Sesión L02- Clases y objetos

01 Primeros pasos con Eclipse y Maven

Eclipse es un IDE (***I**ntegrated **D**evelopment **E**nvironment*), es decir, un entorno de desarrollo, con el que podremos **editar**, **compilar** y **ejecutar** nuestros programas java.

Es uno de los IDEs más usados para programar en Java (y que además nos permite programar con otros lenguajes). Es un IDE muy completo, con multitud de opciones. Lógicamente, no vamos a verlas todas. Iremos introduciéndolas a medida que las vayamos necesitando.

Eclipse organiza el trabajo en ***Workspaces***. Un *workspace* es un DIRECTORIO que contendrá **Proyectos**. Cada proyecto contiene el código fuente de una aplicación java que podremos compilar y ejecutar.

Cuando iniciemos eclipse por primera vez, nos propondrá el *workspace* ***/home/pa/eclipse-workspace***. Usaremos dicho *workspace*.

Cada ***workspace*** contiene una carpeta oculta `.metadata`, que Eclipse usa para guardar la configuración de dicho espacio de trabajo, así como la lista de proyectos que contiene.

De la misma forma, cada ***proyecto*** en eclipse contiene una fichero oculto `.project` con información sobre la configuración relativa a dicho proyecto.

Tanto la carpeta `.metadata`, como el fichero `.project` son específicos de Eclipse (y se generan de forma automática). Por lo tanto, no nos interesa guardar en nuestro repositorio esta información. Básicamente sólo estamos interesados en guardar el código fuente (ficheros.java) y los ficheros `*.launch` que contienen los comandos que usaremos para compilar, ejecutar, borrar los ejecutables, ...)

Una utilidad muy importante de este IDE son las **vistas (*views*)** , que se muestran en diferentes "pestañas" o "paneles", y nos permiten mostrar nuestro proyecto desde diferentes "puntos de vista". Las vistas están accesibles desde "***Window→Show View***", y podemos mostrarlas u ocultarlas a voluntad, según las necesitemos o no en un momento determinado.

Algunas de las **vistas** que usaremos son:

Project Explorer: aquí veremos los diferentes proyectos maven del *workspace*, con la estructura de ficheros correspondiente.

Outline: Es una vista abstracta de nuestras clases. Es decir, muestra la información que define cada clase java. Aquí veremos (cuando seleccionemos una clase java en la vista *project explorer*) el nombre de la clase, el paquete al que pertenece, sus atributos y sus métodos.

Console: Se abrirá automáticamente cuando compilemos/ejecutemos nuestros programas. Nos muestra todos los mensajes generados por maven durante la construcción del proyecto y también los resultados de la ejecución del mismo. La construcción de cualquier proyecto maven siempre termina con **BUILD SUCCESS** (si todo ha ido bien), o **BUILD FAILURE** (si se genera algún error durante el proceso.

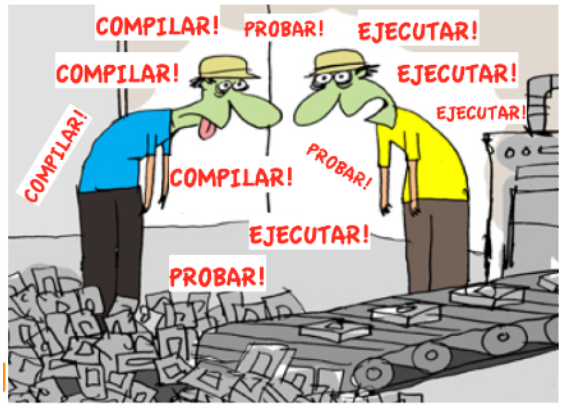
Terminal: Como sugiere su nombre, es un terminal linux, que nos permitirá ejecutar comandos de forma similar al terminal (QTerminal) que lanzamos desde el escritorio de la máquina virtual.

Las vistas en Eclipse, se agrupan en **Perspectivas** (***Perspective***). Una perspectiva no es más que un determinado conjunto de vistas. Siempre trabajaremos con la perspectiva ***Java.***

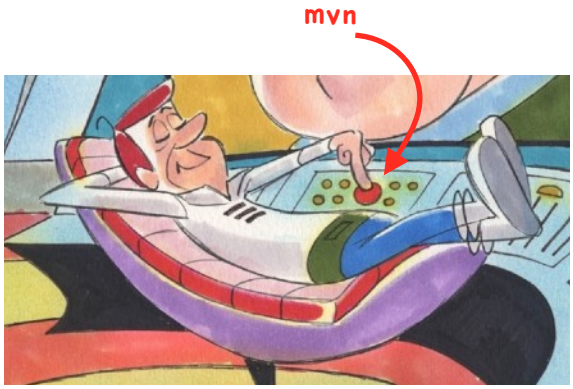
Proceso de *Construcción de Proyectos (build)*

El proceso de **Construcción** (***build***) de un proyecto consiste en una secuencia de acciones que nos permiten, a partir de un conjunto de ficheros de texto (que contienen el código fuente de nuestro programa), conseguir ejecutar dicho programa. Para ello necesitaremos secuenciar actividades como: compilar, ejecutar pruebas, desplegar en un servidor, etc...

Dicha secuencia de tareas que debemos realizar para construir el proyecto se pueden realizar de forma automática, permitiendo que los desarrolladores puedan centrarse en las tareas de desarrollo de código. Maven es una herramienta de este tipo (comando mvn)



PROCESO DE CONSTRUCCIÓN MANUAL



PROCESO DE CONSTRUCCIÓN AUTOMÁTICO

MAVEN.

