



Introducción a Orientación a Objetos en Java

- Originalmente hecha para una clase del 24 de Abril de 2018.
- Segunda Revisión: 20 de Febrero del 2020. Se han arreglado errores y añadido un prólogo con arrays y demás.
- Advertencia: En muchas partes del código se usan nombres de variables con '_' y minúsculas. Esto en Java va en contra de la convención. En Java se sigue el formato 'camelCase'

Hecho por:

- <https://github.com/M-T3K>
- <https://github.com/RESKOM326>



Antes de nada...

- Las 3 herramientas clave antes de programar son: la cabeza, un lápiz y un papel.
- No tengais miedo de usarlas, de verdad os hará las cosas más fáciles.
- Dibujad las cosas y usad estos dibujos para guiaros.
- He mencionado ya que useis un lápiz y un papel antes de programar nada?
- En serio, hacedlo. Hay que pensar antes de programar, y el papel y lapiz (o cualquier alternative) ofrece una manera rápida de hacerlo.
- El código de esta presentación está alojado en [github](#).



Prólogo: Arrays y Punteros

- Es importante tener una buena base para entender el resto de cosas.
- Arrays: Colección de elementos identificados por un Índice. Se almacena de forma que su posición en memoria pueda calcularse a partir de su índice.
- Puntero: Dato que contiene una dirección de memoria, que puede ser una variable, un objeto, una función, etc...
- Dependiendo del lenguaje de programación, la relación puntero-array puede ser:

Arrays en C:

- La longitud **n** se especifica durante la creación. No hay forma **simple** de obtenerla más tarde. Hay que calcularla.
- Responsabilidad del programador saber cuándo termina el array.

Ventaja: Ocupa poco en memoria.

*Almacenado en el Stack: tiempo de acceso más rápido.

Arrays en Java (y otros lenguajes OOP):

- Estructura de datos que contiene un **int length** y un puntero a memoria.
- El programador puede acceder a la longitud en cualquier momento.

Ventaja: Fácil de Usar.

Almacenado en Heap. Tiene mayor tiempo de acceso.

<https://i.stack.imgur.com/7uiSD.jpg>

Arrays en C

- En esta imagen se ve una matriz de 3x2
- Como véis, **arr** tiene un valor de dirección. Es decir, es un puntero.
- Podemos acceder a sus elementos mediante el operador de indexado [].
- Todos sus elementos son arrays de 1 dimensión. Como veis, también almacenan una dirección: son punteros.
- Estas direcciones (**addresses**) son las posiciones en memoria en las que se puede encontrar el array en cuestión.

Nota: Al hacer la operación de indexado es cuando se aplica la fórmula para calcular la posición a partir del índice. Generalmente es similar a:

$$dir = base + i \cdot tamaño$$

Donde *dir* es la dirección del element que queremos, *base* es la posición base del array (generalmente arr[0]), *i* es el índice del elemento que buscamos, y *tamaño* se refiere a lo que ocupa en memoria el tipo de elemento que compone el array (generalmente 4 bytes).

| | Address | Data |
|-----------|---------|---------|
| | . | . |
| | . | . |
| | . | . |
| arr | 1245028 | 1245032 |
| | 1245029 | |
| | 1245030 | |
| | 1245031 | |
| arr[0] | 1245032 | 1245039 |
| | 1245033 | |
| arr[1] | 1245034 | 1245045 |
| | 1245035 | |
| arr[2] | 1245036 | 1245051 |
| | 1245037 | |
| | 1245038 | |
| arr[0][0] | 1245039 | 10 |
| | 1245040 | |
| arr[0][1] | 1245041 | 20 |
| | 1245042 | |
| arr[0][2] | 1245043 | 30 |
| | 1245044 | |
| arr[1][0] | 1245045 | 40 |
| | 1245046 | |
| arr[1][1] | 1245047 | 50 |
| | 1245048 | |
| arr[1][2] | 1245049 | 60 |
| | 1245050 | |
| arr[2][0] | 1245051 | 70 |
| | 1245052 | |
| arr[2][1] | 1245053 | 80 |
| | 1245054 | |
| arr[2][2] | 1245055 | 90 |
| | 1245056 | |
| | . | . |
| | . | . |
| | . | . |

