P1- Entorno de pruebas

Máquina virtual de la asignatura

Para las prácticas en el laboratorio utilizaremos una Máquina Virtual en la que tendréis instalado el software que vamos a necesitar.

Para trabajar con la máquina virtual fuera del laboratorio necesitaréis tener instalado previamente VirtualBox (incluyendo VirtualBox Extension Pack). La máquina virtual (vdi original) la tenéis disponible en los laboratorios en: /usr/local/VDI-PPSS/ppss.vdi

Siempre que utilicemos la máquina virtual desde los laboratorios seguiremos los siguientes pasos:

- Desde VirtualBox, creamos una nueva máquina virtual, con la opción **Nueva:**
  - *Nombre*: Podéis poner cualquier nombre arbitrario
  - *Tipo:* Linux
  - *Version*: Ubuntu (64 bit)
  - *Tamaño de memoria*: Como mínimo necesitaremos 2Gb. Este valor podremos cambiarlo posteriormente si es necesario desde la opción **Configuración**, del menú de VirtualBox
  - *Usar un archivo de disco duro virtual existente.* A continuación seleccionamos el fichero /usr/local/PPSS/*ppss.vdi*
  - Memoria de video: 128 Kb (desde Configuración→Pantalla→Memoria de video)

El valor del tamaño de memoria (ram) y memoria de video podremos cambiarlos posteriormente desde el menú Configuración→Sistema, y Configuración→Pantalla→Memoria de video, de VirtualBox, con la máquina virtual apagada.

Finalmente seleccionamos la máquina virtual y a continuación elegimos la opción **Iniciar**.

Necesitaremos introducir el *login* y *password*, que será **ppss** en ambos casos

Bitbucket y Repositorios remotos con Git

Es una muy buena práctica (e indispensable para futuros trabajos profesionales en grupo) utilizar una herramienta de gestión de versiones remota, para así tener todo nuestro trabajo accesible desde cualquier lugar, y con el historial de versiones, por si nos interesa recuperar alguna versión anterior, crear nuevas versiones a partir de ellas, etc.

Cada uno de vosotros os deberéis crear un r**epositorio remoto privado Git** en Bitbucket, con permisos de lectura al usuario "**ppss-ua**". En dicho repositorio almacenaréis TODO el trabajo de laboratorio del curso, según lo vayáis realizando.

Los que ya tengáis cuenta en Bitbucket, podéis utilizarla. Los que no, tendréis que crearos una cuenta nueva desde https://bitbucket.org

Para crear una cuenta (de forma gratuita) simplemente tendréis que proporcionar vuestro nombre y apellidos, un nombre de usuario y password de vuestra elección, y una dirección de correo electrónico.

Repositorio GIT en Bitbucket

Una vez que tengáis creada la cuenta, deberéis crear UN ÚNICO repositorio que alojará TODO el trabajo de prácticas que realicéis durante el curso, A MEDIDA que lo vayáis realizando. El repositorio que deberéis crear tendrá como nombre:

ppss-Gx-apellido1-apellido2-2016

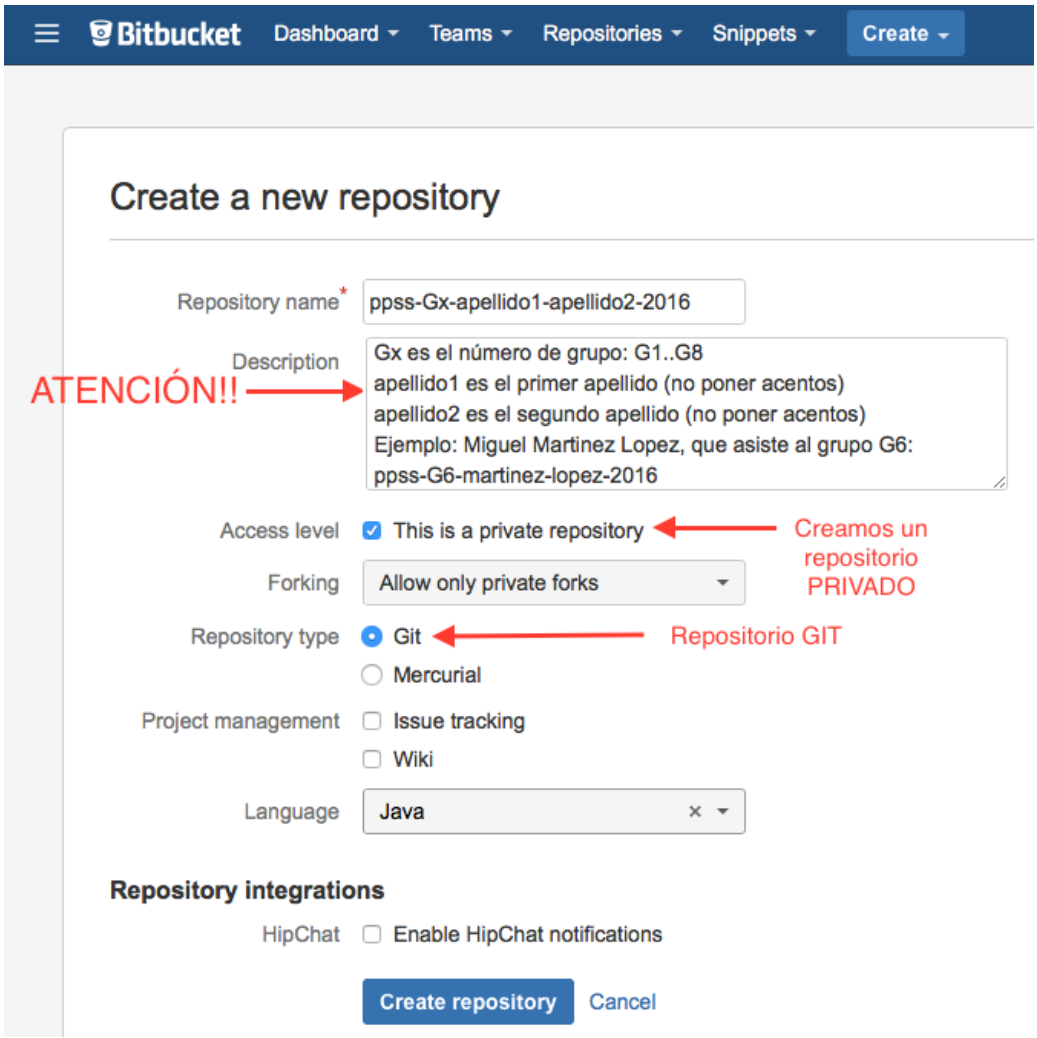
En donde:

- *Gx* es el identificador del grupo de prácticas al que asistís. Los valores posibles son: G1..G8 según la siguiente tabla

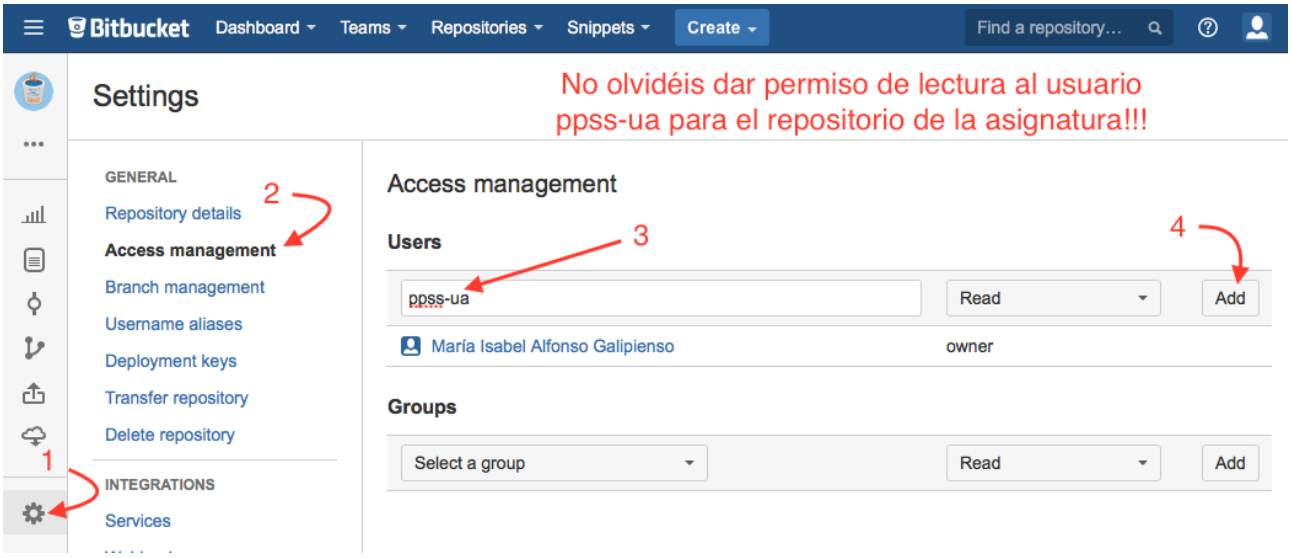
Grupo de prácticas	Identificador
Martes de 11 a 13h (en Laboratorio L18)	G1
Martes de 11 a 13h (en Laboratorio L27)	G2
Martes de 9 a 11h	G3
Lunes de 11 a 13h	G4
Miércoles de 19 a 21h	G5
Miércoles de 17 a 19h	G6
Miércoles de 9 a 11h	G7
Lunes de 17,30 a 19,30h	G8

- *apellido1* es el primer apellido (todo en minúsculas y no llevará acentos)
- *apellido2* es el segundo apellido (todo en minúsculas y no llevará acentos). Si algún alumno no tiene segundo apellido, entonces omitiremos esta parte

A continuación mostramos una captura de pantalla de la creación del repositorio:



Después de crear el repositorio, deberéis dar **permiso de lectura** al usuario **ppss-ua,** desde las "Settings" de vuestro proyecto:



Clonamos el repositorio Git

Una vez creado el repositorio vamos a ver cómo trabajar con él. Supongamos que queremos trabajar en el directorio $HOME/Practicas. El primer paso será CLONAR el repositorio en nuestra máquina (en dicho directorio). Para ello accedemos al repositorio creado en Bitbucket y seleccionamos la action "Clone" (en la parte izquierda de la ventana) y copiamos el valor de HTTPS

A continuación pegamos el contenido copiado en un terminal (desde el directorio $HOME/Practicas), y ejecutamos el comando copiado. Vemos que se crea el directorio:

$HOME/Practicas/ppss-Gx-apellido1-apellido2-2016

Seguidamente ejecutamos:

> cd ppss-Gx-apellido1-apellido2-2016

> git config user.name <nombreUsuario>

Siendo nombreUsuario el nombre que mostrará Git cuando hagamos un commit. Poned vuestro nombre y apellidos. P.ej. "Luis Lopez Perez"

