

Resumen Final de Taller de Programación

Pascal:

Definición de Merge:

La operación de Merge consiste en generar una nueva estructura de datos (arreglos, listas) ordenada a partir de la mezcla de **dos o más estructuras** de datos previamente ordenadas.

Las estructuras que se combinan guardan el mismo orden lógico interno (por ejemplo datos ordenados alfabéticamente).

Otro tipo de Merge es el Acumulador, que la única diferencia que tiene con un Merge normal es que **primero acumula cantidad de un mismo elemento** antes de guardarlo en la nueva estructura.

Recursión:

Definición y características:

Es una metodología para resolver problemas. Permite resolver un problema P por resolución de instancias más pequeñas P1, P2, ... Pn del mismo problema.

Es una metodología para resolver problemas. El problema Pi es de la misma naturaleza que el problema original, pero en algún sentido es más simple.

Una **solución recursiva** resuelve un problema por resolución de instancias más pequeñas del mismo problema.

Un algoritmo recursivo involucra:

- Alguna condición de terminación (implícita/explicita)
- Una auto invocación (llamada recursiva). Se debe garantizar que en un nro. finito de autoinvocaciones se alcanza la condición de terminación.

Búsqueda dicotómica (Pseudocódigo) (Caso Base):

Buscar (vector, **datoABuscar**)

 Si el vector “no tiene elementos” **entonces**

 No lo encontré y termino la búsqueda

Sino

 Determinar el punto medio del vector

 Comparar **datoABuscar** con el contenido del punto medio

 Si coincide **entonces** “Lo encontré”

Sino

 Si **datoABuscar** < contenido del punto medio **entonces**

Buscar (1era mitad del vector, **datoABuscar**)

Sino

Buscar (2da mitad del vector, **datoABuscar**)

Observaciones:

- 1) El módulo realiza invocaciones a sí mismo.
- 2) En cada llamada, el tamaño del vector se reduce a la mitad.
- 3) Existen 2 casos distintos que se resuelven de manera particular y directa (casos base):

a) Cuando el vector "no contiene elementos"

b) Cuando encuentro el datoABuscar

Ordenación de vectores:

El proceso por el cual, un grupo de elementos puede ser ordenado se conoce como algoritmo de ordenación.

Pseudocódigo:

- Ordenación de menor a mayor.

Ordenar (v: tipo_vector; n: valor_de_dimL);

Repetir desde i: =2 hasta n

 guardar dato (a ordenar)

 j: = i-1

 Mientras (j > 0) y (v[j] > dato a ordenar)

 v[j+1]: = v[j]

 j: = j - 1

 guardar dato en v[j+1]

Análisis Teórico:

Supongamos que C: Nro comparaciones y M: Nro de intercambios:

$$n-1 \leq C \leq n(n-1) / 2$$

$$2(n-1) \leq M \leq 2(n-1) + n(n-1) / 2$$

Árboles:

Un árbol es una estructura de datos que satisface tres propiedades:

- Cada elemento del árbol se relaciona con cero o más elementos (hijos).
- Si el árbol no está vacío, hay un único elemento (raíz) y que no tiene padre (predecesor).
- Todo otro elemento del árbol posee un único padre y es un descendiente de la raíz.

Características:

- **Homogénea:** Todos los elementos son del mismo tipo.
- **Dinámica:** Puede aumentar o disminuir su tamaño durante la ejecución del programa.
- **No lineal:** Cada elemento puede tener 0, 1 o más sucesores.
- **Acceso Secuencial.**

¿Cómo se relacionan los nodos de un árbol binario?

Type

```
elemento = tipoElemento;  
arbol = ^nodo;  
nodo = record  
elem: elemento;  
hijoIzq: arbol;  
hijoDer: arbol;
```

end;

Insertar un dato en el arbol:

```
insertar (arbol, dato)  
  si arbol es nil  
    creo nodo_nuevo y pongo el dato y los hijos en nil  
    arbol: = nodo_nuevo  
  sino  
    si el dato en árbol es > dato  
      insertar (hijo_izquierdo_del_arbol, dato);  
    sino  
      insertar (hijo_derecho_del_arbol, dato);
```

Insertar un dato en el arbol sin repeticiones:

```
insertarSinRepetidos (arbol, dato)  
  si arbol es nil  
    creo nodo_nuevo y pongo el dato y los hijos en nil  
    arbol: = nodo_nuevo  
  sino  
    si el dato en árbol es > dato  
      insertarSinRepetidos (hijo_izquierdo_del_arbol, dato);  
    sino  
      si el dato en el árbol < dato  
        insertarSinRepetidos (hijo_derecho_del_arbol, dato);
```

Recorrido de un árbol binario:

Los distintos recorridos permiten desplazarse a través de todos los nodos del árbol de tal forma que cada nodo sea visitado una y solo una vez.