Eclipse permite trabajar con diferentes tipos de proyectos. Uno de esos tipos son los **proyectos Maven**. Maven es una herramienta de **construcción** de proyectos (sirve para realizar de forma automática las acciones de compilación, pruebas, ejecución, ..., del código escrito en java). Maven es una herramienta muy potente y muy utilizada por los desarrolladores Java. A los proyectos que usan Maven como herramienta de construcción se les conoce como proyectos Maven.

Iremos explicando algunos conceptos de Maven y expondremos las **ventajas** de su uso. Una de esas ventajas es que define una determinada estructura (forma de organizar los ficheros del proyecto), de forma que TODOS los proyectos Maven tienen que **organizar sus ficheros** de la MISMA forma. Por lo tanto, cualquier desarrollador que conozca Maven, estará familiarizado con la estructura de los ficheros del proyecto y podrá realizar su trabajo de forma más fácil y rápida.

Un **proyecto Maven** es un directorio que contiene, **necesariamente**:

- fichero ***pom.xml*:** nos va a permitir construir el proyecto. El proceso de construcción recibe el nombre de **BUILD**. Podremos configurar el proceso de construcción en el pom del proyecto, e indicar qué acciones o tareas vamos a realizar y en qué orden. Ejemplos de acciones o tareas son: compilar, probar, ejecutar, desplegar ..., de forma que podremos establecer un orden entre ellas, por ejemplo: compilar-ejecutar o compilar-probar-ejecutar, o compilar-probar-desplegar, ...)
- carpeta ***src/main/java***: contiene el código fuente de nuestro proyecto, formado por un conjunto de **clases** (ficheros `*.java`), organizados en **paquetes**.
- carpeta ***src/main/resources***: contiene ficheros adicionales no de código, necesarios para ejecutar el proyecto (por ejemplo ficheros de imagen, o cualquier otro archivo que use nuestro programa). Estos ficheros serán copiados de forma automática por maven en la carpeta ***target*** durante el proceso de construcción.

Opcionalmente podemos tener las carpetas ***src/test*** que contendrán el código fuente de pruebas. Nosotros NO vamos a necesitar estas carpetas.

Durante el proceso de construcción del proyecto, maven genera un directorio denominado ***target***, en el que se almacenarán los ficheros obtenidos durante dicho proceso, por ejemplo, los ejecutables java (ficheros `*.class`).

Comandos maven que usaremos habitualmente en las prácticas para construir el proyecto:

- `mvn clean`. Elimina el directorio *target* y todo su contenido. En cualquier caso, aunque no usemos este comando, nunca vamos a subir el directorio *target* a GitHub, ya que lo podemos generar en cualquier momento volviendo a construir el proyecto.
- `mvn compile`. Copia los ficheros del directorio *src/main/resources* en el target, y compila los ficheros java del directorio ***src/main/java***.
- `mvn exec:3.4.1:java -Dexec.mainClass=<nombreClase>`. Ejecuta el método ***main()*** de la clase `<nombreClase>` que se encuentra en el fichero **.class** correspondiente. El método *main()* es el "punto de partida" de la ejecución de cualquier programa java. Si no hay un método main() NO podremos ejecutar nuestra aplicación.

Run Configurations de Eclipse.

Para construir el proyecto, ejecutaremos alguno de los comandos maven anteriores. SIEMPRE podemos hacerlo desde un terminal, sin necesidad de ningún IDE, aunque por comodidad lanzaremos los comandos desde Eclipse. Para lo cual tendremos que crear lo que Eclipse llama ***Run Configurations***.

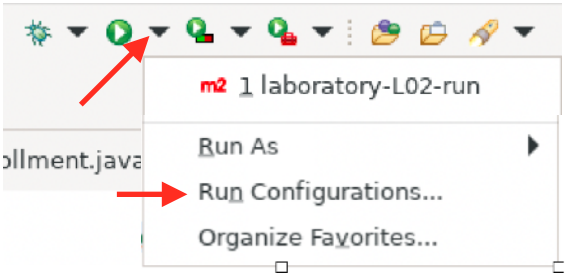
Dependiendo de la práctica, crearemos una o varias *Run configurations* para cada proyecto maven. Una ***Run Configuration*** no es más que un fichero con extensión `.launch` que contiene uno o varios comandos para construir el proyecto. Es decir, en lugar de teclear nosotros los comandos maven desde un terminal, Eclipse se encargará de hacerlo por nosotros cuando pulsemos el "botón" correspondiente (asociado a alguna *Run Configuration*),

Las *Run Configurations* que creemos (ficheros con extensión .launch) los guardaremos en la carpeta ***eclipse-launch-files***, en la raíz de nuestro proyecto maven (al mismo nivel que el pom.xml, y el directorio *src*).

Pasos para crear una nueva ***Run Configuration*** con Eclipse.

Una forma de hacerlo es: desde el menú contextual del proyecto (con botón derecho), elegimos la opción "*Run As→Run Configurations...*".

