**Academia Java – JAVA**

***Clases y objetos***

Tabla de contenido

[CLASES Y OBJETOS 3](#_Toc309840944)

[1. Guerra de sillas (o de cómo lo objetos pueden cambiar tu vida) 4](#_Toc309840945)

[1.1. Las especificaciones! 4](#_Toc309840946)

[En el cubículo de Larry 5](#_Toc309840947)

[En la computadora portátil de Brad (al lado de su café) 5](#_Toc309840948)

[1.2. Cambios en la especificación 6](#_Toc309840949)

[De vuelta en cubículo de Larry 6](#_Toc309840950)

[En la computadora portátil de Brad desde la playa 6](#_Toc309840951)

[Larry se coló unos momentos delante de Brad. 7](#_Toc309840952)

[1.3. Pensando en objetos… 13](#_Toc309840953)

[1.3.1. Ejemplo de aplicación 14](#_Toc309840954)

[1.4. Me hablas de objetos y clases ¿Cuál es la diferencia? 14](#_Toc309840955)

[1.5. Construyendo objetos 16](#_Toc309840956)

[1.5.1. Ejemplo de aplicación 17](#_Toc309840957)

[1.6. Avancemos! Vamos más allá del main! 18](#_Toc309840958)

[1.6.1. Jugando a las adivinanzas! 18](#_Toc309840959)

[1.7. Sé el compilador 24](#_Toc309840960)

[1.8. ¿Quién es quién? 24](#_Toc309840961)

# CLASES Y OBJETOS

Te había comentado en el capítulo anterior que todos serian objetos. Y es cierto! Le dejaremos de lado la programación procedural!

En el tema anterior pusimos todos nuestro código en el método main(). Eso no fue precisamente POO. Ahora si dejaremos de completamente de lado el mundo procedural, y comenzaremos a hacer nuestros algunos objetos.

Revisaremos las diferencias entre una clase y un objeto.

# Guerra de sillas (o de cómo lo objetos pueden cambiar tu vida)

Érase una vez en una tienda de software, dos programadores a los se les dio las mismas especificaciones y se les dijo que implementaran los solicitado. El fastidioso jefe del proyecto obligó a los dos programadores a competir, bajo la promesa que el primero que entregara el producto, obtendría una de las deseadas sillas Aeron TM,que todos los trabajadores de Silicon Valley tienen. Larry, el programador procedural, y Brad, el chico OO, sabían que sería pan comido.



Larry, sentado en su cubículo, pensaba para sí mismo "¿Qué es lo que este programa tiene que hacer? ¿Qué procedimientos se necesitan?". Y él mismo respondió: "**girar** y **reproducirSonido**". Así se fue la construcción de los procedimientos. Después de todo, ¿qué es un programa, si no un montón de procedimientos?



Brad, por su parte, se preparó un nuevo café y pensó para sí mismo: "¿Qué **cosas** encontramos en este programa? ¿Cuáles son los principales *actores*?". Lo primero que piensa es en **Las Formas**. Por supuesto, hay otros objetos en los que también piensa como el usuario, el sonido, y el evento clic. Pero ya tenía una biblioteca de código para esas piezas, por lo que se concentró en construir Formas. Sigue leyendo para ver cómo Brad y Larry construyeron sus programas, y sobre todo para saber la respuesta a la pregunta candente "¿Quién se llevó la Aeron?".

# Las especificaciones!

Se requieren contar con formas geométricas desde GUI. Específicamente un cuadrado, un círculo y un triángulo. Cuando el usuario hace clic en una de las formas forma, esta girará en sentido horario 360º y reproducirá un archivo de sonido AIF específico y asignado a la forma particular.



# En el cubículo de Larry

Como lo ha venido haciendo al menos tropecientas veces, Larry se dedicó a escribir sus **importantes procedimientos**. Codificó **rotar** y **reproducirSonido** en poquísimo tiempo.

rotate(shapeNum) {

// make the shape rotate 360º

}

playSound(shapeNum) {

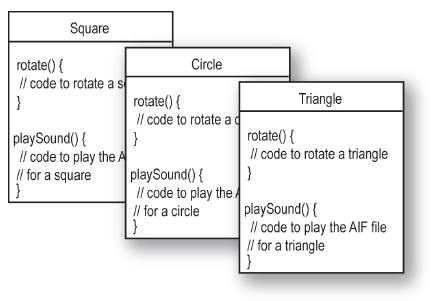
// use shapeNum to lookup which

// AIF sound to play, and play it

}

# En la computadora portátil de Brad (al lado de su café)

Por otro lado, Brad escribió una ***clase*** para cada una de las tres formas.



