Lenguajes de programación

- Un lenguaje formal es un lenguaje cuyos símbolos primitivos y reglas para unir esos símbolos están formalmente especificados.
  - Lenguajes Máquina:
    - De más bajo nivel: secuencias binarias de ceros y unos.
    - Históricamente, los primeros
  - Lenguajes ensambladores:
    - Segunda generación de lenguajes
    - Versión simbólica de los lenguajes máquina
  - Lenguajes de alto nivel:
    - Tercera generación
      - Estructuras de control, variables de tipos
      - Ej: C++, Pascal
  - Lenguajes orientados a problemas: describen la solución, no como conseguirla.
    - Cuarta generación.

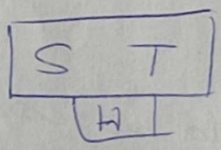
Los lenguajes representan un conjunto de construcción abstractas centradas en resolver problemas más o menos genéricos, en base a una serie de paradigmas de organización del pensamiento y su desarrollo.

- Un traductor es un programa que lee un programa escrito en un lenguaje fuente de alto nivel, y lo traduce a un lenguaje objeto. Dos tipos:
  - Intérprete: ejecuta directamente lo que traduce, sin almacenar en disco la traducción realizada.
  - Compilador: genera un fichero del lenguaje objeto estipulado, que puede posteriormente ser ejecutado tantas veces se necesite sin volver a traducir.
    - El compilador informa de los errores del programa fuente, ya que lo analiza completo

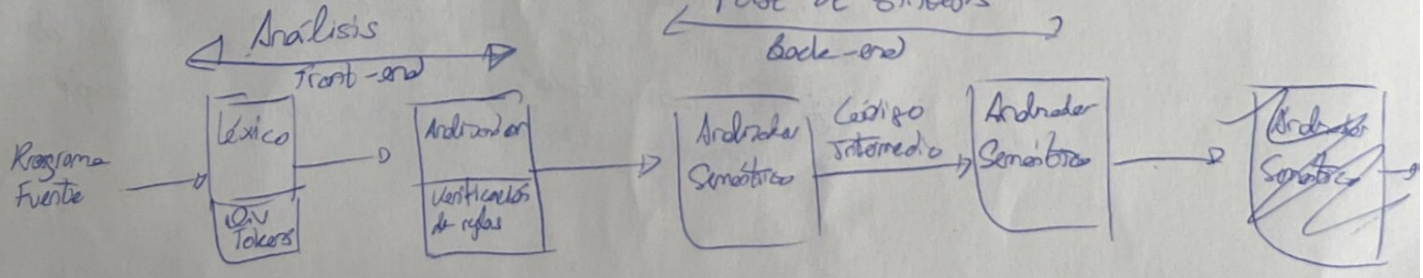
- Intérprete → lo ejecuta según lo va leyendo
  1) Cada vez que escribe una línea el programa comprueba si es correcta
  2) La ejecución es interactiva
  3) Los intérpretes más puros no guardan copia del programa que se está escribiendo

- Compilador → lee un programa escrito en lenguaje fuente, y lo traduce a un lenguaje objeto de bajo nivel. Además generará una lista de los posibles errores que tenga el programa fuente.
  - Más estándar
  - Una vez creado, el programa no necesita el compilador para funcionar.

  Pueden estar implementados en lenguajes de programación distintos del lenguaje fuente (Source (S)) y del objeto (Target (T)). A este tercero se le denomina anfitrión (Host (H))



- Entornos de ejecución:
  - Según implementación:
    - → En metal
    - → Máquina virtual
  - Según tipo de ejecución:
    - → Máquina de pila
    - → Máquina de registros (von Neumann)
- Procesamiento de un lenguaje



→ Programa objeto

¿Qué es GITHUB?

Git es un sistema de control de versiones (VCS) distribuido, lo que significa que es una herramienta útil para rastrear fácilmente los cambios en el código, colaborar y competir. Git permite rastrear los cambios que se realizan en el proyecto facilitando el trabajo en grupo.

Flujo de GITHUB

- **Repositorio**: es el lugar donde se realiza el trabajo del proyecto. Contiene todos los archivos y el historial de revisiones del proyecto.
- **Clonación**: cuando se crea un repositorio con GitHub, se almacena de forma remota en la nube. Se puede clonar un repositorio para sincronizar los datos de la computadora y Git facilitando así la solución de problemas
- **Committing and pushing**: son la forma en la que se puede agregar cambios realizados en la máquina local al repositorio remoto en GitHub.

