INTRODUCCIÓN A LA PROGRAMACIÓN

Semana 5

ING. MARTIN POLIOTTO
Docente a cargo del módulo

Agosto 2020

DIPLOMATURA EN
NUEVAS TECNOLOGÍAS

SECRETARÍA DE EXTENSIÓN UNIVERSITARIA SEU UTN Facultad Regional Córdoba
Ministerio de PROMOCIÓN DEL EMPLEO Y DE LA ECONOMÍA FAMILIAR
Ministerio de CIENCIA Y TECNOLOGÍA
CBA ENTRE TODOS
Oficina de Montevideo

Semana 05

1. Programación Orientada a Objetos

1.1 Conceptos iniciales

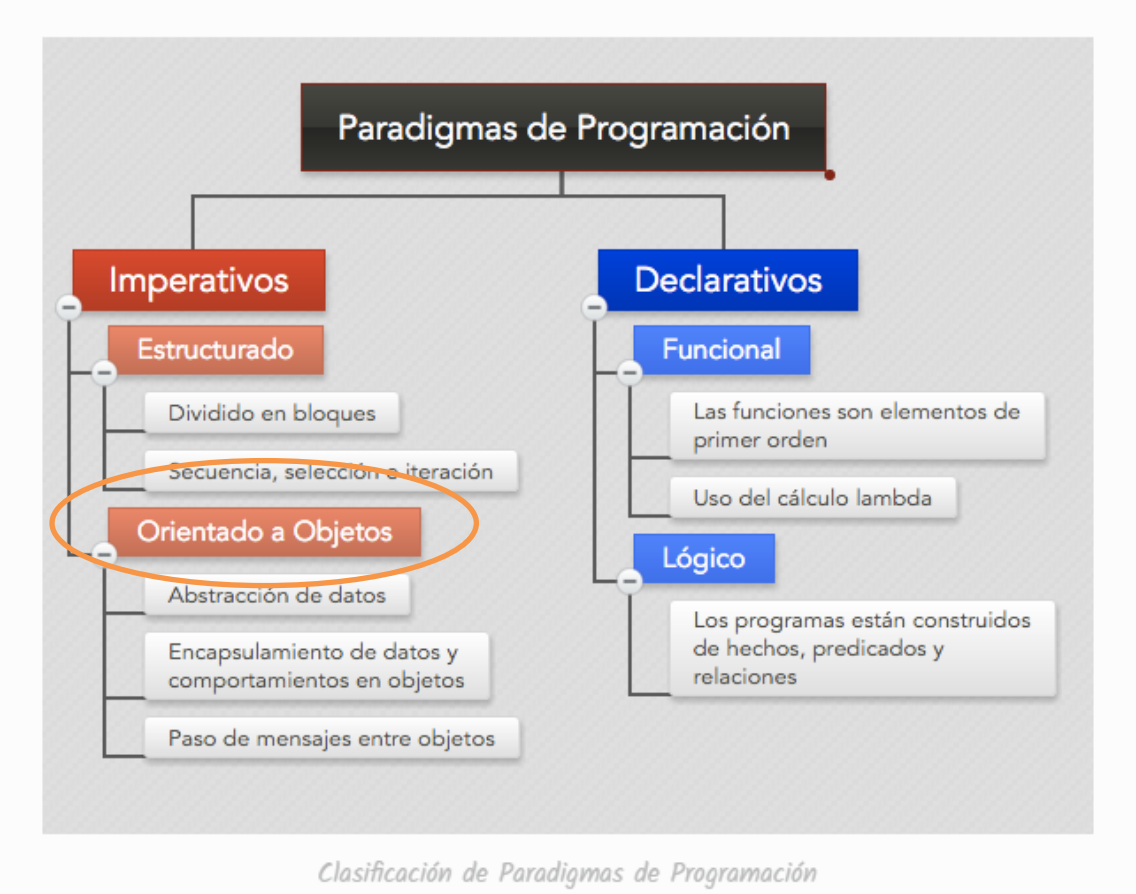
1.1.1 Concepto de **Paradigma**

El concepto de paradigma se utiliza en la vida cotidiana como sinónimo de "marco teórico" para hacer referencia a que algo se toma como **modelo a seguir**. En términos generales, se puede definir al paradigma como la forma de visualizar e interpretar los múltiples conceptos, esquemas o modelos del comportamiento en diversas disciplinas.
Desde las Ciencias Sociales, el concepto se emplea para mencionar a todas aquellas experiencias, creencias, vivencias y valores que repercuten y condicionan el modo en que una persona ve la realidad y actúa en función de ello. Esto quiere decir que **un paradigma es también la forma en que se entiende el mundo.**

1.1.2 Paradigma de la POO

Si nos vamos al ámbito de la Programación también encontramos paradigmas. En este caso nos referimos a un conjunto de convenciones y técnicas que utiliza una comunidad de programadores a la hora de resolver problemas por medio de una computadora.