> git config user.email <emailUsuario>

Siendo emailUsuario el email del usuario en Bitbuket

Cuando ejecutemos comandos git, nos pedirá la contraseña. Una forma de evitar tener que introducir la contraseña cada vez es utilizar el comando:

> git config credential.helper 'cache --timeout=1800'

Esta opción almacena nuestros credenciales de forma temporal en memoria para futuros usos de git. En este caso establecemos que dicho almacenamiento tendrá lugar durante 1800 segundos (media hora). Por defecto el valor de "timeout" es de 900

Si queremos desactivar el temporizador antes de que finalice, utilizaremos el comando:

> git credential-cache exit

Podéis consultar otras opciones para que no os vuelva a pedir la contraseña en: https://confluence.atlassian.com/bitbucketserver/permanently-authenticating-with-git-repositories-776639846.html

Después de clonar el repositorio Git, podemos comprobar que se ha creado el directorio:

$HOME/Practicas/ppss-Gx-apellido1-apellido2-2016/.git

A partir de ahora, cualquier cosa que añadamos en el directorio $HOME/Practicas/ppss-Gx-apellido1-apellido2-2016/ podrá estar sujeto al control de versiones de Git en Bitbucket.

Vamos a crear con algún editor de texto el fichero oculto **.gitignore** con el siguiente contenido:

Creamos el fichero .gitignore

```
# ignore Maven generated target folders
target
# Fichero .DS_Store (Si váis a utilizar Mac)
.DS_Store
```

Este fichero permite a Git ignorar el contenido del directorio *target*, de forma que no se subirá a Bitbucket, ya que no es necesario.

Si ahora tecleamos ***git status*** (desde nuestro directorio de trabajo) nos aparecerá el nombre del fichero que acabamos de crear en color rojo. Este comando nos mostrará todos los cambios que hayamos hecho en el directorio de trabajo (recordemos que es $HOME/Practicas/ppss-Gx-apellido1-apellido2-2016/).

Para subir a Bitbucket el fichero creado (y en general, para subir cualquier cambio que hayamos hecho en el directorio de trabajo), haremos:

Subimos los cambios a Bitbucket

```
> git add .
> git commit -m "Añadido el fichero .gitignore"
> git push
```

Es importante siempre hacer un ***git add*** antes de invocar al comando ***git commit***, ya que este comando sincroniza los cambios (de los que previamente hayamos hecho un *add)* en nuestro repositorio local. También es necesario incluir un comentario en cada *commit* describiendo los cambios que hemos realizado (con el *-m* seguido de una cadena de caracteres entre comillas, con un máximo recomendado de 50 caracteres). El último paso es subirlo al repositorio remoto (con ***git push***)

Si trabajas con tu propio ordenador, a partir de ahora, lo único que tendrás que hacer es utilizar estos tres comandos cada vez que quieras subir los cambios al repositorio de Bitbucket. Recuerda que SIEMPRE deberás hacerlo desde $HOME/Practicas/ppss-Gx-apellido1-apellido2-2016/

Si trabajas en el laboratorio, cuando llegues a casa y trabajes en tu ordenador, tendrás que (la primera vez):

Secuencia de trabajo (la primera vez)

1. clonar el repositorio de Bitbucket
2. trabajar
3. subir los cambios a Bitbucket

Las siguientes veces, supongamos que has trabajado desde los ordenadores del laboratorio, y luego quieres seguir trabajando en casa (y en casa ya habías clonado el repositorio). En ese caso tendrás que hacer primero un *pull* para descargarte los cambios que hiciste en el laboratorio antes de continuar trabajando:

Secuencia de trabajo (el resto de veces)

> git pull
2. trabajar
3. subir los cambios a Bitbucket

Puedes consultar información sobre Git, desde el menú de ayuda de Bitbucket (en la parte superior derecha de la pantalla (opción "Learn Git"). Otro recurso útil puede ser: http://gitref.org/index.html

Maven

Maven es una **herramienta de construcción** de proyectos Java. Por definición, la construcción de un proyecto (*build*) es un **proceso** que comprende **varias** tareas, tales como compilación, *linkado*, pruebas, empaquetado, ejecución, despliegue.... Otros ejemplos de herramientas de construcción de proyectos son Make (para lenguaje C) y Ant (también para lenguaje Java).

En este curso utilizaremos Maven (versión 3.3.9) para construir nuestros proyectos. Maven puede utilizarse desde línea de comandos (comando **mvn**) o desde un IDE. En nuestro caso utilizaremos ambos.

Script de compilación

Normalmente las herramientas de construcción definen lo que se denominan *scripts de compilación*, que no es más que la definición de la **secuencia** de tareas a realizar para **construir** el proyecto.

Esta secuencia de tareas dependerá del tipo de proyecto con el que estemos trabajando, por ejemplo:

- El proceso de *linkado* no es necesario si estamos trabajando en Java,
- El proceso de despliegue no es necesario si nuestra aplicación no va a ejecutarse desde un servidor web o un servidor de aplicaciones
- En ocasiones nos interesará realizar tests unitarios, o tests de integración, o incluir un análisis de pruebas de forma que no permitamos que un proyecto finalice su construcción si no cumple determinados requisitos,...

Dado que cada tipo de proyecto requiere una determinada secuencia de tareas a realizar para su construcción, se requerirán *scripts* diferentes, en función de las necesidades particulares del software que estemos desarrollando. Las herramientas de construcción (*build)* facilitan la creación de estos *scripts de compilación*.

Maven, a diferencia de Make o Ant, permite definir (o modificar) la secuencia de tareas a realizar para construir el 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.