Términos GITHUB

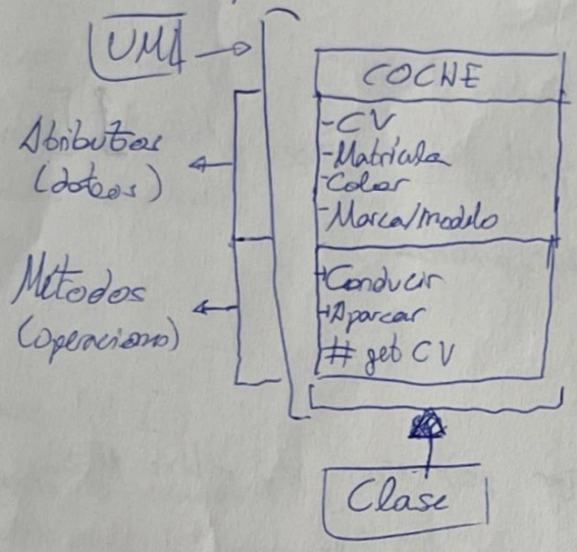
- **Repositorios**: los ya mencionados repositorios, son el lugar donde se lleva a cabo todo el trabajo del proyecto. Para poder navegar fácilmente entre los diferentes repositorios se usa el panel de GitHub. A estos se les puede añadir además un archivo Readme para mostrar a los usuarios las diferentes funciones del programa
- **Ramas**: las ramas se pueden usar para aislar el trabajo que aún no se desea fusionar con el proyecto principal. A menudo se puede crear una rama predeterminada del repositorio Main esto crea una copia del trabajo para poder experimentar con ella.
- **Forks**: son bifurcaciones que permiten copiar un repositorio, frecuentemente usados cuando se participa en un proyecto ajeno. Todo esto sin modificar el proyecto original

- Solicitudes de incorporación de cambios: al trabajar con otras se puede usar una solicitud de incorporación de cambios para informar a otros sobre los cambios deseados además de recibir feedback de otros usuarios.
- Incidencias: son una forma de realizar un seguimiento de las mejoras, tareas o errores del trabajo GitHub. Sirven para poder llevar a cabo un seguimiento de todas las tareas en las que se desea trabajar.

POO (Programación orientada a objetos)

- Abstracción:
  - Es modular para la reutilización de código
  - Herencia para generalización
  - Polimorfismo → dos objetos distintos de tipos distintos puedo tratarlos de forma general. Ej: deportivo y 4x4

- Para poder manejar todo esto apareció el UML, para poder describir el proceso completo del ~~software~~ ingeniería del software. (hacerlo comprensible) (es un tipo de lenguaje gráfico)



En POO tenemos clases e instancias pero no "objetos". Para ejecutar una clase se ~~denota~~ denota el término eventos

Existen tres tipos de niveles de acceso:

- − → Privado
- + → Público
- # → Protegido

Trabajaremos con tipos de datos: string, ...

Java

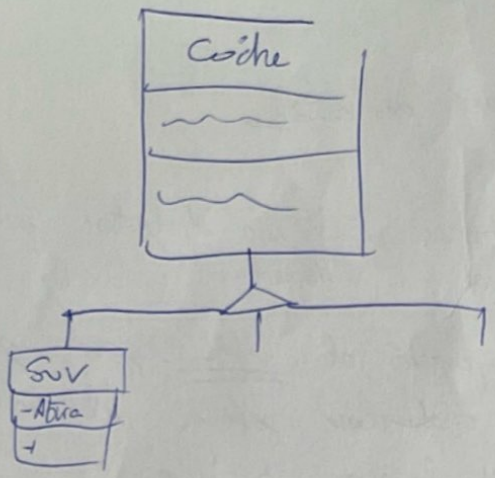
```
class coche [
    int CV
    string Matrícula
    int Color
    string Marca/modelo
coche( ) [
]
coche (int pCV, string pmatrícula,
       int pColor, string pMarca/modelo)
]
```

parámetro

```
~coche( ) [
]
```
sin condición

```
public void conducir( )[ ]
public void aparcar( )[ ]
public protected int getCV[
    return this.CV
]
```

- añadir a una clase más cosas → Sobrecarga
- El primer paréntesis es un constructor por defecto (sin ningún parámetro) Sirve para hacer que funcione lo de los corchetes, inicializar las variables y poderlas operarlas
- ~ sirve para eliminar un destructor



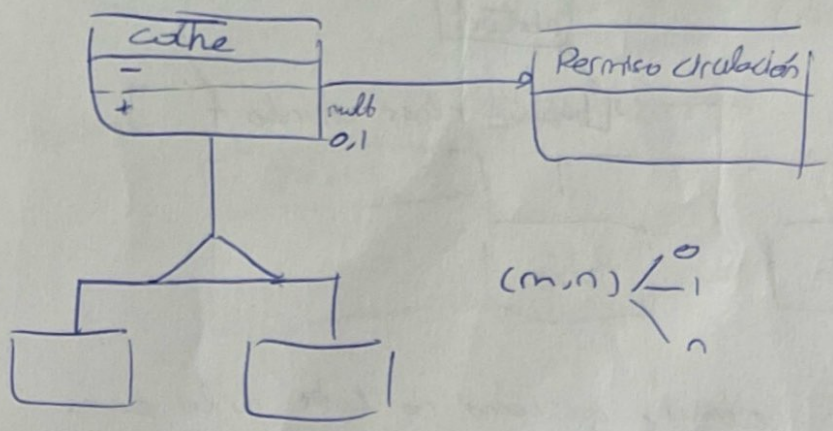

```
class SUV extends coche [
    integer Altura;
    public SUV ( integer pCV , string pMod, . . . , integer pAltura) [
        super ( pCV, pMod, pColor, pMarca)
        this . Altura ; pAltura ;
    ]
public boolean car car subir(integer ps) return ps ≤ this. Altura
]
```

```
class ejemplo coche [                    ← esta clase tiene prioridad antes las demás
    public static void main (string Args [])
    SUV micoche = new SUV ( 200, '9999 777', 1, 'Laus', 18);
    SUV tucoche = new SUV (400, '1111 BBB', 2, 'mercedes', 12)
    if ( micoche.consubir(22)). 17);
        system . out.print ('puede subir')
                 printear
```

Ejercicio

Diseñar un sistema para gestionar estudiantes profesores y carreras

POO - UML



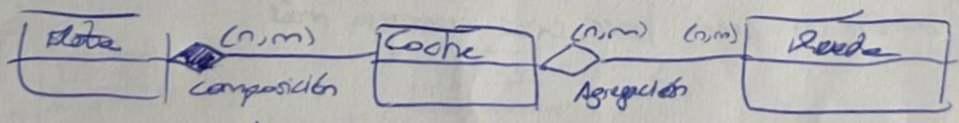

```
class mult [
  data rd;
  data rd = null;
]
```

```
class data [
  pull nombre = "Antonio";
]
```

```
class mult 0,n [
  lista rd = new lista[];
]
```

```
class 1,1 [
  data rd = new data();
  public mult 1_1 (data el) [
  rd = el
]
```

```
class mult 1,n [
  lista rd = new lista();
  public mult 1n ( el ) {
    rd add (el);
  }
```

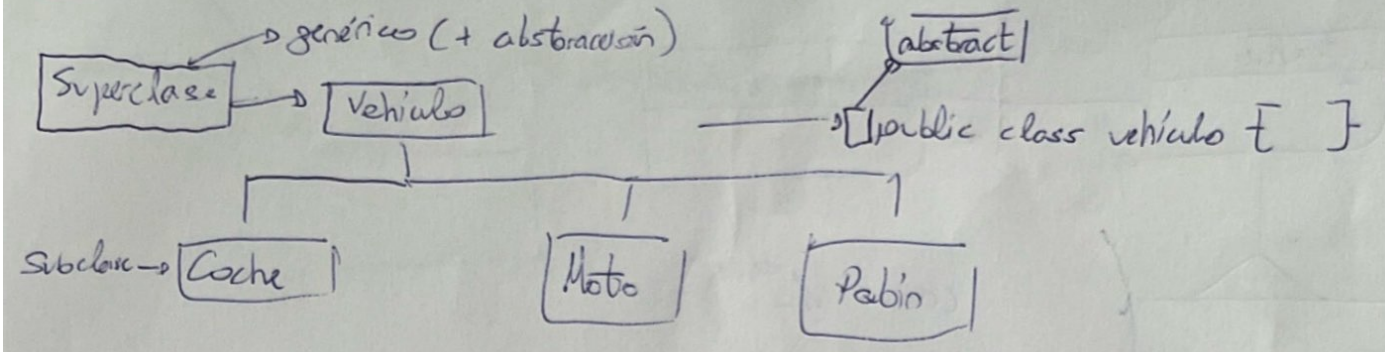


Rueda no puede sobrevivir sin coche

→ agregación, los obj se enlazan para crear otro más complejo

→ composición, representa relaciones de partes enteras y es una forma de agregación.

Abstract y final

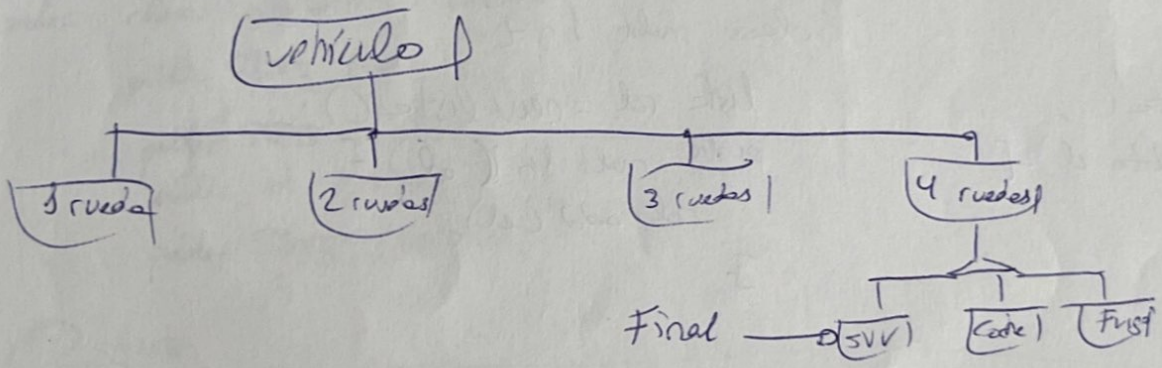


El abstract denota un concepto genérico, que como no existe no lo puedes instanciar

