Programación con Java y Eclipse

Clases y Herencia

Indice

1 Introducción a la Orientación a Objetos

En este apartado se intenta explicar el enfoque de la programación orientada a objetos, proporcionando unas nociones básicas para aquellas personas que no se hayan encontrado con ella todavía, puesto que es algo fundamental para poder comprender la programación Java.

Los programas son simulaciones de modelos conceptuales de la realidad. Estos modelos son, muy a menudo, complejos por necesidad y es tarea del programador reducir esa complejidad a algo más comprensible y manejable: reducirlo a objetos.

En la Programación Orientada a Objetos (POO) casi todo puede ser considerado un objeto.

Los objetos representan cosas, simples o complejas, reales o imaginarias. Una antena parabólica es un objeto complejo y real. Una frase, un número complejo, una receta y una cuenta bancaria también son representaciones de cosas intangibles. Todas son objetos.

Algunas cosas no son objetos, sino **atributos**, valores o características de objetos. Es decir, no todas las cosas son objetos, ni son consideradas normalmente como objetos. Algunas de ellas son simplemente atributos de los objetos como el **color, el tamaño y la velocidad**. Los atributos reflejan el estado de un objeto. Normalmente no tiene sentido considerar la velocidad como un objeto.

Los objetos **encapsulan** sus operaciones y su estado, son islas de estado y comportamiento.

- El comportamiento del objeto está definido por las operaciones
- El estado está definido por los datos o atributos del objeto.

La **encapsulación** también abarca a la ocultación de la información:

- Algunas partes son visibles (el interfaz público)
- Otras partes son ocultas (o privadas)

En la Programación Orientada a Objetos no sólo se modelan los objetos, sino también sus interrelaciones. Para realizar su tarea, un objeto puede delegar trabajos en otro. Este otro puede ser parte integrante de él o ser cualquier otro objeto del sistema.

La gran ventaja de la Programación Orientada a Objetos es que a través del encapsulamiento en objetos se fomenta la ocultación de los detalles de implementación del objeto, de modo que el objeto internamente puede cambiar (evolucionar), y el interfaz público proporcionado ser compatible con el original. Entonces los programas que utilizaban el objeto pueden seguir funcionando sin alteración alguna.

Esto es extremadamente útil al modificar código y mantener las aplicaciones. También hace los objetos extremadamente reutilizables, lo cual además de ser beneficioso desde

el punto de vista de la programación lo es también desde el punto de vista económico.

2 Clases

Las clases detallan como se definen los objetos y de que manera se comportan. Al concretar una clase con unos valores determinados obtenemos un objeto. **Todo en Java es una clase**, describe como funciona una clase o forma parte de una clase.

Para crear las clases se usan variables que contendrán datos, y también se usan métodos, que indican como se tratarán dichos datos.

Si bien existen clases predefinidas en Java que podemos usar en nuestras aplicaciones, podemos crear nuevas clases. Al definir una nueva clase estamos especificando como se representará el objeto y de que manera se comportará. La clase contiene:

- **Campos o variables**. Almacenan datos.
- **Métodos**. Son funciones, que modifican objetos y variables. Son métodos las operaciones matemáticas, por ejemplo, o la acción de imprimir un dato en pantalla.

Ejemplo de definición de una clase:

```
public class MiClase {
      int i;    // Atributo miembro de la clase MiClase

      public MiClase () {// Método constructor de la clase MiClase
      i= 10;
      }

      public int Suma_a_i (int j) { // Método Suma_a_i
            int suma; // Variable local del método Suma_a_i
            suma = i + j;
            return suma;
      }

}
```

2.1. Métodosconstructores

Son métodos que se utilizan para definir el **estado inicial de un objeto cuando son creados. Se define dentro de la clase del objeto en cuestión**, junto con los datos que maneja y los métodos que manejan dichos datos.

Una misma clase puede tener varios constructores, llamándose constructores sobrecargados.

El constructor lleva por nombre el de la clase que define, diferenciándose los constructores sobrecargados entre sí simplemente por los parámetros que le pasamos. Tampoco se le especifica un tipo de dato de retorno.

```
Class Persona {
         String name;
         Persona(String name) {
           this.name = name;
         }
      }
```

Para crear un ejemplar de una clase, llamamos al constructor con la palabra clave **new**.

```
Persona p1 = new Persona("Carlos Torrero");
```

Se pueden usar varios constructores en una clase. Por ejemplo, para crear un objeto Persona del que desconocemos su nombre, podemos indicar un segundo constructor que no tome argumentos y proporcione un valor predeterminado para el nombre:

```
Class Persona {
         String name;
         Persona() {
           this.name= "Jane Doe";
         }
         Persona(String name) {
           this.name = name;
         }
      }
```

No suele ser necesario usar this. Pero **cuando una variable de instancia y otra local comparten un mismo nombre**, hay que diferenciar ambos poniendo un prefijo al acceso a la variable con el **uso de this**.

