada `this` hace referencia al objeto (o instancia) que es del tipo de la clase que estamos codificando. Por lo tanto, gracias a `this`, estamos diciendo que en la asignación el `phone` de la izquierda es del objeto/clase y el `phone` de la derecha es el parámetro del método `setPhone`. Podemos decir que el `this` hace de coletilla e indica que aquello que lleva `this` pertenece al objeto/clase y no es ni un parámetro ni una variable declarados dentro del método. Por ejemplo, si tenemos un objeto de tipo `Contact` asignado a la variable `c1`, entonces cuando llamamos a

`setPhone` haciendo `cl.setPhone("933582340")`, el `this` que utilizamos dentro del método `setPhone` se refiere al objeto `cl`.

- Si el nombre que se quiere asignar al contacto (i.e. atributo `name`) tiene más de 20 caracteres, entonces `setName` no debe asignar dicho nombre y debe imprimir por pantalla el mensaje "[ERROR] Name cannot be longer than 20 characters".

En caso contrario, sí que se debe asignar el nombre al atributo `name`, teniendo en cuenta que el valor del nombre no debe contener espacios al principio ni al final del mismo, p.ej. `setName(" David ")` debe asignar el valor "David", no el valor " David ". La comprobación de si el valor que se quiere asignar tiene 20 o menos caracteres se debe hacer una vez eliminados los espacios iniciales y finales.

- El método `setSurname` tiene dos firmas, es decir, está sobrecargado. El primero de ellos, con un sólo parámetro, asigna el valor pasado como parámetro sin espacios al inicio y al final, con todos los caracteres en mayúsculas y, si hay espacios entre las palabras que componen el apellido, estos serán sustituidos por un único guión "-" entre palabras, p.ej. "Garcia Solorzano" debe ser "GARCIA-SOLORZANO". En cambio, la segunda firma permite indicar qué símbolo debe ir entre las palabras, p.ej. "-", "=", etc.
- El método `getAge` devuelve los años que hay entre el año actual y el año de nacimiento del contacto. Piensa que el "año actual" depende de cuándo se ejecute el programa. Así pues, no pongas 2022 en *hardcoded*. Si el valor del atributo `birthday` es `null` (porque no sabemos la fecha de nacimiento del contacto), entonces `getAge` debe devolver -1.
- Un contacto sólo puede pertenecer a uno de los siguientes grupos: 'F' (*friends*), 'W' (*work*) u 'O' (*others*). Así pues, si se quiere asignar al contacto (i.e. atributo `group`) un valor diferente (incluidos 'f', 'w', y 'o'), entonces `setGroup` no debe asignar dicho valor y debe imprimir por pantalla el mensaje "[ERROR] Group is incorrect".

En caso contrario, sí que se debe asignar el valor al atributo `group`. p. ej: `setGroup('W')` debe asignar el valor 'W'.



**Pista:** Investiga la clase `String` para ver qué métodos te ofrece y así realizar operaciones como "poner todos los caracteres en mayúsculas" de una forma sencilla. Para detectar todos los espacios entre dos palabras, te recomendamos usar una **expresión regular**. Puedes encontrar más información sobre expresiones regulares en:

<https://www.javatpoint.com/java-regex>

También te puede ser útil usar estos dos testers:

<https://regex101.com/>

<https://www.freeformatter.com/java-regex-tester.html#before-output>

O también utiliza el tester que IntelliJ ofrece: presiona `Alt+Enter` cuando el cursor del teclado esté en una expresión regular y aparecerá una ventana flotante.

Investiga también acerca de la clase `LocalDate` de la API de Java para calcular periodos de tiempo en el método `getAge`.



**Nota:** En este punto del Ejercicio 2, tu código para la clase `Contact` debería superar todos los test que te proporcionamos.

Los *getters/setters* de los atributos `phone` y `email` de la clase `Contact` no se prueban con el juego de test público que te proporcionamos. Así pues, a estos métodos se les pasará un test privado puntuable durante la corrección, y serás tú el que tendrás que implementar tu propio juego de test unitarios para comprobar el correcto funcionamiento de dichos métodos.



