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.
- Desarrollar parte del juego de Napakalaki siguiendo el paradigma OO.
- Interpretar los resultados obtenidos tras ejecutar un determinado código.

A) Programación y objetivos

Tiempo requerido: Tres sesiones (seis horas).

Comienzo: **semana del 22 de febrero**

Planificación y objetivos:

Sesión	Semana	Objetivos
S1 (Java)	22-2 al 26-2	• Familiarizarse con el entorno de desarrollo Netbeans. • Conocer las características del lenguaje de programación Java. • Aprender a implementar clases. • Conocer el ámbito de las variables y métodos.
S2 (Java)	29-2 al 4-3	• Conocer las características del lenguaje de programación Ruby. • Familiarizarse con los aspectos básicos de las colecciones de objetos y con algunas de las clases que los manejan en Java. • Aprender a manejar el depurador de Netbeans.
S3 (Ruby)	7-3 al 11-3	• Familiarizarse con las clases que manejan las colecciones de objetos en Ruby. • Conocer el ámbito de las variables y métodos en Ruby. • Apreciar la diferencia entre constructor-inicializador.
La práctica se desarrollará de manera individual.		

Examen y entrega de la primera práctica en la semana del 14 al 18 de marzo.

- **Lugar:** Aula en la que de desarrolla la correspondiente sesión de prácticas.
- **Día:** El correspondiente a la sesión de cada uno el periodo indicado.
- **Duración**: 30 minutos al comienzo de la sesión.
- **Tipo examen:** Realizar pequeñas modificaciones y añadidos sobre el propio código.
  Se pedirá entregar el código completo de la primera práctica con las modificaciones pedidas en el examen.

Enlaces interesantes

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

B) Tareas de la Primera Sesión (Java)

En esta primera sesión vamos a empezar a implementar en Java las cartas de monstruos necesarias para el juego Napakalaki. Para ello crearemos una clase Monster y otras dos clases más necesarias para complementarlo, Prize para el buen rollo del monstruo y BadConsequence para el mal rollo.

Recomendación: Plantéate todo lo que vas haciendo. Si no entiendes algo, pregúntale al profesor.

1) Entorno de desarrollo: Instala y/o ejecuta NetBeans

1. Si trabajas con los ordenadores del aula, ejecuta NetBeans (desde Ubuntu 14.04).
2. Si trabajas con portátil propio, instala Java y NetBeans desde:
   https://netbeans.org/
3. Una vez instalado el entorno, ejecútalo.

2) Crea un proyecto de tipo *aplicación Java*. Para ello, una vez abierto NetBeans, crea un nuevo proyecto denominado ***Napakalaki*** en el directorio que consideres apropiado para guardarlo, indicando que se desea crear un clase que incluya el método *main()* llamada *napakalaki.PruebaNapakalaki.*
Además de crearse el proyecto, también se ha creado un paquete (elemento que aglutina clases) denominado *napakalaki.*

3) En el paquete *napakalaki* crea una clase para el buen rollo denominada *Prize.* Para ello, pulsa el botón derecho del ratón sobre el paquete y selecciona *Nuevo / Clase Java*. Para esa clase define:

- Los atributos: *treasures* y *level,* los dos de tipo *int* y visibilidad privada.
- El constructor de los objetos de esa clase: *Prize(int treasures, int level),* su funcionalidad es darle valor a los atributos definidos.
- *L*os consultores básicos de los atributos definidos anteriormente: *getTreasures*() y *getLevels*(). Su funcionalidad es devolver el valor de dichas variables.

4) Crea un enumerado denominado T*reasureKind* para especificar los tipos de tesoros existentes. Para ello, pulsa botón derecho del ratón sobre el paquete y selecciona *Nuevo / Java Enum* y define los siguientes valores *{ARMOR, ONEHAND, BOTHHANDS, HELMET, SHOES}* (investiga cómo definir un enumerado y sus valores en Java).

5) Crea una clase denominada *BadConsequence para el mal rollo del monstruo*. Para esta clase define:

- Los atributos con visibilidad privada:
  - *text,* de tipo *String*, para representar lo que dice un mal rollo.
  - *levels*, de tipo *int*, para representar los niveles que se pierden.
  - *nVisibleTreasures*, de tipo *int*, para representar el número de tesoros visibles que se pierden.
  - *nHiddenTreasures*, de tipo *int*, para representar el número de tesoros ocultos que se pierden.

