Universidad De San Carlos De Guatemala
Facultad De Ingeniería
Escuela De Ciencias Y Sistemas
Lenguajes Organización de Lenguajes y Compiladores 1
Catedrático: Ing. Mario Bautista
Tutor académico: David Pascual

PRACTICA 1 - PROYECTO 1

OBJETIVOS:

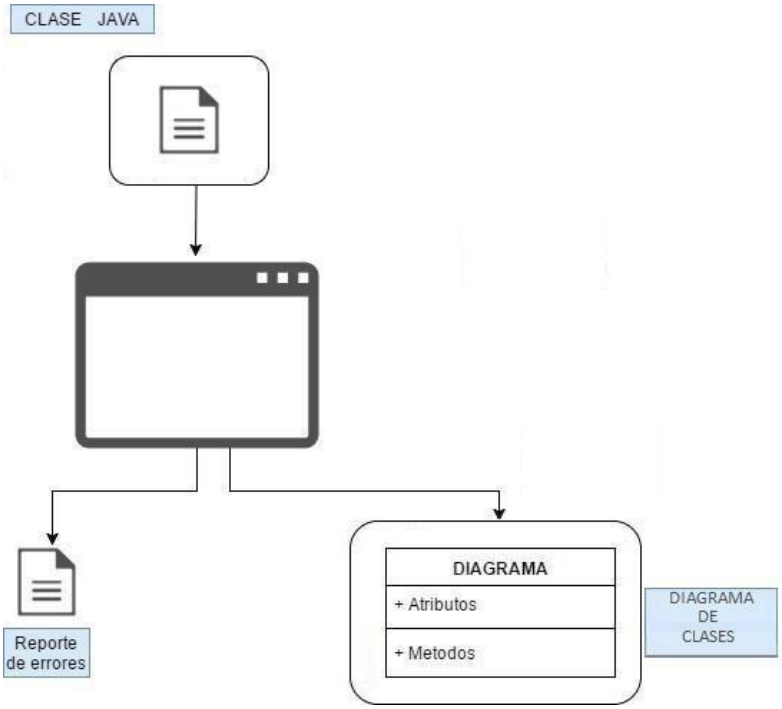
- Que el estudiante practique el desarrollo del análisis léxico y análisis sintáctico utilizando las herramientas Flex y Bison.
- Elaborar una aplicación que ayude a documentar un proyecto en un lenguaje orientado a objetos.

DESCRIPCIÓN DEL PROBLEMA:

La empresa de desarrollo de software OLCEE se ha dado cuenta que al momento de desarrollar un proyecto de software por medio de un lenguaje orientado a objetos, es de vital importancia que el desarrollador documente el proyecto en sí. Uno de los diseños que el desarrollador debe realizar para la comprensión del comportamiento del código es el Diagrama de Clases. Para ello OLCEE se ha propuesto crear un programa que reciba como entrada un archivo el cual contenga uno o más clases en lenguaje Java y a partir de este archivo se debe generar el diagrama de clases respectivo.

La herramienta debe estar en la capacidad reconocer una o más clases en un mismo archivo. **Se podrá crear archivos con extensión .java el cual podrá contener una o varias clases Java y posteriormente generar el diagrama de clases respectivo, se recomienda utilizar el componente treeview para una mejor visualización los archivos.** El programa permitirá eliminar archivos, editarlos y guardar las modificaciones. Cuando el contenido de un archivo haya sido analizado de manera satisfactoria, se procederá a realizar el Diagrama de Clases y mostrarlo dentro de la aplicación. **Se tomaran en cuenta los paradigmas de la programación orientada a objetos.**

Topología de la solución:



DESCRIPCIÓN DEL LENGUAJE:

El lenguaje a analizar será Java, del cual se reconocerán solo las sentencias básicas del lenguaje.

Para una mejor compresión en la descripción, se usara una notación de colores para definir los diferentes tipos de sentencias, la cual se describe a continuación:

COLOR	DESCRIPCIÓN
NARANJA	Opcional, esta parte de la estructura puede o no venir en las sentencias del lenguaje. La parte opcional será encerrada dentro de corchetes "[]".
NEGRO	Obligatorio, esta parte de la estructura es obligatorio que aparezca en la sentencia.
AZUL	Palabras reservadas que pertenecen al lenguaje.
VERDE	Ejemplos.