Al crear una **subclase**, su constructor llama de manera automática a la versión sin argumentos de la clase padre, a menos que se especifique lo contrario.

```
Class Soldier extends Persona {
         String rank;
         String serialNumber;
         Soldier(String name, String rank, String serialNumber)
         {
         this.rank = rank;
         this.serialNumber = serialNumber;
         }
      }
```

Según este ejemplo, y teniendo en cuenta el anterior, **todos los soldados tendrán el nombre "Jane Doe"**, pues como el constructor Soldier no indica lo contrario se llama al constructor de la clase padre para inicializar el estado de Persona.

Si deseamos llamar al constructor de la superclase usamos la llamada **super()** en la primera línea del constructor.

```
Class Soldier extends Person {
         String rank;
         String serialNumber;
         Soldier(String name, String rank, String serialNumber) {
               super(name);
               this.rank=rank;
               this.serialNumber=serialNumber;
         }
      }
```

Cualquier cosa que pasemos a la llamada al constructor de Soldier se pasará al constructor de Person.

El uso de **super está reservado sólo para la primera línea de un constructor.** Si no está presente, se asume una llamada vacía a super.

2.2. Tipos de clase

En el ejemplo anterior (public class MiClase) el modificador era public. Estos modificadores de clase pueden ser:

public

Son accesibles desde otras clases, ya sea de manera directa o por herencia. Para que puedan acceder a las clases que están en paquetes externos antes hay que importar estos.

abstract

Estas clases poseen, al menos, un método abstracto. Estas **clases abstractas no se instancian, sino que se utilizan como base para la herencia.**

final

Es la clase que termina una cadena de herencia. Por tanto, **es lo contrario de las clases abstractas. No pueden heredarse.**

synchronizable

Indica que todos los métodos de la clase son sincronizados, lo que significa que no puede accederse a ellos al mismo tiempo desde distintas tareas. Gracias a esto podemos modificar las mismas variables desde distintas tareas sin temor a que se sobrescriban.

sin modificador

La clase puede ser usada e instanciada por clases dentro del package donde se define.

2.3. Modificadores de acceso

Los modificadores de acceso permiten al diseñador de una clase determinar quien accede a los datos y métodos miembros de una clase.

Los modificadores de acceso preceden a la declaración de un elemento de la clase (ya sea dato o método), de la siguiente forma:

[modificadores]	tipo_variablenombre;
[modificadores]	tipo_devuelto nombre_Metodo (lista_Argumentos);

Existen los siguientes **modificadores de acceso**:

public-Todo el mundo puede acceder al elemento. Si es un dato miembro, todo el mundo puede ver el elemento, es decir, usarlo y asignarlo. Si es un método todo el mundo puede invocarlo.

protected– Se puede acceder desde la propia clase y subclases. También desde cualquier clase del package donde se define la clase.

sin modificador (o package)- Se puede acceder al elemento desde cualquier clase del package donde se define la clase.

private- Sólo se puede acceder al elemento desde métodos de la clase, o sólo puede invocarse el método desde otro método de la clase.

En el ejemplo anterior se definía una variable miembro en **MiClase** a través de “int i;”, en este caso no se ha definido modificador

	Class	**Package**	**Subclass**	**World**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Visibilidad entre los elementos de la clase:

	private	**sin modificador**	**protected**	**public**
Desde la misma clase	Y	Y	Y	Y
Desde una subclase de paquete	N	Y	Y	Y
Desde una no-subclase de paquete	N	Y	Y	Y
Desde una no-subclase de diferente paquete	N	N	N	Y

Guia para usar el control de acceso:

- Usar private para métodos y variables que solamente se utilicen dentro de la clase y que deberían estar ocultas para todo elresto.
- Usar public para métodos, constantes y otras variables importantes que deban ser visibles para todo el mundo.
- Usar protected si se quiere que las clases del mismo paquete y subclases puedan tener acceso a estas variables o métodos.
- No usar nada, dejar la visibilidad por defecto (default, package) para métodos y variables que deban estar ocultas fuera del paquete, pero que deban estar disponibles al acceso desde dentro del mismo paquete. Utilizar protected en su lugar si se quiere que esos componentes sean visibles fuera del paquete.

2.4. Ejemplo de uso

Crearemos un nuevo proyecto llamado tema5.
Dentro crearemos una clase llamada **PruebaModificadoresAcceso** y copiaremos el siguiente código:

```
class Profesor
{
	String nombre;
	int edad;
	protected int añosAntiguedad;
}

class Catedratico extends Profesor//CREA UNA SUBCLASE
{
	int añosCatedratico;
	public double obtenerSalario()
	{
		return(925 + añosAntiguedad * 33.25 + 47.80 * añosCatedratico);
	}
	public void imprimirSalario()
	{
		System.out.println("El salario de "+nombre+" de "+edad+" años
		es"+obtenerSalario()+" €");
	}
}

public class PruebaModificadoresAcceso {
	public static void main(String[]args {
		Catedratico catedratico = new Catedratico();
		catedratico.nombre="Paco";
		catedratico.edad=47;
		catedratico.añosAntiguedad=10;
		catedratico.añosCatedratico=5;
		catedratico.imprimirSalario();
	}
}
```

Ahora prueba lo siguiente:

1- Cambia `String nombre;` a `private String nombre;`

Observa que nos indica errores de que el campo ya no puede ser utilizado en subclases porque no es visible.

