**Introducción a la POO:**
➢ Paradigma: Forma de entender y representar la realidad. Es decir la forma en la que uno tiene que pensar la resolución de los problemas en diferentes escenarios y con diferentes herramientas.

**Paradigma Orientado a Objetos:**
➢ Tenemos entidades llamadas objetos que son los dueños de los datos quienes implementan las funcionalidades que queremos construir. Estos utilizan datos de entrada de una fuente, los procesan y realizan una salida específica.
➢ Cada objeto tiene una responsabilidad primaria que se encarga de implementar.
➢ Metodología de desarrollo de aplicaciones en la cual éstas se organizan como colecciones de cooperativas de objetos, cada uno de los cuales representan una instancia de alguna clase. Es decir tenemos una cooperación entre objetos cada uno de los cuales es una instancia de las clases**.**

**No basta un lenguaje OO para programar orientado a objetos, para eso hay que seguir un paradigma orientado a objetos.**

- Se llama así a cualquier lenguaje de programación que implemente los conceptos definidos en la programación orientada a objetos.
Ejemplos: C++, C#, PHP, Java.

# Conceptos Básicos

## Objeto

Es una entidad autónoma que contiene atributos y comportamiento.
Se combinan datos y la lógica de programación.
Tienen estados (sustantivos) y comportamientos (verbos).
Atributo: info que no es relevante de un objeto para un problema determinado comportamiento: todas aquellas acciones que un objeto puede realizar.
Ejemplo de objeto persona:

**Atributos:**

- Nombre
- Edad
- Peso
- Altura
- Fecha de nacimiento
- etc.

**Comportamiento:**

- Hablar
- Caminar
- Comer
- Imprimir datos
- etc.

## Clase

Una **clase** es una plantilla que define la forma de **un objeto**. Especifica los datos y el código que operará en esos datos. **Java** usa una especificación de **clase** para construir objetos. Los objetos son instancias de una **clase**.

Es una plantilla para la creación de objetos.
Cada clase es un modelo que define un conjunto de variables (atributos) y métodos (comportamientos).

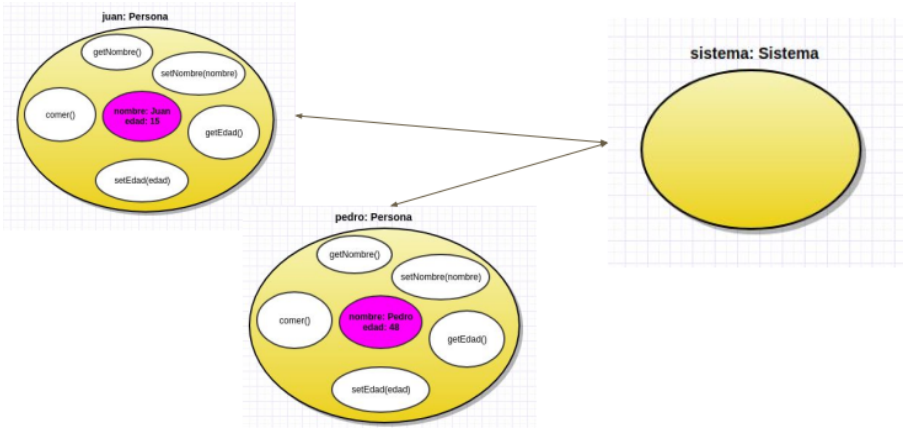
```java
class Persona {

    //Atributos
    String nombre;
    int edad;

    //Métodos
    void caminar() {
        //TODO: cuerpo del método
    }

    String getNombre() {
        return nombre;
    }


}
```

## Instancia

Cada objeto creado a partir de la clase. Es decir, un objeto es una instancia de una clase.
Ejemplo: persona1 = persona.

## Comunicación entre objetos:

Lo realizan a través de los métodos.




## Modificadores de accesos:

➢ **public**: ofrece la máxima visibilidad. Una variable, método o clase será visible desde cualquier clase.
➢ **private**: cuando un método o un atributo es declarado como private, su uso queda restringido al interior de la misma clase.
➢ **protected**: un método o atributo declarado como protected es visible para las clases del mismo paquete y subclases.
➢ (**default**): visibilidad para clases del mismo paquete.

● El modificador de acceso “private” indica que la variable de instancia “nombre” puede ser leída desde la clase Animal. El resto de las clases que quieran leerla deben acceder a través del getter.
● El modificador de acceso “protected” indica que la variable de instancia “nombre” puede ser leída desde las subclases.

## Pilares de la POO

# Abstracción:

Tomar a un elemento de su contexto, nos interesa que es lo que hacemos, cómo lo hace, también que propiedades o datos va a tener que manejar para poder realizar las acciones que sean necesarias.

- Consiste en aislar un elemento de su contexto → ¿Qué hace?
- Se enfoca en la visión externa de un objeto → Separar el comportamiento específico.
- Quitar las propiedades y acciones de un objeto para dejar solo aquellas que sean necesarias.

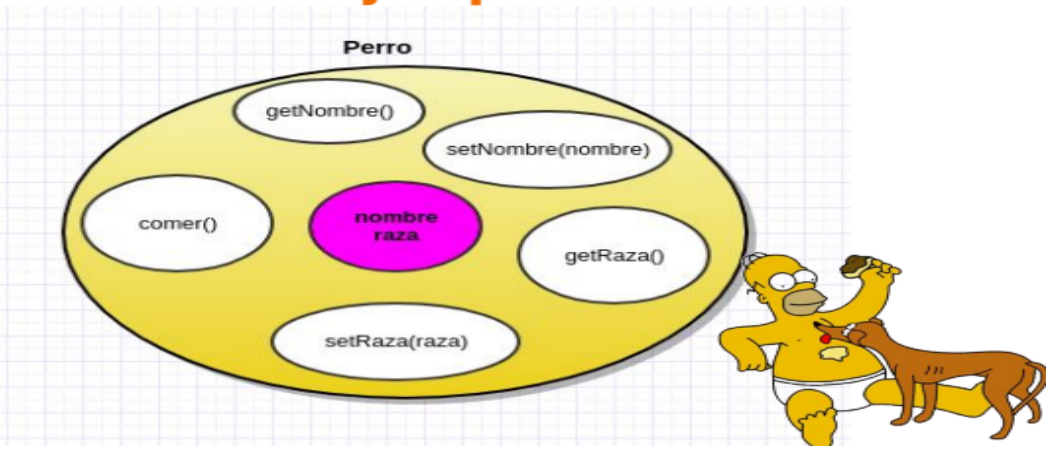
La abstracción es clave para diseñar un buen software.


## **Encapsulamiento**:

Empaquetar el objeto de tal manera que los datos queden aislados del exterior. Ocultamiento de los datos de un objeto de manera que sólo se pueda cambiar mediante las operaciones definidas por ese objeto.