**Nota:** Debes incluir tus test dentro del fichero de test que te proporcionamos.

Para la codificación de estos métodos debes tener en cuenta las siguientes especificaciones/consideraciones:

- Si el teléfono que se quiere asignar al contacto (i.e. atributo `phone`) no cumple con un patrón establecido, `setPhone` no debe asignar dicho teléfono y debe imprimir por pantalla el mensaje `"[ERROR] Phone pattern is incorrect"`.

En caso contrario, sí que debe asignar el teléfono al atributo `phone`. El patrón que deben cumplir los teléfonos es el siguiente:

- La longitud del teléfono no puede ser mayor a 12.
- El número de teléfono puede constar de sólo 9 dígitos.
- Los 9 dígitos del teléfono pueden ir precedidos de un prefijo internacional cuyo formato es un número de dos cifras precedido del símbolo `'+'`.

Algunos ejemplos son:

```
"933582340" → correcto
"+34933582340" → correcto
"+34652668900" → correcto
"+346526689" → incorrecto
"+334933582340" → incorrecto
"93-3582340" → incorrecto
"334933582340" → incorrecto
"+4933582340" → incorrecto
"34+933582340" → incorrecto
"582340" → incorrecto
"+582340" → incorrecto
```

- Si el email que se quiere asignar al contacto (i.e. atributo `email`) no cumple con un patrón establecido, `setEmail` no debe asignar dicho email y debe imprimir por pantalla el mensaje "[ERROR] Email pattern is incorrect".

En caso contrario, sí que debe asignar el email al atributo `email`. El patrón que deben cumplir los emails es el siguiente:

- Un email puede tener cualquier longitud.
- Debe contener un único símbolo '@'.
- Antes del símbolo debe haber una cadena de texto formado exclusivamente por caracteres de la A a la Z (en mayúsculas, en minúsculas o una combinación de ambas) y el símbolo '.' (punto). No podrá contener números.
- El uso del punto es opcional. Es decir, un email no tiene porqué contener puntos.
- Todo email debe empezar por una letra del alfabeto (ya sea en mayúsculas o en minúsculas).
- Después de un punto debe haber al menos una letra del alfabeto (ya sea en mayúsculas o en minúsculas).
- Después del símbolo '@', debe haber una cadena de texto formado exclusivamente por caracteres de la A a la Z (en mayúsculas, en minúsculas o una combinación de ambas) y el símbolo '.' (punto).
- Después del símbolo '@' debe haber al menos un carácter de la A a la Z (en mayúsculas o en minúsculas), un punto y, después del punto, al menos un carácter de la A a la Z (en mayúsculas o en minúsculas). Así pues, el email no puede terminar en punto, pero sí contener subdominios.

Algunos ejemplos son:

`"dpoo@uoc.edu"` → correcto

`"d@d.d"` → correcto

`"DaVId.Garcia.soloRZanO@yahoo.es"` → correcto

`"d.a@d.d"` → correcto

`"uoccoders83@eimt.uoc.edu"` → incorrecto

`".@eimt.uoc.edu"` → incorrecto

`".a@eimt.uoc.edu"` → incorrecto

`"uoccoders@a."` → incorrecto

`"uoccoders.a"` → incorrecto



**Pista:** Para validar el patrón requerido para los parámetros `phone` y `email`, la forma más sencilla es usar un método de la clase `String` que permite comprobar si el texto coincide con una **expresión regular**.



**Requisito mínimo para evaluar este ejercicio:** La clase `Contact` debe pasar todos los test proporcionados.



**Nota:** El estudiante puede recibir una penalización de hasta **1 punto** de la nota obtenida en este ejercicio en función de la calidad del código proporcionado.

*(5 puntos: 3.5 pts. test proporcionados; 1.5 pts. test privados usados durante la corrección)*



## Ejercicio 2 (1 punto)

Antes de empezar debes:

Leer el apartado 3.12 de la Guía de Java que habla sobre excepciones.

Ver los videotutoriales sobre excepciones y cláusulas guarda que hay en [UOCoders](#).