2- Cambia ahora `private String nombre;` a `protected String nombre;`

Observa que ahora no da error, ya que ese campo ahora es accesible por todas las subclases (Catedratico en nuestro caso) y las clases del package

3 Objetos

Un objeto es una instanciación de una clase, y tiene:

- **Un funcionamiento**. Determinado por sus métodos. Llamando a uno de estos métodos activamos el funcionamiento del objeto, realizándose una acción o modificando el estado del propio objeto (o las doscosas).

- **Un estado**. Determinado por sus variables miembro (que pueden ser de instancia o de clase).

Para la creación se completan tres fases:

Declaración. Damos nombre al objeto.
Instanciación. Le asignamos memoria.
Inicialización. Damos un valor inicial a las variables de instancia (opcional).

Por ejemplo:

```
String a;                // Declaración de la variable
a = new String("abc");   // Instanciación e inicialicación
a="xyz";
```

En principio, una aplicación puede tener tantos objetos como queramos, si bien la memoria del ordenador limita el número de objetos que puede haber.
Esto se debe a que los objetos de la aplicación se almacenan en memoria.

3.1. Destrucción deobjetos

A diferencia de lo que ocurre en C++, **en Java no hay destructores de objetos**, ya que hay un reciclador de memoria o garbage collector en cargado de ello.

Cada objeto en Java, sin embargo, hereda de la clase Object el método finalize(), que suele ser usado para reciclar memoria, aunque nada nos asegura que este método sea invocado.

4 Agrupaciones de clases: Paquetes

4.1. Palabra clave Package

Las convenciones del lenguaje Java recomiendan el uso de paquetes (**packages**) para la agrupación y organización de las clases. Estas divisiones se conocen como espacios de nombres (**namespaces**).

No hay una clara definición oficial, pero sí una convención:

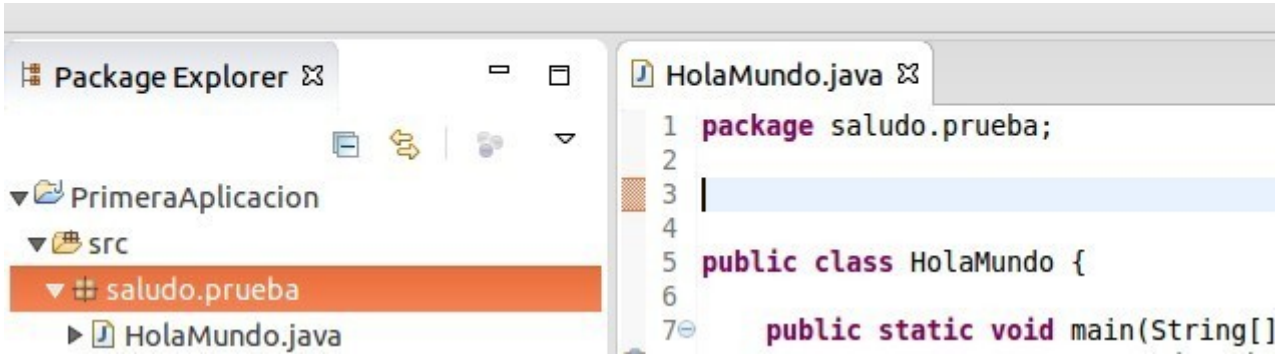
- Los paquetes deben ser identificadores en minúscula, separados por punto
- No utilizar símbolos más allá del "." para separar palabras. ej: juegobola,worldlogic

Debido al gran número de clases predefinidas que tiene Java, estas están agrupadas en paquetes.

Al crear una aplicación, agrupamos las clases que se usan en ella en un paquete particular en el cual se agrupan las clases que están relacionadas, para que estén disponibles para el compilador.

Dicho paquete funciona a modo de directorio, dentro del cual se encuentran los archivos de las distintas clases utilizadas en la aplicación.

Por cada palabra entre puntos: un subdirectorio. Aunque veamos en el explorador de paquetes algo como:



Donde pareciera que existe un directorio "saludo.prueba", en donde se encuentra el archivo "HolaMundo.java", la realidad es que "HolaMundo .java" está en:

/Escritorio/cursoJava/PrimeraAplicacion/src/saludo/prueba/HolaMundo.java

Los paquetes en los que organizamos nuestras clases afectan la visibilidad de métodos, campos, etc. según como establezcamos sus modificadores de acceso.

Mediante la palabra package solo indicamos al compilador donde encontrar nuestras clases particulares.

4.2. Palabra clave Import

Sin embargo, se le indican las clases java predefinidas que vamos a utilizar mediante dos posibles métodos:

- Escribiendo el nombre completo de la clase y el paquete en cuestión (**nombre totalmente cualificado**). Ej:java.applet.Applet

- Usando la sentencia **import** al principio de nuestra clase. Una opción muy conveniente cuando usamos una clase o interfaz de otro paquete, es el uso de la palabra clave import. Si usamos import, no es necesario escribir el nombre completo de la clase con el paquete incluido. Es suficiente con el nombre de la clase. Ej: import java.applet.*;

Por ejemplo si al comienzo de nuestro fichero .java añadimos las siguientes líneas de importación de paquetes:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

Estaremos en disposición de usar en nuestro código las clases definidas en en las librerías AWT y SWING.

