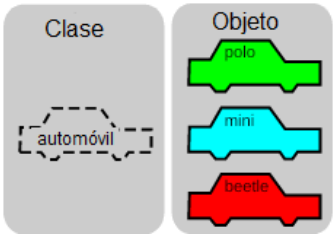
DAM
PROGRAMACIÓN

UD3-3-POO
CONCEPTOS BÁSICOS

Introducción

- La ***Programación Orientada a Objetos*** (POO) tiene como punto central el uso de objetos.
- Un **objeto** es una estructura lógica que representa un elemento o entidad del mundo real.
- La representación de los objetos se basa en los siguientes aspectos:



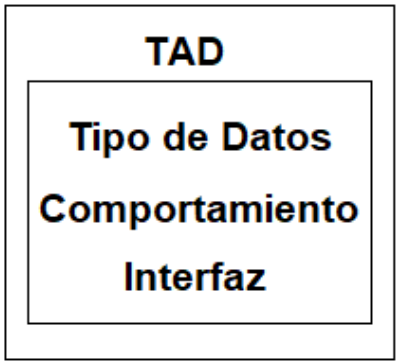
- Para ***identificar*** a los ***objetos*** se usan ***nombres*** que permitan una rápida asociación con el objeto con su representación.
  - Se puede definir un objeto denominado *automóvil* que representa un coche en el mundo real.
- Para representar el ***estado*** **del** ***objeto*** se utilizan ***atributos*** o ***propiedades*** que almacenan diferentes valores que pueden ir cambiando en el tiempo.
  - El coche puede estar representado por velocidad y potencia.
- Para describir el comportamiento del objeto y cómo éste interactúa con otros de su mismo tipo o de otro tipo se definen ***métodos***, que corresponden con acciones específicas del objeto, una vez más, simulando el mundo real.
  - p.e. avanzar, frenar, abrir la puerta derecha, adelantar, etc.

- **La POO se caracteriza por:**
  - **El uso de TDA** (*Tipos abstractos de datos*) para encapsular los datos y las operaciones que se realizan sobre ellos. Los **TDA** permiten que se utilicen sus métodos sin conocer como están implementados.
  - **Diseño modular**. La modularidad es la propiedad que permite subdividir una aplicación en partes más pequeñas, llamadas módulos, cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.
  - **Reutilización del software**. Permite reutilizar partes de una solución para resolver un segundo problema, donde se comparte parte de la lógica resuelta del primer problema.

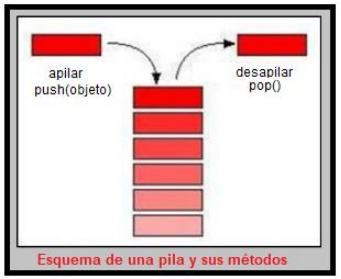
No basta utilizar un lenguaje OO (*Orientado a objetos*) para realizar un programa orientado a objetos, también hay que seguir un paradigma de programación OO

Tipo abstracto de dato (TAD)

- Un **TDA** es un tipo de datos definido por el programador que consta de **estructuras de datos propias** y **operaciones** que se pueden realizar con esos datos y que permiten su uso por medio de una **interfaz**.
- Un **TDA** especifica:
  - Qué se puede almacenar en el TDA.
  - Qué operaciones se pueden realizar sobre/por el TDA.
- **En Java**
  - **Las interfaces se utilizan para definir los TAD.**
  - **Las clases permiten implementar TAD.**



- Ejemplo Java:
  - Una pila es una estructura de datos que permite almacenar y recuperar datos solo por la parte superior de la misma.
  - Las pilas en Java tienen definida una *interfaz,* **Stack**, que determina los métodos que se pueden realizar sobre ellas.



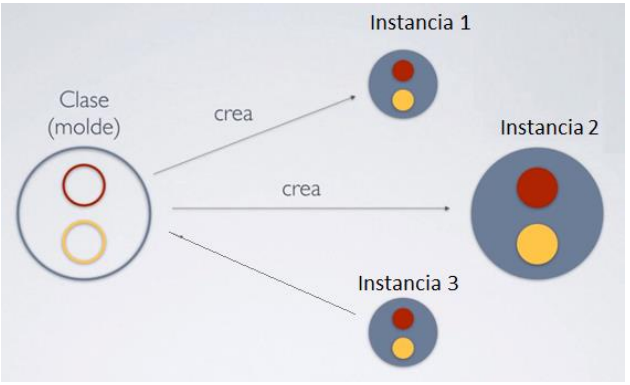
- **Algunos métodos definidos en la interfaz Stack:**
  - **push (*objeto*)**-> inserta un objeto en la cima de la pila.
  - **pop()**-> extrae un objeto de la cima de la pila y lo devuelve.

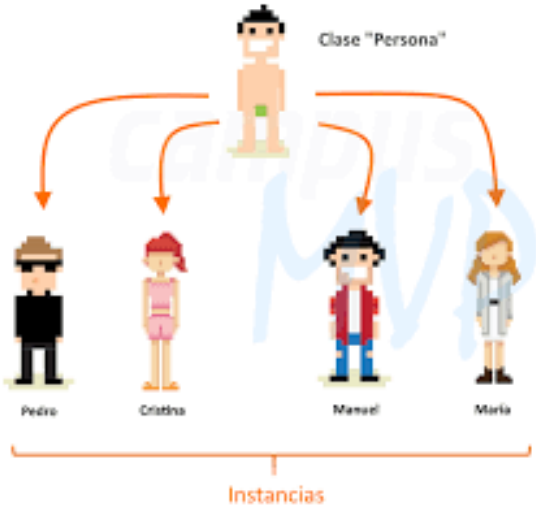
Ejemplo: si se crea una bolsa de chuches como un TDA utilizando la interfaz *Stack*, se deben implementar los métodos para para que:

- El TDA almacene chuches.
- El TDA permita introducir un caramelo y extraerlo.

Terminología básica de la POO

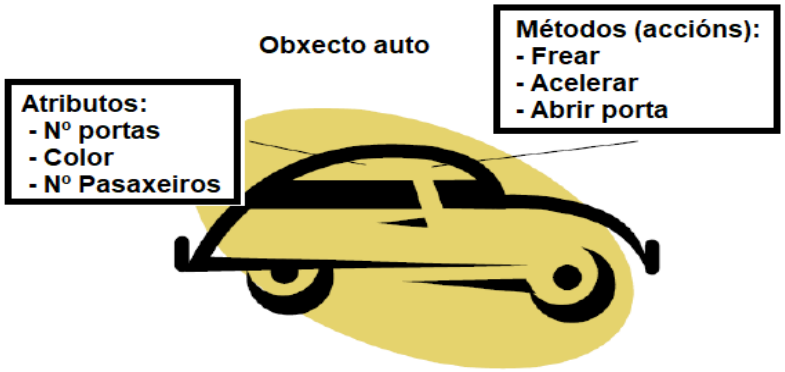
- **Objeto:** Es una abstracción de una entidad del mundo real (una persona, un ordenador…) o un objeto funcional (una base de datos, un fichero, etc.).
- **Clase**. Es una herramienta que permite definir un molde de un objeto mediante un lenguaje de programación. Las clases incluyen atributos y métodos del objeto que moldean. P.e. la clase *coche* debe contener los atributos (color, tamaño, marcha, marca, etc.) y métodos (abrir puerta, acelerar, frenar, etc.) que caracterizan a los coches.
- **Instancia.** Una instancia es un objeto particular de una clase. Una instancia tiene su propio nombre y se diferencia de otras instancias de la misma clase por los valores particulares de sus atributos. P.e. un *mercedes* creado a partir de la clase *coche* de color rojo, mini que puede ejecutar los métodos de la clase *coche (acelerar, frenar, etc.).*





Terminología básica de la POO

- **Atributos o Propiedades.** Los atributos o propiedades son variables de estado cuyo valor se asocia a una instancia particular de una clase de objeto. P.e. color, tamaño, marca, etc.
- **Métodos.** Un método es un algoritmo asociado a una clase, *cuya ejecución se desencadena tras la recepción de un mensaje*. Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un *evento* con un nuevo mensaje para otro objeto del sistema. P.e. *cambiarMarcha (), adelantarCoche ().*
- **Paso de mensajes.** *El envío de mensajes a una instancia de una clase produce la ejecución de un método de la misma*. P.e. *coche1.cambiarMarcha (2), coche1.adelantarCoche (coche2)*



Para **crear una instancia de un objeto** se usa el operador **new** que…

1. Asigna memoria para el objeto
2. Devuelve una referencia a la ubicación de memoria
3. Invoca el constructor de clase

```
public class Principal {
  public static void main(String[] args) {
    String mensaje;

    //Crea una instancia de la clase Coche
    Coche miCoche = new Coche();

    //Ejecuta un método a partir de la instancia
    mensaje = miCoche.arrancar();
    System.out.println(mensaje);

    //Modifica una propiedad a partir de la instancia
    miCoche.vel_max = 200;
    System.out.println("Puede alcanzar una velocidad de "+
                                          miCoche.vel_max);
  }
  //De forma automática, se elimina la instancia de memoria
  //cuando ésta queda inaccesible
}
```

```
public class Coche {
    // atributos:
    String marca;
    double vel_max;
    int potencia;
    String tipo_carburante;
    double velocidad;
    double aceleracion;

    // métodos:
    String arrancar() {
        // instrucciones para arrancar el coche
        return "Esta arrancando";
    }
    String frenar() {
        // instrucciones para frenar el coche
         return "Esta frenando";
    }
    String acelerar() {
        // instrucciones para acelerar el coche
         return "Esta acelerando";
    }
    String girar_derecha(short grados) {
        // instr. para girar a la derecha
         return "Esta girando a la derecha";
    }
}
```

A partir de una **instancia** se puede llamar a las **propiedades** y **métodos** de la clase a la que pertenece.