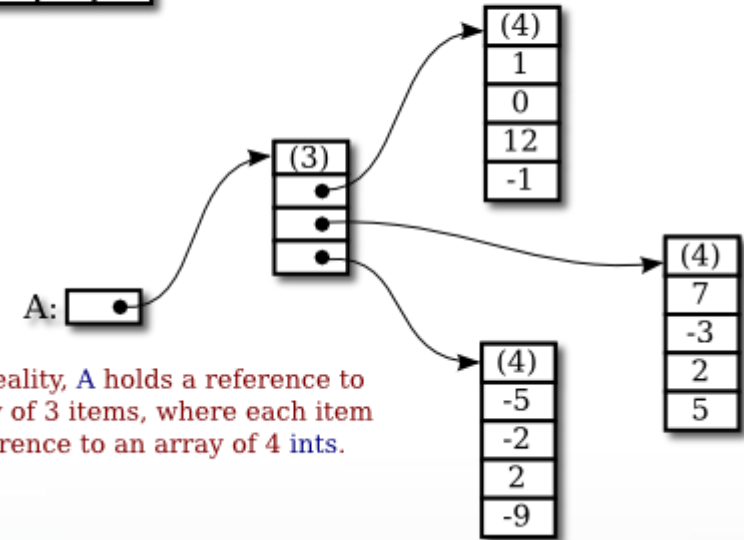
Arrays en OOP

- En esta imagen se ve una matriz de 4x3
- Como véis, **arr** tiene un valor de dirección. El primer elemento es **length**.
- Podemos acceder a sus elementos mediante el operador de indexado `[]`.
- Todos sus elementos son arrays de 1 dimensión. Estos, a su vez, también contienen **length** como su primer elemento, y luego los valores del array como tal.
- Como se puede apreciar, este modelo hace que se requiera más memoria para cada array, pero también hace que sea mucho más fácil usarlos. Hoy día, esta diferencia es nula, pero es fundamental saber como funcionan las cosas a bajo nivel.

A:

| | | | |
|----|----|----|----|
| 1 | 0 | 12 | -1 |
| 7 | -3 | 2 | 5 |
| -5 | -2 | 2 | -9 |

If you create an array `A = new int[3][4]`, you should think of it as a "matrix" with 3 rows and 4 columns.

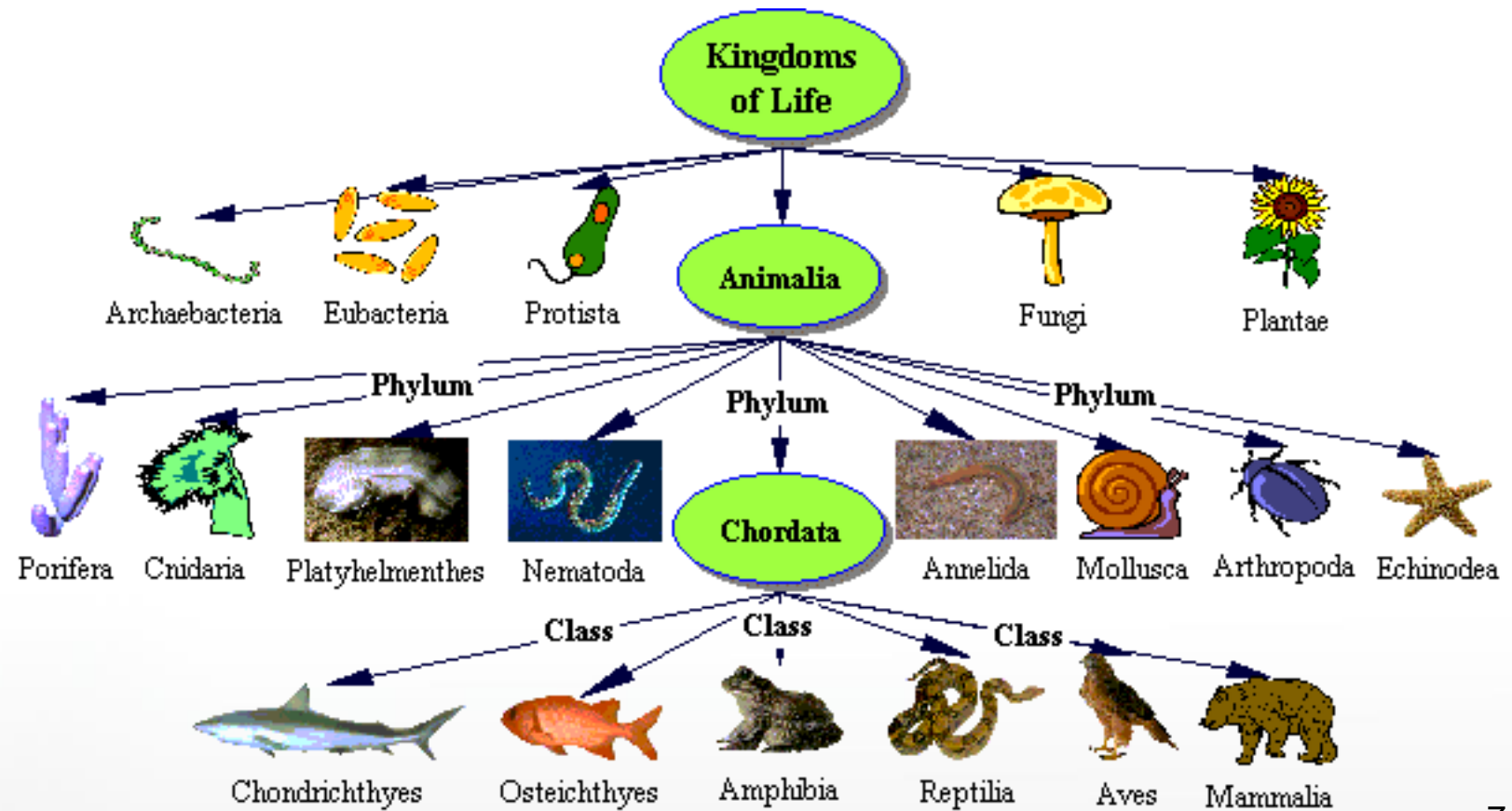


But in reality, `A` holds a reference to an array of 3 items, where each item is a reference to an array of 4 ints.

Palabras Clave

- **Clase:** Es una estructura de datos que modela un objeto.
- **Objeto:** Instancia de una Clase.
- **Instancia:** Cada vez que se llama al constructor de una clase, se forma una nueva instancia de la clase u objeto. Distintas instancias son distintos objetos.
- **This:** Se refiere a **ESTA** instancia en concreto; es decir, este objeto en particular.
- Si todas las personas del mundo fueran objetos, todos serían **instancias distintas** de la clase **Humano**. Te referirías a ti mismo con la palabra **this**, y a los demás con los nombres de objeto que tengan.
- En algunas situaciones, esto no es tan distinto del mundo real. Uno se refiere a sí mismo mediante la palabra “yo” y a los demás por sus nombres. Y si quieres pedirle a Juan algo, digamos un lápiz, esencialmente estás diciendo **juan.prestameUnLapiz()**, por poner un ejemplo.
- Los objetos no tienen por qué representar el mundo real. Son una herramienta para solucionar problemas y tienen desventajas.

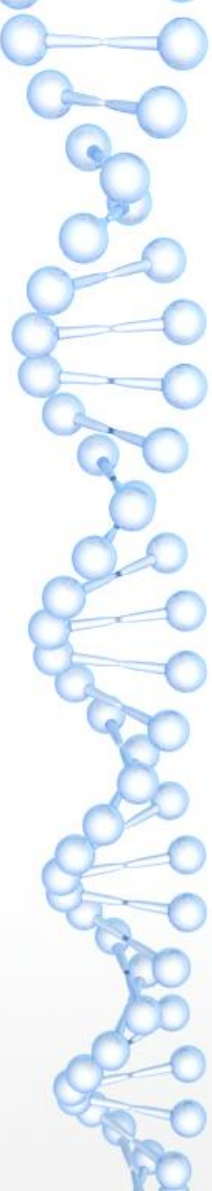
Intentemos representar el mundo real...





Implementacion del Reino Animalia

- .Queremos implementar un Programa que, dentro de lo que cabe, simule de manera realista (bueno, mas o menos) a este reino.
- .No me apetece hacer todo el Reino de animales, asi que me voy a centrar en las Chordata, y mas tarde, en los Mamiferos.
- .Lo primero es crear una clase aplicable a cada miembro del reino animal. Es decir, una clase Animal que contenga datos comunes entre todos los animales.
- .Podéis seguir el desarrollo [aquí](#).



Hagamos un listado de cosas que todos los animales tienen en comun.

- .Cosas que se me ocurren:
- .Edad
- .Tamano
- .Peso
- .Especie (Todo animal pertenece a una especie, no?)
- .Algo mas? Supongamos que no.



Comencemos con la clase Animal

Para crear una clase, hacemos esto:

```
public class Animal {  
  
}
```

Pero claro, esta clase esta vacia!

Podemos comenzar por añadir los atributos que hemos determinado antes:

```
public class Animal {  
  
    // Variables de Campo / Atributos del Objeto  
  
    private String especie; // La especie de nuestro animalito  
  
    private int edad; // La edad de nuestro animalito  
  
    private double peso; // El peso de nuestro animal  
  
    private double tamano; // La altura de nuestro animal  
  
}
```

Esto esta muy bien, pero todavia nos falta bastante!!!



Implementacion

Recordemos que nuestro objetivo es crear un objeto que nos permita representar el reino de los animales de manera simple, pero realista. En esta clase **Animal** nos falta un componente clave de los Objetos: Los Constructores. Un Constructor es un método que instancia la clase. Puede estar vacío, pero normalmente se usa para inicializar los datos dentro de un objeto. Es decir, **asignar un valor especificado a uno (o varios) de los atributos del objeto**. Para la clase **Animal**, debemos crear el siguiente constructor:

```
// Esto es un constructor  
  
public Animal(String especie, int edad, double peso, double tamano) {  
  
    this.especie = especie;  
  
    this.edad = edad;  
  
    this.peso = peso;  
  
    this.tamano = tamano;  
  
}  
  
//-----
```



Getters & Setters

• Si os fijais, vereis que todos los atributos de la clase **Animal** han sido declarados con la palabra **private**. Esto se llama **encapsulacion**, e impide el acceso directo a la variable desde fuera de la **instancia de la clase en la que estamos**. Para explicar, nada mejor que un ejemplo.

• Imaginaos que la palabra **private** guarda los datos de Juan en un cofre cerrado con llave. Solo Juan tiene la llave, así que solo Juan puede acceder a los contenidos de forma directa. Su hermana Juana quiere acceder a esos datos, pero como no tiene la llave, no puede acceder directamente a ellos. Por eso, tiene que pedirle a Juan que le traiga el dato que ella quiere. Esto en programación sería un **Getter**:

Juana → **Juan.getDato();**

Supongamos que Juan le ha dado a Juana el dato, y esto ha enfadado a Juana. También podemos suponer que Juan no le ha dado el dato. En cualquier caso, Juana se ha enfadado con Juan, y ahora quiere modificar el dato en cuestión. Juan es un poco tonto, así que Juana le pide a Juan que modifique un dato. Esto en programación sería un **Setter**:

Juana → **Juan.setDato(modificacion_a_llevar_a_cabo);**



Getters & Setters

• ¿Como se añade esto a la clase Animal? Muy Sencillo:

//Getter

```
public int getEdad() {  
    return this.edad;  
}
```

//-----

//Setter

```
public void setEdad(int edad) {  
    this.edad = edad;  
}
```

//-----

En este caso, solo los he creado para el atributo Edad, pero se hace igual para todos.



class Animal

- La clase ya estaria completada. Ahora nos quedaria añadir metodos a lo que ya tenemos. Un metodo y una funcion no son lo mismo: Un metodo va dentro de un objeto, mientras que una funcion no.
- Para probar, vamos a añadir 2 metodos muy tipicos: **.equals(Object Obj)** y **.toString()**;
- **.equals(Object Obj)** consiste en comparar los valores de los atributos entre 2 objetos. Devuelve un **boolean**. En este caso, el **Object** es un objeto de tipo **Animal**.
- **.toString()** consiste en pasar todos los datos contenidos en el objeto a un String, de forma que sea mas legible.
- Empecemos con **.toString()**, pues es el mas simple de los dos.



.toString()

```
public String toString() {
```

```
    return "Un Especimen de " + this.especie + " de " + this.edad  
        + " años de edad, un peso de " + this.peso  
        + "kg y un tamaño de " + this.tamaño + " metros.";
```

```
}
```

```
//-----
```




.toString()

- Consiste en retornar los contenidos del Objeto (en este caso Animal) en un formato legible. Esto es útil para muchas cosas, desde hacer pruebas hasta imprimir información en la pantalla.
- El formato del String que devuelvas es más bien irrelevante. Tú decides cómo hacerlo. Yo decidí seguir el formato:
 - “Un espécimen de **nombre_de_especie** de **n** años de edad, un peso de **n** kg y un tamaño de **n** metros.”
- Pasemos ahora a **.equals(Object Obj)**, que es mucho más interesante.



.equals(Object Obj)

```
public boolean equals(Object obj) {  
  
    if(obj instanceof Animal) {  
  
        Animal animal = (Animal)(obj);  
  
        return (this.edad == animal.getEdad()  
            && (this.especie.equals(animal.getEspecie()))  
            && (this.peso == animal.getPeso())  
            && (this.tamano == animal.getTamano()));  
  
    }  
  
    else {  
  
        return false;  
  
    }  
  
}
```



.equals(Object Obj)

- En este metodo, usamos **Object Obj** (Un objeto de cualquier tipo). No usamos **Animal** porque queremos que ocurra Overriding con el metodo de la clase **Object**.
- Luego, estamos comprobando si *Obj instanceof Animal*. La utilidad de **instanceof** es comprobar que el objeto de la izquierda (**Obj**) es del mismo tipo que la clase especificada en la derecha (**Animal**). En otras palabras, si **Obj** es un **Animal**.
- Si **Obj** es un **Animal**, entonces tienen los mismos atributos. Por lo tanto, hacemos **casting** del Objeto a uno de tipo **Animal**. Es decir, convertimos **Obj** a un **Animal**. Esto es importante entenderlo, porque si bien llegar a este punto implica que **Obj** es un objeto de tipo **Animal**, el compilador no nos va a dejar acceder a los atributos y metodos de la clase **Animal** desde **Obj**. Por lo que tenemos que hacer al compilador entender esto. Se hace de esta manera: `Animal animal = (Animal)(obj);`
- Y, finalmente, comparamos todos los atributos de **animal** (el objeto convertido a **Animal**) con los atributos de la clase en la que estamos ahora mismo (tambien del tipo **Animal**). Yo lo hago en el **return** pero se podria hacer con **ifs/else**.



Clase Animal Completada

- Nuestra clase base **Animal** ya esta completada. Ahora nos toca añadir otras clases que dependan de la clase **Animal** que acabamos de crear.
- Por simplicidad, voy a dedicarme unica y exclusivamente a las **Chordata**, pero os propongo como ejercicio de practica extenderlo tambien a las demas familias, como los **artropodos**.
- Empecemos Creando la clase **Chordata**. Lo Primero, es determinar que cosas tienen todos los miembros de **Chordata** que no tienen todos los miembros de **Animal**. Por ejemplo,
 - ¿Que tienen los peces y los mamiferos en común?
 - ¿Y los mamiferos y los anfibios y las Aves, pero no tienen en comun con todos los animales?
- Preguntas similares a estas son las que os deberiais hacer a la hora de programar un objeto que **extienda** a otro.
- Al usar **extends** en la declaracion de la clase, todos los atributos y metodos de la clase extendida pasan tambien a la clase extensora. Esto incluye atributos ocultos internos de Programación Orientada a Objetos, como las **Virtual Tables**.



class Chordata extends Animal

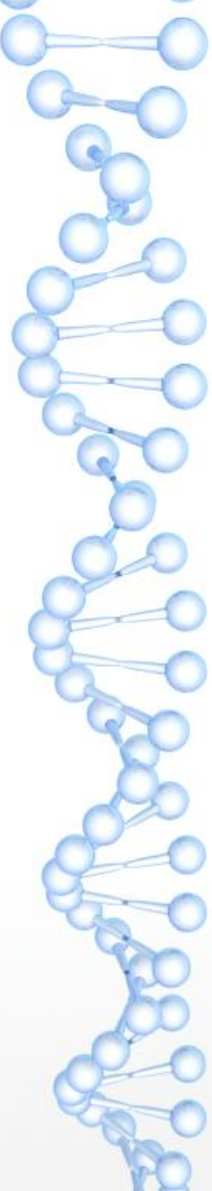
.Eso es más fácil de ver con un ejemplo. Creemos la clase Chordata:

```
public class Chordata extends Animal {}
```

.Aquí estamos haciendo uso de **extends** porque vemos una relacion logica entre todos los **Chordata** y todos los **Animal**, y por lo tanto nos interesa que todos los **Chordata** tengan, como mínimo, todos los datos de los **Animal**. Es decir, al hacer *Chordata extends Animal* en realidad es como hacer lo que se ve en la siguiente diapositiva.

.Al hacer **extends** no solo somos mas rapidos, sino que nos ahorramos una gran cantidad de codigo duplicado.

extends



```
public class Chordata {  
  
    // Variables de Campo / Atributos del Objeto  
  
    private String especie; // La especie de nuestro animalito  
  
    private int edad; // La edad de nuestro animalito  
  
    private double peso; // El peso de nuestro animal  
  
    private double tamano; // La altura de nuestro animal  
    public String toString() {  
  
        //Codigo  
  
    }  
  
    public String equals(Object Obj) {  
  
        //Codigo  
  
    }  
  
}
```



class Chordata

- .Ahora tenemos que pensar en cosas que todo los Chordata tienen. Estas son las que se me ocurren:
- .Cola (Si, todos durante nuestro desarrollo tenemos una cola en algún momento)
- .Espina Dorsal / Notocorda (Por simplicidad, las trataremos como si fueran lo mismo)
- .Hendiduras Faringeas / Pulmones (Nos referiremos a ambos como pulmones por simplicidad)
- .Seguro que hay alguna más, pero esas son las que se me ocurren. Ahora toca implementar estos:

```
public class Chordata extends Animal {  
    private boolean cola;  
    private boolean pulmones;  
    private boolean espina_dorsal;  
}
```




class Chordata extends Animal

- Ahora os sorprendereis de que haya puesto los 3 como **boolean**. ¿Alguien sabe porqué?
- **Cola:** Puse Cola como boolean porque, si bien en algún punto de nuestro desarrollo todos los Chordata tenemos cola, hay especies (como los humanos) que, por lo general, la perdemos antes de nacer.
- **Pulmones:** Puse Pulmones como boolean porque sabemos que un Chordata tiene pulmones o branquias. Los anfibios son un poco raretes, porque tambien poseen respiracion cutanea y demas, pero no tienen los dos. Dependiendo de la especie, tienen uno u otro. **False** representará branquias, mientras que **True** representará pulmones.
- **Espina Dorsal:** Igual que todos tenemos Pulmones o Branquias, tambien tenemos Espina Dorsal o Notocorda. Los humanos, por ejemplo, tenemos Notocorda en el estado de embrion, y luego pasamos a tener Espina Dorsal cuando nos salven vertebras. **False** representará Notocorda y **True** representará la posesión de una Espina Dorsal.
- Ahora debemos actualizar Los metodos de dentr de la clase Chordata.



Constructor de Chordata

```
public Chordata(String especie, int edad, double peso, double tamano,  
    boolean cola, boolean pulmones, boolean espina_dorsal) {  
  
    super(especie, edad, peso, tamano);  
  
    this.colas = cola;  
  
    this.pulmones = pulmones;  
  
    this.espina_dorsal = espina_dorsal;  
  
}
```

La palabra clave aqui es **super**. **Super** sirve para referirse a la superclase (la clase madre o extendida, por asi decirlo), que en este caso seria **Animal**.

super(cosas_aqui) esta llamando al constructor de la superclase. Es esencialmente lo mismo que si hicieras **Animal(cosas_aqui)**

Getters



```
// Getters
```

```
public boolean getCola() {
```

```
    return this.col;
```

```
}
```

```
public boolean getPulmones() {
```

```
    return this.pulmones;
```

```
}
```

```
public boolean getEspina() {
```

```
    return this.espina_dorsal;
```

```
}
```

Setters



```
public void setCola(boolean cola) {
```

```
    this.cola = cola;
```

```
}
```

```
public void setPulmones(boolean pulmones) {
```

```
    this.pulmones = pulmones;
```

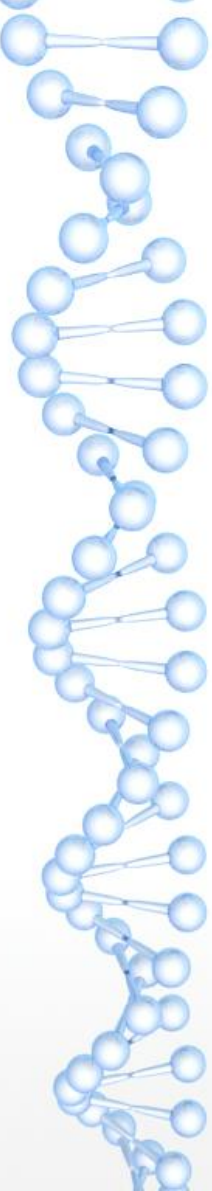
```
}
```

```
public void setEspina(boolean espina) {
```

```
    this.espina_dorsal = espina;
```

```
}
```

.equals



```
public boolean equals(Object Obj) {  
    if(Obj instanceof Chordata) {  
        Chordata animal = (Chordata)(Obj);  
        return super.equals(animal)  
            && (animal.getCola() == this.colas)  
            && (animal.getPulmones() == this.pulmones)  
            && (animal.getEspina() == this.espina_dorsal);  
    }  
    else {  
        return false;  
    }  
}
```

.toString()



```
public String toString() {  
    String cola_string = "No tiene cola";  
    String pulmon_string = "Tiene Branquias";  
    String espina_string = "Notocorda";  
  
    if(this.colas) {  
        cola_string = "Tiene cola";  
    }  
  
    if(this.pulmones) {  
        pulmon_string = "Tiene pulmones";  
    }  
  
    if(this.espina_dorsal) {  
        espina_string = "Espina Dorsal";  
    }  
  
    return super.toString() + " Es un miembro del Filo Chordata, que " + cola_string  
        + ", " + pulmon_string + " y posee una " + espina_string + ".";  
}
```



¿Y Ahora Qué?

.Ahora nos toca crear clases para cada familia dentro del filo Chordata, que serían:

- .Amphibia
- .Aves
- .Chondrychthyes
- .Mammalia
- .Osteichthyes
- .Reptilia



Mammalia

- Voy a centrarme en los mamíferos durante esta presentación, como ya dije que haría anteriormente. ~~Sin embargo, puede (no lo garantizo!) que algunas de las otras clases estén disponibles en mi Github.~~ (No me aburro tanto, hacedlo vosotros y haceis un Pull Request)
- Primero, debemos encontrar aquellas cosas comunes de todos(o casi todos) los mamíferos. Cosas que se me ocurren:
- Pelo
- Glandulas Mamarias
- Sexo (Creo que no existe ninguna especie mamifera hermafrodita)



Constructor

```
public Mammalia(String especie, int edad, double peso, double tamano,  
                boolean cola, boolean pulmones, boolean espina_dorsal,  
                boolean glandulas_mamarias, boolean cuerpo_peludo,  
                boolean sexo) {  
    super(especie, edad, peso, tamano, cola, pulmones, espina_dorsal);  
    this.glandulas_mamarias = glandulas_mamarias;  
    this.cuerpo_peludo = cuerpo_peludo;  
    this.sexo = sexo;  
}
```



Continuamos...

- Continuamos añadiendo los Getters/Setters, los otros metodos del objeto, etc..
- No voy a ponerlos aqui porque son realmente lo mismo que antes otra y otra vez.
- Una vez hayamos terminado, podemos añadir una especie... Como los monos (o incluso los humanos)
- Pero en mi caso voy a crear la clase **AnimalC**, que contendra datos de Animales de Compañía.



Clase Animal de Compañía

- Los animales de compañía tienen un nombre y una dirección.
- Por ello, añadimos estos 2 atributos a la clase, y la completamos con los metodos correspondientes.
- Ahora toca algo interesante: crear un humano. ¿Cómo representamos un humano? Pues hay muchas maneras, pero si usamos objetos, lo más útil sería extender la clase de Animal de compañía. ¿Por qué?
- Los humanos tienen todos los atributos de la clase **Mammalia**, y además tienen un **Nombre** y una **dirección**. La única clase que ya tiene todo esto, es la de **AnimalC** (animal de Compañía). Entonces, tiene sentido usar esta, y no crear una clase nueva entera para un humano.



Clase Humano

- .Bueno la clase Humano no es más que un extensión de **AnimalC**.
- .Pero se puede ver mi implementación [aquí](#).
- .Ahora bien, ¿cómo quedaría esto sin usar objetos?
- .Pues lo podéis ver en el repositorio de github ([Aquí](#)).
- .Técnicamente, esta segunda opción sería más eficiente. Es la que deberíamos implementar cuando nuestro problema no depende de varias clases y solo necesitamos la de Humano. Además, aunque parezca que no, la clase **HumanoFeo** es más corta que la clase Humano, ya que la clase Humano también debe contar las clases de las que descende.
- .Es decir, si tenéis un problema que implica la creación de una o más clases que no tienen relación entre sí, no os inventéis clases adicionales. Solo debería haber **Herencia** si es útil de cara al problema.



Resumen Logico

- .Todos los mamiferos son chordatas.
- .Todo chordata es un animal.
- .Por lo tanto, todo mamífero es un animal.
- .De esa manera, podemos decir que si un humano es un mamífero, entonces un humano es un animal.
- .¿Verdad?
- .Pues asi es como funciona la Programacion Orientada a Objetos.
- .Esto se puede aplicar a cualquier tipo de objeto que se tenga.
- .Este ejemplo se adecúa (más o menos) al mundo real, pero no siempre será así. Ya habeis visto que hemos descrito a los Humanos como Animales de Compañía, y eso no tiene tanto sentido como decir que un Humano es un mamífero.



Ejercicios Propuestos

- Si os apetece practicar estos conceptos, podeis extender este sistema para los demas miembros del filo Chordata; Y si terminais eso, podeis extenderlo a todos los demas del reino Animalia. Y si quereis, tambien podeis hacer las clases existentes mas realistas, crear nuevos objetos para cosas como extremidades y demas.
- Ademias, os invito a que termineis los metodos **toString()** y **equals(Object Obj)** de la clase HumanoFeo (la que no usa OOP).
- Por Ultimo, y si hay una cosa que si que os recomendaria hacer, es intentar crear una estructura de datos que guarde un arbol genealogico de entidades del tipo Humano. Pero para eso, todavia hay que explicar mas cosas.
- Para que sepais crear una estructura de datos que parezca un arbol genealogico, hay que entender punteros a objetos. Esto es más facil de visualizar con un ejemplo de una **Lista Enlazada (Linked List)**.
- Una ArrayList es esencialmente un híbrido entre Array y LinkedList, asi que esto tambien os debería ayudar a entender el concepto de ArrayList (Y prácticamente cualquier otra estructura de datos).

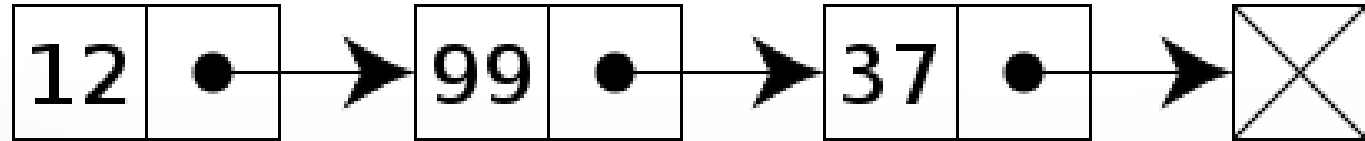


Linked List

- Una estructura de datos fundamental.
- Contiene datos en forma de nodos.
- Cada nodo apunta al siguiente.
- Según la Wikipedia:
 - *“usada para implementar otras estructuras de datos. Consiste en una secuencia de nodos, en los que se guardan campos de datos arbitrarios y una o dos referencias, enlaces o punteros al nodo anterior o posterior”*
- ¿Cómo se podría implementar esto? (Java tiene la suya propia)

Linked List

- Como hemos dicho, los Nodos almacenan datos. Hay maneras de hacer que puedan almacenar cualquier dato. En Java esto se puede hacer con **generics**, y no es muy difícil de hacer. Sin embargo, es añadir una complejidad innecesaria, ya que no queremos crear una Lista de calidad profesional (Java, y muchas otras librerías ya las tienen), sino entender el funcionamiento básico de una Lista.
- Además, cada Nodo debe apuntar al siguiente nodo de la lista, y a la hora de iterar, pasaremos por cada nodo hasta llegar al deseado.
- Siempre se puede hacer que cada nodo también apunte al anterior, pero eso lo podéis hacer vosotros si queréis, no veo la necesidad de demostrar eso.



Nodo

.Si deseamos almacenar un Humano por Nodo, podemos hacer una clase Nodo de esta manera:

```
public class Nodo {  
    private Humano datos;  
    private Nodo siguiente;  
    public Nodo(Humano datos, Nodo siguiente) {  
        this.datos = datos;  
        this.siguiente = siguiente;  
    }  
    //-----  
}
```

.Los demas metodos son Getters, Setters, **.toString()** y **.equals(Object Obj)** nada especiales. Los podeis ver en el archivo completo.



Linked List

.Esta clase ya es más interesante. Tiene una gran cantidad de metodos que la proveen de la funcionalidad que queremos. Comencemos con lo más basico: los atributos y el constructor. Recordad que esto se puede hacer de muchísimas maneras, pero yo lo he decidido hacer así porque quiero hacer la lista lo más simple y fácil de entender posible.

```
public class ListaEnlazada {
```

```
    private Nodo cabeza;
```

```
    private int tamano;
```

```
    public ListaEnlazada() {}
```

```
    //-----
```

```
}
```

.Ahora vendrían los Getters y los Setters, pero los voy a mostrar. Cabe mencionar que tamano no tiene setter.



Linked List: vaciarLista()

- Para vaciar la lista, basta con decir que la cabeza es **null**. Así, el primer nodo será **null** y por tanto todas las referencias a otros nodos también lo serán.
- Luego se dirá que el tamaño es 0.
- Para comprobar si una lista está vacía, basta con comprobar si la cabeza es **null**.

```
public void vaciarLista() {
```

```
    this.cabeza = null;  
    this.tamano = 0;
```

```
}
```

```
//-----
```

```
public boolean estaVacia() {
```

```
    return this.cabeza == null;
```

```
}
```

```
//-----
```

Linked List: getCola()

.Si definimos la cola como el último nodo de la lista, la manera de obtenerlo sería mediante un metodo **getCola()**:

```
public Nodo getCola() {  
    Nodo nodo = this.cabeza;  
    while(nodo.haySiguiete()) {  
        nodo = nodo.getSiguiete();  
    }  
    return nodo;  
}  
//-----
```



Linked List: getNode(int pos)

```
public Nodo getNode(int pos) {
```

.También es útil tener un método que nos permita obtener un nodo basado en su índice de la lista. Para ello, iteramos por la lista hasta que hayamos llegado al índice deseado.