Una vez realizada la tarea anterior debes:

- Abrir el proyecto **PAC2Ex2** con IntelliJ.
- Copiar todo el *package* `edu.uoc.pac2` del `src/main/java` del Ejercicio 1 en el directorio `src/main/java` del proyecto **PAC2Ex2**.
- Modificar los métodos `setName`, `setPhone`, `setEmail` y `setGroup` de la clase `Contact` para que en vez de imprimir por pantalla el mensaje de error, lance una `Exception` con el mensaje correspondiente.
- Ahora el método `getAge` en vez de devolver `-1` cuando `birthday` sea `null`, lanzará una excepción con el mensaje "[ERROR] Birthday is null".
- Hacer las modificaciones que creas convenientes/necesarias para trabajar con excepciones de la manera más eficiente dentro de la clase `Contact`.



**Requisito mínimo para evaluar este ejercicio:** La clase `Contact` debe pasar todos los test proporcionados.

**(1 punto: 1 punto test proporcionados)**

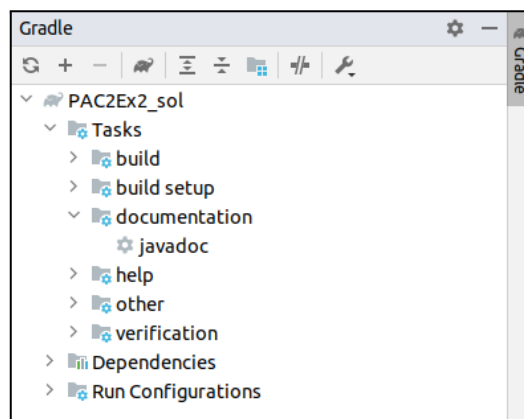
## Ejercicio 3 (1 punto)

Antes de empezar debes:

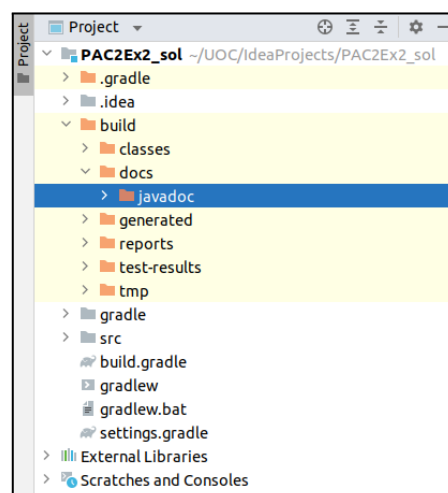
Leer el apartado 5.1 de la Guía de Java que habla sobre Javadoc.

En el proyecto **PAC2Ex2** documenta, con comentarios Javadoc, la clase **Contact** que has codificado en el Ejercicio 2 de manera que describan los elementos de dicha clase: **la clase en sí, sus atributos y métodos (independientemente de su modificador de acceso)**. Los comentarios pueden ser en español o inglés, pero te recomendamos que sean en inglés.

Una vez documentada la clase, vamos a utilizar la tarea de Gradle `documentation→javadoc` para generar la documentación (se podría generar usando directamente la opción de IntelliJ, pero ya que estamos usando Gradle, lo aprovechamos). Haz doble clic en esta tarea de Gradle (ver la siguiente imagen):



Esta tarea genera un directorio llamado `docs` dentro de la carpeta `build` del proyecto **PAC2Ex2** que incluye la documentación generada (dentro de la subcarpeta `javadoc`). La documentación que generes debe incluir la descripción de la clase **Contact** junto con la descripción de todos sus atributos y métodos (independientemente de su modificador de acceso).



Si abres el fichero `index.html` de la carpeta `build/docs/javadoc`, podrás ver el enlace al javadoc de la clase `Contact`. Al clicar en la clase, verás que tenemos toda la información pública de la misma, pero no obtenemos la información de los atributos (que son todos privados) ni de los métodos privados de dicha clase. Para obtener esta información en el javadoc, debemos añadir unas líneas en el fichero de configuración `build.gradle`:

```
javadoc {
    options.memberLevel = JavadocMemberLevel.PRIVATE
}
```

Después de añadir (y guardar) estas líneas al final del fichero, vuelve a ejecutar la tarea de Gradle `documentation→javadoc` para generar nuevamente la documentación, y verás que ahora sí tienes la información de los atributos y métodos privados.



**Requisito mínimo para evaluar este ejercicio:** Debe existir el directorio `docs/javadoc` con la documentación en formato web generada por la tarea `documentation→javadoc` de Gradle.