Ejercicio

- Crea un nuevo proyecto **losCoches** con las siguientes clases:
  - **Coche**
  - **Principal** (*main*)
    - Crear dos instancias de **Coche**
    - Asignar valores a las propiedades de cada instancia.
    - Ejecutar los métodos de cada instancia.
  - Añadir a la clase Coche una propiedad llamada **posición** que almacene su posición en una carrera.
  - Implementar, en la clase **Coche**, el método **adelantar**(**Coche** ***elCocheAdelantado***) y simular un adelantamiento en el método ***main***

Práctica: objeto, clase, instancia, métodos y propiedades

- Hacer una *clase* **Rectangulo** con los siguientes elementos:
  - **Propiedades**

```
int lado1;
int lado2;
```

  - **Métodos**

```
public double area() {
        return lado1*lado2;
}
public double perimetro() {
        return lado1*2+lado2*2;
}
```

- **Acciones en Main**:
  - Crear dos objetos de tipo **Rectángulo**.
  - Asignar valores a las propiedades directamente.
  - Imprimir por pantalla el resultado del área y el perímetro de cada uno de los objetos.

Para consultar los tipos de variables posibles en Java:
https://hoxejaveamos.blogspot.com/?zx=395bb08af976f211
En el bloque de consulta: Java y tipos de datos

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

		Rectangulo r1= new Rectangulo();
		r1.lado1=2;
		r1.lado2=3;

		System.out.println(r1.lado1);
		System.out.println(r1.lado2);

		Rectangulo r2= new Rectangulo();
		r2.lado1=5;
		r2.lado2=6;
		System.out.println(r2.lado1);
		System.out.println(r2.lado2);

		System.out.println(r1.area());
		System.out.println(r1.perimetro());
	}
}
```

Elementos del paradigma POO

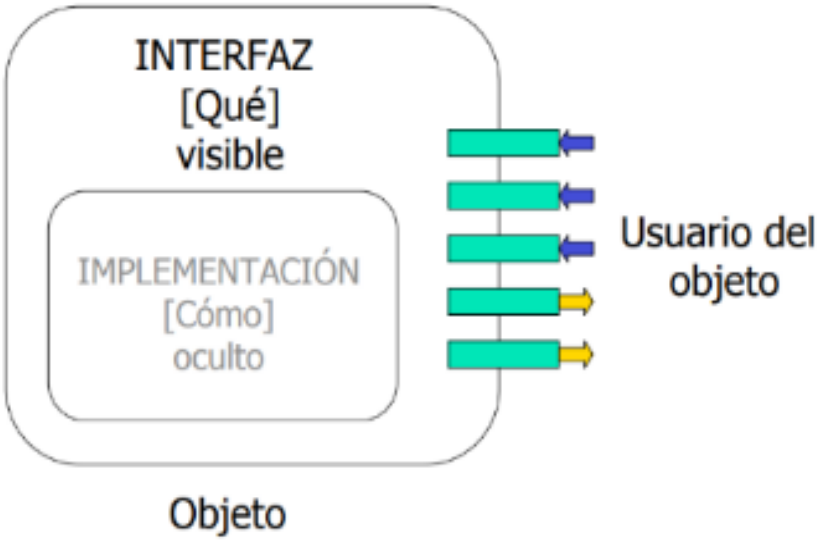
- Para programar en POO, no basta con utilizar un lenguaje OO para programar orientado a objetos, para ello es necesario seguir un paradigma de programación OO.
- Tendremos que aprender a manejar los siguientes conceptos:
  - **Métodos privados, públicos y constructores.**
  - **Encapsulación**.
  - **Composición**
  - **Paso de mensajes**.
  - **Herencia**.
  - **Polimorfismo**.
  - **Otras propiedades:**
    - **Genericidad**.
    - **Recolección basura**.
    - **Excepciones**.

Métodos: privados, públicos y constructores.

- **Métodos privados.**
  - Son métodos que no son accesibles desde fuera de la clase en la que están definidos.
  - Se acompañan de la palabra reservada **private** en su declaración.
- **Métodos públicos**:
  - Son métodos que son accesibles desde fuera de su clase.
  - Se acompañan de la palabra reservada **public** en su declaración.
- **Constructores**:
  - Son métodos que determinan como se construyen las instancias que se crean a partir de la clase.
  - El constructor es un método que se llama igual que la clase y puede tener distintos parámetros de entrada.
  - Puede haber más de un constructor para una clase.
  - Todas las clases tienen un constructor por defecto aunque no se implemente de forma expresa. El constructor por defecto, no tiene ninguna instrucción, solo permite crear el objeto.
  - Cuando se define una un constructor de forma explícita en una clase, el constructor por defecto ya no existe.
  - Los constructores no devuelven ningún valor.
  - Cuando construimos un objeto con **new Objeto()** estamos llamando al constructor.

Encapsulación

- La *encapsulación* permite que los objetos puedan tener información accesible desde el exterior e información privada y oculta al exterior.
- Para ello, los objetos suelen presentar sus *métodos como interfaces públicas y sus atributos como datos privados* e inaccesibles desde otros objetos



Para permitir que otros objetos consulten o modifiquen los atributos de los objetos, las clases suelen presentar ***métodos de acceso***.

El acceso a los datos de los objetos es controlado por el programador, evitando efectos laterales no deseados.

Para generar los métodos
https://hoxejaveamos.blogspot.com/2020/10/netbeans-generar-settersgetters.html

Métodos getter y setter

- En POO se considera una buena práctica:
  - Ocultar todos los atributos.
  - Implementar métodos públicos para acceder a los atributos.
  - Poner como públicos solamente aquellos métodos que sea necesario.
- Para controlar el acceso las propiedades se implementan métodos **getter** y **setter.**
- La implementación de un método set/get para un atributo depende de las necesidades de funcionamiento de la clase.
- Los métodos set/get pueden tener cualquier código y tipo y número de parámetros.

```
public class Coche {
    // atributos:
    private String marca;
    private double vel_max;
    private int potencia;
    private byte posicion;

    //Métodos get que permiten la consulta
    //del valor de los atributos
    public String getMarca() {
        return marca;
    }
    public double getVel_max() {
        return vel_max;
    }
    public int getPotencia() {
        return potencia;
    }
    public byte getPosicion() {
        return posicion;
    }

    //Métodos set que permiten la modificación
    //del valor de los atributos
    public void setMarca(String marca) {
        this.marca = marca;
    }
    public void setVel_max(double vel_max) {
        this.vel_max = vel_max;
    }
    public void setPotencia(int potencia) {
        this.potencia = potencia;
    }
    public void setPosicion(byte posicion) {
        this.posicion = posicion;
    }
}
```

Objeto *this*

- El objeto **this** se utiliza para hacer llamadas a propiedades/métodos de la misma clase.
- **Ejemplo**: en el método **setMarca()** tenemos **marca** como parámetro y como atributo, para diferenciar uno de otro debemos indicar el objeto **this** en la referencia al atributo de la clase actual.

```
public class Coche {
    // atributos:
    private String marca;
    public void setMarca(String marca) {
        this.marca = marca;
    }
```

Ejercicio

- En el proyecto **POO_losCoches** realizar las siguientes modificaciones:
  - En la clase **Coche**:
    - Añadir una propiedad denominada **posición** de tipo **byte** que contendrá la posición relativa del coche en una carrera.
    - Añadir un método **adelantar()** que recibirá como parámetro una instancia de **Coche** al que va a adelantar. Este método modificará las posiciones del coche que adelante y del coche adelantado.
- En la clase **Principal** (*main*):
  - Crear dos instancias de la clase **Coche**
  - Dar valor a las propiedades de las instancias, incluidas la posición inicial en la carrera.
  - Simular el adelantamiento de un coche a otro e imprimir por pantalla la posición cada vez que se produce esta operación.

Coche1-Principal1

Ejemplo

```
public class Cuenta {

    private String titular;
    private String nCuenta;
    private double tipoInteres;
    private double saldo;

    //getters y setters
    public void setTitular(String titular) {
        this.titular = titular;
    }
    public void setNumeroCuenta(String nCuenta, Byte
nivel){
        //Solo deja realizar el cambio a quién tenga
        //un determinado nivel de acceso
        if (nivel==1) this.nCuenta = nCuenta;
    }
    public void setTipoInteres(double tipoInteres) {
        this.tipoInteres = tipoInteres;
    }

    public String getTitular() {
        return titular;
    }
    public String getNCuenta() {
        return nCuenta;
    }
    public double getTipoInteres() {
        return tipoInteres;
    }
    public double getSaldo() {
        return saldo;
    }
}
```

Si no interesa que se acceda al saldo directamente, podemos no implementar **setSaldo**() y que éste solo pueda ser alterado al ingresar o retirar dinero.

```
//método ingreso
public boolean hacerIngreso(double
n) {
  boolean ingresoCorrecto = true;
  if (n < 0) {
      ingresoCorrecto = false;
  } else {
      saldo = saldo + n;
  }
  return ingresoCorrecto;
}

//método reintegro
public boolean hacerReintegro(double
n) {
  boolean reintegroCorrecto = true;
  if (n < 0) {
     reintegroCorrecto = false;
  } else if (saldo >= n) {
    saldo -= n;
  } else {
    reintegroCorrecto = false;
  }
  return reintegroCorrecto;
}
```

Ejercicio

- Implementar la clase Cuenta del ejemplo anterior y crear un proyecto que permita ingresar, retirar dinero y ver el saldo de una cuenta

Constructor de una clase I

Usar la clase **Libro** del Aula Virtual

- Toda clase tiene un **constructor por defecto que reserva memoria para almacenar las propiedades y métodos de la clase**.
- El constructor de una clase tiene como **nombre de método el mismo que la clase** y **no tiene valor de retorno**. En la gran mayoría de los casos es **public**,
- El constructor por defecto es llamado cuando se ejecuta la sentencia **new NombreClase**.

- El constructor por defecto no está implementado en la clase, pero existe.

Constructor de una clase II

- En el momento que se implementa un constructor en una clase, el constructor por defecto desaparece.

