TEORIA PRIMER PARCIAL

Java provee varias herramientas como:

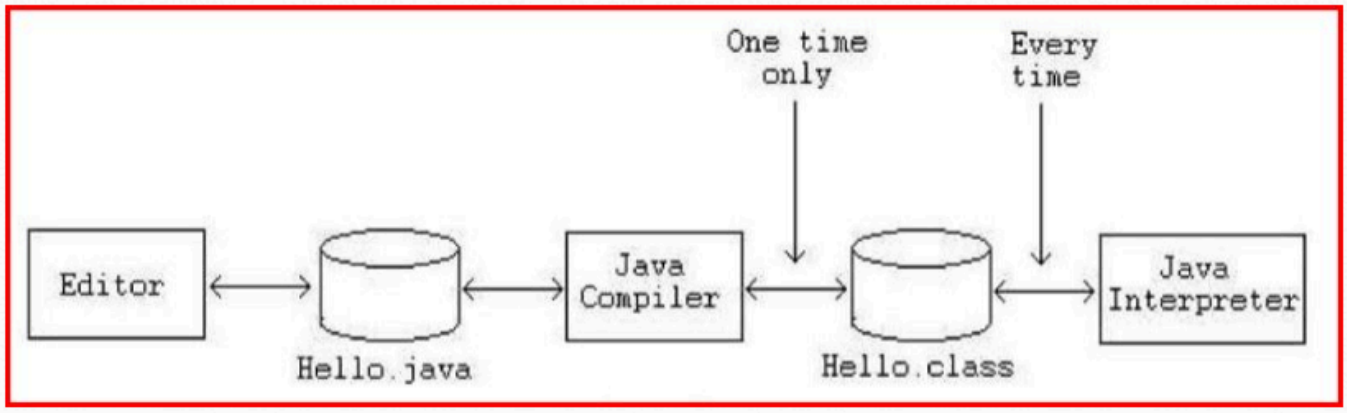
- **JAVAC (Java Compliler)**
  - Transforma el codigo fuente en bytecode. (Hello.java ----> Hello.class)
- **Interprete/JVM (java)**
  - Interpreta y ejecuta los bytescodes del programa.
- **JAVADOC (generador de documentacion)**
- **.JAR (Java ARchive)
  - Junta los ficheros en un archivo, conteniendo los archivos de clase y recursos auxiliares.
  - Esto brinda:
    - **Seguridad**
    - **Compresion**
    - **Control de Version**
- etc.

Para correr una aplicacion de JAVA, la plataforma operativa tiene que tener instalado el **JRE.**

CARACTERISTICAS DE JAVA

- Orientado a objetos
- Independencia de plataforma/portable
- Garbage collector
- Arquitectura robusta / Seguridad
- Lenguaje CASE SENSITIVE

FASES DE UN PROGRAMA



TIPOS DE DATOS BASICOS

Tipo	Tamaño	Clase equivalente
boolean	1 bit	Boolean
char	16 bit	Character
byte	8 bit	Byte
short	16 bit	Short
int	32 bit	Integer
long	64 bit	Long
float	32 bit	Float
double	64 bit	Double

PAQUETES

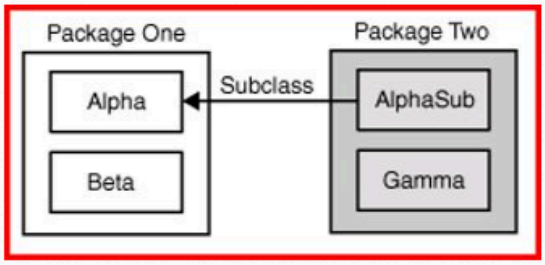
Agrupacion logica de clases bajo un mismo espacio de nombre.

```
package net.programacion.ejemplos;
```

Uso:

```
import net.programacion.net.*;
```

NIVELES DE ACCESO



Modificador	Alpha	Beta	AlphaSub	Gamma
public	SI	SI	SI	SI
protected	SI	SI	SI	NO
default	SI	SI	NO	NO
private	SI	NO	NO	NO

¿COMO ES UN PROGRAMA EN JAVA?

