¿Qué aprendimos?

## Java JRE y JDK compile y ejecute primer programa

Cap1

En la clase introductoria ya hemos aprendido varios temas fundamentales sobre Java.

Hablamos sobre las principales características de Java como:

- Orientado a objetos.
- Parecido a C++.
- Muchas librerías y una gran comunidad.

Además de eso, aprendimos:

- La diferencia entre código fuente y Bytecode.
- Para ejecutar el Bytecode necesitamos tener la máquina virtual de Java.
- El Bytecode es independiente del sistema operativo.

Vimos también los principales componentes de la plataforma Java, que son:

- Java Virtual Machine (JVM).
- Lenguaje Java.
- Librerias Java (API).

En la próxima clase vamos a escribir nuestro primer código en Java. ¡Continuemos!

Cap2.

En esta clase escribiste tu primer código Java y aprendimos:

- Cuál es la diferencia entre JRE y JDK.

- Cómo compilar el código fuente de Java desde la línea de comandos (`javac`).
- Cómo ejecutar Bytecode en la línea de comando (`java`).
- Un programa Java debe escribirse dentro de una clase (`class`).
- Cada instrucción Java debe terminar con `;`.
- Para abrir y cerrar un bloque usaremos las llaves `{}`.
- Un programa Java tiene una entrada que es una función (método) `main`.
- Para imprimir algo en la consola, usamos la declaración `System.out.println ()`.

Cap3

Este capítulo presentó:

- El papel de un IDE y su diferencia con respecto a un editor.
- Cómo descargar el IDE de Eclipse.
- ¿Para qué sirve workspace?
- El concepto de perspectiva.
- Cómo crear un proyecto Java, incluidas las clases y cómo ejecutarlo.
- Cómo mostrar diferentes views.

Cap4

En esta clase comenzamos nuestro aprendizaje con variables y tipos primitivos de Java. Los tipos vistos con más detalle fueron int (entero) y double (decimal). Los cuales usamos para hacer operaciones aritméticas y también concatenar con texto.

Durante el capítulo hablamos de buenas prácticas al nombrar clases y también variables. Comenzamos las clases con mayúscula y nuestras funciones y variables con minúsculas. Hablaremos más sobre esto en el futuro.

Fue posible comprender un poco de conversión de tipos y cómo podemos pasar un valor de un tipo para una variable de otro. En algunos casos no necesitamos hacer nada, ya que el casting es implícito y en otros debemos dejar en claro al compilador que sabemos lo que estamos haciendo, mostrando entre paréntesis el tipo que queremos usar.

Esto concluye esta lección. Siguiente paso: Caracteres. ¡Te espero allí!

Cap5

Este capítulo presentó:

- El concepto y cómo declarar char y String.
- Cómo concatenar Strings.
- Atajo para crear método main.
- Las variables almacenan valores y no referencias.

Cap 6

En esta clase, aprendimos:

- Cómo usar el `if`.
- Cómo usar las operaciones lógicas AND (`&&`) y OR (`||`).
- Trabajar con el alcance de las variables.

También hemos visto algunos atajos en Eclipse:

- `main Ctrl + espacio`, para generar el método `main`.
- `ctrl + shift + f`, para formatear el código fuente.
- `sysout + ctrl + espacio`, para generar la instrucción `System.out.println ()`.

Cap 7

En este capítulo aprendimos:

- La sintaxis de `while` y `for`.
- El operador `+=`.
- El operador `++`.
- Bucles anidados.
- La funcionalidad `Break`.

## Java Orientada a Objetos

Cap1

En esta clase conocimos el paradigma procedural, que se utilizó como práctica de programación antes de la introducción de lenguajes

orientados a objetos. La necesidad de validar el numeroIdentidad en un formulario se utilizó como ejemplo para discutir los principales problemas que pueden aparecer en este paradigma.