Es interesante saber, que las clases del paquete java.lang, son importadas automáticamente. Por ejemplo, si usamos la clase String, no es necesario importarla.

Podemos ver en el ejemplo, como usamos **java.lang.String**, para declarar s1, y tan sólo **String** para declarar s2 y ambos funcionan.

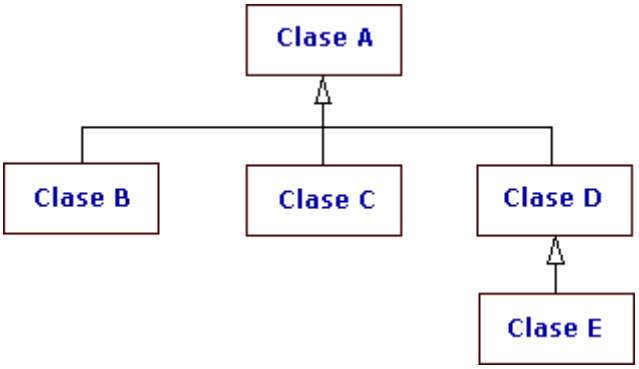
```
public class Ejemplo {
	public static void main(String[] args) {

		java.lang.String s1;
		String s2;

		s1="Hola";
		s2="Adios";
	}
}
```

5 Herencia de clases

Mediante la palabra clave **extends** podemos heredar las características de una clase ya existente. De esta manera nos podemos ahorrar escribir código cuando ya existe un objeto similar.

Al desarrollar una clase ("hijo" o subclase) que desciende de otra ("padre" o superclase), dicha clase hereda los campos y métodos de la clase padre.
También podemos sobrescribir métodos de la clase padre para cubrir mejor las necesidades de la clase hijo, así como ocultar campos, etc.
A la hora de sobrescribir los métodos de la clase padre añadiremos la anotación **@Override**, lo cual hace que el compilador compruebe que estamos sobrescribiendo un método existente en la clase padre.

En Java, no existe la herencia múltiple, es decir, que no puede crearse una clase como una subclase de varias subclases diferentes.



5.1. Ejemplo de uso para Herencia de clases

Crearemos una clase llamada VehiculoTest y copiaremos el siguiente código:

```
Class Vehiculo
{
 int numRuedas = 4;
 int numPuertas = 4;
 boolean llevoRadio = true;
 public Vehiculo()
 {
       showinfo();
 }

 public Vehiculo (int numPuertas,int numRuedas)
 {
   this.numPuertas=numPuertas;
   this.numRuedas=numRuedas;

      showinfo();
  }
```

```
 public void showinfo()
 {
       System.out.println("Soy un vehiculo de "+numPuertas+" puertas y
"+numRuedas+" ruedas");
 }
 public void conducir()
  {
     System.out.println("Conduzco con mis "+numRuedas+" ruedas.");
     if (llevoRadio) System.out.println("Voy escuchando la radio");
  }

}

//Subclase que añade un metodo y sobreescrible uno existente
class Coche extends Vehiculo
{
 public Coche()
 {
   System.out.println("Soy un coche");
  }

  @Override
  public void conducir()
  {
    cierraPuertas();
    super.conducir();
  }

  public void cierraPuertas()
  {
    System.out.println("Cierro las puertas.");
   }

}

//Subclase que sobreescribe el valor de un campo y hace una llamada a un
constructor especifico para inicializar otras variables
class Moto extends Vehiculo
{
  public Moto()
  {
      super(0,2);
      llevoRadio=false;
    System.out.println("Soy una moto");
   }
}

//Clase principal para probarlo
public class VehiculoTest
{
  public static void main(String[]args)
  {
    Coche _coche = new Coche();
    _coche.conducir();

    Moto _moto = new Moto();
    _moto.conducir();
  }
}
```

La salida es:

```
Soy un vehiculo de 4 puertas y 4 ruedas
Soy un coche
Cierro las puertas.
Conduzco con mis 4 ruedas.
Voy escuchando la radio
Soy un vehiculo de 0 puertas y 2 ruedas
Soy una moto
Conduzco con mis 2 ruedas.
```

Como puede verse en este ejemplo, podemos:

- cambiar los valores con los que se inicializa la clase padre llamando al constructor adecuado.
- Cambiar los valores de campos del padre directamente sobrescribiendolos desde la clase hijo
- Cuando hemos sobrescrito un método del padre, y necesitamos llamar a la versión del padre de este método, usamos la palabra clave **super** y el operador punto.
- Para evitar la confusión entre la variable de método y la variable miembro usamos la palabra clave **this**

NOTA:
En el ejemplo hemos creado todas las clases (Vehiculo, Coche, Moto y VehiculoTest) dentro del mismo fichero VehiculoTest.java

Esto lo hacemos por comodidad para aprender los objetivos de cada apartado, aunque en el mundo profesional las clases suelen ir separadas cada una en su fichero y package correspondiente.