- Los siguientes constructores con visibilidad pública:
  - *BadConsequence(String text, int levels, int nVisible, int nHidden)*
    Este constructor inicializa los atributos del objeto construido según los parámetros recibidos.
  - *BadConsequence(String text)*
    Este constructor inicializa los atributos (todos) del objeto construido de manera que este objeto represente un mal rollo que suponga la muerte.
    Recuerda que la muerte de un jugador implica que éste pierde su nivel y todos sus tesoros, tanto ocultos como visibles. Dado que un jugador, según las reglas del juego, nunca va a tener más de nivel 10, ni va a tener más de 10 tesoros ocultos, ni más de 10 tesoros visibles; el objeto puede quedar correctamente inicializado para representar la muerte asignándole el valor 10 a sus atributos *levels, nVisibleTreasures,* y *nHiddenTreasures.*
- *L*os consultores básicos de los atributos definidos anteriormente, *getText()*, *getLevels()*... con visibilidad pública. Su funcionalidad es devolver el valor de dichas variables.

6) En la clase *BadConsequence* define dos atributos de tipo lista de *TreasureKind*, de la siguiente forma:
- *private ArrayList<TreasureKind> specificHiddenTreasures = new ArrayList();*
- Igual para *specificVisibleTreasures.*

7) En la clase *BadConsequence* define un nuevo constructor para dar valor a estos nuevos atributos:

BadConsequence(String text, int levels, ArrayList<TreasureKind> tVisible, ArrayList<TreasureKind> tHidden)

¡Recuerda! Un constructor debe inicializar siempre todos los atributos del objeto que está construyendo. Revisa todos los constructores que llevas definidos hasta el momento y asegúrate que se está cumpliendo ese requisito.

8) Crea la clase *Monster* y añade los atributos con visibilidad privada:
- *name*, de tipo *String*, para representar el nombre del monstruo.
- *combatLevel*, de tipo *int*, para representar el nivel de combate del monstruo.

9) Añade a la clase *Monster* los consultores básicos de los atributos definidos anteriormente, cuya funcionalidad es devolver el valor de dichas variables.

10) En clase *Monster* añade un atributo de tipo *Prize*, otro de tipo *BadConsequence* y el constructor de la clase:

Monster(String name, int level, BadConsequence bc, Prize prize)

11) Define el método *toString*() en las tres clases definidas (*Monster, BadConsequence y Prize*). Éste método devuelve un *String* con el estado del objeto correspondiente, por ejemplo en la clase *Prize*, el método *toString*() tendrá la siguiente implementación:

```
@Override
public String toString(){

    return "Treasures = " + Integer.toString(treasures) + " levels = " + Integer.toString(level)}
```

¿Cuál crees que es el significado de la anotación @Override? Piénsalo durante unos minutos y coméntalo con tu profesor.

12) Escribe código para probar lo que has realizado, creando objetos y consultando su estado con los correspondientes métodos *toString*() que has implementado. Puedes hacerlo dentro del *main*() de la clase *PruebaNapakalaki*, o bien crear un método *main*() en la clase *Monster* (en este caso, asegúrate de ejecutar solo esa clase).

C) Tareas de la Segunda Sesión (Java y Ruby)

En esta sesión vamos a crear todas las cartas de monstruos y realizar algunas consultas sobre ellas en Java. También crearemos las clases necesarias para definir el monstruo en Ruby. Aprenderemos además a manejar el depurador de Netbeans.

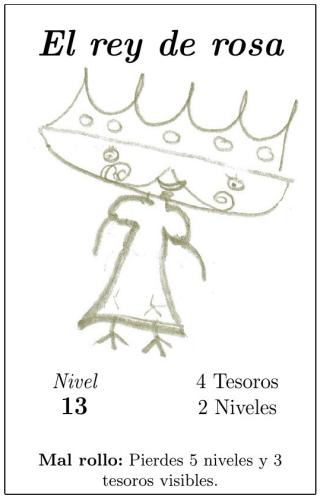
1) Primera parte: Java

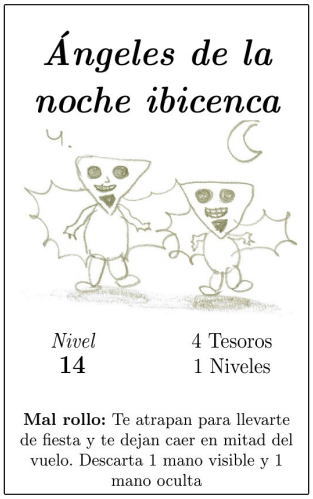
A) Descarga el pdf que contiene las cartas[1] de monstruos.

