Practica 1:

Durante esta práctica aprenderemos a instalar y a usar un IDE (**Visual Studio Code** seguido del **IntelliJ).** Veremos cómo y para qué se usa el modo **debug** de un **IDE**, colocar **puntos de ruptura**, así como a configurar nuestro entorno para trabajar con un determinado lenguaje.

A lo largo de toda la actividad se ha de ir **completando** un documento a modo de **memoria** de **prácticas** que será el documento que se evaluará (puedes usar este mismo documento como referencia para ir contestando a las preguntas y apartados). En el documento se tendrá que contestar a las preguntas que se plantean, así como incluir capturas de pantalla y buenos comentarios para demostrar que se han entendido y seguido las actividades (de esta forma también trabajamos buenas prácticas a la hora de **documentar**).

El documento se subirá al Aula virtual dentro del plazo establecido y tiene que estar **en formato PDF.**

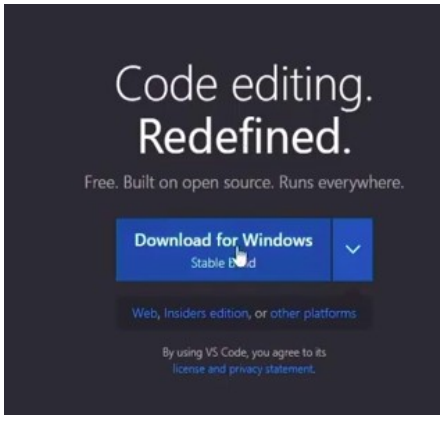
Para está práctica guiada haremos uso de una serie de **scripts** disponibles en el aula virtual.

Recomendación: Cuando vayamos a trabajar en un proyecto nuevo → crear una carpeta dedicada a este fin. Esto es recomendable siempre, y especialmente cuando trabajemos con lenguajes compilados ya que el proceso de compilación generará archivos adicionales que hay que tenerlos bien ubicados (además a medida que aumenta la complejidad de un proyecto hay mas dependencias de fichero de configuración, librerías, etc.).
En este caso pueden crear una carpeta con el nombre que crean mas conveniente y será ahí donde trabajaremos en esta práctica (donde tendremos que pegar nuestros scripts que vamos a ejecutar).

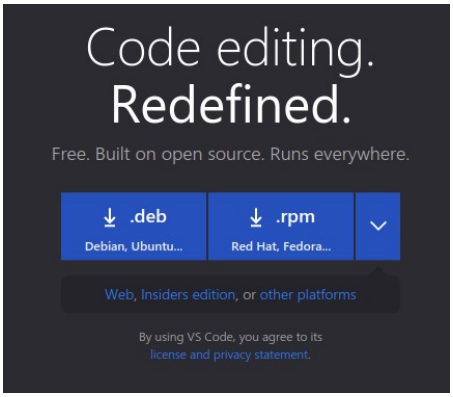
Actividad 1. Empezamos pues con la práctica guiada con **Visual Studio Code**:

Instalación del **IDE** (windows, linux).

Visitamos: https://code.visualstudio.com/
Nos va a detectar el S.O. que tenemos y solamente tendremos que descargarnos el fichero de instalación que nos interese.
Windows:



En sistemas Linux:



En sistemas Linux normalmente descargamos el archivo .deb y luego en la terminal el siguiente comando:

sudo dpkg -i nombre_del_archivo.deb

(Documenta el proceso con comentarios y las capturas que veas necesarias)

Actividad 2.

Para la realización de estás actividades prácticas hay disponibles una serie de **scripts** (los ficheros adjuntos a la actividad en el comprimido "DepurarParte1.zip") con los que trabajaremos.
Trata de **ejecutar** el primer script (**script_1**) y contesta a las preguntas.
¿En que lenguaje de programación está escrito? ¿Es un lenguaje compilado o interpretado?¿Has podido ejecutarlo directamente nada más instalar el Visual Studio Code?¿Que tienes que instalar para ejecutarlo?¿Has tenido que instalar algún plugin adicional?¿Cual?
(adjunta capturas de pantalla para documentar lo que has hecho en cada paso hasta conseguir ejecutar el código)

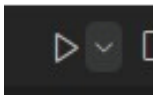
Actividad 3.

Busca **información** sobre el lenguaje en el que están escritos los scripts adjuntos y pon sus principales características, sus puntos fuertes, sus puntos débiles y principales campos de aplicación a día de hoy en el mundo de las tecnologías de la información.

Busca **plugins** que te puedan ayudar a trabajar de forma más eficiente e instala alguno (al menos uno) ¿Cual/es has instalado?¿Para qué sirven? Documenta todo el proceso y el plugin funcionando (es decir como estaba el entorno antes y después de aplicar el plugin, que se vea la funcionalidad del plugin).