.Es importante entender que este método no está bien programado, ya que no estamos usando ninguna cláusula Try/Catch ni ningún Throws. Estas cláusulas nos permitirían evitar errores en el caso de que el nodo fuera **null**. Además, se le podría añadir un check que comprobara si el nodo cabeza es **null**.

```
int i = 0;
```

```
Nodo nodo = this.cabeza;
```

```
while(i < pos && nodo.haySiguiente()) {
```

```
    nodo = nodo.getSiguiente();  
    ++i;
```

```
}
```

```
// Aqui se debería hacer una excepcion  
// en el caso de que la cabeza  
// sea null o que el Nodo nodo sea null  
// Pero no quiero añadir complejidad
```

```
return nodo;
```

```
}
```



Linked List: Anadir(Nodo n)

- Este metodo añade un Nodo nuevo al final de la lista.
- Funciona de una manera muy simple: vamos iterando por la lista hasta que lleguemos al ultimo nodo. Una vez ahí, decimos que el últimoNodo.siguiente es el nodo n.
- Es decir, el siguiente nodo al último nodo se vuelve el nodo n, y por tanto tenemos un nuevo último.

```
public void anadir(Nodo n) {
```

```
    Nodo nodo = this.cabeza;  
    while(nodo.haySiguiente()) {
```

```
        nodo = nodo.getSiguiente();
```

```
    }
```

```
    nodo.setSiguiente(n);  
    ++this.tamano;
```

```
}
```

```
//-----
```


Linked List: quitar()

.Para quitar el último nodo es un poco más complejo. Como siempre, hay varias maneras de hacerlo, pero mi manera consiste en lo siguiente:

.Primero iteramos hasta que el siguiente nodo no tenga otro más. En otras palabras, iteramos hasta que el siguiente elemento sea **la cola**.

.Una vez hayamos llegado al penúltimo nodo, hacemos que el siguiente nodo sea null. Es decir, hacemos que el último nodo en la lista sea null.

.Por último, decrementamos el tamaño por 1.

```
public void quitar() {  
    Nodo nodo = this.cabeza;  
    while(nodo.getSiguiente().haySiguiente()) {  
        nodo = nodo.getSiguiente();  
    }  
    nodo.setSiguiente(null);  
    --this.tamano;  
}  
//-----
```



Linked List: Insertar(Nodo n, int pos)

- Insertar es un metodo bastante importante que consiste en añadir un **nodo n** en una posición **pos** de la lista.
- Una vez más, hay varias maneras de implementar este metodo pero mi manera es esta:
- Primero iteramos desde el primer nodo hasta el anterior a la posición que estamos buscando.
- Luego decimos que el siguiente nodo a **n** es el siguiente al Nodo nodo con el que hemos iterado.
- Y finalmente, que el Nodo siguiente al nodo nodo, es el nodo **n**.

```
public void insertar(Nodo n, int idx) {  
    int i = 0;  
    Nodo nodo = this.cabeza;  
    while(i < idx - 1 && nodo.haySiguiente()) {  
        nodo = nodo.getSiguiente();  
        ++i;  
    }  
    // Aqui hemos llegado al indice  
  
    n.setSiguiente(nodo.getSiguiente());  
    nodo.setSiguiente(n);  
    ++this.tamano;  
}
```

//-----

Linked List: eliminar(int idx)

• Eliminar es practicamente igual que insertar, pero hacemos que el siguiente nodo al Nodo nodo sea el siguiente del siguiente del nodo nodo. Sé que suena un poco raro, pero si usais un papel y representais cada nodo con cuadraditos, os dareis cuenta de que tiene sentido.

```
public void eliminar(int idx) {  
  
    int i = 0;  
    Nodo nodo = this.cabeza;  
  
    while(i < idx - 1 && nodo.haySiguiente()) {  
  
        nodo = nodo.getSiguiente();  
        ++i;  
  
    }  
    nodo.setSiguiente(nodo.getSiguiente().getSiguiente());  
    --this.tamano;  
  
}
```

//-----



*LinkedList:listaContiene(Humano h)

.Este metodo consiste en determinar si la lista contiene un nodo cuyos datos sean iguales al Humano h.

.Simplemente itera por cada nodo dentro de la lista y compara sus datos con el Humano h.

```
public boolean listaContiene(Humano h) {
```

```
    Nodo nodo = this.cabeza;  
    boolean encontrado = false;  
    while(nodo.haySiguiete() && !encontrado) {
```

```
        if(nodo.getDatos().equals(h)) {
```

```
            encontrado = true;
```

```
        }
```

```
    }
```

```
    return encontrado;
```

```
}
```

```
//-----
```



LinkedList::toString()

```
public String toString() {  
  
    String str = "[";  
    Nodo nodo = this.cabeza;  
    int i = 0;  
  
    while(nodo != null) {  
  
        str += i + "->" + nodo.getDatos().getNombre() + ", ";  
        ++i;  
        nodo = nodo.getSiguiente();  
  
    }  
  
    return str += "];"  
  
}
```

//-----



Final

- Si queréis, podéis añadir el metodo **.equals(Object Obj)**
- Espero que os haya resultado útil.