Existen varios métodos que se diferencian en el orden que se visitan:

- Recorrido En – Orden
- Recorrido Pre – Orden
- Recorrido Post – Orden

Recorrido acotado de un árbol binario:

Hemos analizado algunas situaciones que obligan a recorrer todos los nodos del árbol, por ejemplo, imprimir los nodos del árbol. Cuando necesitamos mostrar los datos que están comprendidos entre dos valores determinados del árbol, debemos hacer lo siguiente:

recorridoAcotado (árbol, inf, sup);

si árbol no está vacío

 si el valor en árbol es \geq inf

 si el valor en árbol es \leq sup

 mostrar valor

 recorridoAcotado (hijo_izq_arbol, inf, sup);

 recorridoAcotado (hijo_der_arbol, inf, sup);

 sino

 recorridoAcotado (hijo_izq_arbol, inf, sup);

 sino

 recorridoAcotado (hijo_der_arbol, inf, sup);

Borrar un elemento en el Árbol Binario:

A la hora de borrar un elemento del árbol binario hay que considerar las siguientes situaciones:

1. Si el nodo es una **Hoja**:

Se puede borrar inmediatamente (actualizando direcciones)

2. Si el nodo tiene un **Hijo**:

Si **el nodo tiene un hijo**, el nodo puede ser borrado después que su padre actualice el puntero al hijo del nodo que se quiere borrar.

3. Si el nodo tiene dos Hijos:

- I. Se busca el valor a borrar.

- II. Se busca y selecciona el hijo más a la izquierda del subárbol derecho del nodo a borrar (o el hijo más a la derecha del subárbol izquierdo).
- III. Se intercambia el valor del nodo encontrado por el que se quiere borrar.
- IV. Se llama al borrar a partir del hijo derecho con el valor del nodo encontrado.

borrarElemento (arbol, dato, resultado)

Si arbol es vacío

No se encontró el dato a buscar

Sino

Si el dato en arbol es > dato

borrarElemento (hijo_izq_del_arbol, dato, resultado)

Sino

Si el dato en arbol es < dato

borrarElemento (hijo_der_del_arbol, dato, resultado)

Sino

{se encontró el dato a borrar}

Si tiene solo hijo derecho...

Sino

Si tiene solo hijo izquierdo...

Sino

Buscar el mínimo del subárbol derecho

Reemplazar el valor en arbol por el mínimo

borrarElemento (hijo_der_del_arbol, mínimo, resultado);

Java:

PARTE II

Conceptos básicos de POO. Objeto.

Objeto: **abstracción** de un objeto del mundo real, definiendo qué lo caracteriza (estado interno) y qué acciones sabe realizar (*comportamiento*). Es una entidad que combina en una unidad.

Estado interno: compuesto por datos/atributos que caracterizan al objeto y relaciones con otros objetos con los cuales colabora. Se implementan a través de **variables de instancia**.

Comportamiento: acciones o servicios a los que sabe responder el objeto. Se implementan a través de **métodos de instancia** que operan sobre el estado interno. Los servicios que ofrece al exterior constituyen la **interfaz**.

¿Qué cosas son objetos? ***“Todo es un objeto”***.

Encapsulamiento (ocultamiento de información): Se oculta la implementación del objeto hacia el exterior. Desde el exterior sólo se conoce la interfaz del objeto. Facilita el mantenimiento y evolución del sistema ya que no hay dependencias entre las partes del mismo.

Conceptos básicos de POO. Mensaje.

Envío de Mensaje: provoca la ejecución del método indicado por el nombre del mensaje.

- Puede llevar datos (parámetros del método).
- Puede devolver un dato (resultado del método).

Conceptos básicos de POO. Clase.

Una *clase* describe un conjunto de objetos comunes (mismo tipo). Consta de:

- La declaración de las variables de instancia que implementan el estado del objeto.
- La codificación de los métodos que implementan su comportamiento.

Un objeto se crea a partir de una clase (el objeto es *instancia* de una clase).

Representación gráfica de una clase (Ejemplo: Triangulo):

Triangulo	Nombre de la clase.
lado1, lado2, lado3, colorRelleno, colorLinea	Variables de instancia
double calcularArea () double calcularPerimetro () /* métodos para obtener valores de las variables de instancia. /* métodos para establecer valores de las variables de instancia.	Encabezado de métodos

Conceptos básicos de POO. Instanciación (creación de objeto).

