**Academia Java – JAVA**

***Métodos Extra-Fuertes***

Tabla de contenido

[MÉTODOS EXTRA FUERTES 3](#_Toc310876304)

[1. Construyendo una juego al estilo de acorazado 4](#_Toc310876305)

[1.1. Primero: un diseño a alto nivel 5](#_Toc310876306)

[1.2. Una introducción simple al juego 8](#_Toc310876307)

[Al menos necesitarás dos clases Game y DotCom (punto de combate). En esta parte construirás una versión un poquito más simple del juego, más adelante se la mejoraremos. 8](#_Toc310876308)

[1.3. Desarrollando una clase 9](#_Toc310876309)

[1.3.1. Las tres cosas que haremos cuando escribimos una clase 9](#_Toc310876310)

[1.3.2. Prep code: clase SimpleDotCom 10](#_Toc310876311)

[1.3.3. Test code: escribiendo pruebas para la clase SimpleDotCom 11](#_Toc310876312)

[1.3.4. Real code: el método checkYourself() 13](#_Toc310876313)

[a. Real code: lo nuevo, no visto 15](#_Toc310876314)

[1.3.5. Real code: el final, finalito 15](#_Toc310876315)

[1.3.6. Prepcode para la clase SimpleDotComGame (todo en el main()) 16](#_Toc310876316)

[1.3.7. El método main() del juego 18](#_Toc310876317)

[a. Random y getUserInput 20](#_Toc310876318)

[1.3.8. Una última clase: GameHelper 20](#_Toc310876319)

[1.3.9. A jugar! 21](#_Toc310876320)

[1.3.10. ¡¡¡¡¿¿¿¿Un bicho????!!!! 21](#_Toc310876321)

[1.4. Más acerca de los bucles 22](#_Toc310876322)

[1.4.1. for 22](#_Toc310876323)

[1.4.2. Viaje a través de los bucles 23](#_Toc310876324)

[1.4.3. Diferencias entre for y while 23](#_Toc310876325)

[1.4.4. Un for mejorado 25](#_Toc310876326)

[1.5. Casting de primitivos 26](#_Toc310876327)

[1.6. Sé el compilador 27](#_Toc310876328)

# MÉTODOS EXTRA FUERTES



Vamos a poner un poco de punch en nuestros métodos. Hemos visto variables, juegado con algunos objetos, y escribimos un poco de código. Pero éramos newbies. Ahora necesitamos más. Por ejemplo, más operadores.

Necesitamos más operadores para que podamos hacer cosas un poco más interesantes que ladrar (bark). Y no olvidemos los bucles.

Necesitamos bucles, pero ¿qué pasa con los while? Vamos a necesitar muchos bucles si vamos a programar en serio. Quizás lo inicial podria ser generar números aleatorios. Y convertir cadenas a enteros, yeah, eso sería cool. Eso también seria chido aprender.

¿Y por qué no aprendemos construyendo algo real, para ver qué se siente al codificar (y sobretodo probar!) un programa desde cero.

Qué te parece un juego? Acorazados! Si? Perfecto! Esa será la tarea fuerte ahora y por lo que nos llevará dos capítulos para poder terminarlo.

Vamos a construir una versión simple de este ahora, y luego algo un poco más elegante después.

# Construyendo una juego al estilo de acorazado

Ahora irás tú contra la PC!

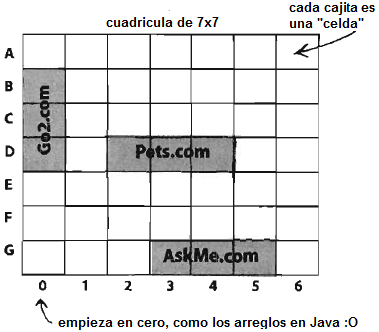
Wait, quizás debamos contarte que a diferencia del verdadero Acorazado, los buques no serán tuyos. Sino que tendrás que hundir el del otro equipo en el menor tiempo posible.

Ah! Y bueno no serán precisamente buques sino que le atinarás a puntos especificos.

Objetivo: Atinarle a todos los puntos de la PC en la menor cantidad de intentos. Con lo cual te asignarán una calificación o nivel, en base a tu performance.

Configuración: cuando el juego sea lanzado, la PC debe elegir 3 puntos de una cuadrícula de 7 x 7. Cuando termine, el juego te pedirá su primera aproximación.