Maven tiene **predefinidas** tres secuencias de tareas a realizar para construir proyectos. Cada una de estas secuencias se denominan **ciclos de vida**. El ciclo de vida más utilizado de los tres es el denominado ciclo de vida por defecto (*default lifecycle)*, y está formado por una lista de más de 20 tareas, denominadas **fases**. Ejemplos de fases son: compile, test, package, deploy,...

FASES

Es muy importante entender el concepto de **fase**, que 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 permite asignar acciones ejecutables que lleven a cabo el proceso de compilación del proyecto, mientras que la segunda está pensada para que se ejecuten las pruebas (lógicamente, la fase de compilación será anterior a la fase de pruebas).

El proceso de construcción de Maven genera un único fichero "empaquetado" de una determinada forma (por ejemplo jar, war, ear, ...). Dependiendo del tipo de empaquetado que se quiera generar, cada fase tiene asociadas por defecto un cierto número de acciones (hay fases que por defecto están "vacías", no tienen asociadas acciones ejecutables).

GOALS Y PLUGINS

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* (o acción) 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*.

Cada *goal* va precedida por el nombre del plugin al que pertenece seguida de ":". Por ejemplo:

- La goal *compiler:testCompile* está asociada por defecto a la fase del ciclo de vida *test-compile* y realiza la compilación del código de pruebas del proyecto (que deberá estar en el directorio /src/test/java).
- La goal *compiler:compile* está asociada por defecto a la fase del ciclo de vida *compile* y realiza la compilación del código fuente del proyecto (que deberá estar en el directorio / src/main/java).

Las goals de un plugin son *configurables* (disponen de un conjunto de variables propias que tienen valores por defecto que podemos cambiar). Por ejemplo podemos cambiar la fase a la que se asocia (se ejecuta) dicha goal en un proyecto. Para conocer qué goals están disponibles para cada plugin tenemos que consultar su documentación en internet. Por ejemplo, el plugin *compiler* está documentado en: https://maven.apache.org/plugins/maven-compiler-plugin/plugin-info.html.

Podemos cambiar la secuencia de ejecución de acciones realizadas por el proceso de construcción simplemente asociando diferentes *goals* a las diferentes fases, y podemos añadir y configurar nuevas acciones simplemente incluyendo los *plugins* que las contienen. Todo ello se realiza a través de un fichero de configuración denominado **pom.xml**. Cualquier proyecto construido con Maven tiene un fichero de configuración pom.xml asociado.

Coordenadas maven

Como resultado del proceso de construcción de un proyecto Maven se obtiene siempre un artefacto (fichero) que se identifica mediante sus **coordenadas**, separadas por ":". Para identificar un artefacto Maven se utilizan, como mínimo, tres campos, o coordenadas, de forma que cualquier artefacto Maven se especifica de forma única (no hay dos artefactos con las mismas coordenadas) como ***groupId:artifactId:version***.

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

Ejemplos de coordenadas: org.ppss:practica1:1.0-SNAPSHOT, org.ppss:practica1:2.0-SNAPSHOT, org.ppss:proyecto3:war:1.0-SNAPSHOT. Los dos primeros ejemplos identifican ficheros que contienen dos versiones diferentes, ambos con extensión .jar (por defecto se asume este tipo de empaquetado). Concretamente el nombre de los ficheros correspondientes son practica1-1.0-SNAPSHOT.jar y practica1-2.0-SNAPSHOT.jar, respectivamente, situados ambos en nuestro repositorio local en la ruta /org/ppss/. El tercer ejemplo hace referencia a un fichero con extensión *.war* (si el empaquetado no es jar, entonces hay que indicarlo de forma explícita con una cuarta coordenada), concretamente se trata del fichero proyecto3-1.0-SNAPSHOT.war, que podemos encontrar en nuestro repositorio local en la ruta /org/ppss/.

Estructura de directorios maven

Todos los proyectos Maven siguen una estructura de directorios similar. Así, por ejemplo, el código fuente del proyecto lo situaremos 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 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 otro lado, cualquier librería externa (ficheros .jar) que utilice nuestro proyecto Maven, puede indicarse en el fichero pom.xml, de forma que, si no se encuentra físicamente en nuestro disco duro, Maven la descarga automáticamente de sus **repositorios**. Es más, si utilizamos una librería, que a su vez depende de otra, Maven automáticamente se encarga de descargarse también esta, y así sucesivamente. Esto hace que nuestros proyectos “pesen” poco, ya que tanto el directorio target, como cualquier librería utilizada por el proyecto se generarán y descargarán automáticamente 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 desarrollado (el directorio src contiene todos los fuentes del proyecto).

Repositorios locales y remotos

Todos los artefactos generados y/o utilizados por Maven se almacenan en **repositorios**. Maven mantiene una serie de repositorios **remotos**, que alojan todos los *plugins* y librerías que podemos utilizar. Cuando ejecutamos Maven por primera vez en nuestra máquina, se crea el directorio .m2, que actuará como nuestro **repositorio local**. Cuando iniciamos un proceso de construcción Maven, primero se consulta 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.

Por todo lo anteriormente mencionado, Maven se ha convertido en un estándar de “facto” para construir proyectos java, y por lo tanto, una herramienta que es importante conocer.

Ejecución de Maven

Para iniciar el proceso de construcción de Maven, utilizaremos 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. Si incluimos una o más fases, o *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 plugi2:goal4

- El comando **mvn fase**, 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.
- El comando **mvn *plugin:goal*** ejecuta únicamente la *goal* que hemos especificado

Utilizaremos Maven durante todo el curso, exceptuando aquellas prácticas que no requieran una implementación de código.

