Primera Práctica (P1)
Lenguaje de Programación Orientada a Objetos: Java

CURSO: 2025-2026
PROFESOR: José Angel Díaz García

Competencias específicas de la primera práctica

- Trabajar con entornos de desarrollo.
- Tomar contacto con el lenguaje OO Java.
- Comenzar a desarrollar un juego llamado Civitas siguiendo el paradigma OO.
- Interpretar los resultados obtenidos tras ejecutar un determinado código.

Objetivos específicos

- Familiarizarse con el entorno de desarrollo **IntelliJ IDEA**.
- Conocer las características básicas del lenguaje de programación Java.
- Aprender a implementar clases, enumerados y paquetes.
- Aprender el uso de constructores e inicializadores.
- Conocer el ámbito de las variables y métodos.
- Familiarizarse con los aspectos básicos de las colecciones de objetos y con algunas de las clases que los manejan en Java.
- Aprender a probar código.
- Aprender a manejar el depurador de IntelliJ.

1. Introducción

Durante el desarrollo de las prácticas de esta asignatura se implementará el juego **Civitas**, del cual ya dispones de las reglas. Se hará en **Java** y por etapas.

En esta práctica se tomará un primer contacto práctico con el paradigma de la orientación a objetos y con el lenguaje **Java**. Para ello se realizará la implementación de una serie de clases y tipos de datos enumerados simples que formarán parte del diseño completo del citado juego Civitas.

Durante el desarrollo de las prácticas hay que ser muy estricto con el nombre que se le da a cada elemento, siguiendo fielmente los guiones y diagramas que se entreguen, haciendo distinción igualmente entre mayúsculas y minúsculas. Ten en cuenta que como estudiante formas parte de un equipo de desarrollo junto con los profesores. Parte del código será proporcionado por los profesores, para que cada vez que se combine código de distintos desarrolladores no haya conflictos de nombres ni otros errores. Todos debemos ceñirnos a la documentación común con la que trabaja el equipo.

2. Herramientas

Para el desarrollo de estas prácticas se utilizará el entorno de desarrollo **IntelliJ IDEA** (versión Community es suficiente). Puedes descargar la última versión en: https://www.jetbrains.com/idea.

Podríamos decir que vamos a realizar una metodología de **desarrollo de software incremental**, en la que en cada práctica añadiremos nuevas funcionalidades al sistema, pasando siempre por el análisis de requisitos y diseño de la herramienta (que será proporcionado por los profesores mediante especificaciones del sistema, su funcionalidad y diagramas UML) y codificación, prueba y testeo, la cual será llevada a cabo por los estudiantes de manera que nos asemejaremos al proceso de desarrollo de software real que podríamos encontrar en una empresa.

3. Desarrollo de esta práctica

En esta práctica se realizará la implementación de una serie de clases y de tipos de datos enumerados. Estas tareas servirán como aproximación a la programación y diseño orientado a objetos.

Lo primero que debes hacer en **IntelliJ** es crear un **nuevo proyecto Java (Java Application)** con el nombre que desees. Todas las tareas de esta práctica se realizarán dentro de un paquete denominado **civitas**.

En Java, para nombrar los consultores se utilizará notación *lowerCamelCase* y los consultores seguirán la siguiente nomenclatura:

```
get<NombreAtributo>
```

De forma equivalente, los modificadores se llamarán:

```
set<NombreAtributo>.
```

3.1. Enumerados

Crea los siguientes tipos enumerados con visibilidad de paquete. En cada caso se proporciona el nombre del tipo y sus posibles valores además de una breve descripción de cada uno.

```
TipoCasilla: {CALLE, SORPRESA, JUEZ, IMPUESTO, DESCANSO}
```

Este enumerado representa todos los tipos de casillas del juego. Las casillas de descanso son aquellas en las que al llegar a ellas al jugador no le ocurre nada ni se produce ningún evento asociado a esa llegada. La salida, la cárcel y las zonas de aparcamiento se considerarán que forman parte de este conjunto de casillas.

```
TipoSorpresa: {IRCARCEL, IRCASILLA, PAGARCOBRAR, PORCASAHOTEL,
PORJUGADOR, SALIRCARCEL}
```