Cómo se juega: como aún no hemos aprendido a construir una interfaz gráfica de usuario, por ello para esta versión usaremos linea de comandos. El juego te pedirá que ingreses la coordenada de una celda (como “A3”, “C5”, etc) y el te responderá “Hit!”, “Miss you”, o “Hundiste a Pets.com”. Si acumulas 3 fallos, pierdes el juego.



La consola de salida mostraria lo siguiente con cada uno de tus intentos:



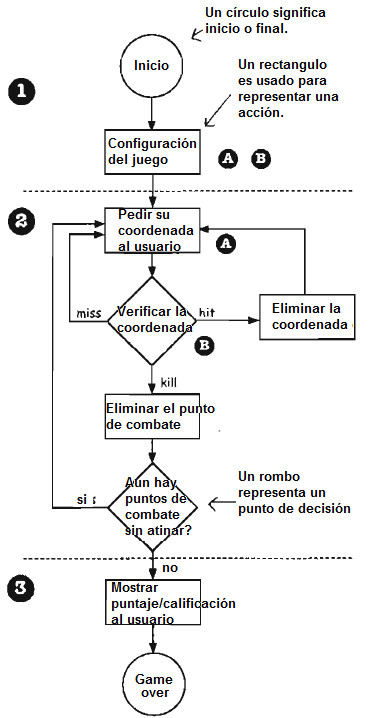
# Primero: un diseño a alto nivel

Sabes que necesitarás clases y métodos, pero cuales deberian ser? Para responder eso, necesitas más información acerca de lo que el juego hará.

Primero debes entender el flujo general del juego. Aquí una idea básica:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | El usuario inicia el juego. |  |
|  |  | El juego crea sus tres puntos de combate. |  |
|  |  | El juego coloca sus tres puntos de combate en la cuadricula virtual. |  |
|  |  | El juego comieza.  Repetir lo siguiente hasta que no hayan más puntos de combate por atinar: |  |
|  |  | El prompt le preguntará al usuario por su coordenada (“A2”, “CO”, etc). |  |
|  |  | El juego verifica la coordenada dada por el usuario, para ver si coincide con alguno de sus puntos de combate. Y toma la acción apropiada. |  |
|  |  | El juego termina.  Le da al usuario su calificación basada en la cantidad de sus intentos. |  |

Un poco más formalmente tendriamos:



Ahora tienes la idea completa del tipo de cosas que el programa necesita hacer. El siguiente paso es determinar los objetos que necesitarás para hacer el trabajo. Recuerda pensar como Brad y no como Larry, enfocate primero en las cosas del programa que en los procedimientos.

# Una introducción simple al juego

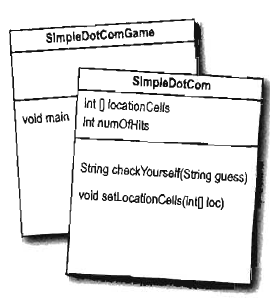
# 

# Al menos necesitarás dos clases Game y DotCom (punto de combate). En esta parte construirás una versión un poquito más simple del juego, más adelante se la mejoraremos.

Todo va a ser simple en este juego. En lugar de una grilla en 2D, solo colocarás los puntos de combate en una sola fila. Y en lugar de tres, solo usarás uno.

El objetivo, seguirá siendo el mismo. El juego aún necesita una instancia de puntos de combate asignada en algún lugar de la fila, obtener la coordenada del usuario, y cuando todos los puntos de combate has sido golpeados, el juego termina. Esto nos ayudará a desarrollar el juego completo rápidamente.

Si podemos hacer este pequeño trabajo, podemos tranquilamente hacer la parte más compleja después.



En esta versión simple, la clase juego no tiene variables de instancia, y todo el código de juego está en el método main. En otras palabras, cuando el programa sea lanzado y el método main() se ejecuta, y crea una sola instancia de DotCom, elije la localización para ella (tres celdas consecutivas para la versión de siete celdas en una sola fila), pregunta al usuario por una coordenada, verifica si el usuario atinó y repite el proceso hasta que los tres puntos de combate sean hallados.

Recuerda que la fila virtual es…virtual Osea, no existe en ninguna parte del programa. Mientras tanto tanto el juego como el usuario saben que el DotCom está oculto en tres celdas consecutivas de un máximo de siete (empezando de cero!), la fila en si misma no está representanda en el código. Podrias estar tentado a construir un array (confiesalo! Lo pensaste! ☺) de siete elementos DotCom pero no es necesario. Todos lo que necesitas es tomar tres celdas de DotCom.