Netbeans y Maven

Netbeans es un IDE muy utilizado para trabajar con diferentes tipos de aplicaciones, entre ellas aplicaciones java. Utilizaremos Netbeans 8.1, y trabajaremos SIEMPRE con proyectos Maven.

Nota sobre la instalación de Netbeans: Asumiremos que la instalación de Netbeans utiliza inglés en lugar de castellano. Es posible que en los laboratorios tengáis la versión en castellano. Si es así, la forma de cambiarlo sin reinstalar Netbeans es la siguiente:

- Editar el contenido del fichero $HOME/netbeans-8.1/etc/netbeans.conf, de forma que:

  Modificamos la línea 46: netbeans_default_options="-J-client -J-Xss2m -J-Xms32m -J-Dapple.laf.useScreenMenuBar=true ..."

  De forma que añadamos al final (o al principio): -J-Duser.language=en

  Por ejemplo, podría quedar como:

  netbeans_default_options="-J-client -J-Xss2m -J-Xms32m -J-Dapple.laf.useScreenMenuBar=true -J-Dapple.awt.graphics.UseQuartz=true -J-Dsun.java2d.noddraw=true -J-Dsun.java2d.dpiaware=true -J-Dsun.zip.disableMemoryMapping=true J-Duser.language=en"

Aunque el uso de un IDE es conveniente, (o incluso imprescindible), no es nada aconsejable que la construcción de nuestra aplicación sea totalmente dependiente de éste, por razones obvias. Por eso, aunque utilicemos Netbeans para desarrollar nuestra aplicación, y dado que vamos a trabajar con proyectos Maven, nos hemos asegurado de que Netbeans ejecute la versión de maven que tenemos instalada en /usr/local, en lugar de su propio Maven (que viene incluido con la instalación). Puedes consultar el valor de "Maven Home", dentro del panel de opciones de Java (Tools>Options) De esta forma la construcción de nuestra aplicación no se verá afectada por un eventual cambio de IDE.

Netbeans identifica los proyectos Maven con una "M" en la parte superior izquierda del icono que representa cada tipo de proyecto.

Ventana Navigator

Cuando trabajemos con proyectos Maven, en el panel de navegación (ventana **Navigator** situada habitualmente justo debajo de la ventana de proyectos) se muestra información sobre las *goals* asociadas al proyecto con el que estemos trabajando en ese momento. Si pulsamos sobre la rueda dentada de color verde que aparece en la parte inferior de la ventana, se mostrarán todas las goals asociadas al proyecto. Éstas pueden ejecutarse desde el IDE desde su menú contextual (menú que aparece al pulsar con el botón derecho del ratón), seleccionando la opción: "Execute Goal" .

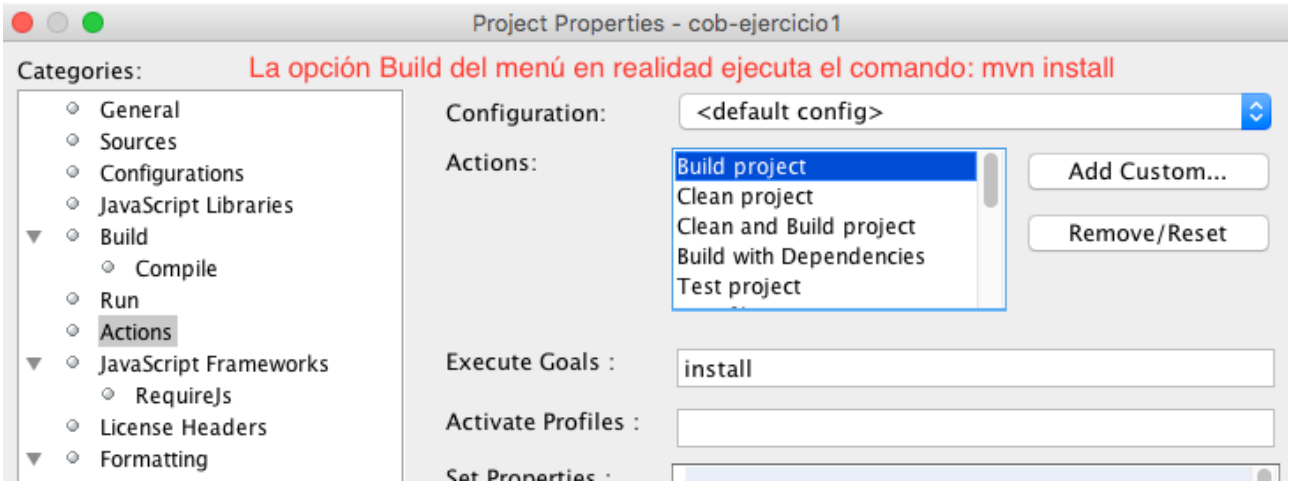
Si seleccionamos un proyecto Maven, veremos un determinado conjunto de opciones desde el menú contextual, concretamente podremos ejecutar la opción Build, Clean and Build, Run,

Test,... Es MUY IMPORTANTE tener claro que estos nombres NO se refieren de forma DIRECTA a ninguna fase ni goal del comando **mvn**, sino que son los nombres que Netbeans da a la ejecución de ciertos comandos Maven.

Actions y comandos Maven

Cada una de las opciones: Build, Clean and Build,... tienen asociada una **Action**, que no es más que el comando Maven se se ejecutará cuando pinchemos con el ratón sobre dicha opción del menú contextual. Para ver cuál es el comando maven que ejecutaremos si hacemos "click" sobre alguna de los elementos de dicho menú, iremos a la opción "Properties" que aparece al final de los ítems del menú contextual de nuestro proyecto, y elegiremos el elemento "Actions" del subpanel denominado "Categories".

En la siguiente figura podemos ver que la *Action* asociada a la opción de menú "Build" consiste en el comando "mvn install". Es decir, si en lugar de elegir dicha opción desde el IDE, tecleamos el comando "mvn install" en un terminal (desde el directorio que contiene el pom.xml de nuestro proyecto) estaremos ejecutando LO MISMO!!!.



También podemos observar que podemos eliminar o modificar cualquier opción del menú contextual (con el botón "Remove/Reset"), o añadir cualquier opción que nos interese, utilizando el botón "Add Custom...", lo cual nos resultará muy útil cuando queramos realizar una construcción particular que utilice fases o *goals* no contempladas por defecto desde el IDE.

El comando "mvn install" ejecuta la fase *install* del ciclo de vida por defecto. En esta fase se copiará en nuestro repositorio local el artefacto generado (en nuestro caso se tratará de un fichero .jar)

Finalmente, comentaremos que lo que muestra la ventana **Projects**, no son los ficheros tal y como aparecen en el disco duro (esto podremos verlo desde la ventana **Files**). Por ejemplo, en los proyectos Maven, podremos ver el contenido del fichero *pom.xml* dentro del icono con forma de carpeta que aparece en último lugar denominada "*Project Files*". Sin embargo, dicho fichero *pom.xml* no se encuentra en ninguna subcarpeta, sino directamente en la raiz del directorio que contiene los ficheros de nuestro proyecto. En nuestro proyecto tampoco habrá ningún subdirectorio con el nombre "Source Packages", "Test Packages", "Dependencies",...

La **estructura física** de nuestro proyecto podemos consultarla desde la ventana **Files** y podemos comprobar que inicialmente (antes de construir el proyecto por primera vez), contiene únicamente el fichero **pom.xml**, y el directorio **src**.

Ejercicios

Para realizar los ejercicios de esta práctica proporcionamos un proyecto maven en el directorio "P1-netbeans" (dentro de Plantillas-P1"). Este directorio contiene un proyecto Maven (directorio src más el fichero pom.mxl con la configuración de construcción del proyecto). El propósito de esta práctica es familiarizarnos con el entorno de trabajo: Netbeans, Maven, y automatización de pruebas.

1. **Ejercicio 1**. Crea el repositorio Git desde Bitbucket que contendrá TODO el trabajo de prácticas. Para ello sigue las instrucciones del apartado "Bitbucket y Repositorios remotos con Git" de este documento. Una vez que hayas creado el repositorio clónalo en tu máquina local. A partir de ahora todo tu trabajo tendrá que estar en este directorio. En lo sucesivo, nos referiremos a él como el **directorio de trabajo**. Crea el fichero *.gitignore* y sube los cambios a Bitbucket.
2. **Ejercicio 2**. Copia el directorio "P1-netbeans" en tu directorio de trabajo, y abre el proyecto desde Netbeans. Realiza lo siguiente:
   A) Anota (y recuerda) la estructura de directorios del proyecto y observa que el código fuente y el código de pruebas están físicamente "separados", pero que lógicamente están en el mismo "lugar", ya que pertenecen al mismo paquete. Esta estructura de directorios es común a CUALQUIER proyecto Maven. Es importante conocerla, ya que el proceso de construcción que realiza Maven asume que determinados artefactos 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
   B) Muestra en el editor la configuración de nuestro proceso de construcción (fichero pom.xml). Concretamente, contiene tres elementos: coordenadas del proyecto, propiedades y dependencias.

   Indica cuáles son exactamente las coordenadas de nuestro proyecto maven.

   La etiqueta <properties> se utiliza para definir y/o asignar/modificar valores a determinadas "variables" de la configuración de nuestro proceso de construcción. Podemos hacer referencia a 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.

   Finalmente especificamos la librería que utilizamos en nuestros tests (fíjate que hemos indicado "test" como valor de *scope*, lo que significa que esta librería sólo se necesita durante la compilación de los tests). Observa también, que cualquier artefacto utilizado por maven se identifica por sus coordenadas. ¿cuáles son en concreto para esta dependencia?

   El pom.xml de nuestra construcción no incluye ningún plugin adicional a los plugins que ya tiene asociados por defecto (y que podemos ver en la ventana "Navigator").
   C) La **clase Triángulo** contiene la implementación de un método cuya especificación asociada es la siguiente: Dados tres enteros como entrada, que representan las longitudes de los tres lados de un triángulo, y cuyos valores deben estar comprendidos entre 1 y 200, el método *tipo_triangulo()* devuelve como resultado una cadena de caracteres indicando el tipo de triángulo en el caso de que los tres lados formen un triángulo válido. El tipo puede ser: "Equilatero", "Isosceles", o "Escaleno". 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. Si esto no se cumple, el método devolverá el mensaje "No es triángulo". Si alguno de los tres lados: a, b, ó c, es mayor que 200 o inferior a 1 mostrará 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.

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.