Una clasificación de estos paradigmas puede graficarse en el siguiente esquema:



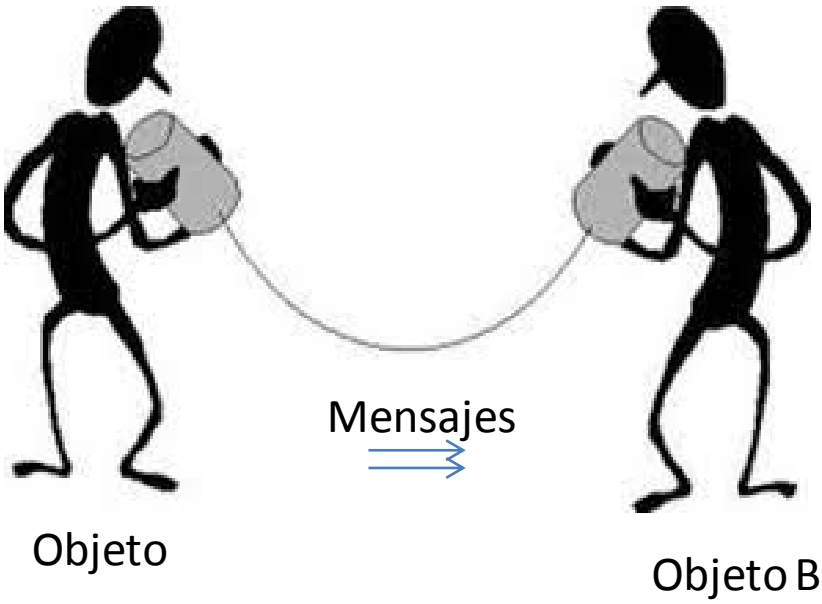
Clasificación de Paradigmas de Programación

Puntualmente nos vamos a centrar en el paradigma resaltado: el paradigma de la **P**rogramación **O**rientado a **O**bjetos (que referenciaremos de ahora como **POO**)

La **POO** busca representar las entidades u objetos del dominio del problema dentro de un programa, en la forma más natural posible. Esto implica que el modelo de programación debería ser lo más parecido a lo que estamos modelando de la realidad. Por ejemplo si nuestro problema se basa en generar el listado de alumnos de los módulos de la diplomatura que se inscribieron en un examen y lo aprobaron, pensaríamos en las entidades: módulos, alumnos, inscripciones, exámenes, etc.

Está basado en la idea de encapsular estado y operaciones en objetos. En general, la programación se resuelve comunicando dichos objetos a través de mensajes. Su principal ventaja es la reutilización de código y su facilidad para pensar soluciones a determinados problemas complejos. El lenguaje de programación elegido para trabajar es **Java**, que pertenece al paradigma orientado a objetos.

Gráficamente:



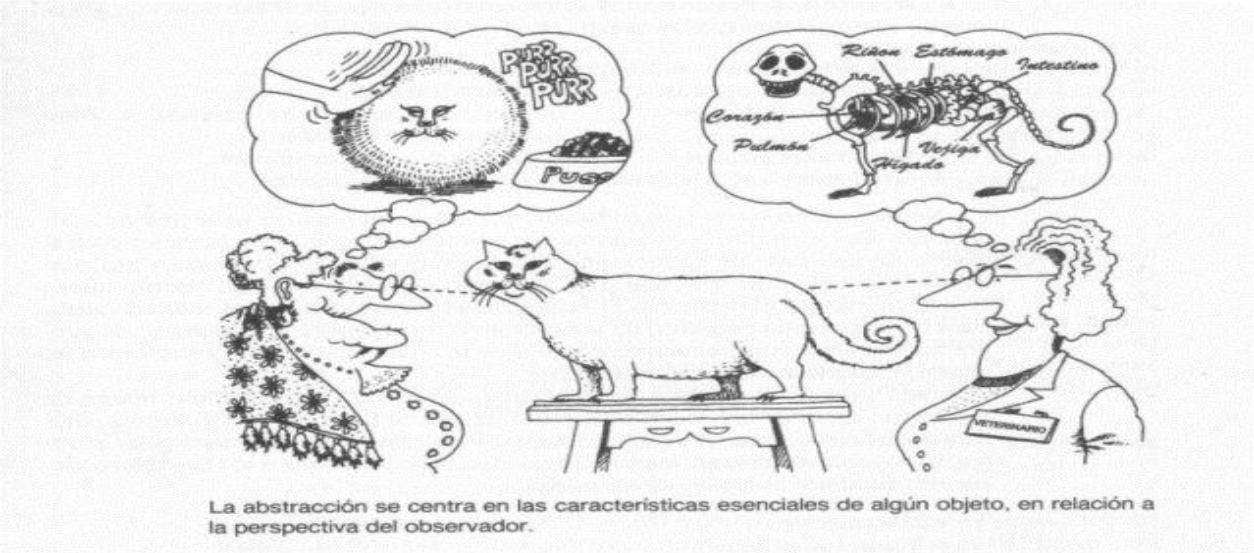
En **POO** todo problema se resuelve como un conjunto de **objetos** que **colaboran** entre sí mediante el paso de **mensajes**.