B) En la clase *PruebaNapakalak*i y dentro del método *main()* define una lista de monstruos haciendo uso de la clase *ArrayList* igual que en la sesión anterior:

ArrayList<Monster> monstruos = new ArrayList();

C) Construye todos los monstruos que están en el pdf descargado e inclúyelos en la lista definida, para ello primero tienes que ver qué tipo de monstruo tenemos y en base a ello construye antes un objeto *BadConsequence* (mal rollo) y otro *Prize* (buen rollo).





Por ejemplo, para la carta de la izquierda sería:

BadConcequence badConsequence = new BadConsequence("Pierdes 5 niveles y 3 tesoros visibles", 5, 3, 0);

Prize prize = new Prize(4,2);

monstruos.add(new Monster("El rey de rosa", 13, badConsequence, prize));

y para la carta de la derecha:

badConsequence = new BadConsequence("Te atrapan para llevarte de fiesta y te dejan caer en mitad del vuelo. Descarta 1 mano visible y 1 mano oculta",0,

new ArrayList(Arrays.asList(TreasureKind.ONEHAND)),

new ArrayList(Arrays.asList(TreasureKind.ONEHAND)));

prize = new Prize(4,1);

monstruos.add(new Monster("Ángeles de la noche ibicenca", 14, badConsequence, prize));

1 Imágenes de las cartas por cortesía de María y Clara.

D) Escribe código para realizar cada una de las siguiente consultas que deben mostrar todos los monstruos (nombre, nivel de combate, buen rollo y mal rollo) que:
   - Tienen un nivel de combate superior a 10.
   - Tengan un mal rollo que implique sólo pérdida de niveles.
   - Su buen rollo indique una ganancia de niveles superior a 1.
   - Su mal rollo suponga la pérdida de un determinado tipo de tesoros ya sea visibles y/o ocultos..

E) Prueba el depurador (apéndice B) de Netbeans (Debug) con una de las consultas. El depurador te permite seguir la traza de ejecución de un programa (paso a paso -F7 o saltando -F8) y averiguar cuando se desee el valor de las variables y estado de los objetos que se están manejando (poniendo puntos de interrupción- breakpoint).

2) Segunda Parte: Ruby

A) Prepara el entorno para trabajar con Ruby
   - Si trabajas con ordenador propio puedes descargar el plugin para poder trabajar con Ruby en Netbeans en (http://plugins.netbeans.org/plugin/38549).
   - Si trabajas en un ordenador del aula, los archivos del plugin se encuentran en

     ~/Escritorio/Departamentos/lsi/pdoo/pluginNetbeansRuby/

   Abre Netbeans y elige la opción de menú *Tools/Plugins.* En la ventana que se abre, seleccionar 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.

B) Crea el proyecto. Procede de la misma forma que cuando creaste el proyecto Java.

   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, bastante buena por cierto

   http://www.ruby-doc.org/core-2.1.3/_lib/racc/rdoc/grammar_en_rdoc.html

C) Crea la clase Prize, para ello procede igual que se hizo con Java.
   - Define el método *initialize(treasures*, *level)* encargado de la inicialización cuando se construye un objeto *Prize*.
   - Define los consultores de las variables de instancia, investiga las dos formas de hacerlo.

D) Define el Modulo *TreasureKind* con los símbolos que se corresponden con cada uno de los tipos de tesoros, por ejemplo: ARMOR = :armor y así para todos. La forma de acceder a ellos sería TrearureKind::ARMOR.

E) Define la clase *BadConsequence* con los mismos atributos y consultores que se indicaron en el apartado 5 y 6 de la primera sesión en Java. En la siguiente sesión definiremos su inicializador y métodos para construir los objetos de esta clase.

F) Define la clase *Monster* con los mismos atributos, inicializador y consultores que se indicaron en el apartado 8 , 9 y 10 de la primera sesión en Java.

G) En la clase PruebaNapakalaki prueba las clases definidas, creando y consultado objetos.

D) Tareas de la Tercera sesión (Ruby)

En esta sesión veremos cómo implementar métodos en Ruby que permitan diferentes formas de crear e inicializar objetos en una misma clase. También crearemos los monstruos del juego en Ruby y realizaremos consultas sobre ellos.

Definiremos un único método *initialize* para dar valor a las variables de instancia de los objetos BadConsequence pero varios métodos de clase para crear los objetos e inicializarlos con diferentes valores en sus parámetros.