De forma alternativa también podemos acceder con botón izquierdo sobre el desplegable que aparece a la derecha del icono con forma de triángulo verde, y desde ahí seleccionamos "*Run Configurations...*":



A continuación, desde el panel de la izquierda, seleccionamos "*Maven Build*" y creamos una nueva *configuration* con botón derecho y la opción "*New Configuration*"

Para cada ***Configuration*** necesitamos indicar (desde la ***pestaña Main***):

- un **nombre**
- el **directorio base** (este será el directorio raíz del proyecto maven que queramos construir). Podemos pulsar el botón ***Workspace...*** y seleccionar el proyecto maven.
- el comando o comandos maven que vamos a usar (campo ***Goals***)
- Adicionalmente, desde la pestaña ***Common***, seleccionaremos "***Shared File***" e indicaremos la ruta donde guardaremos el fichero `.launch` (que será la subcarpeta con nombre ***eclipse-launch-files,*** en el directorio raíz de nuestro proyecto maven).

Pasos para usar una ***Run Configuration*** al importar un proyecto maven.

En algunas prácticas os proporcionaremos la carpeta ***eclipse-launch-files*** con las *Run Configuration* ya creadas. En este caso, Eclipse cargará los ficheros correspondientes a dichas *Run Configurations* y ya tendremos todos los datos.

PERO como puede ser que vosotros trabajéis con otra carpeta de trabajo diferente, es necesario que COMPROBEMOS, **siempre que importemos un proyecto maven que os hayamos proporcionado los profesores**, si el directorio base y la ubicación donde se guarda el fichero .launch es correcta (deberá estar marcada la opción *Shared file*, de la pestaña *Common*).

Pasos para ejecutar una ***Run Configuration***.

Una vez que hemos creado una *Run Configuration,* podemos ejecutarla pulsando el botón ***Run***, de la ventana que contiene los datos de dicha *Run Configuration*.

Cuando se trate de una *Run Configuration* que ya hemos ejecutado como mínimo una vez, ésta nos aparecerá cuando pulsemos sobre el desplegable del botón con el triángulo verde y simplemente hacemos doble click sobre ella para ejecutarla.

En dicho desplegable siempre veremos todas las *Run configurations* que hayamos ejecutado de todos los proyectos que tengamos abiertos en el *Workspace* de Eclipse en ese momento.

Cuando importamos un proyecto maven que esté gestionado por git (y todos los proyectos de nuestro directorio de trabajo lo estarán), Eclipse lo detecta, y nos permite realizar operaciones como *git add*, *git commit*, *git push*, *git pull*... sin necesidad de usar un terminal y teclear cada vez dichos comandos.

Por comodidad, puedes usar el IDE para trabajar con GIT, pero RECUERDA que debes aprenderte los comandos para el examen. Mi consejo es que uses regularmente el terminal para sincronizar los cambios, de esta forma te aprenderás los comandos git más fácilmente.

Una vez que has importando el proyecto verás que Eclipse te va mostrando con diferentes iconos el estado de los ficheros de tu directorio de trabajo (por ejemplo si estos han cambiado desde el último *commit* y no están en el área *stage*, se muestran con el símbolo ">". De esta forma no necesitas usar el comando "*git status*" para ver en qué estado se encuentran dichos ficheros.

Desde el menú contextual de un archivo (o proyecto), y seleccionando la opción ***Team***, podemos añadir dicho fichero (o carpeta) al stage (opción "***+ Add to index***"), hacer un *commit* (opción "***Commit ...***"), push (opción "***Push to origin***", pull (opción "***Pull***") entre otras.

Configuración de Eclipse.

Vamos a usar Eclipse porque es cómodo tener todas las herramientas necesarias para desarrollar nuestros programas en una única aplicación, lo que hará que seamos más eficientes programando. Ahora bien, hay muchos IDES, y no es buena idea depender completamente de un IDE en particular. Por lo tanto vamos a configurar ciertos parámetros en Eclipse para minimizar nuestra dependencia con dicho IDE.

Dado que estamos usando maven como herramienta de construcción de proyectos, si cambiamos de IDE, o simplemente si no disponemos de ninguno, siempre podremos ejecutar los comandos maven desde un terminal, y nuestra aplicación se compilará/ejecutará y funcionará exactamente igual que con el IDE: A continuación indicamos qué tenéis que configurar:

- Accedemos a ***Windows→Preferences→Maven→Installations***

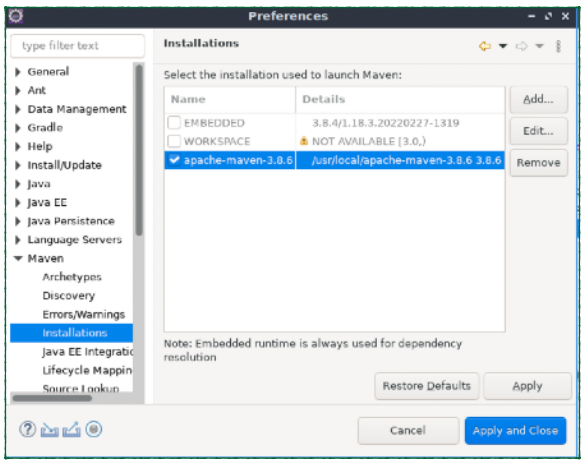
  Nos aseguraremos de que tenemos marcada la línea con el valor:

  /usr/local/apache-maven.3.9.3

  y solamente tiene que estar esa linea seleccionada.

  Eclipse incluye su propio maven. Pero queremos asegurarnos de usar la versión que nos interese, de forma que Eclipse NO altere en modo alguno nuestro proceso de construcción.