```
public Libro(String elTitulo, int elTotalPaginas){
    titulo=elTitulo;
    totalPaginas=elTotalPaginas;
}
```

Si se implementa un constructor, el constructor por defecto deja de existir

~~`public Libro(){ }`~~

~~`Libro l1 = new Libro();`~~

NO puedo llamar al constructor por defecto
Tengo que llamar al constructor implementado

```
Libro l2 = new Libro ("Semillas de cabañas", 91);
```

Si se quiere que se pueda crear un objeto de forma similar al constructor por defecto es necesario implementarlo en la clase.

Sobrecarga de constructores

- Se pueden implementar distintos constructores para una clase, aportando formas diferentes de crear un objeto.
- El crear distintos constructores para una clase se denomina **Sobrecarga de constructores**.
- Se llama a uno u otro constructor según los parámetros que utilicemos en la llamada, por ello, dos constructores no pueden tener el mismo número de parámetros si estos son del mismo tipo.

Constructores definidos en la clase **Libro**

```
//Crea un objeto sin dar valor a sus propiedades
public Libro(){

}
//Crea un objeto dando valor solo al título
public Libro(String elTitulo){
    titulo=elTitulo;
}
//Crea un objeto dando valor al titulo y al total de páginas
public Libro(String elTitulo, int elTotalPaginas){
    titulo=elTitulo;
    totalPaginas=elTotalPaginas;
}
```

Formas de crear una **instancia** de la clase **Libro**

```
Libro l1 = new Libro();

Libro l2 = new Libro ("Semillas de cabañas");

Libro l3 = new Libro ("Semillas de cabañas", 91);
```

Ejercicio

```
public class Principal {

    public static void main(String[] args) {

        Coche c1 = new Coche("toyota", "yaris");

        System.out.println(c1.getMarca());
        System.out.println(c1.getModelo());

        //sobrecarga
        Coche c2 = new Coche("yaris");

        System.out.println(c2.getMarca());
        System.out.println(c2.getModelo());
    }
}
```

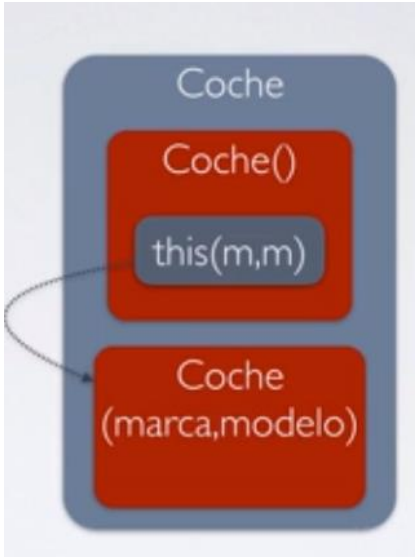
Modificar la clase **Coche** para que funcione este código

Salida por pantalla

```
<terminated>
toyota
yaris
toyota
yaris
```

Método this()

- Los constructores pueden ser llamados desde otro constructor de su misma clase utilizando método **this()**




```java
public class Coche {
   private String marca;
   private String modelo;

//Delegación de constructores
//La primera línea de un constructor
//puede ser la llamada a otro constructor //sobrecargado

   public Coche(String marca, String modelo) {
      this.marca=marca;
      this.modelo=modelo;
   }
   public Coche() {
      this("yaris","toyota");//Tiene que ser la primera línea
   }
   public String getMarca() {
      return marca;
   }
   public void setMarca(String marca) {
      this.marca = marca;
   }
   public String getModelo() {
      return modelo;
   }
   public void setModelo(String modelo) {
      this.modelo = modelo;
   }
}
```

```java
public class Coche {
   private String marca;
   private String modelo;

//Delegación de constructores
//La primera línea de un constructor
//puede ser la llamada a otro constructor //sobrecargado

   public Coche(String marca, String modelo) {
      this.marca=marca;
      this.modelo=modelo;
   }
   public Coche() {
      this("yaris","toyota");//Tiene que ser la primera línea
   }
   public String getMarca() {
      return marca;
   }
   public void setMarca(String marca) {
      this.marca = marca;
   }
   public String getModelo() {
      return modelo;
   }
   public void setModelo(String modelo) {
      this.modelo = modelo;
   }
}
```

El constructor sin parámetros asigna valores a partir de otro constructor que requiere de los parámetros de marca y modelo.

```java
public class Principal {
   public static void main(String[] args) {
      //Crear un objeto a partir de su constructor
      Coche c=new Coche();
      //El objeto tendra asignado valor a sus propiedades
      System.out.println(c.getMarca());
      System.out.println(c.getModelo());
   }
}
```

Crea un objeto de marca *yaris* y modelo *toyota* sin tener que introducir estos valores

Ejercicio

- Crear una clase **Pais** que permita almacenar atributos para:
  - Nombre.
  - Número de habitantes.
  - Capital.
  - Lenguas oficiales del país.
  - Sistema de gobierno: república o monarquía.
- Para crear un objeto **Pais** se debe poder hacer
  - solo aportando el nombre.
  - aportando el nombre y la capital.
  - aportando el nombre, la capital y el sistema de gobierno.
- A la hora de implementar los constructores utilizar el método **this()** cuando sea posible.
- Implementar métodos ***getter*** y ***setter*** para todos los atributos. Utilizar el objetos **this**.
- En el método **main** de debe crear un menú que permita crear países y mostrar sus datos por pantalla. Utilizar un array de **Pais** para esta operación.

ArrayList

La clase **ArrayList** permite trabajar con listas que pueden contener cualquier tipo de objeto.

Declaración de un **ArrayList** de un tipo Objeto, siendo Objeto una clase propia o del JDK de Java

- **ArrayList<Objeto> lista = new ArrayList<Objeto>()**
  - **Ejemplos**:
    - ArrayList<String> lista1 = new ArrayList<String>();
    - ArrayList<Pais> lista2 = new ArrayList<Pais>();

Agregar un elemento a un ArrayList

- **lista.add(unObjeto)**
  - **Ejemplos**:
    - lista1.add("Margarita"):
    - lista2.add(elPais); //siendo elPais un objeto de la clase Pais

Conocer el tamaño actual de un ArrayList

- *int* **lista.size()**
  - **Ejemplos:**
    - lista1.size(); //devuelve un valor entero con nº de elementos que tiene el ArrayList lista1

Acceder a un elemento del ArrayList **por su índice**

- *Objeto* **lista.get(indice)**
  - **Ejemplos:**
    - String nombre = lista1.get(0);
    - Pais unPais = lista2.get(lista2.size()-1); //devuelve el último elemento de lista2

Variables de instancia vs Variables estáticas

- **Declaración dentro de una clase.**
  - **Instancia**-asociada a una instancia:
    [modificador_acceso] tipo variable
  - **Estática**-asociada a una clase:
    [modificador_acceso] static tipo variable

```
public class CuentaBancaria {
    // Variable de instancia
    public double saldo;

    // Variable de clase
    public static int totalCuentas;
    //.....
}
```

- **Variables de instancia**:
  - Pertenecen a los objeto.
  - Cada objeto tienes sus propias variables de instancia.
- **Variables estáticas**:
  - Cada variable estática pertenece a todas las instancias de la clase que se puedan llegar a crear.
  - Existen aunque no exista ningún objeto de la clase.
  - Se inicializan solo una vez, al inicio de la ejecución y antes de cualquier variable de instancia.

```
public static void main(String[] args) {
   //Creación de objetos
   CuentaBancaria miCuenta1 = new CuentaBancaria();
   //Asignación a las vble de clase
   CuentaBancaria.totalCuentas++;

   CuentaBancaria miCuenta2 = new CuentaBancaria();
  //Asignación a las vble de clase
   CuentaBancaria.totalCuentas++;

  //Asignación a las vbles de instancia
   miCuenta1.saldo = 6000;
   miCuenta2.saldo = 8000;

   System.out.println("La cuenta1 tiene como saldo " + miCuenta1.saldo);
   System.out.println("La cuenta2 tiene como saldo " + miCuenta2.saldo);
   System.out.println("El nº de cuentas es " + CuentaBancaria.totalCuentas);
}
```

- **Llamadas a las variables.**
  - **Instancia**

    nombre_instancia.variable
  - **Estática**

    Nombre_Clase.variable

Métodos estáticos vs Métodos de instancia

- El método estático en Java:
  - Es un método que pertenece a la clase y no al objeto.
  - **Un método estático solo puede acceder a variables estáticas**. No puede acceder a variables de instancia.
  - Puede llamar únicamente a otros métodos estáticos y no puede invocar un método no-estático a partir de él.
  - Solo se necesita el nombre de la clase para acceder a un método estático.
  - Un método estático no puede hacer referencia a **this** o **super.**
- Para indicar que un método es estático es necesario utilizar la palabra reservada de Java **static**.
- Para llamar a un método estático, se utiliza la siguiente sintaxis:

  NombredeClase.nombreMetodoEstatico()
- Los métodos de instancia son los vistos hasta ahora en la creación de las clases.
  - Es un método que pertenece a una instancia y solo puede ser llamado a partir de ella.
  - Puede acceder a las variables estáticas, modificando el valor de ésta para todos las instancias de la clase.

Ejemplo de uso de variables y métodos estáticas

```
public class Coche {

    private String marca;
    private String modelo;

    //variable estática
    private static int contador;

    //Constructor sobrecargados
    public Coche(String marca, String modelo) {
       this.marca=marca;
       this.modelo=modelo;
       //Incremento de la propiedad estática de clase
       contador++;
    }
    public Coche() {
       this("yaris","toyota");
       //Incremento de la propiedad estática de clase
       contador++;

    }

    //Método de clase: static
    public static int getContador() {
           return contador;
    }

    //Métodos de instancia
    public String getMarca() {
           return marca;
    }
    public void setMarca(String marca) {
           this.marca = marca;
    }
    public String getModelo() {
           return modelo;
    }
    public void setModelo(String modelo) {
           this.modelo = modelo;
    }
}
```

Variable estática

Método estático

Cada vez que se crea una instancia de la clase se incrementa la variable estática.

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

   //Crear varios objetos
   Coche c1 = new Coche();
   Coche c2 = new Coche();
   Coche c3 = new Coche();

   //Muestra el valor de la vble estática
   System.out.println(Coche.getContador());
 }
}
```

Provoca el incremento de la variable estática para todas las instancias

Constantes estáticas

- Si a una propiedad estática se le añade el modificador **final** evitaremos que se pueda modificar en ningún lugar del programa.

  [modificador_acceso] **static final NOMBRE**

- Por ejemplo, si en una clase incorporamos la siguiente propiedad:

```
static final String LIMITE = 100;
```

  - la variable **LIMITE** no va a poder ser modificada en ningún lugar del programa y funcionará como una constante.

- Los nombres de las variables con el atributo **final** se deben escribir siempre en mayúsculas.

Inicialización de variables

- La **declaración** de una variable se realiza al definir el tipo y el nombre de la variable
  - **int edad;**
- La **inicialización** se trata de la primera asignación de un valor a la variable.
  - **edad = 30**;
- La **inicialización funciona de forma diferente** según el tipo de variable que estemos declarando.