Solo se permite tener **una clase publica por archivo** y tiene que tener el **mismo nombre** que el archivo.
La regla del nombre de archivo solo se aplica si existe una clase pública. Si todas las clases son de tipo '**default**', el nombre del archivo es flexible.
EJ.

Archivo: Primero.java

```
public class Primero{
        public static void main(String[] args){
                System.out.println("Hola mundo");
        }
}
```

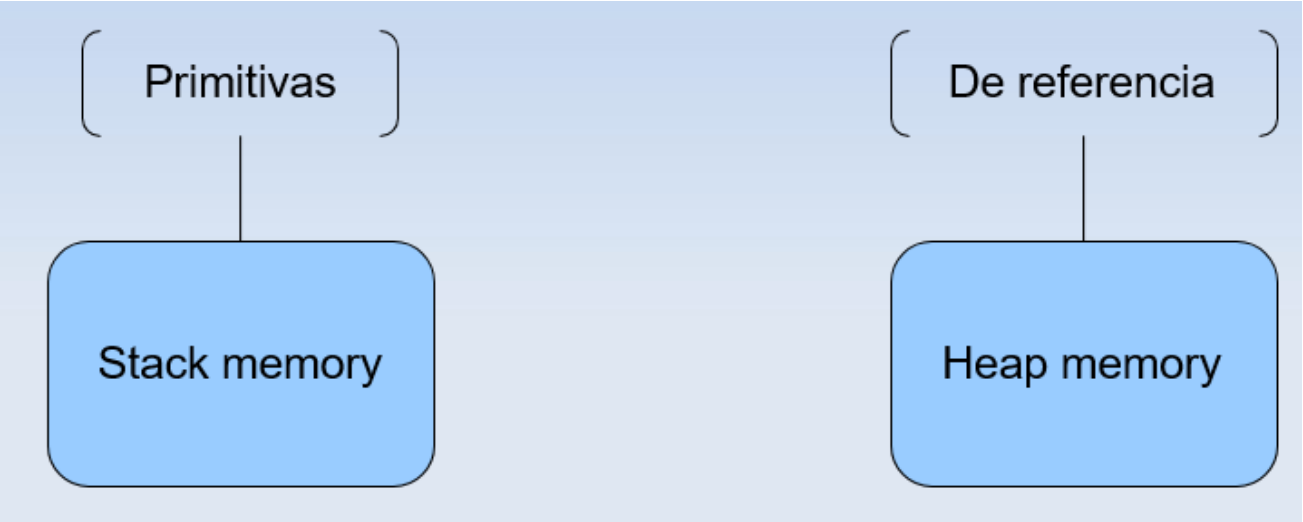
Primero se declara el paquete
Segundo los paquetes importados
y despues el metodo main().

VARIABLES

Hay 2 tipos de variables:

- **Primitivas** (*boolean, int, char,etc)*

    - **Primitivos vs. Clases Equivalentes (Wrappers)**: Cada tipo de dato primitivo tiene una clase "envoltorio" o *wrapper* (ej. `int` -> `Integer`)
    - **Autoboxing y Unboxing**: Java convierte automáticamente entre primitivos y sus wrappers. Ejemplo: `Integer i = 10;` (autoboxing), `int n = i;` (unboxing).
    - **Diferencia clave**: Un wrapper es un objeto, por lo que su valor por defecto es `null`, mientras que el de un primitivo numérico es `0` o `0.0`.
- **de Referencia**: Hacen referencia a objetos.



- **Stack Memory**: Aquí se almacenan las **variables locales** de los métodos (tanto primitivas como las referencias a objetos).
  La memoria se asigna y libera de forma muy rápida y automática siguiendo un orden **LIFO** (Last-In, First-Out) a medida que los métodos son llamados y retornan.
- Heap Memory: se almacenan *todos los objetos creados con `new` y sus variables de instancia. Es un espacio de memoria global para toda la aplicación, y es el área que el Garbage Collector se encarga de limpiar.