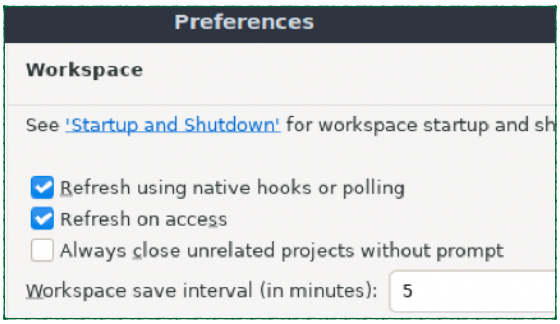
  Nota: En la máquina virtual hemos instalado maven en /usr/local/apache-maven.3.9.3.



- Accedemos a ***Windows→Preferences→General→Workspace***

  Marcaremos las opciones para "refrescar" el workspace de forma automática cuando se detecte algún cambio.

  Si queremos forzar un "refresco" manual, podemos hacerlo en cualquier momento desde el menú contextual del proyecto, o de forma alternativa, pulsando la **tecla F5**.

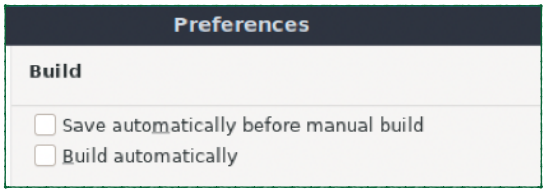


  Puedes dejar marcado con 5 minutos el intervalo de tiempo en el que Eclipse guarda de forma automática el contenido de nuestro *workspace*.

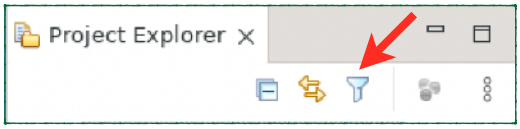
- Accedemos a ***Windows→Preferences→General→Workspace→Build***

  Nos aseguraremos de que las opciones mostradas en la figura estén desmarcadas.

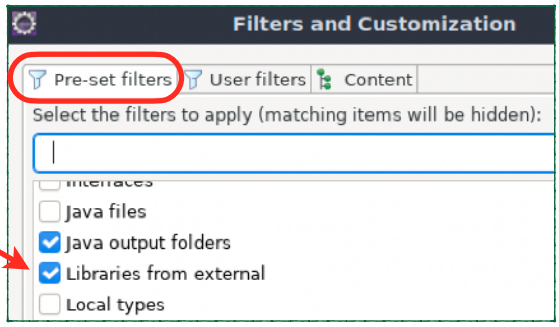
  No queremos que sea Eclipse quien decida cuándo debemos guardar y/o construir nuestro proyecto.



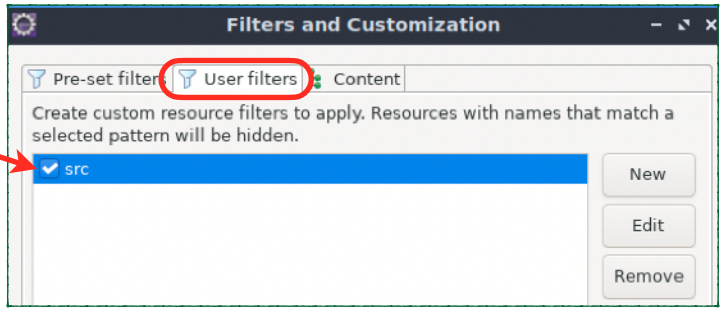
- Desde la vista ***Project Explorer***, pulsamos sobre el icono con forma de "embudo", para "filtrar" la información que se va a mostrar .



Nos aseguraremos de la opción "***Libraries from external***" esté marcada (en la pestaña "***Pre-set filters***").

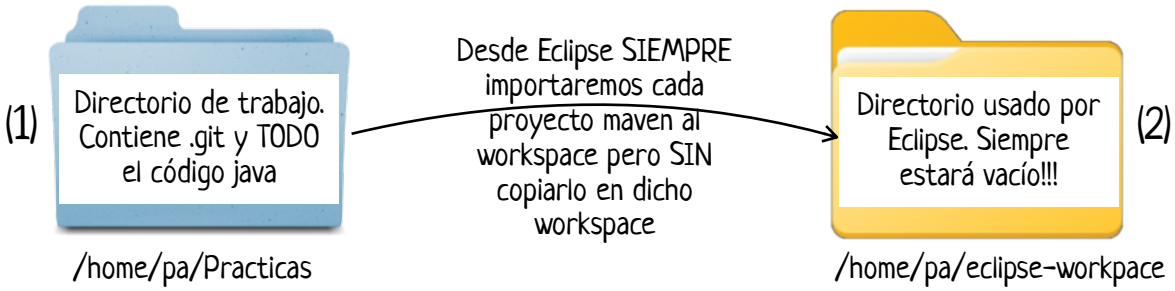


Desde la pestaña "***User Filters***" asegúrate de que tiene el filtro "**src**" y está marcado.



Forma de trabajar con Eclipse

Eclipse requiere trabajar siempre en un ***workspace*** (carpeta). Pero nosotros no queremos guardar en GitHub dicha carpeta. Por lo tanto lo que haremos será tener todo nuestro trabajo de prácticas (el directorio que contiene la carpeta oculta .git) físicamente en otro directorio DIFERENTE del *workspace* de Eclipse!!



TODO el código que trabajemos en prácticas estará SIEMPRE físicamente en (1). Desde Eclipse importaremos los proyectos Maven de (1) en el *workspace* de (2). Obviamente los cambios se guardarán en (1) porque los ficheros físicamente están ahí. Pero cuando trabajemos desde Eclipse, los "veremos" en la vista Project como si estuviesen en el *workspace*.

