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 Qytetet siguiendo el paradigma OO.
- Interpretar los resultados obtenidos tras ejecutar un determinado código.

A) Programación y objetivos

Tiempo requerido: Dos sesiones, S1 y S2 (cuatro horas).

Comienzo: **semana del 17 de septiembre.**

Planificación y objetivos:

Sesión	Semana	Objetivos
S1	17-21 septiembre	• Familiarizarse con el entorno de desarrollo Netbeans. • Conocer las características de los lenguajes de programación Java y Ruby. • Aprender a implementar clases, enumerados, paquetes y módulos. • 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 y Ruby. • Aprender a probar código. • Aprender a manejar el depurador de Netbeans.
S2	24-28 de septiembre	• Abundar en la creación de clases y la prueba de código. • Ser capaz de cambiar el código como consecuencia de un cambio en la especificación.
La práctica se desarrollará tanto en Java como en Ruby en equipos de 2 componentes. El examen es individual.		

Nota: El **examen y entrega** de la primera práctica será junto con la segunda práctica, en la semana del 15 al 18 de octubre para todos los grupos excepto para los grupos A1, C2 y D1 que lo tendrán el 26 de octubre.

Enlaces interesantes

http://groups.diigo.com/group/pdoo_ugr/content/tag/Java
https://groups.diigo.com/group/pdoo_ugr/content/tag/ruby

SESIÓN 1

El objetivo general de las prácticas de esta asignatura es la implementación de un juego llamado Qytetet, basado en el Monopoly(c) de Hasbro. En esta primera sesión vamos a empezar a implementar en Java y Ruby la clase Sorpresa del juego Qytetet. Para ello necesitaremos crear también el enumerado TipoSorpresa.

B) S1. Tareas previas

1) Lee el documento *Qytetet – reglas del juego.pdf* para familiarizarte con las reglas del juego.

2) **Piensa las 10 cartas sorpresa** que vas a utilizar en tu juego. Deberás decidir para cada una un mensaje de texto, un valor y su tipo, según lo especificado en la sección 10 de las normas del juego. Por ejemplo:

```
"¡Felicidades!
hoy es el día de
tu no cumpleaños,
recibes un regalo
de todos"

200

PORJUGADOR
```

```
"La liga anti-
superstición te
envía de viaje al
número 13"


13

IRACASILLA
```

3) Entorno de desarrollo: Instala y ejecuta NetBeans.

   a) Si trabajas con los ordenadores del aula, ejecuta NetBeans (desde Ubuntu 16.04).

   b) Si trabajas con portátil propio, instala Java y NetBeans 8.2. desde: https://netbeans.org/. De todas las opciones de descarga proporcionadas, es suficiente con Java SE.

   c) Una vez instalado el entorno ya puedes ejecutarlo.

C) S1. Tareas en Java

1) **Crea un proyecto de tipo *aplicación Java***. Una vez abierto NetBeans, crea un nuevo proyecto denominado *QytetetJava.* Elige el directorio que consideres apropiado para guardarlo e indica que deseas crear un clase que incluya el método *main()* llamada *modeloqytetet.PruebaQytetet.* Además de crearse el proyecto, también se habrá creado un paquete (elemento que aglutina clases) denominado *modeloqytetet.*

Trata de seguir las normas de buena praxis de Java para nombrar ficheros, clases, métodos, atributos, etc. Puedes usar esta guía de bolsillo de Java como referencia:
https://www.safaribooksonline.com/library/view/java-8-pocket/9781491901083/ch01.html

2) **Define en modeloqytetet el tipo enumerado TipoSorpresa** que especifica los tipos de

sorpresas existentes. Para ello, pulsa botón derecho del ratón sobre el paquete y selecciona *Nuevo / Java Enum* y define los siguientes valores *{PAGARCOBRAR, IRACASILLA, PORCASAHOTEL, PORJUGADOR, SALIRCARCEL}* (investiga cómo definir un enumerado y sus valores en Java).

3) **Define en modeloqytetet la clase Sorpresa** con los siguientes **atributos** de visibilidad privada:
   - Para describir la sorpresa, un atributo ***texto*** de tipo *String*.
   - Para indicar el tipo de la sorpresa, un atributo ***tipo*** de tipo *TipoSorpresa*.
   - Un atributo ***valor*** de tipo *int* que tiene diferentes interpretaciones según se indica en la tabla de la sección 10 del documento de reglas del juego.

