

# Primera Práctica (P1)

## Lenguajes de Programación Orientada a Objetos: Java y Ruby

### Competencias específicas de la primera práctica

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

### Objetivos específicos

- Familiarizarse con los entornos de desarrollo Netbeans y RubyMine.
- Conocer las características básicas de los lenguajes de programación Java y Ruby.
- Aprender a implementar clases, enumerados y paquetes.
- Aprender el uso de constructores.
- 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 Netbeans y de RubyMine.

## 1. Introducción general a las prácticas

Durante el desarrollo de las prácticas de esta asignatura se realizarán dos actividades.

Por un lado habrá **ejercicios** pequeños sobre conceptos concretos para implementar en Java y/o Ruby, lo cual te ayudará a afianzar los conocimientos adquiridos en las clases de teoría. En cada sesión de prácticas una parte de la clase estará destinada a la realización de algunos de estos ejercicios. Serás evaluado, en el laboratorio, sobre los conocimientos que has adquirido realizándolos.

Por otro lado se implementará el juego **Civitas**, del cual ya dispones de las reglas. Se hará en el lenguaje de programación orientado a objetos Java y por etapas.

Durante el desarrollo de Civitas hay que ser muy estricto con el nombre que se le da a cada elemento: ya sea clase, atributo o método; 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 puede ser proporcionado por los profesores. Por ello es fundamental que todos sigamos esta regla básica

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.

En esta práctica se tomará un primer contacto práctico con el paradigma de la orientación a objetos y los lenguajes de programación Java y Ruby. 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.

Aunque habrá ejercicios tanto en Java como en Ruby, el desarrollo del juego Civitas se hará exclusivamente en Java.

No obstante, en los ejercicios que se hagan en Ruby debes prestar atención a todo aquello que sea diferente con respecto a Java. Ambos lenguajes de programación son igual de importantes en el desarrollo de la asignatura. Debido a las características que los diferencian, el buen conocimiento de ambos lenguajes será muy positivo en tu formación sobre el paradigma de la orientación a objetos.

## 1.1. Herramientas

En los ordenadores del aula, puedes arrancar en Linux (Ubuntu 16.04), o en Windows 10 (con el código **w10pdoovi** )

Se cuenta con los siguientes IDEs según el lenguaje y el sistema operativo.

**Java** (tanto en Linux como en Windows)

Para el desarrollo en Java de estas prácticas se utilizará el entorno de desarrollo NetBeans. Puedes descargar la última versión estable en: <https://netbeans.org>. Asegúrate de que la versión descargada incluya al menos soporte para Java SE. En el **Apéndice A** de este documento tienes una guía para la depuración de programas Java con NetBeans.

**Ruby**

En **Windows** se encuentra disponible el entorno de desarrollo RubyMine. En el **Apéndice B** de este documento tienes una guía para la instalación y activación de este entorno de desarrollo. Adicionalmente, también se proporciona una guía para la depuración de programas Ruby con RubyMine.

En **Linux** no hay un IDE para Ruby. Se trabajará con un editor de textos e invocando el intérprete desde la consola de órdenes. El tamaño de los ejercicios propuestos permite trabajar sin IDE.

## 1.2. Exámenes

Los exámenes de prácticas se realizarán en los ordenadores del laboratorio usando el código de arranque **examenubu16**. El ordenador arrancará con Linux (Ubuntu 16.04) sin acceso a Internet, solo se podrá acceder a la plataforma Prado. **Se recomienda probar las prácticas en los ordenadores del aula y con dicha imagen en concreto, para evitar situaciones incómodas durante el desarrollo de los exámenes.**

## 2. Desarrollo de esta práctica

### 2.1. Objetivos

En esta práctica se trabajará en:

- La creación de clases y objetos instancia de dichas clases.
- La instanciación y prueba de clases desde un programa principal.
- La creación de constructores, consultores y modificadores básicos.
- La diferenciación entre estado y comportamiento.
- La diferenciación entre estado e identidad.
- La diferenciación entre atributos de instancia y de clase.
- La diferenciación entre métodos de instancia y de clase.
- La visibilidad pública, de paquete y privada.

### 2.2. Ejercicios

Lo primero que harás será crear un proyecto **Java (Java with Ant / Java Application)** en Netbeans para incluir en él todos los ejercicios que vayas haciendo. El proyecto lo puedes llamar *EjerciciosJava* y el paquete que incluirá todos los ejercicios se llamará *ejercicios*. Cada nuevo ejercicio se incluirá en dicho paquete. La clase principal también se llamará *EjerciciosJava*.

Para los ejercicios en Ruby crearás un proyecto en RubyMine (como se indica en el apéndice B) para incluir en él los ejercicios que vayas haciendo en este lenguaje.

#### Sesión 1

1. Añade al paquete *ejercicios* del proyecto *EjerciciosJava* una nueva clase denominada *Parcela*. Esa clase representará parcelas edificables de una ciudad y tendrá los siguientes atributos de instancia privados:
  - *nombre*, de tipo *String*
  - *precioCompra*, *precioEdificar* y *precioBaseAlquiler* de tipo *float*
  - *numCasas* y *numHoteles*, de tipo *int*.

Crea dicha clase con dichos atributos y añádele un constructor que reciba, en este orden, un nombre de tipo *string*, un precio de compra de tipo *float*, un precio de edificación de tipo *float* y un precio de alquiler base de tipo *float*.

Recuerda que el constructor debe inicializar todos los atributos, piensa cuál será el valor más adecuado para inicializar cada uno de los atributos que no ha recibido un parámetro en el constructor. Ten en cuenta que al crear una parcela solo existe el solar y no tiene edificaciones aún.

Añade un consultor para todos los atributos salvo el atributo *precioAlquilerBase*. En Java, para nombrar los consultores se utilizará notación lowerCamelCase. El nombre de un consultor se forma con el prefijo *get* seguido del nombre del atributo con su primera letra en mayúsculas. Por ejemplo, el consultor para el atributo *nombre* será *getNombre*.