Creación de clases:

Esta se crea a partir de la instrucción *class*, si una clase no trae ninguna opción de acceso la clase será tomada como clase *public*. En un archivo pueden venir una o más clases, la sintaxis para crear una clase es la siguiente:

```
[ public | private | protected] class <id> [extends <id>] {
        <INSTRUCCIONES_CLASE>
}

[ public | private | protected] class <id> [extends <id>] {
        <INSTRUCCIONES_CLASE>
}
```

Ejemplo 1:

```
private class Auto {
        <INSTRUCCIONES_CLASE>
}
```

Ejemplo 2:

```
protected class Ferrari extends Auto {
        <INSTRUCCIONES_CLASE>
}
```

Ejemplo 3:

```
class Rectangulo extends Figura{
        <INSTRUCCIONES_CLASE>
}

class Rectangulo2 extends Rectangulo {
        <INSTRUCCIONES_CLASE>
}
```

Atributos:

Cada clase puede o no tener declarado dentro de su estructura atributos, los cuales están sometidos a restricciones de acceso. Si un atributo no trae opción de acceso, el atributo será tomado como *public*. Un atributo podrá ser inicializado o no al declararse **(no existen arreglos)**.

Los tipos de datos permitidos al momento de declarar un atributo son los siguientes:

- int
- String
- booleanean
- double
- char
- objeto **(instancia de las demás clases existentes en el proyecto)**

Las variables de tipo objeto pueden reservar memoria con la llamada al constructor de la clase instanciada con la siguiente sintaxis **new <ID> ([<PARAMETRO1>,…, <PARAMETROn>]);** o tener un valor de nulo con **null.**

La sintaxis para declarar un atributo es la siguiente:

```
[public | private | protected] <TIPO> <id> [,<id_1>, …, <id_n>] [= <EXPRESION>];
```

Ejemplo 1:

```
String  cad;
```

Ejemplo 2:

```
public int val_1,val_2,val_3 = 10+30*50-persona.getEdad();
```

Ejemplo 3:

```
private Nodo nodo1 = new Nodo("S");
```

Ejemplo 4:

```
Nodo nodo1 = null;
```

Constructores:

Cada clase puede o no tener declarado dentro de su estructura su constructor, los cuales están sometidos a restricciones de acceso. Si un constructor no trae opción de acceso, será tomado como *public*. La sintaxis para declarar un constructor es la siguiente:

```
[public | private | protected] <id> ([<TIPO> <id_1>, <TIPO>,…, <TIPO> <id_n>]) {
        <INSTRUCCIONES_CONSTRUCTOR>
}
```

Ejemplo 1:

```
private Nodo(){
        <INSTRUCCIONES_CONSTRUCTOR>
}
```

Ejemplo 2:

```
public Nodo(String  nombre){
        <INSTRUCCIONES_CONSTRUCTOR>
}
```

Métodos y funciones:

Cada clase puede o no tener declarado dentro de su estructura métodos o funciones, los cuales están sometidos a restricciones de acceso. Si un método o función no trae opción de acceso, será tomado como *public*. La sintaxis para declarar una función o método es la siguiente:

Funciones

```
[public | private | protected] <TIPO> <id> ([<TIPO> <id_1>,…, <TIPO> <id_n>]) {
        <INSTRUCCIONES_FUNCION>
}
```

Ejemplo 1:

```
private Nodo getSiguiente(){
        <INSTRUCCIONES_FUNCION>
}
```

Ejemplo 2:

```
protected String  mostrar(){
        <INSTRUCCIONES_FUNCION>
}
```

Ejemplo 3:

```
double Sumar_Restar(int x, int y, double var1, double var2){
        <INSTRUCCIONES_FUNCION>
}
```

Métodos

```
[public | private | protected] void <id> ([<TIPO> <id_1>, …, <TIPO> <id_n>]) {
        <INSTRUCCIONES_METODO>
}
```

Ejemplo 1:

```
private void Buscar(){
        <INSTRUCCIONES_METODO>
}
```

Ejemplo 2:

```
Protected void Llenar(int x, int y, booleanean bandera){
        <INSTRUCCIONES_METODO>
}
```

Ejemplo 3:

```
void Ejecutar(Nodo padre){
        <INSTRUCCIONES_METODO>
}
```

Sobrescritura de métodos o funciones de la clase padre:

Cada clase que hereda de otra puede o no sobrescribir algún método o función de la clase padre. Si un método o función no trae la opción de acceso será tomado como *public*. La sintaxis para sobrescritura de métodos o funciones de la clase padre es la siguiente:

```
@Override
[public | protected] [ void | <TIPO> ] <id> ( [<TIPO> <id_1>, …, <TIPO> <id_n>] ) {
        <INSTRUCCIONES_CONSTRUCTOR>
}
```

Ejemplo 1:

```
@Override
private Nodo getSiguiente(){
        [<INSTRUCCIONES_FUNCION>]
}
```

Ejemplo 2:

```
@Override
void Ejecutar(Nodo padre){
        [<INSTRUCCIONES_METODO>]
}
```

Acceso a atributos y métodos o funciones de un objeto:

Dentro del lenguaje Java se encuentran el acceso de atributos y métodos o funciones, si deseamos hacer referencia a los atributos de la clase en la que nos encontramos basta con llamarlos por su nombre propio, o bien utilizar la palabra reservada *this.* seguida del nombre del atributo. Cuando deseamos acceder a los atributos y métodos o funciones de un objeto instanciado debemos utilizar el nombre del objeto seguido por punto . y el nombre del atributo, método o función al que deseamos acceder.

La sintaxis para el acceso a atributos y métodos o funciones de un objeto es la siguiente:

```
[this.] <id> [.id1.id2 … .idn ] [ ( [<PARAMETRO1>, …, <PARAMETROn>] ) ]
```

Ejemplo 1:

```
cad = this.cadenaGlobal;
```

Ejemplo 2:

```
this.0bjeto1.atributo = objeto.suma(20,30);
```

Ejemplo 3:

```
nodo1.nombre = this.arbol.raiz.hijo.getNombre();
```

Comentarios:

El lenguaje como Java permite dos opciones para comentar en el código. La sintaxis para realizar los comentarios es la siguiente:

Comentarios de una sola línea

```
// CUALQUIER_TEXTO menos salto de línea> <SALTO_LINEA>
```

Comentarios de una sola línea

```
/** <CUALQUIER_TEXTO> **/
```

Estructura de control IF (ELSE IF y ELSE):

El lenguaje será capaz de analizar la estructura de contol if (else-if y else), la cual puede aparecer solamente dentro de la estructura de un método o función. El lenguaje está en la capacidad de soportar estructuras de control o ciclos anidados. La estructura de la condición if es la siguiente:

```
if ( <CONDICION1> ) {
        <INSTRUCCIONES_IF>
} [else if ( <CONDICION2> ){
        <INSTRUCCIONES_ELSE_IF>
}] [else if ( <CONDICION3> ){
        <INSTRUCCIONES_ ELSE_IF>
}]… [else if ( <CONDICION_N> ){
        <INSTRUCCIONES_ ELSE_IF>
}] [else {
        <INSTRUCCIONES_ELSE >
}]
```

Ejemplo 1:

```
if ( 5> 3){
        <INSTRUCCIONES_IF>
}
```

Ejemplo 2:

```
if ( nodo.nombre == ”S”){
        <INSTRUCCIONES_IF >
} else if ( nodo.nombre == ”E”){
        <INSTRUCCIONES_ELSE_IF>
} else if ( nodo.nombre == ”T”){
        <INSTRUCCIONES_ELSE_IF >
}
```

Ejemplo 3:

```
if ( nodo.nombre == ”S”){
        <INSTRUCCIONES_IF >
} else if ( nodo.nombre == ”E”){
        <INSTRUCCIONES_ELSE_IF>
} else if ( nodo.nombre == ”T”){
        <INSTRUCCIONES_ELSE_IF >
} else {
        <INSTRUCCIONES_ELSE>
}
```

Estructura de control Switch:

El lenguaje será capaz de analizar la estructura de control switch, la cual puede aparecer solamente dentro de la estructura de un método o función. El lenguaje está en la capacidad de soportar estructuras de control o ciclos anidados. La estructura de control swith es la siguiente:

```
switch ( <EXPRESION> ) {
        case <VALOR1>:
                [<INSTRUCCIONES>]
                break;
        [case <VALOR2>:
                [<INSTRUCCIONES>]
                break; ]
        default:
                [<INSTRUCCIONES>]
                break;
}
```

Ejemplo 1:

```
switch ( getNombre() ){
        case “Hola mundo”:
                [<INSTRUCCIONES>]
                break;
        default:
                [<INSTRUCCIONES>]
                break;
}
```