1.1.2.1 Elementos fundamentales

En POO existen cuatro elementos fundamentales del modelo de objetos:

1. Abstracción
2. Encapsulamiento
3. Modularidad
4. Jerarquía

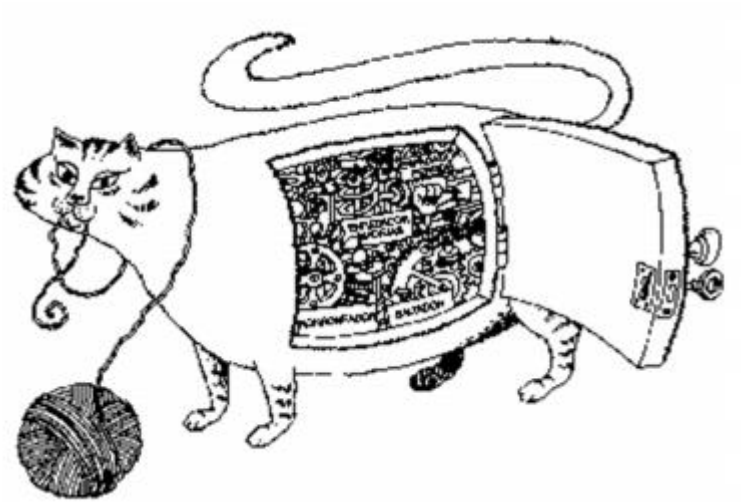
1. **Abstracción:** Capacidad para centrarse en las características o propiedades esenciales de un objeto, dejando de lado otras no tan relevantes para el dominio del problema.



La abstracción se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador.

- Mediante la abstracción vamos a definir nuestras **Clases.**

2. **Encapsulamiento:** es el mecanismo mediante el cual ocultamos la implementación de un objeto.



- No permite ver a los objetos como **cajas negras**. Es decir, quedan ocultos los detalles de implementación.
- Una clase debe tener dos partes: una interfaz (captura solo su vista externa) y una implementación (comprende la representación de la abstracción). La interfaz de una clase es el único lugar donde se declaran todas las suposiciones que un cliente puede hacer acerca de todas las instancias de una clase; la implementación encapsula detalles acerca de los cuales ningún cliente puede realizar suposiciones.

3. **Modularidad**: es la propiedad que tiene un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados

- En la definición anterior **cohesión** se refiere al grado de relación que tienen los componentes que conforman el sistema, es decir en qué grado comparten propiedades comunes. Lo esperado es que cada componente esté formado por elementos altamente relacionados entre sí. Por otro lado, el **acoplamiento** se refiere al grado de relaciones que tiene el componente con otros componentes o cuanto depende de otros para poder realizar sus tareas, en este caso esperamos que los componentes del sistema tengan un grado de acoplamiento bajo, lo que hace más fácil la tarea de mantener cada componente de forma individual.

4. **Jerarquía**: es una clasificación u ordenación de abstracciones.

- Frecuentemente un conjunto de abstracciones forma una jerarquía y la identificación de esas jerarquías en el diseño simplifica en gran medida la comprensión del problema.
- Nuestros modelos de programación se construyen mediante jerarquías de clases heredadas.

Adicionalmente mencionamos un elemento extra: **el Polimorfismo.** Este último, permite que el mismo objeto responda al mismo mensaje de maneras diferentes dependiendo de su tipo. Si bien es un principio muy importante, lo abordaremos en con mayor detalle en las clases siguientes.

