

Java Inicial

(20 horas)





Temario

1. Programación Orientada a Objetos
2. Introducción y Sintaxis Java
3. Sentencias Control Flujo
4. POO en Java
5. Relaciones entre Objetos
6. **Polimorfismo, abstracción e interfaces**
7. Excepciones
8. Conceptos avanzados



Tema 6

Polimorfismo, Abstracción e
Interfaces

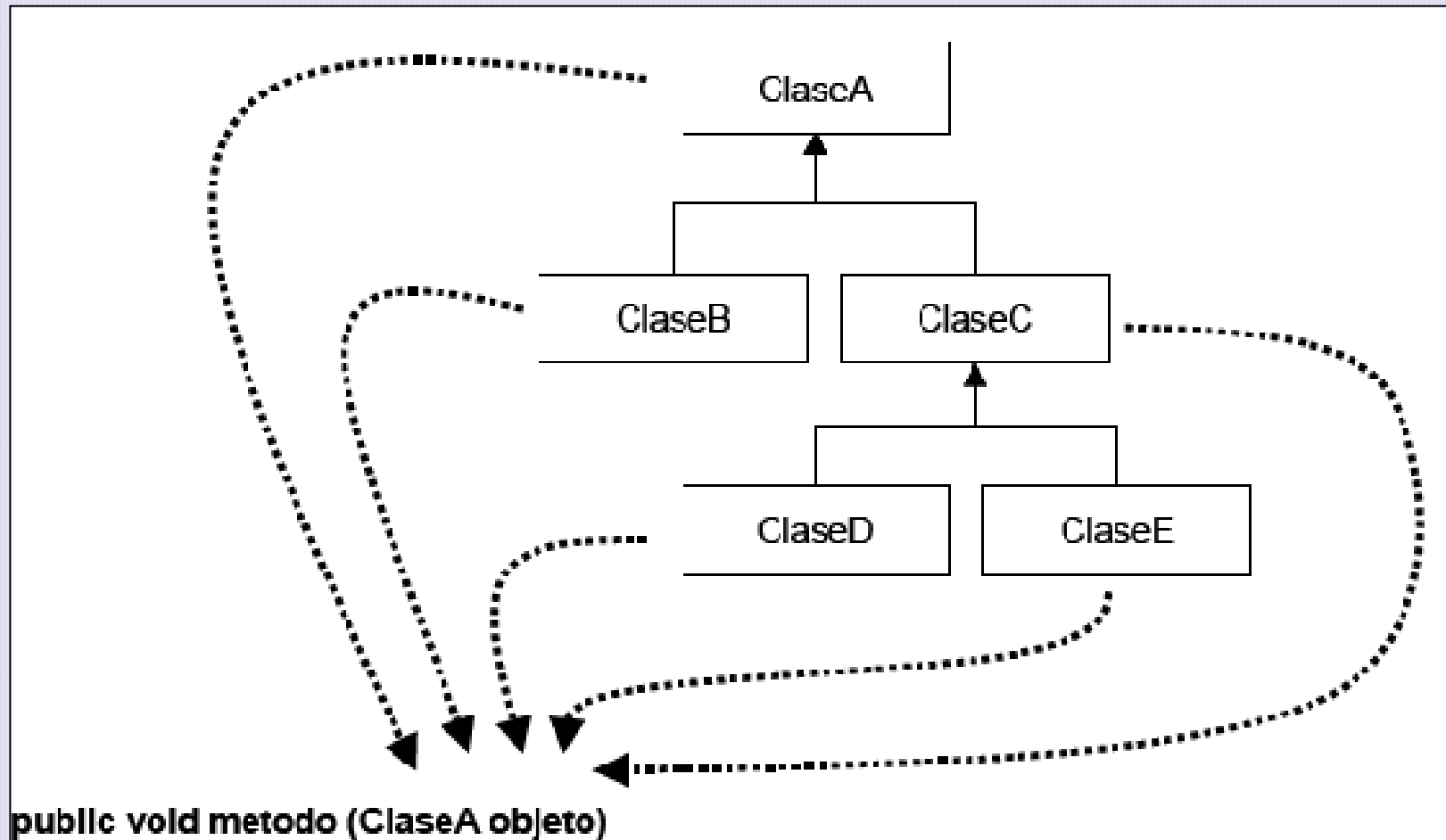


Objetivos

1. Programación Orientada a Objetos
2. Introducción y Sintaxis Java
3. Sentencias Control Flujo
4. POO en Java
5. Relaciones entre Objetos
6. **Polimorfismo, abstracción e interfaces**
7. Excepciones
8. Conceptos avanzados

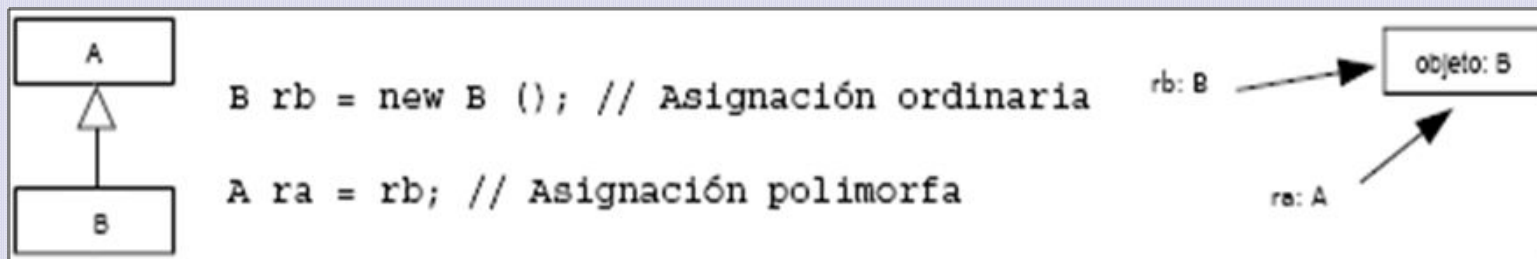
- Polimorfismo
 - Variables polimórficas
- Clases Abstractas
- Interfaces
 - Herencia múltiple

- Polimorfismo: es la capacidad que tiene los objetos de comportarse de *múltiples formas*.



■ Polimorfismo: *Upcasting*

- Únicamente tienen sentido por la existencia de la herencia.
- El polimorfismo (o upcasting) consiste en la posibilidad de que una referencia a objetos de una clase pueda conectarse también con objetos de descendientes de ésta



■ Polimorfismo: *Upcasting* (II)