Añade un consultor, denominado *getPrecioAlquilerCompleto* que calcule y devuelva el precio completo de alquilar dicha parcela teniendo en cuenta tanto el solar como sus edificaciones. Usa la fórmula indicada en las reglas del juego Civitas para calcular el alquiler de una casilla calle.

Añade dos métodos modificadores, *construirCasa()* y *construirHotel()* de tipo *boolean* que lo que hagan sean incrementar en 1 los atributos *numCasas* y *numHoteles* (sin ningún tipo de restricción ni comprobación) y que devuelvan siempre *true* (ya que siempre ha sido posible construir dicha edificación).

**Prueba la clase:** En la clase *EjerciciosJava* que se creó cuando creaste el proyecto *EjerciciosJava* verás que hay un método *main*. En dicho método instancia algunas parcelas, construye algunas casas y hoteles y calcula y muestra su precio de alquiler completo. Comprueba que las cifras calculadas son las que deberían ser. Si algo no funciona bien localiza el error y corrígelo.

2. Realiza el ejercicio anterior en Ruby.

En Ruby los consultores se llamarán exactamente igual que los atributos a los que van asociados. No obstante, en Ruby también pueden crearse los consultores usando *attr\_reader*. No añadas consultores para todo, solo los que se indiquen.

En Ruby se suele usar el estilo *snake\_case* para nombrar atributos, métodos y variables en general. Las palabras se escriben en minúscula y si un nombre está formado por varias palabras se escriben en minúscula separadas entre sí por el carácter '*\_*'. Por ejemplo, *puede\_gestionar*

En todos los archivos Ruby que se usen caracteres especiales, es decir, eñes, vocales con tilde, etc. debe aparecer, **como primera línea del fichero**, la siguiente:

```
# encoding:utf-8 (Algunas veces hay que dejar un espacio entre la almohadilla y encoding)
```

De no hacerlo, esos caracteres no serán reconocidos y el intérprete Ruby se detendrá dando un error.

Para ejecutar una aplicación en Ruby desde la consola solo tienes que situarte en la carpeta que contiene el archivo ruby con el método principal y ejecutar `ruby archivo.rb`

3. Añade a la clase *Parcela* del ejercicio anterior (solo en Java) un método con la siguiente cabecera

```
boolean igualdadIdentidad (Parcela otraParcela);
```

El método debe devolver *true* si el objeto receptor del mensaje es igual, a nivel de identidad, al objeto que ha recibido como parámetro.

Añade a la clase `Parcela` del ejercicio anterior un método con la siguiente cabecera

`boolean igualdadEstado (Parcela otraParcela);`

El método debe devolver `true` si el objeto receptor del mensaje es igual, en cuanto a su estado, al objeto que ha recibido como parámetro. Usa el método `equals` para comparar la igualdad en cuanto a estado de los objetos de tipo `String`.

Prueba estos nuevos métodos usando el mismo método `main` que has usado anteriormente. Crea 2 referencias a parcelas que creas que son iguales en identidad (y compruébalo), crea 2 referencias a parcelas que creas que son iguales en estado pero distintas en cuanto a su identidad (y compruébalo), y por último crea 2 referencias a parcelas que creas que son distintas en estado e identidad (y compruébalo).

## Sesión 2

1. Mejoraremos el ejercicio 1 de la sesión 1 evitando números mágicos, mejorando la legibilidad y el mantenimiento del código.

Añade a la clase `Parcela` (en Java) **variables de clase** de tipo `float` con los siguientes nombres y valores:

- `FACTORALQUILERCALLE = 1.0f;`
- `FACTORALQUILERCASA = 1.0f;`
- `FACTORALQUILERHOTEL = 4.0f;`

Modifica el método `getPrecioAlquilerCompleto` para que haga uso de dichas variables de clase.

Comprueba ejecutando el `main` que la nueva versión de la clase `Parcela` no presenta errores tras esos cambios.

2. Repite el ejercicio anterior en Ruby usando **variables de clase**.
3. Repite de nuevo el ejercicio en Ruby pero esta vez usando **variables de instancia de la clase**.

No borres ni sustituyas nada. Pon entre comentarios lo que fueses a borrar o modificar y crea el nuevo código. De esta forma se pueden ver las dos versiones a la vez.

En esta nueva versión habrás necesitado añadir algunos métodos, ¿verdad? ¿Por qué? Esos métodos que has añadido, ¿cómo son, de instancia o de clase?, ¿y cómo los has invocado desde el método de instancia `getPrecioAlquilerCompleto`?

## 2.3. Proyecto Civitas

No hay ninguna indicación sobre qué se hace en la primera o en la segunda sesión de la práctica. En general el proyecto Civitas se realizará empleando tanto las sesiones de prácticas como parte de tu tiempo de estudio en casa.

