Práctica 4. Arreglos y Herencia en Java

Introducción:
En Java, se pueden crear matrices de tipos de datos primitivos como int, char, boolean y demás, pero también puede crear matrices de clases que ya están en el marco de Java o incluso hechas por usted mismo.
Comenzaremos con los conceptos básicos de las matrices que utilizan tipos de datos primitivos, pero puede pasar directamente a la explicación de la matriz de solo objeto si ya comprende el tema anterior.

Matriz de tipos de datos primitivos:
Cuando desee utilizar una matriz en Java, primero debe declarar una variable capaz de contener una. Para hacerlo, escribe: `int[] integerArray;`
En este ejemplo de declaración de matriz:

- `int` es el tipo de elementos que desea que contenga su matriz.
- los corchetes ([ ]) le dicen a Java que esta variable se usará para contener una matriz.
- `integerArray` es el nombre de la variable que se usa para mantener y luego acceder a nuestra matriz elementos.

Como puede ver aquí, todavía no le hemos dicho a Java cuántos elementos queremos manejar en nuestra matriz, y eso se debe a que la asignación de memoria no se especifica en la declaración de la matriz, se indica después de eso, en la inicialización de la matriz.
Hay muchas formas de inicializar una matriz en Java, por ejemplo:

```
int[] integerArray = new int [10];
int[] integerArray = new int []{1,2,3,4,5,6,7,8,9,10};
int[] integerArray = {1,2,3,4,5,6,7,8,9,10};
```

Ejercicio 1. Imprimir en pantalla el elemento que se encuentra en la posisición 5 para la inicialización del siguiente arreglo: `int[] integerArray1 = {1,20,3,44,5,60,7,88,9,100};`

Cuando inicializa una matriz con:

- La instrucción `new type[number]`:
  Asigna memoria para el número de elementos indicados dentro de los corchetes, correspondiente al tamaño del tipo especificado. Todos estos elementos se inicializan en "0", "falso" o "nulo", según el tipo de datos de la matriz.
- Una instrucción combinada `new type[] {elements}` o simplemente `{elements}`:
  Inicializa una matriz del mismo tamaño que el número de elementos presentes dentro de las llaves, colocándolos todos en su lugar correspondiente dentro de la matriz. Por ejmplo, `IntegerArray [0]` sería "1", `integerArray [1]` sería " 2 "y así sucesivamente). Incluso si la matriz se inicializó con elementos preestablecidos, aún puede cambiar el valor de cualquiera de ellos más adelante.

También es posible crear una instancia de una matriz con un tamaño específico o una matriz de elementos ya establecida, pero no puede hacer ambas cosas al mismo tiempo.

Tanto el primer como el segundo ejemplo de inicialización se pueden aplicar por separado de la declaración de variable, pero el tercero le dará un error de compilación y detendrá la ejecución del programa.

Ejercicio 2. Inicializar un arreglo de tamaño 5 usando 2 metodos diferentes. El primer método inicializará el arreglo con puros ceros y el segundo con números aleatorios.

Arreglos de objetos:
Java permite la creación de matrices para clases de la misma manera que lo haría para tipos de datos primitivos, por ejemplo: `MyClass[] myClassArray;`

De igual forma es posible inicializar los arreglos de manera similar a los arreglos de datos primitivos, por ejemplo:

```
MyClass[] myClassArray = new MyClass[20];
MyClass[] myClassArray = new MyClass[]{new MyClass(),new MyClass()};
MyClass[] myClassArray = {new MyClass(),variableWithAMyClassObject, new MyClass()};
```

Incluso con tantas diferencias visuales entre las tres inicializaciones, es más simple de lo que parece. En primer lugar, Java permite nuevas líneas entre los elementos enumerados con cualquier tipo de separación entre ellos, así que esto:
`MyClass[] myClassArray = new MyClass[]{new MyClass(),new MyClass()};`
Es exactamente lo mismo a esto: `MyClass[] myClassArray = new MyClass[20];`

En donde:

- `MyClass` es el tipo de objetos que nuestra matriz contendrá como elementos.
- Los corchetes ([ ]) le dicen a java que esta variable se usará para manejar una matriz.
- `myClassArray` es nuestro nombre de variable. Este es el final de la declaración de la matriz.
- La inicialización de la matriz, `new class [numberOfElements]` reserva memoria para contener la misma cantidad de objetos de la clase indicada que el número entre corchetes. En este caso, la matriz podrá contener hasta 20 objetos de `MyClass`.