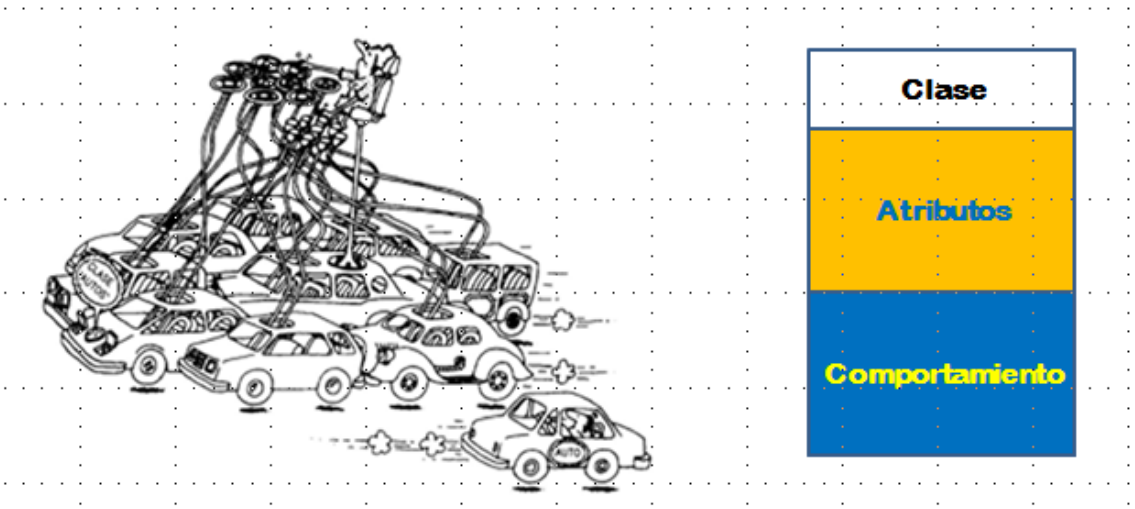
1.2 Clases y Objetos

1.2.1 Clase

Para describir objetos que responden a las mismas características de forma y comportamiento, se declara una **clase.** La definición de una clase incluye esencialmente dos elementos:

- **Atributos**: Son variables que se declaran dentro de la clase, y sirven para indicar la forma de cada objeto representado por esa clase. Los atributos, de alguna manera, muestran lo que cada objeto es, o también, lo que cada objeto tiene.
- **Métodos**: Son funciones, procedimientos o rutinas declaradas dentro de la clase, usados para describir el comportamiento de los objetos descriptos por esa clase. Los métodos, de alguna manera, muestran lo que cada objeto hace.

Gráficamente:



Molde, o **definición** a partir de la cual se modelan las **entidades**

Como sabemos, una aplicación o programa Java típico debe incluir un método especial llamado **main()**. Ese método es el primero que se ejecuta cuando se pide lanzar la aplicación, y desde allí se administra la participación de cada instancia u objeto creado.

En Java la definición de una clase se realiza mediante la palabra reservada **class**, y denota el bloque mínimo de código de nuestros programas. Sintéticamente:

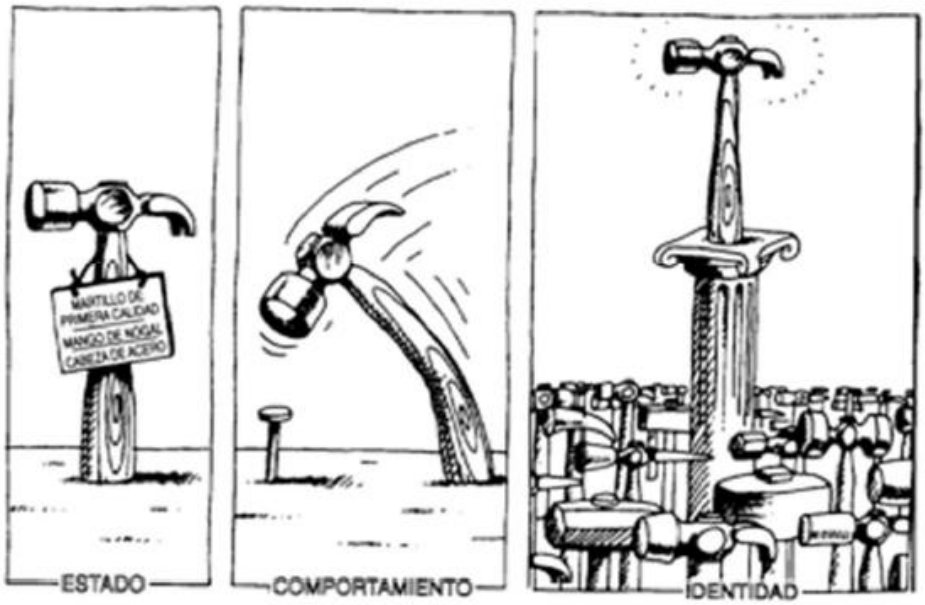
```
public class UnaClase{

    private int id;

    public int getId(){return id;}

}
```