  - **Variables de instancia y clase**:
    - No necesitan se inicializadas, tan pronto se declaran se les asigna un valor predeterminado.

Tipo	Valor por defecto
boolean	false
byte, short, int, long	0
float, double	0.0
char	'\u0000'
Reference Type	null

  - **Variable locales**:
    - Las variables locales deben ser inicializadas antes de su uso, ya que no tienen un valor predeterminado y el compilador no permitirá usar un valor no inicializado.

- *Los campos estáticos se inicializan antes que cualquier variable local o de instancia de la clase.*

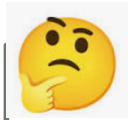
Práctica I

```
public class Ventana_ModeloA {

    //propiedades static y final de la clase
    public static final int ancho = 40;
    public static final int largo = 50;

    //Inicialización de un array de Strings
    public static final String[] colores = {"rojo", "verde", "azul", "gris", "blanco"};

    private String color;

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public static int getAncho() {
        return ancho;
    }

    public static int getLargo() {
        return largo;
    }

    public static String getColores() {
        return Arrays.toString(colores);//método para pasar un array a String
    }
}
```

- Implementa un proyecto que tenga dos tipos de modelos de ventanas, con una clase similar a **ModeloA**.
- Agrega un **contador** a cada clase para que cada vez que se crea una instancia se incremente un contador de la clase.
- Agrega un **constructor** para permitir incluir un **color** a la vez que se crea el objeto.
- En el método **main** debe haber un menú que permita:
  - Simular la **fabricación** de los dos tipos de ventanas de forma diferenciada.
  - Simular la **venta** de los dos tipos de ventanas de forma diferenciada.

Práctica II

- Agregar a cada clase de ventanas un **campo estático** que permita almacenar el **precio** de cada tipo de ventana.
- Cada vez que se realiza una venta de ventanas se deberá mostrar el precio y el coste total en función del número de ventanas vendidas.
- Añadir una opción al menú para ver el total de ventanas vendidas y los ingresos totales por cada tipo de ventana.

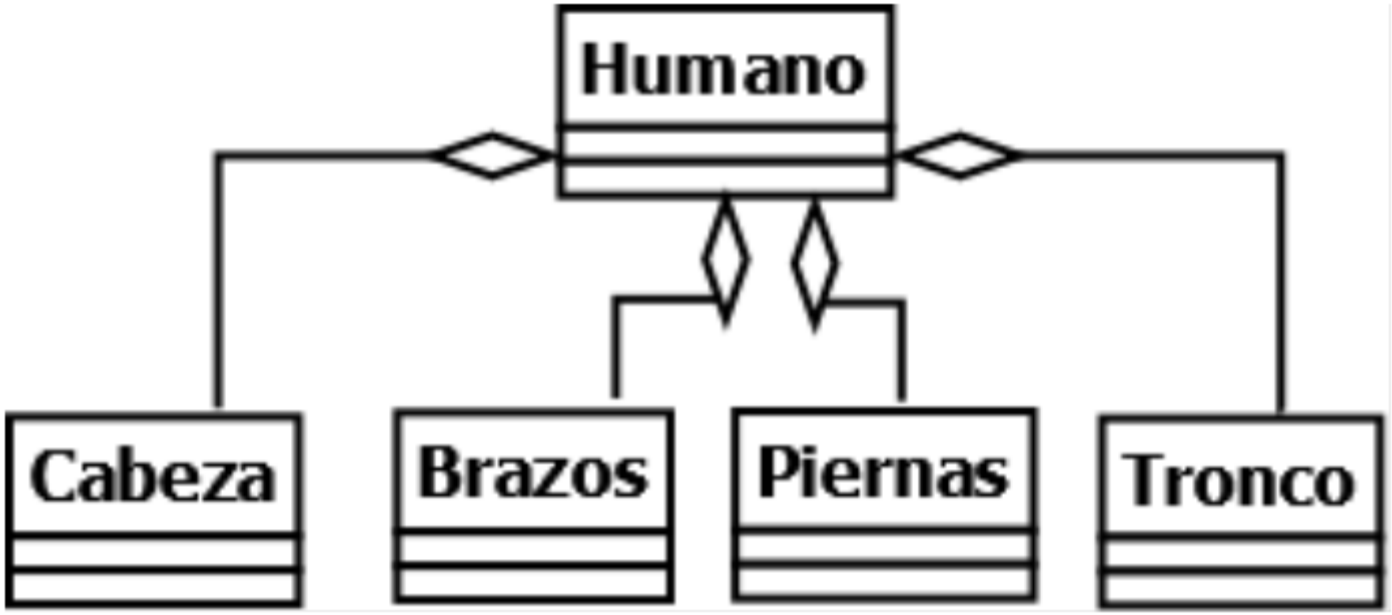


Para reflexionar: Se podría implementar un método estático para modificar el precio del tipo de ventana. ¿Qué problemas podría generar la modificación del precio mediante este método a la hora de calcular los ingresos obtenidos por la venta de las ventanas realizada antes del cambio de precio?

Composición

- La capacidad de una clase de hacer referencia a objetos de otras clases como miembros se le conoce como **composición** y algunas veces como **relación "tiene un"**.
- Un **ejemplo** cercano es la constitución de nuestro cuerpo que está compuesto por varias extremidades: brazos y piernas. A la vez que estás se pueden subdividir en músculos y huesos…y así podemos continuar con células y moléculas.
- En el ejemplo planteado, podemos decir que el cuerpo humano es una jerarquía de composición pues está compuesto por distintos miembros.
- La composición es una forma de reutilización de software, en donde una clase tiene como miembros referencias a objetos de otras clases.

Composición. Representación gráfica



Las jerarquías de composición se representan de forma gráfica utilizando una flecha con la forma 

Composición. Implementación

La clase **Humano** está compuesta por propiedades de las clases **Cabeza**, **Brazos**, **Piernas** y **Tronco**, clases que a la vez pueden estar compuestas de otras.

```
public class Humano {

    private Cabeza laCabeza;
    private Brazos losBrazos;
    private Piernas lasPiernas;
    private Tronco elTronco;

    public Humano(Cabeza laCabeza, Brazos losBrazos, Piernas lasPiernas, Tronco elTronco) {
        this.laCabeza = laCabeza;
        this.losBrazos = losBrazos;
        this.lasPiernas = lasPiernas;
        this.elTronco = elTronco;
    }

    public Cabeza getLaCabeza() {
        return laCabeza;
    }
    public Brazos getLosBrazos() {
        return losBrazos;
    }
    public Piernas getLasPiernas() {
        return lasPiernas;
    }
    public Tronco getElTronco() {
        return elTronco;
    }

    public void setLaCabeza(Cabeza laCabeza) {
        this.laCabeza = laCabeza;
    }
    public void setLosBrazos(Brazos losBrazos) {
        this.losBrazos = losBrazos;
    }
    public void setLasPiernas(Piernas lasPiernas)
    {
        this.lasPiernas = lasPiernas;
    }
    public void setElTronco(Tronco elTronco) {
        this.elTronco = elTronco;
    }
}
```

Composición. Práctica

1. Descarga el proyecto **Proyecto-Composicion-Vehiculo** del aula virtual y analiza la estructura de clases y el método con que se crean los objetos.
2. Implementa un conjunto de clases que tengan una relación de composición con las siguientes características:
   - Las clases deben representar los **Módulos formativos** de un **ciclo de FP**, los **alumnos** y los **docentes**.
   - Lanza una propuesta y la analizaremos en clase.
   - Es necesario implementar operaciones para introducir datos de alumnos y profesores de un módulo formativo.

Paso de mensajes

- Los objetos se comunican entre sí a través del uso de mensajes.
- Esencialmente, el protocolo de un mensaje implica tres partes: **el emisor y el receptor y el mensaje que se transmite**.

Clase Coche

```
public class Coche {

    private String color;
    private int velocidad;
    private float tamaño;
    private Rueda[] ruedas;
    private Motor motor;


    public Coche (String color, int velocidad,
            float tamaño, Rueda[] ruedas,
            Motor motor){
        this.color = color;
        this.velocidad = velocidad;
        this.tamaño = tamaño;
        this.ruedas = ruedas;
        this.motor = motor;

    }

    public void avanzar(){
        motor.inyectarCarburante();
        for (int i=0; i < ruedas.length; i++){
            ruedas[i].girar();
        }
    }

    public static void main (String[] args){

        Rueda[] ruedas = {new Rueda(20,"Dunlop"),
                new Rueda(20,"Dunlop"),
                new Rueda(22, "Dunlop"),
                new Rueda(22, "Dunlop")};
        Coche miCoche = new Coche ("verde", 80,3.2f,
                ruedas, new Motor("Diesel",100));
```

miCoche.avanzar()
ejecuta el método **avanzar()**
que envía un mensaje a la instancia de motor
para que inyecte carburante
motor.inyectaCarburante()

Paso de mensajes

Clase Motor

```
public class Motor {

    private String tipo;
    private int caballos;

    public Motor(String tipo, int caballos){
        this.tipo = tipo;
        this.caballos = caballos;
    }

    public void inyectarCarburante(){...}
}
```

Clase Rueda

```
public class Rueda {

    private double diametro;
    private String fabricante;

    public Rueda (double diametro, String fabricante)
        this.diametro = diametro;
        this.fabricante = fabricante;
    }

    public void girar(){...}

}
```

Paso de mensajes. Práctica

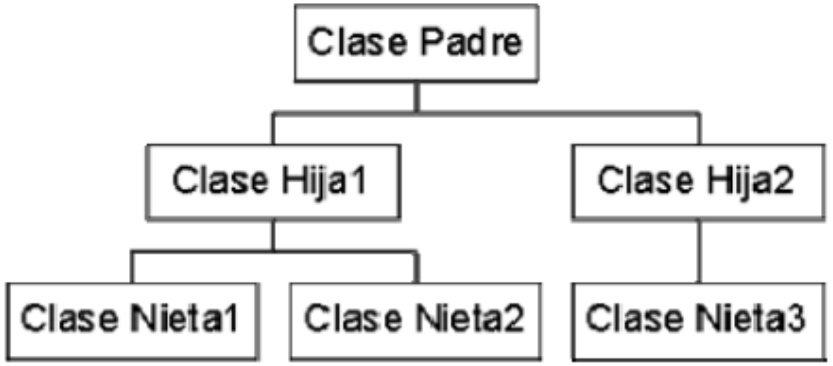
- Modifica la implementación de la clase **Modulo** agregando:
  - un método **matriculacion(Alumno elAlumno)** que matricule al alumno en el módulo. Este método debe tener en cuenta que los módulos pueden matricular a un máximo de **32** alumnos.
  - un método **darBaja(String elDni)** que dé de baja al alumno en un módulo determinado.
  - un método **verProfesor()** que permita ver el nombre del docente que imparte el módulo. Para ver el nombre del docente se deberá llamar a un método **verNombre()** de la clase **Docente**.
  - un método **listarAlumnado()** que permita listar todos el alumnado que está matriculado en el módulo. Para listar los nombres de los alumnos se deberá llamar a un método **verNombre()** de la clase **Alumnado**.