En particular, a medida que otros formularios y desarrolladores necesitan la misma validación de numeroIdentidad, no fue fácil ver que ya había procedimientos y funciones que hicieron este trabajo, ya que los datos y las funciones no tenían un vínculo tan fuerte. Esto podría dar lugar a otra nueva función o fragmento de código con una responsabilidad similar.

Además, conocemos la idea central del paradigma orientado a objetos, que es crear unidades de código que combinen los datos asociados con cierta información con las funcionalidades aplicadas a esos datos (por ejemplo, numeroIdentidad + validación). Son los atributos y métodos.

Cap 2

En esta clase aprendimos sobre atributos y creación de objetos.

Vimos:

¿Qué es una clase? Cómo crear una clase Cómo crear un objeto o instancia de una clase ¿Qué son los atributos? Cómo definir y ver los valores de los atributos ¿Cómo funciona la asignación de una referencia a una variable?

¡Veremos más sobre las clases en la próxima clase! Ahora no solo sus características sino también su comportamiento. ¡Solo comencemos! ¡Te espero allí!

Cap3

En esta clase hablamos sobre el comportamiento que son los métodos.

Vimos:

cómo definir métodos con parámetros y retorno cómo devolver algo usando la palabra clave `return` cómo usar la referencia `this`para acceder a un atributo que podemos pasar una referencia como parámetro de método los métodos se invocan desde la referencia utilizando el operador `.`
Si aún tiene dudas sobre las referencias, `this` y el uso de métodos, tenga la seguridad de que en las próximas clases (y cursos) revisaremos los conceptos y practicaremos mucho más. ¿Continuamos?

Cap4

Aprendimos en este capítulo:

• Darse cuenta de la relación entre clases a través de la composición;

• Ventajas de aislar información repetida en otra clase;

• Detalles de la composición;

• `NullPointerException` para atributos no inicializados y su cuidado.

Cap5

Aprendimos en esta clase:
Atributos privados, restringiendo el acceso a los atributos. Encapsulación de código Métodos de lectura de atributos, los ***getters*** Métodos para modificar atributos, los ***setters*** Getters y Setters de referencia

Cap 6

Aprendimos en esta clase:
Constructor de clases, que permite recibir argumentos e inicializar atributos desde la creación de un objeto. Con esto, la inicialización de los atributos recibidos en el constructor se vuelve obligatoria. Atributos de clase, atributos estáticos. Métodos de clase, métodos estáticos. Ausencia de referencia, del `this`, dentro de los métodos estáticos.

## Java Polimorfismo: Entendiendo herencia e interfaces

## Cap1

En esta aula comenzamos a hablar de herencia y aprendimos:

- Cuáles son los problemas que la herencia resuelve.
- Cómo usar la herencia en Java a través de la palabra llave *extends*
- Al heredar la clase hija gana todas las características (atributos) y todas las funcionalidades (métodos) de la clase madre.
- Conocimos el primer beneficio de la herencia: *La reutilización de código*.

En la próxima clase veremos más detalles sobre la herencia como la palabra llave *super*, *protected* y la sobreescritura de métodos.

Cap 2

En esta clase entramos más a fondo en la herencia. Aprendimos:

- que la clase madre es llamada de super o base class.
- que la clase hija también es llamada de sub class.
- como aumentar la visibilidad de un miembro (atributo, método) a través de *protected*.
- cómo acceder o llamar un miembro (atributo, método) a través de *super*.
- cómo redefinir un método a través de la sobreescritura. En la próxima clase veremos un nuevo beneficio de la herencia, el **Polimorfismo**. ¡Aguarda!

Cap3

En esta clase aprendimos que:

- los objetos no cambian de tipo;
- la referencia puede cambiar, y ahí es donde entra el polimorfismo;
- el polimorfismo permite utilizar referencias más genéricas para comunicarse con un objeto;
- el uso de referencias más genéricas permite desacoplar sistemas.

En el siguiente vídeo, hablaremos sobre cómo se comportan los constructores en la herencia.