1. La cabecera de tu método *initialize* en *BadConsequence* debería ser:

   initialize(aText, someLevels, someVisibleTreasures, someHiddenTreasures, someSpecificVisibleTreasures, someSpecificHiddenTreasures)

2. Define los siguientes tres métodos de clase (en cada uno de ellos se usará *new* para construir el objeto correspondiente):

   BadConsequence.newLevelNumberOfTreasures (aText, someLevels, someVisibleTreasures, someHiddenTreasures)

   BadConsequence.newLevelSpecificTreasures (aText, someLevels, someSpecificVisibleTreasures, someSpecificHiddenTreasures)

   BadConsequence.newDeath (aText)

3. Indica que el método *new* tiene visibilidad privada con: *private_class_method :new*

4. Asegúrate de que has incluido en la cabecera del método *initialize* de la clase Monster todas las variables necesarias (incluidas *Prize* y *BadConsequence)*.

5. Define el método *to_s* (similar a *toString* de Java) en las tres clases definidas (*Monster, BadConsequence y Prize*). Este método devuelve un *String* con el estado del objeto correspondiente, por ejemplo en la clase *Prize*, el método *to_s* tendrá la siguiente implementación:

   ```
   def to_s
       "Tesoros ganados: #{@treasure} \n Niveles ganados: #{@levels}"
   end
   ```

6. En *PruebaNapakalaki* prueba las modificaciones realizadas creando y consultado objetos de las clases modificadas.

7. Construye y rellena el mazo de monstruos, para ello:

   - En *PruebaNapakalak*i define un array en el que se van a incluir todos los monstruos, al igual que se hizo en la sesión de Java.

     monsters = Array.new

   - Construye todos los monstruos que están en el pdf e inclúyelos en el array definido. Por ejemplo, para el primer monstruo de ejemplo de la sesión anterior sería:

     prize = Prize.new(4,2)
     badConsequence = BadConsequence.newLevelNumberOfTreasures('Pierdes 5 niveles y 3 tesoros visibles',5 , 3, 0)
     monsters << Monster.new('El rey de rosa',13,badConsequence,prize)

     Para el segundo ejemplo de la sesión anterior sería:

     prize = Prize.new(4,1)
     badConsequence = BadConsequence.newLevelSpecificTreasures ('Te atrapan para llevarte de fiesta y te dejan caer en mitad del vuelo. Descarta 1 mano visible y 1 mano oculta', 0,[TrearureKind::ONEHAND], [TrearureKind::ONEHAND])
     monsters<< Monster.new('Ángeles de la noche ibicenca', 14, badConsequence, prize)

   ***NOTA**: para no tener problemas con las tildes debes incluir en la primera línea de todos los archivos .rb la siguiente línea: #encoding: utf-8*

8. Escribe el código necesario para llevar a cabo las siguientes funcionalidades:

   - Mostrar todos los monstruos que tengan un nivel de combate superior a 10.
   - Mostrar todos los monstruos que tengan un mal rollo que implique sólo pérdida de niveles.
   - Mostrar todos los monstruos que tengan un buen rollo que indique una ganancia de niveles superior a 1.
   - Mostrar todos los monstruos que tengan un mal rollo que suponga la pérdida de un determinado tipo de tesoros ya sea visibles y/o ocultos. Debe mostrarse el nombre, nivel de combate, buen rollo y mal rollo de cada monstruo.

NOTA: Antes de terminar la práctica comprueba que no te falte ningún monstruo y que las cartas de monstruos sean las que están en *cartasMonstruos.pdf* de Prado.

Apéndices

A: Plantilla para la definición de una clase en Java

```
//Cabecera de la clase
[public] [abstract|final] class nombreDeLaClase [extends NomSuperClase]
                                                 [implements NombreInterface, ...]

//Cuerpo de la clase
{
    //Definición de variables que almacenan el estado de objetos  de esta clase
      [public|protected|private] [static] [final] tipo nomVar [=inicialización];
        ...

    //Definición de los métodos que definan la funcionalidad de los objetos

    //Cabecera del método
    [public|protected|private] [static] [abstract|final] returnType
            nombreMetodo ([listParametros]) [throws listaExcepciones]

   //Cuerpo del método
    {//Código java que implemente el método}
        ...
}
```

B: Depurar el código

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 línea 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");
            Cinta c2 = new Cinta("Move 2");
24          ArrayList<Cinta> cine = new ArrayList();
            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).

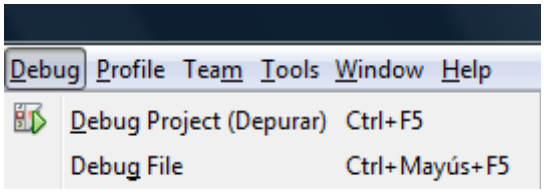


Figura 2: Menú *Debug*.

✔ 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*.

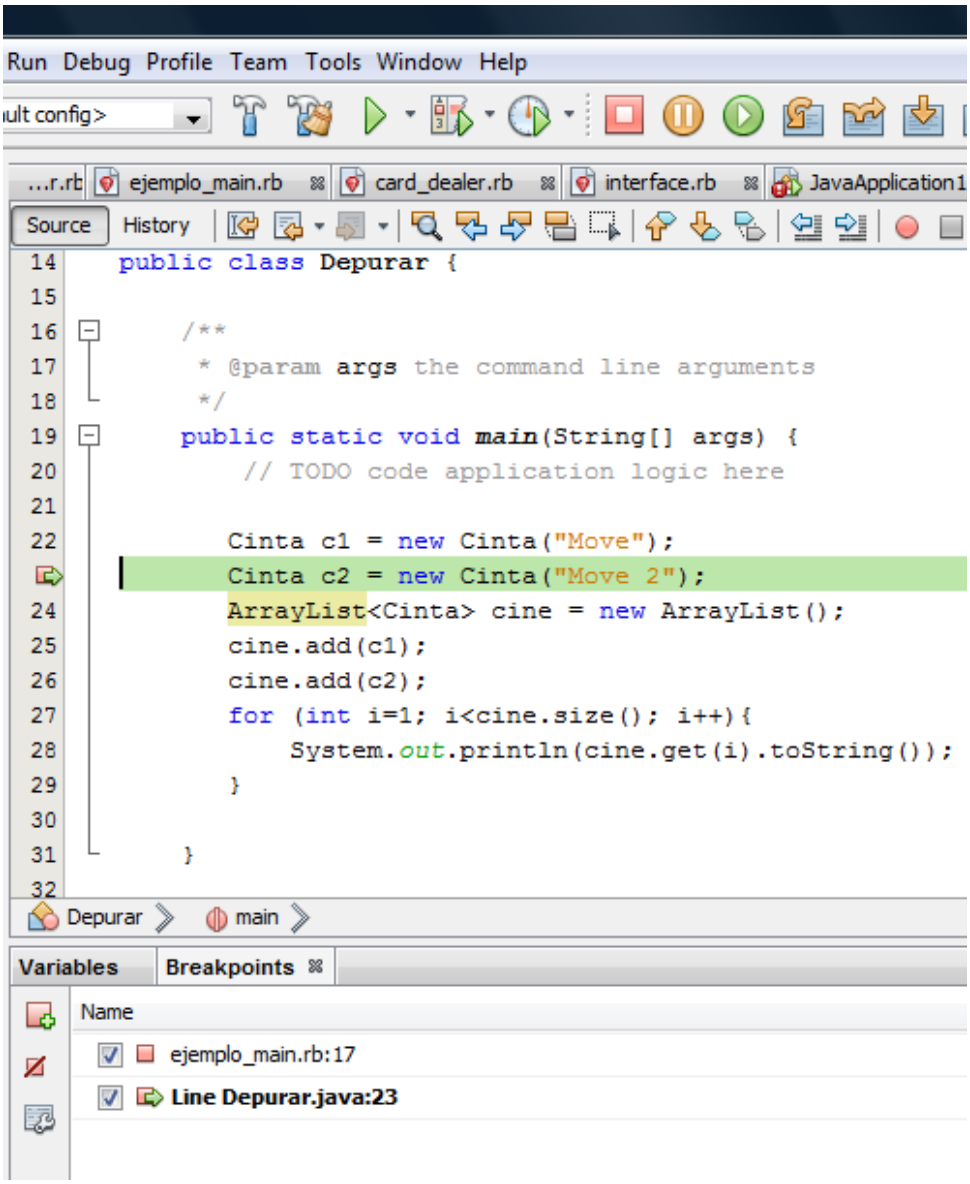


Figura 3: Herramientas de depuración.

✔ 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.

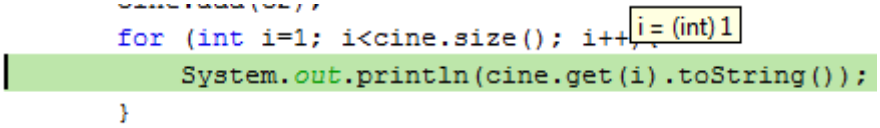


Figura 4: Ver el valor de una variable.

✔ 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.

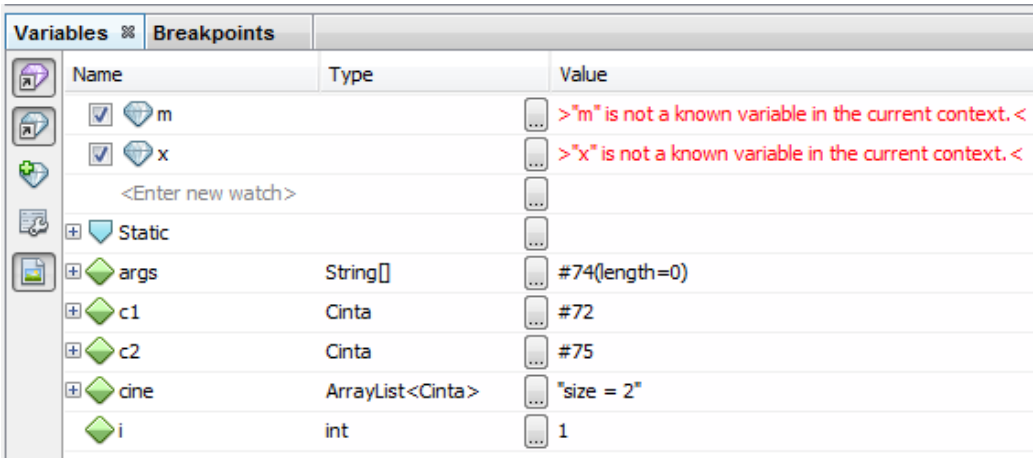


Figura 5: Panel de Variables.

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).

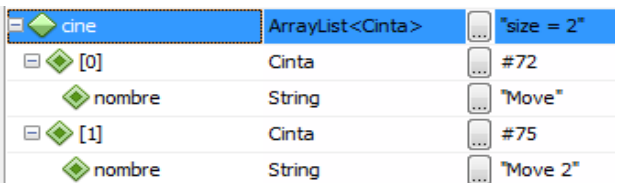


Figura 6: *Detalles sobre la variable cine*.

✔ Si lo deseáis, podéis 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.

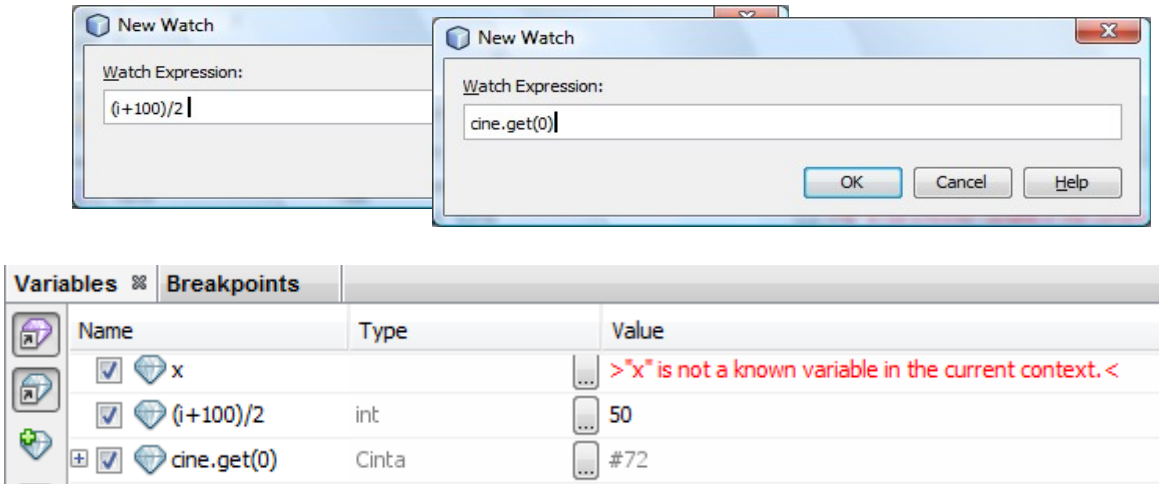


Figura 7: *New Watch*.

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