# Desarrollando una clase

Como programador, seguramente tengas una metodologia/proceso/enfoque para escribir código. La secuencia está diseñada para ayudarte a ver (y aprender) que estamos pensando a medida que vamos codificando la clase. No es la forma como generalmente se realiza el desarrollo, porque en el mundo real se sigue preferencias personales, proyectos o lo que el empleador indique. Asi que aprovecharemos en desarrollar lo que nosotros realmente deseamos y como lo deseamos. Y cuando lo que queremos crear es una clase Java inclinando la linea hacia una “experiencia de aprendizaje”, lo hacemos asi:

* Averiguar lo que la clase tiene que hacer.
* Listar las variables de instancia y métodos.
* Escribir el prepcode para los métoso (tranquilo, lo veremos en unos minutos).
* Escribir las pruebas para los métodos.
* Implementar la clase.
* Probar los métodos.
* Depurar y reimplementar según sea necesario.
* Expresar gratitud que no debemos pasar por el trauma de probar esta aplicación con usuarios reales.

# Las tres cosas que haremos cuando escribimos una clase



* Prep code: una forma de pseucodcódigo, para ayudarte a enfocar en la lógica sin estresarte por la sintaxis.
* Test code: las clases o métodos que pondrán a prueba el código real y validarán que esté trabajando correctamente.
* Real code: la implementación real de la clase.

Entonces nos queda por hacer…

**TO DO:**

Clase SimpleDotCom:

* Escribir prep code.
* Escribir test code.
* Escribir Java code final.

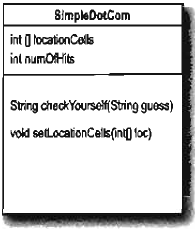
Clase SimpleDotComGame:

Escribir prep code

Escribir test code [no]

Escribir Java code final

# Prep code: clase SimpleDotCom

Seguramente ya tienes idea de cómo el prepcode (nuestra versión de pseudocódigo) funcionará. Si, será a media código Java real y una descripción de la clase. La mayor parte incluye tres partes: declaraciones de variables de instancia, declaraciones de métodos y lógica de los métodos.

La parte más importante del prep code es la lógica de los métodos, porque estos definen que tiene que pasar, lo que más adelante se traducirá en como se hará, cuando tengamos que codificar realmente.

**DECLARA** un *array* de *int* para las posiciones de las celdas. Llamalo locationCells.

**DECLARA** un *int* para guardar el número de hits. Llamalo numOfHits y asignale CERO.

**DECLARA** el método *checkYourself()* que toma el *String* que el usuario ingresa (“1”, “3”, etc) y lo verifica. Para después verificar si el resultado como un “hit”, “miss”, o “kill”.

**DECLARA** el método setter *setLocationCells()* que toma un *array* de *int* (el cual tiene tres posiciones de sus celdas con ints).

**MÉTODO**: *String checkYourself(String userGuess)*

**OBTENER** la suposición del usuario como un parámetro *String*.

**CONVERTIR** la suposición del usuario a un *int*.

**REPETIR** por cada localización de las celdas en el *array* de *int*

**//COMPARAR** la suposición del usuario con la localización de las celda

**SI** la suposición es correcta

**INCREMENTAR** el número de hits

**//AVERIGUAR** si fue la última celda

**SI** la cantidad de hits es 3

**RETORNA** “kill” como resultado

**CASO** **CONTRARIO** si no fue un kill

**RETORNA** “hit”

**FIN** **SI**

**CASO** **CONTRARIO** si la suposición no es correcta

**RETORNA** “miss”

**FIN SI**

**FIN** **REPETIR**

**FIN** **MÉTODO**

**MÉTODO:** *void setLocationCells(int[] cellLocations)*

**OBTENER** la localización como un parámetro *array* de *int*

**ASIGNAR** el parámetro de localización a una variable de instancia

**FIN** **MÉTODO**

# Test code: escribiendo pruebas para la clase SimpleDotCom

Antes de empezar a codificar los métodos, excribe algunos métodos de prueba. Siiiiii, harás las pruebas antes que nada.