Ejemplo 2:

```
switch ( a ){
        case “Hola mundo”:
                [<INSTRUCCIONES>]
                break;
        case “Adios mundo”:
                [<INSTRUCCIONES>]
                break;
        default:
                [<INSTRUCCIONES>]
                break;
}
```

Ejemplo 3:

```
switch ( this.nodo.nombre ) {
        case “S”:
                [<INSTRUCCIONES>]
                break;
        case “E”:
                [<INSTRUCCIONES>]
                break;
        case “T”:
                [<INSTRUCCIONES>]
                break;
        default:
                [<INSTRUCCIONES>]
                break;
}
```

Ciclo while:

El lenguaje será capaz de analizar el ciclo while, el cual puede aparecer solamente dentro de un método o función. El lenguaje está en la capacidad de soportar estructuras de control o ciclos anidados. La estructura del ciclo while es la siguiente:

```
while ( <CONDICION> ){
        <INSTRUCCIONES>
}
```

Ejemplo 1:

```
while ( nodo.hijo1.getSiguiente() != null){
        <INSTRUCCIONES_WHILE>
}
```

Ejemplo 2:

```
while (variable1 < 10 || variable2 > 20+suma(2,4)){
        <INSTRUCCIONES_WHILE>
}
```

Ciclo do-while:

El lenguaje será capaz de analizar el ciclo do-while, el cual puede aparecer solamente dentro de un método o función. El lenguaje está en la capacidad de soportar estructuras de control o ciclos anidados. La estructura del ciclo do-while es la siguiente:

```
do{
        <INSTRUCCIONES_DO_WHILE>
} while ( <CONDICION> );
```

Ejemplo 1:

```
do{
        <INSTRUCCIONES_DO_WHILE>
} while ( nodo.siguiente != null );
```

Ejemplo 2:

```
do{
        <INSTRUCCIONES_DO_WHILE>
} while ( variable1 < 10 );
```

Ciclo for:

El lenguaje será capaz de analizar el ciclo for el cual solo utilizará contadores de tipo *int* o *double*, el ciclo for puede aparecer solamente dentro de un método o función. El lenguaje está en la capacidad de soportar estructuras de control o ciclos anidados. La estructura del ciclo for es la siguiente:

```
for ( <DECLARACION_ASINGANACION> ; <CONDICION>; <ASIGNACION> ) {
        <INSTRUCCIONES_FOR>
}
```

Ejemplo 1:

```
int a;
for ( a=0; a<=10; a++){
        <INSTRUCCIONES_FOR>
}
```

Ejemplo 2:

```
for ( int a=10; a>=0; a= a-1){
        <INSTRUCCIONES_FOR>
}
```

Ejemplo 3:

```
for (objeto.contador =10; objeto.contador >=0; objeto.contador--){
        <INSTRUCCIONES_FOR>
}
```

Sentencia break:

El lenguaje será capaz de analizar la sentencia break **la cual solo podrá encontrarse en cualquier ámbito de algún ciclo o la estructura de control switch.** La estructura de la sentencia break es la siguiente:

break;
Ejemplo 1: `break;`

Sentencia continue:

El lenguaje será capaz de analizar la sentencia **la cual solo podrá encontrarse en cualquier ámbito de algún ciclo.** La estructura de la sentencia continue es la siguiente:

continue;
Ejemplo 1: `continue;`

Declaración de variables locales:

Una variable local podrá ser inicializada o no al declararse **(no existen arreglos)**. Las variables de tipo objeto pueden reservar memoria con la llamada al constructor de la clase instanciada con la siguiente sintaxis **new <ID> (** [<PARAMETRO1>, …, <PARAMETROn>] **);** o tener un valor de nulo con **null.**

La sintaxis para declarar una variable es la siguiente:

<TIPO> <id> [,<id_1>, <id_2, …, <id_n>] [= <EXPRESION>]**;**

Ejemplo 1:
`String cad;`
 	
Ejemplo 2:
`int val_1,val_2,val_3 = 10+30*50;`
 	
Ejemplo 3:
`Nodo nodo1 = null;`
 	
Ejemplo 4:
`Nodo nodo1 = new Nodo("E");`

Asignaciones:

El lenguaje será capaz de analizar asignaciones, la cuales pueden aparecer solamente dentro de un método o función. Las asignaciones a variables de tipo objeto pueden reservar memoria con la llamada al constructor de la clase instanciada con la siguiente sintaxis **new <ID> (** [<PARAMETRO1>, …, <PARAMETROn>] **);** o tener un valor de nulo con **null**.