Desde (2) siempre podremos importar cualquier proyecto maven de nuestro directorio de trabajo (NUNCA lo copiaremos en el *workspace*). Dichos proyectos maven podremos cerrarlos, e incluso borrarlos, con la opción "Delete" del menú contextual del proyecto, pero tendremos que llevar cuidado de NO marcar NUNCA la casilla "*Delete project contents on disk*". Si no marcamos dicha casilla estamos"eliminamos" el proyecto de (2), pero los ficheros siguen estando en (1)

Importar un proyecto maven desde Eclipse.

Siempre importaremos los proyectos maven desde Eclipse mediante la opción :

File→Import...→Maven→Existing Maven Projects.

y en el campo de texto ***Root Directory*** indicaremos el directorio que contiene el fichero pom.xml del proyecto maven (podemos buscarlo desde el botón ***Browse***).

02 Preparamos nuestro directorio de trabajo

Para realizar esta práctica vamos a crear el directorio ***L02-Clases-Objetos***, dentro de nuestro directorio de trabajo (Practicas).

```
> cd Practicas
> mkdir L02-Clases-Objetos
> cd L02-Clases-Objetos
```

Nota: si en algún momento haces un "*clone*" del repositorio remoto sin especificar un nombre de directorio, el nombre de tu directorio de trabajo será el nombre de la nueva carpeta creada al hacer *clone*. El nombre de nuestro directorio de trabajo puede ser cualquiera. Sólo tienes que tener en cuenta que tu directorio de trabajo **siempre** será el directorio que contenga la carpeta oculta **.git.**

A continuación COPIAMOS en `L02-Clases-Objetos` el contenido de la carpeta `L02-Plantillas` que os hemos proporcionado. Suponiendo dicha carpeta está en el directorio */home/pa/Descargas,* podéis hacer la **copia** con:

```
> cp -R /home/pa/Descargas/L02-Plantillas/* .
```

Observa que hay un "espacio" y un "punto" al final del comando. Este "punto" está indicando que copiaremos todos los ficheros de */home/pa/Descargas/L02-Plantillas* en el directorio en el que estamos (*L02-Clases-Objetos*). La opción `-R` es para indicar que queremos copiar todo lo que haya en el directorio *L02-Plantillas*, incluyendo posibles subdirectorios).

También puedes **mover** los ficheros, en lugar de copiarlos con:

```
> mv /home/pa/Descargas/L02-Plantillas/* .
```

Nuestro directorio de TRABAJO tendrá la siguiente estructura:

```
/home/pa/Practicas
├── .git
├── .gitignore
├── L01-Git
│   └── saludo
└── L02-Clases-Objetos
    ├── cita-L02
    ├── dibujo-L02
    └── laboratory-L02
```

Cada uno de los subdirectorios de **`L02-Clases-Objetos`** es un proyecto Maven, y por lo tanto, obligatoriamente contiene un fichero pom.xml

Recuerda que en un proyecto Maven, los ficheros fuente ficheros.java) OBLIGATORIAMENTE estarán en el subdirectorio `src/main/java`

Cada uno de los ficheros `.java` de cada proyecto maven contiene el código de **una clase** java con el mismo nombre que el fichero. Por convención, los nombre de las **clases siempre** comenzarán por una letra **mayúscula**, para así diferenciarlas de los **objetos**, que escribiremos siempre con **minúsculas**. Así, por ejemplo, la clase "*Paciente*" estará implementada en el fichero "Paciente.java", y su código compilado se llamará "*Paciente.class*". Cuando creemos un objeto de la clase *Paciente*, usaremos cualquier nombre que comience por una letra minúscula, como por ejemplo "*paciente1*", "*pedro*", o "*enfermo_leve*".

Recuerda también que un buen programador siempre escribe código "legible". Eso implica que las variables **siempre** tienen nombres representativos de lo que contienen. Por ejemplo, si tenemos dos enteros que representan un dni y un número de teléfono, podemos llamar a las variables "dni" y "telefono", en lugar de "a", "b".

03 Vamos a dibujar: creación de objetos e invocación a métodos

Importa el proyecto maven "***dibujo-L02***" en tu *workspace* de Eclipse desde tu directorio de trabajo (/home/pa/Practicas/L02-Clases-Objetos/dibujo-L02). Al hacerlo, verás que, aunque en los ficheros que te proporcionamos no existe el directorio **target,** Eclipse lo crea automáticamente la primera vez que importas el proyecto**.**

Accede a las "*Run Configuration*" y comprueba si las rutas del directorio base y la ubicación de los ficheros *.launch* son correctas.

Ahora ejecuta "***dibujo-L02-clean***". Esta *Run Configuration* tiene asociado el comando **`mvn clean`**, que como ya hemos visto, lo que hace es BORRAR el directorio target.

Ahora vamos a compilar y ejecutar nuestra aplicación, usando la *Run Configuration* "***dibujo-L02-run***" (puedes consultar el comando maven asociado a la misma, en el campo "goals").

Verás que se imprime el mensaje por pantalla "Ejercicio pendiente de realizar".

Haciendo uso de la vista ***Outline***, descubre qué métodos puedes usar con los objetos de tipo *Circle*, *Square*, *Triangle* y *Person*.

Cada vez que seleccionas una clase, automáticamente la vista *Outline* muestra la descripción abstracta de dicha clase (hablaremos sobre el concepto de abstracción más adelante).

Los nombres de los métodos de cada clase son muy descriptivos, por lo que no te hará falta entender la implementación (por ejemplo, el método `makeVisible()`, lo que hace es mostrar la figura por pantalla).

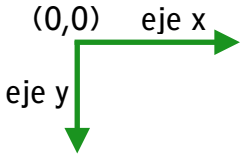
La clase ***Dibujar*** únicamente contiene un método ***main()***, y es el método que tendrás que implementar para realizar el dibujo mostrado en el lienzo de la **Figura 1**, usando círculos, triángulos, cuadrados y una persona.

En el código del método ***main()*** de la clase **Dibujar**, comenta la línea en la que se imprime por pantalla el mensaje que ves al ejecutar, y crea las instrucciones necesarias, para hacer lo siguiente:

- Crea un círculo, cuatro cuadrados, dos triángulos y una persona (los nombres de los objetos deben comenzar por **minúscula**. Utiliza nombres como puerta1, fachada1, sol, ..., de forma que "leyendo" el nombre, sea fácil averiguar qué representa cada objeto.
- Cambia el estado de los objetos que has creado (invocando a métodos de dichos objetos) de forma que se parezcan lo máximo posible al dibujo de la **Figura 1**:

Figura 1. Objetos y estado final

Nota: El origen de coordenadas del lienzo es la esquina superior izquierda del dibujo. Así, por ejemplo `moveHorizontal(-8)` mueve el objeto hacia la izquierda, con `moveVertical(10)` nos movemos hacia abajo



Deberás usar las ***Run Configurations*** para eliminar el *target* y compilar y ejecutar tu código.

Cuando termines el ejercicio, sube a GitHub el trabajo realizado hasta el momento con:

```
> git add .    //desde el directorio que contiene el repositorio git
> git commit -m"Terminado el ejercicio (proyecto dibujo) de L02"
> git push
```

04 Métodos: paso de parámetros y retorno de resultados

Desde Eclipse, importa el proyecto Maven ***laboratory-L02***. Revisa primero las ***Run Configurations*** para asegurarte de que son correctas. Proporcionamos la clase ***Enrollment***. Tendrás que comentar la única línea del su método ***main(),*** e implementar lo siguiente:

Nota: tendrás que invocar al método System.out.println(String cadena) para incluir las líneas en rojo mostradas en el resultado de la ejecución de tu programa. Dicho método recibe como parámetro un objeto de tipo String, que representa una cadena de caracteres. Un valor literal de este tipo de objetos se indica mediante una serie de caracteres entre comillas dobles.

Nota: Para expresar un valor literal de un único carácter, siempre usaremos comillas simples.
Por ejemplo:
char caracter = 'c' --> usamos comillas simples para valores de tipo char
String cadena = "c" --> usamos comillas dobles para valores de tipo String (contiene uno o más caracteres)

El método `println(String cadena)` imprime un salto de línea (carácter asociado a la tecla "return"), después de la cadena pasada por parámetro. El método `print(String cadena)` únicamente imprime los caracteres de la cadena pasada por parámetro:

```
System.out.println("imprime la cadena más un retorno de carro");
System.out.print("imprime la cadena sin retorno de carro");
```

Para imprimir por pantalla un retorno de carro (carácter asociado a la tecla "return"), puedes incluir el carácter '\n' en el String, tantas veces como necesites:

```
System.out.print("linea1 \n\nlinea2 \n\n");
```

- **Crea** varias instancias de la clase *Student,* usando los datos que mostramos a continuación e **imprime** dichos valores por pantalla (usa el método *print()* de la clase *Student*).

  Nombre: "Monica Geller", ID: "A00234", créditos: 24
  Nombre: "Joe Tribiani", ID: "C22044", créditos: 56
  Nombre: "Chandler Bing", ID: "A12003", créditos: 6
  Nombre: "Rachel Green", ID: "B66003", créditos: 12

  Recuerda: Las cadenas de caracteres en Java son de tipo String y SIEMPRE se indican entre comillas dobles.

- Crea dos objetos de tipo **LabClass**, con la siguiente información:
  - ➡ un objeto tendrá una capacidad para 2 estudiantes, como profesor a "Eli", con horario (atributo *timeAndDay* de la clase) "Miércoles, 15h", y se impartirá en el "Aulario 2"
  - ➡ la otra clase tendrá una capacidad para 1 estudiante, como profesor a "Jose Antonio", con horario "Miércoles, 17h", y se impartirá en el "Aulario 2.

- Usa el método ***enrollStudent*** definido en la clase *LabClass* para añadir los dos primeros estudiantes a la clase cuyo profesor es "Eli" y los otros dos a la clase de "Jose Antonio".

- Imprime por pantalla los listados de cada clase (objeto e tipo *LabClass*), invocando al método ***printList()***

Cuando ejecutes el programa debes ver por pantalla el resultado que mostramos a continuación:
Recuerda que:

- lo que se muestra en **negro** es el resultado de invocar a alguno de los métodos de los objetos que usas en tu método *main()*, y
- lo que se muestra en ***rojo*** es el resultado de usar sentencias *System.out.println(),* que deberás incluir en el código del método *main()*

```
Students'list:

  Monica Geller, ID: A00234, credits: 24
  Joe Tribiani, ID: C22044, credits: 56
  Chandler Bing, ID: A12003, credits: 6
  Rachel Green, ID: B66003, credits: 12

Enroll students (Eli):
   Monica Geller added
   Joe Tribiani added

Enroll students (Jose Antonio):
   Chandler Bing added
   The class is full, Rachel Green cannot enroll.

---------------
Lab class Martes, 15h
Instructor: Eli   Room: Aulario 2
Number of students: 2
  Monica Geller, ID: A00234, credits: 24
  Joe Tribiani, ID: C22044, credits: 56

---------------
Lab class Martes, 17h
Instructor: Jose Antonio   Room: Aulario 2
Number of students: 1
  Chandler Bing, ID: A12003, credits: 6
```