Existe un conocido principio de buena práctica, llamado **Principio de Ocultamiento**, por el cual se sugiere que al definir una clase, **los atributos no sean accesibles** en forma directa desde el exterior de la clase, sino que **se usen métodos** (que deberían ser visibles desde fuera) de la misma para consultar sus valores o modificarlos. Este permite que la clase controle qué valores tiene cada atributo y de qué forma esos valores deberían cambiar. Si se permite el acceso a un atributo de la clase desde fuera de ella, podría provocarse que un valor incorrecto, no validado, sea asignado en ese atributo haciendo que desde allí en adelante algún proceso interno de la clase falle…

1.2.2 Objeto

Un **objeto** es una **instancia** creada a partir de una clase, que tiene:

- *Estado*: conjunto de valores de los atributos de un objeto en un momento dado
- *Comportamiento*: cómo actúa y reacciona un objeto en término de cambios de estado y paso de mensajes.
- *Identidad*: es aquella propiedad de un objeto que lo distingue de todos los demás objetos.



- Para crear objetos utilizamos la palabra reservada **new**, que nos permite "pedir" memoria durante su creación.

  *UnaClase unObjeto = **new** UnaClase();*

  La variable unObjeto la llamaremos **referencia** al objeto creado y almacenará la dirección de memoria donde se

alojará el objeto creado. Para nosotros representa **la identidad** del objeto.

1.3 Métodos de acceso

La forma que Java provee para que un programador obligue a respetar el Principio de Ocultamiento son los llamados **calificadores de acceso**. Se trata de ciertas palabras reservadas que colocadas delante de la declaración de un atributo o de un método de una clase, hacen que ese atributo o ese método tengan accesibilidad más amplia o menos amplia desde algún método que no esté en la clase. Así, los calificadores de acceso en Java pueden ser cuatro:

- **public**: un miembro público es accesible tanto desde el interior de la clase (por sus propios métodos), como desde el exterior de la misma (por métodos de otras clases).
- **private**: sólo es accesible desde el interior de la propia clase, usando sus propios métodos.
- **protected**: aplicable en contextos de herencia (tema que veremos más adelante), hace que un miembro sea público para sus clases derivadas y para clases en el mismo paquete, pero los hace privados para el resto.
- "default": un miembro que no sea marcado con ningún calificador de acceso, asume estatus de acceso por defecto, lo cual significa que será público para clases en el mismo paquete, pero privado para el resto. Notar que la palabra "default" en realidad no es una palabra reservada, sino sólo el nombre del estado en que queda el miembro.

Entre los métodos de una clase, el más característico es el llamado método **constructor**. Un constructor es un método cuyo objetivo básico es el de inicializar los atributos de un objeto en el momento en que ese objeto o instancia se crea.

- Un constructor lleva el mismo nombre que la clase que lo contiene. No se debe indicar **ningún tipo devuelto para él**, y puede recibir parámetros como un método normal.
- Por otra parte, tanto los constructores como cualquier otro método de la clase, pueden ser **sobrecargados**. Eso significa que se pueden definir varias versiones del mismo método. El compilador distingue entre las diversas sobrecargas de un método por la forma de su lista de parámetros. Por lo tanto, si un método tiene varias versiones definidas en una clase, las distintas versiones deben diferir en la cantidad de parámetros, en el tipo de los parámetros, o en ambas características.

1.4 Creando nuestra primer clase

Vamos a ejemplificar todo lo expuesto anteriormente con el desarrollo de una clase **Persona** que siga las siguientes condiciones:

- Sus **atributos** son: nombre, edad, sexo (H hombre, M mujer), peso y altura. No queremos que se accedan directamente a ellos. Por defecto, todos los atributos menos el DNI serán valores según su tipo (0 números, cadena vacía para String, etc.). Sexo será Mujer por defecto, usar una constante para ello.

