P0B: Herramientas: Maven+IntelliJ

IMPORTANTE. A TENER EN CUENTA DURANTE TODO EL CURSO

- Debes subir a GitHub las prácticas que realices. **SÓLO SE REVISARÁN** aquellas prácticas subidas a GitHub **antes** de iniciar la sesión de la siguiente práctica.
- **DURANTE** las clases de **prácticas:**
  - con **carácter general** se darán explicaciones sobre las soluciones de la práctica anterior,
  - con **carácter individual** se realizará el seguimiento del trabajo subido por el alumno (siempre y cuando se haya hecho dentro del plazo establecido) y
  - se resolverán las dudas que surjan.
- Cada alumno tiene asignado un profesor de prácticas, que realizará el seguimiento de vuestro trabajo. Las dudas sobre las prácticas debéis trasladarlas a vuestro tutor de prácticas.
- Tu trabajo de prácticas te permitirá comprender y asimilar los conceptos vistos en las clases de teoría. Es fundamental que trabajes bien las prácticas ya que vamos a evaluar no sólo tus conocimientos teóricos sino que sepas aplicarlos correctamente. Por lo tanto, el **resultado** de tu trabajo **PERSONAL** sobre las clases en aula y en laboratorio determinará si alcanzas o no las competencias teórico-prácticas planteadas a lo largo del cuatrimestre.

Una vez que tengamos la máquina virtual preparada, nuestro repositorio Git creado y clonado en la máquina virtual, y nuestra licencia Jetbrains activa, ya estamos en disposición de comenzar con nuestro trabajo práctico.

En esta primera sesión vamos a familiarizarnos en el uso de las herramientas de trabajo:

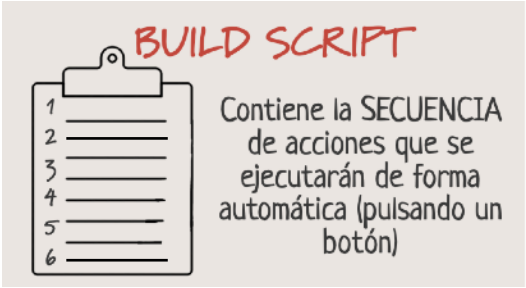
Maven

Maven es una **herramienta de construcción** de proyectos Java. Por definición, la construcción (*build)* de un proyecto es la secuencia de tareas que debemos realizar para, a partir del código fuente, poder usar (ejecutar) nuestra aplicación. Ejemplos de tareas que forman parte del proceso de construcción:

compilación, *linkado*, pruebas, empaquetado, despliegue.... Otros ejemplos de herramientas de construcción de proyectos son *Make* (para lenguaje C), *Ant* y *Graddle* (también para lenguaje Java).

Maven puede utilizarse tanto desde línea de comandos (comando mvn) como desde un IDE.

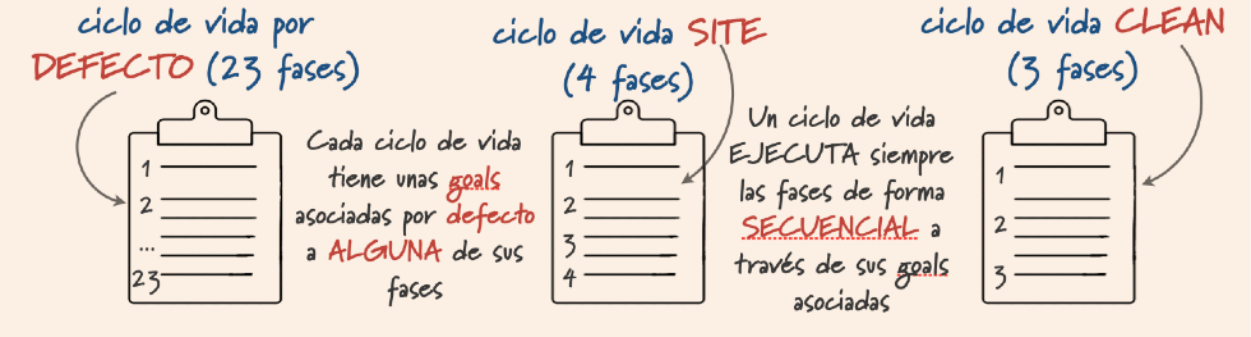
Cualquier herramienta de construcción de proyectos necesita conocer la secuencia de tareas que debe ejecutar para construir un proyecto. Dicha secuencia de tareas se denomina ***Build Script.***



Maven, a diferencia de Make o Ant (Graddle utiliza elementos de Ant y de Maven), permite definir el *build script* para un proyecto de forma declarativa, es decir, que no tenemos que indicar de forma explícita la "lista de tareas" a realizar, ni "invocar" de forma explícita la ejecución de dichas tareas.

La secuencia de tareas "programadas" en un *build script* en un momento determinado, define el **proceso de construcción** de nuestro proyecto (es posible que diferentes proyectos necesiten diferentes secuencias de tareas). Por lo tanto, ejecutar el *build script* es lo mismo que ejecutar el proceso de construcción de un proyecto.

Maven tiene predefinidas TRES secuencias de tareas (*build scripts*) para construir proyectos. Cada una de estas secuencias se denomina **ciclo de vida**.



El ciclo de vida más utilizado es el ciclo de vida por defecto (***default lifecycle***), y está formado por una lista de 23 tareas, denominadas **fases**. Ejemplos de fases son: *compile, test, package, deploy*,... Es importante que tengas claro que una fase es un concepto LÓGICO, no es un ejecutable, sino que podrá tener ASOCIADO algún ejecutable que realice una determinada acción.

FASES maven

(cada fase puede tener asociadas cero o más acciones ejecutables)

Una fase Maven identifica cuál debe ser la naturaleza de la acción o acciones que se ejecuten DURANTE la misma. Por ejemplo, el ciclo de vida por defecto contiene la fase "compile" y la fase "test": la primera la usaremos para asignar acciones ejecutables que lleven a cabo el proceso de compilación del código fuente del proyecto, mientras que la segunda está pensada para que se ejecuten las pruebas unitarias (lógicamente, la fase de compilación será anterior a la fase de pruebas).

GOALS y plugins

(una goal puede asociarse a una fase. Un plugin tiene 1 o varias goals)

Las acciones que se ejecutan en cada una de las fases se denominan **GOALS**. Por ejemplo la fase compile tiene asociada por defecto la ***goal*** (acción, tarea) denominada `compiler:compile`, que lleva a cabo la compilación de los fuentes del proyecto. Cualquier goal pertenece a un **PLUGIN**. Un plugin no es más que un conjunto de goals. Por ejemplo, el plugin `compiler` contiene las goals `compiler:compile` y `compiler:testCompile` (el nombre de la *goal* SIEMPRE va precedida del nombre del *plugin* separado por ":")

El proceso de construcción de maven puede generar un fichero empaquetado (jar, war, ear, ...), en el directorio *target*. Dependiendo del tipo de empaquetado, un ciclo de vida maven tiene asociadas POR DEFECTO ciertas GOALS.