Este método no inicializa objetos dentro de la matriz, solo reserva la memoria para poder contenerlos. Lo que significa que, si usaras un método con cualquiera de los elementos de la matriz sin inicializarlo primero manualmente, obtendrías una `NullPointerException`.

En el siguiente ejemplo:

```
MyClass[] myClassArray = new MyClass[]{new MyClass(),new MyClass()};
```

Lo que realmente está sucediendo es que se está creando una matriz de dos objetos `MyClass` y los instanciamos al mismo tiempo. Como no usamos identificadores para los elementos, los objetos dentro de la matriz son objetos anónimos. La única forma de acceder a ellos será mediante el uso de un índice en la matriz.

Ejercicio 3. Crear 3 objetos usando identificadores y agregarlos a un arreglo, después crear otro arreglo con 3 objetos instanciados en la declaración.

Por último:

```
MyClass[] myClassArray = {new MyClass(),variableWithAMyClassObject, new MyClass()};
```

Aquí se tiene una matriz de objetos inicializada con un conjunto de elementos preestablecido. Este conjunto tiene elementos anónimos y variables de objeto por igual. Cuando se usa una variable para llenar un elemento de una matriz, la variable solo devuelve una copia de sí misma, lo que hace que la variable y el elemento dentro de la matriz sean dos objetos diferentes. Esto también se aplica a las matrices de tipos de datos primitivos.

Se puede acceder a los objetos dentro de una matriz de la misma manera que accedería a un tipo de datos primitivo:

```
myClassArray[1] //sería una copia de variableWithAMyClassObject del ejemplo anterior.
```

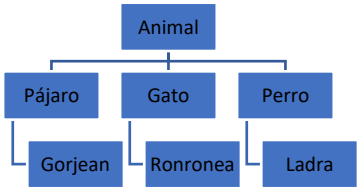
Para acceder a los métodos y atributos de un objeto en una matriz es de la misma manera que lo hacemos con las variables de objeto, sólo recuerde usar el índice correcto del elemento que desea utilizar dentro de la matriz.

```
myClassArray[1].setName("Jose J.");
System.out.println(myClassArray[1].getName());
```

Herencia:

En Programación Orientada a Objetos, definimos las clases como nuestros planos para un objeto que se puede encontrar en el mundo real. Pero, ¿qué sucede cuando tenemos objetos similares con diferentes propiedades? Claro, podemos agregar más atributos y algunos métodos más a nuestra clase ya definida para acomodarlos, pero luego tendríamos una clase muy abarrotada que no utiliza todo el código que hemos escrito en ella la mayor parte del tiempo. También pierde su identidad como modelo para un objeto definido, haciéndolo menos adecuado para su propósito original.

Imaginemos entonces que tenemos una clase llamada "Animal", que podría mantener juntos a algunos de nuestros animales favoritos, como perros, gatos e incluso pájaros. Todos podríamos estar de acuerdo en que, como animales, todos comen y duermen, pero solo un perro ladra, solo un gato ronronea y solo un pájaro gorjea. Ahora bien, tener un perro que ronronea, un gato que gorjea o un pájaro que pueda ladrar sería una vista increíble, pero no sería realista.



Para solucionar este problema, POO tiene un concepto llamado “herencia” que explicaremos a lo largo de esta guía.

En el ejemplo anterior, podemos estar de acuerdo en que el perro, el gato y el pájaro son todos animales y, como tales, todos podrían agruparse en la clase “Animal”.

"Animal" es solo el concepto general para ellos y no toma en cuenta cada una de sus propias características que los hacen diferentes entre sí. Bueno, sabemos que son diferentes pero, ¿eso significa que necesitamos escribir el código de nuestra clase “Animal” en cada una de nuestras clases para nuestros diferentes animales? La respuesta es no, y este es el motivo.