Representa todos los tipos de sorpresas que forman parte del juego Investiga cómo crear estos tipos de datos enumerados en Java. Para usarlos debes poner siempre el nombre del enumerado, seguido del valor, por
ejemplo:

```
TipoCasilla.CALLE
```

3.2. Clases

Tablero

Esta clase tiene la responsabilidad de representar el tablero de juego imponiendo las restricciones existentes sobre el mismo en las reglas de juego. Dispone de atributos y métodos específicos cuya finalidad es asegurar que el tablero construido sea correcto y que no se intente acceder a posiciones no válidas del mismo. Finalmente, también es responsabilidad de la clase Tablero el cálculo de la posición de destino después de una tirada con el dado.

Sus atributos de instancia (todos privados) son los siguientes:

- *numCasillaCarcel de tipo entero para representar el número de la casilla donde se encuentra la cárcel.*
- *casillas cuyo tipo será ArrayList<Casilla>. Este atributo es el contenedor de las casillas del del juego*
- *porSalida de tipo entero para representar el número de veces que se ha pasado por la salida en un turno*
- *tieneJuez de tipo Boolean para representar si el tablero dispone de la casilla de ese tipo.*

Como esta clase depende de la clase **Casilla**, de la que aún no dispones, vamos a crear una temporal para hacer las pruebas. Añade a la clase **Casilla** un atributo de instancia privado llamado nombre, y un constructor, con visibilidad de paquete, que acepte como parámetro una cadena de caracteres y la guarde en el atributo nombre. Crea también un consultor con visibilidad de paquete para el nombre. Posteriormente sustituirás esta clase Casilla temporal por la real.

La clase **Tablero** dispone de un único constructor con visibilidad de paquete que recibe como parámetro un entero que representa el índice de la casilla de la cárcel. El constructor realiza lo siguiente:

- *Guarda el valor del parámetro en el atributo numCasillaCarcel, siempre que ese valor sea mayor o igual que 1. En caso contrario se ignora el parámetro y numCasillaCarcel será igual a 1.*
- *Inicializa casillas a un ArrayList vacío (Array en Ruby) y añade una nueva casilla de nombre "Salida"*
- *Inicializa porSalida a cero*
- *Inicializa tieneJuez a false*

Añade los métodos de instancia privados siguientes:

- ***Boolean correcto ()***: devuelve true si el número de elementos en casillas es mayor que el índice de la casilla de la cárcel (*numCasillaCarcel*) (en ese caso se tiene asegurado que se ha añadido la cárcel en la posición indicada en el constructor) y que se dispone de una casilla tipo Juez. Si se cumplen todas las condiciones el tablero es correcto y puede usarse para jugar.

- ***Boolean correcto (int numCasilla):*** devuelve true si el método anterior también lo hace y además su parámetro es un índice válido para acceder a elementos de casillas. El resto de métodos de instancia de esta clase son los que se detallan a continuación. Todos tendrán visibilidad de paquete.

- ***int getCarcel ()***: es un consultor del atributo *numCasillaCarcel*

- int **getPorSalida ()**: si el valor de porSalida es mayor que 0, decrementa su valor en una unidad y devuelve el valor que tenía porSalida antes de ser decrementado. En caso contrario, simplemente devuelve el valor de porSalida

- ***void añadeCasilla (Casilla casilla)***: si el tamaño de casillas es igual a *numCasillaCarcel*, se añade primero a casillas una casilla denominada "Cárcel". En cualquier caso, después se añade a casillas la casilla pasada como parámetro, y se vuelve a comprobar si el tamaño de casillas es igual a *numCasillaCarcel*, en cuyo caso se añade a casillas una casilla denominada "Cárcel".

- ***void añadeJuez():*** si aún no se ha añadido una casilla juez, se añade y se actualiza el atributo tieneJuez. Se impide por tanto que se puedan añadir varias casillas de este tipo.

- ***Casilla getCasilla (int numCasilla)***: devuelve la casilla de la posición ***numCasilla*** si este índice es válido. Devuelve null en otro caso. Utiliza internamente el método Boolean correcto (int ***numCasilla***).