**Larry pensó que había dado en el clavo. Casi podía sentirse cómodamente sentado en su Aeron y...**

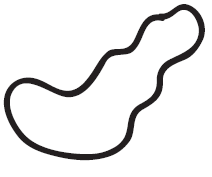
***¡Esperen! ¡Hay un cambio de especificaciones!***

"Bien, *técnicamente* fuiste el primero, Larry", dijo el jefe, "pero tenemos que añadir una nueva *cosita* al programa. Lo cual no va a ser un problema para programadores tan cracks como ustedes dos."

"*Si me dieran un centavo por cada vez que he oído eso*", pensó Larry, a sabiendas de que las especificaciones que empezaban con un-cambio-no-significa-gran-problema eran una **fantasía**. "*Y sin embargo, Brad se ve tan sereno. ¿Qué pasa?*". Sin embargo, Larry se agarró con fuerza a su creencia fundamental de que la forma OO, si bien es simpática, era leeeeeeeeenta. Si alguien quería cambiar su mente, tendría que sacarlo frío y muerto.

# Cambios en la especificación

Se solicita agregar una forma de ameba en la pantalla, que como los demás, cuando el usuario haga clic en la ameba girará al igual que las otras formas, y reproducirá un archivo de sonido .HIF.



# De vuelta en cubículo de Larry

El procedimiento de rotación sigue funcionando tal cual, el código utiliza una tabla de búsqueda para que coincida con un shapeNum a un gráfico. Pero reproducirSonido si que cambiará. Y a todo esto ¿Qué demonios es un archivo. HIF?

Al final, resultó no ser tan importante, pero todavía le ha tocado revolver un poco el código que ya había probado antes. Bueno, el sabe y tiene en cuenta que no importa lo que dice el jefe de proyecto, las especificaciones siempre cambian y eso siempre impacta.

playSound(shapeNum) {

// if the shape is not an amoeba,

// use shapeNum to lookup which

// AIF sound to play, and play it

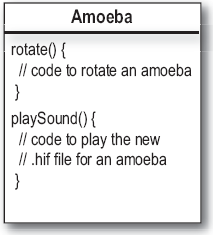
// else

// play amoeba .hif sound

}

# En la computadora portátil de Brad desde la playa

Brad sonrió, tomó un sorbo de su margarita, y se puso a codificar su nueva clase. A veces lo que más amaba sobre OO era que él no tenía que tocar el código que ya había probado y entregado. "La flexibilidad, la extensibilidad,..." se dijo, mientras reflexionaba sobre las ventajas de la OO.



# Larry se coló unos momentos delante de Brad.

(*Hah! Tanto por esos disparates de OO*). Pero la sonrisa en el rostro de Larry desapareció cuando el jefe de proyecto muy, muy molesto, dijo (con tono de decepción), "Oh, no, *no es así* como la ameba debe girar...”.

Resulta que los programadores codificaron la rotación de la siguiente forma:

|  |  |
| --- | --- |
| * Determinar el rectángulo que rodea la forma. * Calcular el centro de ese rectángulo, y girar la forma alrededor de ese punto. |  |

Sin embargo, la forma de ameba se supone que debe girar alrededor de un punto en un extremo, como un reloj de mano.

|  |  |
| --- | --- |
|  |  |
| Punto de rotación de la ameba en el desarrollo de Larry y Brad. | Punto de rotación correcto. |
|  |  |

"*Estoy tostado!*" pensó Larry, visualizándose carbonizado en una tostadora. "Aunque, hmmmm. Podría añadir otro if/else con el procedimiento de rotación, y luego sólo le agrego un poquito de ***código en duro*** para fijar el punto de giro de la ameba. Eso probablemente no va a malograr nada." Sin embargo, la pequeña voz en la parte posterior de la cabeza dijo: "Ohhh eso será un gran error. ¿De verdad piensas que la especificación no va a cambiar otra vez?".