Cap 4

En esta clase, vimos:

- Conceptos de herencia, constructores y polimorfismo
- Usando la anotación @Override
- Los constructores no se heredan
- Se puede llamar a un constructor de clase madre mediante super()

¡En el siguiente vídeo hablaremos sobre cómo se comportan las clases y métodos abstractos! Espere :)

Cap 5

En esta clase aprendimos:

- Qué son las clases abstractas
- Para qué sirven las clases abstractas
- Qué son los métodos abstractos
- Para qué sirven los métodos abstractos

¡En la siguiente clase veremos sobre el uso de **interfaces**!

Cap 6

En esta clase aprendimos que:

- No hay herencia múltiple en Java.
- Conceptos de interfaz.
- Diferencias entre clases abstractas e interfaces.
- Las interfaces son una alternativa a la herencia con respecto al polimorfismo

¡En el próximo capítulo practicaremos un poco más sobre herencia e interfaces!

Cap 7

En esta clase aprendemos:

- Más en profundidad sobre el uso de interfaces
- Trabajamos más profundamente con la herencia
- Vimos otras aplicaciones de herencia e interfaz

## Java Excepciones: Aprende a crear, lanzar y controlar excepciones

Cap1

En esta clase aprendimos:

- Qué es, para qué sirve y cómo funciona la pila de ejecución.
- Qué es la depuración (*debug*) y para qué sirve.
- Cómo utilizar Eclipse y su perspectiva de debug.
- Cómo cambiar entre perspectivas de Eclipse.

Hagas clic aquí para acceder a la documentación oficial y obtener más información sobre la clase Exception.

Cap 2

En esta clase aprendemos:

- Qué son las excepciones, para qué sirven y por qué se utilizarlas.
- Cómo analizar el seguimiento de excepciones o *stacktrace*.
- Manejar excepciones con bloques *try-catch*.
- Manejar una excepción lanzada dentro del bloque *catch*.
- Manejar múltiples excepciones con más de un bloque *catch*o usando *Multi-Catch* usando el *pipe* (|).

Cap 3

En esta clase aprendemos:

- Cómo lanzar excepciones.
- Cómo asignar un mensaje a la excepción.

Cap 4

Si hiciste el ejercicio ¿será Miguel comprendió la clase ?, recordará lo que aprendimos. Para solucionarlo aún más, enumeramos los temas de esta clase:

- Existe una gran jerarquía de clases que representan excepciones. Por ejemplo, *ArithmeticException* es hija de *RuntimeException*, que hereda de *Exception*, que a su vez es hija de la clase de excepciones más ancestral, *Throwable*. Conocer bien esta jerarquía significa saber cómo usar las excepciones en su aplicación.
- *Throwable* es la clase que debe extenderse para poder lanzar un objeto en la pila (usando la palabra reservada *throw*)
- Es en la clase *Throwable* donde tenemos casi todo el código relacionado con las excepciones, incluyendo *getMessage()* e *printStackTrace ()*. El resto de la jerarquía tiene solo algunas sobrecargas de constructores para comunicar mensajes específicos.
- La jerarquía que comenzó con la clase *Throwable* se divide en excepciones y errores. Las excepciones se utilizan en los códigos de aplicación. Los errores son utilizados exclusivamente por la máquina virtual.
- Las clases que heredan de *Error* se utilizan para informar errores en la máquina virtual. Los desarrolladores de aplicaciones no deben crear errores que hereden de *Error*.
- *StackOverflowError* es un error de máquina virtual para informar que la pila de ejecución no tiene más memoria.
- Las excepciones se dividen en dos categorías amplias: las que el compilador comprueba obligatoriamente y las que no.
- Los primeros se denominan *checked* y se crean por pertenecer a una jerarquía que no pasa *RuntimeException*.
- Los segundos están *unchecked* y se crean como descendientes de *RuntimeException*.