Por ejemplo, cuando nuestro proyecto se empaqueta como un **.jar**, las GOALS asociadas al ciclo de vida por defecto son las siguientes:

Fase	plugin : goal	acciones realizadas por la goal
process-resources	`maven-resources-plugin: resources`	Copia *.* de /src/main/resources en target
compile	`maven-compiler-plugin: compile`	Compila *.java de /src/main/java
process-test-resources	`maven-resources-plugin: testResources`	Copia *.* de /src/test/resources en target
test-compile	`maven-compiler-plugin: testCompile`	Compila *.java de /src/test/java
test	`maven-surefire-plugin: test`	Ejecuta los tests unitarios
package	`maven-jar-plugin:jar`	Empaqueta *.class + recursos en un jar
install	`maven-install-plugin: install`	Copia el fichero jar en repositorio local
deploy	`maven-deploy-plugin: deploy`	Copia el fichero jar en repositorio remoto

Una **goal**, por tanto, no es más que un código ejecutable, implementado por algún desarrollador del plugin al que pertenece dicha goal.. Algunos desarrolladores "deciden" que una determinada goal estará asociada POR DEFECTO a una determinada fase de algún ciclo de vida Maven. Todas las goals son **CONFIGURABLES** (disponen de un conjunto de variables (propiedades) propias que tienen valores por defecto y que podemos cambiar). Por ejemplo, podemos cambiar la fase a la que se asociará dicha goal.

La forma de provocar la ejecución de una goal durante una fase consiste simplemente en añadir el plugin que la contiene en el fichero pom.mxl, en la sección <build> (y configurar sus propiedades, si es necesario). Si una goal no tiene asociada una fase por defecto, y no asociamos de forma explícita dicha goal a alguna fase, la goal **NO SE EJECUTARÁ** (aunque incluyamos su plugin en el fichero pom.xml).

Por ejemplo, cuando el empaquetado es ***jar***:

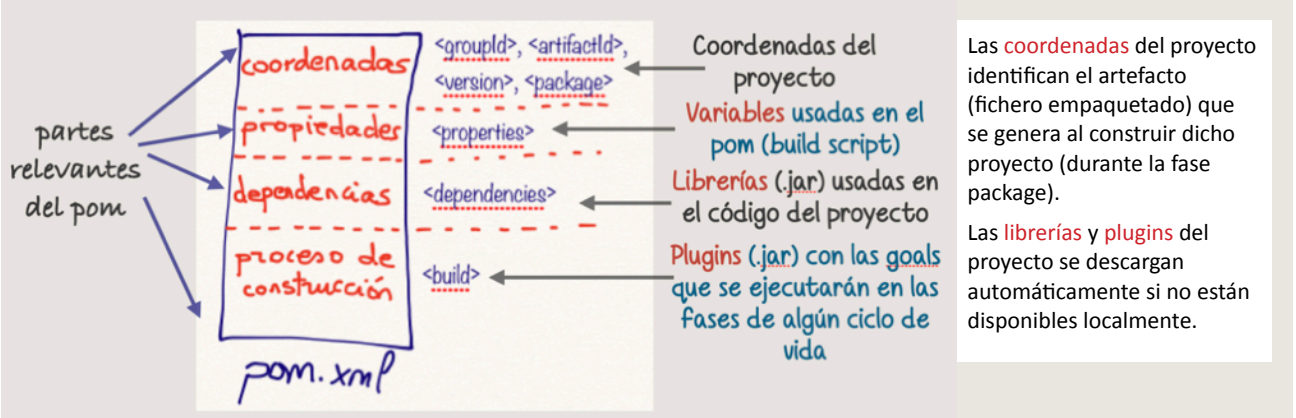
- La GOAL *compiler:**testCompile*** se ejecutará automáticamente durante la fase test-compile.
- La GOAL *compiler:**compile*** se ejecutará automáticamente durante la fase compile
- NO es necesario incluir el plugin ***compiler*** en el pom, a menos que queramos cambiar su configuración por defecto, o necesitemos una versión diferente del plugin incluido por defecto.

pom.xml

(configura el build script del proyecto.)

Cualquier proyecto Maven debe contener en su directorio raíz el fichero **pom.xml**. Dicho fichero nos permitirá configurar la secuencia de acciones a realizar (*build script)* para construir el proyecto, mediante la etiqueta <build>. También podremos indicar qué librerías (ficheros .jar) son necesarias para compilar/ejecutar/probar... nuestro proyecto (etiqueta <dependencies>).

Es importante que sepamos identificar al menos 4 "secciones" en el fichero pom.xml. Cada una de ellas se caracteriza por usar determinadas etiquetas:



Las coordenadas del proyecto identifican el artefacto (fichero empaquetado) que se genera al construir dicho proyecto (durante la fase package).

Las librerías y plugins del proyecto se descargan automáticamente si no están disponibles localmente.

artefactos Maven

(son ficheros que se identifican por sus coordenadas))

Durante el proceso de construcción, Maven usa, y también puede generar, ficheros empaquetados que se identifican mediante sus coordenadas, separadas por ":". Dichos ficheros se denominan artefactos Maven. Para identificar un artefacto Maven se requieren cuatro coordenadas, de forma que cualquier artefacto Maven se especifica de forma única (no hay dos artefactos con las mismas coordenadas).

Las coordenadas que identifican de forma única a un artefacto Maven son, como mínimo: **`groupId:artifactId:version`**

- **`groupId`** es el identificador de grupo. Se utiliza normalmente para identificar la organización o empresa desarrolladora y puede utilizar notación de puntos. Por ejemplo: *org.ppss*
- **`artifactId`** es el identificador del artefacto (nombre del archivo), normalmente es el mismo que el nombre del proyecto. Por ejemplo: *practica1*
- **`version`** es la versión del artefacto. Indica la versión actual del fichero correspondiente. Por ejemplo: *1.0-SNAPSHOT*.

Opcionalmente, se puede usar una cuarta coordenada ***package.***

- **`package`** es la extensión del fichero. Indica el tipo de empaquetado. Esta coordenada es OPCIONAL. Si se omite, se asume que el empaquetado es *jar*.