Herencia

- Nuestro mundo se puede observar como **objetos que se relacionan entre sí de una manera jerárquica**.
- Por ejemplo, un perro es un mamífero, y los mamíferos son animales, y los animales, seres vivos...
- Del mismo modo, **las distintas *clases* de un programa se organizan mediante una *jerarquía***.

Mediante la herencia una clase hija hereda todas las propiedades y métodos no privados de la clase padre, excepto los constructores, que no se heredan.

Así se *simplifican los diseños y se evita la duplicación de código*, al no tener que volver a codificar métodos ya implementados.

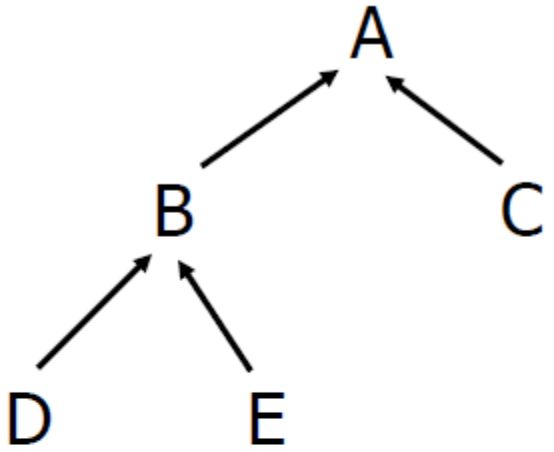


Ejemplo de árbol de herencia

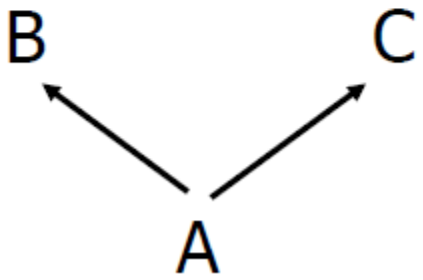
- Durante el proceso de herencia se pueden producir distintas adaptaciones:
  - *Se pueden añadir nuevos atributos.*
  - *Se pueden añadir nuevos métodos.*
  - *Se pueden redefinir métodos con el mismo nombre, bien por **ampliación** (la clase hija amplía la funcionalidad) o por **remplazo** (la clase hija modifica el funcionamiento).*

Herencia. Tipos

- **Herencia simple**:
  - Una clase puede heredad de una única clase.
  - Ejemplos: **Java**, C#.



- **Herencia múltiple**:
  - Una clase puede heredar de varias clases.
  - Ejemplos: Eiffel, C++.



Herencia. Sintaxis y nomenclatura

- Para indicar que una clase hija hereda de una padre, se utiliza la palabra reservada **extends ClasePadre** en la definición de la clase hija.
- Toda clase hereda de la clase **Object** sino se especifica una clase padre concreta.
- **Object** es la clase raíz de la jerarquía de clases del paquete **java.lang**.
- El paquete **java.lang** contiene las clases esenciales y se importa automáticamente sin necesidad de usar **import**.

```
class A {                   // Al no poner extends, deriva implícitamente de Object
    ...
}
class B extends A {         // Deriva de A y, por tanto, de Object
    ...
}
```

El proceso de herencia es ***transitivo***

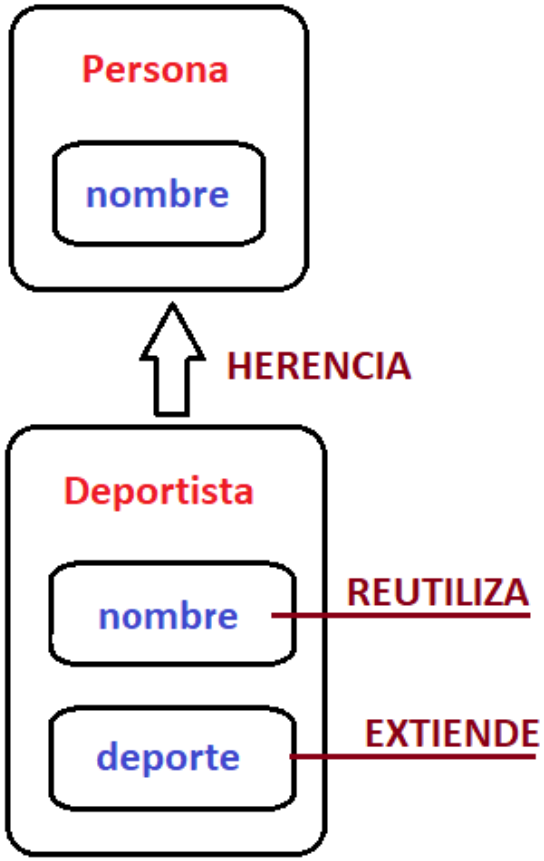
Clase A	Superclase	Padre
Clase B	Subclase	Hijo

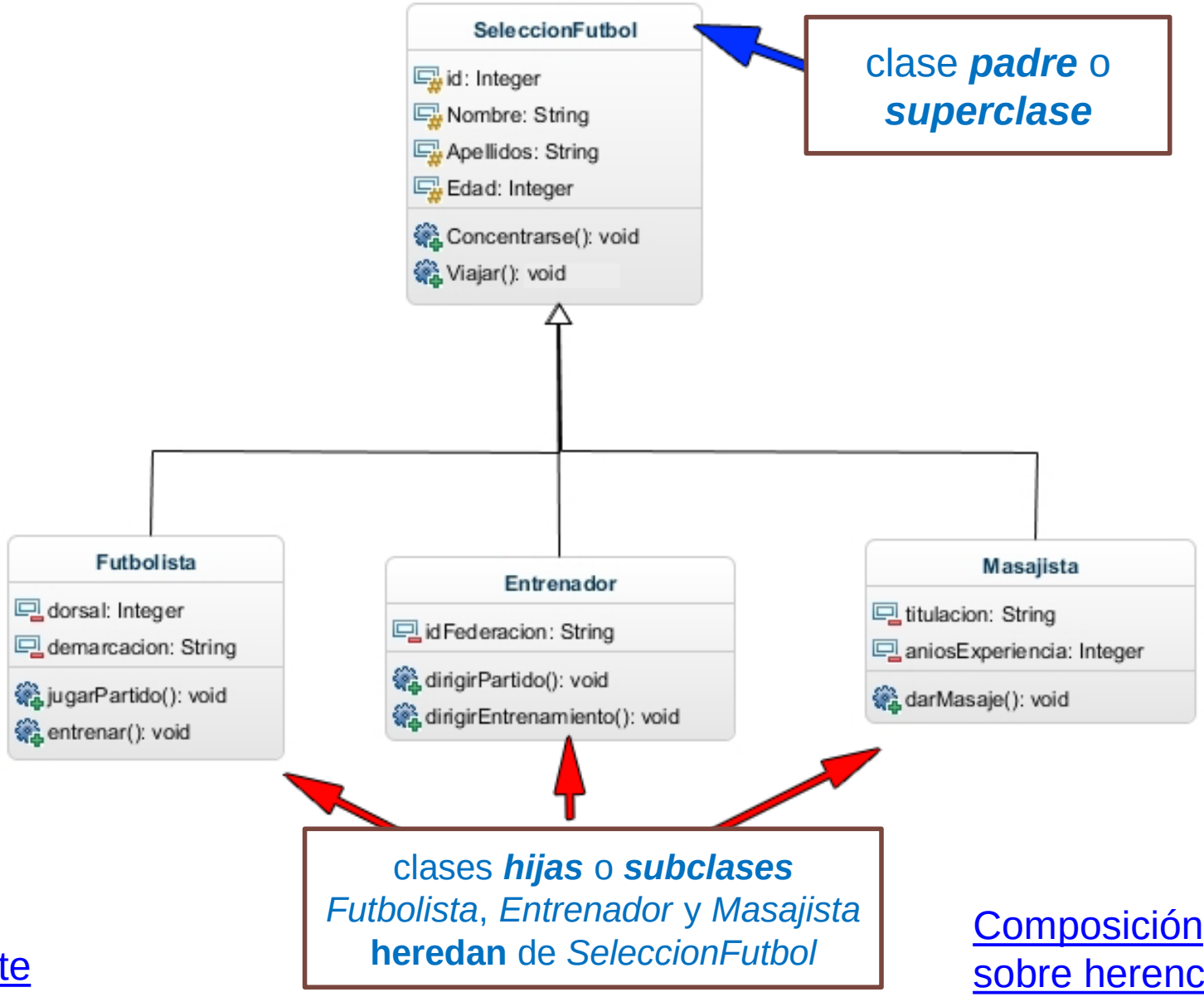
```java
public class Persona {

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

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

Indica que Deportista hereda de Persona

```java
public class Deportista extends Persona {

        private String deporte;

        public String getDeporte() {
                return deporte;
        }

        public void setDeporte(String deporte) {
                this.deporte = deporte;
        }
}
```

```java
public class Principal {

    public static void main(String[] args) {
        Deportista dl = new Deportista();
        dl.setNombre("Manuel");
        dl.setDeporte("Balonmano");