5.2. La clase Object

Es la **clase raíz del árbol de clases Java**. Todas las clases heredan de esta. Proporciona una serie de métodos básicos que heredan todas las clases descendientes. Algunos de ellos son:

Método equals()

Permite comparar dos objetos, pero no de la misma manera que el operador ==, que solo compara si las dos referencias apuntan al mismo objeto. Equals devuelve true si ambos objetos son del mismo tipo y contienen los mismos datos, y false si no es así.

Método getClass()

Podemos utilizar este método para obtener la clase a la que pertenece un objeto. Devuelve un objeto de tipo Class con información importante del objeto que crea la clase. Este método no puede ser sobrescrito.

Método toString()

Convierte el objeto en cuestión en una cadena.

5.3. Modificadores de Acceso y sobrescritura de métodos:

Java al sobrescribir métodos no nos permite reducir la visibilidad, por ello nos impone las siguientes reglas:

- **Métodos declarados public** en la superclase también deben ser declarados public al sobrescribirlos en las subclases
- **Métodos declarados protected** en la superclase deben ser declarados public o protected al sobrescribirlos en las subclases. Nunca pueden ser private
- **Métodos declarados sin modificador** pueden ser declarados public, protected o sin modificador en las subclases.
- **Métodos declarados private** no pueden ser sobrescritos nunca, ni tan siquiera son heredados.

5.4. Ejemplo de uso para sobrescritura de métodos:

Crearemos una clase llamada **PruebaHerenciaMetodos** y copiaremos el siguiente código:

```
class AudioPlayer {
         protected void openSpeaker() {

         }

         void showTV() {

         }
      }

class StreamingAudioPlayer extends AudioPlayer{
         @Override
         public void openSpeaker() {
            // implementation details
         }
         @Override
         public void showTV() {

         }

      }
```

Observad como no muestra error al cambiar la visibilidad de los métodos sobrescritos. Probad a cambiar `public void showTV()` por `private void showTV()` y observad como muestra un error.

6 Polimorfismo

Antes de abordar el polimorfismo, ahora estamos en condiciones de abordar el Casting de tipo objeto.

6.1. Casting de tipo objeto

Es importante recalcar que al forzar un objeto referencia el tipo de dato no cambia, sólo la manera en que el compilador va a tratar a dicho objeto.

Cuando los métodos son declarados para ser genéricos, devolviendo o aceptando un tipo Object, en ocasiones es necesario acceder a la variable por su tipo específico. En este caso es cuando necesitamos el casting.
Si un parámetro va a tener un tipo específico podemos forzarlo, si así lo deseamos. Veamos un ejemplo:

```
public void myMethod(Object param) {
          Number num=(Number)param;
          doubled = num.doubleValue();
          System.out.println("Value is: " + d);
      }
```

Si no conocemos el tipo específico, podemos usar la palabra clave **instance of**.
Esto permite chequear el tipo y luego forzarlo:

```
public void myMethod(Object param) {
         if(param instanceof Number) {
           Number num = (Number)param;
           double d = num.doubleValue();
           System.out.println("Value is: "+d);
         }else{
           System.out.println("Param no es de tipo Number");
         }
      }
```

Si no usáramos instanceof se lanzaría una ClassException en tiempo de ejecución al intentar el forzado entre tipos incompatibles. Eso ocurriría en este caso:

```
        // Create object
        Object ob = newObject();
        // The cast
        String s = (String) ob;
        // The object s refers to is not really a String
        // so a ClassCastException is thrown
        System.out.println(s.length());
```

Sin embargo, el próximo ejemplo sería correcto, ya que el objeto referenciado por la variable ob es un String:

```
            // Create object
            Object ob="Hello";
            // The cast
            String s= (String)ob;
            // The object s refers to really is a String
            // so no ClassCastException is thrown
            System.out.println(s.length());
```

6.2. Polimorfismo

Polimorfismo es la capacidad de un objeto de adquirir varias formas. El uso más común de polimorfismo en programación orientada a objetos se da cuando se utiliza la referencia de una clase padre, para referirse al objeto de la clase hijo.

En Java, todos los objetos son polimórficos ya que cualquier objeto pasaría un Object.

Es importante saber que la única manera de acceder a un objeto es a través de una variable de referencia. La variable de referencia sólo puede ser de un tipo. Una vez declarado el tipo de la variable de referencia, no se puede cambiar.

La variable de referencia puede ser reasignada a otros objetos, siempre y cuando no haya sido declarada "final".

Una variable de referencia puede hacer referencia a**cualquier objeto o cualquier subtipo de su propio tipo**.