La “herencia” se puede implementar para reutilizar métodos y atributos en clases que derivan de una más general. Comencemos por hacer esa clase llamada Animal, ¿de acuerdo? Podemos ser un poco más elaborados esta vez y agregarle un nombre y un atributo de edad.

```
public class Animal{
	String name;
	int age;

	public Animal(String name, int age){
		this.name = name;
		this.age = age;
	}

	Public void eat(){
		System.out.println(name+ " is eating");
	}

	Public void sleep(){
		System.out.println(name+ " is sleeping");
	}

}
```

Es posible crear una clase diferente que contenga los mismos atributos y métodos que otra menos específica, al tiempo que agrega más funcionalidad por sí misma, sin tener que repetir todo el código escrito anteriormente. Podemos lograr esto usando la palabra clave `extends` delante de nuestro nuevo nombre de clase, seguida del nombre de la clase de la que heredamos el código. En este caso, hagamos una clase de perro, por ejemplo:

```
public class Dog extends Animal{}
```

Esta línea le dice a Java que la clase `Dog` contendrá todos los atributos y métodos de la clase `Animal` y, como se dijo antes, aún podrá agregar los suyos. Cuando una clase hereda de otra clase, la primera en este caso se conoce como clase secundaria o subclase, y la clase de la que hereda se conoce como clase principal o superclase.
Con este proceso, puede extender cualquier clase a muchas otras, pero una clase solo puede heredar directamente de una clase a la vez. Con la herencia, puede tener una gran cadena de clases que heredan de otra, dejando la última clase de la cadena con todos los atributos y métodos de todas las clases anteriores.

Por ejemplo, digamos que tenemos una clase "Beagle" que, como raza de perro, heredaría de "Perro" que ya conocemos, hereda de "Animal". Esto hace que "Animal" sea una superclase indirecta de "Beagle", incluso aunque “Perro” está en el medio de la jerarquía, dejando a “Beagle” con todos los métodos y atributos que se encuentran en “Perro” y “Animal” al mismo tiempo.

Ejercicio 4. Busque la clase Object en la documentación de Java. Observe los métodos disponibles y vea si puede usarlos con un objeto Perro o Animal.

```
public class Dog Extends Animal{

	public Dog (String name, int age){
	}

	public void bark(){
		system.out.println(name + ": woof woof");
	}
}
```

Me gustaría que notaras dos cosas antes de continuar, y eso es: una, que no hemos dicho nada sobre las clases que heredan los constructores, y dos, que tenemos un constructor específico para la clase "Perro" con una nueva palabra clave llamada "súper".

Las subclases heredan atributos y métodos, pero no constructores, y eso se debe en parte a que son objetos diferentes en su núcleo y pueden necesitar diferentes procedimientos de inicio en sus constructores. En la definición de subclase, debe haber un constructor que reciba la cantidad apropiada de parámetros para inicializar tanto la clase del padre como los atributos de la clase del hijo. Podemos usar el constructor de nuestra "superclase" para inicializar los atributos que heredamos de ellos usando su propio constructor cuando llamamos a la palabra clave "super" dentro de nuestra subclase.
Esto indica a Java que "envíe" los dos primeros parámetros al constructor de la superclase, porque ahí es donde se definen el nombre y la edad. De esta manera podemos evitar tener que volver a escribir todas las asignaciones para los valores que ya existen en la superclase, y solo enfocarnos en los nuevos valores definidos en la subclase, por ejemplo:

```
public class Cat extends Animal {
	public Cat (String name, int age){
		super (name, age);
	}

	public void purr(){
		System.out.println(name+"is purring");
	}
}

public class Bird extends Animal {
	public Bird (String name, int age){
		super (name, age);
	}

	public void chirp(){
		System.out.println(name+"is chirping");
	}
}
```

Ejercicio 5. Crear otra subclase representando un animal diferente al perro.

Práctica:

1. Crear una clase que implemente un personaje del videojuego “Angry Birds”. Considere qué tipo de atributos y métodos debería tener.
2. Crear una instancia de 3 "pájaros" y organícelos en un arreglo.
3. Piense en las características generales que tendría un personaje de pájaro e implemente una clase padre llamada Pájaro.
4. Implementar subclases para los diferentes personajes considerando sus ataques. Los métodos pueden simplemente imprimir un mensaje en la consola.
5. Haga un programa para probar sus clases. Tenga en cuenta cómo las subclases pueden usar tanto sus propios métodos como los métodos de su superclase.

Actividades:

- Hacer un reporte con portada de hoja completa que incluya el código fuente, el link de su código en GitHub y una captura de pantalla de la práctica funcionando.
- Subir a blackboard código fuente (archivos java) y reporte como archivos separados, no se aceptarán archivos en carpetas comprimidas.