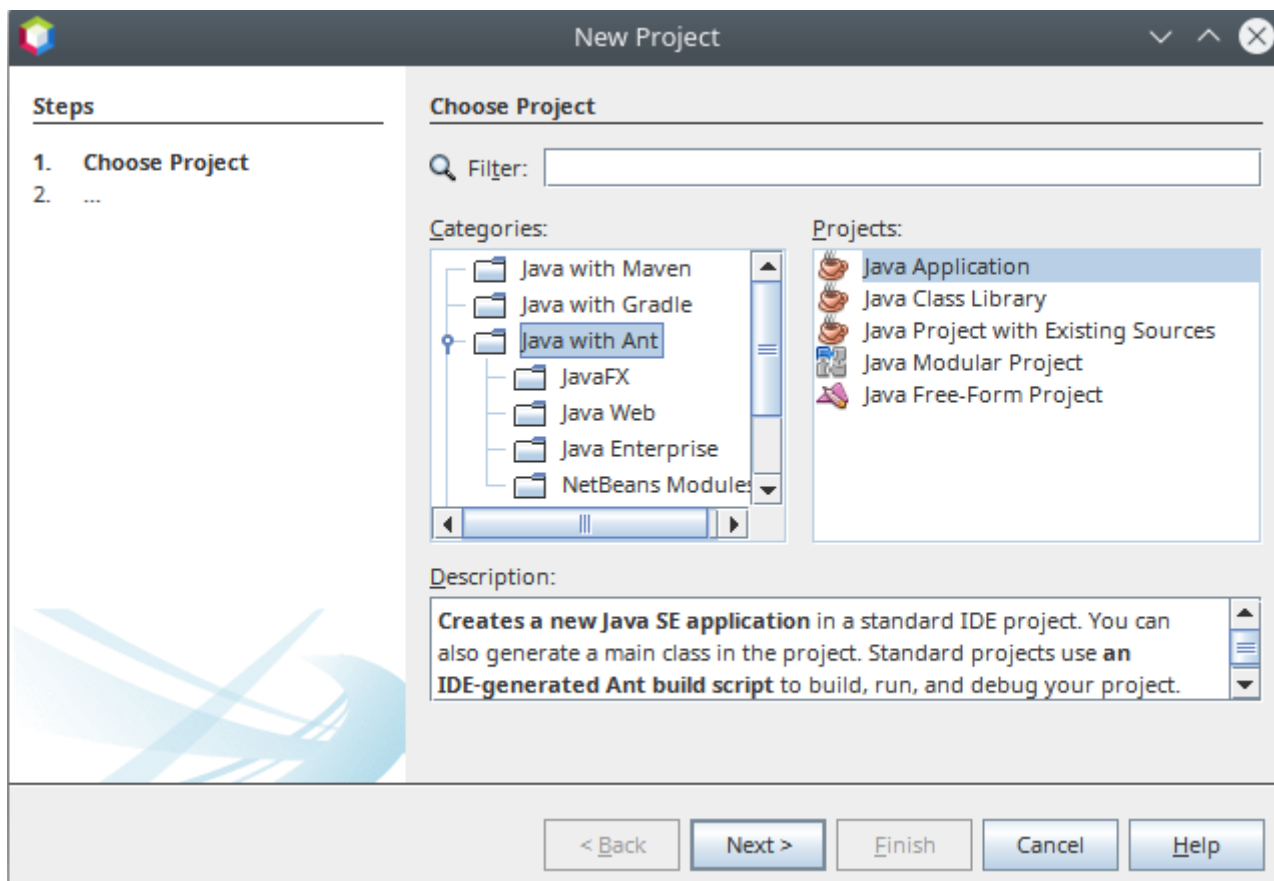
En esta primera práctica se realizará la implementación de una serie de clases y de tipos de datos enumerados.

En Java, para nombrar los consultores se utilizará notación lowerCamelCase y los consultores seguirán la siguiente nomenclatura `get<NombreAtributo>`. De forma equivalente, en Java los modificadores se llamarán `set<NombreAtributo>`.

A continuación se irá detallando cómo implementar cada uno de los enumerados y clases de esta práctica, así como el programa principal para probarlos. Debes seleccionar el paquete y elegir new, el lenguaje (Java siempre, en el proyecto civitas), y luego el tipo de elemento (clase, enumerado, etc. a crear).

## Creación del proyecto

El proyecto se creará usando Ant como gestor de proyectos (**Java with Ant / Java Application**) y se llamará Civitas.



## 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, 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 y las zonas de aparcamiento se considerarán que forman parte de este conjunto de casillas.

**TipoSorpresa: {PAGARCOBRAR, PORCASAHOTEL}**

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*

Incorpora también el enumerado *EstadosJuego* proporcionado por los profesores, tal y como se indica en la sección 4.

## Clases

### \* Casilla

Esta clase tiene la responsabilidad de representar las casillas del juego, recibir a los jugadores y aplicarles a estos lo que le corresponda según la situación del jugador y la casilla en el momento en que un jugador 'cae' en la casilla. Como hay casillas de diferentes tipos la clase tendrá atributos y métodos para representar y gestionar todos los tipos de casilla. Esta clase será una de las que más modificaciones recibirá a lo largo del desarrollo del proyecto Civitas.

Para la versión inicial de esta clase puedes usar la clase Parcela que has hecho en los ejercicios de la primera sesión. Copia y pega todo aquello que sea reutilizable.

Los atributos de instancia que se le van a añadir por ahora (todos privados) son los siguientes:

- **tipo**, cuyo tipo (valga la redundancia) será TipoCasilla. Este atributo es el que indicará si esta instancia de la clase es una casilla calle, sorpresa o descanso. Se consultará cuando haya que hacer un procedimiento diferenciado según el tipo de casilla concreta.
- **Nombre**, de tipo String. Contiene el nombre de la casilla, que en el caso de que la casilla sea calle, será el nombre de la calle.
- **precioCompra**, **precioEdificar**, **precioBaseAlquiler**, de tipo float. Contiene para las casillas de tipo calle los respectivos precios relacionados con dicha calle.
- **numCasas**, **numHoteles**, de tipo int. Contiene para las casillas de tipo calle la cantidad de casas y hoteles que tiene edificadas la calle.

Añádele un **constructor** con la siguiente cabecera:

Casilla (TipoCasilla unTipo, String unNombre,  
float unPrecioCompra, float unPrecioEdificar, float unPrecioAlquilerBase)

Este será el constructor para crear casillas de tipo calle. Puedes reutilizar gran parte del código del constructor que tenías para la clase *Parcela* de los ejercicios, añadiendo el nuevo parámetro de tipo *TipoCasilla* y la inicialización del atributo correspondiente.

Después de que hayas hecho los ejercicios de la sesión 2, añádele también las variables de clase de la clase *Parcela* de los ejercicios y los consultores y el método *getPrecioAlquilerCompleto()* que ya hiciste en la clase *Parcela* de los ejercicios.

Por último, añade un método de instancia con la siguiente cabecera

```
public String toString()
```

Dicho método conformará una cadena con el estado del objeto en un formato que sea cómodamente legible por un humano y se devolverá dicha cadena. El objetivo de estos métodos *toString()* (cada clase tendrá uno) es poder mostrar el estado de dicho objeto por consola mediante la instrucción

```
System.out.println (objeto.toString());
```

Por ejemplo, con relación a las casillas calle, esa cadena puede ser algo de este tipo:

“Calle Gran Vía. Precios: Compra: 1000, Edificar: 500, Alquiler base: 100, Casas: 0, Hoteles: 2”

Pero haz el diseño de esas cadenas de la forma que más te guste.