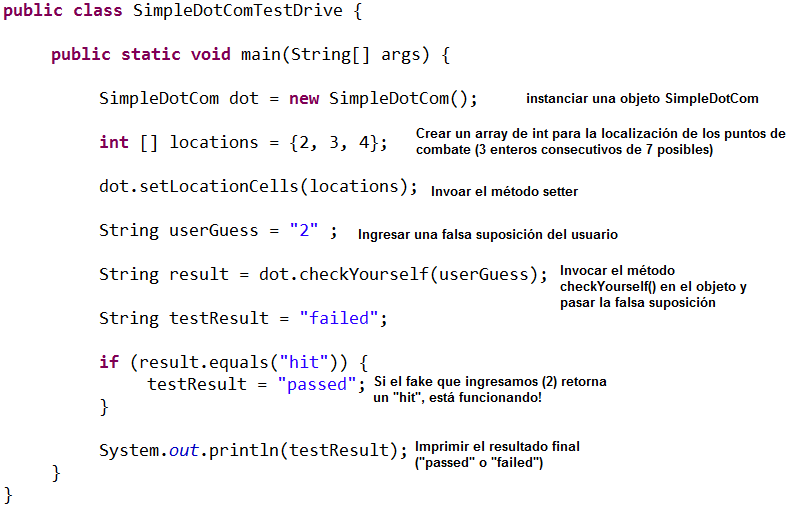
El concepto de hacer las pruebas antes es una de las prácticas de Extreme Programming (XP), y puedes hacerlo muy sencillo para escribir tu codigo. No es que vayas a usar XP, solo usaras una de sus prácticas acerca de sus pruebas. Ahí está lo cool de las prácticas ágiles: usa lo que necesites y de valor. Además, XP suena chido, no?

Entonces necesitamos escribir el código de pruebas que cree el objeto SimpleDotCom y ejecute sus métodos. En realidad par esta clase, solo debemos enfocarnos en el método checkYourself(), sin embargo vamos a tener que implementar también setLocationCells() para lograr que checkYourself se ejecute correctamente.

Dale una mirada al prepcode de checkYourself() (setLocationCells() es muy sencillo, no te preocupes). Ahora preguntate ¿qué código de pruebas implementarás par aprobar que checkYourself() funcionará correctamente?

|  |  |
| --- | --- |
| Basado en el prepcode: | Debemos probar: |
| **MÉTODO**: *String checkYourself(String userGuess)*  **OBTENER** la suposición del usuario como un parámetro *String*.  **CONVERTIR** la suposición del usuario a un *int*.  **REPETIR** por cada localización de las celdas en el *array* de *int*  **//COMPARAR** la suposición del usuario con la localización de la celda  **SI** la suposición es correcta  **INCREMENTAR** el número de hits  **//AVERIGUAR** si fue la última celda  **SI** la cantidad de hits es 3  **RETORNA** “kill” como resultado  **CASO** **CONTRARIO** si no fue un kill  **RETORNA** “hit”  **FIN** **SI**  **CASO** **CONTRARIO** si la suposición no es correcta  **RETORNA** “miss”  **FIN SI**  **FIN** **REPETIR**  **FIN** **MÉTODO** | * Instanciar un objeto SimplDotcom. * Asignar la localización (un array de 3 ints, como {2,3,4}). * Invoca el método checkYourself() pasandole un fake de una suposición del usuario. * Imprimir el resultado para ver si fue correcto (“passed” o “failed”). |

**TAN TAN!** Entonces tenemos…

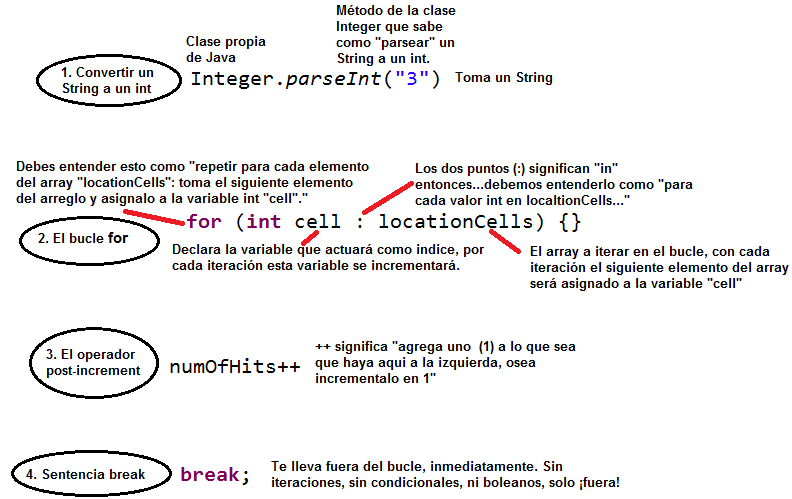


# Real code: el método checkYourself()

|  |  |
| --- | --- |
| **MÉTODO**: *String checkYourself(String userGuess)*  **OBTENER** la suposición del usuario como un parámetro *String*.  **CONVERTIR** la suposición del usuario a un *int*.  **REPETIR** por cada localización de las celdas en el *array* de *int*  **//COMPARAR** la suposición del usuario con la localización de la celda  **SI** la suposición es correcta  **INCREMENTAR** el número de hits  **//AVERIGUAR** si fue la última celda  **SI** la cantidad de hits es 3  **RETORNA** “kill” como resultado  **CASO** **CONTRARIO** si no fue un kill  **RETORNA** “hit”  **FIN** **SI**  **CASO** **CONTRARIO** si la suposición no es correcta  **RETORNA** “miss”  **FIN SI**  **FIN** **REPETIR**  **FIN** **MÉTODO** |  |

# Real code: lo nuevo, no visto

Hay cosas que no habiamos visto antes en esta página. No te preocupes! Los detalles se verán después (¿recuerdas que al final siempre si vimos la encapsulación?). A continuación lo que por el momento te basta saber:



# Real code: el final, finalito

**public** **class** SimpleDotComTestDrive {

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

SimpleDotCom dot = **new** SimpleDotCom();

**int**[] locations = {2, 3, 4};

dot.setLocationCells(locations);

String userGuess = "2";

String result = dot.checkYourself(userGuess);

}

}

**public** **class** SimpleDotCom {

**int**[] locationCells;

**int** numOfHits = 0;

**public** **void** setLocationCells(**int**[] locs){

locationCells = locs;

}

**public** String checkYourself(String stringGuess) {

**int** guess = Integer.*parseInt*(stringGuess);

String result = "miss";

**for** (**int** cell: locationCells){

**if** (guess == cell) {

result = "hit";

numOfHits++;

**break**;

}

}

**if** (numOfHits == locationCells.length){

result = "kill";

}

System.*out*.println(result);

**return** result;

}

}

Hay un pequeño error al acecho. Osea, el código cumple el objetivo, pero a veces…no, mejor no te preocupes por ahora, pero vamos a tener que hacerle frente un poco más adelante.

# Prepcode para la clase SimpleDotComGame (todo en el main())

Hay algunas cosas en las que simplemente tendrás que creer. Por ejemplo, la línea de prepcode que dice: "OBTENER los datos de entrada del usuario". Bueno, te debes imaginar que lo que se quiere implementar es un poco más que solo eso. Como felizmente, usas OO, eso significa que más adelante solo tienes que agregar otra clase/objeto.

Cuando se escribe prepcode, asumes que de alguna forma serás capaz de hacer todo lo que se tiene que hacer, así puedes concentrarte en poner toda tu capacidad en la lógica.

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

**DECLARAR** una variable **int** para manejar el número de intentos del usuario, nombrada numOfGuesses e inicialízala en 0.

**CREA** una nueva instancia de SimpleDotCom

**GENERAR** un número aleatorio entre 0 y 4, que será la posición inicial de la celda de partida

**CREAR** un array de **int** con 3 ints usando la generación aleatoria de números para el primero, luego incrementarlo en 1 y luego en 2 (por ejemplo 3.4,5)