Recuerda subir a GitHub el trabajo realizado hasta el momento con:

```
> git add .        //desde el directorio que contiene el repositorio git
> git commit -m"Terminado el ejercicio laboratory de L02"
> git push
```

05 Estructura de clases y sentencias condicionales

Importa el proyecto Maven ***cita-L02*** y revisa las *Run Configurations* para asegurarte de que son correctas. La clase ***pa.Cita*** representa una cita, definida por:

- un día de la semana (valor entero entre 0 y 6. El 0 representa "lunes" y el 6 se usa para "domingo"
- una hora de inicio (valor entero entre 0 y 24)
- una hora de fin (valor entero entre 0 y 24)

Se definen, además, los siguientes comportamientos:

- `getInicio()` --> devuelve la hora de inicio de la cita
- `getFin()` --> devuelve la hora de fin de la cita
- `getDia()` --> devuelve el dia de la semana de la cita (valor entre 0 y 6)
- `setDia(int dia)` --> cambia el día de la cita al día pasado por parámetro
- `setFranjaHoraria(int inicio, int fin)` --> cambia la hora de inicio y fin de la cita
- `solapa_con(Cita c)` --> devuelve *true* si la cita solapa con la que se pasa por parámetro
- `imprimir()` --> muestra por pantalla los datos de la cita

La clase ***PruebaCitas*** sólo contiene un método ***main()***. Usaremos esta clase para probar el funcionamiento de nuestra implementación.

También se proporciona una *run-configuration:* ***cita-L02-run***, que realiza una construcción del proyecto formada por la secuencia de acciones: borrar el directorio target, compilar el código y ejecutarlo.

Vamos a realizar cambios en el código de la clase ***Cita***.

- Añade (e implementa) los métodos:
  - `private boolean hora_valida(int hora)`
  - `private boolean franja_es_valida(int inicio, int fin)`
  - `private boolean dia_valido(int dia)`

  Son metodos `privados`, lo que implica que NO pueden usarse desde "fuera" de la clase.

  Sólo pueden usarse desde el código de la propia clase, y los vamos a incluir para hacer más legible y mantenible el código.

  El método `hora_valida()` devuelve cierto si el valor de la hora pasada por parámetro está dentro del rango de valores indicados (0..24)

  El método `franja_es_valida()` devuelve cierto si la hora de inicio y fin que se pasan por parámetro tienen valores dentro de los rangos perimitidos (0..24), y la hora de inicio es menor que la hora de fin. Este método debe usar al método `hora_valida()`

  El método `dia_valido()` devuelve cierto si el valor del día pasado por parámetro está dentro del rango de valores indicado (0..6)
- Modifica los métodos `setFranjaHoraria()` y `setDia()` de forma que únicamente se actualicen los datos correspondientes si éstos son valores válidos. En el caso del día, debe ser un valor comprendido entre 0 y 6. En el caso de la franja horaria, los valores de inicio y fin deben ser valores comprendidos entre 0 y 24 y el valor de inicio debe ser inferior al valor de fin. Obviamente, debes hacer uso de los métodos privados que acabas de implementar (`franja_es_valida()` y `dia_valido()`).
- Añade (e implementa) el método:
  - `public boolean solapa_con(Cita c)`

  Dicho método devuelve cierto si hay solape entre las dos franjas horarias de las dos citas a comparar (la cita sobre la que aplicamos el método, y la cita que se pasa por parámetro. Y devuelve falso en caso contrario.

  Para que no haya solape, las dos citas no pueden tener lugar el mismo día ni pueden coincidir en parte de su franja horaria. Por ejemplo, una cita para el martes de 10 a 12h, solapa con otra cita para martes de 11 a 16h.
- Añade (e implementa) los métodos:
  - `private String mostrarDia(int dia)`
  - `public void imprimir()`

  El método `mostrarDia()` devuelve la cadena de caracteres (tipo String) asociada al día pasado por parámetro. Por ejemplo, para el día 2, el método anterior devuelve "Miercoles". Observa que vuelve a tratarse de un método `privado`. Sólo está permitido su uso desde el código de la clase. Debes usar una sentencia ***switch*** porque hace que nuestro código sea más legible.

  El método `imprimir()` debe imprimir por pantalla los datos de la cita: la hora de inicio y fin y el día de la semana.

  Debes usar el operador `+` para concatenar cadenas de caracteres y/o valores numéricos. Por ejemplo:

  ```
  int num = 10;
  ```

  La expresión: (`"El valor consultado es: "+num`) devuelve `"El valor consultado es: 10"`

Ahora modificaremos el código del método ***pa.PruebaCitas.main()*** en el que vamos a usar la clase y comprobar que el funcionamiento del código es el correcto.

- Necesitas crear 6 citas. Y la idea es comprobar si la cita1 solapa con alguna de las 5 restantes.

A continuación mostramos un ejemplo de ejecución en la que puedes ver exactamente los datos de cada una de las citas, y si solapan o no.