CONSTANTES

Para declarar una constante se usa el modificador **final**, y el nombre de la constante va en MAYUSCULA.

ej. `final CONSTANTEFLAG = 32;`

VALORES POR DEFECTO

Cuando no se declara el valor de una variable estos son los valores que toman por defecto:

Variable Type	Default Value
Object reference	`null` (not referencing any object)
`byte, short, int, long`	`0`
`float, double`	`0.0`
`boolean`	`false`
`char`	`'\u0000'`

CASTING

Hay 2 tipos de casting:

- **Casting Implicito:**
  - `int a = 100;`
    `long b = a;` // **casting implícito** porque un int siempre entra en un long.
- **Casting Explicito:**
  - `float a = 100.001f;`
    `int b = (int)a; //` **casting explicito** conlleva posible perdida de precisión.

MODIFICADORES DE ACCESO (Clases)

Una clase puede tener los siguientes modificadores:

- **public**: significa que puede ser **instanciada/extendida** dentro y fuera de su paquete.
- **default**: puede ser **instanciada/extendida dentro de su paquete**. Una clase es default cuando no tiene el modificador de acceso especificado.
- **final**:
  - **En una variable**: La convierte en una **constante**, su valor no puede ser modificado una vez asignado.
  - **En un método**: El método **no puede ser sobrescrito** (`overridden`) por las subclases.
  - **En una clase**: La clase **no puede ser extendida** (no puede tener subclases). La clase `String` es un ejemplo clásico de una clase `final` .
- **abstract**: significa que **no puede ser instanciada pero si extendida** solamente.

- Una clase que tenga un **metodo abstracto** TIENE QUE ser declarada como **abstract** si o si.
- Una clase abstract puede tener todos metodos concretos.
- Una clase `abstract` puede no tener ningún método `abstract`.

MIEMBROS DE UNA CLASE

Son los **atributos y metodos** de una clase.

Modificadores de acceso de los miembros de clase

- **public**: Los miembros pueden ser accedidos y heredades dentro y fuera del paquete.
- **private**: Los miembros pueden ser accedidos dentro de la clase misma.
  No se pueden heredar ni referenciar externamente.
- **protected**: Los miembros de la clase pueden ser accedidos por herencia, no por instanciacion, en cualquier paquete.
- **default**: Lo mismo que protected pero dentro del mismo paquete.

OVERRIDING

-Sobreescribir un metodo heredado para cambiar su comportamiento.
-Es cambiar lo que esta dentro de las llaves { }.
-El modificador de acceso del método sobrescrito en la subclase debe ser **igual o más permisivo** que en la superclase (ej. `protected` puede pasar a `public`, pero `public` no puede pasar a `private`).

-Los metodos sobreescritos tienen que tener **SI O SI la MISMA FIRMA**.
Firma = *nombre , parámetros y tipo de retorno*.

-Un metodo **static** no puede ser sobreescrito.

Ocurre en TIEMPO DE EJECUCION.

OVERLOADING

Ocurre en TIEMPO DE COMPILACION.
El compilador decide QUÉ método llamar basándose en el **número y tipo de parámetros**

Se produce en la **misma clase**.
Consiste en tener varios métodos con el **mismo nombre** pero con **diferentes parámetros** (ya sea en cantidad, tipo u orden).

El tipo de retorno puede cambiar, pero no es suficiente por sí solo para diferenciar los métodos.