Cap 5

En esta clase, aprendimos y practicamos:

- cómo crear un bloque *catch* genérico utilizando la clase *Exception*;
- cómo crear una nueva excepción *SaldoInsuficienteException*;
- cómo transformar la excepción en checked o unchecked.

Cap 6

En esta clase aprendimos:

- que existe un bloque *finally*, útil para cerrar recursos (como conexión);
- cuando hay un bloque *finally*, el bloque de *catch* es opcional;
- que el bloque * finally * se ejecuta siempre, sin o con excepción;
- cómo utilizar *try-with-resources*.
-

## Java y java.lang : Programa la clase Object y String

Cap 1

¿Qué aprendimos?

- Los packages sirven para organizar nuestro código.
- Los packages son parte del FQN (*Full Qualified Name*) de la clase.
- El nombre completo de la clase (FQN) consta de: PACKAGE.NOMBRE_SIMPLE_DE_LA_CLASE
- La definición de package debe ser la primera declaración en el código fuente
- Para facilitar el uso de clases de otros packages, podemos importarlas
- Los *import* son justo después de la declaración del package

- La nomenclatura padrón es usar el nombre de dominio en la web al revés con el nombre del proyecto, por ejemplo:

```
br.com.caelum.geradornotas

br.com.alura.gnarus

br.gov.rj.notas

de.adidas.lagerCOPIA EL CÓDIGO
```

Una vez organizadas nuestras clases, podemos revisar los modificadores de visibilidad que dependen de los paquetes. ¿Continuamos?

Cap 2

¿Qué aprendimos?

En esta clase volvimos a hablar sobre visibilidad y aprendimos:

- Hay 3 palabras clave relacionadas con la visibilidad: *private, protected, public*
- Hay 4 niveles de visibilidad (de menor a mayor):
    - `private` (visible solo en clase)
    - `<<package private>>` (visible en la clase y en cualquier otro miembro del mismo paquete, que puede ser llamado de default)
    - `protected` (visible en la clase y en cualquier otro miembro del mismo paquete y para cualquier hijo)
    - `public` (visible en cualquier paquete)
- Los modificadores pueden ser usados en la definición de la clase, atributo, constructor y método.

Cap 3

¿Qué aprendimos?

En esta sección más ligera vimos y aprendimos:

- Qué comentarios y tags (anotaciones) usar para definir el javadoc
- Cómo generar javadoc en Eclipse
- Que javadoc es una documentación para desarrolladores
- Que las clases estándar de Java también usan javadoc
- Cómo crear nuestra propia librería a través de JAR (J*ava *ARchive)
- Cómo importar librerías al nuevo proyecto
- Cómo crear un JAR ejecutable

En la siguiente clase conoceremos el paquete java.lang.

Cap 4

¿Qué aprendimos?

En esta clase aprendimos y sabemos:

- El package java.lang es el único paquete que no necesita ser importado
- Tiene clases fundamentales que cualquier aplicación necesita, como la clase String y System
- Los objetos de la clase String son inmutables y usamos una sintaxis literal para crear (objeto literal)
- Cualquier método para cambiar la clase String devuelve un nuevo String que representa el cambio
- La clase String es una CharSequence
- Si necesitamos concatenar muchos String debemos usar la clase StringBuilder
- Vimos varios métodos de la clase String como trim, charAt, contains, isEmpty, length, indexOf, replace

En la siguiente clase veremos otra clase fundamental: java.lang.Object

# Java y java útil: Colecciones, wrappers y lamda

En esta clase sobre *arrays* aprendimos:

- Un array es una estructura de datos y se usa para almacenar elementos (valores primitivos o referencias)
- Los arrays usan corchetes ([]) sintácticamente
- ¡Los arrays tienen un tamaño fijo!
- ¡Un array también es un objeto!
- Los arrays son *zero-based*(el primer elemento se encuentra en la posición 0)
- Un array siempre se inicializa con los valores padron.
- Al acceder a una posición no válida recibimos la excepción ArrayIndexOutOfBoundException
- Las matrices tienen un atributo *length* para conocer el tamaño
- La forma literal de crear un array, utilizando llaves {}.