- **Añade el constructor** de la clase con visibilidad pública asegurándote de que se inicializan correctamente los atributos:
  Sorpresa(String texto, int valor, TipoSorpresa tipo)

- **Define con visibilidad de paquete los consultores** de los atributos definidos anteriormente, cuya funcionalidad es devolver el valor de los mismos.

- **Define el método *toString()*** en la clase *Sorpresa*. Éste método devuelve un *String* con el estado del objeto correspondiente. Utiliza la siguiente implementación:

```
@Override
public String toString() {
     return "Sorpresa{" + "texto=" + texto + ", valor=" +
          Integer.toString(valor) + ", tipo=" + tipo + "}";
}
```

4) **Define en modeloqytetet la clase Qytetet y crea en ella los objetos de la clase Sorpresa** que se corresponden con las cartas sorpresa que has definido en el apartado B.2. Para ello:

a) **Define un atributo de visibilidad privado** de nombre *mazo* y de tipo *ArrayList* donde almacenar los objetos Sorpresa. Los objetos *ArrayList* se declaran y construyen de la siguiente manera

```
ArrayList<TipoElementos> mazo = new ArrayList<>();
```

Ten en cuenta que la clase ArrayList está en el paquete java.util y para que pueda ser utilizada debe ser importada. Para ello, añade la siguiente línea después de la declaración del paquete:

```
import java.util.ArrayList;
```

En general debes importar clases (o paquetes completos, por ejemplo “import java.util.*;” para importar todo el paquete java.util) cuando quieras hacer uso de ellas fuera del paquete donde están definidas.

Define también el consultor getMazo con visibilidad de paquete.

b) **Define un método *inicializarCartasSorpresa()* con visibilidad de paquete** para que se creen e incluyan en el *mazo* todos los objetos sorpresa que se corresponden con las cartas sorpresa definidas anteriormente. Por ejemplo, para crear e incluir en el mazo la carta

sorpresa que manda al jugador a la cárcel, suponiendo como haremos ahora que la cárcel esté en la casilla número 9 (en todo caso deberá ser un número entre 0 y 19, pues el tablero tiene 20 casillas, tal y como se explica en la sesión 2 de esta misma práctica), podemos introducir el código siguiente:

```
mazo.add(new Sorpresa ("Te hemos pillado con chanclas y calcetines,
lo sentimos, ¡debes ir a la carcel!", 9, TipoSorpresa.IRACASILLA));
```

NOTA: En la práctica 2 cambiaremos este código para que sea independiente de la posición elegida para la cárcel.

Para añadir una carta sorpresa que te libera de la cárcel podríamos poner:

```
mazo.add(new Sorpresa ("Un fan anónimo ha pagado tu fianza. Sales
de la cárcel", 0, TipoSorpresa.SALIRCARCEL));
```

5) **Prueba el código desarrollado en la clase Qytetet.**

a) Declara una variable de clase ***juego*** en la clase *PruebaQytetet* y haz que apunte a una instancia de la clase Qytetet.

b) En el método *main()* de la clase *PruebaQytetet i*nvoca al método *inicializarCartasSorpresa()* de Qytetet y muestra el contenido del *mazo* usando el método *toString*(). Para mostrar información usa el método *System.out.print()* o *System.out.println().*

¿Qué pasa si omites la llamada al método *toString()* dentro del argumento del método de impresión?

c) Define en la clase *PruebaQytetet* tres métodos de clase privados que devuelvan un *ArrayList* con los objetos Sorpresa que se indican a continuación:

- Método 1: Sorpresas que tienen un valor mayor que 0.
- Método 2: Sorpresas de TipoSorpresa IRACASILLA.
- Método 3: Sorpresas del TipoSorpresa especificado en el argumento del método.

Utiliza un código parecido al siguiente para recorrer las sorpresas del mazo, teniendo en cuenta que para recorrer los objetos de un ArrayList<Tipo> de nombre ***array***, se utilizaría el código siguiente:

```
for (Tipo t: array()) {
 // acciones sobre cada objeto t de array
}
```

d) Muestra el resultado de invocar cada uno de los métodos desde el método *main.* Invoca el tercero de los métodos tantas veces como tipos de sorpresa existen. Para ello recorre el enumerado usando el método *values()* que devuelve una lista con todos sus valores.