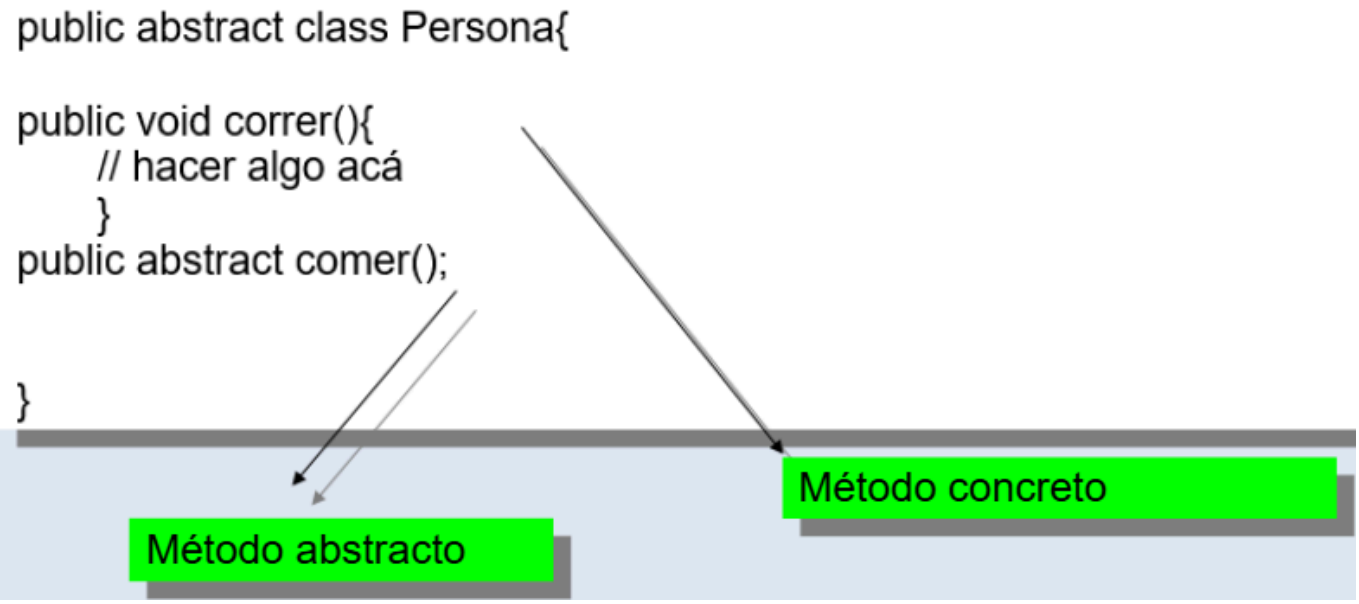
Para sobrecargar un método se puede cambiar el modificador de acceso.

`final`:

- **En una variable**: La convierte en una **constante**, su valor no puede ser modificado una vez asignado.
- **En un método**: El método **no puede ser sobrescrito** (`overridden`) por las subclases.
- **En una clase**: La clase **no puede ser extendida** (no puede tener subclases). Ej. La clase `String` es un ejemplo clásico de una clase `final`.

METODOS ABSTRACTOS

Se usan para que sean sobreescritos por las subclases.