La *instanciación* se realiza enviando un mensaje de creación a la clase.

- Reserva de espacio para el objeto.
- Ejecución el código inicializador o **constructor**.

Un **constructor** puede tomar valores pasados en el mensaje de creación. Inicializa el objeto (variables de instancias) con valores recibidos.

Devuelve la referencia al objeto.

Asociar la referencia a una variable (a través de ella podemos enviarle mensajes al objeto).

Programa orientado a objetos:

- Los programas se organizan como una colección de **objetos** que cooperan entre sí enviándose mensajes.
- Cada objeto es instancia de una **clase**.
- Los objetos se crean a medida que se necesitan.
- El usuario le envía un mensaje a un objeto, en caso de que un objeto conozca a otro puede enviarle un mensaje, así los mensajes fluyen por el sistema.
- Cuando los objetos ya no son necesarios se borran de la memoria.

Desarrollo de Software orientado a Objetos:

- Identificar los objetos a abstraer en nuestra aplicación.
- Identificar las características relevantes de los objetos.

- Identificar las acciones relevantes que realizan los objetos.
- Los objetos con características y comportamiento similar serán instancia de una misma *clase*.

Objetos en Java. Instanciación (creación de objeto).

- Declarar variable para mantener la referencia:
`NombreDeClase miVariable;`
- Enviar a la clase el mensaje de creación y guardar referencia:
`miVariable= new NombreDeClase(valores para inicialización);`
- Se puede unir los dos pasos anteriores:
`NombreDeClase miVariable= new NombreDeClase(...);`

Secuencia de pasos en la instanciación (creación de objeto):

- *Reserva de Memoria.* Las variables de instancia se inicializan a valores por defecto o explícito (si hubiese).
- *Ejecución del Constructor* (código para inicializar variables de instancia con los valores que enviamos en el mensaje de creación).
- *Asignación de la referencia a la variable.*

Objetos en Java. Referencias.

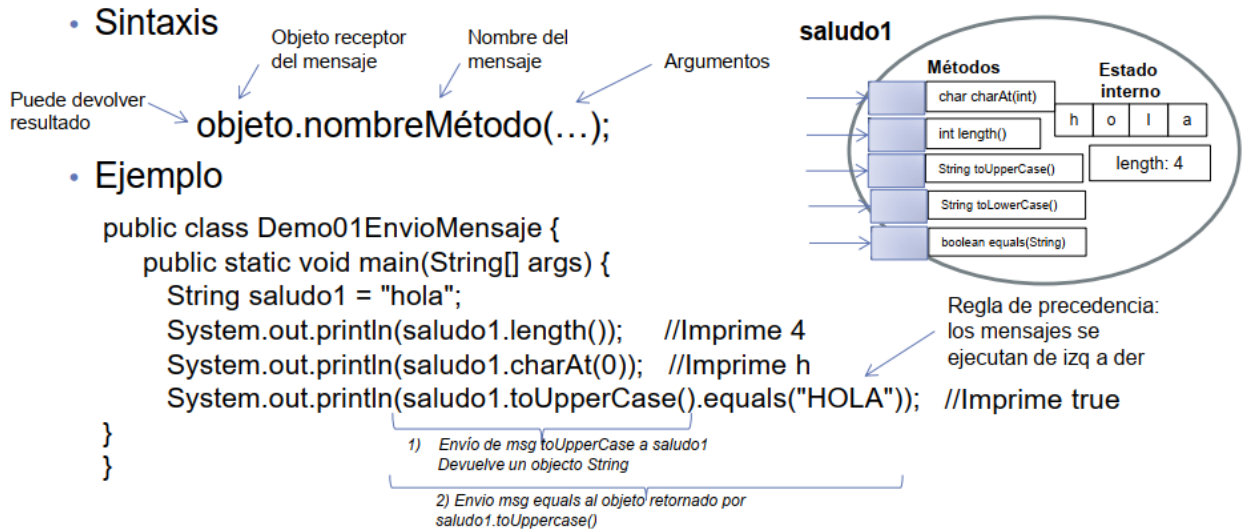
- Referencia a un objeto: ubicación en memoria del objeto.
- El valor null.
- Ejemplo

`String saludo1 = "hola";`

- Asignación: copia referencias.
`String saludo2 = "chau";`
`saludo1 = saludo2;`
- Recolector de basura:
Libera memoria de objetos no referenciados.
- Comparación de objetos con `==` y `!=`
Comparan referencias.
- Comparación del contenido de objetos
Enviar mensaje *equals* al objeto, pasando como argumento el objeto a comparar.