- ***int nuevaPosicion (int actual, int tirada)***: si el tablero no es correcto devuelve -1. En caso contrario se calcula la nueva posición en el tablero asumiendo que se parte de la posición actual y se avanza una tirada de unidades. Esta nueva posición se devuelve. Debes tener en cuenta que si se llega a la última posición del tablero, la siguiente casilla es la de salida (la primera) . Utiliza el operador módulo para realizar el cálculo de la posición de destino. Adicionalmente, el método incrementa el atributo porSalida si se ha producido un nuevo paso por la salida. Este hecho puede comprobarse fácilmente: si la nueva posición no es el resultado de sumar los parámetros actual y tirada, necesariamente se ha terminado una vuelta al tablero y pasado de nuevo por la salida.

- ***int calcularTirada (int origen, int destino)***: devuelve la tirada de dado que se habría tenido que obtener para ir desde la casilla número origen a la casilla número destino. En la mayor parte de los casos el cálculo necesario se limita a restar el origen del destino. Sin embargo, si de esta resta se obtiene una valor negativo, quiere decir que se ha producido un paso por la salida y al resultado anterior es necesario sumarle el número de casillas del tablero para obtener el valor correcto. Ejemplo: En un tablero de 20 casillas, si origen=18 y destino=3, el resultado de la tirada sería 3-18=-15. En ese caso, sumando el tamaño del tablero, se obtendría -15+20=5. Efectivamente, para ir de la casilla 18 a la número 3 se avanzan 5 casillas.

Diario

Esta clase se proporciona implementada y solo debes incorporarla a tu proyecto (ver sección 4). El diario se utilizará para llevar un registro de todos los eventos relevantes que se van produciendo en el juego. El diario solo almacena los eventos pendientes de ser consultados o leídos.

Cada evento se representará mediante una cadena de caracteres con la descripción del mismo. El diario sigue el patrón Singleton. Este patrón tiene como principal característica el hecho de que solo existe una instancia de una clase determinada. La propia clase es la que almacena y proporciona la referencia a esa única instancia.

```
//Java
public class Diario {
//Es un singleton. La propia clase almacena la referencia a la única
instancia
static final private Diario instance = new Diario();
//Constructor privado para evitar que se puedan crear más instancias
private Diario () {
eventos = new ArrayList<>();
}
//Método de clase para obtener la instancia
static public Diario getInstance() {
return instance;
}
….
}
Diario.getInstance() //Así se obtiene la única instancia del diario
```

Dado

Esta clase tiene la responsabilidad de encargarse de todas las decisiones del juego que están relacionadas con el azar, incluidas las relacionadas con el avance del jugador. Esta clase sigue además el patrón Singleton al igual que el Diario, y su visibilidad es de paquete.

Este dado tiene además la particularidad de poder funcionar de un modo que ayude a la detección y corrección de errores en el juego. Para ello, si se activa el modo debug, las tiradas para hacer avanzar al jugador serán siempre de una unidad. De esa forma se estaría forzando a que los jugadores avancen casilla a casilla por el tablero y se podrán hacer las pruebas de forma más sistemática.

Añade a la clase Dado los siguientes atributos de instancia privados:

- ***random*** de la clase Random. Se utilizará para la generación de números aleatorios
- ***ultimoResultado*** de tipo entero
- ***debug*** de tipo Boolean

Tendrá además dos atributos de clase privados
- ***instance*** de la clase Dado
- ***SalidaCarcel*** de tipo entero y cuyo valor será 5

El atributo instance referenciará a la única instancia de **Dado** que existirá en el juego. Será la propia clase Dado la que almacene esa referencia. Puedes consultar la clase Diario ya que esta también utiliza ese mecanismo.

Esta clase tendrá un único constructor privado y sin argumentos, que inicializará todos los atributos de instancia. Por defecto el modo debug estará desactivado. El atributo instance debe tener asociado un consultor también de clase denominado getInstance (en Java). La visibilidad de este método será de paquete.