***(1 punto: 0.5 puntos documentación; 0.5 calidad de los comentarios Javadoc)***

## Ejercicio 4 (3 puntos)

Antes de empezar debes:

Leer el apartado 4.1 de la Guía de Java que habla sobre la clase `String` y las clases relacionadas `StringBuilder` y `StringBuffer`.

Copia en tu *workspace* el proyecto **PAC2Ex4** que te facilitamos con el enunciado y ábrelo con IntelliJ. A continuación debes codificar los `TODO` (del inglés, “*to do*”, en español, “*por hacer*”, o usando una traducción más libre: “*pendiente*”). Utilizando el potencial de las clases `String` y `StringBuilder` codifica los `TODO` del *package* `edu.uoc.pac2`.

En el *package* correspondiente, encontrarás la clase `TextAnalysis`, para la cual te proporcionamos el constructor por parámetros de la clase (no tenemos constructor por defecto). El resto de métodos los debes codificar siguiendo las indicaciones que te damos a continuación:

- `normalizeSentenceEndSymbols`: este método reemplaza del atributo `text` todos los signos de puntuación que hay listados en el atributo `SENTENCE_END_SYMBOLS` (representan los signos que determina el final de una oración) y los sustituye por el signo `"."` (punto). Es decir, si en `text` aparece el signo `"?"`, éste será sustituido por un `"."`. Este método devuelve el texto normalizado con todos los signos de puntuación incluidos en `SENTENCE_END_SYMBOLS` cambiados por `"."`. También actualiza el valor del atributo `text`.

**(0.25 puntos test proporcionados)**

- `getBagOfWords`: este método devuelve un *array* con todas las palabras que hay en el atributo `text` en el orden en que aparecen en `text`, pero en minúscula, sin signos de puntuación y sin espacios al inicio y/o al final de la misma. Si una palabra está repetida en el atributo `text`, ésta aparecerá repetida en el *array* que debe devolver este método.



**Pista:** Para eliminar los signos de puntuación te recomendamos usar una expresión regular que te permita detectar lo que es una letra del alfabeto.

**(0.75 puntos test proporcionados)**

- `getAverageSentenceLength()`: este método devuelve, dado el texto que almacena el atributo `text`, el número de palabras (incluyendo las repeticiones) medio por oración (i.e. texto acabado con un símbolo de `SENTENCE_END_SYMBOLS`), es decir: *(total de palabras en el atributo text) / (total de oraciones en el atributo text)*.

**(1 punto test proporcionados)**



**Pista:** Utiliza los métodos `normalizeSentenceEndSymbols` y `getBagOfWords`.

- `isIncluded`: este método determina si las letras del texto pasado como parámetros aparecen en el atributo `text`. Por ejemplo, si el valor de `text` es `"bxarlpccenp"` y el valor del parámetro es `"apple"`, entonces este método debe mirar si en `text` aparece el carácter `"a"`, `"p"` (dos veces), `"l"`, y `"e"`. Si es así, como es el caso, `isIncluded` devuelve `true`, en caso contrario, `false`.

**(1 punto test proporcionados)**



**Requisito mínimo para evaluar este ejercicio:** El programa debe pasar todos los test proporcionados para los métodos `normalizeSentenceEndSymbols`.



**Nota:** El estudiante puede recibir una penalización de hasta **1 punto** de la nota obtenida en este ejercicio en función de la calidad del código proporcionado.

## Formato y fecha de entrega

Tienes que entregar un fichero \*.zip, cuyo nombre tiene que seguir este patrón: loginUOC\_PEC2.zip. Por ejemplo: dgarciaso\_PEC2.zip. Este fichero comprimido tiene que incluir los siguientes elementos:

- El proyecto de IntelliJ PAC2Ex1 completado siguiendo las peticiones y especificaciones del Ejercicio 1.
- El proyecto de IntelliJ PAC2Ex2 completado siguiendo las peticiones y especificaciones del Ejercicio 2, así como la documentación en formato Javadoc cumpliendo los requisitos indicados en el Ejercicio 3.
- El proyecto de IntelliJ PAC2Ex4 completado siguiendo las peticiones y especificaciones del Ejercicio 4.

El último día para entregar esta PEC es el **7 de noviembre de 2022** antes de las 23:59. Cualquier PEC entregada más tarde será considerada como no presentada.