Envío de mensaje al objeto.

<https://docs.oracle.com/javase/7/docs/api/java/lang/String.html>



Declaración de clase:

➤ Sintaxis.

```
public class NombreDeClase {
    /* Declaración del Estado del objeto */
    /* Declaración de Constructor (es) */
    /* Declaración de Métodos que implementan acciones */
}
```

Declaración del Estado.

Estado Interno Datos de tipos primitivos Referencias a otros objetos.	Pseudocodigo TipoPrimitivo nombreDato; NombreDeClase nombreDato;	Ejemplo double precio; String titulo;
Anteponer a la declaración la palabra private para lograr encapsulamiento (ocultamiento de la información).	Las variables de Instancias pueden ser accedidas solo dentro de la clase que las declara.	Ejemplo: private double precio; private String titulo;
En la declaración del dato se puede dar un valor inicial (inicialización explícita).	Ejemplo: private double precio = 10.5 private String titulo = "Java: A Beginner's Guide"	

Ejemplo:

```
public class Libro {  
  
    /* Declaración del Estado */  
  
    private String titulo;  
    private String primerAutor;  
    private String editorial;  
    private int añoEdicion;  
    private String ISBN;  
    private double precio;  
  
}
```

Observaciones: Los datos correspondientes al estado toman un valor por defecto cuando no se inicializan explícitamente. (numéricos => 0; boolean => false; char => "; objetos => null)

Declaración del Comportamiento.

- *Sintaxis:*

public: indica que el método forma parte de la interfaz.

TipoRetorno: tipo de dato primitivo / nombre de clase / void (no retorna dato).

nombreMetodo: verbo seguido de palabras. Convención de nombres.

Lista de parámetros: datos de tipos primitivos u objetos.

- TipoPrimitivo nombreParam // NombreClase nombreParam
- Separación por coma.
- Pasaje por valor únicamente

Declaración de variables locales. Ámbito. Tiempo de vida. (Declaración idem que en Main)

Cuerpo. Código puede utilizar estado y modificarlo (variables de instancia) – devolver resultado **return**.

Declaración del comportamiento. Parámetros.

- Parámetros: únicamente pasaje por valor

a) *Parámetro dato primitivo:*

- **Parámetro formal** recibe **copia del valor** del parámetro actual .
- Si se modifica el parámetro formal, no altera el parámetro actual.

Main

```
Libro l1 = new Libro();  
...  
int x = 1;  
l1.hacerUno(x);  
System.out.println(x); ¿Qué imprime?
```

```
public class Libro{  
    ...  
    public void hacerUno(int y){  
        y++;  
    }  
}
```

Imprime: 1

Declaración del comportamiento. Parámetros.

- Parámetros: únicamente pasaje por valor

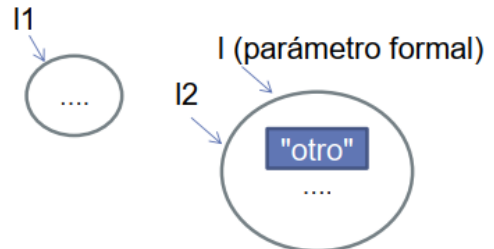
b) *Parámetro objeto:*

- **Parámetro formal** recibe **copia de la referencia** del parámetro actual.
- Si se modifica el estado interno del objeto parám. formal, el cambio en el estado es visible en el parám. actual.

Main

```
Libro l1 = new Libro();  
Libro l2 = new Libro();  
l2.setTitulo("Java");  
...  
l1.hacerDos(l2);  
System.out.println(l2.getTitulo()); ¿Qué imprime?
```

```
public class Libro{  
    ...  
    public void hacerDos(Libro l){  
        l.setTitulo("otro");  
    }  
}
```



Imprime: "otro"

Declaración del comportamiento. Parámetros.

- Parámetros: únicamente pasaje por valor

b) Parámetro objeto:

- **Parámetro formal** recibe **copia de la referencia** del parámetro actual.
- Si se modifica la referencia del parám. formal, el parám. actual sigue referenciando al mismo objeto.

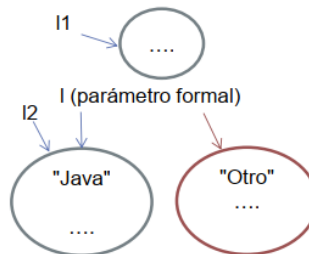
Main

```
Libro l1 = new Libro();
Libro l2 = new Libro();
l2.setTitulo("Java");
...
l1.hacerTres(l2);
System.out.println(l2.getTitulo());
```

¿Qué imprime?