Igual que en el ejercicio anterior, lo que aparece en ***rojo*** es el resultado de usar sentencias *System.out.println(),* que deberás incluir en el código del método *main()*

```
Datos de la cita:                    <-- cita 1
    -hora inicio: 10
    -hora fin: 12
    -dia semana: Jueves

Datos de la cita:                    <-- cita 2
    -hora inicio: 11
    -hora fin: 13
    -dia semana: Jueves
Estas dos citas SOLAPAN

Datos de la cita:
    -hora inicio: 10
    -hora fin: 12
    -dia semana: Jueves

Datos de la cita:                    <-- cita 3
    -hora inicio: 7
    -hora fin: 12
    -dia semana: Jueves
Estas dos citas SOLAPAN

Datos de la cita:
    -hora inicio: 10
    -hora fin: 12
    -dia semana: Jueves

Datos de la cita:                    <-- cita 4
    -hora inicio: 10
    -hora fin: 11
    -dia semana: Jueves
Estas dos citas SOLAPAN

Datos de la cita:
    -hora inicio: 10
    -hora fin: 12
    -dia semana: Jueves

Datos de la cita:                    <-- cita 5
    -hora inicio: 16
    -hora fin: 18
    -dia semana: Jueves
Estas dos citas NO solapan

Datos de la cita:
    -hora inicio: 10
    -hora fin: 12
    -dia semana: Jueves

Datos de la cita:                    <-- cita 6
    -hora inicio: 10
    -hora fin: 11
    -dia semana: Miercoles
Estas dos citas NO solapan
```

Si te fijas en la ejecución, en el método `main()` tendrás mucho CÓDIGO REPETIDO!!

La duplicación de código es algo que siempre se debe intentar reducir al mínimo posible.

Podemos reducir dicha duplicación creando un nuevo método `privado` en nuestra clase ***PruebaCitas***.

El nuevo método debe ser `estático`, puesto que el método ***main()*** también lo es. Esto lo explicaremos más adelante.

Puedes definir el método como:

```
public static void comparar(Cita citaA, Cita citaB)
```

Deduce tú mismo cuál debe ser el código del nuevo método comparar().

Consejo: Implementa una primera versón del método main() sin usar el método comparar().

Y después implementas el método comparar y modificas el método main(). Pon entre comentarios todo el código duplicado y verás claramente que necesitas muchas menos líneas para implementar lo mismo.

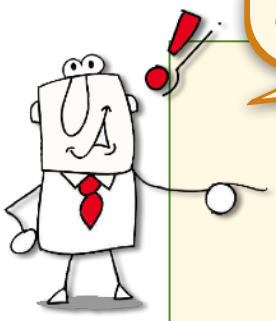
Fíjate en algo MUY importante: el programa debe comportarse de forma idéntica, independientemente de si usas el método comparar() o no.

Al proceso de alterar la estructura del código SIN cambiar la funcionalidad del mismo, para mejorar su mantenibilidad recibe el nombre de REFACTORIZACIÓN

Sube a GitHub el trabajo realizado hasta el momento:

```
> git add .   //desde el directorio que contiene el repositorio git
> git commit -m"Terminado el ejercicio  (proyecto citas) de L02"
> git push
```

RECUERDA!



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

- Las diferentes vistas de Eclipse y para qué se usan: Project Explorer, Outline, Editor, Console
- La estructura de ficheros de un proyecto Maven
- La forma de ejecutar un proyecto Maven desde Eclipse (Run configurations)
- ¿Qué contiene el directorio target y por qué no es necesario guardarlo en el repositorio git?
- ¿Qué contiene cada fichero .java y por qué no puede ejecutarse dicho fichero?
- ¿Cuales son las acciones que llevamos a cabo cuando construimos nuestro proyecto java y por qué necesitamos usar una herramienta de construcción de proyectos (como por ejemplo Maven)?
- La diferencia entre una clase y un objeto
- Las diferentes "partes" de una clase: atributos (propiedades) y métodos.
- Las diferencias entre de constructores y métodos accessor y mutator.
- Cómo se crean nuevos objetos en Java
- Un objeto siempre se almacena por referencia, mientras que los tipos primitivos se almacenan por valor
- Qué es el estado de un objeto
- Cómo se inicializa y cambia el estado de los objetos en java
- Cómo podemos interaccionar con los diferentes objetos
- Para qué sirve el método main()
- Cómo se pasan los parámetros en Java y para qué sirven
- Cuál es el flujo de control de una sentencia condicional (if y switch)
- Los atributos de una clase los definiremos como privados. La palabra reservada "private" indica que dicho elemento no puede ser accedido desde fuera de la clase (o del lugar en el que haya sido definido dicho elemento)
- Los métodos de una clase tienen acceso a todos los atributos de dicha clase
- Las variables declaradas dentro de un método sólo son accesibles dentro de dicho método.
- Los métodos de una clase pueden invocarse directamente desde el código de dicha clase
- Un métodos siempre pertenece a una clase. Por lo tanto, para refererirnos a dicho método siempre lo haremos indicando la clase a la que pertenece, p.ej. ClaseX.metodo1() indica que el método "metodo1" pertenece a la clase "ClaseX"
- Un método puede ser void, esto significa que no devuelve nada (y por lo tanto no termina con una sentencia return. Si un método no es void, necesariamente debe terminar con una sentencia return, y debe devolver un valor del tipo especificado
- Para facilitar la mantenibilidad del código NO vamos a permitir que uséis más de una sentencia RETURN en un método. Tampoco se permite usar la sentencia BREAK excepto en un switch
- Cuando estemos implementando deberemos evitar en lo posible DUPLICAR CÓDIGO, para ello necesitaremos modificar (REFACTORIZAR) dicho código