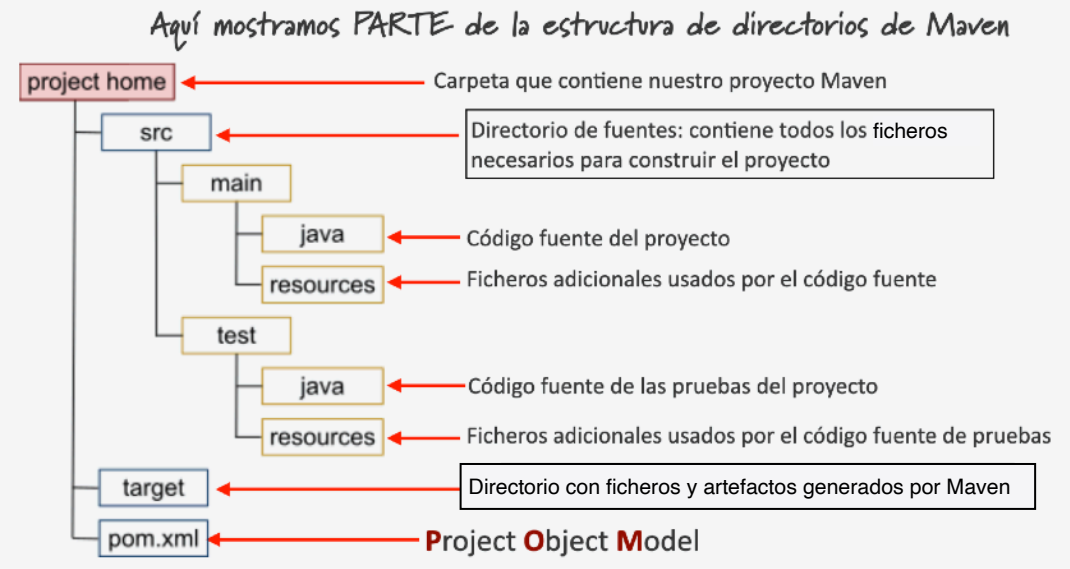
Los artefactos usados por Maven se almacenan en un repositorio local Maven, situado en *$HOME/.m2/repository.* Las coordenadas se usan para identificar exactamente la ruta del fichero (artefacto maven) en el repositorio maven (que puede ser local o remoto). Por ejemplo:

- `org.ppss:practica1:1.0-SNAPSHOT` representa al fichero $HOME/.m2/repository/org/ppss/practica1/1.0-SNAPSHOT/practica1-1.0-SNAPSHOT.jar
- `org.ppss:practica1:2.0-SNAPSHOT` representa al fichero $HOME/.m2/repository/org/ppss/practica1/2.0-SNAPSHOT/practica1-2.0-SNAPSHOT.jar
- `org.ppss:proyecto3:1.0-SNAPSHOT:war` representa al fichero $HOME/.m2/repository/org/ppss/proyecto3/1.0-SNAPSHOT/proyecto3-1.0-SNAPSHOT.war

estructura de directorios Maven

(la misma en TODOS los proyectos Maven)

TODOS los proyectos Maven usan la MISMA estructura de directorios. Así, por ejemplo, el código fuente del proyecto estará en el directorio **src/main/java**, y el código que implementa las pruebas del proyecto siempre lo encontraremos en el directorio **src/test/java**. Los ficheros y/o artefactos generados durante la construcción, por ejemplo los ficheros .class, siempre estarán en el directorio **target** (o alguno de sus subdirectorios). El directorio target se genera automáticamente en cada construcción del proyecto, por eso no necesitamos "guardarlo" en GitHub.



Por otro lado, cualquier LIBRERÍA EXTERNA (ficheros .jar) usada en nuestro proyecto, debe incluirse en el fichero pom.xml (en la sección ***<dependencies>***).Maven se encarga de descargar dicha librería si es necesario. Es más, si utilizamos una librería, que a su vez depende de otra, Maven automáticamente se encarga de descargarse también esta última, y así sucesivamente. Esto hace que nuestros proyectos "pesen" poco, ya que no será necesario incluir ni el directorio target, ni ninguna librería y/o plugin utilizados por el proyecto. Éstos se descargarán de forma automática, si es necesario, cada vez que construyamos el proyecto. Por tanto, si queremos "llevarnos" nuestro proyecto a otra máquina, únicamente necesitamos el fichero pom.xml, y el directorio src del proyecto.

repositorios locales y remotos Maven

(almacenan artefactos Maven)

Todos los ficheros y artefactos generados y/o utilizados por Maven se almacenan en repositorios. Maven mantiene una serie de repositorios remotos, que alojan los plugins y librerías que podemos utilizar. Cuando ejecutamos Maven por primera vez en nuestra máquina, se crea el directorio **.m2** (en nuestro $HOME), que será nuestro repositorio local. Cuando iniciamos un proceso de construcción Maven, primero se consulta en nuestro repositorio local, para ver si contiene todos los artefactos necesarios para realizar la construcción. Si falta algún artefacto en nuestro repositorio local, Maven automáticamente lo descargará de algún repositorio remoto. Si borramos el directorio .m2, éste se volverá a crear.

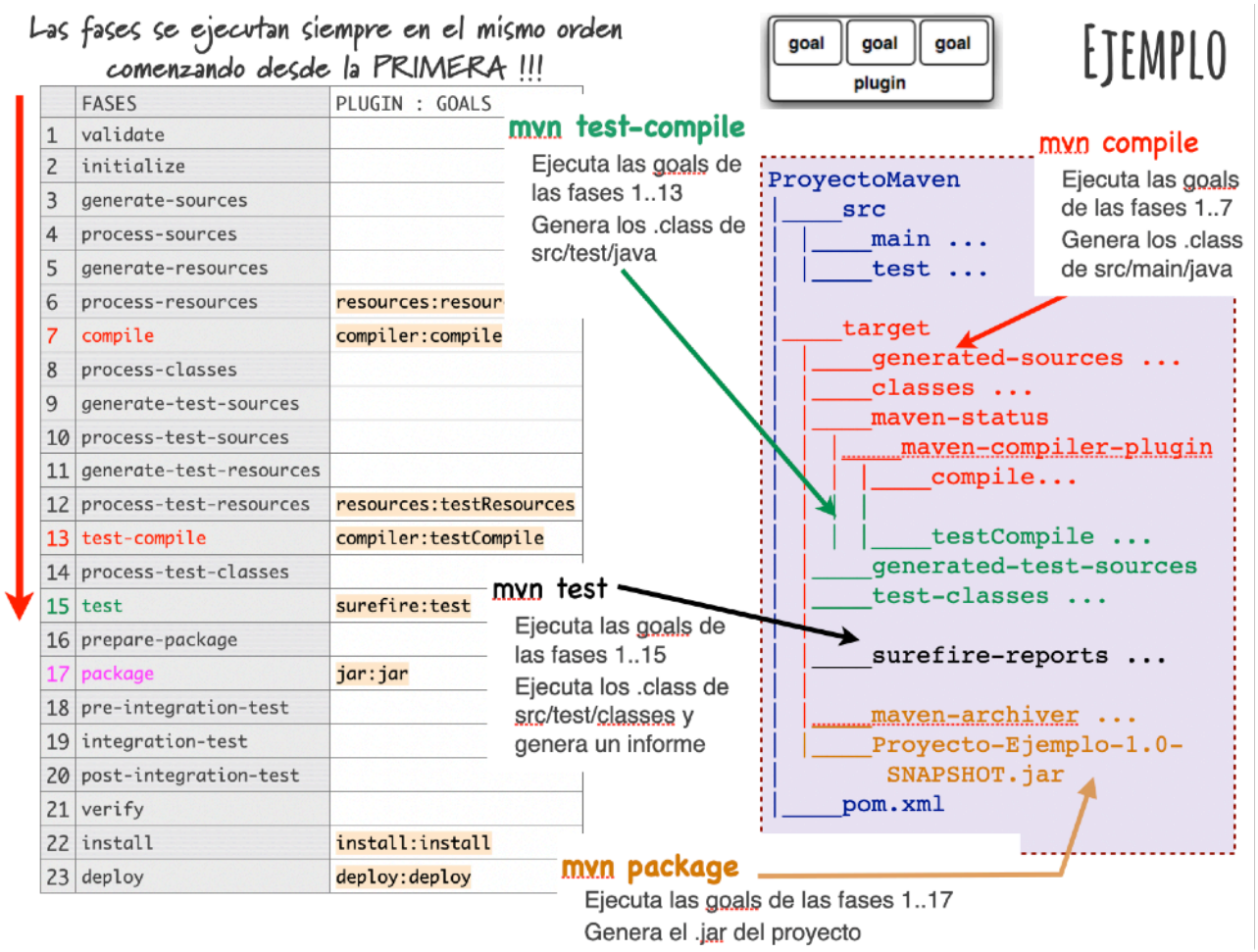
Ejecución de Maven

(mvn fase/goal)