Imprime: "Java"

```
public class Libro{
    ...
    public void hacerTres(Libro l){
        l = new Libro();
        l.setTitulo("otro");
    }
}
```



Definición de clases. Ejemplo

public class Libro {

```
private String titulo;
private String primerAutor;
private String editorial;
private int añoEdicion;
private String ISBN;
private double precio;
```

**Estado
(características)**

**Métodos
(acciones)**

```
public String getTitulo(){
    return titulo;
}
public void setTitulo(String unTitulo){
    titulo = unTitulo;
}
public double getPrecio{
    return precio;
}
```

```
...
public void setPrecio(double unPrecio){
    precio = unPrecio;
}

public String toString(){
    String aux = titulo + " por " + primerAutor + " - " +
        añoEdicion + " - ISBN: " + ISBN;
    return aux;
}
```

aux: variable local al método

Libro.java

Generar una clase para representar libros. Un Libro se caracteriza por: título, nombre del primer autor, editorial, año de edición, ISBN, precio.

El libro debe saber:




- Devolver el valor de cada atributo.
- Modificar el valor de cada atributo.
- Devolver su representación en formato String.

Repr. "Java: A Beginner's Guide por Herbert Schildt - 2014 - ISBN: 978-0071809252"

Concepto de herencia:

Introducción

- Diferentes tipos de objetos con características y comportamiento común.

<u>Triángulo</u>	<u>Círculo</u>	<u>Cuadrado</u>
 <ul style="list-style-type: none"> • Lado1 / lado2 / lado3 • color de línea • color de relleno 	 <ul style="list-style-type: none"> • radio • color de línea • color de relleno 	 <ul style="list-style-type: none"> • lado • color de línea • color de relleno
<ul style="list-style-type: none"> • Devolver y modificar el valor de cada atributo lado1 / lado2 / lado3 color de línea / color de relleno • Calcular el área • Calcular el perímetro 	<ul style="list-style-type: none"> • Devolver y modificar el valor de cada atributo radio color de línea / color de relleno • Calcular el área • Calcular el perímetro 	<ul style="list-style-type: none"> • Devolver y modificar el valor de cada atributo lado color de línea / color de relleno • Calcular el área • Calcular el perímetro

Herencia como solución:

- Permite que la clase **herede** características y comportamiento (atributos y métodos) de otra clase (clase padre o superclase). A su vez, la clase define características y comportamiento propio.
- Ejemplo. Se define **lo común en una clase Figura** y las clases **Triángulo, Círculo y Cuadrado lo heredan**.
- Ventaja: **reutilización de código**.

Herencia. Ejemplo.

- Diagrama de clases.

Las clases forman una jerarquía.

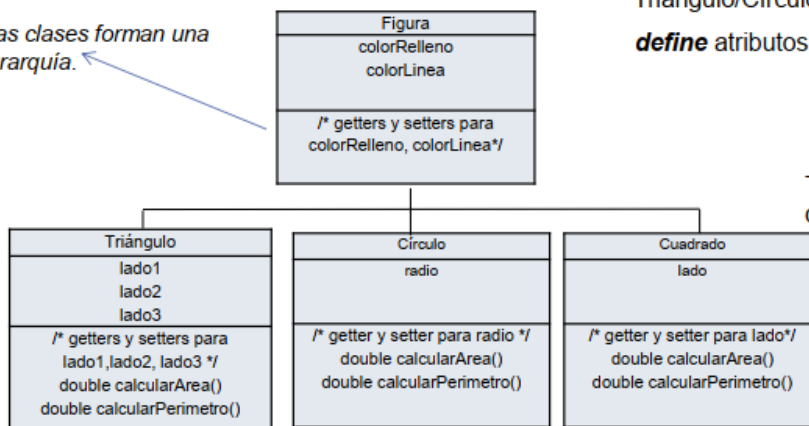


Figura es la **superclase** de Triángulo/Círculo/Cuadrado

define atributos y comportamiento **común**

Triángulo/Círculo/Cuadrado son **subclases** de Figura.

heredan atributos y métodos de Figura

definen atributos y métodos **propios**

definen constructores.