- El sentido del polimorfismo es realizar una **generalización**, olvidar los detalles concretos de uno o varios objetos de distintas clases y buscar un punto común a todos ellos en un ancestro
- Se trata de algo que realiza comúnmente la mente humana durante el proceso de razonamiento

■ Polimorfismo: *Paso argumentos*

- Las conexiones polimorfas se realizan a veces de manera implícita en el **paso de argumentos** a una operación.

```
public class AppGestorTareas {
    private static void esperarEjecutar (TareaPeriodica tp)
    {
        while (!tp.necesitaEjecucion ());
        tp.ejecutarTarea ();
    }

    public static void main (String[] args) {
        TPREloj tpr = new TPREloj ();
        TPAviso tpa = new TPAviso ("Ha pasado un minuto", 60);
        TPEjecucion tpe = new TPEjecucion ("/bin/sync", 120);

        esperarEjecutar (tpr);
        esperarEjecutar (tpa);
        esperarEjecutar (tpe);
    }
}
```

- Polimorfismo: Variables polimórficas:
 - Puede contener referencias a objetos de diferentes clases
 - Ejemplo:
 - Necesitamos implementarnos una agenda para almacenar 500 datos ... de Personas o Empleados.
 - ¿Cómo definiremos la clase Agenda?
 - ¿Cómo definiremos el array que lo almacenara?
 - Tipo Persona?
 - Tipo Empleado?
 - Dos arrays independientes → Mal uso memoria!

- Polimorfismo: Variables polimórficas (II)
 - Si tomamos la decisión en base al tamaño:
 - Clase Persona > Clase Empleado

```
class TestPolimorfismo
{
    public static void main(String args[])
    {
        Empleado objetos[] = new Empleado[500];
        objetos[0] = new Empleado();
        objetos[1] = new Persona();
    }
}
```

- Lo anterior da error de compilación

```
Empleado objetos[ ] = new Empleado[500];
```

```
objetos[0] = new Empleado();
```

Si es correcto porque todo Empleado es un Empleado (obviamente)

```
objetos[1] = new Persona();
```

No es correcto ya que NO TODA Persona es un Empleado

Por tanto esta solución NO ES válida.

- Polimorfismo: Variables polimórficas (III)
 - Si tomamos la decisión de la clase más genérica:

```
class TestPolimorfismo
{
    public static void main(String args[])
    {
        Persona objetos[] = new Persona[500];
        objetos[0] = new Empleado();
        objetos[1] = new Persona();
    }
}
```

- Ahora no da Error ¿por qué?

Pero ¿Y el tamaño de los objetos?

Es en el momento de la instanciación cuando se reserva espacio en memoria para un objeto de la clase Persona o un objeto de la clase Empleado, lo que se almacenará en el array será la posición que ocupará dicho objeto en memoria.

■ Polimorfismo: Variables polimórficas (IV)

• Pero siempre debe quedar claro que tras la conexión polimorfa únicamente podemos acceder a las operaciones pertenecientes a la clase asociada a la referencia. El resto de operaciones del objeto no son accesibles a través de esta referencia

```
TestPolimorfismo.java:8: cannot resolve symbol
symbol   : method setDepartamento (java.lang.String)
location: class Persona
    objetos[0].setDepartamento("VENTAS");
                    ^
```

SOLAMENTE PODREMOS INVOCAR MÉTODOS DE LA CLASE DE DECLARACIÓN.

- Polimorfismo: Variables polimórficas (V)
 - Problemas de las variables polimórficas:
 - Solución 1: Hacemos un Cast

```
public static void main(String args[])
{
    Persona objetos[] = new Persona[500];
    objetos[0] = new Empleado();
    objetos[1] = new Persona();
    ((Empleado) objetos[0]).setDepartamento("VENTAS");
}
```

```
public static void main(String args[])
{
    Persona objetos[] = new Persona[500];
    objetos[0] = new Empleado();
    objetos[1] = new Persona();
    int i;
    for(i=0; i<objetos.length; i++)
    {
        if (objetos[i] instanceof Empleado)
            ((Empleado) objetos[i]).setDepartamento("VENTAS");
    }
}
```

■ Clases Abstractas

- Existen clases que representan conceptos genéricos y es ilógico instanciar objetos.
- Puede ser imposible o inútil la implementación de ciertas operaciones.
- La utilidad de este tipo de clases está en la aplicación de herencia para obtener clases que representan conceptos concretos
 - La clase *TareaPeriodica* es un claro ejemplo: por sí sola no tiene utilidad, pero simplifica mucho la construcción de las otras tres clases. De hecho, la operación *ejecutarTarea* en *TareaPeriodica* no tiene una implementación útil

■ Clases Abstractas (II)

- Una clase ***abstracta*** es aquella que tiene una funcionalidad definida
 - Pero que no se puede implementar dicha funcionalidad.
 - Motivo: Se trate de una clase demasiado genérica.
- A la funcionalidad no implementada se le llama **método abstracto**
- Sintaxis

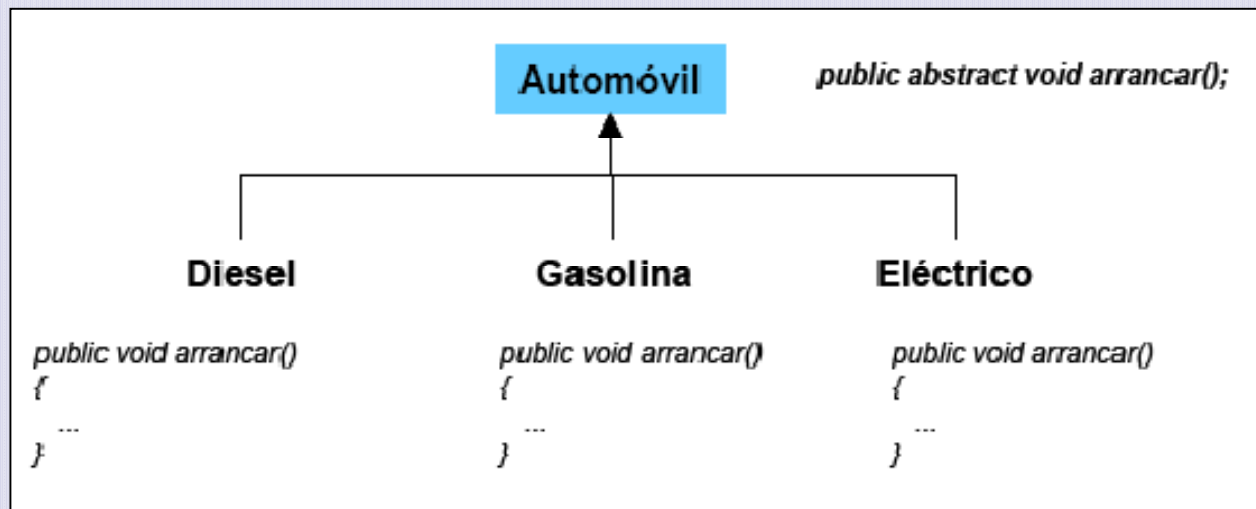
```
public abstract class ClaseAbstracta
{
    ...
    public abstract tipo metodoAbstracto();
    ...
}
```

■ Clases Abstractas (III)

- Las operaciones abstractas deben ser implementadas obligatoriamente en alguna de las subclases para que la clase correspondiente sea instanciable.
- Una clase abstracta puede no tener ninguna operación abstracta, pero una clase que contenga al menos una operación abstracta debe ser declarada como abstracta

■ Clases Abstractas (II):

- No se puede crear objetos (No hay implementación)
- Los métodos abstractos también se heredan.



- La clase automóvil define el método `arrancar()` por que todos los coches tienen dicha funcionalidad.
 - Al no conocer el motor → no puede implementar el método

■ Clases Abstractas: Ejemplo

```
public abstract class Figura
{
    //NO TIENE ATRIBUTOS
    //NO TIENE CONSTRUCTORES
    //NO TIENE métodos de ACCESO
    //Declaración e Implementación de métodos de COMPORTAMIENTO
    public abstract void escalar(int e);
    public abstract void moverAbs(int x, int y);
    public abstract void moverRel(int dx, int dy);
    public abstract double area();
}
```

■ Clases Abstractas: Ejemplo

```
public class Rectangulo
{
    //Declaracion de ATRIBUTOS
    int base, altura;
    Punto2D origen;
    //Declaración e Implementación de CONSTRUCTORES
    public Rectangulo(){...}
    public Rectangulo(int x, int y, int b, int h){...}
    public Rectangulo(Punto2D p, int b, int h){...}
    public Rectangulo(Rectangulo r){...}
    //Declaración e Implementación de métodos de ACCESO
    public void setOrigen(Punto2D p){...}
    public void setBase(int b) {...}
    public void setAltura(int h) {...}
    public Punto2D getOrigen(){...}
    public int getBase(){...}
    public int getAltura(){...}
    //Declaración e Implementación de métodos de COMPORTAMIENTO
    public String toString(){...}
    public void moverAbs(int x, int y) {...}
    public void moverRel(int dx, int dy) {...}
    public double area(){...}
    public void escalar(int e) {...}
}
```

Genéricos

■ Clases Abstractas: Ejemplo

```
public class Circulo
{
    //Declaracion de ATRIBUTOS
    private Punto2D origen;
    private int radio;
    //Declaración e Implementación de CONSTRUCTORES
    public Circulo(){...}
    public Circulo(Punto2D p, int r) {...}
    public Circulo(Circulo c) {...}
    //Declaración e Implementación de métodos de ACCESO
    public void setOrigen(Punto2D p) {...}
    public void setRadio(int r) {...}
    public Punto2D getOrigen(){...}
    public int getRadio(){...}
    //Declaración e Implementación de métodos de COMPORTAMIENTO
    public String toString(){...}
    public void moverAbs(int x, int y) {...}
    public void moverRel(int dx, int dy) {...}
    public void escalar(int e) {...}
    public double area(){...}
}
```

Genéricos

- ¿Qué ocurriría si el Círculo y el Rectángulo heredasen de otra clase anteriormente?
 - No podríamos tener la clase figura ya que Java no permite **herencia múltiple**
- Para esto tenemos los Interfaces
 - Clase que declara solo métodos sin implementarlos

```
public interface NombreInterface
{
    public tipo metodoUno( Lista de argumentos );
    public tipo metodoDos( Lista de argumentos );
    ...
}
```

- Permite Herencia múltiple

■ Ejercicio

- Implementar en una clase concreta (SopaLetrasImplementacion) la clase SopaLetras que representa una sopa de letras de 7x7.
- `Public abstract void crearSopa(char[] c)`
- `Public abstract void pintarSopaLetras()`
- `Public abstract boolean existeLetra?(char c)`
- `Public abstract boolean contienePalabra?(char[] c)`

- Pistas que pueden ayudar:
 - Se pueden pasar de `char[]` a `String` con `String s=new String(char[]);`
 - Se puede buscar un substring en un `String s` con el metodo `s.contains(String)`

■ Interfaces

- La idea de clase abstracta, llevada al extremo, nos lleva en Java a las interfaces. Una interfaz es similar a una clase totalmente abstracta
- Sirven para especificar las operaciones que obligatoriamente deben implementar una serie de clases

■ Interfaces

- Sintaxis para indicar que una clase implementa un interfaz.

```
class MiClase implements NombreInterface
{
    ...
}
```

- Podemos implementar varios interfaces

```
class MiClase implements NombreInterfacel, NombreInterface2
{
    ...
}
```

- Puede ser utilizado para definir una variable
 - Puede incluir un objeto que implemente dicho interfaz

```
NombreInterface variable;
```

■ Interfaces

□ Permite herencia múltiple

```
public interface Comportamiento1
{
    ...
}
public interface Comportamiento2
{
    ...
}
public Comportamiento1Y2 extends Comportamiento1, Comportamiento2
{
    ...
}
```

□ No es necesario poner el *public abstract* a los métodos → redundante

```
public interface MiInterface
{
public abstract void escalar(int e);
public abstract void moverAbs(int x, int y);
public abstract void moverRel(int dx, int dy);
public abstract double area();
}
```

■ Interfaces: Ejemplo Rectángulo / Círculo

□ Interface

```
public interface MiInterface
{
    public abstract void escalar(int e);
    public abstract void moverAbs(int x, int y);
    public abstract void moverRel(int dx, int dy);
    public abstract double area();
}
```

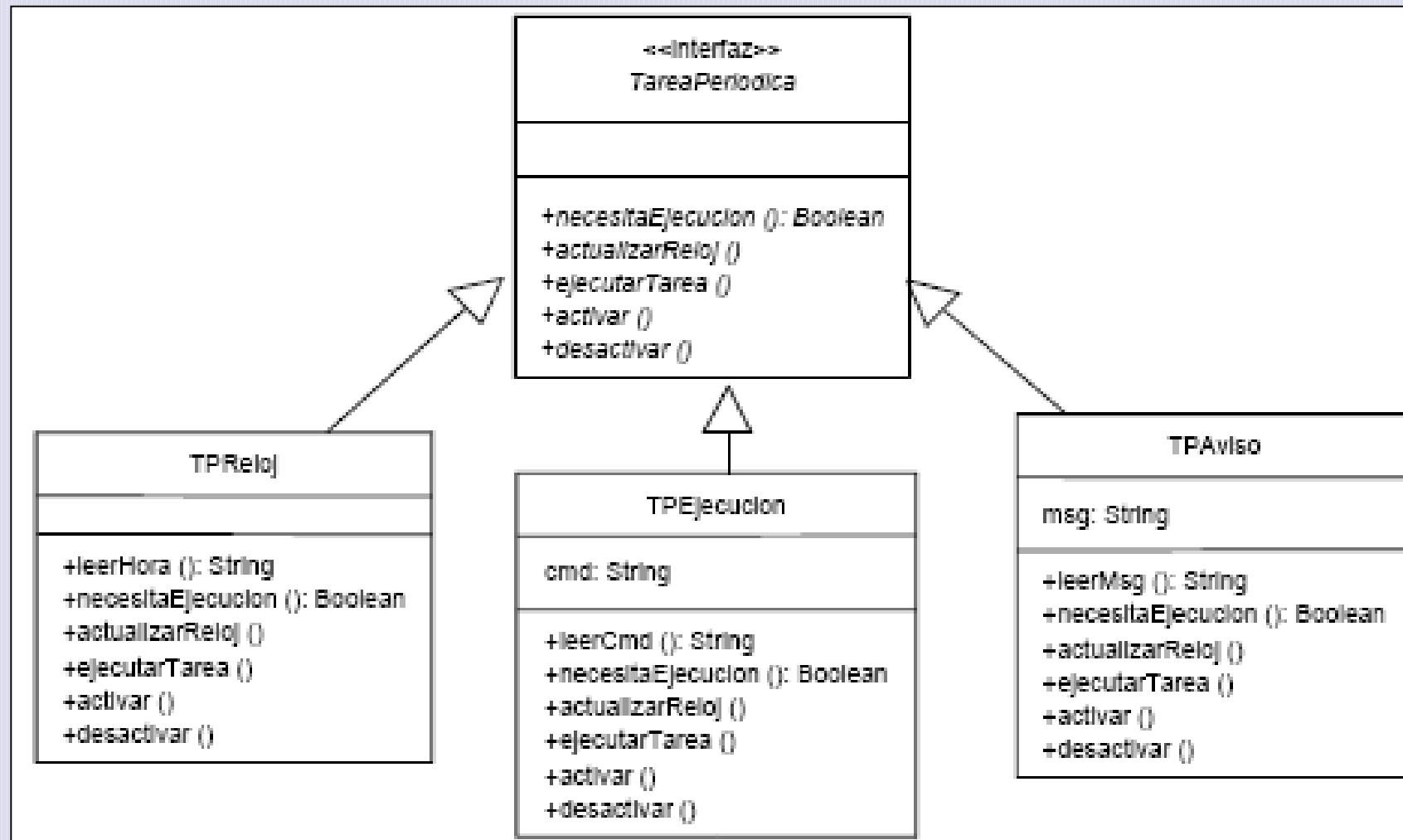
□ Rectángulo

```
class Rectángulo implements MiInterface
{
    ...
}
```

□ Círculo

```
class Circulo implements MiInterface
{
    ...
}
```

■ Interfaces: Ejemplo Tarea periódica



■ Ejemplo del workspace

- Existen una serie de clases y interfaces (Animal.java, AnimalInterface.java...) que generalizan el concepto de animal, con una serie de métodos y atributos.
- Por otro lado, las clases Perro muestran como utilizar la clase Animal con distintas relaciones (herencia, interfaces...)



Conclusiones

1. Programación Orientada a Objetos
2. **Introducción y Sintaxis Java**
3. Sentencias Control Flujo
4. POO en Java
5. Relaciones entre Objetos
6. Polimorfismo, abstracción e interfaces
7. Excepciones
8. Conceptos avanzados

- Polimorfismo
 - Variables polimórficas
- Clases Abstractas
- Interfaces
 - Herencia múltiple