## \* 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:

- *casillas* cuyo tipo será *ArrayList<Casilla>*. Este atributo es el contenedor de las casillas del juego
- *porSalida* de tipo boolean para representar si el jugador que tiene el turno ha pasado por la salida o no en dicho turno.

La clase *Tablero* dispone de un único constructor con visibilidad de paquete que no recibe parámetros. El *constructor* realiza lo siguiente:

- Inicializa *casillas* a un *ArrayList* vacío y añade una nueva casilla de nombre “Salida”
- Inicializa *porSalida* a false

Añade los métodos de instancia privados siguientes:

- *boolean correcto (int numCasilla)*: devuelve true si 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.

- ***boolean computarPasoPorSalida ()***: devuelve el atributo *porSalida* y lo deja en false.
- ***void añadeCasilla (Casilla casilla)***: Añade a *casillas* la casilla pasada como parámetro.
- ***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)***: 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 establece el atributo *porSalida* a true 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.

### \* 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. Cada vez que añadas un evento al diario pon la descripción que estimes conveniente. Pon una descripción que te indique lo que ha ocurrido, esta descripción debe serte útil cuando estés jugando o depurando el código. Por ejemplo, “El jugador tal ha pasado por la salida”.

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;
}
// el resto de métodos de la clase Diario
}

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*. Puede inicializarse al declararlo (inicialización de campo). Se utilizará para la generación de números aleatorios. Hay que usar los métodos que proporciona la clase *Random* en vez de usar metodologías obsoletas como esas que realizan operaciones aritméticas a partir de *Math.random()*.
- *ultimoResultado* de tipo entero
- *debug* de tipo boolean

Tendrá además dos atributos **de clase** privados

- *instance* de la clase Dado

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, es decir, se establecerá a false.

El atributo *instance* debe tener asociado un consultor también de clase denominado *getInstance*. La visibilidad de este método será de paquete.

Añade los siguientes métodos de instancia con visibilidad de paquete:

- **int tirar ()**: genera y devuelve un número aleatorio entero entre 1 y 6 si el modo *debug* está desactivado y siempre será un 1 si está activado. El número a devolver se almacena en el atributo *ultimoResultado*.
- **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 (siendo *n* el número total de jugadores) que representa el índice del jugador agraciado en el sorteo. Al igual que en C++, el índice en un ArrayList en Java será 0 para el primer elemento y n-1 para el último.
- **void setDebug (boolean d)**: es el modificador del atributo *debug*. También deja constancia en el diario del modo en el que se ha puesto el 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.

Para evitar el error ocasionado por usar una clase no existente deberás crear una clase *Sorpresa* temporal, 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.

Esta clase dispondrá además del método privado:

- **void init ()**: Este método inicializa los atributos *sorpresas* a un contenedor vacío, *barajada* a false y *usadas* a 0. Este método será lo primero que se invoque desde los constructores, de manera que un constructor primero inicializa los atributos a su valor por defecto (el que se establece en el método *init* para después modificar los atributos que sean necesarios según lo que le corresponda a cada constructor.

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 una variable local de tipo *Sorpresa* y se devuelve una referencia a esa carta sorpresa. Será la sorpresa a aplicar al jugador que ha caído en una casilla sorpresa. Investiga los métodos que existen en Java para barajar colecciones de objetos.

## Programa principal

Cuando creaste el proyecto Civitas se creó automáticamente una clase con el mismo nombre con un método de clase denominado *main* para hacer las funciones de programa principal. Se usará este método *main* para realizar pruebas de las clases creadas en esta práctica.

Es igual a lo que ocurrió en los ejercicios con la clase Parcela, se creó una clase con un método *main* para probar poder probar el resto de las clases del proyecto.

En el método *main* crea el código para realizar las siguientes tareas (para ello, añade los consultores simples -getters- que sean necesarios en las distintas clases):

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 el método *getUltimoResultado()* de *Dado*.
4. Muestra al menos un valor de cada tipo enumerado.
5. Crea un objeto *Tablero* y haz las siguientes pruebas: añade algunas calles al tablero, obtén dichas casillas y muestra su estado por consola usando el método *toString* de la clase *Casilla*.
6. Crea algunos bucles sobre la totalidad de las calles para calcular y mostrar cuál es la calle más cara (en cuanto a su precio de compra), la más barata y el precio medio de las calles.
7. Usa la clase *Diario*, aprovecha y prueba todos los métodos de *Diario*.
8. 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, Netbeans 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**. Sintácticamente puede estar bien pero semánticamente puede tener errores. Por ejemplo, si estás en la casilla 2 y tras tirar el dado sale un 1, deberías obtener que estás en la casilla 3. Si obtienes que estás en la casilla 7 claramente hay un error que habrá que localizar y corregir. Durante todo el desarrollo que hagas de Civitas, las pruebas deben encaminarse a encontrar este tipo de errores y corregirlos.

Se aconseja implementar diversos métodos de clase en la clase Civitas y que sean invocados desde el método *main* de dicha clase para probar todo el código desarrollado. Cada estudiante 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 (explicado en el Apéndice A) 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 cómo se va modificando el valor de los atributos.

### 3. 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 a tu proyecto:

- Averigua la ruta de tu proyecto: haz clic con el botón derecho sobre el proyecto en NetBeans y selecciona la opción “*Properties*”. Debes hacer clic sobre el nodo del que cuelgan todos los elementos del proyecto en el panel izquierdo.
- En Java: 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.