Añade los siguientes métodos de instancia con visibilidad de paquete:

- **int *tirar* ()**: genera un número aleatorio entero entre 1 y 6 si el modo debug está desactivado. Siempre devuelve 1 en modo debug. El número generado se almacena en el atributo ultimoResultado.

- ***Boolean salgoDeLaCarcel ()***: se tira el dado y devuelve true si sale un valor que según las normas del juego permita que el jugador salga de la cárcel.

- ***int quienEmpieza (int n)***: este método se utiliza para decidir el jugador que empieza el juego. Devolverá un número entero entre 0 y n-1 (índice del jugador elegido)

- ***void setDebug (Boolean d)***: es el modificador del atributo debug. También deja constancia en el diario del modo en el que se ha puesto del dado (utilizando el método ocurreEvento de Diario)

- ***int getUltimoResultado()***: consultor del atributo ultimoResultado con visibilidad de paquete.

MazoSorpresas

Esta clase representa el mazo de cartas sorpresa. Además de almacenar las cartas, las instancias de esta clase velan por que el mazo se mantenga consistente a lo largo del

juego y para que se produzcan las operaciones de barajado cuando se han usado ya todas las cartas. Tiene además métodos especiales para la gestión de la carta de sorpresa de salida de la cárcel, que el jugador recibe y mantiene en su poder hasta que es usada. Esto implica por tanto que esa carta debe salir y volver al mazo. Al igual que en el caso anterior, para poder probar esta clase deberás crear una clase Sorpresa temporal, aunque en este caso, para la prueba de esta práctica, esta clase puede estar totalmente vacía. En la siguiente práctica se definirá la clase Sorpresa definitiva y los diferentes tipos de sorpresa que existen.

Los atributos de instancia (todos privados) de esta clase son los siguientes:

- ***sorpresas*** de tipo ArrayList<Sorpresa> para almacenar las cartas Sorpresa.
- ***barajada*** de tipo Boolean para indicar si ha sido barajado o no.
- ***usadas*** de tipo entero para contar el número de cartas del mazo que han sido ya usadas
- ***debug*** de tipo Boolean para activar/desactivar el modo depuración. Cuando está activo este atributo, el mazo no se baraja, permitiendo ir obteniendo las sorpresas siempre en el mismo orden en el que se añaden.
- ***cartasEspeciales*** de tipo ArrayList<Sorpresa>. Este atributo almacenará la carta sorpresa del tipo SALIRCARCEL mientras se considere retirada del mazo (y por tanto en posesión de un jugador). Se ha definido como una colección para poderle dar más funcionalidad al juego.
- ***ultimaSorpresa*** para guardar la última sorpresa que ha salido.

Esta clase dispondrá además el método privado:

-***void init ()***: Este método inicializa los atributos sorpresas y cartasEspeciales a un contenedor vacío, barajada a false y usadas a 0.

Esta clase dispone de dos constructores con visibilidad de paquete:

- Constructor con parámetro. El constructor de esta clase recibe como parámetro el valor para el atributo debug e inicializa este atributo. Además llama al método init, y si el modo debug está activado, se informa de este hecho a través del diario.

- Constructor sin parámetros. Este constructor simplemente llama al método init y fija el valor de debug a false.

El resto de métodos a implementar en esta clase son los siguientes, y tendrán todos visibilidad de paquete:

- ***void alMazo (Sorpresa s)***: si el mazo no ha sido barajado se añade la sorpresa que recibe como argumento al mazo. En caso contrario no se hace nada ya que no se permite añadir cartas a un mazo que ya está en uso.

- ***Sorpresa siguiente ()***: si el mazo no ha sido barajado o si el número de cartas usadas es igual al tamaño del mazo, se baraja el mazo (salvo que el modo debug esté activo), se fija el valor de usadas a cero y el de barajada a true. Posteriormente, se incrementa el valor de usadas, se quita la primera carta sorpresa de la colección de sorpresas, se añade al final de la misma, se guarda en el atributo ultimaSorpresa y se devuelve una referencia a esa carta sorpresa. Será la sorpresa que hay activa en ese momento para

el juego. Investiga los métodos que existen en Java para barajar colecciones de objetos.

- ***void inhabilitarCartaEspecial (Sorpresa sorpresa)***: la sorpresa que recibe como argumento es una carta especial. Si está en el mazo, se quita del mismo para que no se use, y se añade a cartasEspeciales. Si la operación se realiza se deja constancia en el diario.

- ***void habilitarCartaEspecial(Sorpresa sorpresa)***: si la sorpresa que recibe como argumento está en cartasEspeciales, se saca de ahí y se añade al final de la colección de sorpresas, para habilitarla para su uso. Si la operación se realiza se deja constancia en el diario. Es la operación complementaria a inhabilitarCartaEspecial.

3.3. Programa principal

Crea una clase denominada **TestP1** que tenga asociado un método de clase denominado main para hacer las funciones de programa principal.

En el método main crea el código para realizar las siguientes tareas:

1. Llama 100 veces al método *quienEmpieza*() de Dado considerando que hay 4 jugadores, y calcula cuantas veces se obtiene cada uno de los valores posibles. Comprueba si se cumplen a nivel práctico las probabilidades de cada valor.
2. Asegúrate de que funciona el modo debug del dado activando y desactivando ese modo, y realizando varias tiradas en cada modo.
3. Prueba al menos una vez los métodos *getUltimoResultado*() y *salgoDeLaCarcel*() de Dado.
4. Muestra al menos un valor de cada tipo enumerado.
5. Crea un objeto MazoSorpresas y haz las siguientes pruebas: añade dos sorpresas al mazo, obtén la siguiente sorpresa en juego, inhabilita y habilita la segunda carta añadida. Dado que MazoSorpresas
6. usa la clase Diario, aprovecha y prueba todos los métodos de Diario.
7. Crea un tablero, añádele varias casillas y comprueba con el depurador que efectivamente la estructura del mismo es la que esperabas. Intenta provocar las situaciones erróneas controladas en la clase Tablero (por ejemplo, que la posición de la cárcel sea mayor que el tamaño del tablero) y comprueba que la gestión de las mismas es la correcta. Finalmente, realiza distintas tiradas con el dado y asegúrate de que se calcula correctamente la posición de destino en el tablero. Llegado este punto te habrás dado cuenta que ante un error tipográfico habitual como escribir mal un atributo o el nombre de un método, IntelliJ en Java avisa mientras se escribe pudiendo subsanar el error al instante. De todos modos, en Java, el hecho de que un programa compile sin errores no significa que esté libre de errores.

Se aconseja realizar pequeños programas principales que permitan probar todo el código desarrollado. Cada alumno debe ser su principal crítico y diseñar las pruebas para intentar encontrar errores en su código. Una prueba de código se considera que ha tenido éxito si produce la detección de un error.

Para probar el código en Java es muy útil hacer uso del depurador y del método toString()

(investiga como funciona). En Java, usando el depurador, sigue paso a paso la creación de los objetos del primer punto y observa como se va modificando el valor de los atributos.

4. Incorporando código suministrado por los profesores a tu proyecto

Como parte del desarrollo de las prácticas de la asignatura, se te proporcionarán ya implementadas algunas clases para que solo tengáis que incorporarlas a vuestro proyecto. En esta práctica se os proporciona el enumerado EstadosJuego y la clase Diario. Sigue los siguientes pasos para incorporar el código que te proporcionamos para ser incorporado a tu proyecto:

- Averigua la ruta de tu proyecto: haz clic con el botón derecho sobre el proyecto y selecciona la opción “Properties”. Debes hacer clic sobre el nodo del que cuelgan todos los elementos del proyecto en el panel izquierdo.
- Si la ruta de tu proyecto es R, en R/src verás una carpeta por cada paquete de tu proyecto. Ahora mismo solo debes tener la carpeta civitas correspondiente al paquete con el mismo nombre. Copia los ficheros fuente suministrados en R/src/civitas/

No debes modificar ninguna de las clases suministradas por los profesores. Si hay un problema de concordancia repasa en primer lugar las clases que ya has creado para ver si siguen totalmente las especificaciones dadas. Si después de ese repaso persiste el problema, consulta con tu profesor.

5. Facilitando la compilación del proyecto durante el desarrollo

En Java, para evitar problemas de compilación provocados por no disponer del proyecto completo, puedes utilizar la siguiente sentencia como única línea del cuerpo de los métodos de los que solo tienes la cabecera:

```
throw new UnsupportedOperationException(“No implementado”);
```

Apéndice A: Pequeña introducción al modo debug en IntelliJ Idea

Es muy probable que durante la ejecución de un programa aparezcan errores. Algunos errores de programación pueden ser evidentes y fáciles de solucionar, pero es posible que haya otros que no sepáis de dónde vienen. Para rastrear estos últimos, os recomendamos

que utilicéis el depurador. Depurar un programa consiste en analizar el código en busca de errores de programación (bugs). Los entornos de desarrollo suelen proporcionar facilidades para realizar dicha tarea. En el caso de IntelliJ el procedimiento de depuración es muy sencillo:

Lo primero que debéis hacer es establecer un punto de control (breakpoint) en la sentencia del programa donde se desea que la ejecución se detenga para comenzar a depurar. Para ello, únicamente hay que hacer clic en el número de línea donde se encuentra dicha instrucción (line breakpoint). La línea, en el ejemplo la número 12 (figura 1), se resaltará en rosa.

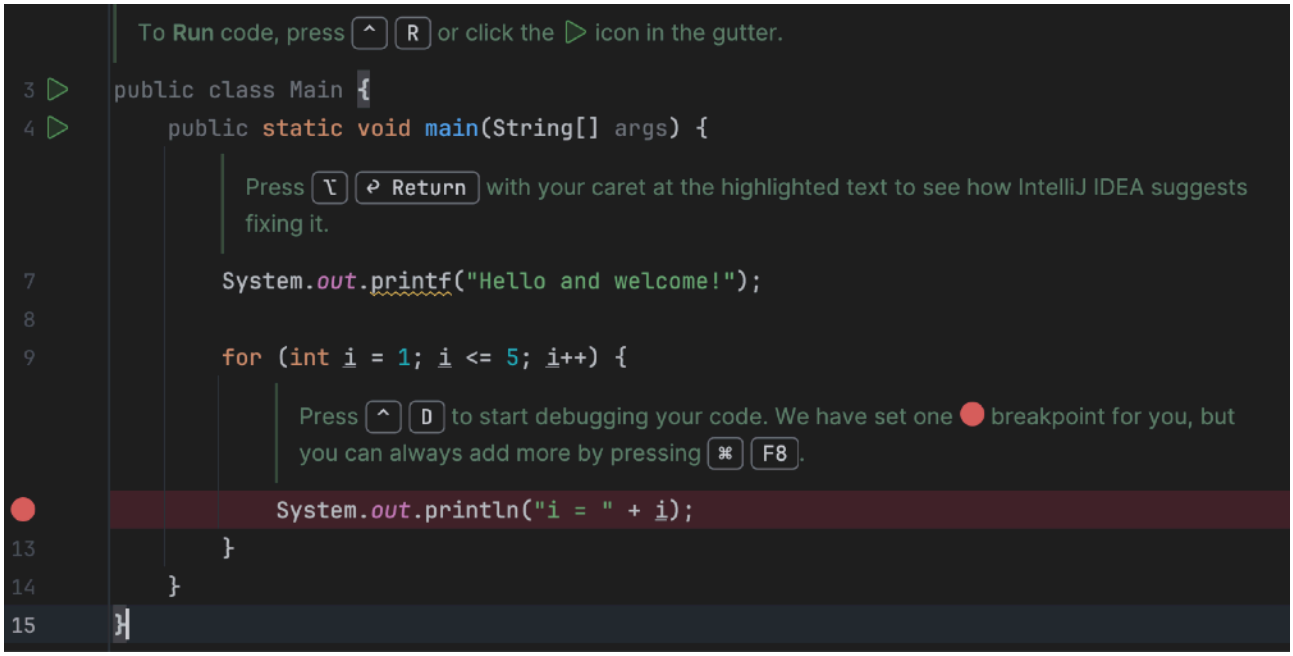

FIGURA 1: LINE BREAKPOINT.