Para iniciar el proceso de construcción de Maven, usamos el comando **mvn** seguido de la **fase** (o fases) que queramos realizar, o bien indicando la **goal**, o goals que queremos ejecutar de forma explícita (separadas por espacios). Las goals se ejecutarán una por una en el mismo orden que hemos indicado. Por ejemplo, si tecleamos: mvn fase1 fase2 plugin1:goal3 plugin2:goal4, será equivalente a ejecutar: mvn fase1, mvn fase2, mvn plugin1:goal3, y mvn plugin2:goal4, en este orden.

El comando ***mvn <faseX>*** ejecuta todas las goals asociadas a todas y cada una de las fases, siguiendo exactamente el orden de las mismas en el ciclo de vida correspondiente, desde la primera, hasta la fase que hemos indicado (<faseX>).

El comando ***mvn plugin:goal*** ejecuta únicamente la goal que hemos especificado



	FASES	PLUGIN : GOALS
1	validate	
2	initialize	
3	generate-sources	
4	process-sources	
5	generate-resources	
6	process-resources	resources:resour
7	compile	compiler:compile
8	process-classes	
9	generate-test-sources	
10	process-test-sources	
11	generate-test-resources	
12	process-test-resources	resources:testResources
13	test-compile	compiler:testCompile
14	process-test-classes	
15	test	surefire:test
16	prepare-package	
17	package	jar:jar
18	pre-integration-test	
19	integration-test	
20	post-integration-test	
21	verify	
22	install	install:install
23	deploy	deploy:deploy

IntelliJ IDEA Ultimate 

IntelliJ es un IDE muy utilizado para trabajar con diferentes tipos de aplicaciones, entre ellas aplicaciones java y Maven. Nosotros trabajaremos SIEMPRE con proyectos Maven.

En esta primera práctica empezaremos a familiarizarnos con el uso de esta herramienta. Veamos primero algunos conceptos importantes:

- **Project**. Todo lo que hacemos con IntelliJ IDEA se realiza en el contexto de un **Proyecto**. Los proyectos no contienen en sí mismos elementos tales como código fuente, *scripts* de compilación o documentación. Son el nivel más alto de organización en el IDE, y contienen la definición de determinadas propiedades. Para los que estéis familiarizados con Eclipse, un proyecto sería similar a un *workspace* de Eclipse. La configuración de los datos contenidos en un proyecto se puede almacenar en un directorio denominado **.idea**, y es creado y mantenido automáticamente por IntelliJ.
- **Module**. Un Módulo es una unidad funcional que podemos compilar, probar y depurar de forma independiente. Los módulos contienen, por lo tanto, ficheros que contienen el código fuente, scripts de compilación, tests, descriptores de despliegue, y documentación. Sin embargo un módulo no puede existir fuera del contexto de un proyecto. La información de configuración de un módulo se almacena en un fichero denominado **.iml**. Por defecto, este fichero se crea automáticamente en la raíz del directorio que contiene dicho módulo. Un proyecto IntelliJ puede contener uno o varios módulos. Para los que estéis familiarizados con Eclipse, un módulo sería similar a un *proyecto* de Eclipse
- **Facet**. Las **Facetas** representan varios *frameworks,* tecnologías y lenguajes utilizados en un módulo. El uso de facetas permite descargar y configurar los componentes necesarios de los diferentes frameworks. Un módulo puede tener asociadas varias facetas. Algunos ejemplos de facetas son: Android, AspectJ, EJB, JPA, Hibernate, Spring, Struts, Web, Web Services,...
- **Run/Debug Configuration**. Podemos configurar la ejecución de determinadas acciones (como por ejemplo arrancar/parar un servidor de aplicaciones, lanzar un *script* de compilación, ...), de forma que quede guardada con un determinado nombre y la podamos lanzar a voluntad, simplemente con un *click* de ratón. IntelliJ tiene varias configuraciones predefinidas, y podemos crear nuevas configuraciones a partir de éstas. Para los que estéis familiarizados con Eclipse, una configuración de ejecución en IntelliJ sería similar al mismo concepto en Eclipse.

➪ Creación de un proyecto IntelliJ a partir de un proyecto Maven existente 

Indicamos dos formas, que usaremos en los ejercicios de esta sesión.

Una posible forma de hacerlo es a partir del **fichero pom.xml** presente en cualquier proyecto Maven. Para ello simplemente:

- Desde el menú principal elegimos File→Open (u opción Open cuando abrimos IntelliJ).
- En el cuadro de diálogo seleccionamos el fichero pom.xml, y pulsamos OK. Nos preguntará si queremos abrir como fichero o como proyecto. Elegiremos como proyecto. Se nos preguntará si confiamos en dicho proyecto, y le diremos que sí.