6) Prueba el **depurador** de Netbeans (Debug) con una de las consultas para seguir el flujo de ejecución según se explica en el Apéndice.

D) S1. Tareas en Ruby

1) **Prepara el entorno** para trabajar con Ruby
   - Puedes descargar el plugin para poder trabajar con Ruby en Netbeans en: ~/Escritorio/Departamentos/lsi/pdoo/pluginNetbeansRuby/
   - Abre Netbeans y elige la opción de menú *Tools/Plugins.* En la ventana que se abre, selecciona la pestaña *Downloaded* y haz clic en el botón *Add Plugins*. Navega hasta la carpeta donde están los archivos del plugin, visualízalos todos y elígelos todos. Haz clic sobre el botón *Install* y reinicia Netbeans.
2) **Crea un proyecto** de tipo *aplicación Ruby*. Para ello, una vez abierto NetBeans, crea un nuevo proyecto denominado ***QytetetRuby***. Cambia el nombre del fichero principal (main.rb) por *prueba_qytetet.rb*.

   En los siguientes enlaces puedes consultar dudas referentes al lenguaje Ruby:

   http://rubytutorial.wikidot.com/ruby-15-minutos, o
   http://rubylearning.com/satishtalim/tutorial.html

   Y por supuesto, la documentación oficial del lenguaje:
   http://ruby-doc.org/core-2.4.1/

NOTAS:

- Todo lo realizado en Ruby deberá formar parte del **módulo *ModeloQytetet***. Presta atención pues tendrás que indicarlo explícitamente en cada fichero.
- Para no tener problemas con las tildes y otros caracteres especiales debes incluir en la primera línea de código de todos los archivos .rb la siguiente línea: `#encoding: utf-8`
- En Ruby no hay declaración explícita de tipos de variables ni hay tipos primitivos.
- Los atributos son privados en Ruby.
- Los atributos de instancia se pueden usar sólo dentro de cualquier método de instancia anteponiendo a su nombre @. Lo recomendable es crearlos e inicializarlos dentro del método *initialize*.
- Los atributos de clase se pueden crear en cualquier lugar del código asociado a la clase y anteponiendo a su nombre @@.
- Los métodos de clase se declaran usando el nombre de la clase o *self* para indicar que sólo ella puede ejecutar el método,

```
class Ejemplo
def Ejemplo.metodoDeClase
  o
def self.metodoDeClase
```

- El equivalente del método ***toString*** en Ruby es ***to_s***.

- La clase equivalente a *ArrayList* en Ruby se llama *Array*. Crear un array será por tanto:

```
Array.new
```

3) Define el módulo ***TipoSorpresa*** para implementar enumerados con los símbolos que se corresponden con cada uno de los tipos de sorpresa, por ejemplo:
`PAGARCOBRAR = :Pagar_cobrar` y así para todos.
La forma de acceder a ellos sería `TipoSorpresa::PAGARCOBRAR`
o bien `TipoSorpresa.const_get(PAGARCOBRAR)`

4) **Crea la clase *Sorpresa*** de forma equivalente a lo hecho en Java.

a) **Añade el inicializador** equivalente a lo especificado en el constructor de Java.

b) **Define** de manera implícita **consultores básicos** para todos los atributos.
1. ¿Cómo se haría de forma explícita?

c) **Define el método *to_s*()** en la clase *Sorpresa* de forma equivalente al método toString() de Java. El método *to_s* tendrá la siguiente implementación:

```
def to_s
1. "Texto: #{@texto} \n Valor: #{@valor} \n Tipo: #{@tipo}"
end
```

5) **Crea la clase *Qytetet*** de forma equivalente a lo hecho en Java.

a) **Crea el atributo de instancia *mazo.***

b) **Crea el consultor de *mazo*.**

c) **Define el método *inicializarCartasSorpresa***. Como en Java, en el método se crearán e incluirán en el *mazo* todos los objetos sorpresa. Por ejemplo, para crear y añadir al mazo la carta sorpresa que te libera de la cárcel:

```
mazo<< Sorpresa.new("Un fan anónimo ha pagado tu fianza. Sales de
la cárcel", 0, TipoSorpresa::SALIRCARCEL)
```