La sintaxis para asignación de variables es la siguiente:

<VARIABLE> = <VALOR>;
Ejemplo 1: `this.cad = "Hola " + "mundo!";`

Ejemplo 2:

```
0bjeto1.atributo = 10+30*50+suma(20,30);
```

Ejemplo 3:

```
nodo1 = new Nodo("S");
```

Ejemplo 4:

```
this.atributo.nodo1 = null;
}
```

Asignaciones por operador de incremento o decremento:

El lenguaje será capaz de analizar asignaciones por operador de incremento (++) o decremento (--), la cuales pueden aparecer solamente dentro de un método o función.

La sintaxis para asignación de variables es la siguiente:

<VARIABLE> <INCREMENTO_O_DECREMENTO>;

Ejemplo 1:

```
this.contador--;
```

Ejemplo 2:

```
0bjeto1.atributo++;
```

Ejemplo 3:

```
variable--;
```

Ejemplo 4:

```
this.atributo.contador++;
```

Llamada métodos de un objeto:

Dentro de las sentencias del lenguaje se puede hacer llamada a los métodos, si deseamos hacer referencia a los métodos o funciones de la clase en la que nos encontramos basta con llamarlos por su nombre propio y sus parámetros necesarios. Cuando deseamos acceder a los métodos de un objeto instanciado debemos utilizar el nombre del objeto seguido por punto "." y el nombre método al que deseamos acceder y sus parámetros necesarios.

La sintaxis para el acceso a atributos y métodos de un objeto es la siguiente:

[this.] <id> [.id1.id2 … .idn] ([<PARAMETRO1>, …, <PARAMETROn>]) ;
Ejemplo 1: **Llamada();** **Ejemplo 2:** **Iniciar("Hola", 1+2, 20.5, false);** **Ejemplo 3:** **this.atributo.llamada("Parametro1", 2+10);** **Ejemplo 4:** **arbol.raiz.hijo.setNombre("S");**

Expresiones lógicas, relacionales y aritméticas:

Estas expresiones se construyen a partir de operadores lógicos, relacionales y aritméticos, los operados admitidos son los siguientes:

OPERADORES LOGICOS	
Operador	**Descripción**
&&	AND, suma lógica
||	OR, multiplicación lógica
!	NOT, negación
OPERADORES RELACIONALES	
<	Menor que
>	Mayor que
<=	Menor o igual
>=	Mayor o igual
==	Igual
!=	Diferente

OPERADORES ARITMETICOS	
+	Suma
-	Resta
*	Multiplicación
/	División
SIGNOS DE AGRUPACION	
()	Paréntesis

DATOS IMPORTATES:

- El lenguaje a analizar es lenguaje Java, por lo que las sentencias antes descritas están sometidas a todas las restricciones que permite el lenguaje (**acepto las sentencias break, continue y return las cuales deberán de ser reportadas como error sintáctico si no vienen en el ámbito correcto**).
- La llamada de variables, métodos, funciones, acceso a atributos o métodos de clases u objetos instanciados pueden venir en cualquier parte del código permitido.
- Dentro de las instrucciones de flujo y ciclos, pueden venir cualquier cantidad de estructuras de control y ciclos, declaraciones y asignaciones anidadas.

DIAGRAMA DE CLASES:

El diagrama de clases debe manejar **solamente las siguientes relaciones**, ya que el programa de OLCEE es una versión beta que generara diagramas de clases básicos y no tan complejos:

Herencia o generalización:

Agregación:

Para generar el diagrama de clases se usara la herramienta **graphviz**, y el diagrama generado debe ser mostrado en la aplicación.

Para generar el diagrama se tomara en cuenta el paradigma de polimorfismo que vera representado en la sobrecarga de métodos.

MANEJO Y RECUPERACION DE ERRORES:

Para generar el diagrama de clases así como la traducción de código, la entrada debe estar libre de errores léxicos y sintácticos. **La aplicación debe estar en la capacidad de poder recuperarse de errores léxicos y sintácticos**, para corregir de forma eficiente el archivo y generar el diagrama de clases del proyecto analizado.