Actividad 4. Depuración de código (modo Debug) y puntos de ruptura (breakpoints)

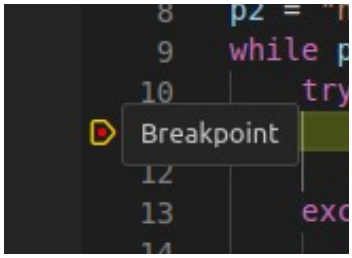
Acerca del modo debug y los puntos de ruptura:
Para empezar a depurar código tenemos que ejecutar nuestro programa en **modo debug**. En VS Code, en la esquina superior derecha click en el simbolo:



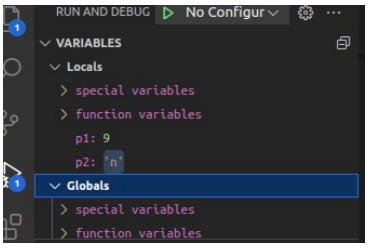
Y seleccionar la opción “Debug” en el desplegable.

El modo debug nos permite la opción de ejecutar nuestro código línea a línea **a partir de un punto de ruptura** que nosotros definamos.

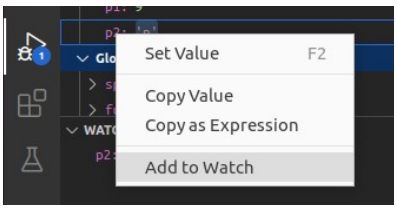
Punto de ruptura: Se marca haciendo click en la parte izquierda de nuestro código (a la izquierda del marcador del número de línea). La ejecución (en modo debug!) se parará cuando llegue a ese punto.



Una vez parada allí la ejecución, podemos chequear los valores de variables que tenemos declaradas (es como sacar una foto al estado del programa en ese momento). En la parte izquierda de la interfaz podemos ver los valores de la variables:



(También se pueden ver los valores de las variables situando el ratón encima de la variable, encima del código).
También si el valor de una variable/s es importante, podemos h**acerle un seguimiento** haciendo click en botón derecho → **add to watch**.



Una vez parada la ejecución en el punto de ruptura, podemos controlar la ejecución del programa **línea a línea** y ver como el programa se va ejecutando paso a paso. Para la ejecución de nuestro código en este modo tenemos **4 opciones** (se pueden alternar en la misma ejecución):

- **Step Over:** Ejecuta línea a linea “por encima”, es decir si en una línea se hace uso de una función, **no** entra dentro de la ejecución de la función. Cada click en este botón ejecuta una línea.

- **Step Into:** Ejecuta línea a línea y si en una línea se usa una función, **entra** en ésta y sigue ejecutando paso a paso líneas dentro de esta función. Cada click en este botón ejecuta una línea.

- **Step Out:** Si la ejecución en modo debug está ejecutando una línea dentro de una función con el Step Out salimos de esa función. Si no hay funciones de las que salir ejecutarlo es como ejecutar el botón de “Continue”.

- **Continue:** Hace una ejecución normal, como si no estuviera en modo debug (es decir no va línea a línea) y se para **en el siguiente punto de ruptura** que se haya marcado (si lo hubiera, si no lo hay se terminaría de ejecutar el programa normalmente).

Ahora que sabemos como va la depuración y los puntos de ruptura realiza las siguiente actividades:

4. 1. Pon un punto de ruptura en el **script_1.py** donde creas conveniente y ejecuta línea a línea viendo como se va ejecutando el código y los valores que van imprimiéndose por pantalla. Prueba con las **4 opciones** de ejecución (continue, step over, step into, step out). ¿Que diferencias hay entre los 4 modos de ejecución **para este caso**?

4.2. Ahora ejecuta el **script_2.py** y coloca nuevamente al menos un punto de ruptura. Prueba con las 4 opciones de ejecución (continue, step over, step into, step out). ¿Que diferencias hay entre los 4 modos de ejecución? (Saca capturas de pantalla dónde se vean las diferencias si es que las hay)

4.3. Ejecuta el **script_3.py** en modo normal. Saca captura de pantalla de lo que ocurre. ¿Que está ocurriendo? Pon un punto de ruptura y ejecuta paso a paso para dar con el error y solucionarlo ¿Que has hecho para solucionarlo? (Adjunta capturas de pantalla de la ejecución paso a paso hasta dar con el problema y solucionarlo).

4.4. Ejecuta el **script_4.py** ¿Se produce algún error?¿Por qué? Explica que está ocurriendo.

4.5. Ejecuta el **script_5.py** paso a paso fíjate en los valores que van tomando las variables (situándote encima con el ratón y también observando en la parte izquierda en la sección de "Variables"). Comenta lo que va haciendo el código y documenta lo que ves (que tipo de ejecuciones has hecho: step over, step into,etc).

4.6. Ejecuta el **script_6.py**. Este script como verás, se parece al anterior con la diferencia de que es el usuario el que pasa por teclado el valor de los operandos. Prueba a ejecutar el script con varios valores. ¿Que pasa si no le metes un número y le metes letras? (Documenta el proceso como has hecho anteriormente donde quede reflejado todas las pruebas que has hecho).

4.7. Ejecuta el **script_7.py** en modo normal para entender qué hace el programa.
En teoría el usuario introduce por teclado un número y el programa calcula si es un numero par o impar; después pregunta si quiere de salir de la ejecución o si quiere seguir. Si se introduce una 's' se sale de la ejecución; si se introduce una 'n' sigue pidiendo al usuario que introduzca otro número.

Prueba a introducir distintos valores por teclado para poner a prueba el programa (documenta indicando los valores que has probado con capturas de pantalla). ¿Funciona correctamente? ¿Hay algún comportamiento que no debería tener? ¿Si en lugar de poner 'y' o 'n' ponemos lo mismo pero con mayúsculas qué pasa?¿Por qué?
Prueba a poner puntos de ruptura y ejecutar el código línea a línea para dar con el **error** y **solucionarlo**.

Actividad 5: Configuración del IDE para otro lenguaje: Java

Ahora vamos a ejecutar algunos proyectos en Java. Para ello hay que descargarse y descomprimr el archivo adjunto "**DepurarJava.zip"** aquí encontraremos varias carpetas que iremos abriendo y viendo lo que tienen.

Abre la carpeta "**Depurar1**" y trata de ejecutar el fichero "**Principal.java**". ¿Puedes ejecutarlo? ¿Que has tenido que instalar para poder ejecutarlo? Documenta todo el proceso con los plugins que hayas necesitado instalar.
Pon puntos de ruptura y ejecuta el código paso a paso como hiciste en la Actividad 2.

Ejecuta en la terminal (dentro del propio IDE tienes una terminal para ejecutar los comandos), el siguiente comando:

Muestra lo que te devuelve y coméntalo. ¿Que hace cada uno de los elemento de Java que tienes instalados?¿Te cuadra con con lo que hemos visto en teoría cobre los lenguajes compilados?

Actividad 6: Compilando y generando el Bytecode de la solución.

Cuando distribuyes tu aplicación **Java**, generalmente proporcionas los archivos **.class** **(es decir el bytecode)** en lugar del código fuente. Esto permite que otros ejecuten la aplicación sin necesidad de tener acceso al código fuente.

Para generar dicho archivo, generalmente se hace un “Build” del código, que es basicamente el proceso de compilar y genera el fichero comentado. Par el caso del IDE Visual Studio Code, al ser un IDE de propósito bastante genérico (podemos usarlo para prácticamente cualquier lenguaje instalando los plugins necesarios como hemos visto) este proceso no es tan intuitivo como con otros IDEs especificos para Java. En este caso para hacer proceso tendremos que ejecutar el siguiente comando en la terminal:

javac Principal.java

Ejecuta el código y comenta que ha pasado ¿Se te ha generado el Bytecode?¿Puedes abrir el fichero con un editor de texto?

Actividad 6: IDE IntelliJ.

Descarga e instala el I**DE IntelliJ**. (https://www.jetbrains.com/es-es/idea/)

Una vez hecho esto abre la carpeta Depurar1 y repite la actividad 5 pero con este IDE (la parte de ejecutar el comando java –version no hace falta).
¿Has podido encontrar las mismas opciones de depuración que tenías para el visual Studio Code?

Nota: Si en el proceso de compilación con Visual Studio se generaron archivos/carpetas, bórralas para evitar conflictos (puede haber creado ademas del .class alguna carpeta oculta).

Actividad 7:

Abre desde IntelliJ la carpeta “Depurar2” y ejecuta el fichero .java.¿Puedes ejecutarlo?¿Te devuelve algún error?¿A que se debe (pista: comenta la línea que te da el problema y depura paso a paso observando los valores de las variables)?

Haz lo mismo con la carpeta “Depurar3”

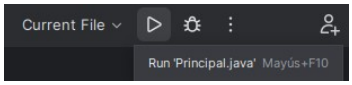
Actividad 8:

Abre desde IntelliJ la carpeta “Depurar4” y ejecuta el fichero .java. Comenta porqué se produce el error y modifica el código para que se muestre algo por pantalla.

Actividad 9:

Con el código corregido vamos ahora a hacer el “Build” de la solución. Para ello nos vamos a las opciones del IDE en la parte superior: “Build” → “Build Project” (o CTRL+F9)

Si ejecutamos la aplicación en modo normal:



También nos va a crear a crear el .class, pero si vamos a hacer una entrega es recomendable hacer el build.
¿Se te ha generado el Bytecode?¿Donde se te ha generado? Documenta todo el proceso.