✗ vehículo v = new vehículo ();

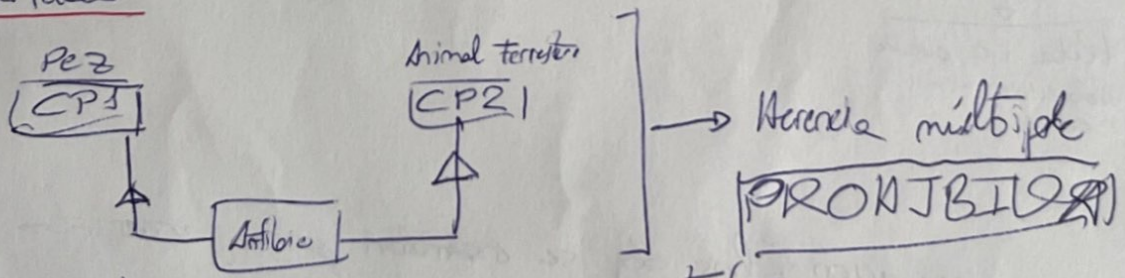
Se pueden programar métodos para todas las subclases a partir de ella

✓ Vehículo v = new SUV();

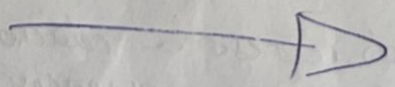


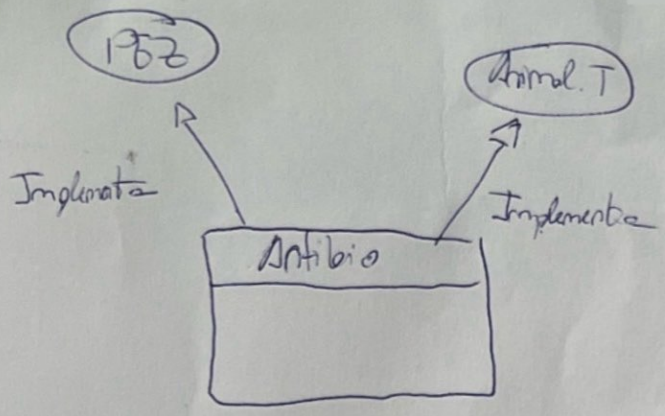
Una clase es final cuando no puedo hacer una jerarquía más

Interfaces



Para evitar esto se usa la interfaz





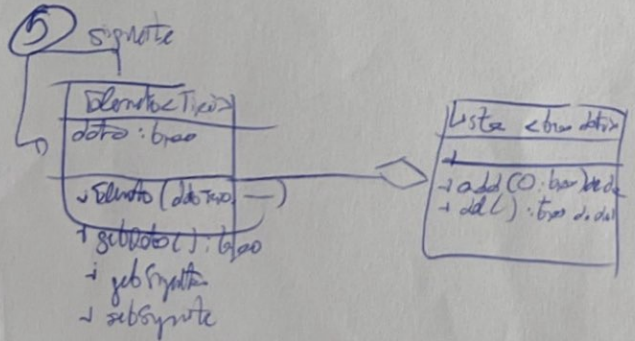

```
public interface PEZ {

    public String tipo( );
    public string toString( );

}
```

```
public interface Animal {

    public int getNombre( );

}
```

son estas no hay código

La interfaz funciona como una caja vacía que se rellena con la implementación de los métodos de las interfaces (todos)

```
public class anfibio implements Pez, Animal {
    public String tipo( ) { }
    public string toString( ) { }
    public int getNombre( ) { }
    public String obtenerEstado( ) { }
}
```



1 | 2 | 2.2

a) | b) |

③

```
package exemplo {
    public class {
        double lado;
        private quadrado() {}
        public quadrado (double lado){
            this.lado = lado;
        }
        public double getPerim(){
            return . this.lado * 4
        public double getArea(){
            retur this.lado * lado
        }
    }
}
```

④

```
public abstract class Ventana {
        public abstract void mover();

public class Ventana extends Janela {
        integer ~~area~~ dimensão;
        @Override
        void mover(){
        }
        void cerrar(){
        }
```