6.3. Ejemplo de uso

Crearemos en nuestra package tema5 una clase llamada PruebaPolimorfismo y copiaremos el siguiente código:

```
class Base {
      public void print() {
            System.out.println("Base");
      }

}

class HijoA extends Base {
      public void print() {
            System.out.println("A");
      }

}
```

```
Class HijoB extends Base {
      public void print() {
            System.out.println("B");
      }

}

public class PruebaPolimorfismo {
      public static void main(String[] args) {
            Base[] bs = new Base[3];//LOS OBJETOS SON DE LA CLASE PADRE
            bs[0] = new Base();
            bs[1] =new HijoA();
            bs[2] =new HijoB();
            PruebaPolimorfismo p = new
            PruebaPolimorfismo();
            p.imprimir(bs);

      }

      private void imprimir(Base[] bs) {
            for(int  i= 0; i<bs.length; i++) {
                  bs[i].print();//EJECUTA EL METODO SOBREESCRITO POR LOS HIJOS
            }

}
```

La salida es:

```
Base
A
B
```

7 Clases abstractas e Interfaces

7.1. Clases abstractas

Una clase abstracta declara la existencia de métodos pero no la implementación de todos sus métodos (o sea, las llaves { } y las sentencias entre ellas).

Una clase abstracta puede contener métodos no-abstractos pero al menos uno de los métodos debe ser declarado abstracto.

Para declarar una clase o un método como abstractos, se utiliza la palabra reservada **abstract**.

```
abstract class Drawing
{
   abstract void miMetodo(int var1, int var2);
   String miOtroMetodo(){
         return null;
    }
}
```

Una clase abstracta no se puede instanciar pero si se puede heredar y las clases hijas serán las encargadas de agregar la funcionalidad a los métodos abstractos. Si no lo hacen así, las clases hijas deben ser también abstractas.

7.2. Interfaces

El concepto de interfaz puede parecer similar al de clase abstracta. De hecho, posee declaración y definición. Sin embargo, mientras que en una clase abstracta podemos definir los métodos que queramos, en una interfaz no podemos definir métodos.

Una interface es una variante de una clase abstracta con la condición de que todos sus métodos deben ser abstractos. Si la interface va a tener atributos, éstos deben llevar las palabras reservadas **public final** y con un valor inicial ya que funcionan como constantes por lo que, por convención, su nombre va en mayúsculas.

```
public interface MiInterfaz {
    public final double PI = 6.14;
    public final int CONST = 75;

    void put(int dato);
    int get();
}
```

Para implementar las interfaces de la clase usaremos implements, y las distintas interfaces separadas por comas.

```
Class Miclase extends MiSuperClase implements MiInterfaz {
          // Cuerpo de la clase
}
```

Al implementar la interfaz, es necesario que coincidan los métodos de la clase con los de los métodos declarados en esa interfaz.

Estas interfaces son útiles cuando queremos declarar métodos que implementarán varias clases.

7.3. Ejemplo de uso

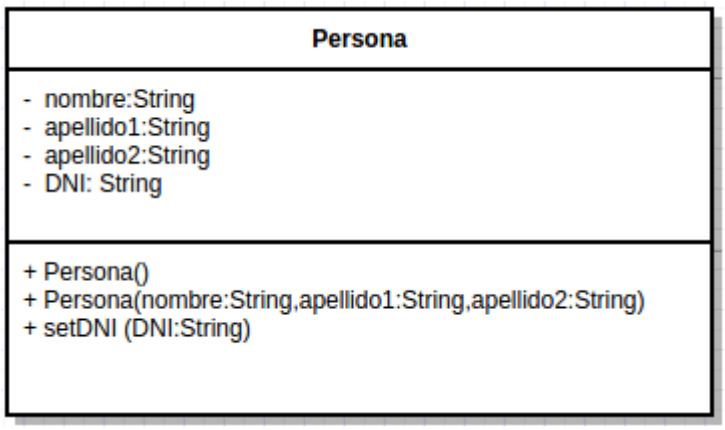
Crearemos en nuestra package tema5 una clase llamada ProfesorApp y copiaremos el siguiente código:

```
interface IProfesor {

    public void identificarse();
}


abstract class ProfesorBase implements IProfesor {
    String nombre;

    public void setNombre(String nombre)
    {
      this.nombre=nombre;
    }
    public void identificarse() {
        System.out.println("Me llamo " + this.nombre);
        masInfo();
    }

    protected abstract void masInfo();

}

class ProcessorFP extends ProfesorBase

    {@Override
    protected voidmasInfo(){
       System.out.println("Soy profesor especialista en FP");
    }

}

public classProfesorApp{
      public static void main(String[] args{
            ProcessorFP _profesor = new ProcessorFP();
            _profesor.setNombre("Paco");
            _profesor.identificarse();
      }
}
```

8 Ejercicios

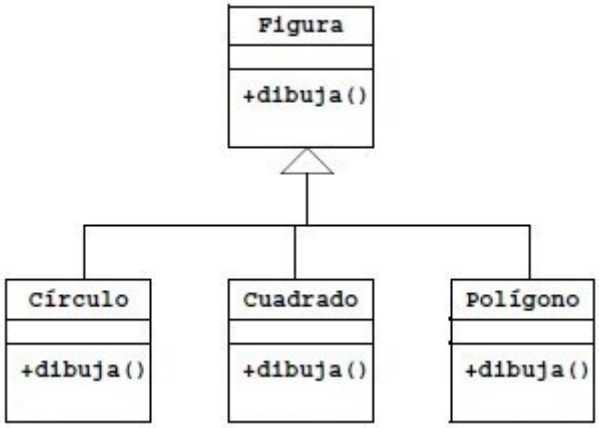
1- Crea una clase Persona

- con los campos privados: nombre, apellido1, apellido2 yDNI
- con dos constructores (uno sin parametros, y otro que inicialice los campos nombre y apellidos)
- con un metodo para establecer elDNI



2- Crea el siguiente esquema, en el que la clase padre Figura tiene un método dibuja que será sobreescrito por sus clases hijas. Por ejemplo la clase Círculo hará:

- dibuja()-->Escribirá por pantalla soy un circulo

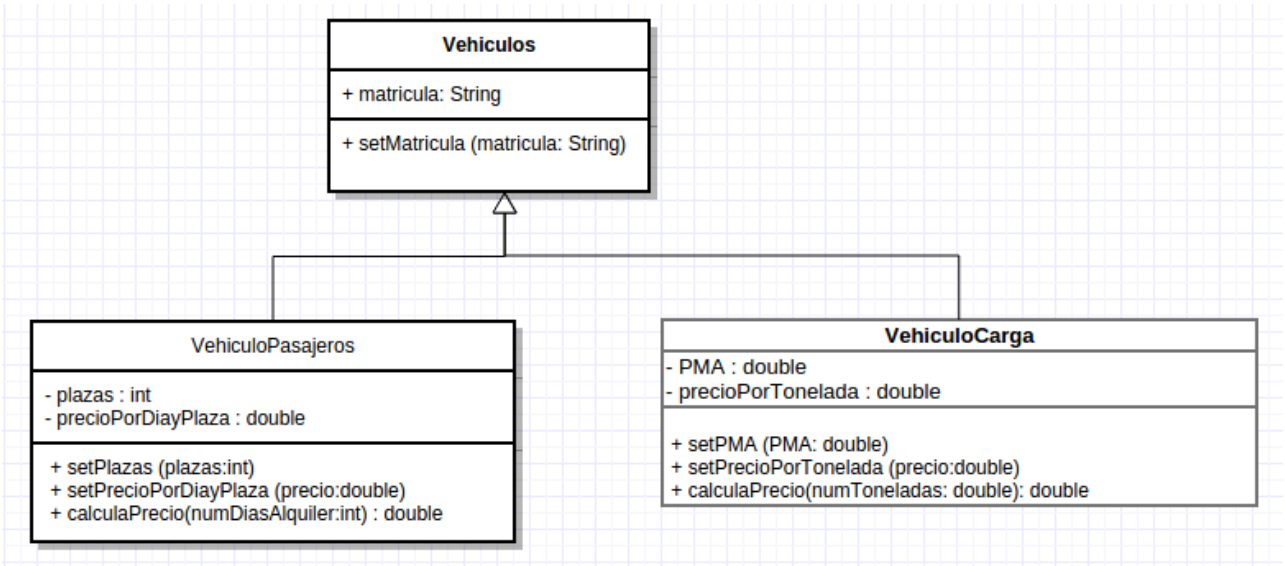


Para ello y por comodidad (aunque si quieres puedes crear clase en un fichero separado) te aconsejamos crear una clase llamada RecorridoFiguras y añade en la parte superior las clases pedidas:

```
//class Figura ...

//class Circulo ...

//class Cuadrado...

//class Poligono...

public class RecorridoFiguras {


	public static void main(String[]args) {
		Figura[] figuras = new Figura[3];
		figuras[0]=new Circulo();
		figuras[1]=new Cuadrado();
		figuras[2]=new Poligono();
		for(inti=0;i<figuras.length;i++)
		{
			((Figura) figuras[i]).dibuja();
		}
	}
}
```

El código del método main nos permite probar nuestras clases.

3- Queremos crear una aplicación conforme al siguiente esquema



Para ello crearemos una clase llamada **EjercicioVehiculos** que contendrá estas clases y sus herencias. El método calculaPrecio de VehiculoPasajeros hará el siguiente cálculo:

precio=precioPorDiayPlaza * plazas * numDiasAlquiler;