Reporte de errores léxicos y sintácticos:

No.	Descripción	Tipo de error	Línea	Columna
1	^ no pertenece la lenguaje	Léxico	12	45
2	Se esperaba … (se encontró id …)	Sintáctico	45	12
3	…	…	…	…

El reporte debe ser en formato HTML y poder ser visualizado al momento de terminar el análisis. Como parte adicional se manejara **excepciones de control en la aplicación**, es decir, **debe verificar que si una clase hace referencia a otra clase heredando o instanciándola en sus atributos, dicha clase debe existir dentro del proyecto igual que los procedimientos reescritos en sus clases hijas,** de lo contrario mostrara un error al usuario indicando el motivo del error, para esto se puede mostrar un reporte debe ser en formato HTML y poder ser visualizado al momento de terminar el análisis.

NOTAS IMPORTANTES:

- **Se debe realizar de manera individual.**
- Las herramientas utilizadas para realizar análisis léxico y sintáctico serán Flex y Bison.
- La aplicación se desarrollara en el lenguaje de programación C++ utilizando el IDE QT.
- El lenguaje es case sensitive, por lo cual distingue entre minúsculas y mayúsculas (el lenguaje base es la sintaxis de Java).
- Como se mencionó antes, la herramienta para generar el diagrama de clase será únicamente Graphviz.
- **Copias parciales o totales tendrán una nota de cero (0) puntos y se notificará a la escuela para que se apliquen las sanciones correspondientes.**
- **En el caso de no cumplir con alguna de las indicaciones antes mencionadas o con los entregables descritos a continuación NO se calificará; por lo cual, se tendrá una nota de cero puntos.**

ENTREGABLES:

Para el tener derecho se contara con los siguientes entregables para su calificación:

- Archivos Flex y Bison.
- Código fuente.
- Ejecutable.

PRACTICA 1:

La Practica 1 consistirá en realizar las funcionalidades de la interfaz gráfica (desplegar directorio de archivos “treeview”, crear, modificar, guardar, eliminar y analizar archivos) así como también poder realizar el análisis léxico y sintáctico de los archivos de entrada de la calificación y en el caso de que existan errores léxicos y sintácticos desplegarlos automáticamente.

FECHA LÍMITE DE ENTREGA PRACTICA 1 DOMINGO 10 DE DICIEMBRE DE 2017 ANTES DE LAS 11:59 PM

PROYECTO 1:

El Proyecto 1 consistirá poder desplegar el diagrama de clases de los respectivos archivos de entrada utilizados en la calificación. Así como la recuperación de errores léxicos y sintácticos y visualizar los errores de control en el caso de existieran errores de este tipo.

FECHA LÍMITE DE ENTREGA PROYECTO 1 DOMINGO 17 DE DICIEMBRE DE 2017 ANTES DE LAS 11:59 PM

Ejemplos de archivos de entrada:

EJEMPLO 1:

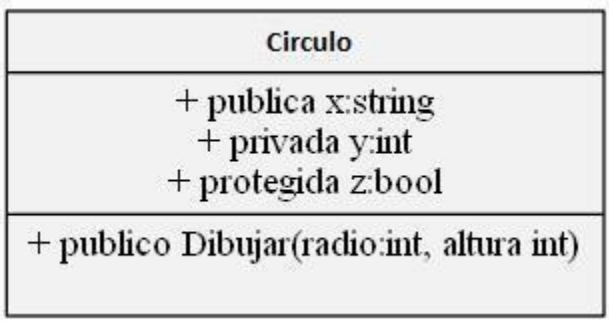
Entrada1.java

```
class Circulo {

    public String  x;
    private int y;
    protected boolean z;

    public void Dibujar(int radio, int altura){
            if (x > radio && this.y < altura) {
                    this.x = "Se va a dibujar un circulo"
            }
    }
}
```

Diagrama de clases:



EJEMPLO 2:

Entrada2.java

```
public class persona {
        public String  nombre;
        public int edad;
        public int correo;
        private int nit;
        private int dpi;
```

```
        public void saltar(int numero){
                if (numero>10){
                        int salto = numero;
                }else{
                        int salto = numero;
                }

        }

        public int Edad(){
                this.edad = 18;
        }

        private boolean Casado(boolean bandera){
                bandera = true;
        }
}

protected class hombre extends persona{
        private int NoSeguro;

        private void cantar(){
                String  valor;
                valor = "El hombre canta";
        }
}

private class mujer extends persona{
        int NoMartenidad;

        String  Colocar_Nombre(String  nombre_1, String
                nombre_2){ String  nombre = nombre_1 + nombre_2;
        }
}

public class carro{
        String  color;
        String  modelo;
        int anio;
        persona Duenio;
}
```

Diagrama de Clases: