RESUMÉN DE JAVA (2ª EVALUACIÓN)

ASPECTOS BÁSICOS DEL LENGUAJE

INTRODUCCIÓN A LA ORIENTACIÓN A OBJETOS

1. CLASES

La clase es la unidad fundamental de programación en Java.
Un programa Java Orientado a Objetos está formado por un conjunto de clases. A partir de esas clases se crearán objetos que interactuarán entre ellos enviándose mensajes para resolver el problema.
Una clase representa al conjunto de objetos que comparten una estructura y un comportamiento comunes, por así decirlo, una clase sería un plano de un edificio y un objeto sería el edificio construido.
Puede considerarse como una *plantilla* o *prototipo* de objetos: define los atributos que componen ese tipo de objetos y los métodos que pueden emplearse para trabajar con esos objetos.
Las clases incluyen por tanto atributos y métodos. Los atributos definen el estado de cada objeto de esa clase y los métodos su comportamiento.
Los atributos debemos considerarlos como la zona más interna, oculta a los usuarios del objeto. El acceso a esta zona se realizará a través de los métodos.

La sintaxis general para definir una clase en Java es:

```
[modificadorDeAcceso] class NombreClase [extends NombreSuperClase]
                                [implements NombreInterface1, NombreInterface2, … ] {

    //atributos de la clase (0 ó más atributos)
    [modificadorDeAcceso]  tipo nombreAtributo;


    //métodos de la clase (0 ó más métodos)
    [modificadorDeAcceso] tipoDevuelto nombreMetodo([lista parámetros])
                                                        [throws listaExcepciones]{
            // instrucciones del método
            [return valor;]
    }
}
```

Todo lo que aparece entre corchetes es opcional, por lo tanto la definición mínima de una clase en Java es:

```
class NombreClase{

}
```

Como hemos visto antes, el concepto de clase incluye la idea de ocultación de datos, que básicamente consiste en que no se puede acceder a los datos directamente (zona privada), sino que hay que hacerlo a través de los métodos de la clase.
De esta forma se consiguen dos objetivos importantes:

- Que el usuario no tenga acceso directo a la estructura interna de la clase, para no poder generar código basado en la estructura de los datos.
- Si en un momento dado alteramos la estructura de la clase todo el código del usuario no tendrá que ser retocado.

El **modificador de acceso** se utiliza para definir el nivel de ocultación o visibilidad de los miembros de la clase (atributos y métodos) y de la propia clase.

Los modificadores de acceso **ordenados de menor a mayor visibilidad** son:

MODIFICADOR DE ACCESO	EFECTO	APLICABLE A
private	Restringe la visibilidad al **interior de la clase**. Un atributo o método definido como private solo puede ser usado en el interior de su propia clase.	Atributos Métodos
<Sin modificador>	Cuando no se especifica un modificador, el elemento adquiere el *acceso por defecto o friendly*. También se le conoce como acceso de package (paquete). Solo puede ser usado por las **clases dentro de su mismo paquete**.	Clases Atributos Métodos
protected	Se emplea en la herencia. El elemento puede ser utilizado por **cualquier clase dentro de su paquete y por cualquier subclase** independientemente del paquete donde se encuentre.	Atributos Métodos
public	Es el nivel máximo de visibilidad. El elemento es visible desde cualquier clase.	Clases Atributos Métodos

class: palabra reservada para crear una clase en Java.

NombreClase: nombre de la clase que se define. Si la clase es pública, el nombre del archivo que la contiene debe coincidir con este nombre. Debe describir de forma apropiada la entidad que se quiere representar. Los nombres deben empezar por mayúscula y si está formado por varias palabras, la primera letra de cada palabra irá en mayúsculas.

extends *NombreSuperclase*: (opcional) extends es la palabra reservada para indicar la herencia en Java. NombreSuperClase es la clase de la que hereda esta clase. Si no aparece extends la clase hereda directamente de una clase general del sistema llamada **Object.**

Object es la raíz de toda la jerarquía de clases en Java.

Es la superclase de las que heredan directa o indirectamente todas las clases Java.

Cuando una clase deriva de otra, se llama **superclase** a la clase base de la que deriva la nueva clase (clase derivada o subclase) **La clase derivada hereda todos los atributos y métodos de su superclase**.

implements *NombreInterface1*, *NombreInterface2*, ... : (opcional) implements es la palabra reservada para indicar que la clase implementa el o los interfaces que se indican separados por comas.

Un interface es un conjunto de constantes y declaraciones de métodos (lo que en C/C++ equivaldría al prototipo) no su implementación o cuerpo.

Si una clase implementa un interface está obligada a implementar todos los métodos de la interface. Veremos los interface más adelante.

- En Java solo puede haber una clase pública por archivo de código fuente .java
- El nombre de la clase pública debe coincidir con el nombre del archivo fuente. Por ejemplo, si el nombre de la clase pública es Persona el archivo será Persona.java
- En una aplicación habrá una clase principal que será la que contenga el método main. Esta clase deberá haber sido declarada como pública

Junto al modificador de acceso pueden aparecer **otros modificadores** aplicables a clases, atributos y métodos:

MODIFICADOR	EFECTO	APLICABLE A
abstract	Aplicado a una clase, la declara como clase abstracta. No se pueden crear objetos de una clase abstracta. Solo pueden usarse como superclases. Aplicado a un método, la definición del método se hace en las subclases.	Clases Métodos
final	Aplicado a una clase significa que no se puede extender (heredar), es decir que no puede tener subclases. Aplicado a un método significa que no puede ser sobrescrito en las subclases. Aplicado a un atributo significa que contiene un valor constante que no se puede modificar	Clases Atributos Métodos
static	Aplicado a un atributo indica que es una variable de clase. Esta variable es única y compartida por todos los objetos de la clase. Aplicado a un método indica que se puede invocar sin crear ningún objeto de su clase.	Atributos Métodos
volatile	Un atributo volatile puede ser modificado por métodos no sincronizados en un entorno multihilo.	Atributos
transient	Un atributo transient no es parte del estado persistente de las instancias.	Atributos
Sincronizad	Métodos para entornos multihilo.	Métodos

Ejemplo clase Persona en Java

```
public class Persona {

   private String nombre;
   private int edad;

   public void setNombre(String nom) {
      nombre = nom;
   }

   public String getNombre() {
```

```
        return nombre;
    }

    public void setEdad(int ed) {
        edad = ed;
    }

    public int getEdad() {
        return edad;
    }
}
```

Ejemplo: Desarrolla la clase adecuada para el siguiente texto:

Cada Pueblo del Oeste tiene varios componentes: establos, cantinas, comisarios, y un par de alborotadores. Un pueblo estándar tendría tres establos y estaría localizado al oeste del Mississippi en alguna fecha alrededor de 1850.

Sol:

```
public class PuebloDelOeste
{
        int establos;
        int cantinas;
        int comisarios;
        int alborotadores;
        String locacion;
        int tiempo;


  public PuebloDelOeste()
  {
        establos = 3;
        locacion = "Oeste de los Estados Unidos";
                tiempo = 1850;
        }
}
```

MIEMBROS DE UNA CLASE: ATRIBUTOS Y MÉTODOS

ATRIBUTOS

Una clase puede tener cero o más atributos.
Sirven para almacenar los datos de los objetos. En el ejemplo anterior almacenan el nombre y la edad de cada objeto Persona.
Se declaran generalmente al principio de la clase.
La declaración es similar a la declaración de una variable local en un método.
La declaración contiene un modificador de acceso de los vistos anteriormente: private, package, protected, public.
Pueden ser variables de tipo primitivo o referencias a objetos.
En la clase Persona se ha declarado edad de tipo primitivo y nombre String. Ambas private y por lo tanto solo accesibles desde los métodos de la propia clase.
private String nombre;
private int edad;
Los atributos toman el valor inicial por defecto:

- 0 para tipos numéricos
- '\0' para el tipo char
- null para String y resto de referencias a objetos.

También se les puede asignar un valor inicial en la declaración aunque lo normal es hacerlo en el constructor.
El valor de los atributos en cada momento determina el estado del objeto.
Podemos distinguir dos tipos de atributos o variables:

- **Atributos de instancia**: son todos los atributos no static. Cada objeto de la clase tiene sus propios valores para estas variables, es decir, cada objeto que se crea incluirá su propia copia de los atributos con sus propios valores.
- **Atributos de clase**: son los declarados **static**. También se llaman atributos estáticos. Un atributo de clase no es específico de cada objeto. Solo hay una copia del mismo y su valor es compartido por todos los objetos de la clase. Un atributo de clase existe y puede utilizarse aunque no existan objetos de la clase. Podemos considerarlo como una *variable global* a la que tienen acceso todos los objetos de la clase.

  Para acceder a un atributo de clase se escibe:

  NombreClase.Atributo;

  Por ejemplo, en la clase Persona podemos añadir un atributo contadorPersonas que indique cuantos objetos de la clase se han creado. Sería un atributo de clase ya que no es un valor propio de cada persona:

  static int contadorPersonas;

  Cada vez que se crea una persona podemos incrementar su valor:

  Personas.contadorPersonas++;

  Si lo declaramos además como private:

  private static int contadorPersonas

  solo podremos acceder al atributo a través de un método.

MÉTODOS

Una clase puede contener cero o más métodos.
Definen el comportamiento de los objetos de la clase.
A través de los métodos se accede a los datos de la clase.
Desde el punto de vista de la POO **el conjunto de métodos de la clase se corresponden con el conjunto de mensajes a los que los objetos de esa clase pueden responder**.
Al conjunto de métodos de una clase se le llama **interfaz** de la clase.
Es conveniente que todas las clases implementen los métodos de acceso ó setters/getters para cada atributo.
Los métodos pueden clasificarse en:

- **Métodos de instancia**: Son todos los métodos no static. Operan sobre las variables de instancia de los objetos pero también tienen acceso a los atributos estáticos.
  La sintaxis de llamada a un método de instancia es:
  idObjeto.metodo(parametros); // Llamada típica a un método de instancia
  Todas las instancias de una clase comparten la misma implementación para un método de instancia.
  Dentro de un método de instancia, el identificador de una variable de instancia hace referencia al atributo de la instancia concreta que hace la llamada al método (suponiendo que el identificador del atributo no ha sido *ocultado* en el método).
- **Métodos de clase**: Son los métodos declarados como **static**. Tienen acceso solo a los atributos estáticos de la clase. No es necesario instanciar un objeto para poder utilizar un método estático.
  Para acceder a un método de clase se escibe:

  NombreClase.metodo;

Por ejemplo para la clase Fecha podemos escribir un método estático que incremente el contador de personas:

```
public static void incrementarContador(){
      contadorPersonas++;
}
```

Para invocar a este método:

Persona.incrementarContador();

La API de Java proporciona muchas clases con métodos estáticos, por ejemplo, los métodos de la clase Math: Math.sqrt(), Math.pow(), etc.

2. OBJETOS

Un objeto es una instancia de una clase. En un programa el objeto se representa mediante una variable. Esta variable contiene la dirección de memoria del objeto.
Cuando se dice que Java no tiene punteros estamos diciendo que Java no tiene punteros que podamos ver y manejar como tales en otros lenguajes como C/C++, pero debemos saber que todas las referencias a un objeto son de hecho punteros, es decir, son variables que contienen la dirección de memoria del objeto que representan.

Para crear un objeto se deben realizar dos operaciones:
- Declaración
- Instanciación

Declaración de un objeto

En la declaración se crea la referencia al objeto, de forma similar a cómo se declara una variable de un tipo primitivo.
La referencia se utiliza para manejar el objeto.
La sintaxis general para declarar un obejto en Java es:

```
NombreClase referenciaObjeto;
```

Por ejemplo, para crear un objeto de la clase Persona creamos su referencia así:

```
Persona p;
```

La referencia tiene como misión almacenar la dirección de memoria del objeto. En este momento la referencia p almacena una dirección de memoria nula (null).

Instanciación de un objeto

Mediante la instanciación de un objeto se reserva un bloque de memoria para almacenar todos los atributos del objeto.
Al instanciar un objeto solo se reserva memoria para sus atributos. No se guardan los métodos para cada objeto. Los métodos son los mismos y los comparten todos los objetos de la clase.
La dirección de memoria donde se encuentra el objeto se asigna a la referencia.
De forma general un objeto se instrancia en Java así:

```
referenciaObjeto = new NombreClase();
```

new es el **operador Java para crear objetos**. Mediante new se asigna la memoria necesaria para ubicar el objeto y devuelve la dirección de memoria donde empieza el bloque asignado al objeto.
Por ejemplo:
p = new Persona();
Las dos operaciones pueden realizarse en la misma línea de código:

```
NombreClase referenciaObjeto = new NombreClase();
```

Por ejemplo:
Persona p = new Persona();
Se ha creado un objeto persona y se ha asignado su dirección a la variable p.

Utilización

Una vez creado manejaremos el objeto a través de su referencia.

En general, el acceso a los atributos se realiza a través del operador punto, que separa al identificador de la referencia del identificador del atributo:
referenciaObjeto.Atributo;
Las llamadas a los métodos para realizar las distintas acciones se llevan a cabo separando los identificadores de la referencia y del método correspondiente con el operador punto:
referenciaObjeto.Metodo([parámetros]);
Por ejemplo:
Para asignar valores a los atributos del objeto p lo debemos hacer a través de sus métodos ya que nombre y edad son private:
p.setNombre("Alonso");
p.setEdad(20);

Si intentamos hacer algo así:
p.edad = 20;
el compilador nos avisará del error ya que intentamos acceder de forma directa a un atributo privado.
La utilización del modificador **private** sirve para implementar una de las características de la programación orientada a objetos: el ocultamiento de la información o **encapsulación**.
La declaración como público de un atributo de una clase no respeta este principio de ocultación de información. Declarándolos como privados, no se tiene acceso directo a los atributos del objeto fuera del código de la clase correspondiente y sólo puede accederse a ellos de forma indirecta a través de los métodos proporcionados por la propia clase.
Una de las ventajas prácticas de obligar al empleo de un método para modificar el valor de un atributo es asegurar la consistencia de la operación.
Por ejemplo, el método setEdad de la clase Persona lo podemos escribir para evitar que se asigne que asignen valores no permitidos para el atributo edad:

```
public void setEdad(int ed) {
      if(ed>0)
         edad = ed;
      else edad = 0;
}
```

Con este método, una instrucción p.setEdad(-20); asegura que no se asignará a edad un valor no válido.
Si el atributo edad fuese público podemos escribir p.edad = -20; provocando que edad contenga un valor no válido.

Podemos crear tantos objetos como sean necesarios:
Persona q = new Persona(); //Crea otro objeto persona

En una asignación del tipo:
q = p;
no se copian los valores de los atributos, sino que se tiene como resultado una única instancia apuntada por dos referencias distintas

El objeto referenciado previamente por q se queda sin referencia (inaccesible).
El **recolector de basura** de Java elimina automáticamente las instancias cuando detecta que no se van a usar más (cuando dejan de estar referenciadas).

3. CONSTRUCTOR EN JAVA

Un **constructor** es un método especial de una clase que se llama automáticamente siempre que se declara un objeto de esa clase.
Su función es inicializar el objeto y sirve para asegurarnos que los objetos siempre contengan valores válidos.

Cuando se crea un objeto en Java se realizan las siguientes operaciones de forma automática:

1. Se asigna memoria para el objeto.
2. Se inicializan los atributos de ese objeto con los valores predeterminados por el sistema.
3. Se llama al constructor de la clase que puede ser uno entre varios.

El constructor de una clase tiene las siguientes características:

Tiene el mismo nombre que la clase a la que pertenece.

En una clase puede haber varios constructores con el mismo nombre y distinto número de argumentos (se puede sobrecargar)

No se hereda.

No puede devolver ningún valor (incluyendo void).

Debe declararse público (salvo casos excepcionales) para que pueda ser invocado desde cualquier parte donde se desee crear un objeto de su clase.

MÉTODO CONSTRUCTOR POR DEFECTO

Si para una clase no se define ningún método constructor se crea uno automáticamente por defecto.
El **constructor por defecto** es un constructor sin parámetros que no hace nada. Los atributos del objeto son iniciados con los valores predeterminados por el sistema.
Por ejemplo, en la clase Fecha:

```
import java.util.*;
public class Fecha {

   private int dia;
   private int mes;
   private int año;

   private boolean esBisiesto() {
      return ((año % 4 == 0) && (año % 100 != 0) || (año % 400 == 0));
   }

   public void setDia(int d) {
     dia = d;
   }

   public void setMes(int m) {
      mes = m;
   }

   public void setAño(int a) {
      año = a;
   }
```

```
public void asignarFecha() {
    Calendar fechaSistema = Calendar.getInstance();
    setDia(fechaSistema.get(Calendar.DAY_OF_MONTH));
    setMes(fechaSistema.get(Calendar.MONTH));
    setAño(fechaSistema.get(Calendar.YEAR));
}

public void asignarFecha(int d) {
    Calendar fechaSistema = Calendar.getInstance();
    setDia(d);
    setMes(fechaSistema.get(Calendar.MONTH));
    setAño(fechaSistema.get(Calendar.YEAR));
}

public void asignarFecha(int d, int m) {
    Calendar fechaSistema = Calendar.getInstance();
    setDia(d);
    setMes(m);
    setAño(fechaSistema.get(Calendar.YEAR));
}

public void asignarFecha(int d, int m, int a) {
    setDia(d);
    setMes(m);
    setAño(a);
}

public int getDia() {
    return dia;
}

public int getMes() {
    return mes;
}

public int getAño() {
    return año;
}

public boolean fechaCorrecta() {
    boolean diaCorrecto, mesCorrecto, anyoCorrecto;
    anyoCorrecto = (año > 0);
    mesCorrecto = (mes >= 1) && (mes <= 12);
    switch (mes) {
        case 2:
            if (esBisiesto()) {
                diaCorrecto = (dia >= 1 && dia <= 29);
            } else {
                diaCorrecto = (dia >= 1 && dia <= 28);
            }
            break;
        case 4:
        case 6:
        case 9:
        case 11:
            diaCorrecto = (dia >= 1 && dia <= 30);
            break;
```

```
            default:
                diaCorrecto = (dia >= 1 && dia <= 31);
        }
        return diaCorrecto && mesCorrecto && anyoCorrecto;
    }
}
```

no se ha definido ningún constructor, por lo que al declarar un objeto el compilador utilizará un constructor por defecto.
En este caso el método constructor por defecto es:

```
Fecha(){}
```

Vamos a añadir un constructor a la clase Fecha para poder iniciar los atributos de los nuevos objetos con valores determinados que se envían como parámetros. Si los valores corresponden a una fecha incorrecta se asigna la fecha del sistema:

```
public Fecha(int d, int m, int a) {
        dia = d;
        mes = m;
        año = a;

        if (!fechaCorrecta()) {
            Calendar fechaSistema = Calendar.getInstance();
            setDia(fechaSistema.get(Calendar.DAY_OF_MONTH));
            setMes(fechaSistema.get(Calendar.MONTH));
            setAño(fechaSistema.get(Calendar.YEAR));
        }
}
```

Con este método constructor podremos crear objetos de la siguiente manera:

Fecha fecha1 = new Fecha(10,2,2011);

Se crea un objeto fecha1 y se le asignan esos valores a sus atributos.

Si se crea un objeto con una fecha no válida:

Fecha fecha2 = new Fecha(10,20,2011);

A dia, mes y año se les asignará los valores del sistema.

Java crea un constructor por defecto si no hemos definido ninguno en la clase, pero si en una clase definimos un constructor ya no se crea automáticamente el constructor por defecto, por lo tanto si queremos usarlo deberemos escribirlo expresamente dentro de la clase.

En este ejemplo hemos definido un método constructor por lo que también es necesario poner el método constructor por defecto.

```
public Fecha(){}
```

Este constructor se invocará cuando se declare un objeto sin parámetros.

La clase tiene ahora dos constructores por lo que podemos crear objetos Fecha de dos formas:

```
Fecha f1 = new Fecha();   //se ejecuta el constructor sin parámetros
Fecha f2 = new Fecha(1,1,2010); //se ejecuta el constructor con parámetros
```

INVOCAR A UN MÉTODO CONSTRUCTOR

Un constructor se invoca cuando se crea un objeto de la clase mediante el operador new.
Si es necesario invocarlo en otras situaciones, la llamada a un constructor solo puede realizarse desde dentro de otro constructor de su misma clase y debe ser siempre la primera instrucción.
Si tenemos varios métodos constructores en la clase, podemos llamar a un constructor desde otro utilizando **this**.
Por ejemplo, si en el constructor con parámetros de la clase Fecha fuese necesario llamar al constructor sin parámetros de la misma clase escribimos:

```
public Fecha(int d, int m, int a) {
        this();  //llamada al constructor sin parámetros
        dia = d;
        …….
}
```

CONSTRUCTOR COPIA

Se puede crear un objeto a partir de otro de su misma clase escribiendo un constructor llamado **constructor copia**.
Este constructor copia los atributos de un objeto existente al objeto que se está creando.
Los constructores copia tiene un solo argumento, el cual es una referencia a un objeto de la misma clase que será desde el que queremos copiar.
Por ejemplo, para la clase Fecha podemos escribir un constructor copia que permita crear objetos con los valores de otro ya existente:

```
Fecha fecha = new Fecha(1,1,2011);   //se invoca al constructor con parámetros

Fecha fecha1 = new Fecha(fecha);  //se invoca al constructor copia
```

El constructor copia que escribimos para la clase es:

```
//constructor copia de la clase Fecha
public Fecha(final Fecha f) {
        dia = f.dia;
        mes = f.mes;
        año = f.año;
}
```

El constructor copia recibe una referencia al objeto a copiar y asigna sus atributos uno a uno a cada atributo del objeto creado.
Declarar el parámetro como final no es necesario pero protege al objeto a copiar de cualquier modificación accidental que se pudiera realizar sobre él.

Un **método abstracto** es un método declarado pero no implementado, es decir, es un método del que solo se escribe su nombre, parámetros y tipo devuelto pero no su código.

Los métodos abstractos se escriben sin llaves {} y con ; al final de la declaración.
Por ejemplo:
public abstract area();
Un método se declara como abstracto porque en ese momento (en esa clase) no se conoce cómo va a ser su implementación.
Por ejemplo: A partir de una clase Polígono se pueden derivar las clases Rectángulo y Triángulo. Ambas clases derivadas usarán un método *área*. Podemos declararlo en Figura como abstracto y dejar que cada clase lo implemente según sus necesidades.

Al incluir el método abstracto en la clase base se obliga a que todas las clases derivadas lo sobrescriban con el mismo formato utilizado en la declaración.
Si una clase contiene un método abstracto se convierte en clase abstracta y debe ser declarada como tal.

La forma general de declarar un método abstracto en Java es:
[modificador] **abstract** tipoDevuelto nombreMetodo([parámetros]);

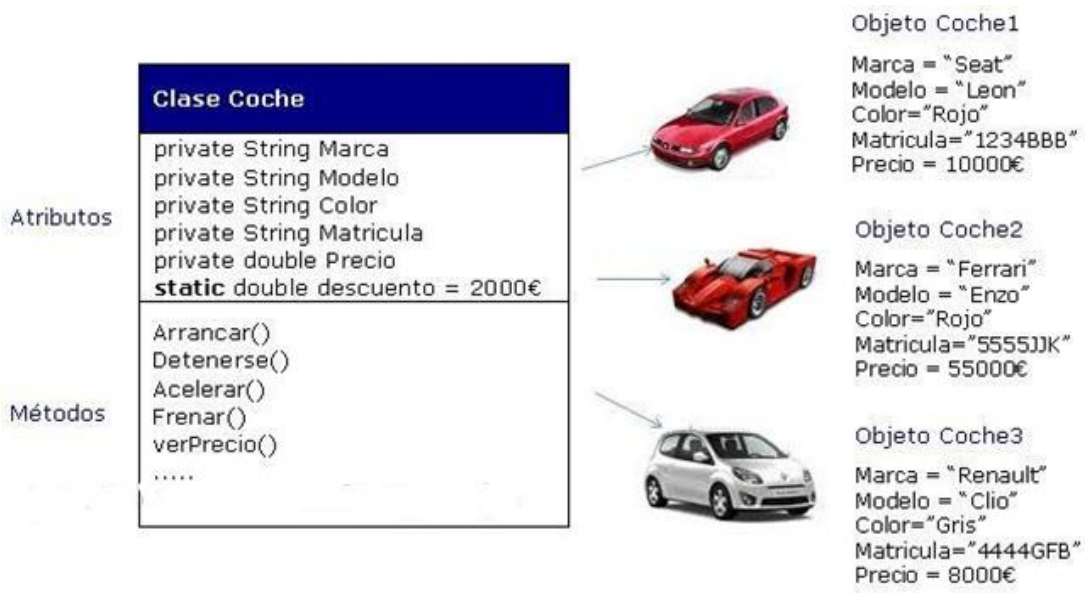
4. ATRIBUTOS Y MÉTODOS ESTÁTICOS O DE CLASE

Los atributos y métodos estáticos también se llaman **atributos de clase** y **métodos de clase**.

Se declaran como **static**.

Supongamos una clase Coche sencilla que se utiliza en un programa de compra-venta de coches usados y de la que se crean 3 objetos de tipo Coche:

La clase contiene 6 atributos: marca, modelo, color, matrícula, precio y descuento. Supongamos que el descuento es una cantidad que se aplica a todos los coches sobre el precio de venta. Como este dato es el mismo para todos los coches y es un valor que se puede modificar en cualquier momento no debe formar parte de cada coche sino que es un dato que deben compartir todos. Esto se consigue declarándolo como **static**.



Un Atributo static:
- No es específico de cada objeto. Solo hay una copia del mismo y su valor es compartido por todos los objetos de la clase.
- Podemos considerarlo como una *variable global* a la que tienen acceso todos los objetos de la clase.
- Existe y puede utilizarse aunque no existan objetos de la clase.

Para acceder a un atributo de clase se escibe:
NombreClase.atributo

Un Método static:
- Tiene acceso solo a los atributos estáticos de la clase.

- No es necesario instanciar un objeto para poder utilizarlo.

Para acceder a un método de clase se escribe:

NombreClase.método()

Ejemplo:

Vamos a escribir una clase Persona que contendrá un atributo contadorPersonas que indique cuantos objetos de la clase se han creado.
contadorPersonas debe ser un atributo de clase ya que no es un valor que se deba guardar en cada objeto persona que se crea, por lo tanto se debe declarar static:

```
static int contadorPersonas;
```

Un ejemplo de uso desde fuera de la clase Persona:
System.out.println(Persona.contadorPersonas);
Si lo declaramos como **private**:

```
private static int contadorPersonas;
```

solo podremos acceder al atributo desde fuera de la clase a través de métodos **static**:

```
public static void incrementarContador(){
        contadorPersonas++;
}
public static int getContadorPersonas() {
        return contadorPersonas;

}
```

En este caso un ejemplo de uso puede ser:
System.out.println(Persona.getContadorPersonas());

Cada vez que se crea una persona se incrementará su valor.
Si no es private, desde otra clase podemos hacerlo así:
Persona.contadorPersonas++;
Si es private, desde otra clase debemos incrementarlo así:
Persona.incrementarContador();

```
//Clase Persona
public class Persona {

    private String nombre;
    private int edad;
    private static int contadorPersonas;

    public Persona() {
    }

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public void setNombre(String nom) {
        nombre = nom;
    }
```

```
    public String getNombre() {
        return nombre;
    }

    public void setEdad(int ed) {
        edad = ed;
    }


    public int getEdad() {
        return edad;
    }

    public static int getContadorPersonas() {
        return contadorPersonas;
    }

    public static void incrementarContador() {
        contadorPersonas++;
    }
}

//Clase Principal
public class Estatico1 {
    public static void main(String[] args) {
        Persona p1 = new Persona("Tomás Navarra", 22);
        Persona.incrementarContador();
        Persona p3 = new Persona("Jonás Estacio", 23);
        Persona.incrementarContador();
        System.out.println("Se han creado: " + Persona.getContadorPersonas() + "
personas");
    }
}
```

En lugar de utilizar el método incrementarContador() cada vez que se crea un objeto, podemos hacer el incremento de la variable estática directamente en el constructor.

El código de la clase Persona y de la clase principal quedaría ahora así:

```
//Clase Persona

public class Persona {

    private String nombre;
    private int edad;
    private static int contadorPersonas;

    public Persona() {
        contadorPersonas++;
    }

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
        contadorPersonas++;
    }
```

```
    public void setNombre(String nom) {
        nombre = nom;
    }

    public String getNombre() {
        return nombre;
    }

    public void setEdad(int ed) {
        edad = ed;
    }

    public int getEdad() {
        return edad;
    }

    public static int getContadorPersonas() {
        return contadorPersonas;
    }
}

//Clase Principal
public class Estatico1 {

    public static void main(String[] args) {
        Persona p1 = new Persona("Tomás Navarra", 22);
        Persona p3 = new Persona("Jonás Estacio", 23);
        System.out.println("Se han creado: " + Persona.getContadorPersonas() + "
personas");
    }
}}
```

5. HERENCIA EN JAVA

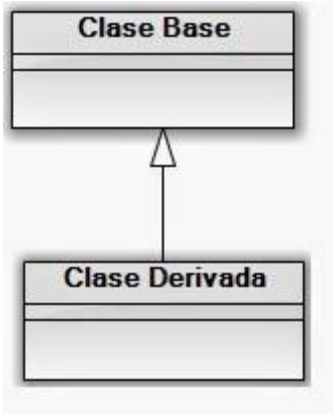
HERENCIA, CLASES ABSTRACTAS, CLASES FINALES, CASTING, OPERADOR INSTANCEOF

La herencia es una de las características fundamentales de la Programación Orientada a Ojetos.

Mediante la herencia podemos **definir una clase a partir de otra ya existente**.

La clase nueva se llama **clase derivada** o subclase y la clase existente se llama **clase base** o superclase.

En UML la herencia se representa con una flecha apuntando desde la clase derivada a la clase base.



La clase derivada hereda los componentes (atributos y métodos) de la clase base.

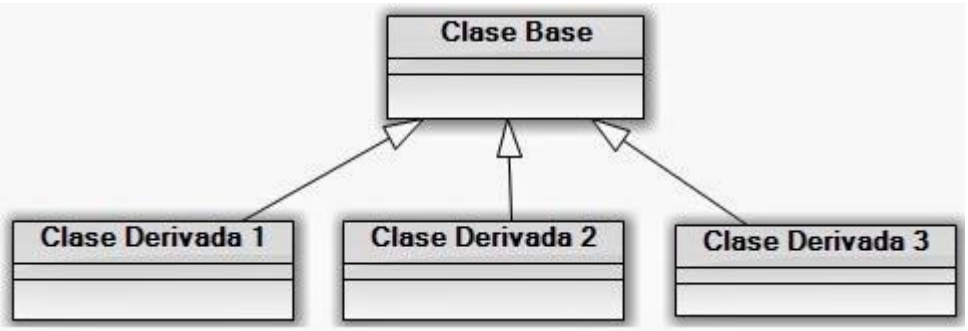
La finalidad de la herencia es:

- **Extender** la funcionalidad de la clase base: en la clase derivada se pueden **añadir** atributos y métodos nuevos.
- **Especializar** el comportamiento de la clase base: en la clase derivada se pueden **modificar** (sobrescribir, override) los métodos heredados para adaptarlos a sus necesidades.

La herencia permite la **reutilización del código**, ya que evita tener que reescribir de nuevo una clase existente cuando necesitamos ampliarla en cualquier sentido. Todas las clases derivadas pueden utilizar el código de la clase base sin tener que volver a definirlo en cada una de ellas.

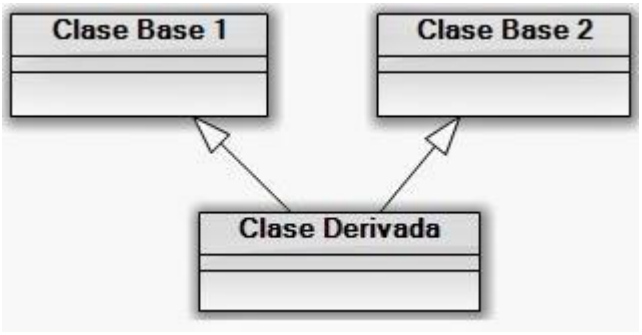
Reutilización de código: El código se escribe una vez en la clase base y se utiliza en todas las clases derivadas.

Una clase base puede serlo de tantas derivadas como se desee: Un solo padre, varios hijos.



Herencia múltiple en Java: Java no soporta la herencia múltiple. Una clase derivada solo puede tener una clase base.

Diagrama UML de herencia múltiple no permitida en Java



La herencia expresa una **relación "ES UN/UNA"** entre la clase derivada y la clase base.

Esto significa que **un objeto de una clase derivada es también un objeto de su clase base**.

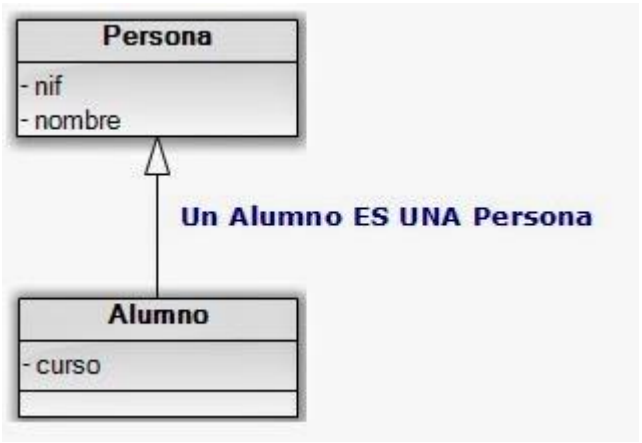
Al contrario NO es cierto. Un objeto de la clase base no es un objeto de la clase derivada.

Por ejemplo, supongamos una clase Vehiculo como la clase base de una clase Coche. Podemos decir que un Coche es un Vehiculo pero un Vehiculo no siempre es un Coche, puede ser una moto, un camión, etc.

Un objeto de una clase derivada es a su vez un objeto de su clase base, por lo tanto se puede utilizar en cualquier lugar donde aparezca un objeto de la clase base. Si esto no fuese posible entonces la herencia no está bien planteada.

Ejemplo de herencia bien planteada:

A partir de una clase Persona que tiene como atributos el nif y el nombre, podemos obtener una clase derivada Alumno. Un Alumno es una Persona que tendrá como atributos nif, nombre y curso.



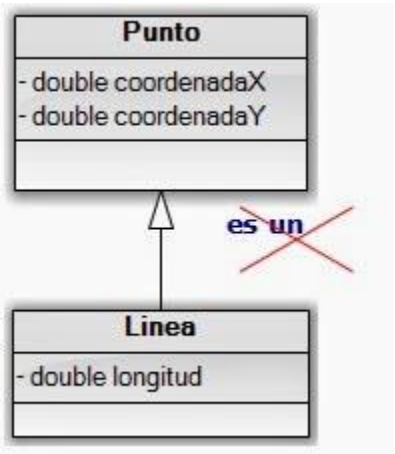
Ejemplo de herencia mal planteada:

Supongamos una clase Punto que tiene como atributos coordenadaX y coordenadaY.

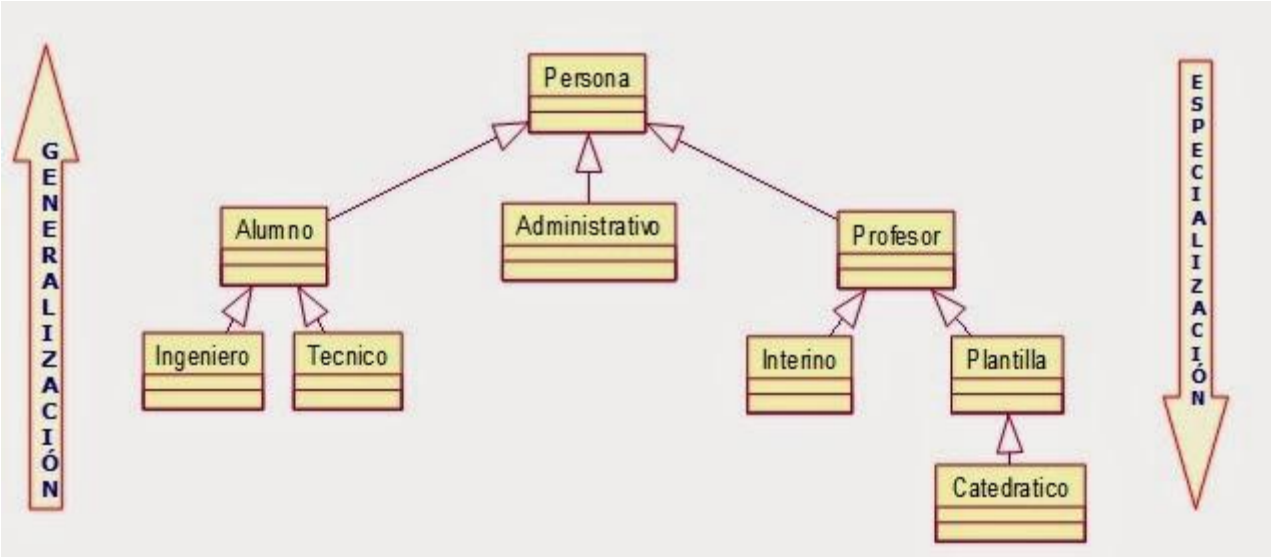
Se puede crear una clase Linea a partir de la clase Punto. Simplificando mucho para este ejemplo, podemos considerar una línea como un punto de origen y una

longitud. En ese caso podemos crear la Clase Linea como derivada de la clase Punto, pero el planteamiento no es correcto ya que no se cumple la relación *ES UN*

Una Linea NO ES un Punto. En este caso no se debe utilizar la herencia.



Una clase derivada a su vez puede ser clase base en un nuevo proceso de derivación, formando de esta manera **una Jerarquía de Clases**.



Las clases más generales se sitúan en lo más alto de la jerarquía. Cuánto más arriba en la jerarquía, menor nivel de detalle.

Cada clase derivada debe implementar únicamente lo que la distingue de su clase base.

En java todas las clases derivan directa o indirectamente de la clase Object.

Object es la clase base de toda la jerarquía de clases Java.

Todos los objetos en un programa Java son Object.

CARACTERÍSTICAS DE LAS CLASES DERIVADAS

Una clase derivada hereda de la clase base sus componentes (atributos y métodos).

Los constructores no se heredan. Las clases derivadas deberán implementar sus propios constructores.

Una clase derivada puede acceder a los miembros públicos y protegidos de la clase base como si fuesen miembros propios.

Una clase derivada no tiene acceso a los miembros privados de la clase base. Deberá acceder a través de métodos heredados de la clase base.

Si se necesita tener acceso directo a los miembros privados de la clase base se deben declarar protected en lugar de private en la clase base.

Una clase derivada puede añadir a los miembros heredados, sus propios atributos y métodos (extender la funcionalidad de la clase).

También puede modificar los métodos heredados (especializar el comportamiento de la clase base).

Una clase derivada puede, a su vez, ser una clase base, dando lugar a una jerarquía de clases.

MODIFICADORES DE ACCESO JAVA

	ACESO EN			
MODIFICADOR	**PROPIA CLASE**	**PACKAGE**	**CLASE DERIVADA**	**RESTO**
private	**SI**	**NO**	**NO**	**NO**
<Sin modificador>	**SI**	**SI**	**NO**	**NO**
protected	**SI**	**SI**	**SI**	**NO**
public	**SI**	**SI**	**SI**	**SI**

HERENCIA EN JAVA. SINTAXIS

La herencia en Java se expresa mediante la palabra **extends**

Por ejemplo, para declarar una clase B que hereda de una clase A:

```
public class B extends A{

….
```

```
}
```

Ejemplo de herencia Java: Disponemos de una clase Persona con los atributos nif y nombre.

```
//Clase Persona. Clase Base

public class Persona {
    private String nif;
    private String nombre;
    public String getNif() {
        return nif;
    }
    public void setNif(String nif) {
        this.nif = nif;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

Queremos crear ahora una clase Alumno con los atributos nif, nombre y curso. Podemos crear la clase Alumno como derivada de la clase Persona. La clase Alumno contendrá solamente el atributo curso, el nombre y el nif son los heredados de Persona:

```
//Clase Alumno. Clase derivada de Persona

public class Alumno extends Persona{
    private String curso;
    public String getCurso() {
        return curso;
    }
    public void setCurso(String curso) {
        this.curso = curso;
    }
}
```

La clase alumno hereda los atributos nombre y nif de la clase Persona y añade el atributo propio curso. Por lo tanto:

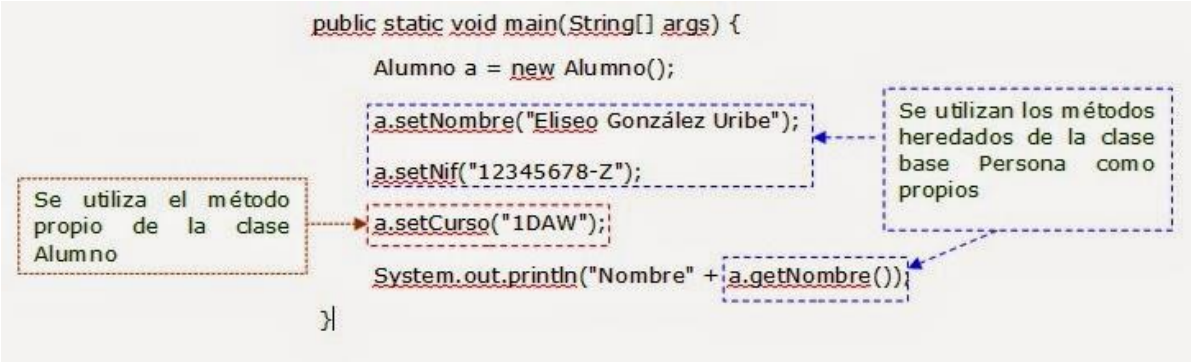
Los atributos de la clase Alumno son nif, nombre y curso.

Los métodos de la clase Alumno son: getNif(), setNif(String nif), getNombre(), setNombre(String nombre), getCurso(), setCurso(String curso).

La clase Alumno aunque hereda los atributos nif y nombre, no puede acceder a ellos de forma directa ya que son privados a la clase Persona. Se acceden a través de los métodos heredados de la clase base.

La clase Alumno puede utilizar los componentes public y protected de la clase Persona como si fueran propios.

Ejemplo de uso de la clase Alumno:



En una jerarquía de clases, cuando un objeto invoca a un método:

1. Se busca en su clase el método correspondiente.

2. Si no se encuentra, se busca en su clase base.

3. Si no se encuentra se sigue buscando hacia arriba en la jerarquía de clases hasta que el método se encuentra.

4. Si al llegar a la clase base raíz el método no se ha encontrado se producirá un error.

REDEFINIR MIEMBROS DE LA CLASE BASE EN LAS CLASES DERIVADAS

Redefinir o modificar métodos de la clase Base (Override)

Los métodos heredados de la clase base se pueden redefinir (también se le llama modificar o sobreescribir) en las clases derivadas.

El método en la clase derivada se debe escribir con el mismo nombre, el mismo número y tipo de parámetros y el mismo tipo de retorno que en la clase base. Si no fuera así estaríamos sobrecargando el método, no redefiniéndolo.

El método sobrescrito puede tener un modificador de acceso menos restrictivo que el de la clase base. Si por ejemplo el método heredado es protected se puede redefinir como public pero no como private porque sería un acceso más restrictivo que el que tiene en la clase base.

Cuando en una clase derivada se redefine un método de una clase base, se oculta el método de la clase base y todas las sobrecargas del mismo en la clase base.

Si queremos acceder al método de la clase base que ha quedado oculto en la clase derivada utilizamos:

```
super.metodo();
```

Si se quiere evitar que un método de la clase Base sea modificado en la clase derivada se debe declarar como método **final**. Por ejemplo:

```
public final void metodo(){
```

```
.....

}
```

Ejemplo:

Vamos a añadir a la clase Persona un método leer() para introducir por teclado los valores a los atributos de la clase. La clase Persona queda así:

```
public class Persona {
   private String nif;
    private String nombre;
    public String getNif() {
        return nif;
    }
    public void setNif(String nif) {
        this.nif = nif;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public void leer(){
        Scanner sc = new Scanner(System.in);
        System.out.print("Nif: ");
        nif = sc.nextLine();
        System.out.print("Nombre: ");
        nombre = sc.nextLine();
    }
}
```

La clase Alumno que es derivada de Persona, heredará este método leer() y lo puede usar como propio.

Podemos crear un objeto Alumno:

Alumno a = new Alumno();

y utilizar este método:

a.leer();

pero utilizando este método solo se leen por teclado el nif y el nombre. En la clase Alumno se debe sobreescribir o modificar este método heredado para que también se lea el curso. El método leer() de Alumno invocará al método leer() de Persona para leer el nif y nombre y a continuación se leerá el curso.

La clase Alumno con el método leer() modificado queda así:

```
//Clase Alumno. Clase derivada de Persona

public class Alumno extends Persona{
    private String curso;
    public String getCurso() {
        return curso;
```

```
    }
    public void setCurso(String curso) {
        this.curso = curso;
    }
    @Override  //indica que se modifica un método heredado
    public void leer(){
        Scanner sc = new Scanner(System.in);
        super.leer(); //se llama al método leer de Persona para leer nif y nombre
        System.out.print("Curso: ");
        curso = sc.nextLine(); //se lee el curso
    }
}
```

Como se ha dicho antes, cuando en una clase derivada se redefine un método de una clase base, se oculta el método de la clase base y todas las sobrecargas del mismo en la clase base. Por eso para ejecutar el método leer() de Persona se debe escribir super.leer();

Redefinir atributos de la clase Base

Una clase derivada puede volver a declarar un atributo heredado (Atributo public o protected en la clase base). En este caso el atributo de la clase base queda oculto por el de la clase derivada.

Para acceder a un atributo de la clase base que ha quedado oculto en la clase derivada se escribe: **super.atributo;**

CONSTRUCTORES Y HERENCIA EN JAVA. CONSTRUCTORES EN CLASES DERIVADAS

Los constructores no se heredan. Cada clase derivada tendrá sus propios constructores.

La clase base es la encargada de inicializar sus atributos.

La clase derivada se encarga de inicializar solo los suyos.

Cuando se crea un objeto de una clase derivada se ejecutan los constructores en este orden:

1. Primero se ejecuta el constructor de la clase base

2. Después se ejecuta el constructor de la clase derivada.

Esto lo podemos comprobar si añadimos a las clases Persona y Alumno sus constructores por defecto y hacemos que cada constructor muestre un mensaje:

```
public class Persona {
    private String nif;
    private String nombre;
    public Persona() {
        System.out.println("Ejecutando el constructor de Persona");
    }
    /////// Resto de métodos
}
```

```
public class Alumno extends Persona{
   private String curso;
   public Alumno() {
      System.out.println("Ejecutando el constructor de Alumno");
   }
   /////// Resto de métodos
}
```

Si creamos un objeto Alumno:

Alumno a = new Alumno();

Se muestra por pantalla:

```
Ejecutando el constructor de Persona

Ejecutando el constructor de Alumno
```

Cuando se invoca al constructor de la clase Alumno se invoca automáticamente al constructor de la clase Persona y después continúa la ejecución del constructor de la clase Alumno.

El constructor por defecto de la clase derivada llama al constructor por defecto de la clase base.

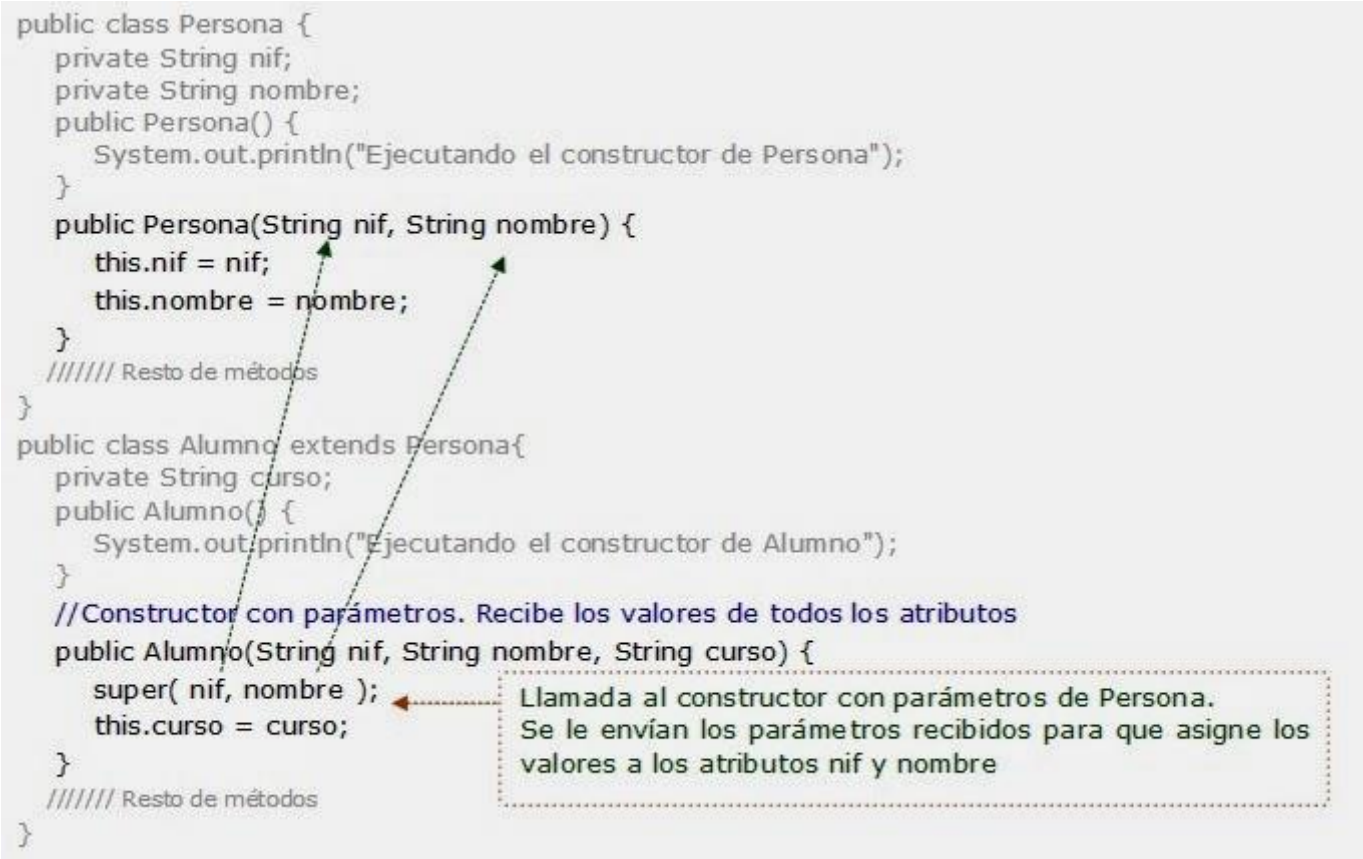
La instrucción para invocar al constructor por defecto de la clase base es: **super();**

Todos los constructores en las clases derivadas contienen de forma implícita la instrucción super() como primera instrucción.

```
public Alumno() {
      super(); //esta instrucción se ejecuta siempre. No es necesario escribirla
      System.out.println("Ejecutando el constructor de Alumno");
}
```

- Cuando se crea un objeto de la clase derivada y queremos asignarle valores a los atributos heredados de la clase base:
- La clase derivada debe tener un constructor con parámetros adecuado que reciba los valores a asignar a los atributos de la clase base.
- La clase base debe tener un constructor con parámetros adecuado.
- El constructor de la clase derivada invoca al constructor con parámetros de la clase base y le envía los valores iniciales de los atributos. Debe ser la primera instrucción.
- La clase base es la encargada de asignar valores iniciales a sus atributos.
- A continuación el constructor de la clase derivada asigna valores a los atributos de su clase.

Ejemplo: en las clases Persona y Alumno anteriores añadimos constructores con parámetros:



Ahora se pueden crear objetos de tipo Alumno y asignarles valores iniciales. Por ejemplo:

Alumno a = new Alumno("12345678-Z","Eliseo Gonzáles Manzano","1DAW");

CLASES FINALES

Si queremos evitar que una clase tenga clases derivadas debe declararse con el modificador **final** delante de class:

```
public final class A{

        ........

}
```

Esto la convierte en clase final. Una clase final no se puede heredar.

Si intentamos crear una clase derivada de A se producirá un error de compilación:

```
public class B extends A{ //extends A producirá un error de compilación

        ........

}
```

CLASES ABSTRACTAS

Una clase abstracta es una clase que NO se puede instanciar, es decir, no se pueden crear objetos de esa clase.

Se diseñan solo para que otras clases hereden de ella.

La clase abstracta normalmente es la raíz de una jerarquía de clases y contendrá el comportamiento general que deben tener todas las subclases. En las clases derivadas se detalla la implementación.

Las clases abstractas:

- Pueden contener cero o más métodos abstractos.

- Pueden contener métodos no abstractos.

- Pueden contener atributos.

Todas las clases que hereden de una clase abstracta deben implementar todos los métodos abstractos heredados.

Si una clase derivada de una clase abstracta no implementa algún método abstracto se convierte en abstracta y tendrá que declararse como tal (tanto la clase como los métodos que siguen siendo abstractos).

Aunque no se pueden crear objetos de una clase abstracta, sí pueden tener constructores para inicializar sus atributos que serán invocados cuando se creen objetos de clases derivadas.

La forma general de declarar una clase abstracta en Java es:

```
[modificador] abstract class nombreClase{

   ……..

}
```

Ejemplo de clase Abstracta en Java: Clase Polígono.

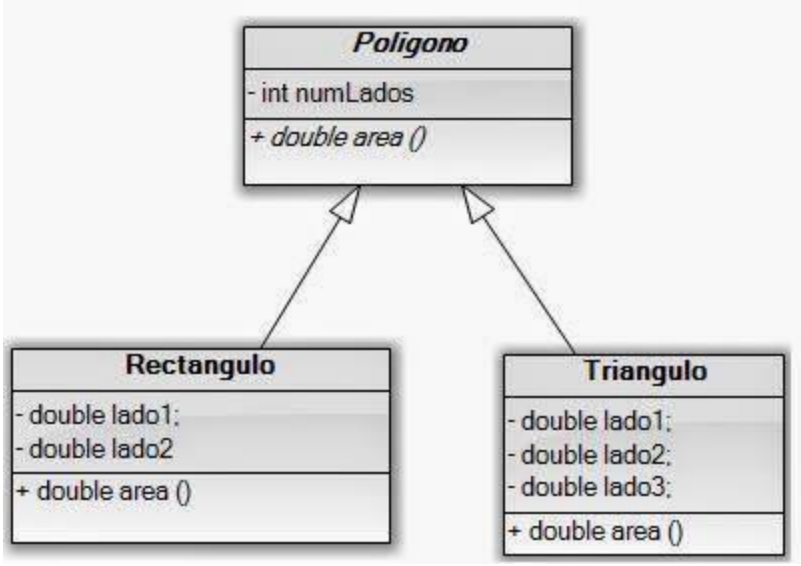
```
//Clase abstracta Poligono

public abstract class Poligono {

   private int numLados;

   public Poligono() {
   }

   public Poligono(int numLados) {
      this.numLados = numLados;
   }

   public int getNumLados() {
      return numLados;
   }

   public void setNumLados(int numLados) {
```

```
        this.numLados = numLados;
    }
    //Declaración del método abstracto area()
    public abstract double area();
}
```

La clase Poligono contiene un único atributo *nublados.* Es una clase abstracta porque contiene el método abstracto area().

A partir de la clase Poligono vamos a crear dos clases derivadas Rectangulo y Triangulo. Ambas deberán implementar el método area(). De lo contrario también serían clases abstractas.

El diagrama UML es el siguiente:



En UML las clases abstractas y métodos abstractos se escriben con su nombre en cursiva.

```
//Clase Rectangulo

public class Rectangulo extends Poligono{

    private double lado1;
    private double lado2;

    public Rectangulo() {
    }

    public Rectangulo(double lado1, double lado2) {
        super(2);
        this.lado1 = lado1;
        this.lado2 = lado2;
```

```
    }

    public double getLado1() {
        return lado1;
    }

    public void setLado1(double lado1) {
        this.lado1 = lado1;
    }

    public double getLado2() {
        return lado2;
    }

    public void setLado2(double lado2) {
        this.lado2 = lado2;
    }

    //Implementación del método abstracto area()
    //heredado de la clase Polígono
    @Override
    public double area(){
        return lado1 * lado2;
    }
}

//Clase Triangulo
public class Triangulo extends Poligono{

    private double lado1;
    private double lado2;
    private double lado3;

    public Triangulo() {
    }

    public Triangulo(double lado1, double lado2, double lado3) {
        super(3);
        this.lado1 = lado1;
        this.lado2 = lado2;
        this.lado3 = lado3;
    }


    public double getLado1() {
        return lado1;
    }

    public void setLado1(double lado1) {
        this.lado1 = lado1;
    }

    public double getLado2() {
        return lado2;
    }
    public void setLado2(double lado2) {
        this.lado2 = lado2;
```

```
    }

    public double getLado3() {
        return lado3;
    }

    public void setLado3(double lado3) {
        this.lado3 = lado3;
    }

    //Implementación del método abstracto area()
    //heredado de la clase Polígono
    @Override
    public double area(){
        double p = (lado1+lado2+lado3)/2;
        return Math.sqrt(p * (p-lado1) * (p-lado2) * (p-lado3));
    }
}
Ejemplo de uso de las clases:
public static void main(String[] args) {
        Triangulo t = new Triangulo(3.25,4.55,2.71);
        System.out.printf("Área del triángulo: %.2f %n" , t.area());
        Rectangulo r = new Rectangulo(5.70,2.29);
        System.out.printf("Área del rectángulo: %.2f %n" , r.area());
}
```

CASTING: CONVERSIONES ENTRE CLASES

UpCasting: Conversiones implícitas

La herencia establece una relación *ES UN* entre clases. Esto quiere decir que un objeto de una clase derivada es también un objeto de la clase base.

Por esta razón:

*Se puede asignar de forma implícita una referencia a un objeto de una clase derivada a una referencia de la clase base. Son **tipos compatibles**.*

*También se llaman conversiones ascendentes o **upcasting**.*

En el ejemplo anterior un triángulo en un Poligono y un cuadrado es un Poligono.

Poligono p;

Triangulo t = new Triangulo(3,5,2);

Como un triángulo es un polígono, se puede hacer esta asignación:

```
p = t;
```

La variable p de tipo Poligono puede contener la referencia de un objeto Triangulo ya que son tipos compatibles.

Cuando manejamos un objeto **a través de una referencia a una superclase** (directa o indirecta) **solo se pueden ejecutar métodos disponibles en la superclase**.

En el ejemplo, la instrucción

p.getLado1();

provocará un error ya que p es de tipo Poligono y el método getLado1() no es un método de esa clase.

Cuando manejamos un objeto a **través de una referencia a una superclase** (directa o indirecta) y **se invoca a un método que está redefinido en las subclases se ejecuta el método de la clase a la que pertenece el objeto no el de la referencia**.

En el ejemplo, la instrucción

p.area();

ejecutará el método area() de Triángulo.

DownCasting: Conversiones explícitas

Se puede asignar una referencia de la clase base a una referencia de la clase derivada, siempre que la referencia de la clase base sea a un objeto de la misma clase derivada a la que se va a asignar o de una clase derivada de ésta.

También se llaman conversiones descendentes o **downcasting**.

Esta conversión debe hacerse mediante un **casting**.

Siguiendo con el ejemplo anterior:

```
Poligono p = new Triangulo(1,3,2);  //upcasting

Triangulo t;

t = (Triangulo) p; //downcasting
```

Esta asignación se puede hacer porque p contiene la referencia de un objeto triángulo.

Las siguientes instrucciones provocarán un error de ejecución del tipo **ClassCastException**:

```
Triangulo t;

Poligono p1 = new Rectangulo(3,2);

t = (Triangulo)p1;   //----> Error de ejecución
```

p1 contiene la referencia a un objeto Rectangulo y no se puede convertir en una referencia a un objeto Triangulo. No son tipos compatibles.

EL OPERADOR instanceof

Las operaciones entre clases y en particular el downcasting requieren que las clases sean de tipos compatibles. Para asegurarnos de ello podemos utilizar el operador instanceof.

instanceof devuelve true si el objeto es instancia de la clase y false en caso contrario.

La sintaxis es:

```
Objeto instanceof Clase
```

Ejemplo:

```
Triangulo t;

Poligono p1 = new Rectangulo(3,2);

if(p1 instanceof Triangulo)

  t = (Triangulo)p1;

else

  System.out.println("Objetos incompatibles");
```

1. Serialización y persistencia de objetos

La **serialización** es la transformación de un objeto en una secuencia de bytes que pueden ser posteriormente leídos para reconstruir el objeto original.

El objeto serializado pueda guardarse en un fichero o puede enviarse por red para reconstruirlo en otro lugar. Puede crearse en un sistema Windows y enviarlo. por ejemplo, a otro sistema que utilice Linux.

Guardar objetos de forma que existan cuando la aplicación haya terminado se conoce como **persistencia**.

Para poder transformar el objeto en una secuencia de bytes, el objeto debe ser **serializable**.

Un objeto es **serializable** si su clase implementa la interface Serializable.

La interface Serializable se encuentra en el paquete java.io Es una interface vacía. No contiene ningún método.

```
public interface Serializable{

}
```

Sirve para indicar que los objetos de la clase que lo implementa se pueden serializar. Solo es necesario que una clase la implemente para que la máquina virtual pueda serializar los objetos.

Si un objeto contiene atributos que son referencias a otros objetos éstos a su vez deben ser serializables.

Todos los tipos básicos Java son serializables, así como los arrays y los String.

2. Persistencia de Objetos en Ficheros

Para escribir objetos en un fichero binario en Java se utiliza la **clase ObjectOutputStream** derivada de OutputStream.

Un objeto ObjectOutputStream se crea a partir de un objeto FileOutputStream asociado al fichero.

El constructor de la clase es:

```
ObjectOutputStream(OutputStream nombre);
```

Lanza una excepción **IOException**.

Por ejemplo, las instrucciones para crear el fichero personas.dat para escritura de objetos serían éstas:

```
FileOutputStream fos = new FileOutputStream ("/ficheros/personas.dat");

ObjectOutputStream salida = new ObjectOutputStream (fos);
```

La clase proporciona el **método writeObject(Object objeto)** para escribir el objeto en el fichero. Lanza una **IOException**

El método defaultWriteObject() de la clase ObjectOutputStream realiza de forma automática la serialización de los objetos de una clase. Este método se invoca en el método writeObject().

defaultWriteObject() escribe en el stream de salida todo lo necesario para reconstruir los objetos:

- La clase del objeto.

- Los miembros de la clase (atributos).

- Los valores de los atributos que **no** sean **static** o **transient**.

El método defaultReadObject() de la clase ObjectInputStream realiza la *deserialización* de los objetos de una clase. Este método se invoca en el método readObject().

Ejemplo 1:

Ejemplo de persistencia de objetos en fichero en Java

Programa que escribe 3 objetos de tipo Persona en el fichero personas.dat.

La clase Persona es la siguiente:

```
//Clase Persona
import java.io.Serializable;
public class Persona implements Serializable{
    private String nif;
    private String nombre;
    private int edad;
```

```
    public Persona() {
    }

    public Persona(String nif, String nombre, int edad) {
        this.nif = nif;
        this.nombre = nombre;
        this.edad = edad;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }

    public String getNif() {
        return nif;
    }

    public void setNif(String nif) {
        this.nif = nif;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}



//Clase Principal

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class Serial1 {

    public static void main(String[] args) {

        FileOutputStream fos = null;
        ObjectOutputStream salida = null;
        Persona p;

        try {
            //Se crea el fichero
            fos = new FileOutputStream("/ficheros/personas.dat");
            salida = new ObjectOutputStream(fos);
            //Se crea el primer objeto Persona
            p = new Persona("12345678A","Lucas González", 30);
```

```
            //Se escribe el objeto en el fichero
            salida.writeObject(p);
            //Se crea el segundo objeto Persona
            p = new Persona("98765432B","Anacleto Jiménez", 28);
            //Se escribe el objeto en el fichero
            salida.writeObject(p);
            //Se crea el tercer objeto Persona
            p = new Persona("78234212Z","María Zapata", 35);
            //Se escribe el objeto en el fichero
            salida.writeObject(p);

        } catch (FileNotFoundException e) {
            System.out.println("1"+e.getMessage());
        } catch (IOException e) {
            System.out.println("2"+e.getMessage());
        } finally {
            try {
                if(fos!=null) fos.close();
                if(salida!=null) salida.close();
            } catch (IOException e) {
                System.out.println("3"+e.getMessage());
            }
        }

    }
}
```

3. Serialización y Herencia

Para serializar objetos de una jerarquía solamente la clase base tiene que implementar el interface Serializable. No es necesario que las clases derivadas implementen la interfaz.

Si una clase es serializable lo son también todas sus clases derivadas.

Ejemplo 2:

Ejemplo de herencia y serialización:

Programa Java que escribe en un fichero tres objetos de tipo Empleado. Empleado es una clase derivada de la clase Persona del ejemplo anterior. Como la clase Persona es serializable, no es necesario indicar que la clase Empleado también lo es. Empleado es serializable por el hecho de heredar de Persona.

```
//Clase Empleado
public class Empleado extends Persona{
    private double sueldo;
    public Empleado(String nif, String nombre, int edad, double sueldo) {
        super(nif, nombre, edad);
        this.sueldo = sueldo;
    }

    public Empleado() {
    }
```

```
    public double getSueldo() {
        return sueldo;
    }
    public void setSueldo(double sueldo) {
        this.sueldo = sueldo;
    }
}


//Clase principal
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class Serial3 {

    public static void main(String[] args) {
        FileOutputStream fos = null;
        ObjectOutputStream salida = null;
        Empleado emp;
        try {
            fos = new FileOutputStream("/ficheros/personas.dat");
            salida = new ObjectOutputStream(fos);
            emp = new Empleado("12345678A","Lucas González", 30, 1200.40);
            salida.writeObject(emp);
            emp = new Empleado("98765432B","Anacleto Jiménez", 28, 1000);
            salida.writeObject(emp);
            emp = new Empleado("78234212Z","María Zapata", 35, 1100.25);
            salida.writeObject(emp);

        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {
            try {
                if(fos!=null) fos.close();
                if(salida!=null) salida.close();
            } catch (IOException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

4. Serialización y clases compuestas

Cuando una clase contiene un atributo que es una referencia a otro objeto, la clase a la que pertenece dicho atributo también debe ser serializable.

Ejemplo 3:

Ejemplo de serialización de una clase compuesta:

Programa java que escribe en un fichero tres objetos de tipo Alumno. Alumno es una clase derivada de Persona y contiene un atributo Fecha:

```
//Clase Fecha
import java.io.Serializable;
public class Fecha implements Serializable{
    private int dia;
    private int mes;
    private int año;

    public Fecha(int dia, int mes, int año) {
        this.dia = dia;
        this.mes = mes;
        this.año = año;
    }
    public Fecha() {
    }
    public int getAño() {
        return año;
    }

    public void setAño(int año) {
        this.año = año;
    }

    public int getDia() {
        return dia;
    }

    public void setDia(int dia) {
        this.dia = dia;
    }

    public int getMes() {
        return mes;
    }

    public void setMes(int mes) {
        this.mes = mes;
    }

}


//Clase Alumno
public class Alumno extends Persona{
    private Fecha fechaMatricula;

    public Alumno(String nif, String nombre, int edad, Fecha fechaMatricula) {
        super(nif, nombre, edad);
        this.fechaMatricula = new Fecha();
        setFechaMatricula(fechaMatricula);
    }

    public Alumno() {
```

```
    }

    public Fecha getFechaMatricula() {
        return fechaMatricula;
    }

    public void setFechaMatricula(Fecha fechaMatricula) {
        this.fechaMatricula.setDia(fechaMatricula.getDia());
        this.fechaMatricula.setMes(fechaMatricula.getMes());
        this.fechaMatricula.setAño(fechaMatricula.getAño());
    }

}


//Clase principal
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class Serial5 {

    public static void main(String[] args) {
        FileOutputStream fos = null;
        ObjectOutputStream salida = null;
        Alumno a;
        Fecha f;
        try {
            fos = new FileOutputStream("/ficheros/alumnos.dat");
            salida = new ObjectOutputStream(fos);
            f = new Fecha(5,9,2011);
            a = new Alumno("12345678A","Lucas González", 20, f);
            salida.writeObject(a);
            f = new Fecha(7,9,2011);
            a = new Alumno("98765432B","Anacleto Jiménez", 19, f);
            salida.writeObject(a);
            f = new Fecha(8,9,2011);
            a = new Alumno("78234212Z","María Zapata", 21, f);
            salida.writeObject(a);

        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {
            try {
                if(fos!=null) fos.close();
                if(salida!=null) salida.close();
            } catch (IOException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

5. Leer objetos de ficheros

Para leer los objetos contenidos en un fichero binario que han sido almacenados mediante ObjectOutputStream se utiliza la **clase ObjectInputStream** derivada de InputStream.

Un objeto ObjectInputStream se crea a partir de un objeto FileInputStream asociado al fichero.

El constructor de la clase es:

```
ObjectInputStream(InputStream nombre);
```

Lanza una excepción **IOException**.

Por Ejemplo, las instrucciones para crear el objeto ObjectInputStream para lectura de objetos del fichero personas.dat serían éstas:

```
FileInputStream fis = new FileInputStream ("/ficheros/personas.dat");

ObjectInputStream entrada = new ObjectInputStream (fis);
```

La clase proporciona el **método readObject()** que devuelve el objeto del fichero (tipo Object).

Es necesario hacer un **casting** para guardarlo en una variable del tipo adecuado.

Lanza una **IOException**

Ejemplo4:

Ejemplo de lectura de objetos contenidos en un fichero en Java

Programa que lee los objetos del fichero creado en el Ejemplo 1:

```
//Clase principal

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;

public class Serial2 {

   public static void main(String[] args) {

      FileInputStream fis = null;
      ObjectInputStream entrada = null;
      Persona p;

      try {

         fis = new FileInputStream("/ficheros/personas.dat");
         entrada = new ObjectInputStream(fis);
         p = (Persona) entrada.readObject(); //es necesario el casting
         System.out.println(p.getNif() + " " + p.getNombre() + " " + p.getEdad());
         p = (Persona) entrada.readObject();
```

```
            System.out.println(p.getNif() + " " + p.getNombre() + " " + p.getEdad());
            p = (Persona) entrada.readObject();
            System.out.println(p.getNif() + " " + p.getNombre() + " " + p.getEdad());
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
        } catch (ClassNotFoundException e) {
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {
            try {
                if (fis != null) {
                    fis.close();
                }
                if (entrada != null) {
                    entrada.close();
                }
            } catch (IOException e) {
                System.out.println(e.getMessage());
            }
        }

    }
}
```

Ejemplo 5:
Programa Java que lee los objetos del fichero creado en el Ejemplo 2:

```
//Clase Principal
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;


public class Serial4 {

    public static void main(String[] args) {
        FileInputStream fis = null;
        ObjectInputStream entrada = null;
        Empleado emp;
        try {
            fis = new FileInputStream("/ficheros/personas.dat");
            entrada = new ObjectInputStream(fis);
            emp = (Empleado) entrada.readObject();
            System.out.println(emp.getNif() + " "
                               + emp.getNombre() + " " + emp.getEdad() + " " +
emp.getSueldo());
            emp = (Empleado) entrada.readObject();
            System.out.println(emp.getNif() + " "
                               + emp.getNombre() + " " + emp.getEdad() + " " +
emp.getSueldo());
            emp = (Empleado) entrada.readObject();
            System.out.println(emp.getNif() + " "
                               + emp.getNombre() + " " + emp.getEdad() + " " +
emp.getSueldo());
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
```

```
        } catch (ClassNotFoundException e) {
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.out.println(e.getMessage());
        } finally {
            try {
                if (fis != null) {
                    fis.close();
                }
                if (entrada != null) {
                    entrada.close();
                }
            } catch (IOException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

Ejemplo 6:

Programa Java que lee los objetos del fichero creado en el Ejemplo 3:

```
//Clase Principal
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;
import persona.Alumno;

public class Serial6 {

    public static void main(String[] args) {
        FileInputStream fis = null;
        ObjectInputStream entrada = null;
        Alumno a;
        try {
            fis = new FileInputStream("/ficheros/alumnos.dat");
            entrada = new ObjectInputStream(fis);
            a = (Alumno) entrada.readObject();
            System.out.println(a.getNif() + " " + a.getNombre() + " " + a.getEdad()
                          + " " + a.getFechaMatricula().getDia() + "-"
                          + a.getFechaMatricula().getMes() + "-"
                          + a.getFechaMatricula().getAño());
            a = (Alumno) entrada.readObject();
            System.out.println(a.getNif() + " " + a.getNombre() + " " + a.getEdad()
                          + " " + a.getFechaMatricula().getDia() + "-"
                          + a.getFechaMatricula().getMes() + "-"
                          + a.getFechaMatricula().getAño());
            a = (Alumno) entrada.readObject();
            System.out.println(a.getNif() + " " + a.getNombre() + " " + a.getEdad()
                          + " " + a.getFechaMatricula().getDia() + "-"
                          + a.getFechaMatricula().getMes() + "-"
                          + a.getFechaMatricula().getAño());
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
        } catch (ClassNotFoundException e) {
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.out.println(e.getMessage());
```

```
        } finally {
            try {
                if (fis != null) {
                    fis.close();
                }
                if (entrada != null) {
                    entrada.close();
                }
            } catch (IOException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

6. El modificador transient

Un atributo transient no se serializa.

En el siguiente ejemplo, la clase Cliente tiene dos atributos: el nombre del cliente y la contraseña. El atributo contraseña es transient por lo tanto si se serializa para escribir un objeto Cliente en un fichero solo se escribirá el atributo nombre.

```
public class Cliente implements Serializable{
    private String nombre;
    private transient String passWord;
    public Cliente(String nombre, String passWord) {
            this.nombre=nombre;
            this.passWord= passWord;
    }
}
```

6. POLIMORFISMO

POLIMORFISMO EN JAVA, ENLAZADO DINÁMICO

El polimorfismo es una de las características fundamentales de la Programación Orientada a Objetos y está estrechamente relacionado con la herencia.

Una jerarquía de clases, los métodos y clases abstractas, la sobrescritura de métodos y las conversiones entre clases de la jerarquía sientan las bases para el polimorfismo. Es necesario entender bien estos conceptos para comprender el polimorfismo. En esta entrada se explican y se ven algunos ejemplos de herencia.

El polimorfismo se puede definir como la cualidad que tienen los objetos para responder de distinto modo a un mismo mensaje.

Para conseguir un comportamiento polimórfico en un programa Java se debe cumplir lo siguiente:

- Los métodos deben estar declarados (métodos abstractos) y a veces también pueden estar implementados (métodos no abstractos) en la clase base.

- Los métodos debes estar redefinidos en las clases derivadas.

- Los objetos deben ser manipulados utilizando referencias a la clase base.

Ejemplo de polimorfismo en Java:

Vamos a ver más claro todo lo anterior con un ejemplo. Tenemos la siguiente clase abstracta Polígono:

```
//Clase abstracta Poligono
public abstract class Poligono {
   private int numLados;

   public Poligono() {
   }

   public Poligono(int numLados) {
      this.numLados = numLados;
   }

   public int getNumLados() {
      return numLados;
   }


   public void setNumLados(int numLados) {
      this.numLados = numLados;
   }

   //Sobreescritura del método toString() heredado de Object
   @Override
   public String toString(){
      return "Numero de lados: " + numLados;
   }

   //Declaración del método abstracto area()
   public abstract double area();

}
```

Esta clase tiene un atributo entero *numLados*. Además contiene el método abstracto area() y se ha modificado (Override) el método toString() heredado de Object.

A partir de la clase Poligono vamos a crear las clases Rectangulo y Triangulo como derivadas de ella.

```
//Clase Rectangulo hereda de Poligono
public class Rectangulo extends Poligono{
   private double lado1;
   private double lado2;

   public Rectangulo() {
```

```
    }

    public Rectangulo(double lado1, double lado2) {
        super(2);
        this.lado1 = lado1;
        this.lado2 = lado2;
    }

    public double getLado1() {
        return lado1;
    }

    public void setLado1(double lado1) {
        this.lado1 = lado1;
    }

    public double getLado2() {
        return lado2;
    }

    public void setLado2(double lado2) {
        this.lado2 = lado2;
    }

    // Sobreescritura del método toString() heredado de Poligono
    @Override
    public String toString() {
        return "Rectangulo " + super.toString() +
                "\nlado 1 = " + lado1 + ", lado 2 = " + lado2;
    }

    //Implementación del método abstracto area() heredado de Poligono
    @Override
    public double area(){
        return lado1 * lado2;
    }
}


//Clase Triangulo hereda de Poligono
public class Triangulo extends Poligono{
    private double lado1;
    private double lado2;
    private double lado3;
    public Triangulo() {
    }
    public Triangulo(double lado1, double lado2, double lado3) {
        super(3);
        this.lado1 = lado1;
        this.lado2 = lado2;
        this.lado3 = lado3;
    }

    public double getLado1() {
        return lado1;
    }
```

```
    public void setLado1(double lado1) {
        this.lado1 = lado1;
    }

    public double getLado2() {
        return lado2;
    }

    public void setLado2(double lado2) {
        this.lado2 = lado2;
    }

    public double getLado3() {
        return lado3;
    }

    public void setLado3(double lado3) {
        this.lado3 = lado3;
    }

    // Sobreescritura del método toString() heredado de Poligono
    @Override
    public String toString() {
        return "Triangulo " + super.toString() +
                "\nlado 1 = " + lado1 + ", lado 2 = " + lado2 + ", lado 3 = " + lado3;
    }

    //Implementación del método abstracto area() heredado de Poligono
    @Override
    public double area(){
        double p = (lado1+lado2+lado3)/2;
        return Math.sqrt(p * (p-lado1) * (p-lado2) * (p-lado3));
    }
}
```

Vamos a escribir un programa para utilizar estas clases y ver como funciona el polimorfismo. Vamos a crear objetos de tipo Triángulo y Rectángulo y los guardaremos en un ArrayList. En lugar de tener dos arrays distintos uno para triángulos y otro para rectángulos, los guardaremos todos juntos utilizando un ArrayList de Poligonos.

El programa creará objetos, leerá sus datos por teclado y los guardará en el ArrayList de Polígonos. A continuación se recorrerá el array y se mostrarán los datos de cada objeto y su área.

```
//Clase Principal
public class Polimorfismo1 {

    static Scanner sc = new Scanner(System.in);
    // ArrayList de referencias a objetos de la clase base Poligono
    static ArrayList<Poligono> poligonos = new ArrayList<Poligono>();

    public static void main(String[] args) {
        leerPoligonos();
        mostrarPoligonos();
    }

    //Se pide por teclado el tipo de Poligono a leer y se ejecuta el método
    //leer correspondiente
```

```
    public static void leerPoligonos() {
        int tipo;
        do {
            do {
                System.out.print("Tipo de poligono 1-> Rectangulo 2-> Triangulo 0->
FIN >>> ");
                tipo = sc.nextInt();
            } while (tipo < 0 || tipo > 2);
            if (tipo != 0) {
                switch (tipo) {
                    case 1:
                        leerRectangulo();
                        break;
                    case 2:
                        leerTriangulo();
                        break;
                }
            }
        } while (tipo != 0);
    }

    //Se crea un rectángulo y se añade al array
    public static void leerRectangulo() {
        double l1, l2;
        System.out.println("Introduzca datos del Rectángulo");
        do {
            System.out.print("Longitud del lado 1: ");
            l1 = sc.nextDouble();
        } while (l1 <= 0);
        do {
            System.out.print("Longitud del lado 2: ");
            l2 = sc.nextDouble();
        } while (l2 <= 0);
        Rectangulo r = new Rectangulo(l1, l2);
        poligonos.add(r);
        //En esta instrucción se produce una conversión implícita o upcasting
        //Se asigna una referencia de una clase derivada (Rectangulo)
        //a una referencia de la clase base (Poligono) ya que el ArrayList es
        //de tipo Poligono
    }

    //Se crea un triángulo y se añade al array
    Public static void leerTriangulo() {
        double l1, l2, l3;
        System.out.println("Introduzca datos del Triangulo");
        do {
            System.out.print("Longitud del lado 1: ");
            l1 = sc.nextDouble();
        } while (l1 <= 0);
        do {
            System.out.print("Longitud del lado 2: ");
            l2 = sc.nextDouble();
        } while (l2 <= 0);
        do {
            System.out.print("Longitud del lado 3: ");
            l3 = sc.nextDouble();
        } while (l3 <= 0);
```

```
        Triangulo t = new Triangulo(l1, l2, l3);
        poligonos.add(t);
        //conversión implícita o upcasting igual que en el método anterior
    }

    public static void mostrarPoligonos() {
        //Se recorre el ArrayList poligonos que contiene
        //referencias a Triangulos y Rectangulos.
        //A p de tipo Poligono se le asignarán mediante upcasting referencias a
objetos
        //de tipo Triangulo o Rectangulo
        for(Poligono p: poligonos){

            System.out.print(p.toString());
            System.out.printf(" area: %.2f %n", p.area());
        }
    }
}
```

En estas dos instrucciones se produce el **polimorfismo**:

```
System.out.print(p.toString());
System.out.printf(" area: %.2f %n", p.area());
```

Después de ejecutar el método leerPoligonos, el ArrayList poligonos contiene mezclados triángulos y rectángulos. Por ejemplo, podría contener lo siguiente:

triangulo	triangulo	rectangulo	triangulo	rectangulo	rectangulo	triangulo	...

Mediante el bucle for

for(Poligono p: poligonos)

se recorre el array y se asigna a la variabale p de tipo Poligono (Clase base) cada elemento del array. En la instrucción p.toString() mediante p se invoca al método toString(). Pero, ¿a qué método toString se ejecutará? Se ejecutará el método toString de las clase derivada a la que pertenece el objeto referenciado por la variable p (Triangulo o Rectangulo)

Lo mismo ocurre cuando se invoca al método area(). Se ejecutará el método area de la clase derivada a la que pertenece el objeto referenciado por p.

En este ejemplo se produce el polimorfismo ya que:

- Los métodos toString() y area() están declarados en la clase base. El método toString está además implementado en la clase base.
- Estos métodos están redefinidos en las clases derivadas.
- Se invocan mediante referencias a la clase base Poligono.

El polimorfismo es posible porque, como ya hemos visto en la entrada correspondiente a la Herencia, cuando se invoca un método mediante una referencia a una clase base, se ejecuta la versión del método correspondiente a la clase del objeto referenciado y no al de la clase de la variable que lo referencia.
Por lo tanto, el método que se ejecuta se decide **durante la ejecución del programa.**
A este proceso de decidir en tiempo de ejecución qué método se ejecuta se le denomina **enlazado dinámico.**
El enlazado dinámico es el mecanismo que **hace posible el polimorfismo**.
El enlazado dinámico es lo opuesto a la habitual vinculación estática o enlazado estático que consiste en decidir en tiempo de compilación qué método se ejecuta en cada caso.
En el programa anterior podemos ver donde se produce enlazado dinámico y donde enlazado estático:

```
public void mostrarPoligonos() {
        for(Poligono p: poligonos){
```

```
            //Para que el polimorfismo se lleve a cabo se está
            //aplicando el enlazado dinámico.
            //Durante la ejecución se decide qué método se ejecuta.
            System.out.print(p.toString());
            System.out.printf(" area: %.2f %n", p.area());
        }
}

public static void main(String[] args) {
        // Enlazado estático. Cuando se compila el programa
        //ya se sabe que serán esos métodos los que se van a ejecutar.
        leerPoligonos();
        mostrarPoligonos();
}
```

7. INTERFACES

Una interface es un archivo .java formado por un conjunto de métodos abstractos. Además puede contener un conjunto de constantes públicas.

Podemos considerar una interface como una **clase abstracta pura:** todos sus métodos son abstractos y si tiene atributos son todos constantes.

En java una interface o interfaz se crea de forma similar a como se crea una clase utilizando la palabra clave **interface** en lugar de class.

```
[public] interface NombreInterface{
            declaraciones de métodos abstractos;
            [atributos constantes;]
}
```

Lo que aparece entre corchetes es opcional.

La interface puede definirse public o sin modificador de acceso, y tiene el mismo significado que para las clases.

Si tiene el modificador public el archivo .java que la contiene debe tener el mismo nombre que la interfaz. Igual que las clases, al compilar el archivo .java de la interface se genera un archivo .class

Todos los métodos de una interface son públicos y abstractos aunque no se indique explícitamente. Por lo tanto, no es necesario escribir en cada método *public abstract*.

Una interface puede contener atributos constantes. **Las constates declaradas son públicas y estáticas** aunque no se indique explícitamente. Por lo tanto, se pueden omitir los modificadores *public static final* cuando se declara el atributo. Las constantes se deben inicializar en la misma instrucción de declaración.

Diferencias entre una clase abstracta y una interface:

- En la interface todo método es abstracto y público sin necesidad de declararlo. Una clase abstracta puede tener métodos abstractos y no abstractos.

- Los atributos declarados en una interface son public static final. Una clase abstracta puede contener también atributos de otro tipo (de instancia, no constantes, private, etc).

Las interfaces juegan un papel fundamental en la creación de aplicaciones Java ya que permiten interactuar a objetos no relacionados entre sí. Utilizando interfaces es posible que clases no relacionadas, situadas en distintas jerarquías de clases sin relaciones de herencia, tengan comportamientos comunes.

Las interfaces definen un protocolo de comportamiento y proporcionan un formato común para implementarlo en las clases.

Los nombres de las interfaces suelen acabar en *able* aunque no es necesario: configurable, arrancable, dibujable, etc.

Ejemplo de interface en Java: interface Relacionable

```
//Interface que define relaciones de orden entre objetos.
public interface Relacionable {
   boolean esMayorQue(Object a);
   boolean esMenorQue(Object a);
   boolean esIgualQue(Object a);
}
```

IMPLEMENTAR UNA INTERFACE

Para indicar que una clase implementa los métodos de una interface se utiliza la palabra clave **implements**.

Las clases que implementan una interfaz deben **implementar todos los métodos abstractos contenidos en la interface**. De lo contrario serán clases abstractas y deberán declararse como tal.

Ejemplo de clase Java que implementa una interface: Una clase Linea que implementa la interfaz Relacionable. En ese caso se dice que la clase **Linea es Relacionable**

```
public class Linea implements Relacionable {

   private double x1;
   private double y1;
   private double x2;
   private double y2;

   public Linea(double x1, double y1, double x2, double y2) {
      this.x1 = x1;
      this.y1 = y1;
      this.x2 = x2;
      this.y2 = y2;
   }

   public double longitud() {
      double l = Math.sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
      return l;
   }


   ////////////////////////////////////////////////////////////////////////////
```

```
    //
    //       Implementación de los tres métodos de la interface            //
    //
    ////////////////////////////////////////////////////////////////////////////