6) Las **pruebas en Ruby** se van a realizar de forma equivalente a Java. Para ello, en el fichero *prueba_qytetet.rb, dentro del módulo ModeloQytetet*:

a) **Define la clase *PruebaQytetet*** y el atributo de clase *juego* con una nueva instancia de Qytetet. Ten en cuenta que al crear un fichero tipo módulo no se define la clase y habrá que hacerlo de forma explícita, dentro del módulo *ModeloQytetet*.

b) **Declara un método de clase denominado** *main*, pues tampoco existe en Ruby el equivalente a la clase main de Java y habrá que hacerlo de forma explícita, como método de clase.

c) **Define los métodos de clase para hacer búsquedas** de forma equivalente al punto 5

Java e **invócalos** desde el *main*, de forma equivalente a como hiciste en Java. Para invocar el método que busca por tipo de sorpresa, tantas veces como tipos de sorpresa existen, puedes obtener la lista de enumerados usando el método *constants*, por ejemplo: `TipoSorpresa::constants` y usar *const_get* para acceder a cada valor.

Para mostrar información en la interfaz de texto, usa el método *puts.* Ten en cuenta que para poder usar el nombre de una clase definida en otro archivo es necesario incluir el siguiente código al principio del fichero:

```
require_relative "sorpresa"
```

Por ejemplo, para usar la clase *Sorpresa*, que está definida en el fichero *sorpresa.rb.*

Usa el método *inspect* para mostrar más información sobre los objetos que devuelven los métodos, así por ejemplo:

```
puts objetoQytetet.mazo.inspect
```
mostraría el contenido del mazo de un objeto de Qytetet creado.

d) **Invoca al método *main*** desde fuera de la clase PruebaQytetet:

```
module ModeloQytetet
      class PruebaQytetet
            #Contenido de la clase PruebaQytetet
      end
      PruebaQytetet.main
end
```

SESIÓN 2

En esta sesión se define el tablero del juego Qytetet y sus casillas. Para ello definiremos la clase Tablero y la clase Casilla. Por último, para verificar que se han alcanzado los objetivos de aprendizaje de esta práctica, se sugerirán modificaciones en los requisitos que implicarán cambios en la implementación.

E) S2. Tareas previas

Cada equipo deberá decidir la localidad o barrio, real o ficticio, que será representado en su versión de Qytetet y definir así los nombres de las distintas calles que conforman su tablero. El tablero estará compuesto por 20 casillas, de ellas 12 son de tipo CALLE, 3 son de tipo SORPRESA, 1 de tipo CARCEL, 1 de tipo JUEZ, 1 de tipo PARKING, 1 de tipo IMPUESTO y 1 de tipo SALIDA, esta última posicionada en la casilla número 0.

Para todas las calles que especifiquéis, pensad en sus títulos de propiedad según lo especificado en el documento *Qytetet - Reglas del juego.pdf* y tened en cuenta los siguientes rangos:

- El precio de compra debe estar entre 500 y 1000€
- El precio de edificación debe estar entre 250 y 750€
- El factor de revalorización de la venta estará entre el 10 y el 20% y puede ser positivo si la vivienda se revaloriza o negativo si se devalúa.
- El precio base del alquiler debe estar entre 50 y 100€
- El precio base de la hipoteca debe estar entre 150 y 1000€

F) S2. Tareas en Java

1) **Define dentro del paquete modeloqytetet el tipo enumerado *TipoCasilla*** para especificar los tipos de casillas existentes: *SALIDA, CALLE, SORPRESA, CARCEL, JUEZ, IMPUESTO, PARKING*.

2) **Crea, dentro del paquete modeloqytetet, la clase *TituloPropiedad*** y
   - Declara los siguientes atributos con visibilidad privada:
     - Para indicar el nombre de la calle, un atributo *nombre* de tipo *String*.
     - Para indicar si el título de propiedad está hipotecado o no, un atributo *hipotecada* booleano.
     - Para indicar su precio de compra, un atributo *precioCompra* de tipo *int*.
     - Para indicar el precio base (sin tener en cuenta las edificaciones) que debe pagar quien caiga en la casilla, un atributo *alquilerBase* de tipo *int.*
     - Para indicar cuánto se revaloriza el título de propiedad en el periodo transcurrido entre su compra y su venta, un atributo *factorRevalorizacion* de tipo *float.*
     - Para indicar cuál es el valor base de su hipoteca, un atributo *hipotecaBase* de tipo int.
     - Para indicar cuánto cuesta edificar casas y hoteles, atributo *precioEdificar* de tipo *int*.
     - Para indicar el número de hoteles y casas edificados en ese título, los atributos *numHoteles* y *numCasas* de tipo *int.*
   - Define el constructor teniendo en cuenta que no todos los valores para los atributos tienen por qué pasarse como argumento, ya que en ocasiones es preferible usar valores por defecto. En nuestro caso usaremos valores por defecto para los atributos *hipotecada* (false), *numHoteles* (0) y *numCasas* (0).

- Define los consultores básicos para todos los atributos, el modificador básico del atributo *hipotecada* y el método *toString().*
  NOTA: Para crear métodos básicos (constructores, consultores y modificadores básicos, *toString*...) puedes usar la opción "*Insertar código*" del menú contextual (botón derecho) que aparece cuando ponemos el cursor del ratón sobre cualquier línea dentro del texto de una clase dada).

3) **Crea, dentro del paquete modeloqytetet, la clase *Casilla*** y:
   - Declara los siguientes atributos con visibilidad privada:
     - Para indicar el número de la casilla, que equivale a su posición en el tablero, un atributo *numeroCasilla de tipo int.*
     - Para indicar el coste de esa casilla, que, en el caso de casillas de tipo CALLE, será tomado del *precioCompra* de su título de propiedad, un atributo *coste* de tipo *int.*
     - Para indicar el tipo de la casilla, un atributo *tipo* de tipo *TipoCasilla.*
     - Para asociar la casilla a su título de propiedad, un atributo *titulo* de tipo *TituloPropiedad.*
   - Define dos constructores: uno para las casillas que no son de tipo CALLE y otro para las que son de este tipo y que por tanto tienen título de propiedad. Presta atención a que no todos los valores de los argumentos deben recibirse como parámetro, ya que algunos se pueden inicializar con valores por defecto u obtenerlo de otra parte. En concreto en cada constructor uno de los atributos tiene un valor por defecto. Piensa cuáles son y no los pongas como parámetros. Además, en el constructor de casillas de tipo CALLE, el coste se obtiene del *precioCompra* de su *titulo*.
   - Define consultores básicos para todos los atributos.
   - Define el modificador básico *setTitulo()* como método privado, para ser usado desde el constructor de las casillas de tipo calle y modifica el constructor para hacer uso del mismo.
   - Define el método *toString()*. Asegúrate de que si las casillas tienen título de propiedad, éste se incluya adecuadamente.

4) **Crea, dentro del paquete modeloqytetet, la clase *Tablero*** y:
   - Declara los siguientes atributos:
     - Para contener las casillas, un atributo privado *casillas* de tipo *ArrayList*.
     - Para indicar la cárcel, un atributo privado *carcel* de tipo *Casilla*.
   - Define un constructor sin argumentos, el consultor básico de los dos atributos y el método *toString()*.
   - Define el método privado *inicializar():void*, que será invocado por el constructor. En este método:
     - Asigna un nuevo *ArrayList* al atributo *casillas.*
     - Crea todas las casillas del tablero que habéis diseñado en vuestro equipo. Para cada casilla de tipo CALLE debes crear su título de propiedad correspondiente al crear la casilla.
     - Inclúyelas en el ArrayList. Ten en cuenta que el *numeroCasilla* serán números consecutivos comenzando por 0 y que debes intercalar casillas de tipo CALLE con otros tipos.
     - Inicializa el atributo *carcel* para que apunte a la casilla correspondiente a la cárcel.

5) **Acciones sobre clases creadas en la sesión 1**:
   - En *Qytetet* crea un atributo privado *tablero,* de tipo *Tablero* y añade un consultor básico para dicho atributo. Crea el método privado *inicializarTablero():void* de forma que se cree una instancia de Tablero a la que apunte la variable *tablero*.

- Como ya tenemos definido en el tablero la posición de la cárcel, **modifica en el método *inicializarCartasSorpresa()* de la clase *Qytetet*** la creación de la carta sorpresa que envía al jugador a la cárcel, de forma que el número de casilla que corresponde a la cárcel se obtenga con el siguiente código:

```
tablero.getCarcel().getNumeroCasilla()
```

6) **Pruebas**:

- Modifica el método main de la clase ***PruebaQytetet*** creado en la sesión anterior el código necesario para probar que el tablero y las cartas sorpresa están bien definidos.
  - Para ello añade al final del código de la sesión anterior, el código necesario para inicializar y mostrar el tablero.
  - Si te da una *NullPointerException* en el código anterior piensa a qué puede deberse (puedes usar el debugger si te ayuda) y modifica el código para eliminar dicha excepción.

G) S2. Tareas en Ruby

Realiza lo mismo que se ha especificado hasta ahora para Java, pero teniendo en cuenta las peculiaridades de Ruby vistas en clase de teoría.

1) **Define los módulos *TipoCasilla* (fichero tipo_casilla.rb) y *TipoSorpresa* (fichero tipo_sorpresa.rb) para simular enumerados**: debes poner estos módulos dentro del módulo ModeloQytetet e incluir los valores de los enumerados como objetos *symbol* de Ruby.

2) **Define las clases dentro del módulo ModeloQytetet**: *PruebaQytetet* (fichero *prueba_qytetet*), *Qytetet* (fichero *qytetet*), *TituloPropiedad* (fichero *titulo_propiedad*), *Casilla* (fichero *casilla.rb*), *Sorpresa* (fichero *sorpresa.rb*) y Tablero (fichero t*ablero.rb*) teniendo en cuenta que en Ruby no existe declaración estática de tipos y que en vez de paquetes (como ModeloQytetet), en Ruby usaremos módulos con el mismo nombre. Crea dos constructores de casilla. Repasa para ello el material de teoría donde se explica cómo utilizar varios constructores y cómo declarar métodos privados en Ruby.

NOTAS:

- La palabra reservada para indicar que un método de clase es privado es *private_class_method*.

- Para indicar que el modificador de un atributo *atri* tendrá visibilidad privada en Ruby, es necesario: i) indicar que el atributo tendrá modificador (con *attr_writter* o *attr_accessor* según corresponda) y ii) añadir la línea donde se indica que dicho modificador tendrá visibilidad privada:

```
private :atri=
```

3) **Pruebas**. Realiza todo lo indicado en el apartado pruebas de Java, usando para ello el fichero *prueba_qytetet.rb.*

H) Ejercicios de autoevaluación

Haz una copia de los proyectos antes de realizar los cambios propuestos en los puntos siguientes, pues para la práctica 2 seguiremos con el código tal y como ha quedado hasta el apartado anterior.

Haz los cambios necesarios en tus proyectos Java y Ruby para que:

1) Los tableros tengan un tipo que indique si son ciudades, pueblos o barrios. Crea tableros de distintos tipos y muéstralos.
2) Los títulos de propiedad puedan ser de alto standing o no. Esto dependerá del precio de edificación. Decide qué valor determina si el título es de alto standing o no y qué debes modificar para que se pueda saber si un título de propiedad lo es o no lo es.

Apéndice: 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  }
```

Line Breakpoint

Figura 1: *Line Breakpoint*.

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

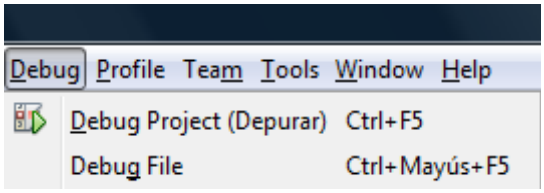

Figura 2: Menú *Debug*.

✔ 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*.

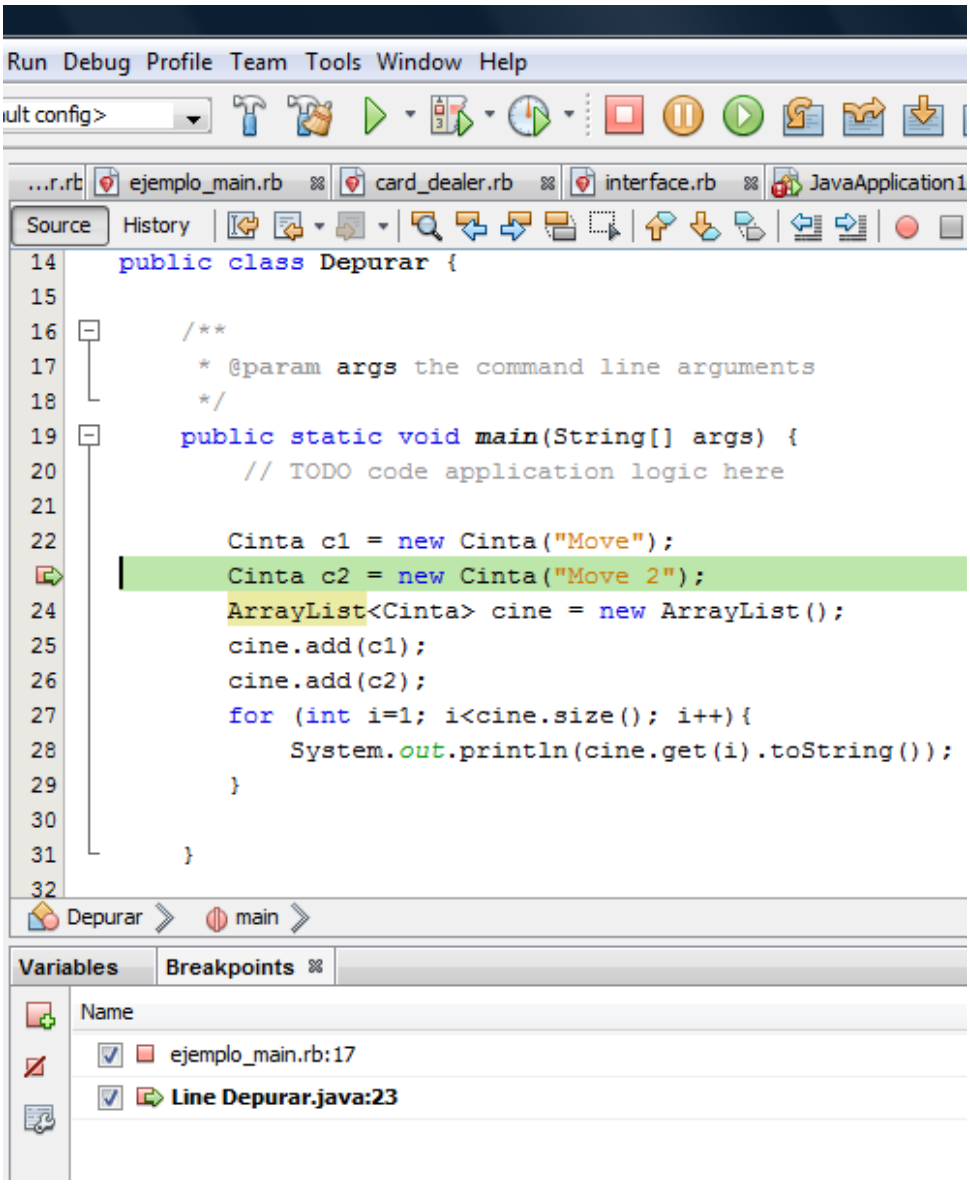

Figura 3: Herramientas de depuración.

✔ 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.

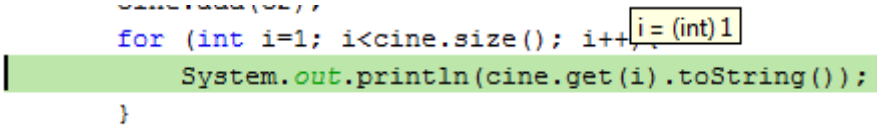

Figura 4: Ver el valor de una variable.

✔ Adicionalmente, en el panel informativo de Variables (abajo en figura 3), 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.

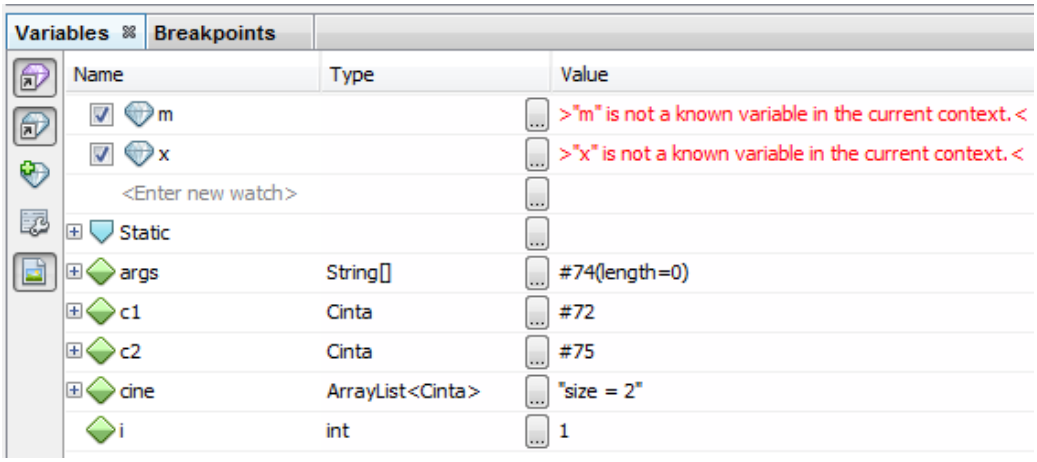

Figura 5: Panel de Variables.

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).

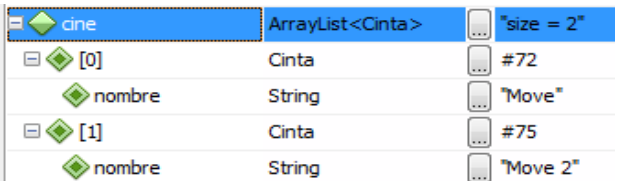

Figura 6: *Detalles sobre la variable cine*.

- ✔ Si lo deseáis, podeis situaros sobre una variable y pulsando *New Watch* en el menú contextual que se despliega, podéis definir un centinela (w*atch)* que permitirá escribir una expresión y consultar su valor en el contexto actual con dicha variable. Por ejemplo, en la figura 7, una operación aritmética definida sobre el valor de **i**, o la llamada a un método del objeto **cine**. Los *watchs* aparecen en el panel de variables, tal y como se observa en la figura.

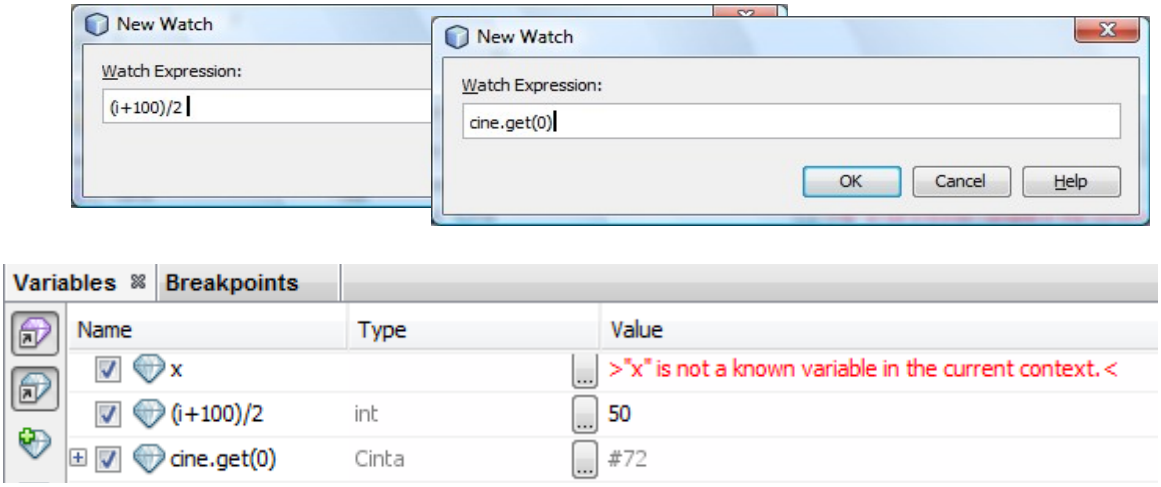

Figura 7: *New Watch*.

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