Entornos de Desarrollo
Tema 8. UML (IV)

1º CFGS: Desarrollo de Aplicaciones Web
IES Severo Ochoa - Elche
2011/2012

Entornos de Desarrollo

Tema 8. UML (IV)

por: Javier Martín Juan

Revisión: 5d130ef647ac

Última actualización: 6 de febrero de 2012

Reconocimientos:

- Francisco Aldarias Raya - Plantillas LaTeX

Índice

1. Objetivos

- Realizar diagramas de clases

2. Diagramas de clases

Los diagramas de estados son un tipo de diagramas estructurales. A diferencia de los diagramas de comportamiento, que modelan el comportamiento del sistema a lo largo del tiempo, los diagramas estructurales describen una vista estática del sistema.

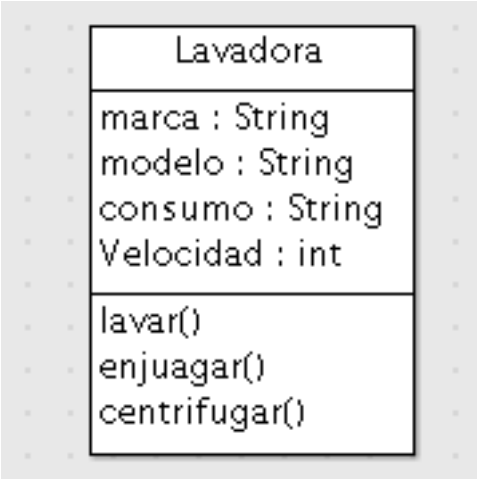
Clases y objetos

Una **clase** es una descripción de un conjunto de objetos con las mismas propiedades (atributos) y el mismo comportamiento (métodos o funciones).

En programación orientada a objetos, una clase es una plantilla que representa un concepto del mundo real y se utiliza para crear objetos. Por ejemplo, si pensamos en las lavadoras, podemos abstraerlas en una clase *Lavadora* junto con sus **atributos** (marca, modelo, consumo, velocidad) y las operaciones, funciones o **métodos** (lavar, enjuagar, centrifugar).

A partir de una clase se pueden crear **objetos**, también llamados **instancias de la clase**. Los objetos son concreciones de una clase: todos los objetos de una clase comparten el mismo comportamiento, pero sus atributos pueden tener valores distintos.

Las clases en UML se representan con una tabla dividida en 3 partes: nombre de la clase, atributos de la misma, y métodos de la misma.



Visibilidad

El acceso a métodos y atributos de objetos de otras clases depende de la **visibilidad**. Hay tres visibilidades distintas. Ordenadas de mayor a menor, son:

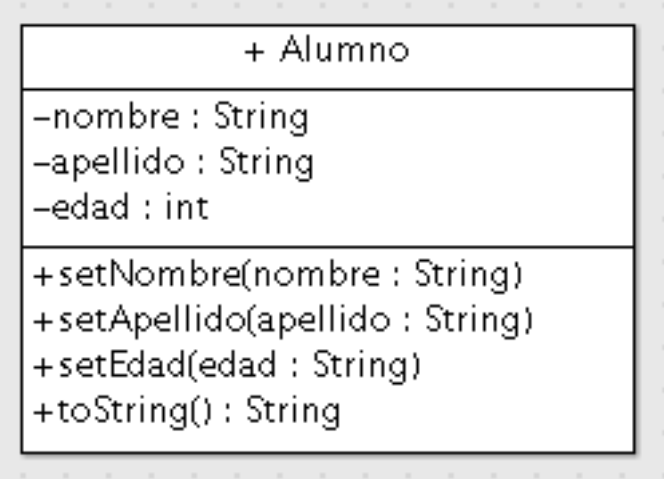
- **Pública (public)**: Se representa con el signo +. Cualquier clase puede acceder a cualquier atributo o método declarado como público.
- **Protegida (protected)**: Se representa con el signo #. Sólo pueden ver el atributo o método protegido los elementos de la propia clase o las clases hijas.
- **Privada (private)**: Se representa con el signo -. Sólo pueden ver el atributo o método privado los elementos de la propia clase.

Incluso en con la menor visibilidad (privada), un objeto siempre puede acceder a cualquier atributo o método de la clase a la que pertenece y usarlo sin restricción.

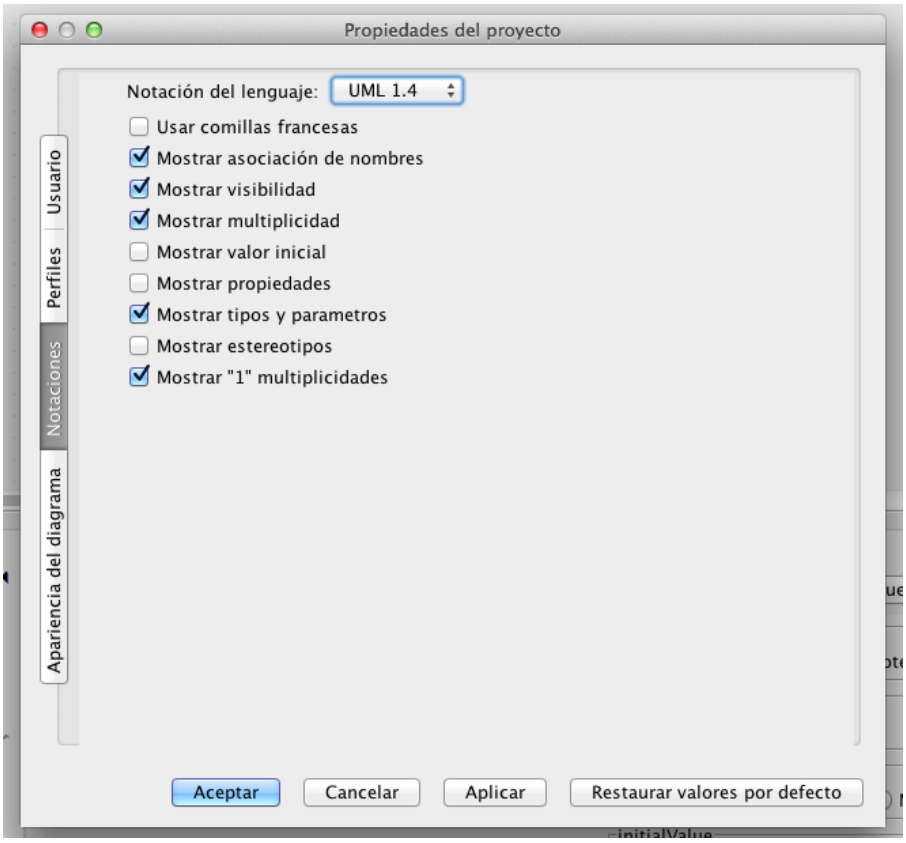
Ejemplo: Alumno

```
class Alumno {
  // atributos
  private String nombre;
  private String apellido;
  private int edad;

  // metodos de acceso
  public void setNombre(String nombre) {
    this.nombre = nombre;
  }
  public void setApellido(String apellido) {
    this.apellido = apellido;
  }
  public void setEdad(int edad) {
    this.edad = edad;
  }
  public String toString () {
    return(nombre + " tiene " + edad + " a~nos.");
  }
}
```

La representación de la clase Alumno en UML sería:



Nota: en ArgoUML hay que ir a las propiedades del proyecto y activar la opción para que se muestre la visibilidad:



El modificador *static*

En Java, el modificador *static* se puede aplicar a atributos (variables o constantes) y a métodos:

Cuando se aplica el modificador *static* a un método significa que dicho método se puede utilizar sin necesidad de instanciar la clase. Un ejemplo de método estático es el punto de entrada a la aplicación: *main()*. En general, cuando un método no necesita acceder a variables del objeto y sólo depende de los parámetros que se le pasen, se puede poner como *static*.

Análogamente, cuando se aplica *static* a un atributo (constante o variable), éste se comparte entre todos los objetos de la misma clase, y además se puede utilizar sin necesidad de instanciar la clase.

A los atributos (variables o constantes) y métodos que llevan el modificador *static* se les llama **atributos estáticos** o **atributos de clase** y **métodos estáticos** o **métodos de clase**.

A los atributos y métodos que no llevan *static* (el comportamiento por defecto), se les llama **atributos de instancia** y **métodos de instancia**.

Ejemplo: atributos y métodos estáticos

Matematicas.java:

```
public class Matematicas {
  // Constante estatica
  public static final double PI = 3.1416;
```

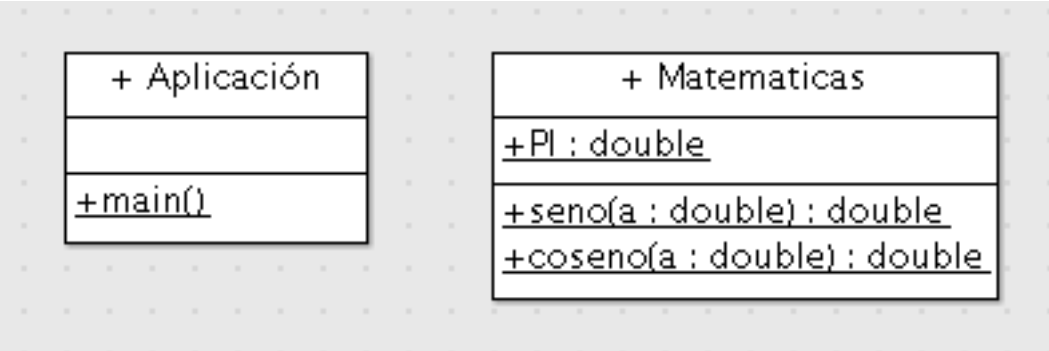
```
  // Metodos estaticos
  public static double seno(double a) {
    ...
  }

  public static double coseno(double a) {
    ...
  }
}
```

Aplicacion.java:

```
public class Aplicacion {
  // main es siempre estatico
  public static void main(String args[]) {
    System.out.println("PI vale " + Matematicas.PI);
    System.out.println("Coseno de 30: " + Matematicas.coseno(30));
  }
}
```

En este ejemplo, los métodos *seno* y *coseno* únicamente necesitan el parámetro *a* para operar, así que se han declarado como estáticos. Si no fueran estáticos, habría que instanciar inútilmente un objeto de la clase *Matematicas* para poder llamarlos. Lo mismo pasaría con la constante *PI*.

Los métodos y atributos estáticos se indican en el diagrama de clase subrayando el elemento en cuestión. La representación en UML del código anterior sería:



3. Relaciones entre clases

En una aplicación mínimamente compleja habrá varias clases cuyos métodos se llamarán unos a otros. Para que un método de la clase A pueda llamar a otro método de la clase B, la clase A debe *poseer* una referencia a un objeto de la clase B. Esta relación de posesión se representa con líneas que unen las distintas clases en el diagrma.

Las relaciones pueden ser varias:

- Dependencia
- Asociación
- Agregación
- Composición

Relación de dependencia

Se usa cuando una clase A utiliza brevemente a la clase B. Por "breve" se entiende normalmente lo que dura una llamada a un método.

Se representa como una línea punteada acabada en una flecha en "V" que va desde la clase A a la clase B.

Se suele utilizar cuando a un método de la clase A se le pasa como parámetro o devuelve un objeto de la clase B.

Ejemplo: dependencia

La clase *Mapa* tiene un método que devuelve las coordenadas a partir una dirección. La relación entre ambas clases dura lo que dura la llamada al método, ya que las coordenadas se devuelven sin almacenarse, por tanto hay una relación débil de **dependencia**.

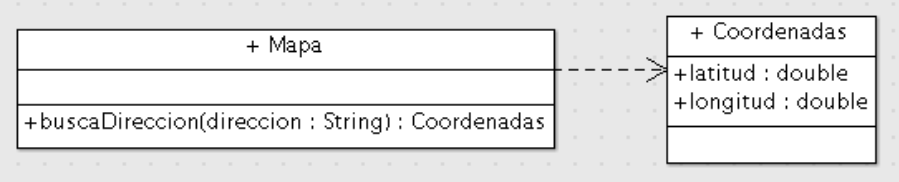
Mapa.java:

```
public class Mapa {
  public Coordenadas buscaDireccion(String d) {
    ...
  }
}
```

Coordenadas.java:

```
public class Coordenadas {
  public double latitud;
  public double longitud;
}
```

El diagrama que captura dicha relación sería:



Relación de asociación

La **asociación** implica una mayor relación que la simple **dependencia**: la relación es persistente y normalmente dura más que una llamada a un método.

Se representa como una línea continua, opcionalmente acabada en una flecha en "V" (según la navegabilidad).

Navegabilidad

Si la flecha apunta de la *ClaseA* a la *ClaseB*, se lee como "*ClaseA* tiene una *ClaseB*" y se dice que la asociación o **navegabilidad** es **unidireccional**. En Java, lo más habitual es que haya un atributo en la clase A que hace referencia a un objeto de la clase B.

Si no dibujamos la flecha, se lee "*ClaseA* tiene una *ClaseB* y *ClaseB* tiene una *ClaseA*", y se dice que la asociación o **navegabilidad** es **bidireccional**. En Java, ambas clases tendrían atributos que se hacen referencia recíprocamente.

Ejemplo: asociación

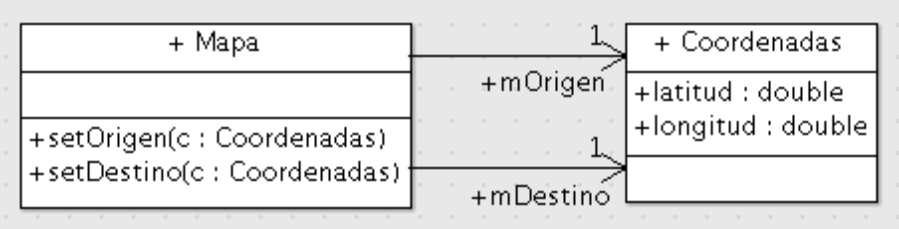
Queremos que nuestro mapa pinte la ruta desde un origen a un destino, ambos definidos por sus coordenadas (latitud, longitud). En este ejemplo, la clase *Mapa* **tiene** dos objetos de tipo *Coordenadas* que se almacenan como variables para poder usarlas más tarde en el cálculo de la ruta más óptima. En este caso hay una relación duradera, por lo que no es una **dependencia**, sino una **asociación**.

La asociación es unidireccional porque *Mapa* necesita conocer las *Coordenadas* de origen y destino, pero las coordenadas no necesitan saber a qué mapa pertenecen.

Mapa.java:

```
public class Mapa {
  public Coordenadas mOrigen;
  public Coordenadas mDestino;

  public void setOrigen(Coordenadas origen) {
    mOrigen = origen;
  }

  public void setDestino(Coordenadas destino) {
    mDestino = destino;
  }
}
```

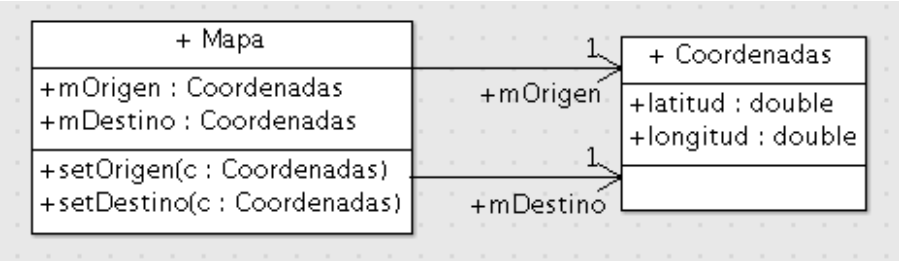
El diagrama UML que captura la relación es:



Los "1" indican la multiplicidad, que veremos más adelante. En el ejemplo, la clase *Mapa* tendrá un solo origen y un solo destino.

Las coordenadas de origen y destino se guardarán en dos variables: *mOrigen* y *mDestino*. Los nombres de estas variables aparecen rotulados en las flechas, **en el lado contrario a la clase que las contiene**.

Importante: un error común cuando representamos asociaciones es volver a incluir los atributos dentro de la clase. Lo siguiente sería **incorrecto**:



Relaciones de agregación y composición

Según la referencia de UML, la **agregación** y la **composición** son "asociaciones que representan una relación parte-todo". Se puede leer como "*ClaseB* es un componente de *ClaseA*".

Las diferencias entre ellas son:

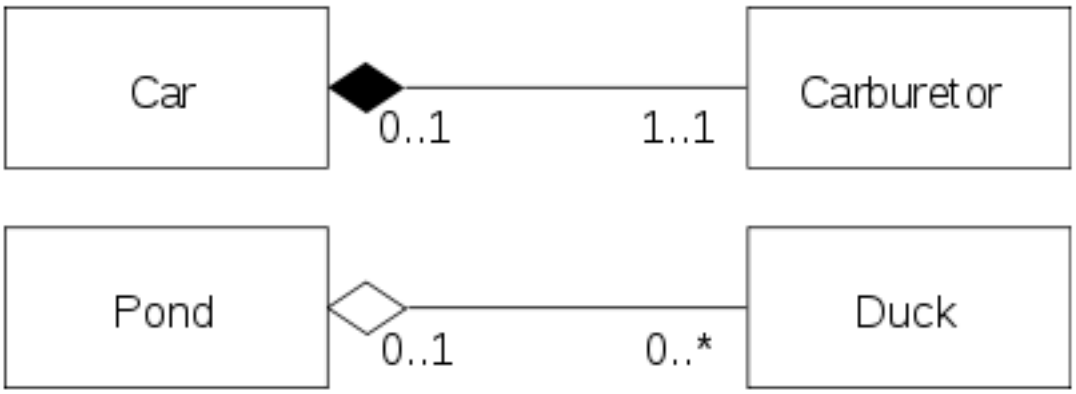
En la **agregación**, los objetos agregados pueden seguir existiendo independientemente del objeto que los agrega. UML no precisa con demasiado detalle la semántica de la agregación así que a veces es difícil distinguirla de una **asociación**. Ante la duda, usaremos la asociación.

En la **composición**, por el contrario, la vida de los objetos compuestos está íntimamente ligada a la del objeto que los compone, de manera que si éste se destruye, los demás no podrían desempeñar su función.

La agregación se representa uniendo las dos clases (todo / parte) con una línea continua y poniendo un rombo hueco en la clase "todo".

La composición es igual, pero con el rombo relleno.

Ejemplos: agregación y composición



La primera relación indica que el Carburador es un **componente** del Coche. La relación de composición sugiere que la vida del carburador está ligada a la del coche, de modo que uno sin el otro no podrían desempeñar su función.

En la segunda relación un estanque **agrega** a varios patos, así que tanto uno como los otros pueden existir independientemente.

Las implementaciones en Java de la agregación y la composición son idénticas a la asociación:

Coche.java:

```
public class Coche {
  private Carburador carburador;
}
```

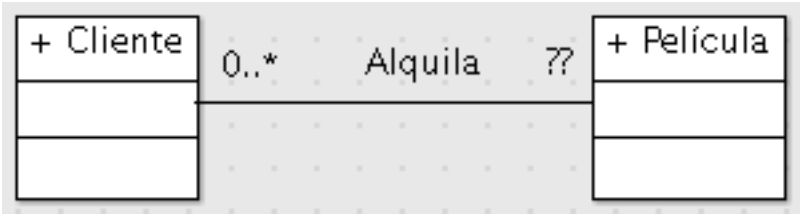
Estanque.java:

```
public class Estanque {
  private List<Pato> patos;
}
```

En el caso del estanque utilizamos un objeto List<Pato> porque la multiplicidad del diagrama indica que un estanque puede tener 0 o más patos.

Roles y multiplicidad

Las relaciones anteriores se pueden rotular indicando el rol que tienen las clases en la relación. Cuando la relación es de posesión (se puede leer como "A tiene B") se puede omitir para no abarrotar el diagrama. En el siguiente ejemplo, un cliente "alquila" películas:



Además, como hemos visto en algunos ejemplos anteriores, los extremos de la relación se pueden rotular con su **multiplicidad**, que indica cuántos objetos de una clase se pueden relacionar como mínimo y como máximo con objetos de la otra clase.

En el ejemplo anterior, una película se sabe que puede no haber sido alquilada, o puede haberse alquilado muchas veces, por tener 0..*.

Pero veamos las posibilidades que aparecen según la multiplicidad que pongamos en el otro extremo de la relación:

- 1: Una instancia de cliente debe estar relacionada con exactamente una instancia de película, ni más ni menos.
- 1..1: Es otra forma de representar lo anterior.
- 0..*: Un cliente puede no tener ninguna película alquilada, o puede tener varias (sin límite).
- 0..3: Un cliente puede alquilar como mucho 3 películas.
- 2..3: Un cliente debe alquilar como mínimo 2 películas, pero como mucho puede alquilar 3.

Generalización (herencia)

La relación de **generalización** entre dos clases indica que una de las clases (la **subclase**) es una especialización de la otra (la **superclase**). Si ClaseA es la superclase y ClaseB la subclase, la generalización se podría leer como "ClaseB es una ClaseA" o "ClaseB es un tipo de ClaseA".

En Java (y la mayoría de lenguajes orientados a objetos), la generalización se conoce como **herencia**. La herencia se utiliza para abstraer en una superclase los métodos y/o atributos comunes a varias subclases.

A la subclase también se le puede llamar **clase hija** o **clase derivada**.

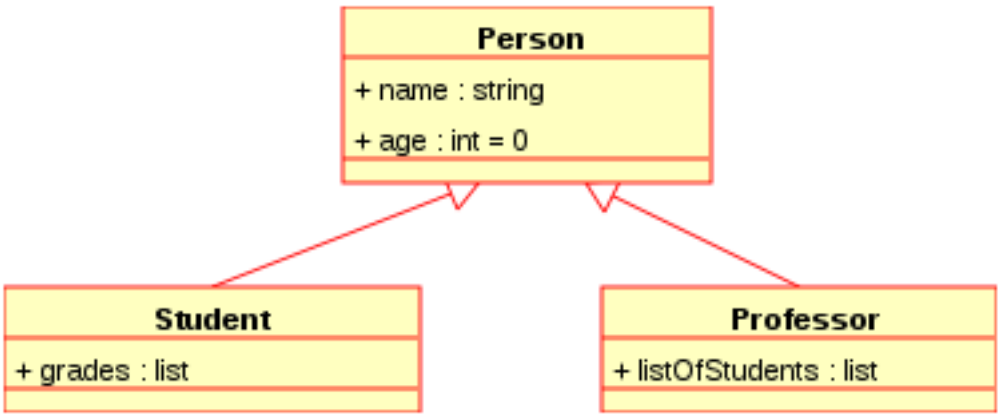
A la superclase también se le puede llamar **clase padre** o **clase base**.

Ejemplo: generalización (herencia)

En Java:

```
public class Person {
  public String name;
  public int age = 0;
}

public class Student extends Person {
  public List grades;
}

public class Professor extends Person {
  public List listOfStudents;
}
```

En UML:



Métodos abstractos

En una clase los métodos pueden ser **abstractos**, lo que significa que para esos métodos no se proporciona ninguna implementación.

Una **clase** abstracta es una clase que tiene al menos un método abstracto.

Al tener uno o más métodos sin implementación, las clases abstractas **no se pueden instanciar**.

Una clase que extiende a una clase abstracta debe implementar los métodos abstractos (escribir el código) o bien volverlos a declarar como abstractos, con lo que ella misma se convierte también en clase abstracta.

Ejemplo: clases abstractas

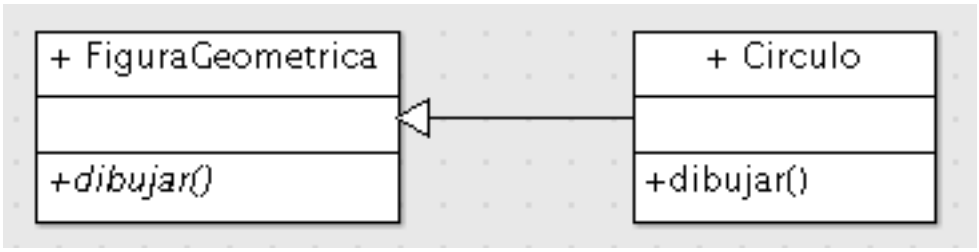
FiguraGeometrica.java:

```
abstract class FiguraGeometrica {
    abstract void dibujar();
}
```

Circulo.java:

```
class Circulo extends FiguraGeometrica {
  void dibujar() {
    // codigo para dibujar Circulo
  }
}
```

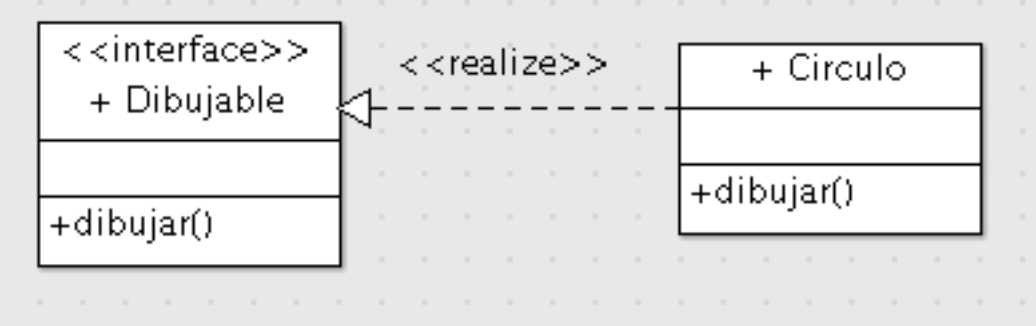
En UML, los métodos abstractos se expresan poniendo el nombre del método en cursiva:



Realización (interfaces)

En Java, un interfaz es similar a una clase abstracta pura, es decir una clase donde todos los métodos son abstractos (no se implementa ninguno). Permite al diseñador de clases establecer la forma de una clase (nombres de los métodos, parámetros y tipos de retorno), pero no bloques de código.

En UML, la implementación de un interfaz equivale a la **realización**, y se representa con una línea discontinua acabada en una flecha triangular hueca:



En Java:

```
interface Dibujable {
  void dibujar();
}

public class Circulo implements Dibujable {
  void dibujar() {
    // Codigo
  }
}
```

Los interfaces sirven para añadir comportamiento a las clases. La ventaja es que para invocar ese comportamiento no necesitamos conocer la clase del objeto. En el siguiente ejemplo tenemos una lista con dos animales que implementan el interfaz *Hablador*. Podemos llamar a sus métodos *hablar()* sin que importe demasiado qué clase de animal sean:

```
import java.util.Iterator;
import java.util.List;
import java.util.ArrayList;

interface Hablador {
  public void habla();
}

class Gato implements Hablador {
  public void habla() {
    System.out.println("Miau");
  }
}
```

```
class Cuco implements Hablador {
  public void habla() {
    System.out.println("Cu cu");
  }
}

public class Test {
  public static List<Hablador> animales = new ArrayList<Hablador>();
  public static void main(String [] args) {
    animales.add(new Gato());
    animales.add(new Cuco());

    Iterator<Hablador> i = animales.iterator();
    while (i.hasNext()) {
      Hablador animal = i.next();
      animal.habla();
    }
  }
}
```

¿Clases abstractas o interfaces?

En Java, ¿qué diferencia hay entre un interface y una clase que tiene todos sus métodos abstractos?

En primer lugar, se diferencian en el modo como se definen y se utilizan:

- Clase abstracta pura:

```
abstract public class FiguraGeometrica {
  abstract void dibujar();
}

public class Circulo extends FiguraGeometrica {
  void dibujar() {
    // Codigo
  }
}

public class Test {
  public static void main(String args[]) {
    FiguraGeometrica fg = new Circulo();
    circulo.dibujar();
  }
}
```

- Interfaz:

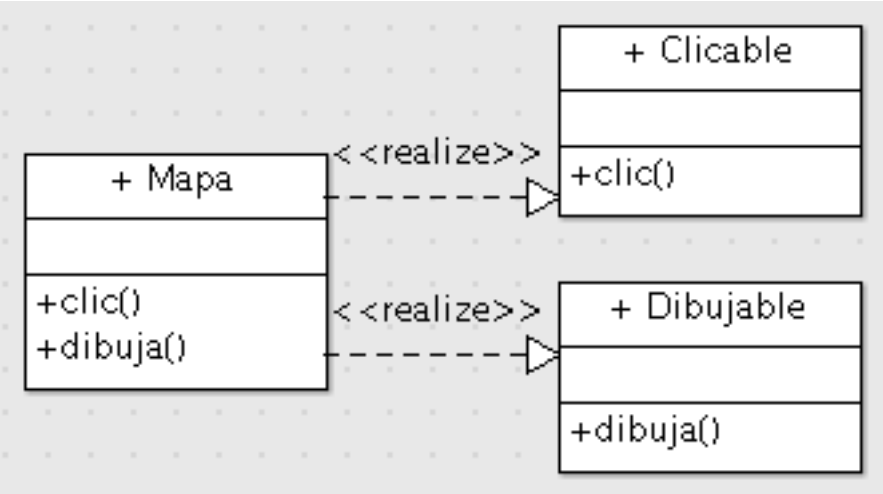
```
interface Dibujable {
  void dibujar();
}

public class Circulo implements Dibujable {
  void dibujar() {
    // Codigo
  }
}

public class Test {
  public static void main(String args[]) {
    Dibujable d = new Circulo();
    circulo.dibujar();
  }
}
```

Un interfaz puede también contener variables, pero siempre *static* y *final*. Por el contrario, una clase abstracta pura sí que puede tener atributos de instancia.

¿Herencia o interfaces?

¿Qué diferencia hay entre la herencia y los interfaces?

La diferencia más importante es que una clase puede implementar múltiples interfaces, pero sólo puede heredar de (como máximo) una clase padre. Por ejemplo, la clase Mapa implementa los interfaces *Dibujable* y *Clicable*. El primero sirve para que el mapa se dibuje en la pantalla, y el segundo para atender los movimientos del ratón:



4. Ejercicios propuestos

Ejercicio 8.1

Convertir el siguiente código a diagrama de clases:

```
public class Circulo {
  protected double x,y; // coordenadas del centro
  protected double r;   // radio del circulo

  /**
   * Crea un circulo a partir de su origen su radio.
   * @param x La coordenada x del centro del circulo.
   * @param y La coordenada y del centro del circulo.
   * @param r El radio del circulo. Debe ser mayor o igual a 0.
   */
  Circulo(double x, double y, double r) {
    this.x=x;
    this.y = y;
    this.r = r;
  }

  /**
   * Calculo del area de este circulo.
   * @return El area (mayor o igual que 0) del circulo.
   */
  public double area() {
    return Math.PI * r * r;
  }

  /**
   * Indica si un punto esta dentro del circulo.
   * @param px componente x del punto
   * @param py componente y del punto
   * @return true si el punto esta dentro del circulo o false en otro caso.
   */
  public boolean contiene(double px, double py) {
    /* Calculamos la distancia de (px,py) al centro del circulo (x,y),
       que se obtiene como raiz cuadrada de (px-x)^2+(py-y)^2 */
    double d = Math.sqrt((px-x)*(px-x)+(py-y)*(py-y));

    // el circulo contiene el punto si d es menor o igual al radio
    return d <= r;
  }
}
```

Ejercicio 8.2

Diseñar el diagrama de clases para modelar una aplicación que realice un inventario informático en una empresa. Cada departamento puede tener varios ordenadores. Cada ordenador está compuesto por una serie de piezas:

- Una placa base, de la cual queremos conocer el modelo.
- De 1 a 8 módulos de RAM, de los que queremos conocer su capacidad y tipo.
- Un procesador, del que queremos saber el modelo y la velocidad.
- Un lector DVD (opcional).
- Al menos un disco duro, del que queremos saber su capacidad.

Ejercicio 8.3

Queremos obtener un sistema de información que almacene datos de:

- Vehículos: dueño, puertas, edad.
- Coches: Las propiedades de vehículo, y además, si es descapotable y con las funciones de subir y bajar capota.
- Camioneta: Las de vehículo, y además tara, carga y con las función de cargar kilos.

Ejercicio 8.4

En una empresa queremos guardar información de sus empleados y de los clientes con las siguientes características:

- De los empleados queremos guardar su sueldo bruto. También queremos calcular el sueldo neto a partir del bruto.
- De los clientes queremos guardar su teléfono.
- Los clientes y los empleados tienen las siguientes características comunes: la edad y el nombre.
- Los directivos son un tipo de empleado. De ellos queremos conocer su categoría.
- Queremos una función que permita imprimir por pantalla todos los datos de cada persona.

Ejercicio 8.5

Traducir el siguiente código en Java a diagrama de clases:

```
interface I1 {
  abstract void test(int i);
}
interface I2 {
  abstract void test(String s);
}
```

```
public class MultInterfaces implements I1, I2 {
  public void test(int i) {
    System.out.println("In MultInterfaces.I1.test");
  }
  public void test(String s) {
    System.out.println("In MultInterfaces.I2.test");
  }
  public static void main(String[] a) {
    MultInterfaces t = new MultInterfaces();
    t.test(42);
    t.test("Hello");
  }
}
```

5. Bibliografía y documentación adicional

- UML Reference Manual, 2nd Edition
  Booch, Jacobson, Rumbaugh, Ed. Addison-Wesley
- UML for Java Programmers
  Robert C. Martin, Ed. Prentice Hall
- Learning UML 2.0
  Miles, Hamilton, Ed. O'Reilly
- UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)
  Martin Fowler, Ed. Addison-Wesley
- Aprendiendo UML en 24 horas
  Joseph Schmuller, Ed. Prentice Hall
- UML 2.0 - Pocket Reference
  Dan Pilone, Ed. O'Reilly
- Análisis y Diseño Estructurado y Orientado a Objetos de Sistemas Informáticos
  Amescua et al, Ed. McGraw-Hill
- Diseño Orientado a Objetos con UML
  Raúl Alarcón, Grupo Eidos