    @Override
    public boolean esMayorQue(Object a) {
        if (a == null) {
            return false;
        }
        if (!(a instanceof Linea)) {
            return false;
        }
        double lon = ((Linea) a).longitud();
        return longitud() > lon;
    }

    @Override
    public boolean esMenorQue(Object a) {
        if (a == null) {
            return false;
        }
        if (!(a instanceof Linea)) {
            return false;
        }
        double lon = ((Linea) a).longitud();
        return longitud() < lon;
    }

    @Override
    public boolean esIgualQue(Object a) {
        if (a == null) {
            return false;
        }
        if (!(a instanceof Linea)) {
            return false;
        }
        double lon = ((Linea) a).longitud();
        return longitud() == lon;

    }

    // Sobreescritura del método toString heredeado de Object

    @Override
    public String toString() {
        return "\nCoordenadas inicio linea: " + x1 + " , " + y1
                + "\nCoordenadas final linea: "+  x2 + " , " + y2
                + "\nLongitud: " + longitud();
    }
}
```

La interfaz la puede implementar cualquier clase por ejemplo una clase Fraccion:

```
public class Fraccion implements Relacionable{
    private int num;
    private int den;
```

```
public Fraccion() {
    this.num = 0;
    this.den = 1;
}
public Fraccion(int num, int den) {
    this.num = num;
    this.den = den;
    simplificar();
}
public Fraccion(int num) {
    this.num = num;
    this.den = 1;
}
public void setDen(int den) {
    this.den = den;
    this.simplificar();
}
public void setNum(int num) {
    this.num = num;
    this.simplificar();
}
public int getDen() {
    return den;
}
public int getNum() {
    return num;
}
//sumar fracciones
public Fraccion sumar(Fraccion f) {
    Fraccion aux = new Fraccion();
    aux.num = num * f.den + den * f.num;
    aux.den = den * f.den;
    aux.simplificar();  //se simplifica antes de devolverla
    return aux;
}
//restar fracciones
public Fraccion restar(Fraccion f) {
    Fraccion aux = new Fraccion();
    aux.num = num * f.den - den * f.num;
    aux.den = den * f.den;
    aux.simplificar();  //se simplifica antes de devolverla
    return aux;
}
//multiplicar fracciones
public Fraccion multiplicar(Fraccion f) {
    Fraccion aux = new Fraccion();
    aux.num = num * f.num;
    aux.den = den * f.den;
    aux.simplificar();  //se simplifica antes de devolverla
    return aux;
}
//dividir fracciones
public Fraccion dividir(Fraccion f) {
    Fraccion aux = new Fraccion();
    aux.num = num * f.den;
    aux.den = den * f.num;
    aux.simplificar();  //se simplifica antes de devolverla
```

```
        return aux;
    }
    //Cálculo del máximo común divisor por el algoritmo de Euclides
    private int mcd() {
        int u = Math.abs(num); //valor absoluto del numerador
        int v = Math.abs(den); //valor absoluto del denominador
        if (v == 0) {
            return u;
        }
        int r;
        while (v != 0) {
            r = u % v;
            u = v;
            v = r;
        }
        return u;
    }
    private void simplificar() {
        int n = mcd(); //se calcula el mcd de la fracción
        num = num / n;
        den = den / n;
    }

    // Sobreescritura del método toString heredeado de Object
    @Override
    public String toString() {
        simplificar();
        return num + "/" + den;
    }

    ////////////////////////////////////////////////////////////////////////////
    //                                                                        //
    //       Implementación de los tres métodos de la interface              //
    //                                                                        //
    ////////////////////////////////////////////////////////////////////////////

    @Override
    public boolean esMayorQue(Object a) {
        if (a == null) {
            return false;
        }
        if (!(a instanceof Fraccion)) {
            return false;
        }
        Fraccion f = (Fraccion) a;
        this.simplificar();
        f.simplificar();
        if ((num/(double)den) <= (f.num/(double)f.den)) {
            return false;
        }
        return true;
    }

    @Override
    public boolean esMenorQue(Object a) {
        if (a == null) {
            return false;
```

```
        }
        if (!(a instanceof Fraccion)) {
            return false;
        }
        Fraccion f = (Fraccion) a;
        this.simplificar();
        f.simplificar();
        if ((num/(double)den) >= (f.num/(double)f.den)) {
            return false;
        }
        return true;
    }

    @Override
    public boolean esIgualQue(Object a) {
        if (a == null) {
            return false;
        }
        if (!(a instanceof Fraccion)) {
            return false;
        }
        Fraccion f = (Fraccion) a;
        this.simplificar();
        f.simplificar();
        if (num != f.num) {
            return false;
        }
        if (den != f.den) {
            return false;
        }
        return true;
    }
}
```

Ejemplo de **interface** Java que *solo* contiene *constantes*:

```
//Interfaz que contiene días relevantes en una aplicación de banca.
public interface IDiasOperaciones {
    int DIA_PAGO_INTERESES = 5;
    int DIA_COBRO_HIPOTECA = 30;
    int DIA_COBRO_TARJETA = 28;
}
```

Una clase que implemente esta interface puede usar las constantes como si fuesen propias. Por ejemplo:

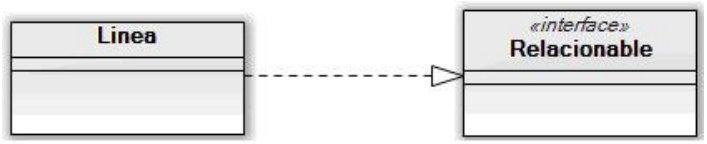
```
//Uso de las constantes dentro de una clase que implementa la interface IDIasOperaciones
public class Banco implements IDiasOperaciones{
    ……….
    public void mostrarInformacionIntereses(){
        System.out.println("El día " + DIA_PAGO_INTERESES
                           + " de cada mes se realiza el pago de intereses");
    }
    ……….
}
```

Una clase que no implemente la interfaz puede usar las constantes escribiendo antes el nombre de la interfaz. Por ejemplo:

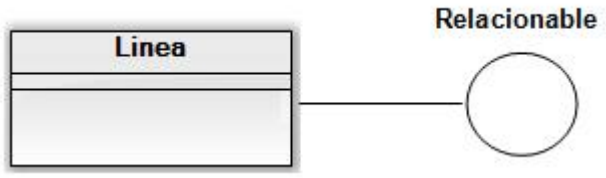
```
//Uso de las constantes dentro de una clase que NO implementa la interface IDIasOperaciones
public static void main(String[] args) {
        ……….
        System.out.println("Los intereses se pagan el día "
                                + IDiasOperaciones.DIA_PAGO_INTERESES);
        System.out.println("La hipoteca se paga el día "
                               + IDiasOperaciones.DIA_COBRO_HIPOTECA);
        ……….
    }
```

Si una clase implementa una interfaz todas sus clases derivadas heredan los métodos implementados en la clase base y las constantes definidas en la interfaz.

En **UML** una clase que implementa una interface se representa mediante una flecha con línea discontinua apuntando a la interface:



O también se puede representar de forma abreviada:

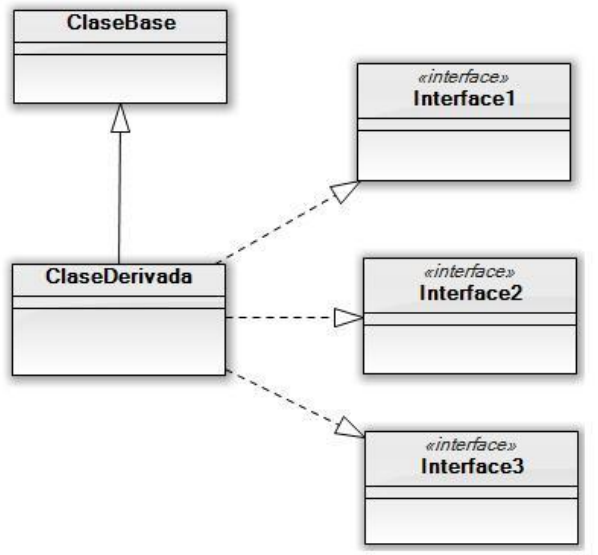


Una clase puede implementar más de una interface. En este caso los nombres de las interfaces se escriben a continuación de *implements* y separadas por comas:

```
public class UnaClase implements NombreInterface1, NombreInterface2, .....
```

El lenguaje Java no permite herencia múltiple, pero las interfaces proporcionan una alternativa para implementar algo parecido a la herencia múltiple de otros lenguajes. En java una clase solo puede tener una clase base pero puede implementar múltiples interfaces.

```
public class ClaseDerivada extends ClaseBase implements Interface1, Interface2,
Interface3, .....
```



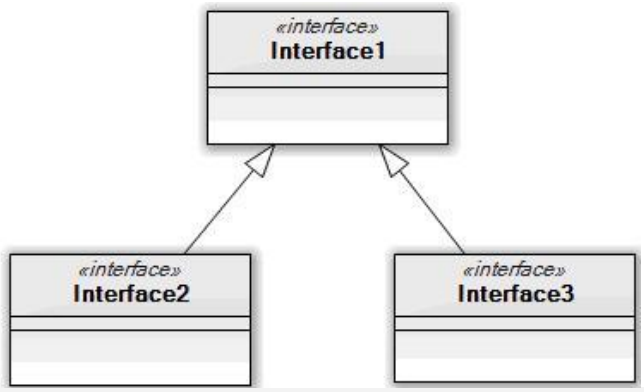
Cuando hay una implementación múltiple, es posible que dos interfaces tengan atributos o métodos con el mismo nombre. La clase que implementa las interfaces recibirá métodos o atributos con el mismo nombre. A esto se le llama **colisión**.

Java establece una serie de reglas para solucionar las colisiones:

- Para las colisiones de nombres de atributos se obliga a especificar el nombre de la interfaz base pertenecen al utilizarlos. (NombreInterfaz.atributo)
- Para las colisiones de los nombres de métodos:
  - Si tiene el mismo nombre y diferentes parámetros se produce sobrecarga de métodos, permitiéndose que existan varias maneras de llamar al método.
  - Si sólo cambia el valor devuelto da un error de compilación indicando que no se pueden implementar los dos.
  - Si coinciden en sus declaraciones, se debe eliminar uno de los dos.

HERENCIA ENTRE INTERFACES

Se puede establecer una jerarquía de herencia entre interfaces, igual que con las clases.




```
public interface Interface2 extends Interface1{
…….
}
```

```
public interface Interface3 extends Interface1{
…….
}
```

En este ejemplo Interface2 e Interface3 heredan los métodos y constantes de Interface1 y además pueden añadir los suyos.

INTERFACES Y POLIMORFISMO

La definición de una interface implica una definición de un nuevo tipo de referencia y por ello **se puede usar el nombre de la interface como nombre de tipo**.

El nombre de una interfaz se puede utilizar en cualquier lugar donde pueda aparecer el nombre de un tipo de datos.

Si se define una variable cuyo tipo es una interface, se le puede asignar un objeto que sea una instancia de una clase que implementa la interface.

Volviendo al ejemplo del principio, las clases Linea y Fraccion implementan la interfaz Relacionable. Entonces podemos escribir las instrucciones:

```
Relacionable r1 = new Linea(2,2,4,1);
Relacionable r2 = new Fraccion(4,7);
```

Utilizando interfaces como tipo se puede aplicar el polimorfismo para clases que no están relacionadas por herencia.

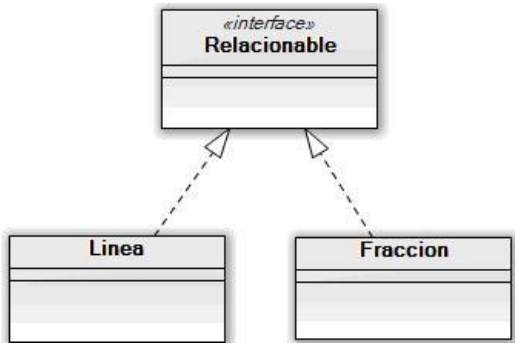
Por ejemplo, podemos escribir:

```
System.out.println(r1);  //ejecuta toString de Linea
System.out.println(r2);  //ejecuta toString de Fraccion
```

Podemos crear un array de referencias de tipo Relacionable y asignarles referencias a objetos de clases que la implementan.

```
Relacionable [] array = new Relacionable[3];
array[0] = new Linea(2,2,4,1);
array[1] = new Fraccion(4,7);
array[2] = new Linea(14,3,22,1);
for(Relacionable r: array)
        System.out.println(r);
```

En este caso dos clases no relacionadas, *Linea* y *Fraccion*, por implementar la misma interfaz *Relacionable* podemos manejarlas a través de referencias a la interfaz y aplicar polimorfismo.



Las interfaces proporcionan más polimorfismo que el que se puede obtener de una simple jerarquía de clases.