Para comenzar la depuración deberemos hacer uso del botón de inicio de debug situado en la parte superior derecha del IDE, lo reconoceréis por el diseño en forma de bug/bicho.

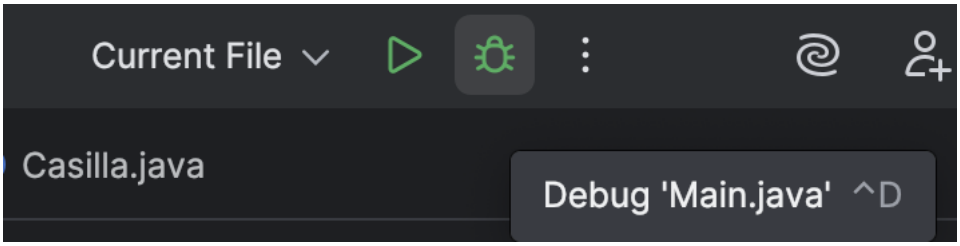

FIGURA 1: INICIO DE DEPURACIÓN.

Una vez que el flujo de control del programa llegue a un breakpoint, la ejecución se pausará para que podáis seguirla y la línea de código correspondiente se coloreará en azul además el breakpoint aparecerá con un tick superpuesto, indicando que se ha llegado y parado la ejecución en este punto correctamente.

Además, aparecerá una barra de herramientas de depuración (parte inferior izquierda en la figura 3) y un panel de depuración (abajo en la figura 3): threads & variables.

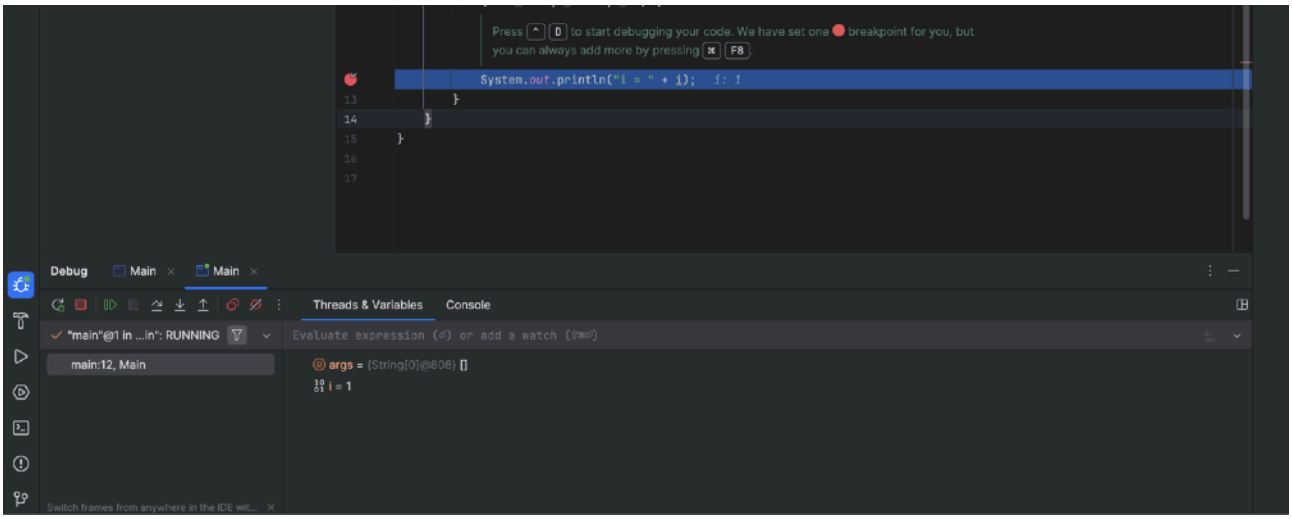

FIGURA 3: PANEL DE DEPURACIÓN.