**INVOCAR** el método setLocationCells() de la instancia de SimpleDotCom

**DECLARAR** una variable **boolean** que represente el estado del juego, llámala isAlive

**ASIGNA** **true** a isAlive

**MIENTRAS** los puntos de combate vivan (isAlive == **true**)

**OBTENER** la data ingresada por el usuario

//VERIFICAR el intento del usuario

**INVOCAR** el método checkYourself() de la instancia SimpleDotCom

**INCREMENTAR** la variable numOfGuesses

//VERIFICAR que el punto de combate haya muerto

**SI** el resultado es “kill”

**ASIGNAR** a isAlive **false** (lo que significa que saldremos del bucle y no regresaremos)

**IMPRIMIR** el número de intentos

**FIN** **SI**

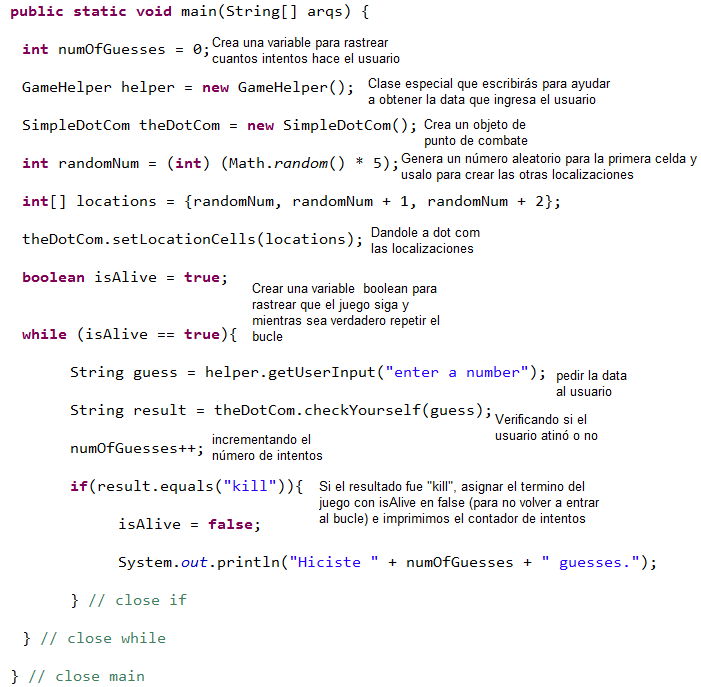
**FIN** **MIENTRAS**

**FIN** **MÉTODO**

|  |  |
| --- | --- |
| * Tu programa Java debe comenzar con un diseño de alto nivel. * Típicamente debes escribir tres cosas cuando creas una nueva clase:   + prepcode   + testcode   + realcode (Java) * El prepcode debe describir que se hace, no como se hace. La implementación viene después. * Usa el prepcode para ayudar al diseño de código de pruebas. * Escribe el código de pruebas antes de implementar tus métodos. * Elegir bucles for, antes que while, cuando sabes cuantas veces vas a repetir el código. * Usa el operador de pre/post incremento para agregar 1 a la variable (x++) * Usa el operador de pre/post decremento para sustraer 1 de una variable (y--) * Usa Integer.parseInt() para obtener un valor int desde un String. * Integer.parseInt() funciona solo si el String representa un digito. * Usar break para salir tempranamente de un bucle (aún cuando la condicional se siga cumpliendo). | C:\Users\ingas\Desktop\Atencion2.png |

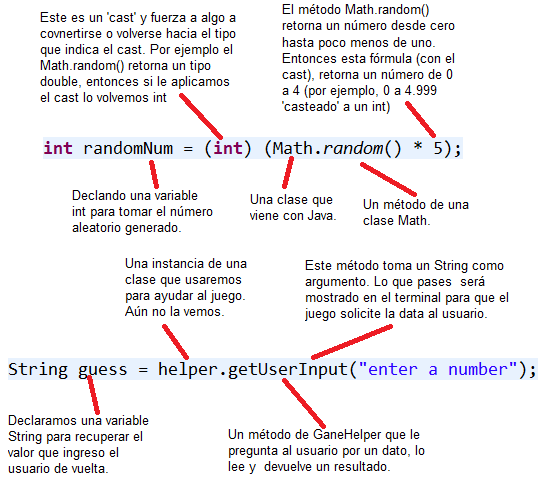
# El método main() del juego

Como lo hiciste con la clase SimpleDotCom, piensa en las partes que este código podria necesitar para mejorar.



# Random y getUserInput

Hay dos cositas que tenemos que explicar en este momento. Unos detallitos acerca de la clase GameHelper.



# Una última clase: GameHelper

Clase para dot com: hecho.

Clase para juego: hecho.

Solo queda la clase helper - la que viene con el método getUserinput(). El código que recuperará la información de ingreso del usuario es lo que veremos. Aunque no tan a fondo, eso se verá más adelante.