### 4. 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:

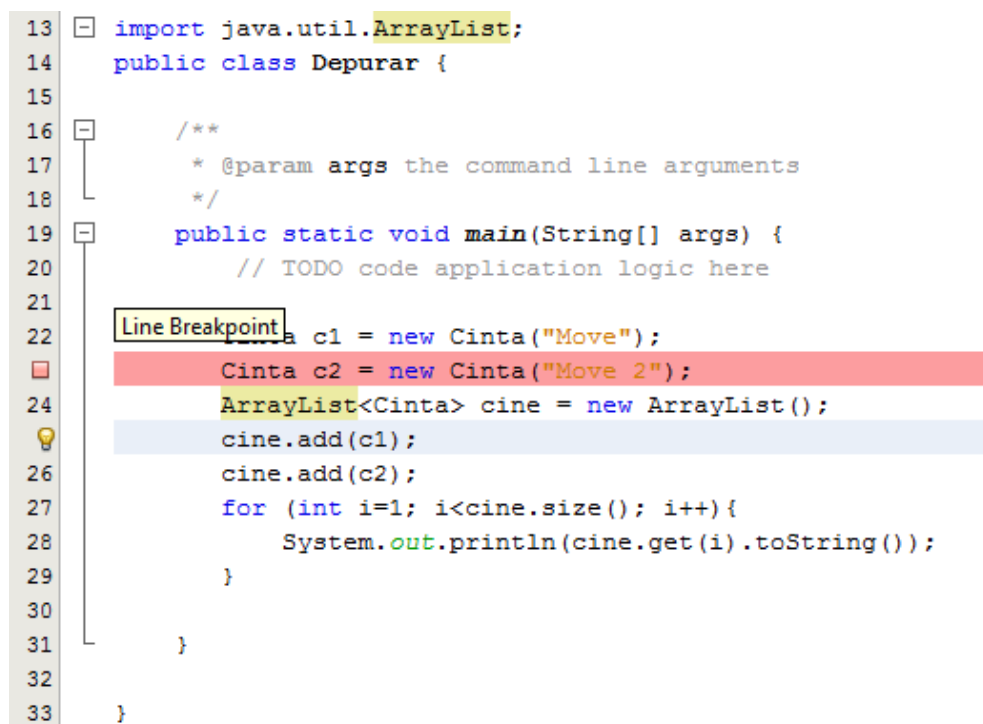
```
throw new UnsupportedOperationException("No implementado");
```

## Apéndice A: Depuración del código en Netbeans

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 **Netbeans** 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 23 (figura 1), se resaltará en rosa.



```
13 import java.util.ArrayList;
14 public class Depurar {
15
16     /**
17      * @param args the command line arguments
18      */
19     public static void main(String[] args) {
20         // TODO code application logic here
21
22         Cinta c1 = new Cinta("Move");
23         Cinta c2 = new Cinta("Move 2");
24         ArrayList<Cinta> cine = new ArrayList();
25         cine.add(c1);
26         cine.add(c2);
27         for (int i=1; i<cine.size(); i++){
28             System.out.println(cine.get(i).toString());
29         }
30     }
31 }
32
33 }
```

The screenshot shows the NetBeans IDE with a Java file named 'Depurar.java'. A line breakpoint is set on line 23, which is highlighted in pink. The code defines a 'Cinta' class and a 'Depurar' class with a 'main' method. The 'main' method creates two 'Cinta' objects, adds them to an 'ArrayList' named 'cine', and prints the contents of the list.

Figura 1: Line Breakpoint.

Después, pulsar **Control+F5** o la correspondiente opción del menú **Debug** para comenzar la depuración (figura 2).

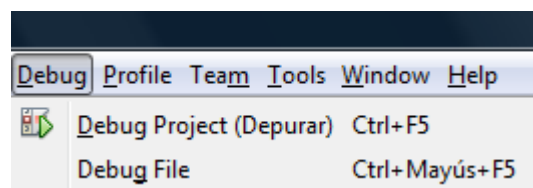


Figura 2: Menú Debug.

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 verde. Además, aparecerá una barra de herramientas de depuración (arriba en la figura 3, después del *play*) y dos paneles de depuración (abajo en la figura 3): variables y *breakpoints*.

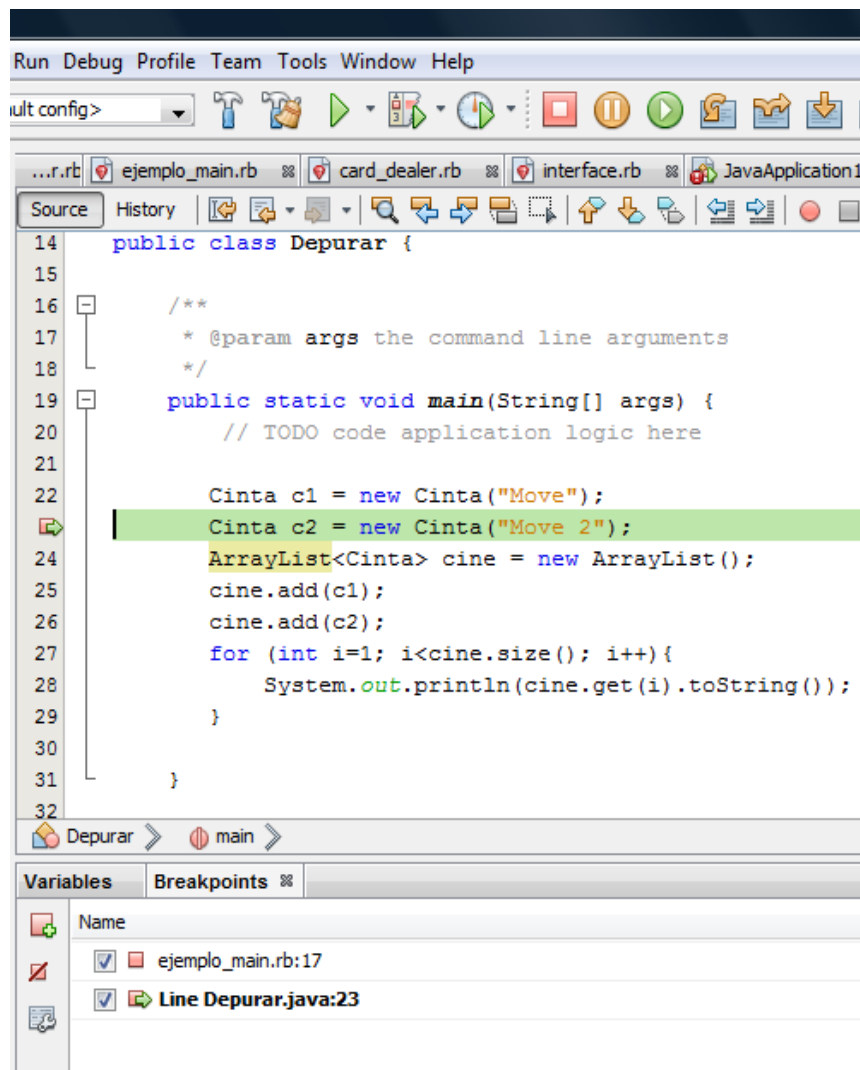


Figura 3: Herramientas de depuración.