- Ocultamiento de los datos de un objeto de manera que sólo se pueda cambiar mediante las operaciones definidas por ese objeto.
- Empaquetamiento → Objetos aislados desde el exterior.

# Encapsulamiento - Ejemplo



# ¿Cómo lograr la abstracción?

En un nivel superior, Abstracción es un proceso de ocultar los detalles de implementación y mostrar solo la funcionalidad para el usuario. Solo indica cosas importantes para el usuario y oculta los detalles internos, es decir. Mientras envía SMS, simplemente escriba el texto y envíe el mensaje. Aquí, no le importa el procesamiento interno de la entrega del mensaje. La abstracción se puede lograr utilizando la clase abstracta y el método abstracto en Java.

## Clase abstracta

Una clase que se declara “abstracta” se llama como una clase abstracta. Puede tener métodos abstractos y métodos concretos. Una clase normal no puede tener métodos abstractos.

## Método abstracto

Un método sin cuerpo se conoce como Método abstracto. Debe ser declarado en una clase abstracta. El método abstracto nunca será definitivo porque la clase abstracta debe implementar todos los métodos abstractos.

**Reglas del método abstracto**

- Los métodos abstractos no tienen una implementación; solo tiene firma de método
- Si una clase usa un método abstracto, debe declararse abstracto. Lo opuesto no puede ser cierto. Esto significa que una clase abstracta no necesariamente tiene un método abstracto.
- Si una clase regular extiende una clase abstracta, entonces esa clase debe implementar todos los métodos abstractos del padre abstracto

## Diferencia entre abstracción y encapsulación

| Abstracción | Encapsulación |
| --- | --- |
| La abstracción resuelve los problemas en el nivel de diseño. | La encapsulación lo resuelve a nivel de implementación. |
| La abstracción consiste en ocultar detalles no deseados mientras se muestra la información más esencial. | La encapsulación significa ocultar el código y los datos en una sola unidad. |
| La abstracción permite centrarse en lo que debe contener el objeto de información | La encapsulación significa ocultar los detalles internos o la mecánica de cómo un objeto hace algo por razones de seguridad. |

## Diferencia entre clase abstracta e interfaz

| Clase abstracta | Interfaz |
| --- | --- |
| Una clase abstracta puede tener métodos abstractos y no abstractos. | La interfaz solo puede tener métodos abstractos. |
| No admite herencias múltiples. | Es compatible con herencias múltiples. |
| Puede proporcionar la implementación de la interfaz. | No puede proporcionar la implementación de la clase abstracta. |
| Una clase abstracta puede tener métodos públicos protegidos y abstractos. | Una interfaz solo puede tener métodos abstractos públicos. |

| | |
|---|---|
| Una clase abstracta puede tener una variable final, estática o estática con cualquier especificador de acceso. | La interfaz solo puede tener una variable final estática pública. |

## Ventajas de la abstracción

- El principal beneficio de usar una clase abstracta es que te permite agrupar varias clases relacionadas como hermanos.
- La abstracción ayuda a reducir la complejidad del diseño y el proceso de implementación del software.

## ¿Cuándo usar Abstract Methods & Abstract Class?

Los métodos abstractos se declaran principalmente cuando dos o más subclases también hacen lo mismo de diferentes maneras a través de diferentes implementaciones. También extiende la misma clase de Resumen y ofrece diferentes implementaciones de los métodos abstractos.
Las clases abstractas ayudan a describir los tipos genéricos de comportamientos y la jerarquía de clases de programación orientada a objetos. También describe subclases para ofrecer detalles de implementación de la clase abstracta.

# Resumen:

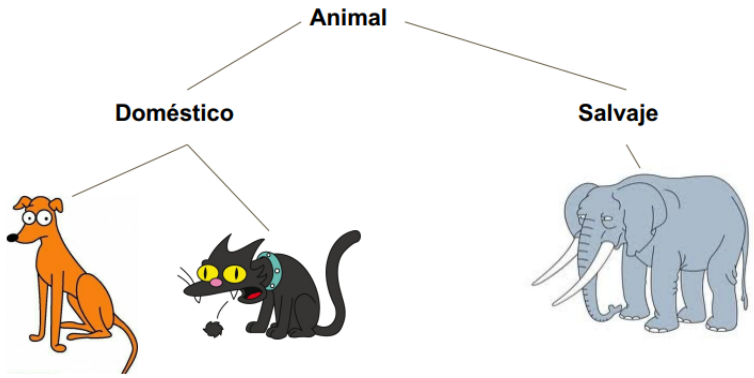
- La abstracción es el proceso de seleccionar conjuntos de datos importantes para un Objeto en su software y omitir los insignificantes.
- Una vez que haya modelado su objeto usando Abstraction, se podría usar el mismo conjunto de datos en diferentes aplicaciones.
- En Java, la abstracción se logra utilizando clases e interfaces abstractas. Estudiaremos en detalle acerca de las clases abstractas y las interfaces en otros tutoriales.

# Herencia:

¿Qué es Herencia?
Podemos definir la herencia como la capacidad de crear clases que adquieren de manera automática los miembros (atributos y métodos) de otras clases que ya existen, pudiendo al mismo tiempo añadir atributos y métodos propios.

➔ **Herencia**:Realizar especializaciones de un tipo de clase para especificar de mejor manera su comportamiento y también los datos que debe almacenar.
➢ La herencia es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente.
➢ Permite compartir automáticamente métodos y características entre clases.
➢ Está fuertemente ligada a la reutilización de código.

La herencia es un pilar importante de **OOP (Programación Orientada a Objetos)**. Es el mecanismo en Java por el cual una clase permite heredar las características (atributos y métodos) de otra clase. Aprenda más a continuación.
En el lenguaje de Java, una clase que se hereda se denomina superclase. La clase que hereda se llama subclase. Por lo tanto, una subclase es una versión especializada de una superclase. Hereda todas las variables y métodos definidos por la superclase y agrega sus propios elementos únicos.




## Terminología importante de Herencia :

- **Superclase**: la clase cuyas características se heredan se conoce como superclase (o una clase base o una clase principal).Cada clase tiene sólo una clase padre (superclase). Una superclase puede tener cualquier número de subclases.
- **Subclase**: la clase que hereda la otra clase se conoce como subclase (o una clase derivada, clase extendida o clase hija). La subclase puede agregar sus propios campos y métodos además de los campos y métodos de la superclase. Puede tener solo una superclase (padre).
- **Reutilización**: la herencia respalda el concepto de “reutilización”, es decir, cuando queremos crear una clase nueva y ya hay una clase que incluye parte del código que

queremos, podemos derivar nuestra nueva clase de la clase existente. Al hacer esto, estamos reutilizando los campos/atributos y métodos de la clase existente.

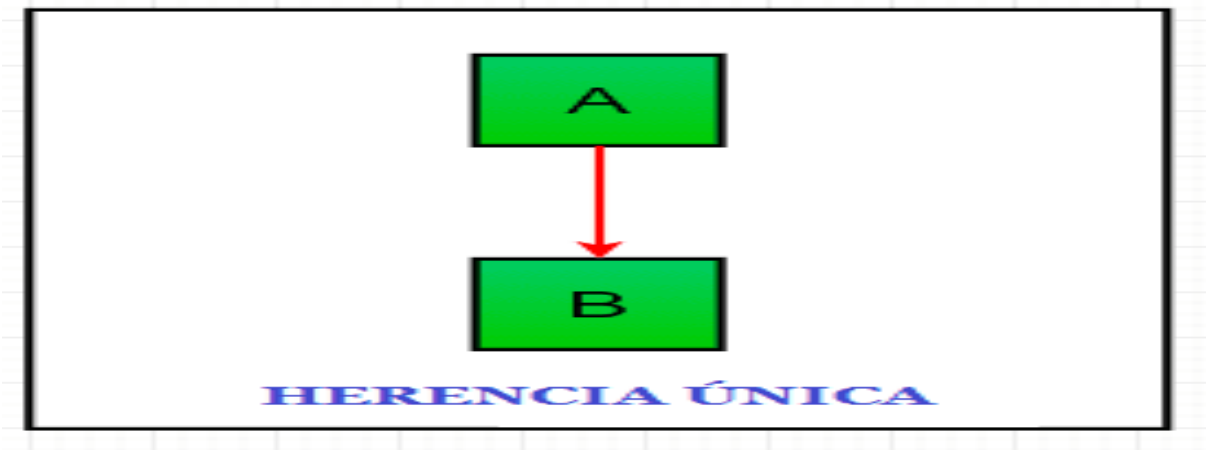
## Ventajas de Herencia:

- **Reutilización del código**: En aquellos casos donde se necesita crear una clase que, además de otros propios, deba incluir los métodos definidos en otra, la herencia evita tener que reescribir todos esos métodos en la nueva clase.
- **Mantenimiento** de aplicaciones existentes: Utilizando la herencia, si tenemos una clase con una determinada funcionalidad y tenemos la necesidad de ampliar dicha funcionalidad, no necesitamos modificar la clase existente (la cual se puede seguir utilizando para el tipo de programa para la que fue diseñada) sino que podemos crear una clase que herede a la primera, adquiriendo toda su funcionalidad y añadiendo la suya propia.

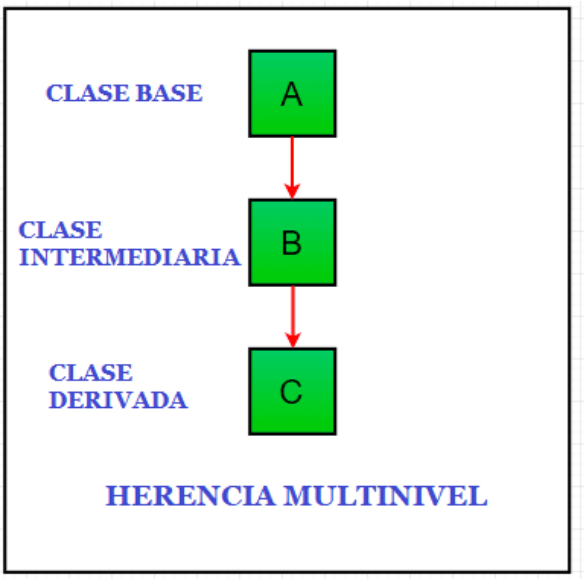
La palabra clave **utilizada para la herencia es extends**

## Tipos de Herencia:

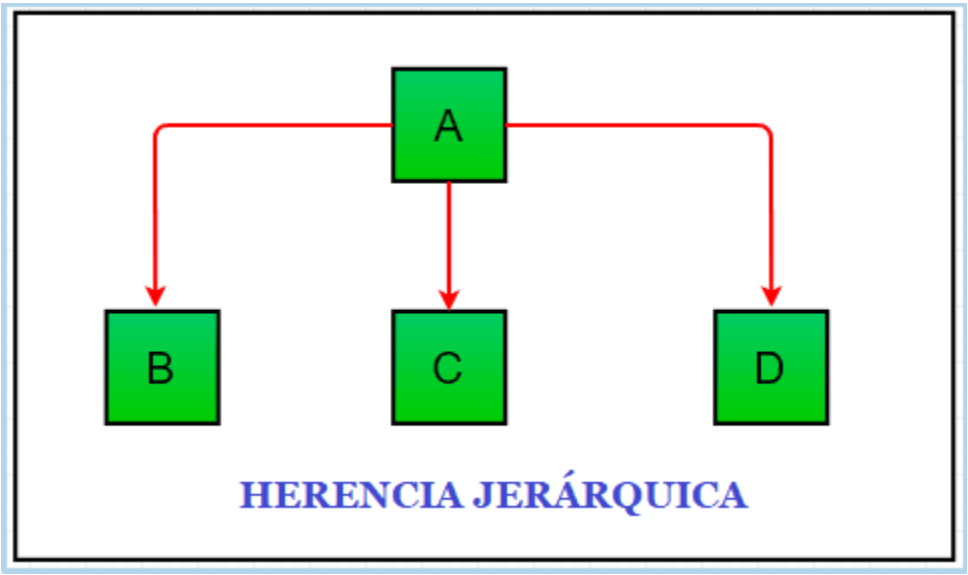
- **Herencia única**: en la herencia única, las subclases heredan las características de solo una superclase. En la imagen a continuación, la clase A sirve como clase base para la clase derivada B.