Solo copia el siguiente código junto con las otras clases:

**public** **class** GameHelper {

**public** String getUserlnput(String prompt){

String inputLine = **null**;

System.*out*.print(prompt +" ");

**try** {

BufferedReader is = **new** BufferedReader(**new** InputStreamReader(System.*in*));

inputLine = is.readLine();

**if** (inputLine. length () == 0) **return** **null**;

} **catch** (IOException e) {

System.*out*.println("IOException: " + e);

}

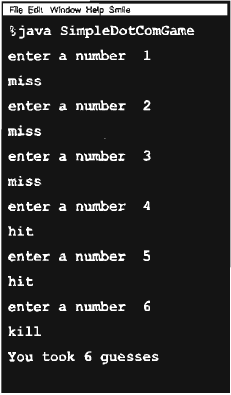
**return** inputLine;

}

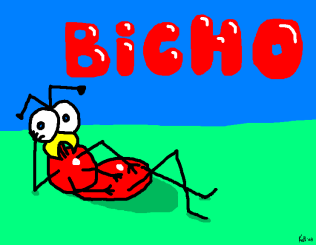
}

# A jugar!

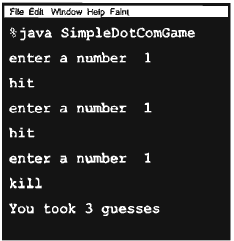
Aquí los resultado cuando ingresamos 1, 2, 3, 4, 5, 6:



# ¡¡¡¡¿¿¿¿Un bicho????!!!!



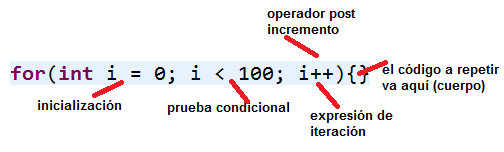
Aquí lo que pasa cuando ingresamos 1, 1, 1.



# Más acerca de los bucles

Ya hemos cubierto la mayor parte del juego en el código (versión mejorada más adelante). Y no queremos interrumpir tu trabajo con detallitos. Asi, que mejor vamos a revisar los bucles….

# for



Lo cual significa “repetir 100 veces”.

El compilador lo ve:

* Crea la variable i y asignale 0.
* Rpetir mientras i sea menor a 100.
* Al final de cada iteración agregar 1 a i.

Parte 1: Inicialización

Usa esta parte para declarar e inicializar una variable y usarla dentro del cuerpo del bucle. Generalmente lo tomarás como un contador. En estas versiones puedes inicializar más de una variable aquí, pero lo veremos después.

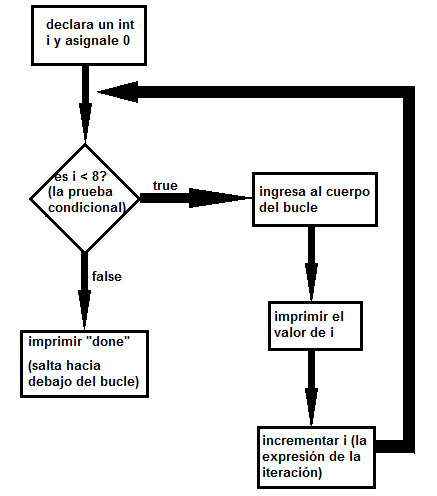
Parte 2: Pruebas condicional

Aquí es donde va la prueba condicional. Sea lo que sea que vaya, debe resolverse con un valor booleano (true o false). Puedes hacer la prueba directamente o invocar una función que te devuelva un boolean.

Parte 3: Expresión de iteración

En esta parte, colocar una o más cosas que quieras que sucedan despues de terminar cada iteración.

# Viaje a través de los bucles



**for**(**int** i = 0; i < 8; i++){

System.*out*.println(i);

}

System.*out*.println("done");

# Diferencias entre for y while

Un bucle while solo tiene una prueba condicional, no tiene una inicialización de variables o una expresión de iteración. Un bucle while es bueno cuando no sabes cuantas veces se va a iterar un bucle, entonces hasta que algún evento suceda y cambie el valor del la condición, se va a seguir iterando. Pero si sabes cuantas veces vas a ejecutar el bucle (por ejemplo, recorrer un array usando su longitud o 7 veces exactas), un bucle for es lo más ideal. Aquí el bucle anterior usando while:

**int** i = 0;

**while**(i < 8){

System.*out*.println(i);

i++;