        System.out.println(dl.getNombre());
        System.out.println(dl.getDeporte());
    }
}
```



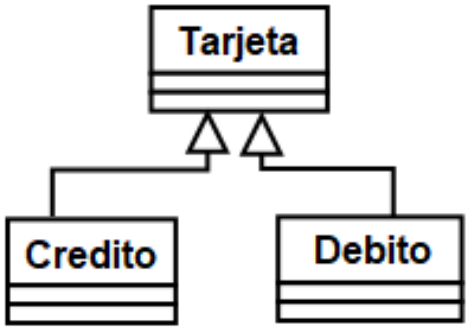


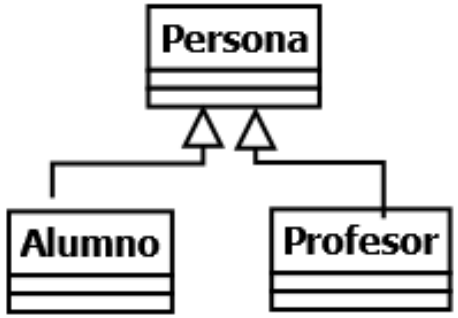
Fuente

Composición sobre herencia

Herencia. Diseño de jerarquías

- **Generalización** (*Factorización*):
  - Se detectan clases con un comportamiento común.
  - Ejemplos:
    - Un alumno puede ser profesor y viceversa.
- **Especialización** (*Abstración*):
  - Se detecta que una clase es un caso especial de otra.
  - Ejemplo:
    - Las tarjetas pueden ser de dos tipos, crédito y débito.
    - Si una tarjeta es de crédito no puede ser de débito.





Herencia. Práctica 1

- Modificar el proyecto de **Módulos/Alumno/Docente** para establecer un árbol de herencia al añadir una clase llamada **Persona** que tenga los atributos y métodos comunes de **Alumno** y **Docente**.
- Modificar la clase **Alumno** y **Docente** para adaptarlas a la nueva situación (eliminar los métodos que ahora estarán en **Persona** y agregar la palabra reservada **extends** en el lugar adecuado)
- Fijarse que al añadir la clase padre **Persona** no será necesario modificar nada de la clase **main.**

Herencia. super()

- El método **super()** nos aporta una herramienta para poder llamar a un constructor de la clase padre desde una clase hija.
- En todo constructor de una clase hija *se llama por defecto al constructor de la clase padre*.

La primera línea de todo constructor debe ser la llamada al constructor de la clase padre. Si esta instrucción no existe, **se produce de igual forma la llamada al constructor por defecto de la clase padre**.

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

```
public class Principal {

    public static void main(String[] args) {

        Persona p1 = new Persona("pedro");
        System.out.println(p1.getNombre());

        Deportista d1 = new Deportista("gema");
        System.out.println(d1.getNombre());
    }
}
```

```
public class Deportista extends Persona {

    private String deporte;

    public Deportista(String nombre) {
        //llamara al constructor de la clase padre que
        //tenga un parametro con String
        super(nombre);
    }

    public String getDeporte() {
        return deporte;
    }

    public void setDeporte(String deporte) {
        this.deporte = deporte;
    }
}
```

Objeto *this I*

- El objeto **this** se utiliza para hacer llamadas a propiedades y/o métodos de la misma clase.
- **Ejemplo**: en el método **setDeporte()** tenemos **deporte** como parámetro y como atributo, para diferenciar uno de otro debemos indicar el objeto **this** en la referencia al atributo.

```
public class Deportista extends Persona {

    private String deporte;

    public String getDeporte() {
        return deporte;
    }

    public void setDeporte(String deporte) {
        this.deporte = deporte;
    }
}
```

Objeto *this II*

- **Ejemplo**: en el método **display()** se llama al método **show()** que está en la misma clase.

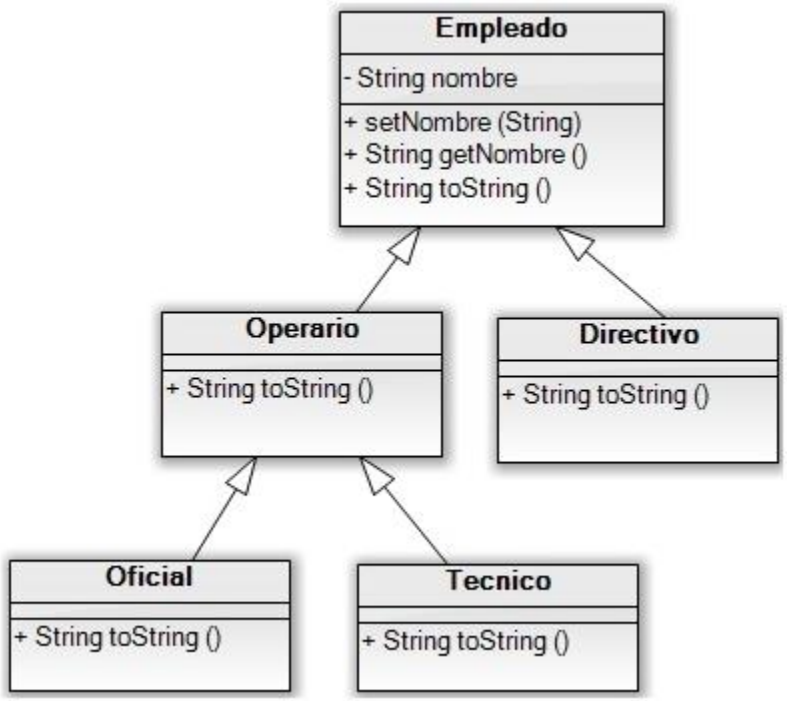
```
class Test {
    public void show() {
        System.out.println("Dentro del método show");
    }
    public void display() {
        // método que llama a show()
        this.show();
        System.out.println("Dentro del método display");
    }

    public static void main(String args[]) {
        Test t1 = new Test();
        t1.display();
    }
}
```

Muestra por Output

```
Dentro del método show
Dentro del método display
```

Ejercicio 1

- Codifica la siguiente jerarquía y crea un elemento de cada tipo de clase de la misma.



Herencia. Ocultación de atributos

- **Atributos**: Los atributos no se pueden redefinir, sólo se ocultan.
  - Si la clase hija define un atributo con el mismo nombre de la clase padre, éste queda inaccesible.
  - El campo de la superclase todavía existe pero no se puede acceder a él de forma directa, es necesario utilizar **super**.

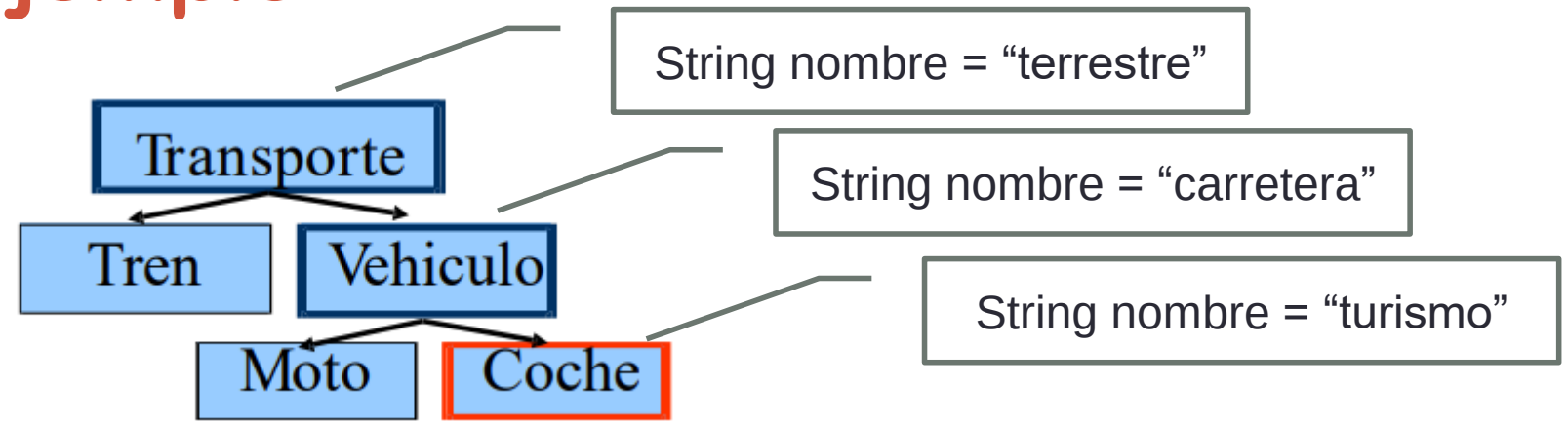
```
class Persona {
    public String nombre = "Juan";
}

class Alumno extends Persona {
    public int nombre = 10003041;

    public void verNombrePersona(){
        System.out.println(super.nombre);
    }
}
```

```
public class Principal {

    public static void main(String[] args) {

        Persona laPersona = new Persona();
        Alumno elAlumno = new Alumno();

        System.out.println(laPersona.nombre);
        System.out.println(elAlumno.nombre);

        elAlumno.verNombrePersona();

    }
}
```

```
Juan
10003041
Juan
```

Salida por pantalla

Herencia. Ocultación de atributos. Ejemplo



- Desde la clase **Coche**, indicar el contenido de;
  - **nombre** → *turismo*
  - **this.nombre** → *turismo*
  - **super.nombre** → *carretera*
  - ~~**super.super.nombre**~~ → no es correcta la sintaxis

Objeto *super*

- Los métodos de una clase derivada también pueden **ocultar a los métodos** de la padre si se llaman igual y tienen tipos de parámetros equivalentes.
- El objeto **super** se utiliza para hacer llamadas a propiedades o métodos de la superclase que estén redefinidas en la clase actual.