En el próximo capítulo hablaremos un poco más sobre arrays (de tipo Object) y veremos cómo funciona este parámetro del método *main*.

Cap 2.

En esta clase aprendimos:

- Un array de tipo *Object* puede contener cualquier tipo de referencia.

- Cuando convertimos una referencia genérica a una referencia más específica, necesitamos usar un *type cast*.
- El *cast* solo compila cuando es posible, aún así puede fallar al ejecutarse.
- Cuando falla el *type cast*, podemos recibir un *ClassCastException*.
- Para recibir valores al llamar al programa Java en la línea de comando, podemos usar la matriz *String[]* en el método *main*.

¡En la próxima clase comenzaremos a hablar de listas! Espere :)

Cap3.

En esta clase comenzamos a hablar sobre la lista y llegamos a conocer la clase *java.util.ArrayList*. Aprendimos:

- Que la clase *java.util.ArrayList* encapsula el uso de array y ofrece varios métodos de más alto nivel.
- Que una lista guarda referencias.
- Cómo usar los métodos *size, get, remove*.
- Cómo usar *foreach* para iterar a través de *ArrayList*.
- Que los generics parametrizan clases
- Que en el caso de *ArrayList* podemos definir el tipo de los elementos mediante generics.

Este es solo el comienzo de este poderoso paquete *java.util*. ¡En la próxima clase bucearemos más!
DISCUTIR EN EL FORO

Cap 4.

En esta clase aprendemos:

- Cómo implementar el método *equals* para definir la igualdad.
- Que el método *equals* es usado por las listas.

- Que hay más de una lista, *java.util.LinkedList*
- La diferencia entre *ArrayList* y *LinkedList*
- La interfaz *java.util.List* que define los métodos de la lista

En el próximo capítulo veremos otra implementación de la interfaz *List*

Cap 5

En esta clase vimos:

- *java.util.Vector*, que es un *ArrayList* threadsafe.
- La interfaz *java.util.Collection* que es la interfaz de todas las colecciones.
- Las listas son secuencias que aceptan elementos duplicados.
- Los conjuntos (*java.util.Set*) también son colecciones, pero no aceptan duplicados ni listas.

¡En la siguiente clase resolveremos el problema de guardar primitivos en las listas!

Cap 6

En esta clase nos enfocamos en las clases de*WRAPPERS* y aprendimos que.

- Para cada primitivo hay una clase llamada *Wrapper*.
- Para almacenar un primitivo en una colección, necesita crear un objeto que envuelva el valor.
- La creación del objeto *Wrapper* se llama *autoboxing*.
- La extracción del valor primitivo del objeto *Wrapper* se llama *unboxing*.
- El *autoboxing* y *unboxing* ocurren automáticamente.
- Las clases *wrapper* tienen varios métodos auxiliares, por ejemplo para el *parsing*.
- Todas las clases *wrappers* que representan un valor numérico tienen la clase *java.lang.Number* como madre.

¡En la próxima clase aprenderemos a ordenar las listas!

Cap7.

En esta clase fundamental e importante aprendimos que:

- Para ordenar una lista necesita definir un criterio de ordenación
- Hay dos formas de definir este criterio
- A través de la interfaz del *comparador*
- A través de la interfaz *Comparable* (*orden natural*)
- El algoritmo de ordenación ya se ha implementado
- En la lista en el método de *sort*
- En la clase *Collections* por el método *sort*
- La clase *Collections* es una fachada con varios métodos auxiliares para trabajar con colecciones, principalmente listas

Respire hondo, ya que estamos casi al final de este curso, sin embargo todavía tenemos que aprender (¡por fin!) las famosas expresiones *lambda*. ¿Estás listo para continuar?

Cap8.