- Se implantarán los **constructores**:
  - Un constructor por defecto.
  - Un constructor con el nombre, edad y sexo, el resto por defecto.
- Los **métodos propios** que se implementarán son:
  - **calcularIMC()**: calculara si la persona está en su peso ideal (peso en kg/(altura^2 en m)), si esta fórmula devuelve un valor menor que 20, la función devuelve un -1, si devuelve un número entre 20 y 25 (incluidos), significa que está por debajo de su peso ideal la función devuelve un 0 y si devuelve un valor mayor que 25 significa que tiene sobrepeso, la función devuelve un 1.
  - **esMayorDeEdad()**: indica si es mayor de edad, devuelve un booleano.
  - **toString()**: permite devolver el estado de una persona como una cadena.

Desarrollo:

```
public class Persona {

   private String nombre;

   private int edad;

   private char sexo;

   private float altura;

   private float peso;

      private final char SEXO = 'M'; // constante SEXO, notar

   //que se utilizó la palabra reservada 'final' para hacer

   //que dicha variable ya no pueda cambiar su valor.
```

```
    //Notar que el identificador de la variable se declaró en Mayusculas.
    //Constructor por defecto.
    public Persona() {
       nombre = "S/N";
       edad = 0;
       sexo = SEXO;
       altura = peso = 0;
    }
    //sobrecargar con 3 parámetros:
    public Persona(String nombre, int edad, char sexo) {
       this.nombre = nombre;
       this.edad = edad;
       this.sexo = comprobarSexo(sexo);
       altura = peso = 0;
    }
    //getters y setters:
    public String getNombre() {
       return nombre;
    }
  public void setNombre(String nombre) {
       this.nombre = nombre;
    }
    public int getEdad() {
```

```
    return edad;
  }
  public void setEdad(int edad) {
    this.edad = edad;
  }
  public char getSexo() {
    return sexo;
  }
  public void setSexo(char sexo) {
    this.sexo = comprobarSexo(sexo); // mediante el método auxiliar 'comprobarSexo'
    //se valida que el atributo sexo solo puede contener: H o F
  }
  public float getAltura() {
    return altura;
  }
  public void setAltura(float altura) {
    this.altura = altura;
  }
  public float getPeso() {
    return peso;
  }
  public void setPeso(float peso) {
    this.peso = peso;
```

```
    }
    // Métodos propios:
    public int calcularIMC(){
        int aux = -1;
        float imc = peso /(altura * altura);
        if(imc >= 20 && imc <= 25)
            aux = 0;
        else if(imc > 25)
            aux = 1;

        return aux;
    }
    public boolean esMayorEdad(){
        return edad >= 18;
    }
    public String toString() {
        //Devuelve el 'Estado' del objeto como una cadena:
        return "Nombre: " + nombre  + "| Edad=" + edad + " años| Sexo: " + sexo + "|
Altura: " + altura + "| Peso: " + peso;
    }
    //Método privado: solo accesible desde dentro de la clase.
    //Permite comprobar que el atributo sexo solo asumirá los valores H o M.
    private char comprobarSexo (char sexo){
        return (sexo == 'H' || sexo == 'h')?'H':SEXO;
```

```
    }
}
```

- Notar que los atributos se definieron como **private** y los métodos como **public** (Tomaremos como regla general, a menos que se indique lo contrario: los atributos serán privados y los métodos públicos)
- Definiremos tanto métodos de acceso (getters y setters) como atributos tenga la clase. Éstos accesores son muy sencillos: cada uno de los métodos tipo get retorna el valor del atributo con el cual se asocia, y cada método tipo set cambia el valor de ese atributo tomando el nuevo valor como parámetro.
- Al menos un constructor es necesario para inicializar los atributos del objeto
- El método **comprobarSexo()** lo definimos como **privado** para que solo pueda ser llamado desde dentro de la clase.
- Aunque no se indique explícitamente, toda clase en Java hereda o define a partir de **Object**, la cual provee ya definidos una serie de métodos elementales, comunes a todo Objeto.

1.5 Creación de objetos

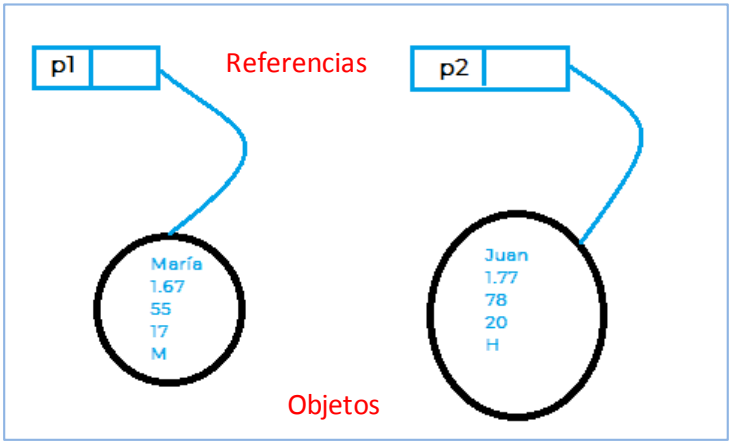
Supongamos ahora que queremos crear dos objetos Persona para empezar a trabajar con ellos. Para ello vamos a utilizar una segunda clase de tipo *Main class*, que por convención llamaremos **Ejecutable** y que mediante su método **main**() crearemos las instancias y les enviaremos los mensajes correspondientes.