```
public class MiClase {
    int i;
    public MiClase() {
        i = 10;
    }

    public void sumaAi(int j) {
        i = i + j;
    }
}
```

```
public class MiNuevaClase extends MiClase {
    public void sumaAi( int j ) {
            i = i + ( j/2 );
        //llama a la superclase no así misma (llamada recursiva)
            super.sumaAi( j );
        }
}
```

- En el siguiente código, el constructor establecerá el valor de i a 10, después lo cambiará a 15 y finalmente el método **suma()** de la clase padre **MiClase** lo dejará en 25:

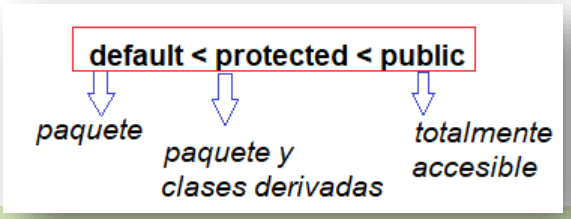
```
MiNuevaClase mnc = new MiNuevaClase();
mnc.sumaAi( 10 );
```

Se puede utilizar super para acceder a los métodos de la clase padre, pero no super.super para acceder a los de la "abuelo".

Sobreescritura (*Overriding*) de métodos en una clase derivada

- Una clase derivada hereda todos los métodos de su clase padre.
- Si en la clase derivada se sobrescribe un método de la clase padre, el método de la clase padre se **oculta** a la clase derivada.
- Para que un método sobrescriba a otro método de la clase padre correctamente, debe tener:
  - **Mismo número de argumento y del mismo tipo**
  - **No tener un acceso más restrictivo que el original.**

```
Public class Instrumento{

  public String tipo;

  public void tocar(){
    System.out.println("Tocar Instrumento");
   }
}


class Guitarra extends Instrumento {

 @Override
 public void tocar() {
    System.out.println("Tocar La Guitarra");
 }

}
```

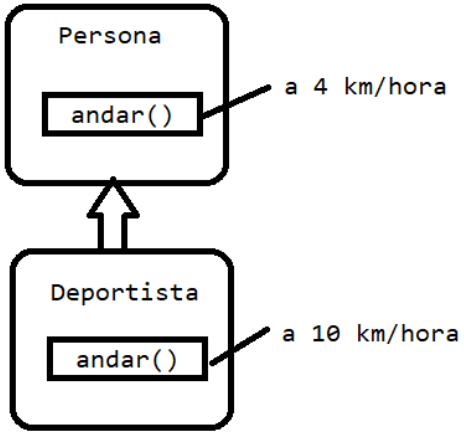


Sobreescritura (*Overrriding*) de métodos en una clase derivada

- Se pueden sobrescribir métodos en las clases derivadas con el objetivo de ampliar la funcionalidad de los métodos que se heredan de la superclase.
- Se aconseja que los métodos que sobrescriban un método de una clase derivada estén acompañados de la anotación **@Override** .
  - El uso de la **anotación @Override** es opcional pero conviene usarla para forzar al compilador a verificar que se está sobrescribiendo correctamente un método, y de ese modo evitar errores en tipos de ejecución, que serían mas difícil de detectar.

```
class LaSuperclase {
   public void metodoA() {
       System.out.println("Superclase A");
   }
}
class LaDerivada extends LaSuperclase {
  @Override
   //Aquí el uso de @Override sería correcto
   public void metodoA() {
       System.out.println("Derivada A");
   }
   //@Override...aquí generaría un error
   //porque no sobrescribe ningún método
   public void metodoB() {
       System.out.println("Derivada B");
   }
}
```

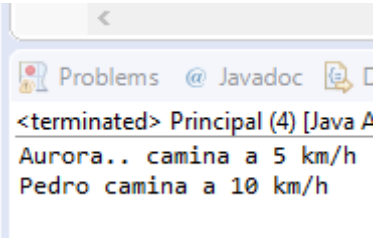
Ejercicio 1 overriding




```
public class Persona {
    private String nombre;

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

    public void andar() {
        System.out.println(getNombre() + ".. camina a 5 km/h");
    }
}
```

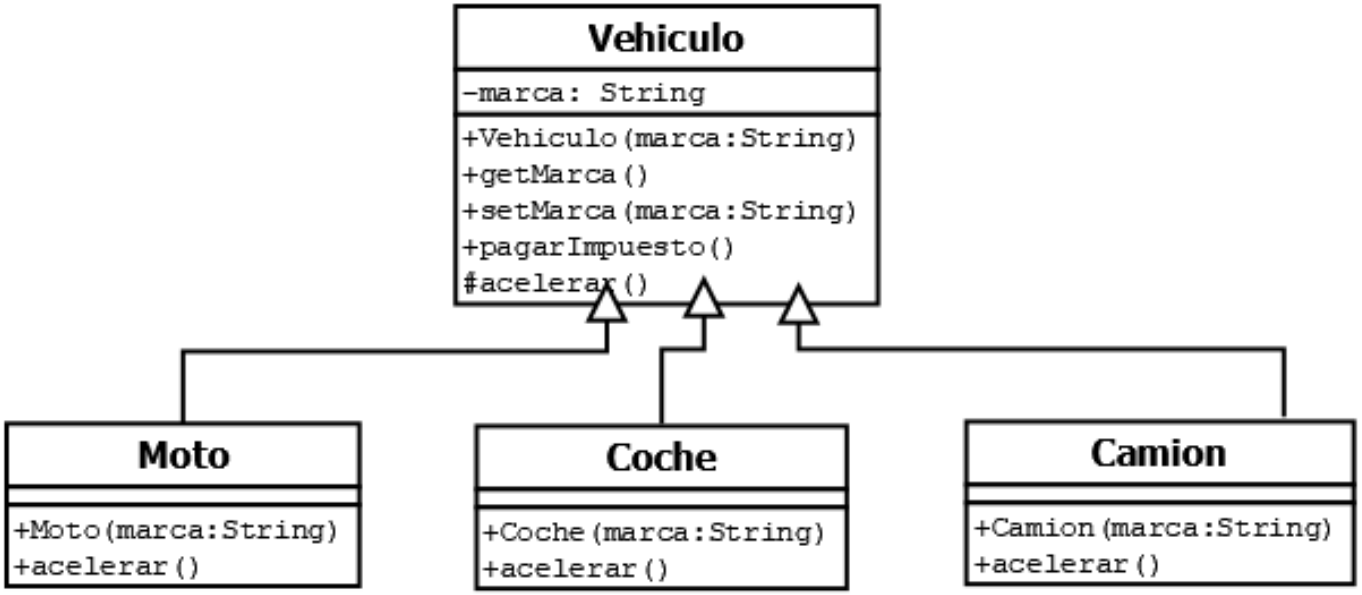
```
public class Deportista extends Persona {
    private String deporte;

    public String getDeporte() {
        return deporte;
    }
    public void setDeporte(String deporte) {
        this.deporte = deporte;
    }

     @Override
    //Método que sobreescribe el método andar
    public void andar() {
        System.out.println(getNombre() + " camina a 10 km/h");
    }
}
```

```
public static void main(String[] args) {
        Persona p1 = new Persona();
        p1.setNombre("Aurora");
        p1.andar();//A 5 km/hora

        Deportista d1 = new Deportista();

        d1.setNombre("Pedro");
        d1.andar();//A 10 km/hora
}
```



Ejercicio 2 overriding

- Implementar las clases según el gráfico **UML** y el método **main**.
- El método que se sobrescribe es el método *acelerar()*. Hacer que muestre un mensaje diferente para cada clase.
- Crear un objeto de cada tipo en **main** y llamar al método **acelerar()** desde cada una de las instancias creadas.



Herencia. Métodos (final)

- En Java se puede aplicar el modificador **final** a un **método** para indicar que no puede ser redefinido.

```
class A {
    final void metodo() {
        System.out.println("Este es un método final.");
    }
}
```

```
class B extends A {
    void metodo() {
        System.out.println("Este método es erróneo!");
    }
}
```

```
metodo() in B cannot override metodo() in A
  overridden method is final
----
(Alt-Enter shows hints)
```

Herencia. Clases (final)

- Asimismo, el modificador **final** es aplicable a una **clase** indicando que no se puede heredar de ella.

```
final class A {
   //...
}
//La siguiente clase no es correcta
//Una clase final no puede ser superclase
class B extends A {
    //ERROR
}
```

cannot inherit from final A

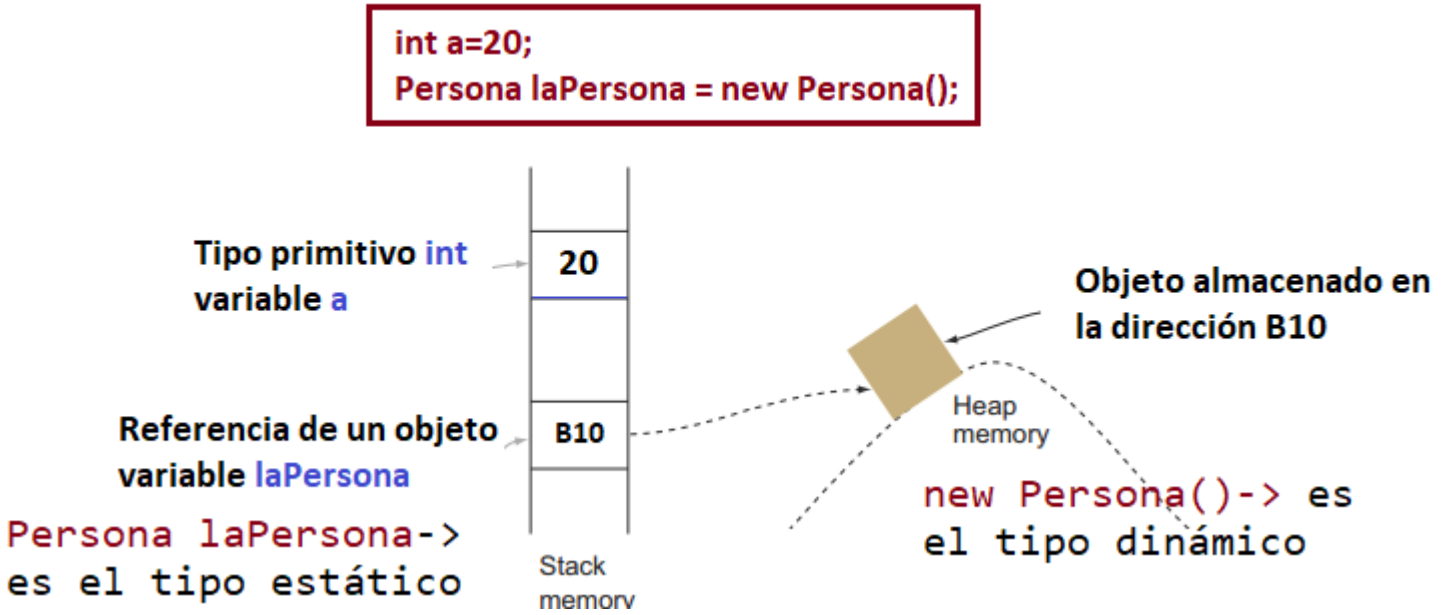
(Alt-Enter shows hints)

Destrucción de objetos

- Los objetos son almacenados en memoria en una zona reservada para el programa que se está ejecutando denominada *heap*.
- La asignación de memoria a los objetos se adapta a las necesidades del objeto para permitir el almacenamiento de sus métodos y propiedades.
- La referencia de la localización del objetos en la memoria *heap* se almacena en el *stack*.
- Cuando un objeto ya no es referenciado desde el *stack* porque el ámbito de la referencia ya no es hábil, el objeto sigue ocupando espacio en el heap.
- Java dispone de un mecanismo automático denominado **recolección basura** *(garbage collection)* que libera la memoria de los objetos que no están referenciados en el **stack**.

Polimorfismo

- *Polimorfismo* quiere decir "*un objeto y muchas formas*". En POO, ***el polimorfismo hace referencia a la capacidad de una variable de instancia de referenciar en tiempo de ejecución a instancias de distintas clases***. En la POO el polimorfismo se sustenta en las relaciones jerárquicas de las clases.
- La relación polimórfica solo se puede dar entre clases que se tengan una relación de herencia donde *el tipo dinámico debe ser descendiente del tipo estático*.



Polimorfismo. Declaración, referencia e instancia

- Cuando una clase **A** es una *superclase* de **B**, se puede crear un objeto de la siguiente forma:

  A miObj= new B()

  A es el tipo **estático** (*la referencia*) y **B** el tipo **dinámico** (*la instancia*)

- De esta forma la instancia **miObj** podrá acceder **únicamente** a los **métodos y propiedades definidos en A**, no tendrá acceso a los métodos ni a las propiedades definidas en **B**, a no ser que se sobrescriban métodos en **B**.