**De vuelta en cubil de Larry**

Larry pensó que lo mejor era añadir los puntos de rotación como argumentos al de procedimiento rotación. **Una gran cantidad de código se vio afectada.** Probar, recompilar, todo el camino nuevamente.

rotate(shapeNum, xPt, yPt) {

// if the shape is not an amoeba,

// calculate the center point

// based on a rectangle,

// then rotate

// else

// use the xPt and yPt as

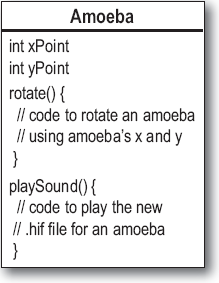
// the rotation point offset

// and then rotate

}

**Brad en la computadora portátil desde el jardín del Festival de Telluride Bluegrass**

Sin perder el tiempo ni el ritmo, Brad modificó el **método** de rotación, pero sólo en la clase Ameba. **El no tocó las pruebas, y lo trabajado del código compilado de otras partes del programa.** Para dar la ameba un punto de giro, agregó un **atributo** que todas las amebas tendrían. Modificó, probó y envió (vía Wi-Fi) la nueva versión de su programa.



**Por lo tanto, Brad, el chico OO, consiguió la silla, ¿no?**

No tan rápido. Larry ha encontrado un error en el enfoque de Brad. Y, ya que estaba seguro de que él ha conseguido la silla (y con ella también una cita con Lucy, la chica de contabilidad, que él asegura impresionará con la silla).

Larry: Usted tiene código duplicado! El procedimiento de rotación está en todas esas cosas…las formas que has usado.

Brad: Es un **método**, no un procedimiento. Y son **clases**, no *cosas*.

Larry: Lo que sea. Es un diseño tonto. Tienes que realizar el mantenimiento de *cuatro* diferentes "métodos" de rotación. ¿De qué forma puede ser bueno eso?

Brad: ¡Oh! Supongo que no viste el diseño final. Te voy a enseñar cómo funciona la **herencia** OO, Larry.



|  |
| --- |
|  |
|  |
|  |

Puede leer esto como, "Square hereda de Shape", "Circle hereda de Shape", y así sucesivamente. Quité los métodos rotate() y playSound () de las otras formas, y ahora sólo hay una sola copia para mantener. A la clase Shape se le denominará superclase de las otras cuatro. Los otros cuatro son subclases de Shape. Las subclases heredan los métodos de la superclase. En otras palabras, si la clase Shape tiene una funcionalidad, las subclases automáticamente obtendrán la misma funcionalidad.

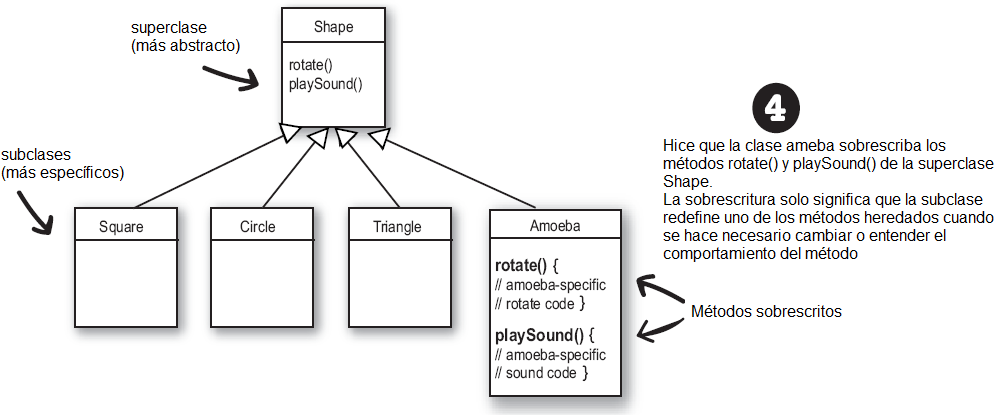
**¿Qué hay acerca del método rotate() de Ameba?**

Larry: Bueno, ese no es el gran problema aquí – sino que la forma de Ameba tiene los procedimientos rotate() y playSound() diferentes?

Brad: Métodos.

Larry: Como sea. ¿Cómo la ameba puede hacer algo diferente si “hereda” la funcionalidad de la clase Shape?

Brad: Ese es el último paso. La clase Ameba sobrescribe los métodos de la clase Shape. Luego, en tiempo de ejecución, la JVM sabe exactamente que método rotate() ejecutar cuando alguien le pida a Ameba girar.



Larry: Cómo le “dices” a una Ameba que haga algo? ¿No deberías llamar al procedimiento, lo siento – método, y luego decirle que es lo que debe rotar?

Brad: Esta es algo realmente cool acerca de OO. Cuando llegue el momento de decirle al triangulo que rote, el código invoca (llama) al método rotate() en un objeto triangulo. El resto del programa realmente no tiene por qué preocuparse cómo lo hace el triangulo. Y cuando sea necesario agregar añadir algo nuevo al programa, se codificará una nueva clase para un nuevo tipo de objeto, entonces los nuevos objetos tendrán su propio comportamiento.

**Al igual que a ti, el suspenso me está matando! Quién se llevó la silla?**

|  |  |
| --- | --- |
|  | **Amy del segundo piso!**  (sin que nadie lo sepa, el Jefe de Proyecto entregó las especificaciones a tres programadores). |

**¿Qué te gusta acerca de OO?**

"Me ayuda a diseñar de una manera más natural. Las cosas evolucionan naturalmente”.

-Joy, de 27 años, Arquitecto de Software

"No jugar con el código que ya he probado, sólo para agregar una nueva función."

-Brad, de 32 años, programador

"Me gusta que los datos y los métodos que operan sobre esos mismos datos se encuentren en una sola clase."

-Josh, de 22 años, bebedor de cerveza

"La reutilización de código en otras aplicaciones. Cuando escribo una nueva clase, puedo hacerla lo suficientemente flexible para ser utilizado nuevamente, más adelante."

-Chris, de 39 años, Gerente de Proyecto

"No puedo creer lo que Chris ha dicho! Él no ha escrito una línea de código en 5 años."

-Daryl, de 44 años, trabaja para Chris

"¿Además de la silla?"

-Amy, de 34 años, programador

**Brain Power: bomba de tiempo a las neuronas**

Acabas de leer una historia acerca de la pelea entre un programador procedural con un programador OO. Ya tienes una visión general de algunos de los principales conceptos OO, incluyendo clases, métodos y atributos. Vamos a pasar el resto de este capítulo revisando las clases y objetos (volveremos a la herencia y sobrescritura más adelante).

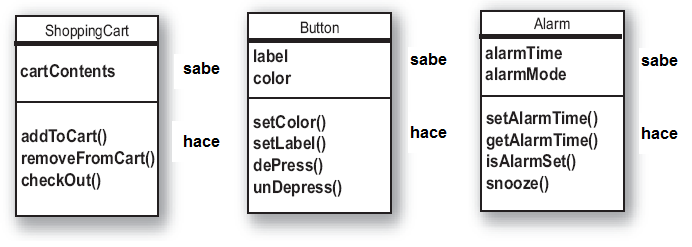
En base a lo que has visto hasta ahora (y lo que sabes de algún lenguaje orientado a objetos con los que has trabajado anteriormente), tomate unos momentos para pensar en estas preguntas:

* ¿Qué es lo fundamental en lo que necesitas pensar a la hora de diseñar una clase Java?
* ¿Cuáles son las preguntas que necesitas preguntarte a ti mismo?

# Pensando en objetos…

Cuando diseñas una clase, piensa acerca de los objetos que serán creados desde este tipo de clase. Piensa en:

* Las cosas que el objeto conoce.
* Las cosas que el objeto hace.



|  |  |
| --- | --- |
| Cosas que el objeto sabe sobre sí mismo se llaman:   * Variables de la instancia |  |
| Cosas que un objeto hace se llaman:   * Métodos |

Las cosas que un objeto ***sabe*** acerca de él mismo son llamadas variables de la instancia. Ellas representan el estado (a través de la data) del objeto, y si puedes tener valores únicos para cada objeto de ese tipo.

Piensa en una **instancia** como otra forma de decir **objeto**.

Las cosas que un objeto puede hacer son llamadas métodos. Cuando diseñas una clase, piensas en que datos del objeto necesitas saber de él y también en los métodos que van a operar sobre esos datos. Es muy común para un objeto tener métodos que lean y escriban los valores de las variables de la instancia.

Por ejemplo, los objetos del tipo Alarma tienen una variable que toma la horaDeAlarma y dos métodos, uno que asigna y otro que lee la horaDeAlarma.

Entonces los objetos tienen variables y métodos diseñados como parte de la clase.

# Ejemplo de aplicación

Completa lo que un objeto de tipo Televisor debe conocer y debe hacer.

|  |  |
| --- | --- |
|  |  |

# Me hablas de objetos y clases ¿Cuál es la diferencia?

|  |  |
| --- | --- |
|  | Una clase no es un objeto, pero sirve la construirlo. |
| Una clase es el proyecto original de un objeto. Le dice a la JVM como crea/hace un objeto de un tipo en particular. Cada objeto creado desde esa clase tiene sus propios valores para sus variables. Por ejemplo, podrías usar la clase Button para hacer docenas de botones diferentes, cada botón podría variar en color, tamaño, forma, etiqueta y así sucesivamente. |  |
|  | **Una más…**  Míralo de esta forma. Podríamos usar una analogía con una persona. Si nosotros tenemos la clase persona, de la cual conocemos su nombre, edad, fechaDeNacimiento y dni. Estos representan los estados de las personas a través de los diferentes valores que estas variables pueden tomar para cada una de las instancias que se vayan a crear de esta clase (Adrián, Juan y Alejandra).  Estas variables pueden modificar sus valores a través de métodos, entre ellos alguno que permita algún tipo de validación para asegurarnos que algunos de los atributos tengan valores únicos (verificarDniUnico()). |

# Construyendo objetos

¿Qué es lo que necesitas para crear y usar un objeto? ¡Solo dos clases! Una clase para el tipo de objeto que deseas usar (Dog, AlarmClock, Television, etc.) y otra clase para probar tu nueva clase.

En la clase de pruebas es donde colocarás tu método principal. Asimismo, en este método crearás tus objetos, accederás a ellos y modificarás los valores de sus variables.

Desde este punto en adelante, notarás el uso de clases en varios ejemplos. La clase real – la clase cuyos objetos quieres usar, y la otra clase que será la clase de pruebas, la cual se llamará <comoSeaQueSeLlameTuClase>TestDrive. Por ejemplo, si creamos la clase Persona, también crearemos la clase PersonaTestDrive.

Solo la clase <nombreDeTuClase>TestDrive tendrá el método main() y su propósito será crear objetos del nuevo tipo y a través del uso del operador punto (.) accederás a los métodos y variables de los nuevos objetos. Esto se hará más claro en los siguientes ejemplos.

Tenemos la clase Dog:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
| C:\Users\ingas\Desktop\Atencion2.png | * Si ya tienes algunos conocimientos de POO, te habrás dado cuenta que no estás usando la encapsulación. Eso se verá más adelante. |

# Ejemplo de aplicación

|  |  |
| --- | --- |
| Haremos la aplicación de todo lo explicado a través de un ejemplo de películas!  La clase MovieTestDrive crea objetos (instancias) de la clase Movie y usa el operador punto (.) para asignar las variables de la instancia un valor en específico. Además, también invoca (llama) al único método para cada uno de los objetos creados. |  |

Tenemos la clase Movie:

**public** **class** Movie {

String title;

String genre;

**int** rating;

**void** playIt() {

System.*out*.println("Playing the movie");

}

}

Y la clase de pruebas:

**public** **class** MovieTestDrive {

**public** **static** **void** main(String[] args) {

Movie one = **new** Movie();

one.title = "Gone with the Stock";

one.genre = "Tragic";

one.rating = -2;

Movie two = **new** Movie();

two.title = "Lost in Cubicle Space";

two.genre = "Comedy";

two.rating = 5;

two.playIt();

Movie three = **new** Movie();

three.title = "Byte Club";

three.genre = "Tragic but ultimately uplifting";

three.rating = 127;

}

}

# Avancemos! Vamos más allá del main!

Usar el main, no es realmente usar POO. Una verdadera aplicación OO, requiere de objetos hablando con otros objetos.

Hay dos formas de usar el main():

* Para probar las clases reales.
* Para lanzar/inicializar tu aplicación Java.

Una verdadera aplicación Java no es más que una conversación entre objetos. Con conversación se refiere a objetos llamado a métodos de otros objetos.

Anticipándonos un poquito a como sería una aplicación Java real, haremos un pequeño ejemplo (si, otro más). Quizás como estamos en los primeros temas del curso lo encuentre un poco torpe e ineficiente. Es más, es posible que desees mejorarlo y eso es precisamente lo que se hará más adelante. Por ahora el objetivo es ver la comunicación entre objetos.

# Jugando a las adivinanzas!

**Resumen:**

El juego de adivinanzas involucrará un objeto "juego" y tres objetos 'jugador'. El juego generará un número aleatorio entre 0 y 9, y los tres objetos jugadores intentarán adivinar. (No dijimos que sería un juego muy emocionante!)

**Clases:**

|  |  |
| --- | --- |
| GuessGame.class |  |
| Player.class |  |
| GameLauncher.class |  |

**La lógica:**

1. La clase GameLauncher es donde la aplicación se lanza; es el que tendrá el método main().
2. En la clase main(), un objeto GuessGame es creado y su método startGame() es llamado.
3. El método startGame() de GuessGame es donde se desarrolla el juego. Crea los tres jugadores, luego “piensa” en un número aleatorio (el objetico a adivinar por los jugadores). Luego, pregunta a cada jugador por su conjetura, verifica el resultado, e imprime la información acerca de si hubo algún ganador o reinicia todo de nuevo.

|  |  |
| --- | --- |
| **public** **class** **GuessGame** {  Player p1;  Player p2;  Player p3; | GuessGame tiene tres variables para los tres objetos jugadores. |
| **public** **void** startGame() {  p1 = **new** Player();  p2 = **new** Player();  p3 = **new** Player(); | Crea los tres objetos jugadores y los asigna a tres instancias. |
| **int** guessp1 = 0;  **int** guessp2 = 0;  **int** guessp3 = 0; | Declara tres variables para tomar los tres números adivinados por los jugadores. |
| **boolean** p1isRight = **false**;  **boolean** p2isRight = **false**;  **boolean** p3isRight = **false**; | Declara tres variables para asignar el resultado (verdadero o falso) acerca si los jugadores acertaron o no. |
| **int** targetNumber = (**int**) (Math.*random*() \* 10);  System.*out*.println ("I'm thinking of a number between 0 and 9..."); | Generando el número que los jugadores deben adivinar. |
| **while**(**true**) {  System.*out*.println("Number to guess is " + targetNumber);  p1.guess();  p2.guess();  p3.guess(); | Llamar al método guess () de cada jugador. |
| guessp1 = p1.number;  System.*out*.println("Player one guessed " + guessp1);  guessp2 = p2.number;  System.*out*.println("Player two guessed " + guessp2);  guessp3 = p3.number;  System.*out*.println("Player three guessed " + guessp3); | Obteniendo el número que cada jugador ha propuesto (resultado de correr el método guess ()). |
| **if** (guessp1 == targetNumber) {  p1isRight = **true**;  }  **if** (guessp2 == targetNumber) {  p2isRight = **true**;  }  **if** (guessp3 == targetNumber) {  p3isRight = **true**;  } | Verificar si algún jugador acertó con el número generado aleatoriamente por la  clase. Si algún jugador acertó, se asigna a su correspondiente variable el valor de verdadero (recuerdas que asignamos falso por defecto? ☺) |
| **if** (p1isRight || p2isRight || p3isRight)  {  System.*out*.println ("We have a winner!");  System.*out*.println ("Player one got it right? " + p1isRight);  System.*out*.println ("Player two got it right? " + p2isRight);  System.*out*.println ("Player three got it right? " + p3isRight);  System.*out*.println("Game is over");  **break**; // game over, so break out of the loop  } | Si algún jugador ha ganado (el operador || significa OR) lo reporta. |
| **else** {  // we must keep going because nobody got it right!  System.*out*.println ("Players will have to try again.");  } // end if/else  } // end loop  } // end method  } // end class | Si nadie ganó se mantiene en el bucle y pregunta a los jugadores por otro juego. |

**Ejecutando el juego:**

El código que debe ir en la clase Player.java es el siguiente:

**public** **class** Player {

**int** number = 0;

**public** **void** guess() {

number = (**int**) (Math.*random*() \* 10);

System.*out*.println("I'm guessing " + number);

}

}

Y finalmente, el main:

**public** **class** GameLauncher {

**public** **static** **void** main (String[] args) {

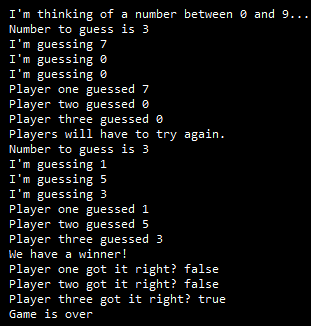
GuessGame game = **new** GuessGame();

game.startGame();

}

}

Al ejecutar el GameLauncher.java obtenemos:



|  |  |
| --- | --- |
| C:\Users\ingas\Desktop\Atencion2.png | **Java saca la basura**  Cada vez que se crea un objeto en Java, entra a una zona de memoria conocida como Heap. Todos los objetos - no importa cuándo, dónde o cómo se creen - viven en el Heap. Pero este Heap, no es cualquier pedazo de memoria, el Java Heap se llama en realidad el Garbage-Collectible Heap. Cuando creas un objeto, Java le asigna un espacio de memoria en el Heap de acuerdo a la necesidad del objeto en particular. Un objeto, con 15 variables, por ejemplo, necesitará probablemente más espacio que un objeto con sólo dos variables. ¿Pero qué sucede cuando se necesita para recuperar ese espacio asignado? ¿Cómo se saca un objeto de la pila cuando se haya terminado el trabajo con él? Java te gestiona la memoria! Cuando la JVM "observa" que un objeto no puede va a ser seguir siendo usado, este se convierte en candidato para la recolección de basura. Y si estás quedándote sin memoria, el recolector de basura se ejecuta, elimina los objetos inalcanzables, y libera espacio, de forma que puede ser reutilizado. Más adelante lo verás a más detalle. |

|  |  |
| --- | --- |
| * POO te permite extender un programa sin tener que tocar el código previamente trabajado y probado. * Todo el código Java está definido en clases. * Una clase describe como hacer de un objeto una clase. Una clase es como un proyecto original, el template o carcasa. Podrías decir que una clase es como un recipiente y los objetos serian galletas que se preparan en el. * Un objeto ***se conoce*** y sabe lo que ***hace***. * Las cosas que conoce el objeto de si mismo son llamados variables. Ellas representan el estado de un objeto. * Las cosas que un objeto hace son llamadas métodos. Ellos representan el comportamiento de un objeto. * Cuando creas una clase, probablemente crearás una clase de pruebas, en la cual crearás objetos de tu nueva clase y probarás en ella. * Una clase puede heredar variables y métodos de una super clase abstracta. * En tiempo de ejecución, un programa en Java no es más que una conversación entre los objetos que creamos. | C:\Users\ingas\Desktop\Atencion2.png |

# Sé el compilador

Para seguir ejercitándonos haremos unos pequeños ejercicios donde nos convertiremos (momentáneamente) en el compilador de Java:

|  |  |
| --- | --- |
| **class** TapeDeck {  **boolean** canRecord = **false**;  **void** playTape() {  System.*out*.println("tape playing");  }  **void** recordTape() {  System.*out*.println("tape recording");  }  }  **class** TapeDeckTestDrive {  **public** **static** **void** main(String[] args) {  t.canRecord = **true**;  t.playTape();  **if** (t.canRecord == **true**) {  t.recordTape();  }  }  } |  |
| **class** DVDPlayer {  **boolean** canRecord = **false**;  **void** recordDVD() {  System.*out*.println("DVD recording");  }  }  **class** DVDPlayerTestDrive {  **public** **static** **void** main(String[] args) {  DVDPlayer d = **new** DVDPlayer();  d.canRecord = **true**;  d.playDVD();  **if** (d.canRecord == **true**) {  d.recordDVD();  }  }  } |  |

# ¿Quién es quién?

De acuerdo a lo que contenga la columna izquierda, identificar en la columna derecha a quien se refiere. Las opciones son:

|  |  |  |  |
| --- | --- | --- | --- |
| Clase | Método | Objeto | Variable de instancia |

|  |  |
| --- | --- |
| **Soy un compilado a partir de un archivo .java.** | *Clase* |
| **Los valores de mis variables de instancia pueden ser diferentes a los valores de mis amigos.** | *Objeto* |
| **Me comporto como una plantilla.** | *Clase* |
| **Me gusta hacer las cosas.** | *Método* |
| **Puedo tener muchos métodos.** | *Clase, objeto* |
| **Yo represento un "estado".** | *Variable de instancia* |
| **Tengo comportamiento.** | *Objeto, clase* |
| **Estoy en los objetos.** | *Método, variable de instancia* |
| **Yo vivo en el Java Heap.** | *Objeto* |
| **Acostumbro crear instancias de objetos.** | *Clase* |
| **Mi estado puede cambiar.** | *Objeto, variable de instancia* |
| **Declaro métodos.** | *Clase* |
| **Puedo cambiar en tiempo de ejecución.** | *Objeto, variable de instancia* |