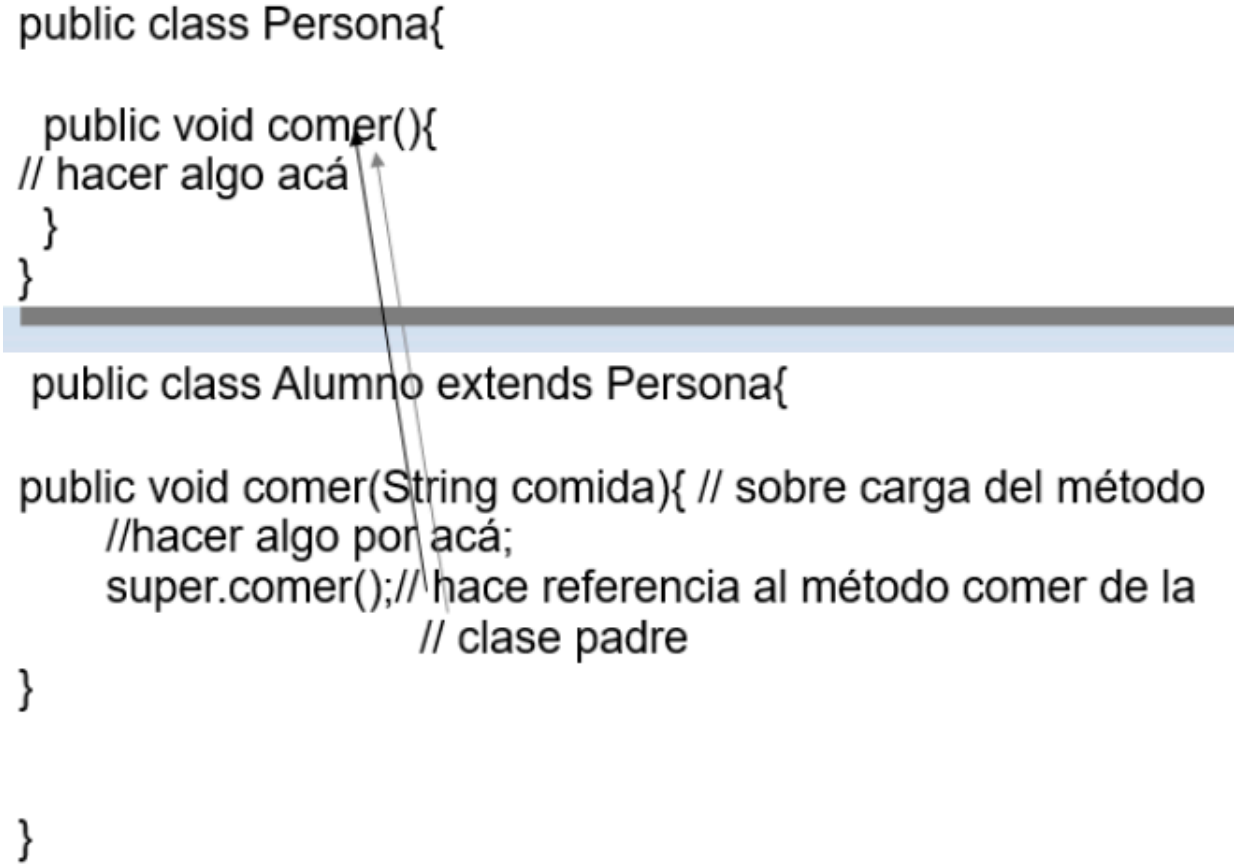


REFERENCIAS THIS Y SUPER

- **this**:
  Es una referencia al **objeto actual** (la instancia sobre la que se está ejecutando el método).
  Uso 1: Diferenciar una variable de instancia de un parámetro de método con el mismo nombre.

  Uso 2: Invocar otro constructor de la misma clase. Se usa como `this(...)` y **debe ser la primera línea** de código en el constructor.
- **super**: Se usa para referenciar a los **miembros**(atributos y/o metodos) y constructores de la superclase en cuestion.
  - **Uso 1**: Acceder a un miembro (atributo o método) de la superclase. Es útil cuando una subclase ha sobrescrito un método y necesita invocar la versión original del padre.
  - **Uso 2**: Invocar el constructor de la superclase. Se usa como `super(...)` y **debe ser la primera línea** de código en el constructor de la subclase. Si no se llama

explícitamente, el compilador inserta una llamada a `super()` (el constructor sin argumentos del padre) de forma implícita.



JavaBean

Bean: es una clase modelo java que cumple con los siguientes requisitos:

1. Todos los atributos **PRIVATE**.
2. **Getters y Setters** para todos sus atributos (todos en public).
3. Tener al menos un **constructor vacio/simple sin parametros**.
4. Que sea **Serializable**. (implements Serializable).

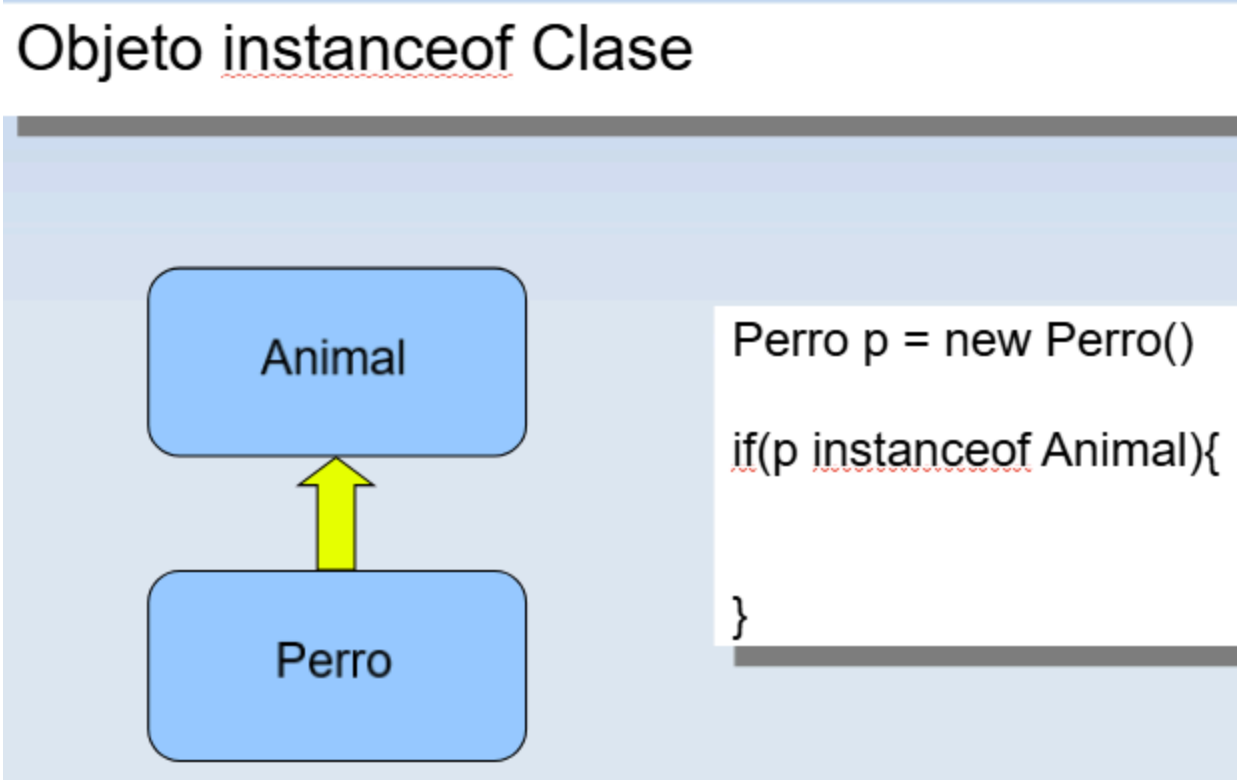
Si bien los getters y setters deben ser públicos, la clase **puede tener métodos privados auxiliares**. Por lo tanto, *no es un requisito que todos sus métodos sean públicos.*

NORMAS DE ESCRITURA EN JAVA

- Codigo en **CamelCase**
- **Clases** empiezan con mayuscula.
- **variables** y **metodos** empiezan con minusculas.
- **paquetes** en minusculas
- **constantes** en mayusculas

instanceof

Es un operador que nos indica si un objeto es de una determinada clase o interfaz.
Retorna un BOOLEANO.



CASTING CON VARIABLES DE REFERENCIA

```
class Animal {  void makeNoise() {System.out.println("generic noise"); }
}
class Dog extends Animal {
  void makeNoise() {System.out.println("bark"); }
  void playDead() { System.out.println("roll over"); }
}
class CastTest2 {
  public static void main(String [] args) {
    Animal [] a = {new Animal(), new Dog(), new Animal() };
    for(Animal animal : a) {
      animal.makeNoise();
      if(animal instanceof Dog) {
        animal.playDead();       //error ?
      }
    }
  }
}
```

```
if(animal instanceof Dog) {

Dog d = (Dog) animal;    // para que compile el programa anterior reemplazar lo rojo con
                         // esto
  d.playDead();
}
```

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

class DogTest {
  public static void main(String [] args) {
    Dog d = new Dog();
    Animal a1 = d;           // upcast ok sin cast explicito
    Animal a2 = (Animal) d;  // upcast ok con cast explicito
  }
}
```

UPCAST impicito y explicito

ENUM

Son enumeraciones que se usan para no declarar muchas constantes, es decir **UN CONJUNTO DE CONSTANTES**.

Funciona como un tipo de dato más.

Pueden ser **public** o **default** solamente

Los '**enum**' deben ser declarados como un *tipo de nivel superior* o como *miembros estáticos de una clase*.

No pueden declararse dentro de un metodo.

El ; es **OPCIONAL**.

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING }

class Coffee {
CoffeeSize size;
}

public class CoffeeTest1 {

public static void main(String[] args) {
                Coffee drink = new Coffee();
            drink.size = CoffeeSize.BIG;
      }
}
```

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

    enum CoffeeSize { BIG, HUGE, OVERWHELMING } //Error no se puede
                                    // declarar enum dentro de
                                    // métodos
        Coffee drink = new Coffee();

        drink.size = CoffeeSize.BIG;

  }
}
```

CONSTRUCTORES

- Los constructores tienen **TODOS los modificadores** (private, public, protected,default).
- NO retornan valor
- El constructor **standard NO tiene argumentos**.
- Las clases **abstractas TIENEN** constructores
- Las **interfaces NO TIENEN** constructores
- Los constructores pueden **llamarse dentro de otro constructor**.

VARIABLES Y METODOS STATIC

Los miembros static pueden ser referenciados/usados sin crear un objeto de la clase antes, es decir que se llama a partir de su clase.

Contexto Estático: Los miembros `static` pertenecen a la **clase**, no a una instancia particular. Existe una sola copia de una variable estática, compartida por todos los objetos de esa clase.
Restricciones:

- Un método `static` no puede acceder a miembros de instancia (no estáticos) directamente, porque no está asociado a ningún objeto.
- Un método `static` no puede usar las palabras clave `this` o `super`.

ej. **Frog.frogCount**

```
class Frog {
   static int frogCount = 0;  //Declara e inicializa
                                          // variable estática
   public Frog() {
     frogCount += 1;  // modifica el valor en el constructor
   }
}
class TestFrog {
   public static void main (String [] args) {
     new Frog();
     new Frog();
     new Frog();
     System.out.print("frogCount:"+Frog.frogCount); //Acceso al la
                                                    // variable estática
   }
}
```

```
class Animal {
  static void dostuff() {
    System.out.print("a ");
  }
}
class Dog extends Animal {
  static void dostuff() {        // es una redefinicion
                                 // no una sobre escritura
    System.out.print("d ");
  }
  public static void main (String [] args) {
    Animal [] a = {new Animal(), new Dog(), new Animal()};
    for (int x = 0; x < a.length; x++)
      a[x].doStuff();            // invoca al método estático
  }
}
```

Ahi vemos que NO ES UNA SOBREESCRITURA sino que una redefinicion.

INTERFACES

Es una colección de **definiciones de métodos** (sin implementaciones) y de **valores constantes**.
Se utilizan para definir un **protocolo de comportamiento** que puede ser implementado por cualquier clase.
Constantes: Todas las variables declaradas en una interfaz son implícitamente `public`, `static` y `final` (constantes), incluso si no se escriben esos modificadores.

Se da mediante la palabra reservada `implements`

Unico caso en donde hay *HERENCIA MULTIPLE*.

```
abstract class Ball implements Bounceable { } // ok no tiene que implementar ningún
                                              //método ya que la clase es abstracta
```

- Las interfaces solo pueden **extender/herenciar otras interfaces** y pueden **extender muchas interfaces** a la vez. (ej. `interface Bounceable extends Moveable, Spherical`)
- Sus metodos siguen las mismas reglas que la sobreescritura.
- Las clases pueden implementar varias interfaces.

- Las clases pueden extender e implementar en la misma declaracion (1.extends y 2.implements)

En *Java*, un interface es un **tipo de dato de referencia**, y puede utilizarse en muchos de los sitios donde se pueda utilizar cualquier tipo (como en un argumento de métodos y una declaración de variables)