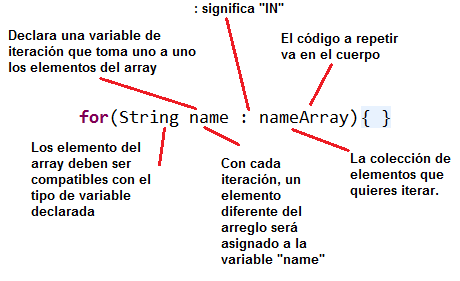
}

System.*out*.println("done");

|  |  |
| --- | --- |
| **Operador de incremento (++) y decremento (--)**  La forma corta de agregar o sustraer 1 de una variable es:  x++ para agregar que es lo mismo para x = x + 1;  Ambas formas significan lo mismo en este contexto:  “agregar 1 al valor actual de x” o “incrementa a x en 1”  Y x-- es lo mismo que x = x – 1;  Además tenemos que si usamos los operadores asi:  --x o ++x significa que antes de usarlo de decrementará o incrementará el valor d la variable x. Lo cual es diferente a lo antes mostrado, pues si se usa x++ o x—primero se usa el valor actual y luego de modifica el valor. Entonces, si tenemos:  **int** x = 0;  **int** z = ++x;  Significa que z tendrá el valor de 1. Pero si hacemos:  **int** x = 0;  **int** z = x++;  z tendrá valor de 0. | C:\Users\ingas\Desktop\Atencion2.png |

# Un for mejorado

Desde la versión 5 de Java (Tiger), se ha incorporado un segundo tipo de bucle for, al cual se ha llamado mejroa del for. Que hace más sencilla la iteración de los elemento de un array o cualquier colección.



Lo cual signfica: “Por cada elemento en nameArray, asigna el elemento a la variable “name”, y ejecuta el cuerpo del bucle.

El compilador lo ve:

* Crea una variable String llamada name y asignale nulo.
* Asigna el primer valor de nameArray a name.
* Ejecuta el cuerpo del bucle (el delimitado por las llaves).
* Asigna el siguiente valor de nameArray a name.
* Repetir mientras aún haya elementos en el array.

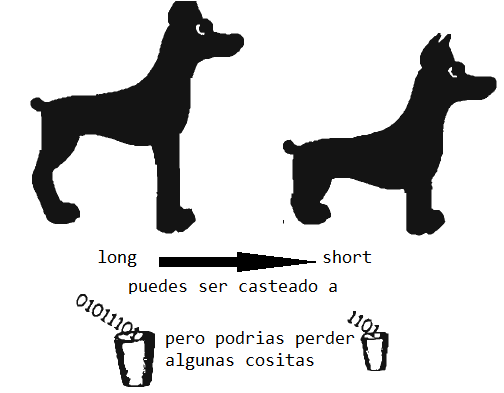
Parte 1: Declaración de variable de iteración

Usa esta parte para declarar e inicializar una variable que usarás dentro del cuerpo del bucle. Con cada iteración del bucle, esta variale tomará un elemento diferente desde la colección. El tipo de esta variable debe ser compatible con los elementos del array. Por ejemplo no puede declarar un int como variable de iteración para ser usado con un array de String[].

Parte 2: La colección

Esta debe ser una referencia a un array u otro tipo de colección. No te preocupes por los no-array aún, los veremos más adelante.

# Casting de primitivos



En temas anteriores, conversamos acerca del tamaño de los primitivos y como no podias “meter” valores grandes directamente en variables de menos capacidad:

**long** y = 42;

**int** x = y; //no funcionará

Un long es más grande que un int y el compilador no puede asegurarte que un long pueda alcanzar en un int. Para forzarlo tendriamos que usar el operador de cast:

**long** y = 42;

**int** x = (**int**)y;

Colocando el cast le dices al compilador que tome el valor de y para que coloque en x. Si el valor de x “desborda”, quedará un valor extraño pero entendible:

**long** y = 40002; //40002 excede el límite de de los 16 bits de un short

**int** x = (**int**)y; //x ahora es igual a -25534

Aún asi, el compilador te lo permitirá. Y más aún:

**float** f = 3.14f;

**int** x = (**int**)f; //x sera igual a 3

# Sé el compilador

Para seguir ejercitándonos haremos unos pequeños ejercicios donde nos convertiremos (momentáneamente) en el compilador de Java y determinaremos cual de las tres opciones será la salida correcta del siguiente programa:

|  |  |
| --- | --- |
| **public** **class** Output {  **public** **static** **void** main(String [] args) {  Output o = **new** Output();  o.go() ;  }  **private** **void** go() {  **int** y = 7;  **for**(**int** x = 1; x < 8; x++) {  y++;  **if** (x > 4) {  System.*out*.print(++y + " ");  }    **if** (y > 14) {  System.*out*.println(" x = " + x);  **break**;  }  }  }  } |  |
|  |  |
|  |  |
|  |  |