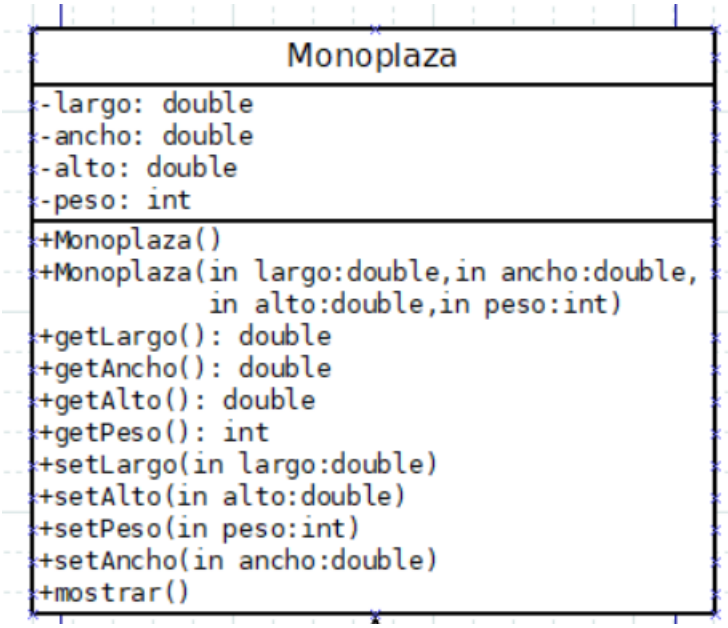
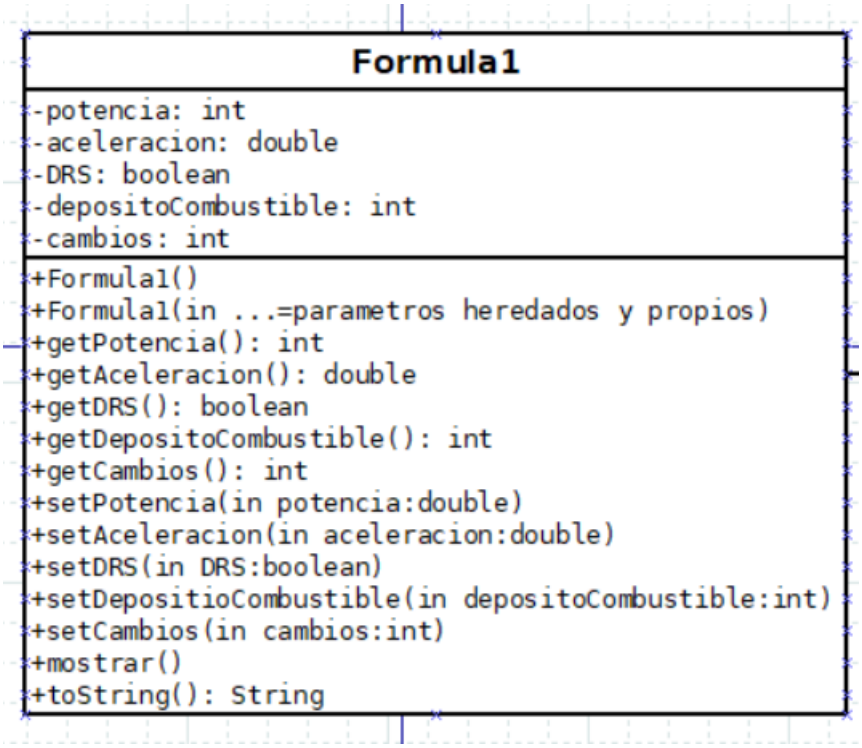
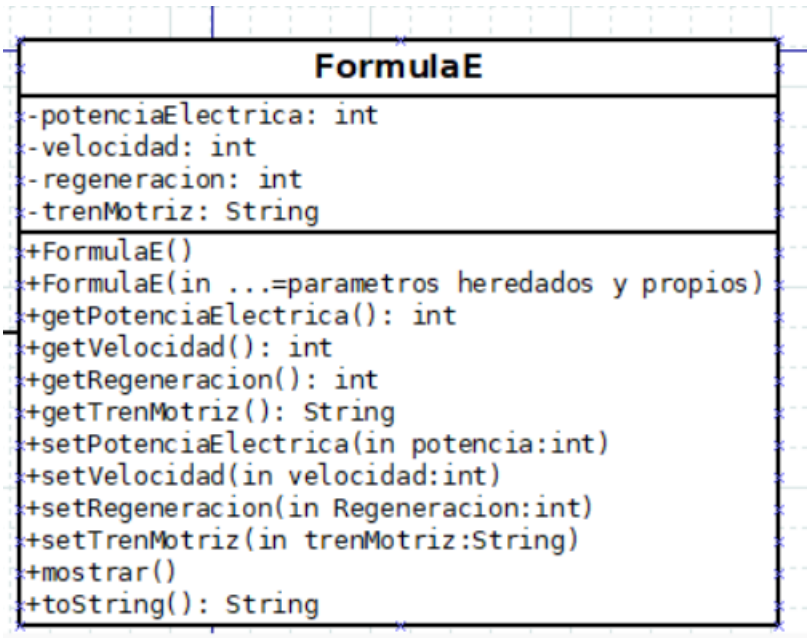
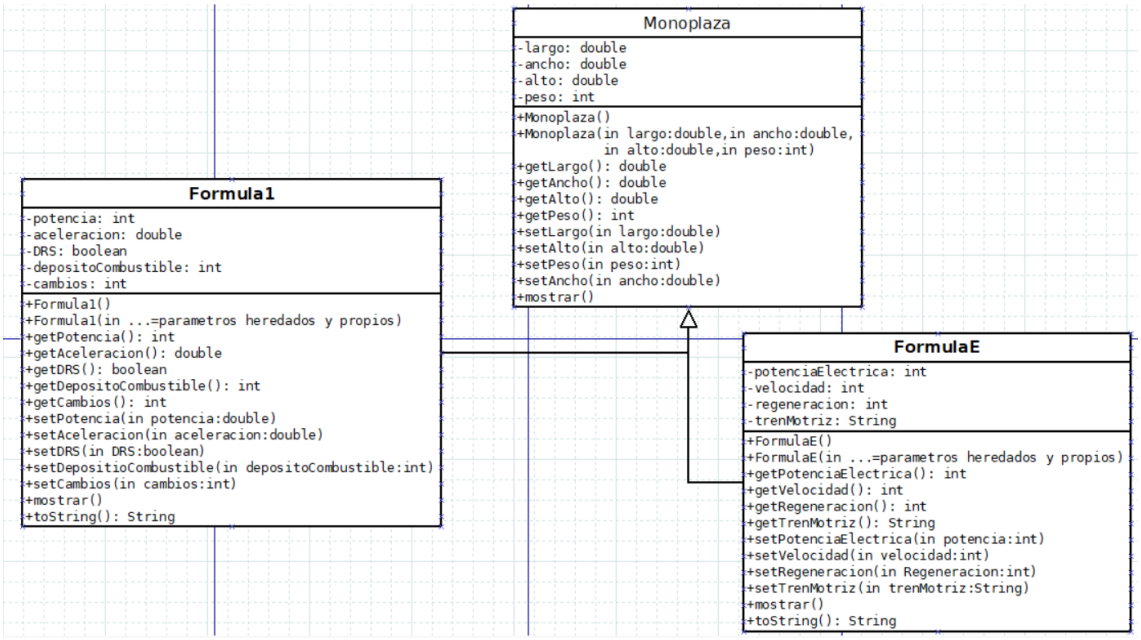
Ejercicio. Jerarquía de Clases, herencia y polimorfismo

Tenemos las siguientes clases y su relación de herencia. Se representan las clases y la relación de herencia mediante diagramas UML:




Formula1
-potencia: int
-aceleracion: double
-DRS: boolean
-depositoCombustible: int
-cambios: int
+Formula1()
+Formula1(in ...=parametros heredados y propios)
+getPotencia(): int
+getAceleracion(): double
+getDRS(): boolean
+getDepositoCombustible(): int
+getCambios(): int
+setPotencia(in potencia:double)
+setAceleracion(in aceleracion:double)
+setDRS(in DRS:boolean)
+setDepositioCombustible(in depositoCombustible:int)
+setCambios(in cambios:int)
+mostrar()
+toString(): String


FormulaE
-potenciaElectrica: int
-velocidad: int
-regeneracion: int
-trenMotriz: String
+FormulaE()
+FormulaE(in ...=parametros heredados y propios)
+getPotenciaElectrica(): int
+getVelocidad(): int
+getRegeneracion(): int
+getTrenMotriz(): String
+setPotenciaElectrica(in potencia:int)
+setVelocidad(in velocidad:int)
+setRegeneracion(in Regeneracion:int)
+setTrenMotriz(in trenMotriz:String)
+mostrar()
+toString(): String



Hay que realizar las siguientes tareas:

- Implementar la jerarquía de clases. ¿Hay alguna clase abstracta ? Argumenta la respuesta de tu implementación.
- Constructores:
  - Vació.
  - Parametrizado. Los constructores parametrizados de las clases derivadas reciben todos los parámetros.
- Getters/setters: los indicados en el diagrama UML.
- Método mostrar: muestra el contenido de la instancia del objeto. Se indica el nombre del atributo y el valor. Hay que imprimir en pantalla: una cabecera (Formula 1 o Formula E) y un cuerpo de datos (un atributo por línea).
- toString: redefinición del método heredado de la clase Object, para devolver un string con el contenido del objeto concreto. Este método está incluido en ambas clases

derivadas y debe ser implementado en cada clase. una cabecera (Formula 1 o Formula E) y un cuerpo de datos (un atributo por línea).

- Clase “Flow” que contenga el método “main”. Se realiza lo siguiente:
    - Declarar un objeto de la clase Monoplaza. Crear un objeto de la clase Formula1 y otro objeto de la clase FormulaE. Los datos a contener en cada objeto se muestran en las imágenes que aparecen más abajo.
    - Mostrar el contenido de los objetos derivados.
    - Asignar el objeto creado de Formula1 al objeto declarado de la clase base y llamar al método mostrar ¿ Qué ocurre? ¿ Que código se ejecuta ? ¿ Que explicación encuentras ?
- Hacer lo mismo con el otro objeto. ¿ Qué sucede ? ¿ Que explicación tiene ?

Datos de las instancias creadas:

Características	Fórmula 1
Dimensiones: largo	5,6-5,7 metros
Dimensiones: ancho	+/- 2 metros
Altura	0,95 metros
Peso	746 kg con piloto
Motor	V6 turbo 1,6 litros – 880-1000 CV a 15.000 rpm + ERS (163 CV durante 33,3 segundos por vuelta)
Aceleración (0 a 100 km/h)	2,4 segundos
DRS	Sí
Depósito de combustible	110 litros
Caja de cambios	Secuencial de 8 velocidades
Ruedas	13 pulgadas
Tiempo de pole en Montmeló en 2020	1:15.584

Formula E

Longitud	5,320 metros
Altura	1,050mm
Anchura	1,780mm
Distancia entre ejes	3,100 metros
Peso (incluído el piloto)	920kg (batería 450kg)
Potencia máxima	200kW
Regeneración máxima	100kW
Velocidad máxima	140mph
Tren motriz	Trasero
Neumáticos	Michelin

¿ Qué dificultades aparecen al implementar las clases ? y ¿ la jerarquía de clases ? ¿ Cuantas consultas has realizado por tu cuenta para comprender esas dificultades ? ¿ Cuál es tu solución ?

La jerarquía de clases permite reutilizar código. Código que es común en los descendientes se incluye en el padre y posteriormente los hijos lo heredan y no hay que volver a escribir el mismo código (parámetros y métodos) porque lo reciben por herencia.

Los métodos que se usan de forma diferente en cada descendiente son abstractos en el padre. De esta forma cada hijo se ve obligado a reescribir el método(s), pues el compilador avisa de la obligatoriedad de redefinición de código.

El polimorfismo se hace a través del padre y el operador de asignación (=). Cada descendiente creado con el operador new es una variable, llamada instancia de clase u objeto, con una dirección

de memoria propia. La dirección de memoria se denomina referencia de memoria o simplemente referencia. Igualando un objeto descendiente, creado, a un padre, declarado, hacemos que el padre contenga al hijo y se ejecuta la versión del método del hijo. Asignando otro descendiente diferente al padre conseguimos que se ejecute el método del nuevo descendiente. Si el método a ejecutar es sólo de un hijo, lógicamente se ejecuta ese método, pero si es un método común y abstracto en el padre se ejecuta el método del descendiente asignado al padre.

Código

```
public abstract class  Monoplaza {
    private double largo;
    private double ancho;
    private double alto;
    private int peso;

    public Monoplaza() {
        largo = 0;
        ancho = 0;
        alto = 0;
        peso = 0;
    }

    public Monoplaza(double largo, double ancho, double alto, int peso) {
        super();
        this.largo = largo;
        this.ancho = ancho;
        this.alto = alto;
        this.peso = peso;
    }

    public double getLargo() {
        return largo;
    }

    public double getAncho() {
        return ancho;
    }
```

```
public void setAncho(double ancho) {
    this.ancho = ancho;
}

public void setAlto(double alto) {
    this.alto = alto;
}

public void setPeso(int peso) {
    this.peso = peso;
}

public abstract void mostrar();

@Override
public String toString() {
    return "Monoplaza\n Largo=" + largo + "\nAncho=" + ancho + "\nAlto="
+ alto + "\nPeso=" + peso + "\n";
}
```

```
public class Formula1 extends Monoplaza {
    private int potencia;
    private double aceleracion;
    private boolean DRS;
    private int depositoCombustible;
    private int cambios;

    public Formula1() {
        super();
        potencia = 0;
        aceleracion = 0;
        DRS = false;
        depositoCombustible = 0;
        cambios = 0;
    }

    public Formula1(double largo, double ancho, double alto, int peso, int potencia,
            double aceleracion, boolean dRS, int depostitoCombustible, int cambios) {
        super(largo, ancho, alto, peso);
        this.potencia = potencia;
        this.aceleracion = aceleracion;
        DRS = dRS;
        this.depositoCombustible = depositoCombustible;
        this.cambios = cambios;
    }

    public int getPotencia() {
        return potencia;
    }
```

```
public double getAceleracion() {
    return aceleracion;
}

public boolean isDRS() {
    return DRS;
}

public int getDepositoCombustible() {
    return depositoCombustible;
}

public int getCambios() {
    return cambios;
}

public void setPotencia(int potencia) {
    this.potencia = potencia;
}

public void setAceleracion(double aceleracion) {
    this.aceleracion = aceleracion;
}

public void setDRS(boolean dRS) {
    DRS = dRS;
}

public void setDepostitoCombustible(int depositoCombustible) {
    this.depositoCombustible = depositoCombustible;
}


  public void setCambios(int cambios) {
      this.cambios = cambios;
  }

  public void mostrar() {
      System.out.println(this.toString());
  }

  @Override
  public String toString() {
      return super.toString() + "\nFormula1 \nPotencia=" + potencia + "\nAceleracion="
              + aceleracion + "\nDRS=" + DRS
              + "\nDeposito Combustible=" + depositoCombustible + "\nCambios=" + cambios;
  }
```

```
public class FormulaE extends Monoplaza{
    private int potenciaElectrica;
    private int velocidad;
    private int regeneracion;
    private String trenMotriz;

    public FormulaE() {
        super();
        potenciaElectrica = 0;
        velocidad = 0;
        regeneracion = 0;
        trenMotriz = "";
    }

    public FormulaE(double largo, double ancho, double alto, int peso, int potenciaElectrica,
            int velocidad, int regeneracion, String trenMotriz) {
        super(largo, ancho, alto, peso);
        this.potenciaElectrica = potenciaElectrica;
        this.velocidad = velocidad;
        this.regeneracion = regeneracion;
        this.trenMotriz = trenMotriz;
    }

    public int getPotenciaElectrica() {
        return potenciaElectrica;
    }

    public int getVelocidad() {
        return velocidad;
    }
```

```
public int getRegeneracion() {
    return regeneracion;
}

public String getTrenMotriz() {
    return trenMotriz;
}

public void setPotenciaElectrica(int potenciaElectrica) {
    this.potenciaElectrica = potenciaElectrica;
}

public void setVelocidad(int velocidad) {
    this.velocidad = velocidad;
}

public void setRegeneracion(int regeneracion) {
    this.regeneracion = regeneracion;
}

public void setTrenMotriz(String trenMotriz) {
    this.trenMotriz = trenMotriz;
}

public void mostrar() {
    System.out.println(this.toString());
}
```

```java
@Override
public String toString() {
    return super.toString() + "\nFormulaE \n Potencia Electrica=" + potenciaElectrica
            + "\nVelocidad=" + velocidad + "\nRegeneracion="
            + regeneracion + "\ntrenMotriz=" + trenMotriz;
}
```

```java
public class Flow {

    public static void main(String[] args) {
        //declarando objetos
        Monoplaza M1;
        Formula1 F1;
        // creando objetos
        F1 = new Formula1(5.6,2,0.95,746,1000,2.4,true,110,8);
        FormulaE FE = new FormulaE(5.3,1.78,1.05,920,200,260,100,"Trasero");

        // ejecución habitual. Cada objeto ejecuta su métodos
        F1.mostrar();
        FE.mostrar();

        // polimorfismo. Asignado el hijo al padre, el padre ejecuta el método del hijo asignado
        M1 = F1;
        M1.mostrar();
        M1 = FE;
        M1.mostrar();

        //M1.setDRS(false); esta instrucción da error, pues el padre no tiene ese método, ni siquiera
        // como abstracto.
        // Aquí esta la forma de conseguir acceder al método propio del descendiente que no esta
        // registrado en el padre.

        if (M1 instanceof Formula1) {
            ((Formula1) M1).setDRS(false);
        }
    }
}
```