Ahora se puede trabajar de tres modos diferentes: pulsando **F7 o (STEP INTO)** o **F8 (STEP OVER), o Shift+F8 (STEP OUT)**

- Si se pulsa F7, el control entrará en el método invocado en la instrucción actual (en verde). En nuestro ejemplo entraría dentro del bucle y podríamos ir paso a paso ejecutando el mismo.

- Si se pulsa F8, el control salta a la siguiente instrucción del programa. Es decir, si tuviéramos por ejemplo dos invocaciones a métodos consecutivas, esta opción ejecutará los métodos sin entrar paso a paso por ellos, por lo que la usaremos esta opción cuando estemos seguros que el bug no está en estos métodos que vamos a "saltar".

- Tendremos otra opción que es STEP OUT, que ejecutará el método en cuestión y saldrá del mismo.

En **IntelliJ,** podremos ver el valor de las variables mientras ejecutamos el programa en modo debug directamente sobre el código, ademas también podremos modificar el código en tiempo de debug. En este caso, podemos ver como las variables i y j tienen el valor de 5 y 1 respectivamente, bien abajo en el area de variables o sobre el mismo código (FIGURA 4)

Si lo deseáis, podéis situaros sobre una variable y pulsando New Watch en el menú contextual que se despliega, podéis definir un centinela (watch) que permitirá escribir una expresión y consultar su valor en el contexto actual con dicha variable.

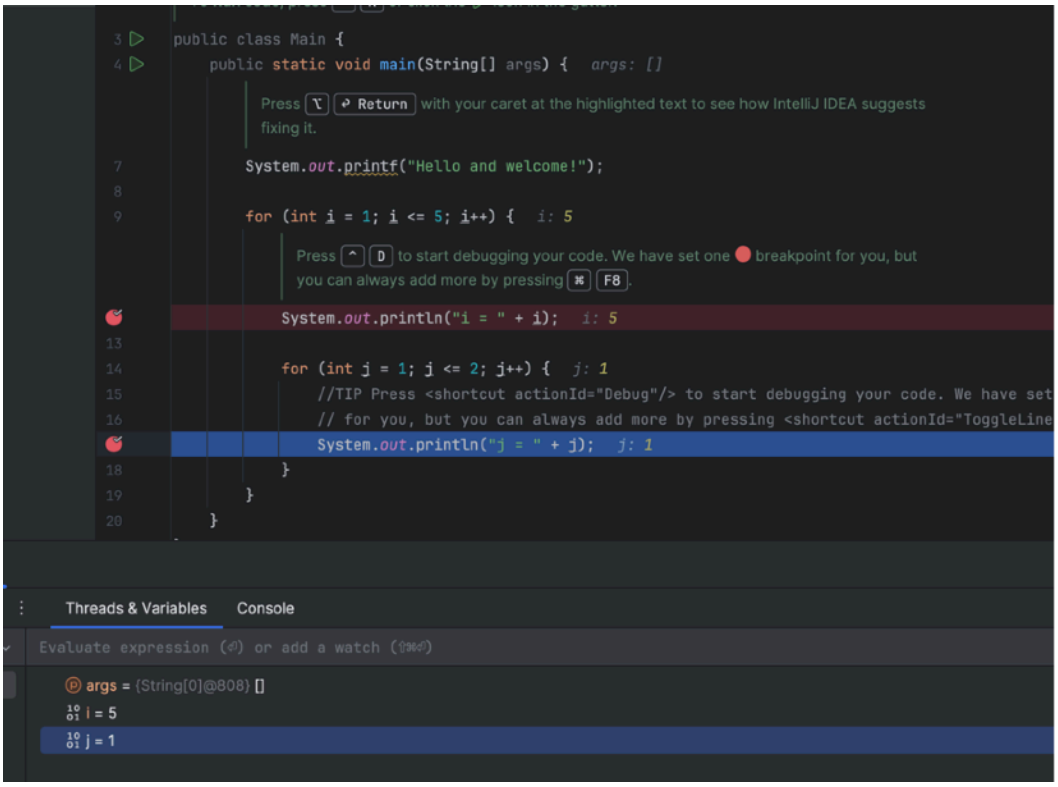

FIGURA 4: VER EL VALOR DE UNA VARIABLE