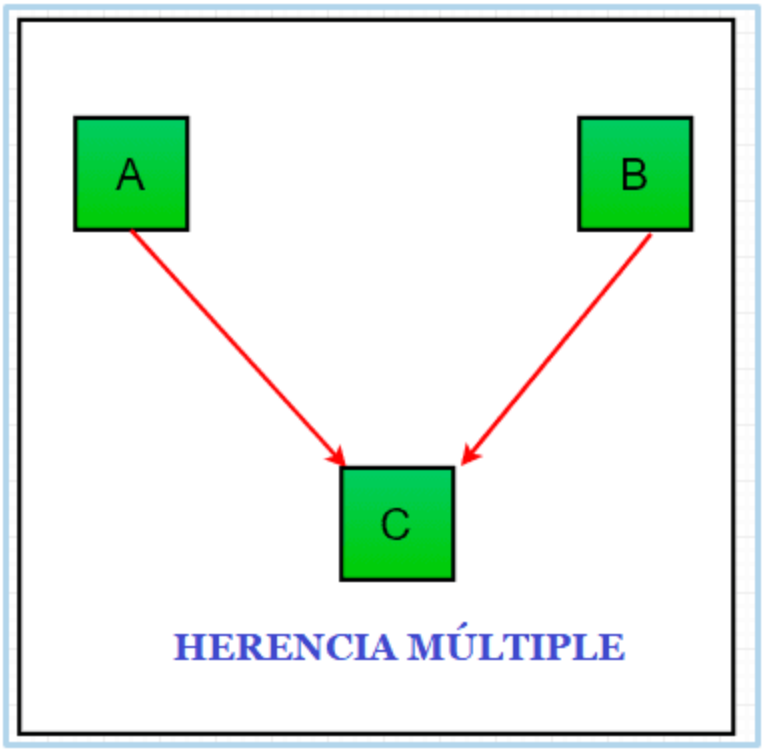

- **Herencia Multinivel**: en la herencia multinivel, una clase derivada heredará una clase base y, además, la clase derivada también actuará como la clase base de otra clase. En la imagen inferior, la clase A sirve como clase base para la clase derivada B, que a su vez sirve como clase base para la clase derivada C. En Java, una clase no puede acceder directamente a los miembros de los “abuelos”.



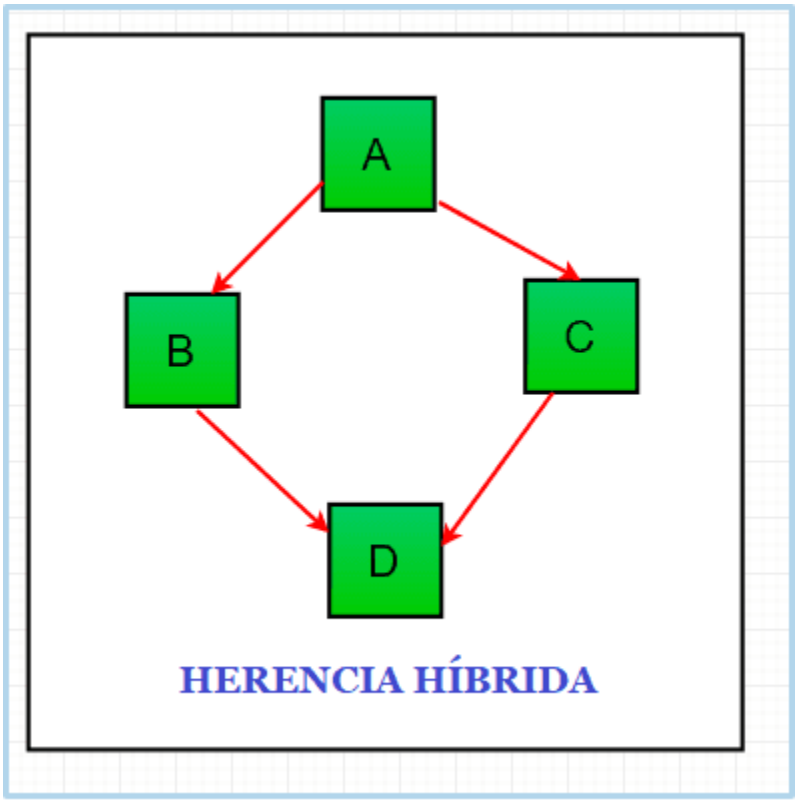

**Herencia Jerárquica**: en la herencia jerárquica, una clase sirve como una superclase (clase base) para más de una subclase. En la imagen inferior, la clase A sirve como clase base para la clase derivada B, C y D




- **Herencia Múltiple (a través de interfaces)**: en Herencia múltiple, una clase puede tener más de una superclase y heredar características de todas las clases principales. Tenga en cuenta que Java no admite herencia múltiple con clases. En Java, podemos lograr herencia múltiple **sólo a través de Interfaces**. En la imagen a continuación, la Clase C se deriva de la interfaz A y B.




**Herencia Híbrida (a través de Interfaces)**: Es una mezcla de dos o más de los tipos de herencia anteriores. Como Java no admite herencia múltiple con clases, la herencia híbrida tampoco es posible con clases. En Java, podemos lograr **herencia híbrida sólo a través de Interfaces**



## Herencia y Abstracción

➔ Se puede pensar en una jerarquía de clases como la definición de conceptos abstractos en lo alto de la jerarquía.
➔ Las subclases no están limitadas al estado y comportamiento provisto por la superclase → pueden agregar variables y métodos además de los que ya heredan. (Si la clase padre tiene un método ej hacer ruidos en las clases hijas puede haber un método particular de cada una de esas clases).
➔ Las clases hijas también pueden sobreescribir los métodos que heredan.

## Datos importantes acerca de la herencia en Java

- **Superclase predeterminada:** excepto la clase Object, que no tiene superclase, cada clase tiene una y solo una superclase directa (herencia única). En ausencia de cualquier otra superclase explícita, cada clase es implícitamente una subclase de la clase Object.
- **La superclase** solo puede ser una: una superclase puede tener cualquier cantidad de subclases. Pero una subclase sólo puede tener una superclase. Esto se debe a que Java no admite herencia múltiple con clases. Aunque con interfaces, la herencia múltiple es compatible con java.
- **Heredar constructores:** una subclase hereda todos los miembros (campos, métodos y clases anidadas) de su superclase. Los constructores no son miembros, por lo que no son heredados por subclases, pero el constructor de la superclase puede invocarse desde la subclase.
- **Herencia de miembros privados:** una subclase no hereda los miembros privados de su clase principal. Sin embargo, si la superclase tiene métodos públicos o protegidos (como getters y setters) para acceder a sus campos privados, estos también pueden ser utilizados por la subclase.

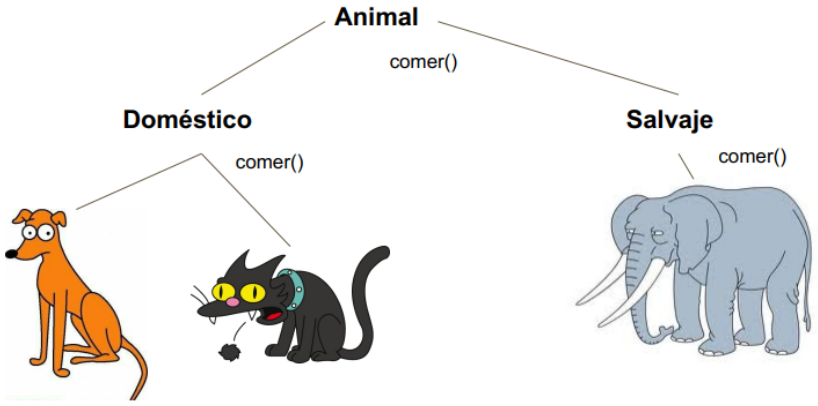
# Polimorfismo:

En programación orientada a objetos, el polimorfismo es la capacidad que tienen los objetos de una clase en ofrecer respuesta distinta e independiente en función de los parámetros (diferentes implementaciones) utilizados durante su invocación.

- Varias formas de responder el mismo mensaje → Muchos mensajes con el mismo nombre en diferentes clases.
Es como un mismo método responde de diferentes maneras dependiendo el contexto en el que lo llamen o los parámetros que se le manden.

➔ Se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos.

➔ Está ligado a la herencia.
➔ Los métodos que se pueden invocar son los que contiene la clase del tipo declarado.
➔ Si hay métodos declarados en la subclase (Perro) que no pertenecen a la superclase (Animal), no se pueden invocar.
● Casting: se denomina casting a la conversión de una referencia de un tipo a otro.
● Java no permite la herencia múltiple.




# Polimorfismo - Ejemplo

```
public abstract class Animal {

      //Variable de instancia
      protected String nombre;

      //Constructor con parámetros
      public Animal(String nombre) {
         this.nombre = nombre;
      }

      public abstract void emitirSonido();

}
```

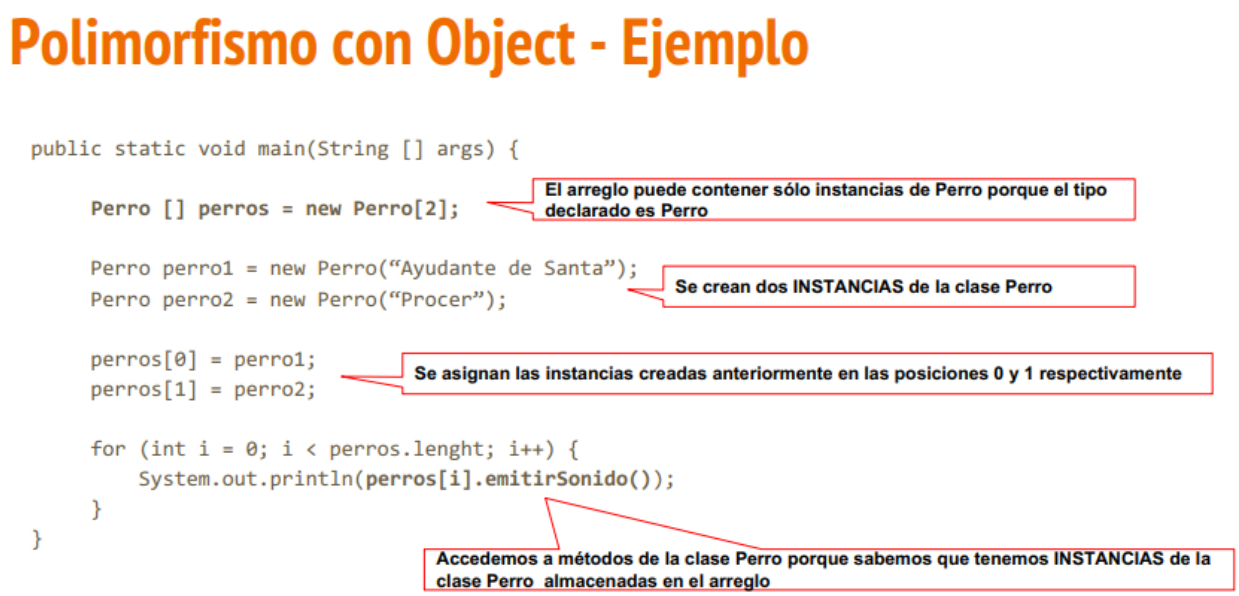
```
public class Perro extends Animal {
      @Override
      public void emitirSonido() {
           System.out.println("Guau!");
      }
}
```

```
public class Gato extends Animal {
      @Override
      public void emitirSonido() {
           System.out.println("Miau!");
      }
}
```

## Formas de polimorfismo:

➢ Sobrecarga de métodos: los mensajes se diferencian en los parámetros.
➢ Sobreescritura de métodos: un hijo sobreescribir un método de la clase padre.
➢ Vinculación dinámica: Herencia.

Polimorfismo con objetos:




# Instanceof

Obj instanceof clase;

Al lado izquierdo tenemos la instancia y el lado derecho el nombre de la clase y como resultado el operador **instanceOf** devuelve un resultado booleano. Con este resultado (true o false, verdadero o falso) podemos determinar si el objeto al que apunta una referencia dada es una instancia de una clase o interfaz concretas.

# Variables:

➔ Una variable es un identificador que representa una palabra de memoria que contiene información.
<tipo> <identificador>;
<tipo> <identificador> = <valor>;

## Clasificación de variables:

➔ **Variables de instancia**: los valores que pueden tomar son únicos para cada instancia.
Ejemplo:
Public class Person {
int age;

}
➔ **Variables de clase:** se declaran con el modificador static para indicarle al compilador que hay exactamente una copia de la variable, y es compartida por todas las instancias de la misma clase.
Ejemplo:
Public class Bicicleta {
static int cantRuedas = 2;
}
➔ **Variables locales**: la determinación viene desde la ubicación en donde la variable fue creada, es decir, local al método. Sólo es visible al método donde fue declarada y no puede ser accedida desde el resto de las clases.
Ejemplo:
public void incrementarContador() {
int contador = 0; contador = contador + 1;
}
➔ **Parámetros:** se pasan entre métodos.
Ejemplos:
public void incrementarContador(int contador) {
contador = contador + 1;
}

## Reglas de las variables

1) Siempre deben comenzar con una letra.
2) Los caracteres subsecuentes pueden ser letras, dígitos, $ o _.
3) Usar palabras descriptivas y no abreviadas.
4) Los nombres no deben contener palabras reservadas en Java.
5) Si el nombre contiene más de una palabra, se capitaliza la primera letra de cada palabra subsecuente.

# JVM (Java Virtual Machine)

➔ **Zona de datos** → donde se almacenan las instrucciones del programa, las clases con sus métodos y constantes. No se puede modificar en tiempo de ejecución.
➔ **Stack** → Aquí se almacenan las instancias de los objetos y los datos primitivos (int, float, etc.)
➔ **Heap** → zona de memoria dinámica. Almacena los objetos que se crean.

## JVM - Tareas principales

➔ Reservar espacio en memoria para los objetos creados.
➔ Liberar la memoria no usada.
➔ Asignar variables a registros y pilas.

➔ Llamar al sistema huésped para ciertas funciones, como los accesos a los dispositivos.
➔ Vigilar el cumplimiento de las normas de seguridad de las aplicaciones Java.

## JVM - Garbage Collector

➔ Técnica por la cual el ambiente de objetos se encarga de destruir y asignar automáticamente la memoria heap.
➔ Un objeto podrá ser “limpiado” cuando desde el stack ninguna variable haga referencia al mismo.

# Tipo objeto

➔ Son instancias de clases, que pueden estar predefinidas o definidas por el programador.
➔ Cuando un objeto es instanciado utilizando el operador new, se retorna una dirección de memoria.
➔ La dirección en memoria contiene una referencia a una variable.

# Modificador static & non-static:

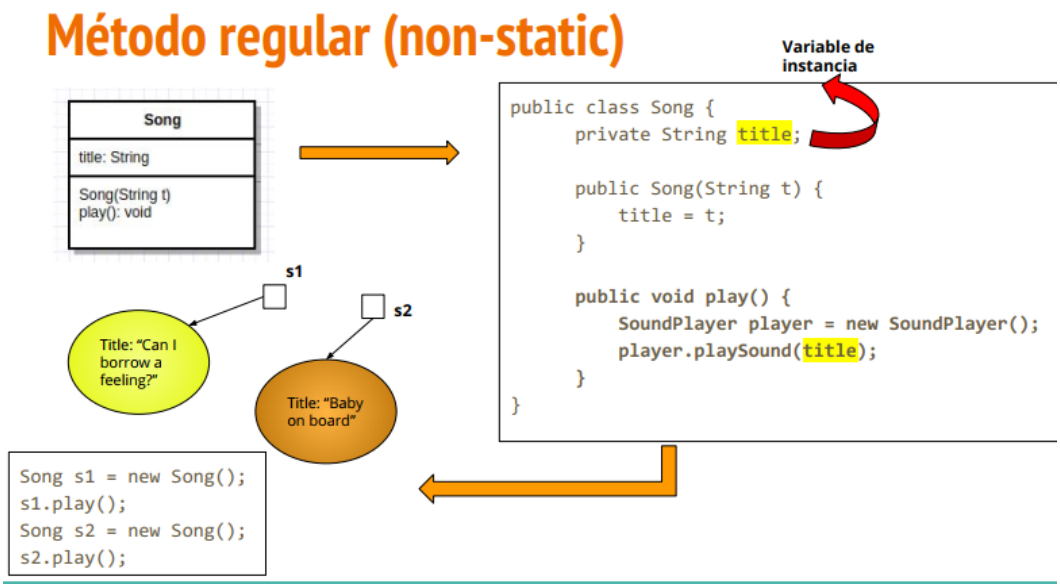
➔ Métodos estáticos: Métodos que no usan valores variables de instancia. No es necesario crear una instancia de la clase para utilizarlos Es aquel que no requiere una instancia para ser ejecutado.

Variables estáticas: son un valor compartido por todas las instancias de una clase, esto quiere decir que si yo creo múltiples instancias de una clase y modifico un valor utilizando alguna de ellas, este valor será modificado en todas las instancias.

```
public static int min(int a, int b) {
    //return a or b
}
```

No hay instancias de objetos

Método regular (non-static):




## Métodos regulares vs. estáticos:

➔ Un método declarado con la palabra reservada static nos indica que se puede invocarlo sin necesidad de crear una instancia de la clase. Es decir, no necesitamos crear un objeto a partir de la clase para llamar a este tipo de métodos.
➔ Se pueden combinar métodos regulares y estáticos en la misma clase.
➔ Los métodos estáticos no pueden usar variables de instancia ya que no sabrían como apuntarlos.
➔ Los métodos estáticos no pueden usar métodos regulares, porque usan variables de instancias.

# Constantes:

Un valor es constante utilizando la palabra final.

➔ La palabra reservada final indica que una vez inicializada, el valor de la variable no puede cambiar. Se les tiene que dar un valor si o si por que sino va a quedar en null.
➔ Generalmente se establecen como public para que puedan ser accedidas desde cualquier lugar de nuestro código. Para poder reutilizarlas.
➔ Son estáticas para que no sea necesario crear una instancia de la clase para poder usarlas.
➔ **EL NOMBRE DE UNA CONSTANTE DEBE ESTAR EN MAYÚSCULA.**

# Final

➔ La palabra reservada final no es sólo para variables estáticas.
➔ Se puede usar final para variables de instancias, variables locales, parámetros de métodos y clases.
➔ Indica que el valor no puede cambiar una vez que fue inicializado.
➔ Una variable final significa que no puede cambiar su valor.
➔ Un método final significa que no puede sobreescribirse.
➔ Una clase final significa que no puede tener subclases.

# OVERRIDE:

La anotación **@Override** en Java es utilizada para: Indicar la sobre escritura de un método. Un ejemplo sería: Cuando se crea una interface para la implementación de un objeto en una clase, esta permite sobreescribir la el comportamiento, funcionalidad del método en la clase que se implementa la interfase

Te permite reescribir el método en una clase hija otorgándole un comportamiento propio.

# Clases abstractas:

Una clase abstracta no es más que una clase común la cual posee atributos, métodos, constructores y por lo menos un método abstracto. Una clase abstracta no puede ser instanciada, solo heredada.

Son aquellas que por sí mismas no se pueden identificar como algo concreto es decir, no existe como tal en un mundo real, pero si poseen determinadas características que son comunes en otras clases que pueden ser creadas a partir de ellas.
La clase abstracta es similar a una clase concreta, posee atributos y métodos pero tiene una condición: si o si al menos uno de sus métodos debe ser abstracto.

## Clase abstracta - Método abstracto

➔ Un método abstracto se caracteriza por dos detalles:
- Está precedido por la palabra clave **abstract**.
- No tiene cuerpo y su encabezado termina con punto y coma.

➔ Si un método se declara como abstracto, se debe marcar la clase como abstracta →No puede haber métodos abstractos en una clase concreta.
➔ Los métodos abstractos deben implementarse en las clases concretas (subclases).
→ @Override

## Clase abstracta - Constructores?

➢ Las clases abstractas no se pueden instanciar!!!
➢ Es posible definir constructores en las superclases, pero no es posible crear Instancias.
Animal animal = new Animal(); (NO SE PUEDE).

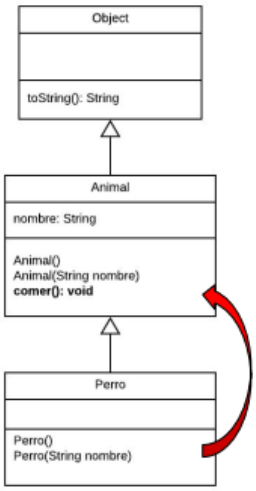
# Super

➔ Se usa para acceder a los atributos y métodos de una clase padre como también inicializar atributos de la clase padre a través del llamado de los métodos de constructores de las clases hijos.
➔ Se utiliza para invocar métodos de la superclase. super.metodo();
➔ En el caso de los constructores, debe ser la primer línea en el constructor de la subclase. super(); ó super(lista de parámetros);
➔ Si el constructor en la subclase no invoca explícitamente al constructor de la superclase, el compilador de Java lo inserta automáticamente.

# Ejecución dinámica de métodos

➔ Los métodos sobrescritos de las subclases tienen precedencia sobre los métodos de las subclases.
➔ La búsqueda del método comienza al final de la jerarquía, entonces la última redefinición de un método es la que se ejecuta primero.

Ejemplo:




**Ejemplo 1:** Se quiere ejecutar el método comer() de la clase Perro.

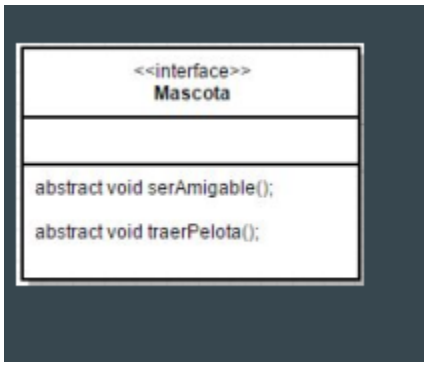
```
public static void main(String [] args) {
   Perro perro = new Perro("Ayudante de Santa");
   perro.comer();
}
```

**Para acceder a los métodos de la clase Perro, es necesario crear una instancia.**

1) Se busca el método **comer()** en la subclase Perro.
2) Como no se encuentra la sobreescritura, se sube en la jerarquía hasta que se encuentra y se ejecuta.

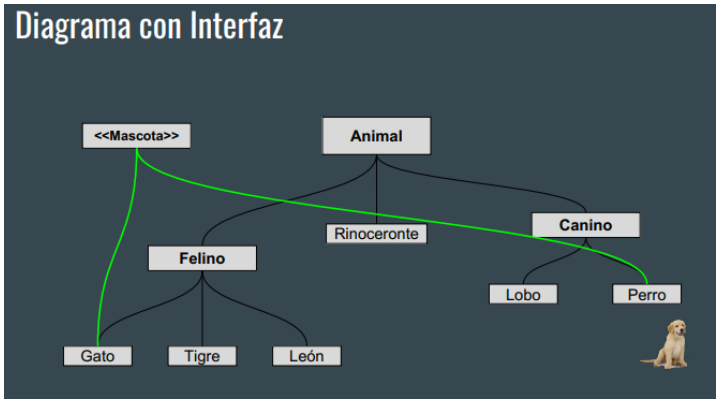
# Interfaces:

➔ Todos los métodos dentro de una interfaz son públicos y abstractos,de esta forma la clase que implemente dicha interfaz debe implementar los métodos.
➔ Con la palabra **Implements** implementamos las interfaces.
➔ Una vez que creamos la interfaz, cuando la vayamos a usar si o si tenemos que implementar estos métodos con un override y reescribirlas.
➔ Las interfaces las escribimos así: <<nombre de la interfaz>>.
➔ cada método que tengamos en la interfaz lo debemos implementar en cada clase que nos indique.
● Las interfaces te permite hacer herencia múltiple.
● Java no te permite hacer herencia múltiple pero lo podes emular con interfaz.




```
package com.utn.learning.interfaces;

public interface Mascota {

    abstract void serAmigable();
    abstract void traerPelota();
}
```

```java
package com.utn.learning.interfaces;

public class Perro extends Canino implements Mascota {

    @Override
    public abstract void serAmigable() {...};

    @Override
    public abstract void traerPelota() {...};
}
```




# Collection Api:

➔ Es un objeto que representa un grupo de objetos.
➔ El framework collection es una arquitectura unificada para representar y manipular colecciones, lo que permite manipular independientemente de los detalles de la implementación.

## Ventajas:

➔ Reduce los esfuerzos de programación al proveer estructuras de datos y algoritmos que no tenemos que escribir.
➔ Provee implementaciones de alta performance.
➔ Fomenta la reutilización del software al proporcionar una interfaz para colecciones y algoritmos para manipularlas.

# Interfaz Collection:

➔ Es la raíz de todas las interfaces y clases relacionadas con colecciones de elementos.
➔ Algunas colecciones permiten elementos duplicados mientras que otras no.
➔ Otras colecciones pueden tener los elementos ordenados mientras que en otras no existe orden alguno.

# ¿Por qué es una interfaz?

➔ Es la manera más genérica para representar un grupo de elementos.
➔ Puede ser usada para pasar colecciones de elementos o manipularlas de la manera más general.
➔ Gracias a esta interfaz, podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección.
➔ Se trata de métodos definidos por la interfaz que obligatoriamente han de implementar las subinterfaces o clases que hereden de ella.

Algunos métodos definidos por ejemplo:

```
- boolean add(E e)
- int size()
- boolean isEmpty()
- boolean remove(E e)
- void clear()
- Object[] toArray()
```

# List:

➔ Es la interfaz encargada de agrupar una colección de elementos en forma de lista, es decir uno detrás de otro.
➔ Acepta elementos duplicados.
➔ Al igual que en los arreglos, el primer elemento está en la posición 0.

Algunos métodos definidos:

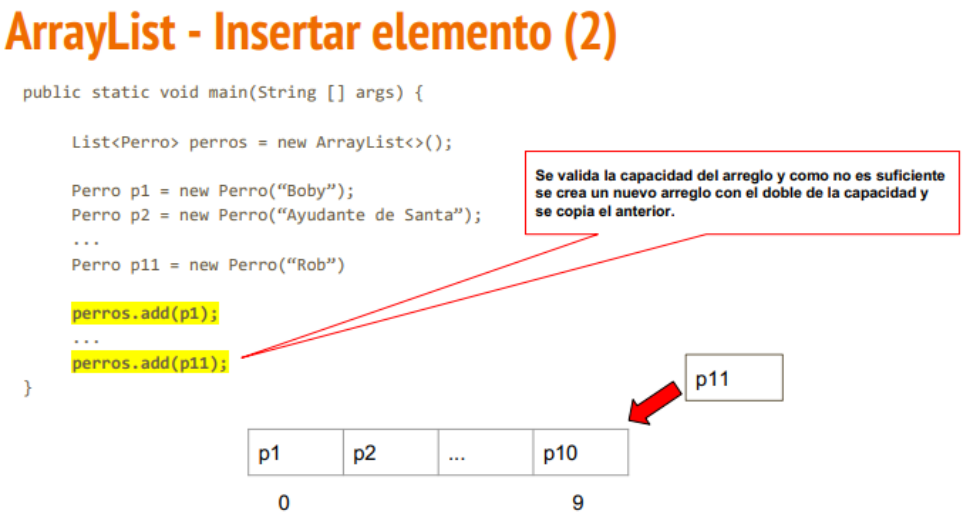
```
- E get(int index)
- E set(int index, E element)
- E add(int index, E element)
- E remove(int index)
- int indexOf(Object o)
- Object[] toArray()
```

# ArrayList:

➔ Basa la implementación de la lista en un array de tamaño variable.

➔ Un beneficio de usar esta implementación es que las operaciones de acceso a elementos, capacidad y saber si está vacía,se realizan de forma eficiente y rápida.
➔ Todo arraylist tiene una propiedad de capacidad, aunque cuando se añade un elemento esta capacidad puede incrementarse.

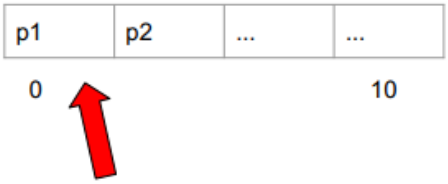
## Insertar elemento en ArrayList


## Lectura de un elemento en ArrayList

Como la estructura interna es un arreglo, se accede a la variable a través del índice correspondiente.

```java
public static void main(String [] args) {

    List<Perro> perros = new ArrayList<>();

    Perro p1 = new Perro("Boby");
    Perro p2 = new Perro("Ayudante de Santa");

    perros.add(p1);

    System.out.println(perros.get(0).getNombre());
}
```




Es lo mismo que: p1.getNombre()

## Eliminar elemento en ArrayList

### ArrayList - Eliminar elemento

1) A través del índice:

```java
public static void main(String [] args) {

    List<Perro> perros = new ArrayList<>();

    Perro p1 = new Perro("Boby");
    Perro p2 = new Perro("Ayudante de Santa");

    perros.remove(0);

}
```

2) A través de igualdad:

```java
public static void main(String [] args) {

    List<Perro> perros = new ArrayList<>();

    Perro p1 = new Perro("Boby");
    Perro p2 = new Perro("Ayudante de Santa");

    perros.remove(p1);

}
```

# Método equals en collection:

➔ El método equals() está estrechamente relacionado con la función hashCode()
➔ Si sobreescribimos equals deberemos de sobrescribir también hashCode.
➔ hashCode debe cumplir que si dos objetos son iguales, según la función equals, debe de dar el mismo valor para ambos objetos.

**Si a.equals(b) entonces a.hashCode() == b.hashCode()**

Ejemplo equals:

```
public class Perro extends Animal {

    ...
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
          return true;
        if (obj == null)
          return false;
        if (getClass() != obj.getClass()
          return false;
        Perro otroPerro = (Perro) obj;
        if (nombre.equals(otroPerro.equals())
           return true;
        return false;
    }
}
```

# Método hashCode:

➔ Este método viene a complementar al método equals y sirve para comparar objetos de una forma más rápida en estructuras Hash ya que únicamente nos devuelve un número entero.

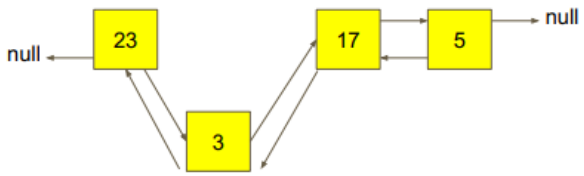
# LinkedList:

➔ Su implementación se basa en una lista doblemente vinculada de tamaño ilimitado.
➔ Al igual que ArrayList, también implementa la interfaz List.
➔ Su estructura está formada por Nodos, cada nodo contiene dos enlaces: uno a su nodo predecesor y otro a su nodo sucesor.

## Insertar elemento en LinkedList:

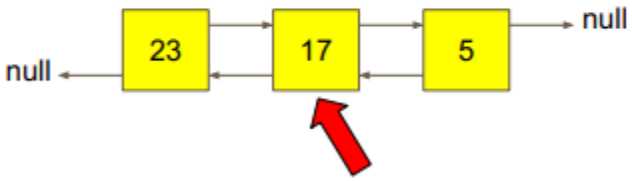
1) Agregar al final → boolean add(E e)


2) Agregar con índice → void add(int index, E element)



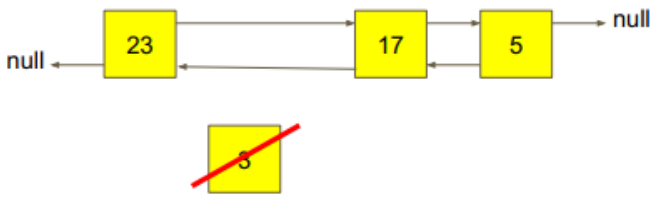
## Leer elemento en LinkedList:

- E get(int index)



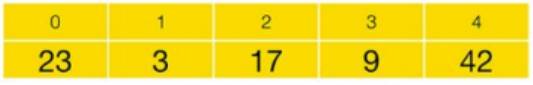

## Eliminar elemento en LinkedList:

- E remove(int index)




# ArrayList vs LinkedList:

➔ ArrayList está basada en una estructura de datos del tipo arreglo, mientras que LinkedList está basada en una lista doblemente vinculada.





➔ Almacenar elementos en un ArrayList consume menos memoria y generalmente es más rápido en tiempos de acceso.
➔ Agregar o eliminar elementos usualmente es más rápido en LinkedList, pero como normalmente se debe iterar hasta la posición en la que se desea agregar o eliminar el elemento, la pérdida de rendimiento a veces es más grande que la ganancia (no siempre).

# Vector:

➔ Un Vector es similar a un array que crece automáticamente cuando alcanza la capacidad inicial máxima.
➔ También puede reducir su tamaño.
➔ La capacidad siempre es al menos tan grande como el tamaño del vector. Es decir, nunca va a poder tener capacidad más grande de lo que es el tamaño.

Vector vector = new Vector(20, 5);

➔ El vector se inicializa con una dimensión inicial de 20 elementos. Si rebasamos dicha dimensión y guardamos 21 elementos la dimensión del vector crece a 25. Cada vez que nos pasemos del tamaño del vector va a crecer 5.

Vector vector=new Vector(20);

- Si se rebasa la dimensión inicial guardando 21 elementos, la dimensión del vector se duplica.

Vector vector=new Vector();

- Se inicializa con dimensión inicial de 10 elementos. La dimensión del vector se duplica si se rebasa la dimensión inicial.

# Vector vs. ArrayList:

➔ Vector es sincronizada mientras que ArrayList no lo es. Si no trabajamos en un entorno multihilos utilizar ArrayList.
➔ La estructura de ambos está basada en array.
➔ Ambos pueden crecer y reducirse en forma dinámica, sin embargo la forma en la que se redimensionan es diferente.

# Stack

Básicamente es una pila.

➔ Representa una estructura LIFO (last in - first out).
➔ Es una subclase de Vector, por lo tanto su estructura también está basada en un array.
➔ Al igual que Vector, Stack está sincronizada.

Operaciones básicas Stack:

- **push** → introduce un elemento en la pila.
- **pop** → saca un elemento de la pila.
- **peek** → consulta el primer elemento de la cima de la pila.
- **empty** → comprueba si la pila está vacía.
- **search** → busca un determinado elemento dentro de la pila y devuelve su posición dentro de ella.