D) La **clase TrianguloTest** contiene la implementación de cuatro casos de prueba (los cuatro métodos anotados con @Test) asociados a la especificación del apartado C anterior. ¿Cuáles son exactamente? Identifícalos como C1, C2, C3, y C4, y muéstralos en una tabla con cuatro filas con la siguiente información:

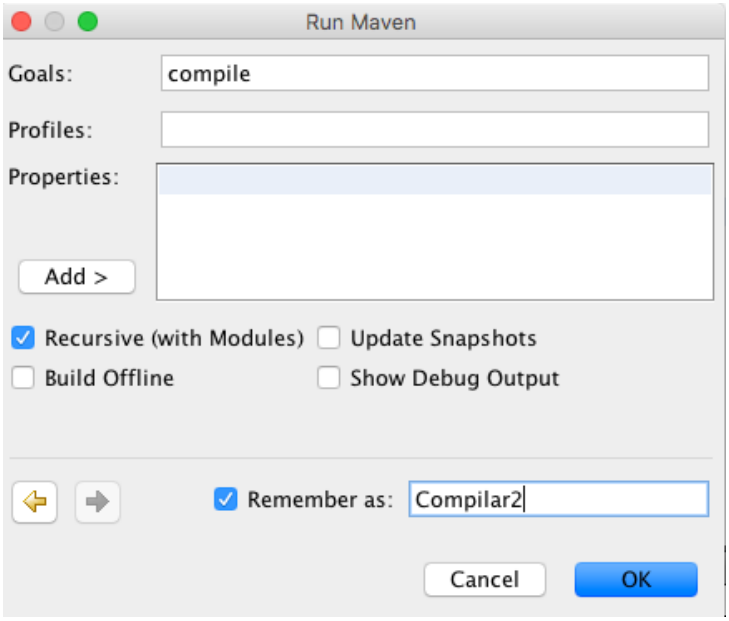
Identificador del Caso de prueba	Dato de entrada 1	...	Dato de entrada n	Resultado esperado	Resultado real

Esta tabla se denomina "**Tabla de casos de prueba**". Recuerda esta estructura porque la utilizaremos en sesiones posteriores cuando diseñemos los casos de prueba.

E) Observa la implementación de cada test y verás que todos ellos siguen la misma lógica de programa. Anota el algoritmo que refleja dicha lógica. Recuérdalo porque lo utilizaremos también en sesiones posteriores.

3. **Ejercicio 3**. Vamos a compilar el programa. Podemos hacerlo de dos formas: desde el menú contextual del proyecto, o desde la ventana Navigator. Vamos a realizar ambas.

A) Comenzaremos **creando una nueva Action** (desde el menú contextual del proyecto, según se ha explicado en el apartado "Netbeans y Maven" de este documento). La nueva action se llamará "Compilar", y tendrá asociado el comando maven "mvn compile" (ejecución de la fase "compile" de Maven). Una vez creada la nueva action, en el menú contextual del proyecto, y desde la opción "Custom" nos aparecerá una nueva opción con el nombre "Compilar" que podremos utilizar a partir de ahora. Pruébala, verás que se ha creado el directorio *target* (para ver este directorio tienes que hacerlo desde la ventana "Files"). A continuación ejecuta "Clean" desde el menú contextual del proyecto. Esta acción ejecuta el comando "mvn clean" que "borra" el directorio *target* creado (y todo su contenido). Otra forma de crear Actions es a través de la opción "Custom->Goals" desde el menú contextual del proyecto. Prueba a crear otra Action desde aquí con el nombre "Compilar2", asociándole la fase "compile", tal y como se muestra en la siguiente figura:



Puedes observar desde la ventana "Files" que después de crear las *Actions* se ha creado un nuevo fichero **nbactions.xml** en el directorio raíz de nuestro proyecto Maven. Puedes ver que contiene un fichero xml con la información de las dos nuevas *Actions* que hemos creado (compilar y compilar2).

B) De forma alternativa, y como ya se ha explicado, la ventana "**Navigator**" nos muestra las *goals* asociadas con nuestro pom.xml. Concretamente, la fase "compile" de maven tiene asociada por defecto la goal "*compiler:compile*" (podríamos verlo ejecutando la opción "Show documentation" desde el menú contextual de este ítem en la ventana "Navigator"). Por lo tanto, para compilar el programa podemos hacer doble click sobre la goal "*compiler:compile*". Pruébalo, verás que se vuelve a crear el directorio target. Por lo general invocaremos siempre los comandos maven a través de las fases del ciclo de vida, en lugar de ejecutar *goals* individuales, por lo que nos crearemos las Actions que necesitemos.

C) Anota la nueva **estructura de directorios creada** y qué artefactos contienen. Esta nueva estructura, así como la ubicación de los artefactos también es común para CUALQUIER proyecto maven. Ahora vamos a ejecutar la fase “clean”. Observa lo que ocurre y anota lo que realiza dicha fase. Ahora vuelve a compilar el proyecto. Fíjate en la secuencia de acciones que se muestran en la ventana inferior y en que NO se han ejecutado los tests.

D) Ahora vamos a ejecutar los tests. Para ello necesitamos ejecutar la fase de maven "test". Netbeans ya nos proporciona una Action para este propósito, accesible a través de la opción "Test" del menú contextual de nuestro proyecto. Al ejecutar dicha opción verás que en la ventana "Output" en la parte inferior del IDE veremos en todo momento lo que ocurre al ejecutar cualquier comando maven. Concretamente, deberemos ver esto:

```
Results :
Failed tests:   testTipo_trianguloC3(ppss.TrianguloTest):
expected:<[Valor c fuera del rango permitid]o> but was:<[No es un
triangul]o>

Tests run: 5, Failures: 1, Errors: 0, Skipped: 0
------------------------------------------------------------------------
[INFO] BUILD FAILURE
------------------------------------------------------------------------
```

En este caso “**Tests run**” indica el número total de tests ejecutados.“**Failures**” indica el número de tests cuyo resultado esperado NO coincide con el real. Observa que junit proporciona un tercer tipo de resultado: “**Error**”, del que hablaremos 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. De hecho vemos también que por pantalla se muestra la razón del fallo de ejecución del test. 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 fases del proceso, concretamente en la fase de pruebas.

Alternativamente, podemos ver los resultados de ejecución de los casos de prueba de forma gráfica en la ventana "Test Tesults" que Netbeans ha abierto en la parte inferior del IDE. Desde el margen izquierdo de esta ventana aparecen opciones para mostrar/ ocultar los tests que han tenido éxito o tienen errores. La primera vez que ejecutéis los tests veréis que por defecto no muestra los tests que han tenido éxito. **Nota**: Si esta ventana no se abre de forma automática, podemos mostrarla desde "Window->IDE Tools->Test Results".

E) Para poder concluir nuestro proceso de construcción con éxito (BUILD SUCCESS) necesitamos eliminar el problema que provoca el fallo de ejecución. En este caso se trata de averiguar por qué el informe del resultado de la ejecución del caso de prueba correspondiente es un fallo. Obviamente 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 que realmente hemos obtenido. Identifica la causa y modifica el código convenientemente (recuerda que este proceso se llama DEPURACIÓN, o *debugging*). A continuación vuelve a ejecutar la fase test (repite el proceso hasta que los cuatro tests estén en "verde", y el proceso de construcción termine con: BUILD SUCCESS

F) 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 razona la conveniencia o no de incluir C5 al conjunto de tests. De la misma forma razona si son necesarios los tests C2 y C3. Basándote en tu razonamiento anterior, piensa en dos posibles casos de prueba adicionales que "aporten valor" al conjunto de casos de prueba (no sean innecesarios) justificando tu respuesta.

4. **Ejercicio 4**. La clase Matricula contiene el método `calculaTasaMatricula()` que devuelve el valor de las tasas de matriculación de un alumno en función de la edad, de si es familia numerosa, y si es o no repetidor, de acuerdo con la siguiente tabla (asumiendo que se aplican sobre un valor inicial de tasa=500 euros):

	Edad < 25	Edad < 25	Edad 25..50	Edad 51..64	Edad ≥ 65
Edad	SI	SI	SI	SI	SI
Familia Numerosa	NO	SI	SI		
Repetidor	SI				
Valor tasa-total	tasa + 1500	tasa/2	tasa/2	tasa -100	tasa/2

A) En este caso, hemos proporcionado la implementación de un único test, en la clase MatriculaTest. Implementa 5 nuevos tests que no sean "redundantes", y ejecuta dichos tests. Si encuentras algún error, depúralo.

B) Rellena una tabla de casos de prueba, con los cinco tests que ya tienes implementados y añade tantos casos de prueba como consideres necesarios para detectar posibles errores de ejecución en la implementación del método. En las siguientes sesiones hablaremos sobre cómo obtener dicha tabla de casos de prueba, de forma efectiva y eficiente.

C) Pregunta a tus compañeros cuántas filas tiene la tabla que han confeccionado. Pregunta a tus compañeros si han sido capaces de encontrar errores en el código ejecutando alguno de los nuevos tests implementados. Indica en qué te has basado para decidir los 5 nuevos casos de prueba que te hemos pedido, y contrasta con tus compañeros en qué se han basado ellos para tomar su decisión.

Y ya para terminar:

Lo que debes hacer SIEMPRE al final de tu sesión de trabajo: Sube el trabajo realizado a Bitbucket siguiendo las instrucciones del apartado "Bitbucket y repositorios remotos con Git", **ANTES DE APAGAR LA MÁQUINA.** Es decir, desde tu directorio de trabajo, ejecuta los comandos:

- git add .
- git commit -m "P1 ejercicio1 terminado"
- git push

Recuerda incluir un mensaje describiendo el trabajo realizado. Ten en cuenta que **CADA VEZ que se inicia una sesión en Linux en las máquinas del laboratorio, la MÁQUINA VIRTUAL SE RESTAURA A SU ESTADO ORIGINAL**. Eso significa que si no lo has subido tu trabajo a Bitbucket, éste se perderá y no lo podrás recuperar.

MUY IMPORTANTE (para todo el curso): Recuerda que tu trabajo de prácticas te permitirá comprender y asimilar los conceptos vistos en las clases de teoría. Y 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.

La evaluación será presencial, individual, y de forma ESCRITA. Por lo que es muy importante que seáis capaces de EXPRESAR, de forma clara y ordenada, y por ESCRITO, los conocimientos teórico-prácticos adquiridos, por lo que te recomendamos que tomes apuntes en clase.

Cada sesión en el laboratorio está pensada para que asimiléis los conceptos vistos en el aula de teoría. Por ello deberéis analizar y anotar la “conexión” de cada sesión práctica con la de teoría, y muy especialmente destacar las ideas fundamentales, para posteriormente trabajar el detalle (y no al revés).

Es fundamental que confeccionéis vuestro propio material de estudio, utilizando el material que os proporcionaremos (transparencias, enunciados de prácticas, ejemplos,...). Concretamente, en prácticas deberéis anotaros vuestras propias conclusiones, pasos que habéis seguido, respuestas las preguntas planteadas,..., de forma que no sea necesario volver a hacer las prácticas para preparar los exámenes.

Recuerda también que la asimilación de los conceptos requiere una dedicación PERSONAL de forma CONTINUADA durante TODO el cuatrimestre.