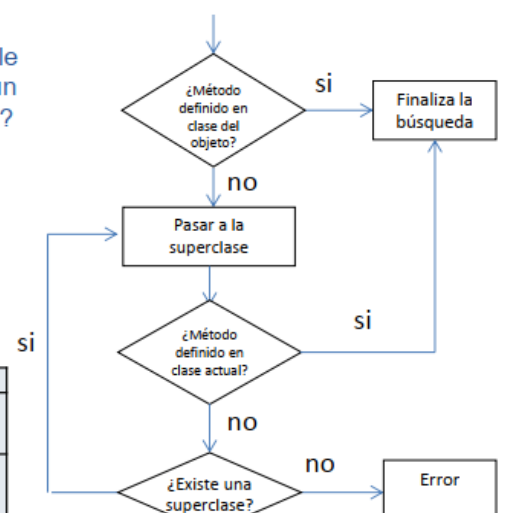
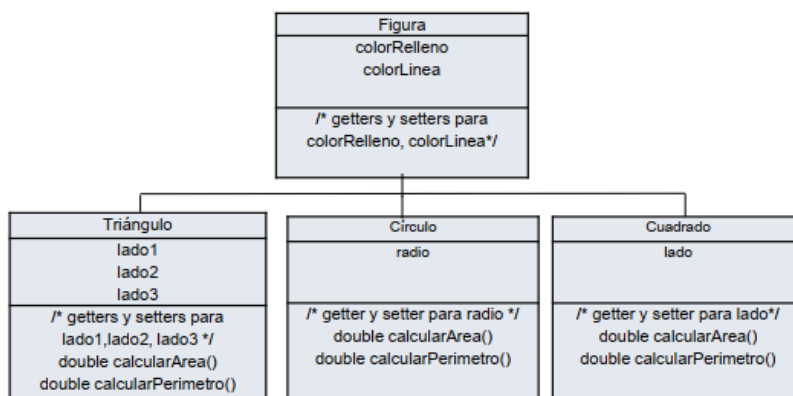
deben implementar `calcularArea()` y `calcularPerimetro()` pero de manera diferente ... **POLIMORFISMO**

Búsqueda de método en la jerarquía de clases

- Ejemplo ... en el main

```
Cuadrado c = new Cuadrado(10,"rojo","negro");
System.out.println(c.calcularArea());
System.out.println(c.getColorRelleno());
```

¿Qué mensajes le puedo enviar a un objeto cuadrado?



Herencia en Java:

- Definición de relación de Herencia. Palabra clave **extends**.

```
public class NombreSubClase extends NombreSuperClase{  
    /* Definir Atributos propios */  
    /* Definir Constructores propios */  
    /* Definir métodos propios */  
}
```

- La **SubClase** hereda
 - ✓ Atributos declarados en la **SuperClase**. Como son privados son accesibles solo en los métodos de la clase que los declaro. En la Subclase accederlos de getters y setters heredados. Ejemplo:
 - getColorRelleno ()
 - setColorRelleno ()
 - ✓ Metodos de instancia declarados en la **SuperClase**.
- La **SubClase** puede declarar
 - ✓ Atributos / métodos / constructores propios.

Clases y métodos Abstractos:

- **Clases Abstractas:** Es una clase que no puede ser instanciada (no se pueden crear objetos). Define características y comportamiento común para un conjunto de clases (subclases). Puede definir **métodos abstractos** (sin implementación) que *deben* ser implementados por las subclases. Por ejemplo:
 - La clase **Figura** es abstracta.
 - **Figura** puede declarar métodos abstractos, por ejemplo: **calcularArea / calcularPerimetro**.
- **Declaración de clase abstracta:** anteponer *abstract* a la palabra class.

```
public abstract class NombreClase {  
    /* Definir atributos */  
    /* Definir métodos no abstractos (con implementación) */  
    /* Definir métodos abstractos (sin implementación) */  
}
```

- ❖ **Declaración de método abstracto:** Sólo se pone el encabezado del método (sin código) anteponiendo **abstract** al tipo de retorno.

public **abstract** TipoRetorno nombreMetodo (lista parámetros formales);

Ejemplo

Superclase

```
public abstract class Figura{
    private String colorRelleno, colorLinea;

    public String getColorRelleno(){
        return colorRelleno;
    }
    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }
    ....

    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}
```

MÉTODOS ABSTRACTOS

Subclase

```
public class Cuadrado extends Figura{
    private double lado;

    /*Constructores*/
    public Cuadrado(double unLado,
                     String unColorR,
                     String unColorL){
        lado=unLado;
        colorRelleno=unColorR,
        colorLinea=unColorL;
    }
}
```

colorRelleno y colorLinea declarados "private" en Figura

setColorRelleno(unColorR);
setColorLinea(unColorL);

El objeto se envía un mensaje a si mismo

setColorRelleno(unColorR) **equivale a**
this.setColorRelleno(unColorR)

this es el objeto que está ejecutando

¿Cómo se busca el método a ejecutar en la jerarquía de clases?

Ejemplo

Superclase

```
public abstract class Figura{
    private String colorRelleno, colorLinea;

    public String getColorRelleno(){
        return colorRelleno;
    }
    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }
    ....

    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}
```

MÉTODOS ABSTRACTOS

Subclase

```
public class Cuadrado extends Figura{
    private double lado;

    /*Constructores*/
    public Cuadrado(double unLado,
                     String unColorR, String unColorL){
        lado=unLado;
        setColorRelleno(unColorR);
        setColorLinea(unColorL);
    }

    /* Metodos getLado y setLado */
    ....

    public double calcularPerimetro(){
        return lado*4;
    }
    public double calcularArea(){
        return lado*lado;
    }
}
```

Implementa

Ejemplo

Superclase

```
public abstract class Figura{
    private String colorRelleno, colorLinea;

    public String getColorRelleno(){
        return colorRelleno;
    }
    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }
    ....

    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}
```

MÉTODOS ABSTRACTOS

Subclase

```
public class Cuadrado extends Figura{
    private double lado;

    /*Constructores*/
    public Cuadrado(double unLado,
        String unColorR, String unColorL){
        lado=unLado;
        setColorRelleno(unColorR);
        setColorLinea(unColorL);
    }

    /* Metodos getLado y setLado */
    ....

    public double calcularPerimetro(){
        return lado*4;
    }
    public double calcularArea(){
        return lado*lado;
    }
}
```

Otra opción:
en vez de utilizar
directamente la v.i.
lado podemos hacer
que el objeto se
envíe un mensaje a
sí mismo para
modificar/obtener
dicho valor.
¿Cómo?

Implementa

Ejemplo

Superclase

```
public abstract class Figura{
    private String colorRelleno, colorLinea;

    public String getColorRelleno(){
        return colorRelleno;
    }
    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }
    ....

    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}
```

MÉTODOS ABSTRACTOS

Subclase

```
public class Cuadrado extends Figura{
    private double lado;

    /*Constructores*/
    public Cuadrado(double unLado,
        String unColorR, String unColorL){
        setLado(unLado);
        setColorRelleno(unColorR);
        setColorLinea(unColorL);
    }

    /* Metodos getLado y setLado */
    ....

    public double calcularPerimetro(){
        return getLado()*4;
    }
    public double calcularArea(){
        return getLado()*getLado();
    }
}
```

Otra opción:
en vez de utilizar
directamente la v.i.
lado podemos hacer
que el objeto se envíe
un mensaje a sí
mismo para
modificar/obtener
dicho valor.
**Buena práctica
en POO**

Implementa

```
public class Cuadrado extends Figura{
    private double lado;
```

Subclases

```
/*Constructores*/
public Cuadrado(double unLado,
    String unColorR,
    String unColorL){
    setLado(unLado);
    setColorRelleno(unColorR);
    setColorLinea(unColorL);
}

/* Metodos getLado y setLado */
/* Métodos calcularArea y calcularPerimetro */

public String toString(){
    String aux = "CR:" + getColorRelleno() +
        "CL:" + getColorLinea() +
        "Lado: " + getLado();
    return aux;
}
}
```

Código
replicado

Solución:
Factorizar código
común en la
superclase e
"invocarlo" desde
las subclases

```
public class Circulo extends Figura{
    private double radio;
```

```
/*Constructores*/
public Circulo(double unRadio,
    String unColorR,
    String unColorL){
    setRadio(unRadio);
    setColorRelleno(unColorR);
    setColorLinea(unColorL);
}

/* Metodos getRadio y setRadio */
/* Métodos calcularArea y calcularPerimetro */

public String toString(){
    String aux = "CR:" + getColorRelleno() +
        "CL:" + getColorLinea() +
        "Radio:" + getRadio();
    return aux;
}
}
```



```

public abstract class Figura{
    private String colorRelleno, colorLinea;

    public Figura(String unCR, String unCL){
        setColorRelleno(unCR);
        setColorLinea(unCL);
    }
    public String toString(){
        String aux = "CR:" + getColorRelleno() +
            "CL:" + getColorLinea();

        return aux;
    }
    public String getColorRelleno(){
        return colorRelleno;
    }
    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }
    ....
    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}

```

```

public class Cuadrado extends Figura{
    private double lado;

    /*Constructores*/
    public Cuadrado(double unLado,
        String unColorR,
        String unColorL){
        super(unColorR, unColorL);
        setLado(unLado);
    }

    /* Metodos getLado y setLado */
    /* Métodos calcularArea y calcularPerimetro */

    public String toString(){
        String aux = super.toString() +
            "Lado:" + getLado();

        return aux;
    }
}

```

super(...)
Invoco al constructor de la superclase.
Al declarar un constructor en la superclase esta invocación debe ir como primera línea

super es el objeto que está ejecutando **super.toString()** =>El objeto se envía un mensaje a si mismo.
La búsqueda del método inicia en la clase superior a la actual.

```

public abstract class Figura{
    private String colorRelleno, colorLinea;

    public Figura(String unCR, String unCL){
        setColorRelleno(unCR);
        setColorLinea(unCL);
    }
    public String toString(){
        String aux = "CR:" + getColorRelleno() +
            "CL:" + getColorLinea();

        return aux;
    }
    public String getColorRelleno(){
        return colorRelleno;
    }
    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }
    ....
    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}

```

```

public class Cuadrado extends Figura{
    private double lado;

    /*Constructores*/
    {
        Añadir a la representación string el
        valor del área. Pero ...
        ¿en qué método toString?
    }

    /* Metodos getLado y setLado */
    /* Métodos calcularArea y calcularPerimetro */

    public String toString(){
        String aux = super.toString() +
            "Lado:" + getLado();

        return aux;
    }
}

```

```

public abstract class Figura{
    private String colorRelleno, colorLinea;

    public Figura(String unCR, String unCL){
        setColorRelleno(unCR);
        setColorLinea(unCL);
    }
    public String toString(){
        String aux = "Area:" + this.calcularArea() +
            "CR:" + getColorRelleno() +
            "CL:" + getColorLinea();

        return aux;
    }

    /* Métodos getters y setters */
    ....

    ....
    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}

```

```

public class Cuadrado extends Figura{
    private double lado;

    En toString() de Figura.
    Evitamos repetir código en subclases.
    ¿Qué calcularArea() se ejecuta?
    ¿Cuándo se determina?

    {

    }

    /* Metodos getLado y setLado */
    /* Métodos calcularArea y calcularPerimetro */

    public String toString(){
        String aux = super.toString() +
            "Lado:" + getLado();

        return aux;
    }
}

```

```

public abstract class Figura{
    private String colorRelleno, colorLinea;

    public Figura(String unCR, String unCL){
        setColorRelleno(unCR);
        setColorLinea(unCL);
    }
    public String toString(){
        String aux = "Area:" + this.calcularArea() +
                    "CR:" + getColorRelleno() +
                    "CL:" + getColorLinea();

        return aux;
    }

    /* Métodos getters y setters */
    ....

    ....
    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}

```

```

public class Cuadrado extends Figura{
    private double lado;

    /* Metodos getLado y setLado */
    /* Métodos calcularArea y calcularPerimetro */

    public String toString(){
        String aux = super.toString() +
                    "Lado:" + getLado();
        return aux;
    }
}

```

Polimorfismo: objetos de clases distintas responden al mismo mensaje de distinta forma.
Binding dinámico: se determina en tiempo de ejecución el método a ejecutar para responder a un mensaje.
Ventaja: Código genérico, reusable.

Resumen de Paradigma orientada a Objeto (POO):

- ❖ **Encapsulamiento:** Permite construir componentes autónomos de software, es decir, independientes de los demás componentes. La independencia se logra ocultando detalles internos (implementación) de cada componente. Una vez encapsulado, el componente se puede ver como una caja negra de la cual sólo se conoce su interfaz.
- ❖ **Herencia:** Permite definir una nueva clase en términos de una clase existentes. La nueva clase hereda automáticamente todos los atributos y métodos de la clase existente, y a su vez puede definir atributos y métodos propios.
- ❖ **Polimorfismo:** Objetos de clase distintas puede responder a mensajes con selector (nombre) sintácticamente idéntico de distinta forma. Permite realizar códigos genéricos, altamente reusables.
- ❖ **Binding Dinámico:** Mecanismo por el cual se determina en tiempo de ejecución el método (código) a ejecutar para responder a un mensaje.

Beneficios de la Programación Orientada a Objetos (POO):

- **Natural:** El programa queda expresado usando términos del problema a resolver, haciendo que sea más fácil de comprender.
- **Fiable:** La POO facilita la etapa de prueba del SW. Cada clase se puede probar y validar independientemente.
- **Reusable:** Las clases implementadas pueden reusarse en distintos programas. Además, gracias a la herencia podemos reutilizar el código de una clase para generar una nueva clase. El polimorfismo también ayuda a crear código más genérico.
- **Fácil de mantener:** Para corregir un problema, nos limitamos a corregirlo en un único lugar.