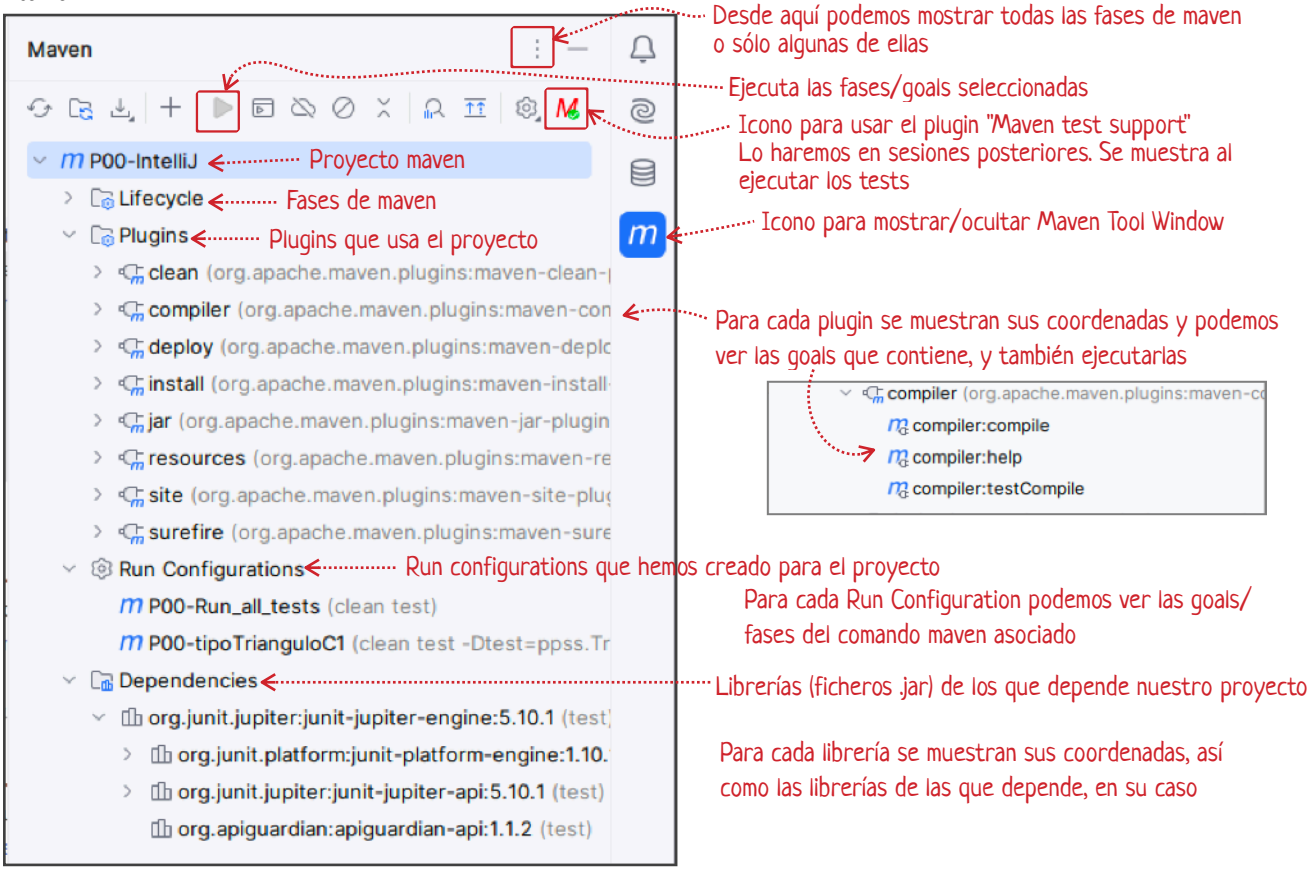
De forma alternativa, también podremos usar la opción File→Open y seleccionar la **carpeta** que contiene el fichero pom.xml. Nos preguntará si confiamos en dicho proyecto, y le diremos que sí.

Al abrir el proyecto, veremos un icono en el extremo izquierdo la barra superior con 4 líneas horizontales. Lo seleccionaremos cada vez que necesitemos usar alguna de las opciones de la barra de herramientas de IntelliJ.

En la máquina virtual usamos openjdk 17. Siempre que abramos un proyecto, nos aseguraremos de que lo tiene asociado. Para ello, desde la barra de herarmientas de IntelliJ seleccionamos ***File->Project Structure → Project Settings → Project → SDK***, y seleccionamos: "**17 version 17.0.9**". En el cuadro de texto ***Languaje level***, seleccionaremos "*17-Sealed types, always-strict floating-point semantics*". Para guardar los cambios pulsamos sobre el botón "***Apply***".

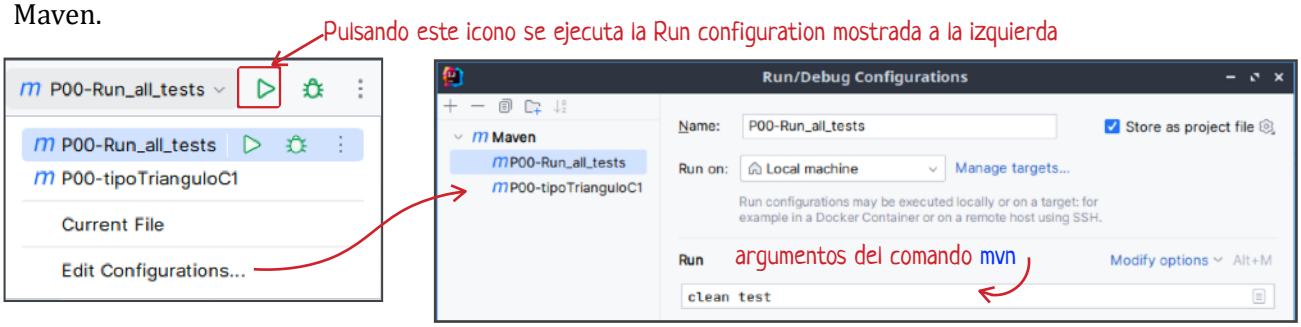
IntelliJ IDEA Maven Tool Window

IntelliJ permite mostrar diferentes ***"Tool Windows"*** en las que se visualizan diferentes perspectivas (vistas) del proyecto. Una de estas "vistas" es la ventana de Maven (***"Maven tool window"***), que usaremos cuando trabajemos con un proyecto Maven. A continuación mostramos el aspecto de dicha ventana.



IntelliJ IDEA Run Configurations

Usaremos las Run Configurations para realizar diferentes construcciones del proyecto, utilizando comandos maven. IntelliJ proporciona diferentes plantillas. Nosotros trabajaremos con las plantillas Maven.



Cuando creemos una run configuration siempre marcaremos la opción ***Store as project file***, y la guardaremos en la carpeta con el nombre ***intellij-configurations***, al mismo nivel que el pom. No se trata de un directorio maven, sino de una carpeta que hemos añadido para no "perder" las *run configurations* al subir el trabajo de prácticas a GitHub. Por defecto, las *run configurations* se guardan en la carpeta ***.ide***a, la cual es ignorada por git.

Para **crear** una run configuration usaremos el icono "**+**" de la ventana emergente que aparece cuando seleccionamos "***Edit Configurations***"

Ejercicios

El directorio **Plantillas-P0B** contiene un proyecto maven (directorio **P00-IntelliJ**) que usaremos para realizar los ejercicios.

Para poder hacer los ejercicios necesitarás:

Git/ GitHub

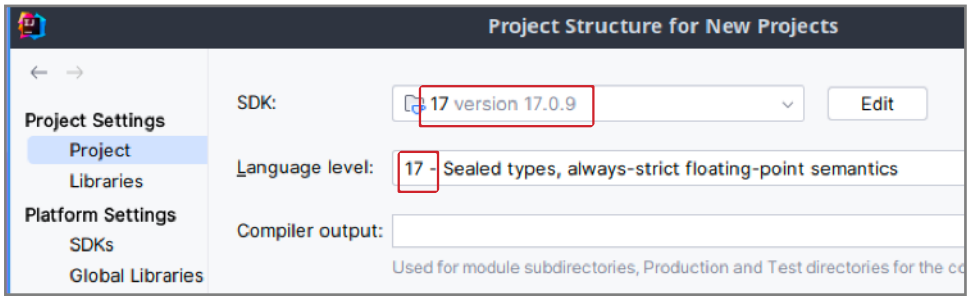
- **Situarte en tu directorio de trabajo** (directorio que contiene la carpeta oculta .git). Si ha creado tu repositorio Git y lo has clonado en tu máquina, tu directorio de trabajo, en el que debes hacer los ejercicios, será: **$HOME/practicas/ppss-2024-Gx-apellido1-apellido2**.
  IMPORTANTE!!! RECUERDA que a partir de ahora, TODO lo que hagas en prácticas estará en algún subdirectorio de tu directorio de trabajo.
- **COPIAR** el directorio **P00-IntelliJ** en tu directorio de trabajo y sitúate en él. A partir de aquí, se pide:

Ejercicio 1: proyecto Maven P00-IntelliJ

IntelliJ

Abre el proyecto Maven P00-IntelliJ a partir del directorio que contiene el pom.xml (opción Open). Asegúrate de que la ventana ***Maven Tools*** está visible.

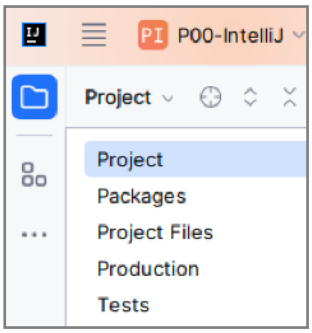
Recuerda que cada proyecto maven tiene que tener asociado jdk 17. En lugar de indicarlo cada vez, podemos usar la opción: ***File→New Projects Setup→ Structure → Project Settings →Project.*** y seleccionar SDK 17 e indicar que el nivel del lenguaje también será el 17, tal y como mostramos en la siguiente imagen.



Se pide:

Maven *Directory Layout*

A) Observa la **estructura de directorios del proyecto**. La información del proyecto puede mostrarse desde diferentes "perspectivas". Si quieres ver exactamente las carpetas físicas del disco duro debes mostrar la vista "***Project Files***", en lugar de "***Project***", que es la que tendrás por defecto (tal y como se muestra en la imagen de la derecha). Puedes anotar dicha estructura en un fichero .txt para facilitar tu proceso de estudio.



La estructura de directorios es la misma en CUALQUIER proyecto Maven. Es importante conocerla, ya que el proceso de construcción de Maven asume que determinados ficheros están situados en determinados directorios. Por ejemplo, si el código fuente de las pruebas lo implementásemos en el directorio /src/main/java, no se ejecutarían dichos tests aunque lanzásemos la fase "test" de Maven

Maven *pom.xml*

B) Muestra en el editor la configuración de nuestro proceso de construcción (**fichero pom.xml**). Verás que el fichero xml contiene información sobre: las coordenadas, propiedades, dependencias y sobre la construcción del proyecto (etiqueta <build>)

En el pom se definen las coordenadas de nuestro proyecto, y además hace referencia a tres artefactos: uno en la sección de dependencias y dos en la sección <build>, cada uno de los cuales tiene sus propias coordenadas. Deberías saber identificarlas. Recuerda que los artefactos maven se identifica por sus coordenadas. Nuestro proyecto maven se empaquetará de alguna forma y se

generará el correspondiente artefacto. Deberías saber qué tipo de empaquetado tiene nuestro proyecto maven viendo sus coordenadas, así como el nombre y la ruta del artefacto generado.

Fíjate también que los artefactos usados en nuestro pom, se usan en secciones (etiquetas) diferentes, y que cada sección tiene un propósito diferente.

La etiqueta ***<properties>*** se utiliza para definir y/o asignar/modificar valores a determinadas "variables" usadas en nuestro pom.xml. Podemos usar propiedades ya predefinidas (por ejemplo la propiedad "*project.build.sourceEncoding*"), o podemos definir cualquier propiedad que nos interese. A partir de Maven 3 es OBLIGATORIO especificar en el pom.xml un valor para la propiedad "`project.build.sourceEncoding`", por lo que esta línea aparecerá en todos los ficheros pom.xml de nuestros proyectos.

Observa que hemos especificado el valor "test" para la etiqueta ***<scope>*** en uno de los artefactos. Dicho valor indica que el artefacto en cuestión se necesita durante la compilación de los tests. Si esta etiqueta se omite, su valor por defecto es "compile" y significa que el artefacto es necesario para compilar los fuentes del proyecto.

El pom.xml de nuestra construcción incluye dos plugins, los cuales ya están incluidos por defecto, puesto que el empaquetado del proyecto es jar. Es decir, que si no los añadimos al pom, maven los tendrá en cuenta igualmente, lo que ocurre es que las versiones por defecto de los plugins que usa maven son anteriores a las que nosotros queremos usar. Se trata de los plugins ***maven-surefire-plugin***. y ***maven-compiler-plugin***. Para ver qué versiones se incluyen por defecto debes comentar los plugins en el pom, pulsar sobre el primer icono a la izquierda de la ventana Maven Projects("Reload All Maven Projects"), y consultar las versiones de los plugins desde dicha ventana. Recuerda que un comentario xml empieza con `<!--` y termina con `-->`.

Averigua qué versión se incluye por defecto de ambos plugins.

C) Con respecto al **código fuente**, la **clase Triángulo** contiene la implementación del método "*tipo_triangulo()*" cuya especificación asociada es la siguiente: Dados tres enteros como entrada, que representan las longitudes de los tres lados de un triángulo, el método devuelve como resultado una cadena de caracteres indicando el tipo de triángulo ("Equilatero", "Isosceles", o "Escaleno"), o la cadena "No es un triángulo"·si los tres lados proporcionados no forman un triángulo. Para que los tres lados proporcionados como entrada puedan formar un triángulo tiene que cumplirse la condición de que la suma de dos de sus lados tiene que ser siempre mayor que la del tercero. Los datos pasados por parámetro deben ser valores comprendidos entre 1 y 200, Si alguno de los tres lados: a, b, ó c excede dicho rango, entonces el método devolverá el mensaje "Valor x fuera del rango permitido", siendo x el carácter a, b, ó c, en función de que sea el primer, segundo, o tercer valor de entrada el que incumpla la condición, y con independencia de que los tres lados formen o no un triángulo.

Maven Source Code

Este es uno de los ejemplos más utilizados en la literatura sobre pruebas, quizá porque contiene una lógica clara, pero a la vez compleja. Fue utilizado por primera vez por Gruenberger en 1973, aunque en una versión algo más simple.

Fíjate que esta especificación nos proporciona el conjunto S que hemos visto en la sesión de teoría.

D) La clase **TrianguloTest** contiene la implementación de cuatro ***casos de prueba*** (los cuatro métodos anotados con @Test) asociados a la especificación del apartado anterior. Después de estudiar la teoría deberías ser capaz de identificar dichos casos de prueba, y crear la correspondientes tabla. Puedes identificar cada caso de prueba como C1, C2, C3, y C4, y deberías tener claro cuántas columnas necesitas en la tabla y lo que significa cada una de ellas.

Maven Test Code

Identificador del Caso de prueba	Dato de entrada 1	...	Dato de entrada n	Resultado esperado

Cada uno de los tests tiene un nombre que indica cuál debería ser el resultado cuando se dan ciertas condiciones sobre las entradas, siguiendo el formato:

`<id>_<nombre_método>_should_<resultado_esperado>_when_<condiciones_sobre_las_entradas>`

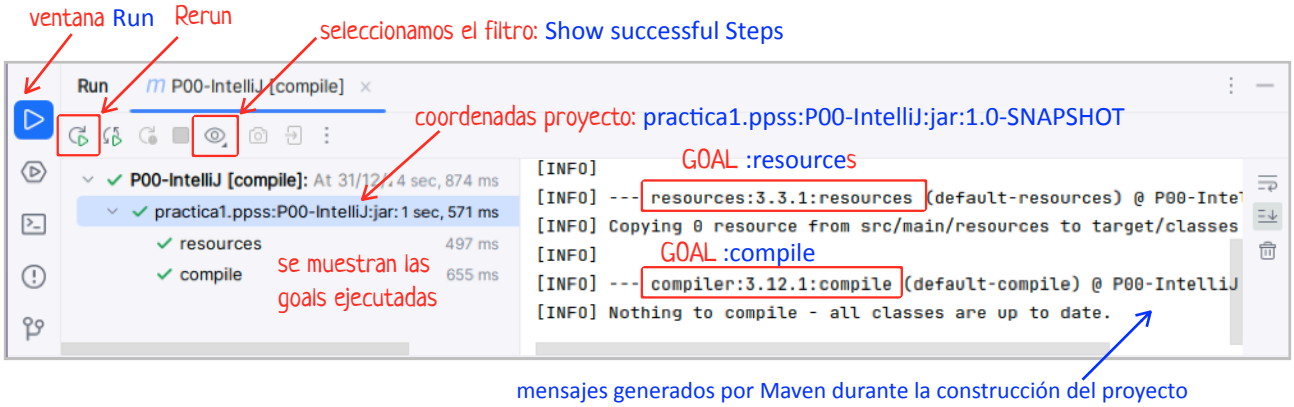
Observa la implementación de cada test y verás que todos ellos siguen la misma lógica de programa. Mira los comentarios. Este algoritmo sigue un patrón conocido como **AAA**: ***A**rrange-**A**ct-**A**ssert*, y debes usarlo SIEMPRE!!!

Ejercicio 2: construcción del proyecto (fases compile, test, clean)

Maven Build (compile)

Vamos a "construir" el programa. En este caso sólo vamos a compilarlo. Para ello haremos doble click sobre la **fase *compile*** desde la ventana "Maven Projects". Esta acción en el IDE es equivalente a ejecutar desde línea de comando la orden: ***mvn compile*** (puedes comprobarlo ejecutando dicho comando desde un terminal).

Al ejecutar la fase *compile*, el resultado se muestra automáticamente en una ventana en la parte inferior del IDE (ventana ***Run***) en donde verás los mensajes que genera Maven durante el proceso de construcción. Tal y como mostramos en la siguiente imagen:



Familiarízate con la ventana **Run**. Si seleccionamos alguno de los "pasos" (goals ejecutadas), en el panel de la derecha sólo se mostrarán los logs correspondientes a dicha goal.

Desde aquí podemos volver a repetir el proceso de construcción pulsando sobre el triángulo verde.

Familiarízate con esta ventana y la información que muestra. Prueba a marcar/desmarcar las opciones desde el icono con forma de "ojo", y a repetir el proceso de construcción usando el icono "*Rerun*".

Después de ejecutar la fase ***compile:***

Maven Build (clean)

A) El proceso de construcción habrá generado un directorio nuevo en nuestro proyecto maven: el directorio ***target***. Fíjate en la nueva estructura de directorios creada, y qué ficheros contienen. Esta nueva estructura, también es común para CUALQUIER proyecto maven, por lo que deberás conocerla. Ahora vamos a ejecutar la **fase *clean*.** Observa lo que ocurre y fíjate en las goals que se ejecutan. Ahora vuelve a compilar el proyecto. De nuevo verás las goals ejecutadas en la ventana inferior. Verás que NO se han ejecutado los tests

B) Para **ejecutar los tests** vamos a hacer doble click sobre la **fase *test***. Ahora desde la ventana **Run** selecciona el elemento ***P00-IntelliJ*** para poder ver en el panel de la derecha todos los **logs de maven**, que incluirán el informe de las pruebas realizadas con JUnit. Concretamente:

- "**Tests run**" indica el número total de tests ejecutados.
- "**Failures**" indica el número de tests cuyo resultado esperado NO coincide con el real.
- "**Error**" (hablaremos de él en sesiones posteriores).

Verás que uno de los tests falla, es decir representa un fallo de ejecución (*failure*). Esto significa, como ya hemos indicado, que el valor del resultado esperado y el real NO coinciden (en pantalla se muestra la razón de esta discrepancia). Observa también algo muy importante, el resultado de la construcción es: **BUILD FAILURE**, es decir, el proceso de construcción (*mvn test*) no se ha completado con éxito puesto que se han detectado problemas en la ejecución de alguna de las goals (concretamente durante la ejecución de la goal *surefire:test*).

Cuando ejecutamos la fase "**test**" de maven se ejecutan **TODOS** los tests del proyecto (es decir, si tuviésemos otra clase con métodos anotados con @Test, también se ejecutarían).

Hemos proporcionado a modo de ejemplo una *Run configuration* para uno de los tests de dicha clase (***P00-tipoTrianguloC1***). Ejecútala para comprobar su funcionamiento. Crea una nueva *Run configuration* para que ejecute todos los tests del proyecto, con el nombre ***P00-Run_all_tests***

debugging

C) Para poder concluir nuestro proceso de construcción con éxito: **BUILD SUCCESS**, necesitamos eliminar el problema/s que provoca el fallo de ejecución. Tendremos que averiguar por qué el

informe del resultado de la ejecución del caso de prueba correspondiente es un fallo. Hemos cometido un error en la implementación del método que estamos probando, que hace que el resultado esperado (resultado que debería dar si estuviese bien implementado) no es el real. Identifica la causa y modifica el código para que el resultado real sea el correcto (recuerda que este proceso se llama DEPURACIÓN, o *debugging*). A continuación vuelve a ejecutar la fase test. Repite el proceso hasta que consigamos un informe de pruebas sin "*failures*" ni "*errors*", y el proceso de construcción termine con: BUILD SUCCESS.

Test Case Design

D) Observa qué tienen en común el test C1 y un posible test adicional C5 con datos de entrada: a=7,b=7,c=7, y piensa en la conveniencia o no de incluir C5 al conjunto de tests. De la misma forma razona si son necesarios los tests C2 y C3. Añade dos posibles casos de prueba adicionales que "aporten valor" al conjunto de casos de prueba (no sean innecesarios).

Es importante tener claro que la **secuencia de pasos** para realizar las pruebas es ésta:

- primero tenemos que obtener (diseñar) los casos de prueba que usaremos para comprobar que el programa funciona correctamente (apartado A). Para ello tenemos que "elegir" ***datos de entrada*** concretos, y asociar un ***resultado esperado***, de acuerdo con el comportamiento correspondiente de la especificación.
- A continuación tenemos que implementar los tests, y
- finalmente ejecutarlos para obtener el informe de pruebas.

El **informe de pruebas** obtenido (proporcionado por JUnit), nos indica si hemos detectado defectos en nuestro programa o no. Es muy importante que te quede claro que un informe JUnit sin errores no significa que el programa esté libre de ellos. Es decir, nuestras pruebas sólo pueden demostrar la presencia de errores (no la ausencia de los mismos).

Ejercicio 3: construcción del proyecto (fases *package*, *install*)

Hasta ahora hemos lanzado la ejecución de las fases del ciclo de vida de maven clean, compile y test" Vamos a ejercitar otras dos fases importantes: la fase package, y la fase install.

Maven Build (package)

A) Ejecuta la **fase *package*** (desde la ventana maven) y observa los cambios en la estructura del proyecto. Fíjate qué acciones (goals) se han ejecutado para construir el proyecto y qué artefactos/ficheros nuevos se han generado. Es importante que tengas claro qué artefactos/ficheros se generan en cada fase. Modifica uno de los tests, de forma que dé un resultado fallido, ejecuta la fase clean, y a continuación la fase package de nuevo, y observa lo que ocurre. De igual forma, ahora, en lugar de introducir un error en los tests, edita el fichero Triangulo.java, quita un punto y coma para provocar un error de compilación y vuelve a ejecutar las fases clean y package. Tienes que tener claro el resultado obtenido y por qué se obtiene dicho resultado.

Maven Build (install)

B) Vuelve a reparar todos los errores introducidos y ejecuta la **fase *install***. En este caso el resultado es menos "obvio" ya que en esta fase se "instala" (copia) el artefacto generado en la fase anterior (fichero con extensión .jar) en el repositorio local. El repositorio local de maven se encuentra en $HOME/.m2/repository. En el repositorio local se almacenan todos los artefactos que maven ha utilizado para construir el proyecto. Deberías saber la ruta exacta de cada artefacto a partir de sus coordenadas. Si borras el directorio .m2 no importa, maven lo volverá a crear automáticamente durante la próxima ejecución del comando mvn. La primera vez que ejecutemos maven, si el repositorio local no existe, entonces se crea, y maven se descarga todos aquellos ficheros que necesita de sus repositorios remotos. Esto significa que, a medida que vayas ejecutando maven y necesitando los artefactos (ficheros jar, war, ear, pom,...) para construir el proyecto, tu repositorio local irá creciendo de tamaño. Si maven encuentra en el repositorio local el artefacto correspondiente, no será necesario proceder a su descarga, por lo que se reducirá el tiempo de construcción del proyecto.

Guardamos nuestro trabajo en GitHub

Recuerda que debes guardar TODO tu trabajo de prácticas en GitHub. Deberías "subir" a GitHub los ejercicios según los vas realizando (no esperes a tenerlos todos, podrías perder tu trabajo si tienes algún problema con la máquina virtual).

Git/ GitHub

Para ello simplemente debes usar los comandos git desde un terminal y **desde tu directorio de trabajo** (ppss-2024-Gx-apellido1-apellido2):

- git add .
- git commit -m"Ejercicio P00-X terminado"
- git push

No esperes a tener todos los ejercicios hechos para subir tu trabajo a GitHub. Puedes hacer tantos commits como quieras y/o necesites.

Resumen

¿Qué conceptos y cuestiones me deben quedar CLAROS después de hacer la práctica?

GIT

- Herramienta de gestión de versiones. Nos permite trabajar con un repositorio local que podemos sincronizar con un repositorio remoto (usaremos GitHub para guardar el repositorio remoto).

MAVEN

- Herramienta automática de construcción de proyectos java.. El build script se especifica de forma declarativa en el fichero pom.xml, en el que encontramos varias partes bien diferenciadas: coordenadas, propiedades, dependencias y plugins. (Hay más secciones, pero de momento sólo veremos estas cuatro)
- Un artefacto maven es un fichero usado y/o generado por el proceso de construcción de maven y que se identifica mediante sus coordenadas.. Dichas coordenadas nos permiten conocer en qué ruta de nuestro repositorio local se almacenará dicho fichero, así como el nombre exacto del mismo.
- Los proyectos maven requieren una estructura de directorios fija. El código de los tests está físicamente separado del código fuente de la aplicación.
- Maven tiene predefinidos tres ciclos de vida (o build scripts) para construir un proyecto. Cada ciclo de vida está formado por una secuencia ordenada de fases, cada fase puede tener asociadas unas goals. Una goal siempre pertenece a un plugin. El resultado del proceso de construcción maven puede ser Build failure o Build success.
- El comando mvn permite ejecutar tanto una fase (de alguno de los ciclos de vida) como una goal. En el primer caso se ejecutan todas las goals asociadas a todas las fases del ciclo de vida, desde la primera, hasta la fase especificada. En el segundo caso únicamente ejecutaremos la goal indicada..
- Construiremos el proyecto a través de IntelliJ. Los comandos maven a ejecutar los guardaremos en ficheros xml llamados Run Configurations. Dichos ficheros los crearemos en la carpeta intellij-configurations (NO es una carpeta de maven, y el nombre de dicha carpeta es arbitrario)
- También podemos ejecutar cualquiera de las fases de los ciclos de vida de maven directamente desde la ventana Maven Tools. Igualmente podemos ver/ejecutar cualquiera de las goals de los plugins que usa nuestro pom. Para cada plugin, se muestran sus coordenadas..

TESTS

- Un test automatiza la ejecución de un caso de prueba, el cual representa un comportamiento del elemento a probar (cada comportamiento está formado por datos concretos de entrada + resultado concreto esperado)
- Una vez definido (diseñado) el caso de prueba para un test, éste puede implementarse como un método java anotado con @Test (anotación JUnit). Los tests se compilan y se ejecutan en diferentes momentos durante el proceso de construcción de un proyecto.
- El algoritmo de un test es siempre el mismo. Y sigue un patrón conocido como AAA (Arrange-Act-Assert). El test comprueba (verifica) si el comportamiento especificado en S coincide con el comportamiento implementado en P, siendo S el conjunto de todos los comportamientos especificados y P el conjunto de todos los comportamientos implementados)
- Dependiendo de cómo hayamos diseñado (definido) los casos de prueba, detectaremos más o menos errores (discrepancias entre el resultado esperado y el resultado real).
- Es imposible detectar todos los posibles errores (ya que necesitaríamos un número de casos de prueba impracticable), por lo tanto, nuestras pruebas sólo pueden demostrar la presencia de defectos en el código, pero nunca pueden demostrar la ausencia de ellos. Por lo tanto, como testers, buscaremos detectar el máximo número de errores posibles (efectividad) con el menor número posible de casos de prueba (eficiencia)
- Para detectar la presencia de un defecto en el código, vamos a EJECUTAR dicho código. A esta forma de proceder para detectar defectos se la conoce como pruebas DINÁMICAS, y es la aproximación que vamos a seguir durante todo el curso.