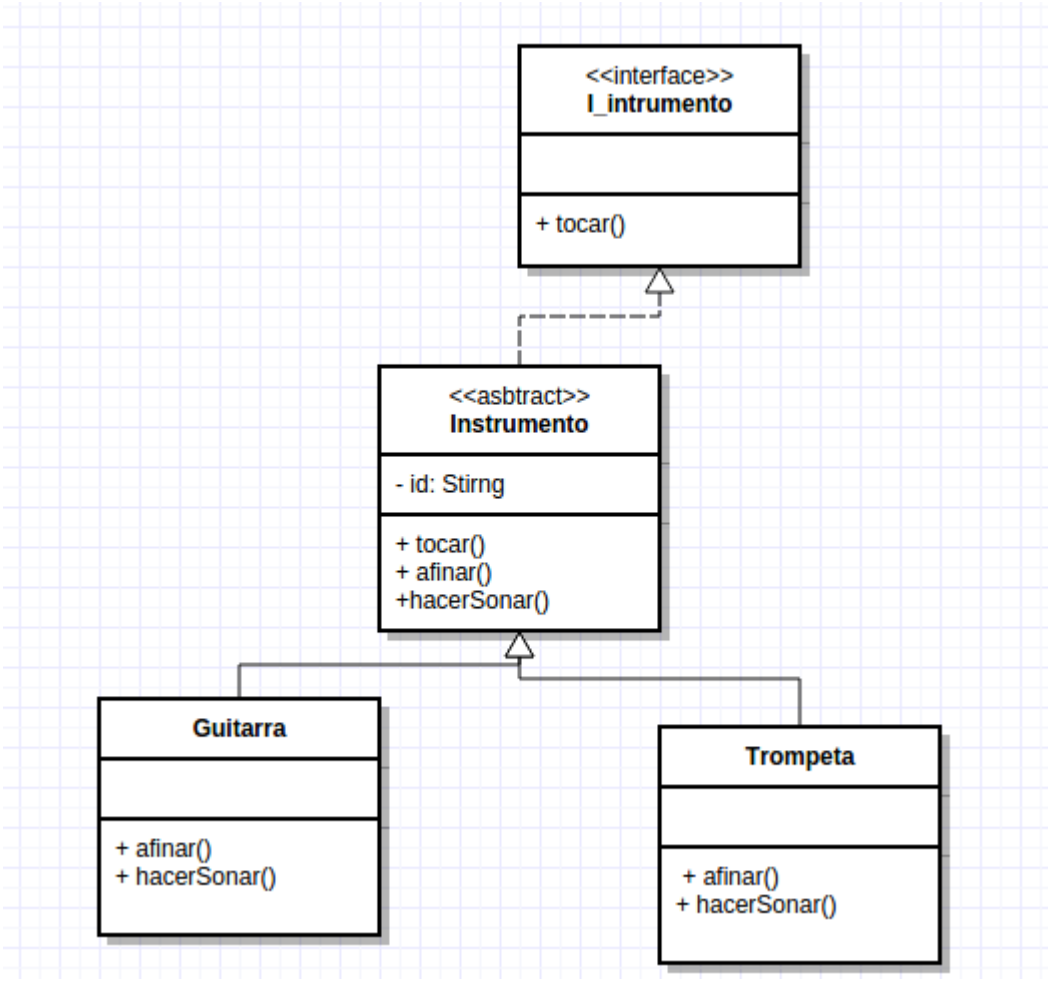
y el método calculaPrecio de VehiculoCarga hará el siguiente cálculo:

precio=precioPorTonelada * numToneladas;

El código a completar de la aplicación será:

```
class Vehiculos{/*A COMPLETAR*/}
class VehiculoPasajeros extendsVehiculos{/*A COMPLETAR*/}
class VehiculoCarga extendsVehiculos{/*A COMPLETAR*/}

public class EjercicioVehiculos {

	public static void main(String[] args) {

		VehiculoPasajeros vehiculoPasajeros = new VehiculoPasajeros();
		vehiculoPasajeros.setMatricula("1234GTX");
		vehiculoPasajeros.setPlazas(5);
		vehiculoPasajeros.setPrecioPorDiayPlaza(4);
		System.out.println("El precio de alquiler del coche "
			+vehiculoPasajeros.matricula+
			+" para 3 dias es "+vehiculoPasajeros.calculaPrecio(3));

		VehiculoCargavehiculoCarga=newVehiculoCarga();
		vehiculoCarga.setMatricula("3431HBH");
		vehiculoCarga.setPMA(3300);
		vehiculoCarga.setPrecioPorTonelada(60);
		System.out.println("El precio de alquiler del camión "
			+vehiculoCarga.matricula+
			+" para 2000 Kg es "+vehiculoCarga.calculaPrecio(2));

	}

}
```

4- Queremos crear una aplicación sobre instrumentos. Para ello implementaremos el siguiente esquema



La clase **Guitarra** mostrará por pantalla en el método afinar `"Afino las cuerdas de la guitarra"` y en el método hacerSonar mostrará `"Rasgo las cuerdas"`

La clase **Trompeta** mostrará por pantalla en el método afinar `"Afino mi trompeta y sus teclas"` y en el método hacerSonar mostrará `"Soplando por la boquilla suena mi trompeta"`

El alumno debe completar el siguiente código

```
interface I_instrumento {
      public void tocar();
      }

abstract class Instrumento implements I_instrumento
      {
            public void tocar()
            {
             afinar();
             hacerSonar();
            }
            protected abstract void afinar();
            protected abstract void hacerSonar();
      }

class Guitarra extends Instrumento
{
      /*CODIGO A COMPLETAR*/
}

class Trompeta extends Instrumento
{
      /*CODIGO A COMPLETAR*/
}
public class TestInstrumentos {
        public static void main(String[]args)
        {
            Instrumento instrumento = newGuitarra();
            instrumento.tocar();

            instrumento = new Trompeta();
            instrumento.tocar();
        }
}
```