Ahora se puede trabajar de dos modos diferentes: pulsando **F7** o **F8**.

- Si se pulsa F7, el control entrará en el método invocado en la instrucción actual (en verde). En nuestro ejemplo se ejecutaría el constructor de la clase Cinta, parándose en la primera línea de dicho método.
- Si se pulsa F8, el control salta a la siguiente instrucción del programa. En nuestro ejemplo se ejecutaría el constructor (línea 23 del código) y se pararía en la línea 24. Usaréis, por tanto, esta opción cuando estéis seguros de que el *bug* no está ni se deriva del método invocado.

Si se han definido varios *breakpoints* en el fichero o proyecto, podéis usar la opción **F5** que continuará ejecutando instrucciones hasta el siguiente *breakpoint*, donde se pausará de nuevo la ejecución. Si se pulsa F5 continuará la ejecución hasta el siguiente punto de control. No tiene sentido en nuestro ejemplo, pues solo hemos definido un *breakpoint*.

En cualquier momento podéis situar el cursor sobre una variable del programa y, si la variable está activa (en ámbito), obtendréis su valor. En el ejemplo (figura 4), la variable *i* tiene valor 1 en ese preciso instante de la ejecución.

```

    for (int i=1; i<cine.size(); i++) {
        System.out.println(cine.get(i).toString());
    }

```

Figura 4: Ver el valor de una variable.

Adicionalmente, en el panel informativo de Variables (ver figura 3, abajo), podéis consultar el valor de cualquier variable activa en el contexto de ejecución actual. En el ejemplo (figura 5) es posible examinar que **i** tiene valor 1, y también el tipo de las variables, por ejemplo que **c1** es un objeto de la clase **Cinta**.

Variables	Breakpoints	
Name	Type	Value
<input checked="" type="checkbox"/> m		>"m" is not a known variable in the current context.<
<input checked="" type="checkbox"/> x		>"x" is not a known variable in the current context.<
<Enter new watch>		
+ Static		
+ args	String[]	#74(length=0)
+ c1	Cinta	#72
+ c2	Cinta	#75
+ cine	ArrayList<Cinta>	"size = 2"
+ i	int	1

Figura 5: Panel de Variables.

Si pulsamos el símbolo + que aparece junto a cada variable, aparecen más detalles sobre ésta (figura 6). Por ejemplo, para el **ArrayList cine** podemos conocer su tamaño (2) y cada uno de sus elementos (dos objetos de la clase **Cinta**). Y para un objeto de la clase **Cinta** podemos inspeccionar sus atributos (en este caso, nombre).

[-] cine	ArrayList<Cinta>	"size = 2"
[-] [0]	Cinta	#72
[-] nombre	String	"Move"
[-] [1]	Cinta	#75
[-] nombre	String	"Move 2"

Figura 6: Detalles sobre la variable **cine**.

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. Por ejemplo, en las figuras 7 y 8, una operación aritmética definida sobre el valor de **i**, o la llamada a un método del objeto **cine**. Los *watches* aparecen en el panel de variables, tal y como se observa en las figuras.



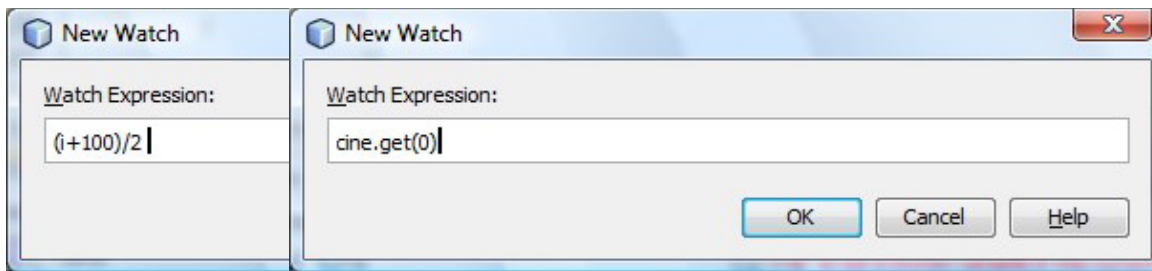


Figura 7: New Watch, expresión

Variables		Breakpoints	
	Name	Type	Value
<input checked="" type="checkbox"/>	x		>"x" is not a known variable in the current context.<
<input checked="" type="checkbox"/>	(i+100)/2	int	50
<input checked="" type="checkbox"/>	cine.get(0)	Cinta	#72

Figura 8: New Watch. consulta

Finalmente, para detener la depuración y continuar con la ejecución normal usaremos Mayúscula+F5 o el botón correspondiente.

## Apéndice B: Instalación y activación de RubyMine. Depuración de código Ruby.

### Instalación

RubyMine es un IDE profesional desarrollado por la empresa JetBrains (<https://www.jetbrains.com>).