Desarrollo:

```
public class Ejecutable {

    public static void main(String[] args) {

        //Crear una persona con los valores por defecto:

        Persona p1 = new Persona();

        //Asignar los valores de los atributos del objeto p1

        p1.setNombre("María");

        p1.setAltura(1.67f);

        p1.setPeso(55);

        p1.setEdad(17);


        //Crear una persona con valores enviados como parámetros:

        Persona p2 = new Persona("Juan", 20, 'H');

        p2.setAltura(1.77f);

        p2.setPeso(78);

        //Mostrar el estado de las personas:

        System.out.println("Persona p1: " + p1.toString());

        System.out.println("Persona p2: " + p2.toString());


        //Es mayor de edad María?

        if (p1.esMayorEdad()) {

            System.out.println("Objeto María es mayor de edad");

        } else {
```

```
            System.out.println("Objeto María NO es mayor de edad");
        }
        //Y el IMC de Juan?
        int imc = p2.calcularIMC();
        if (imc == -1) {
            System.out.println("Objeto Juan está en su peso ideal");
        } else if (imc == 0) {
            System.out.println("Objeto Juan está por debajo de peso ideal");
        } else {
            System.out.println("Objeto Juan está por encima de su peso ideal");
        }
    }
}
```

- Notar que para acceder a los métodos de los objetos Persona utilizamos el punto (.)
- En la línea: p2.calcularIMC() estamos pasando el mensaje **calcularIMC()** al objeto **p2.**
- Gráficamente en memoria:



1.6 Relaciones entre clases

Las clases, al igual que los objetos, no existen aisladamente. Para un dominio de problema específico, las abstracciones suelen estar relacionadas por vías muy diversas formando una estructura de clases más complejo.

En total existen tres tipos básicos de relaciones entre clases:

- Generalización/Especificación: denota un tipo de relación "es un" o mejor aún, "se comporta como". Este tipo de relación lo implementamos mediante un mecanismo en Java llamado **Herencia**.
- Asociación: que denota un tipo de relación "tiene un", indicando alguna dependencia semántica entre clases de otro modo independientes, muchas veces referenciada como hermano-hermano. En programación se implementan mediante referencias. Un atributo de una clase es una referencia a un objeto de otra.

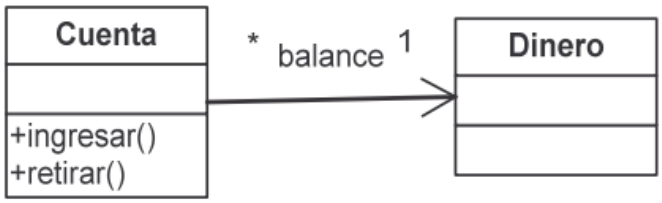
  Dentro de esta categoría podemos encontrar relaciones específicas que denotan **todo-parte**, como son el caso de la **Agregación** o la **Composición**.
- Dependencia: que denota una relación de uso.

Para poder representar gráficamente las relaciones entre clases utilizaremos la notación definida por **UML** (Lenguaje unificado de Modelado), que se trata de un estándar que se ha adoptado a nivel internacional por numerosos organismos y empresas para crear esquemas, diagramas y documentación relativa a los desarrollos de software (programas informáticos).

1.6.1 Asociación

Es una relación estructural que describe una conexión entre los objetos.

Aunque las asociaciones suelen ser bidireccionales (se pueden recorrer en ambos sentidos), en ocasiones es deseable hacerlas unidireccionales (restringir su navegación en un único sentido).



En este caso la Clase cuenta tendrá como atributo un objeto de la clase Dinero llamado balance.

Notar que además de calificar la relación con el nombre **balance** es posible indicar la cantidad de instancias que estará vinculadas en la relación. A esto último se lo suele llamar **Multiplicidad** de la asociación**.** En el ejemplo anterior una cuenta tendrá asociado solo un objeto dinero, pero un objeto dinero puede estar asociado con muchas cuentas. Esto último se representa mediante el símbolo asterisco (*)

1.6.2 Dependencia

Relación (más débil que una asociación) que muestra la relación entre un cliente y el proveedor de un servicio usado por el cliente.

- Cliente es el objeto que solicita un servicio.
- Servidor es el objeto que provee el servicio solicitado.

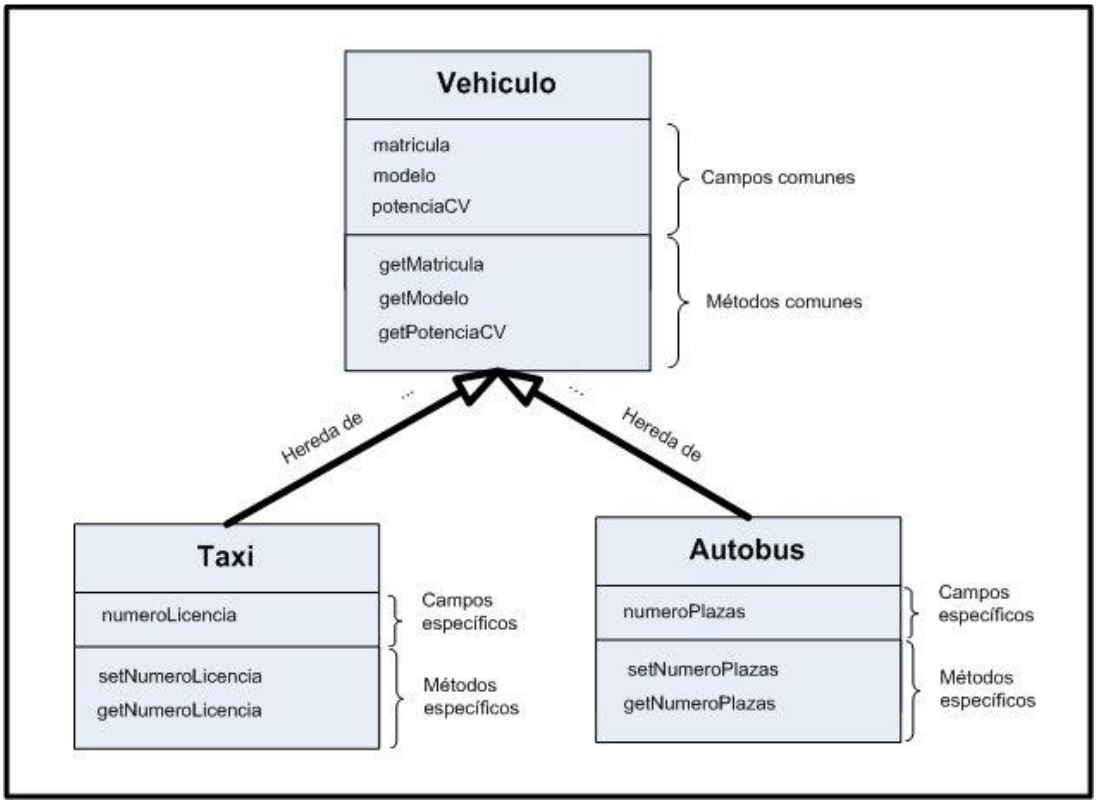
Gráficamente, la dependencia se muestra como una línea discontinua con una punta de flecha que apunta del cliente al proveedor. Por ejemplo supongamos que estamos modelando una ecuación de segundo grado y que necesitamos que un objeto ecuación puede resolver el valor de sus raíces a partir de tener como atributos los coeficientes a, b y c.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Para resolver una ecuación de segundo grado hemos de recurrir a los métodos **sqrt() y pow()** de la clase Math. Éstos nos permiten calcular la raíz cuadrada y potencia respectivamente. Es decir para poder desarrollar el método **resolver()** en la clase Ecuación **se usan** los servicios de la clase de utilidad Math. Este tipo de relación transitoria de uso es la que llamamos **dependencia.**

1.6.3 Generalización/Especificación

En muchas situaciones, objetos de distintas clases pueden tener atributos similares y exhibir comportamientos parecidos. Es posible generalizar todas las propiedades y comportamientos comunes en abstracciones más generales y dejar los propios en clases particulares. Gráficamente:



- Notar que la herencia se denota con una flecha que termina en triángulo.

- Programáticamente las clases hijas o subclases extienden de las clases padre o superclases. Sintácticamente este tipo de relación se implementa de manera diferente a las asociaciones, pero esto será abordará en detalle más adelante.