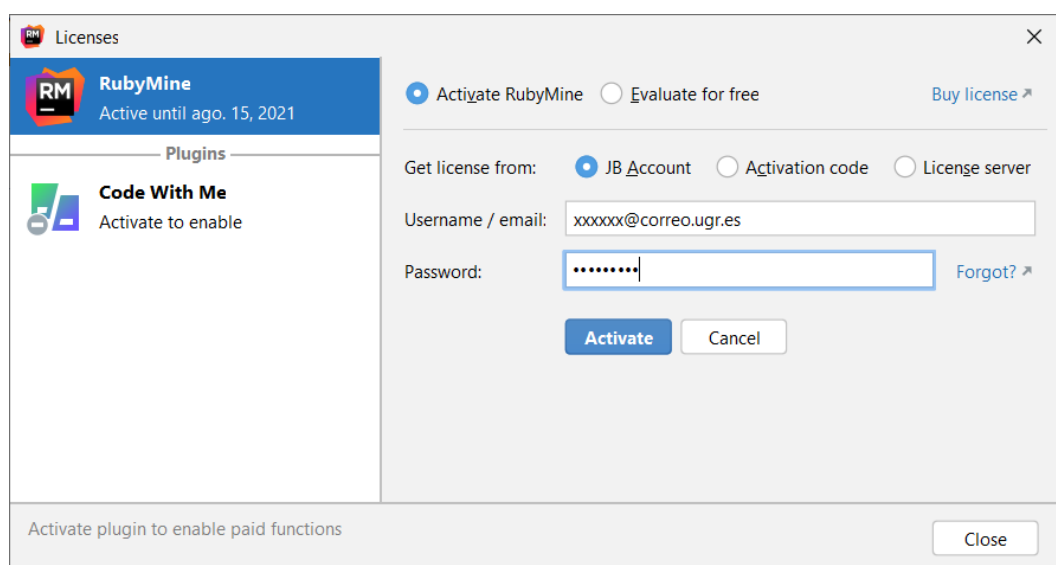
RubyMine no incluye el intérprete de Ruby, por lo que el primer paso en la instalación consiste en descargar e instalar este intérprete para la plataforma deseada (Windows, Linux o Mac). Toda la información para la instalación de Ruby la tienes aquí: <https://www.ruby-lang.org/en/documentation/installation>

Una vez instalado el intérprete de Ruby, el siguiente paso es instalar RubyMine, descargándolo desde aquí: <https://www.jetbrains.com/ruby/download> y siguiendo los pasos indicados aquí <https://www.jetbrains.com/help/ruby/2021.1/installation-guide.html#standalone>, que serán distintos en función de la plataforma seleccionada (Windows, Linux o Mac).

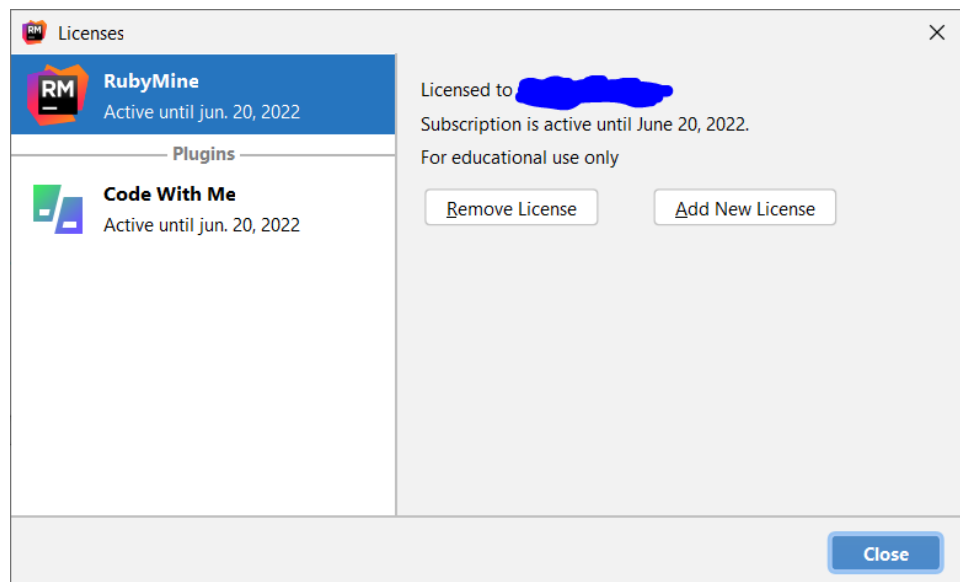
### Activación

La primera vez que se ejecuta RubyMine se nos solicitará activarlo o evaluarlo de manera gratuita durante 30 días. JetBrains proporciona una licencia educativa para estudiantes que permite activar RubyMine de manera gratuita. Para obtener esta licencia tienes que registrarte, **utilizando tu correo institucional @correo.ugr.es**, aquí: <https://www.jetbrains.com/shop/eform/students>

Una vez recibida la confirmación de tu licencia educativa, podrás introducir los datos de tu cuenta de JetBrains en RubyMine y activarlo tal y como se muestra a continuación:

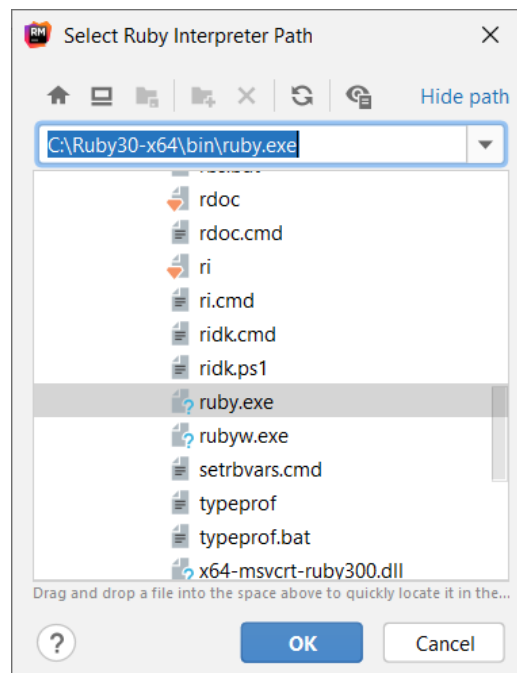


Debiendo obtener un mensaje de confirmación similar al siguiente:



## Creación del primer proyecto Ruby y Depuración de código

Al pulsar en el botón “New Project”, se solicitará dónde queremos almacenar el proyecto y el SDK de Ruby a utilizar. En el primer proyecto a crear no aparecerá ningún SDK seleccionado, aunque lo hayamos instalado correctamente en el primer paso de la instalación. Por tanto debemos desplegar la lista de SDKs disponibles y pulsar en la opción “Add Ruby SDK...”. Automáticamente se buscará un intérprete de Ruby en nuestro PC y aparecerá seleccionado para añadirlo, apareciendo algo similar a lo que se muestra en la siguiente imagen:



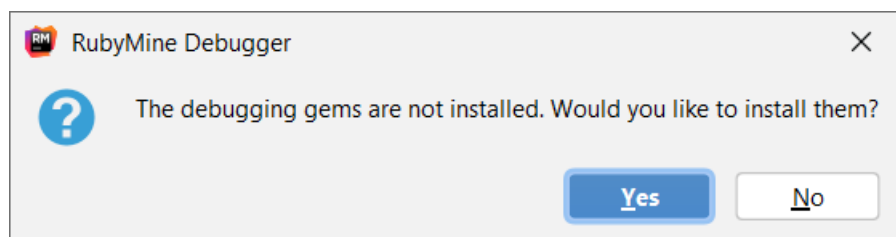
Una vez creado el proyecto, le añadimos clases utilizando “File → New → Ruby File/Class”.

Para depurar un programa se sigue un proceso completamente análogo al descrito previamente para NetBeans. En primer lugar se establece un *breakpoint* pinchando con el ratón a la derecha del número de la línea donde queremos que se pare la ejecución para ir posteriormente avanzando paso a paso. Aparecerá el punto rojo del *breakpoint* y la línea se resaltará en color rosa:


```
5 module Hotel
6
7 class Main
8
9   def self.visualiza(hab)
10    cadena = "Número: " + hab.getNumero().to_s + ", camas matrimonio: " + hab.getCamasM().to_s +
11            ", camas individuales: " + hab.getCamasI().to_s
12    if (hab.instance_of?(HabitacionFamiliaNumerosa))
13      cadena += ", literas: " + hab.getLiteras().to_s
14    end
15    puts cadena
16  end
```

A continuación, el depurador se inicia pulsando Mayús+F9 o sobre el icono del bicho (*bug*) o seleccionando “Run → Debug”.

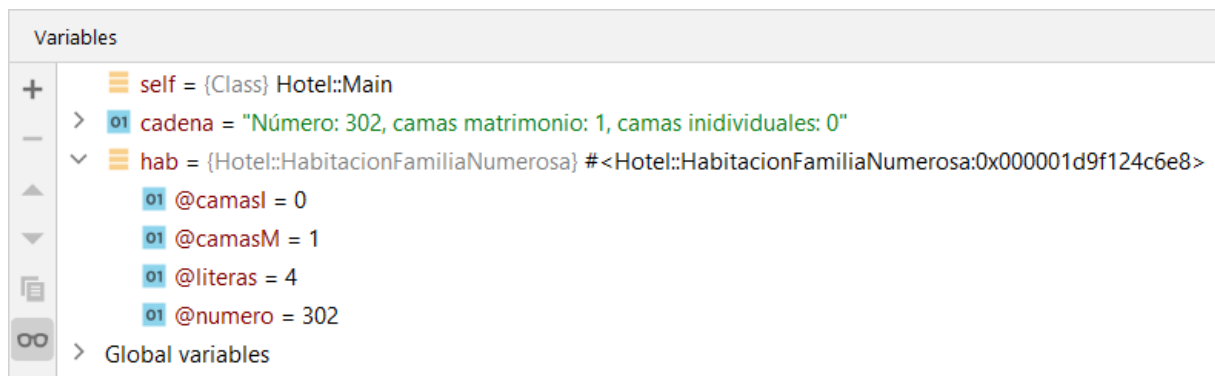
La primera vez que se intenta depurar un programa aparecerá el siguiente mensaje indicando que es necesario instalar unos plugins (*gems*) para la depuración:



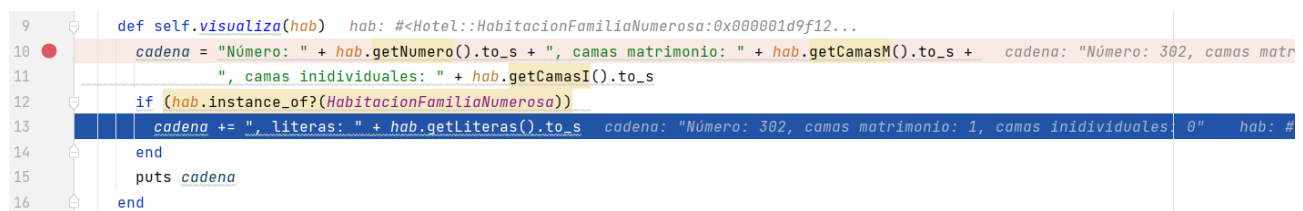
Pulsamos “Yes” y esperamos a que se descarguen las gemas y se instalen. Ten en cuenta que este proceso puede tardar un poco, ya que es necesario descargar ficheros desde internet y compilarlos.

Una vez que el flujo de control del programa llegue al *breakpoint*, la ejecución se pausará para que puedas seguirla paso a paso y la línea de código correspondiente se coloreará en color azul. Además, aparecerá en la parte inferior de la ventana una pestaña *Debug* que contiene la barra de herramientas de depuración ( *Debugger*  ) y dos paneles de depuración (*Variables* y *Console*).

A partir de aquí se sigue un proceso similar al descrito previamente para NetBeans para ejecutar paso a paso cada línea de código (teclas F7 y F8 o los botones correspondientes de la barra de herramientas), para seguir hasta el siguiente *breakpoint* (F9), para ver el valor de una variable situando el cursor sobre ella y para ver los variables de las distintas variables que están en el ámbito de ejecución en el panel *Variables*:



Los valores mostrados en la imagen anterior para la variable *cadena* y el objeto *hab* se corresponden con la ejecución del programa parada en la línea 13, tal y como se muestra en la siguiente imagen:



Para obtener más detalles de una variable se pulsa sobre el símbolo “>” que hay junto a ella. Pulsando sobre el icono + del panel *Variables* se puede añadir un “New watch” de manera análoga a la descrita anteriormente para NetBeans.

Para finalizar, indicar que según se va avanzando paso a paso, la salida estándar del programa se muestra en el panel “Console”.