```
class A {
    public void m1() {
        //Aqui se hace algo...
    }
}

class B extends A {
    public void m2() {
        //Aqui se hace otra cosa...
    }
}
```

Dadas las siguientes clases….

Si definimos las siguientes referencias en *main:*

```
B obj1 = new B(); // Tipo estático: B  Tipo dinámico: B
A obj2 = new B(); // Tipo estático: A  Tipo dinámico: B
B obj3 = new A(); // Tipo estático: B  Tipo dinámico: A. ¡ERROR!
```

Tipo estático: B Tipo dinámico: B

obj1 puede hacer referencia solo a los métodos y propiedades de **B.**

Tipo estático: A Tipo dinámico: B

obj2 puede hacer referencia solo a los métodos y propiedades de **A.**

Tipo estático: B Tipo dinámico: A

No se puede crear un objeto con el tipo estático de una subclase del tipo dinámico. Es un error de compilación!!

Según el gráfico tenemos tres clases y **Deportista** e **Informático** que tienen como superclase **Persona**.

Java permite crear un objeto de la siguiente forma

Persona p1=new Deportista()

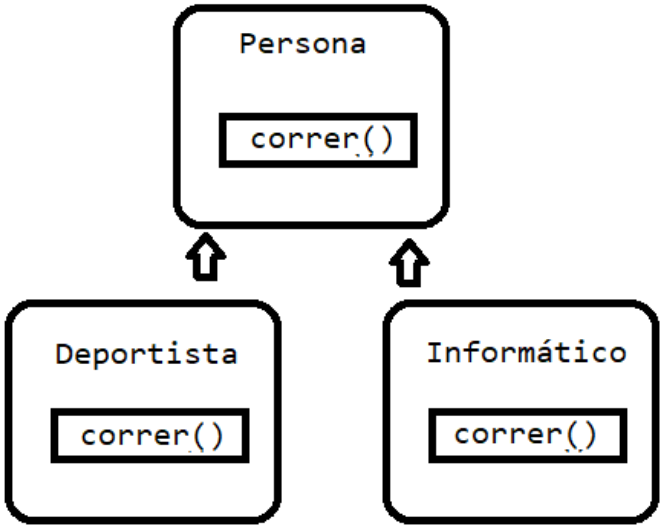
Esto es posible porque la clase **Deportista** es hija de la clase **Persona**.

Pero si ejecutamos

p1.correr()

Nos podemos preguntar que método ejecuta, el de **Persona** o el de **Deportista**.

En el uso del **polimorfismo** cada objeto utiliza los métodos sobrescritos del tipo dinámico, para este caso, el de **Deportista**.



La instancia p1 podría utilizar:

- Sus **métodos propios**, *siempre que estén también declarados en la clase padre*, es decir, el objeto **p1** del tipo **Persona** que apunta al objeto **Deportista** solo puede hacer referencia a los métodos que estén definidos en **Persona**, pero los ejecutará con la implementación que tenga en **Deportista**.
- Los **atributos** a los que accede serán los de la clase **Persona**... aunque estén definidos de forma específica en su clase.
- Los métodos de la clase **Persona** que no estén sobrescritos en la clase **Deportista**.
- *En ningún caso podrá acceder ni a las propiedades ni a los métodos implementados **sólo** en la clase **Deportista**.*

Polimorfismo. Ventajas

- **Simplifica el uso de métodos**

```
//polimorfismo=> que cada objeto ejecuta
//su método sobrescrito en new Clase()
Persona p1=new Deportista ("Pedro");
p1.andar();

Persona p2=new Informatico ("Rosa");
p2.andar();
```

- **Oculta la jerarquía de clases a los programadores**

```
public static void main(String[] args) {
//polimorfismo=> permite ocultar la estructura de clases
        Persona p1=new Deportista ("Pedro");

        Persona p2=new Informático ("Rosa");

        iniciarCaminoPersona(p1);
        iniciarCaminoPersona(p2);
}

//Al llamar al método andar() no tengo que saber de
//que tipo específico es el objeto. Se ejecutará el método
//andar() de la clase a la que pertenezca el objeto que se
//pasa por parámetro
public static void iniciarCaminoPersona(Persona p) {
         p.andar();
}
```

Ejercicio I

- Suponemos la superclase **Instrumento** que tiene una clase derivada **Guitarra.**

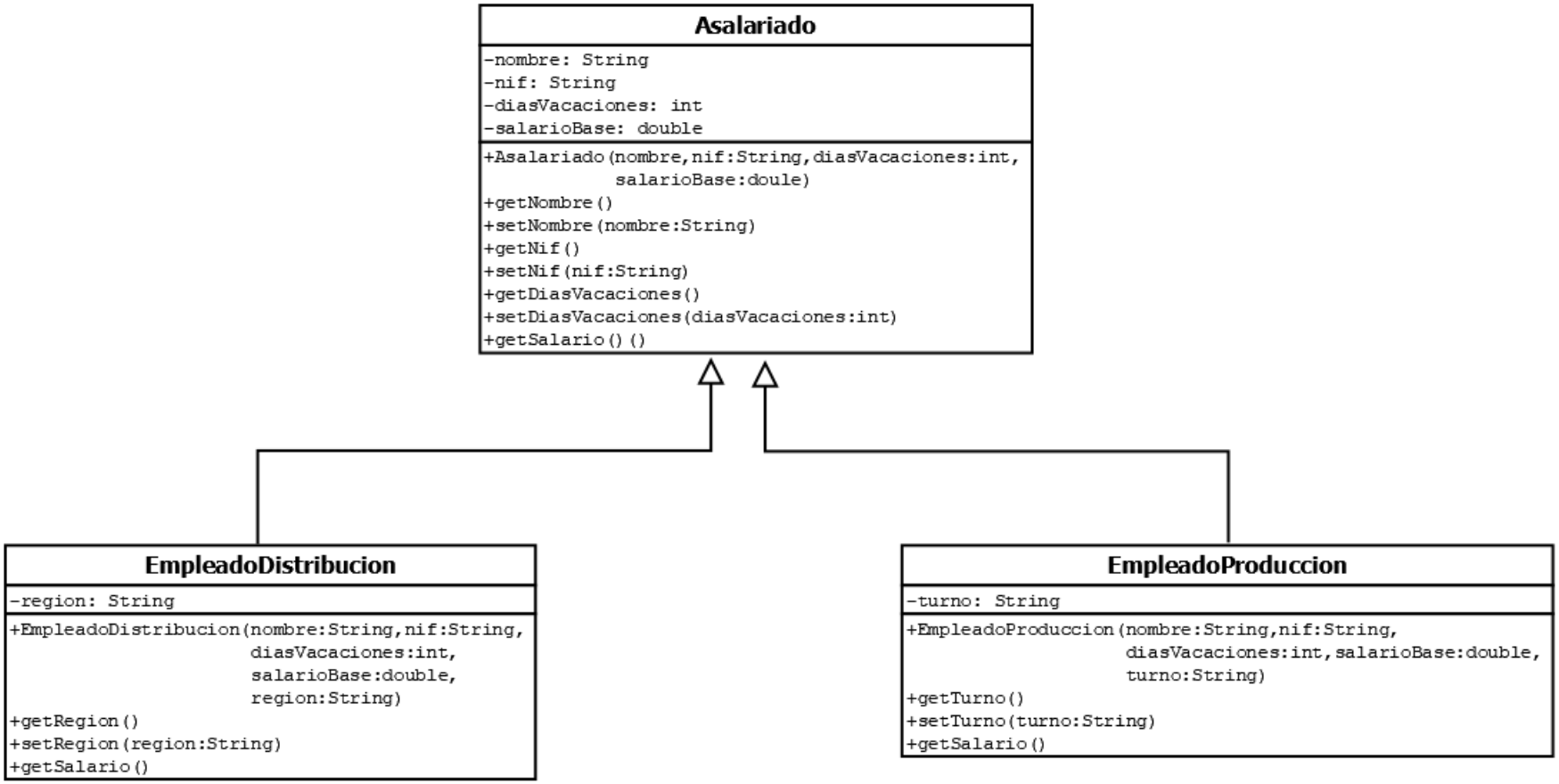
```
class Instrumento {
  public String tipo() { return "Instrumento"; }
  public String tocar() { return "Tocar "+tipo(); }
  public String afinar() { return "Afinar "+tipo(); }
}

class Guitarra extends Instrumento {
  public String tipo() { return "Guitarra"; }
  public String tocar() { return "Tocar "+tipo(); }
  public String afinar() { return "Afinar "+tipo(); }
}
```

Implementar Piano, Saxofón y Ukelele de forma equivalente

```
public class Musica {

    // No importa el tipo de Instrumento,
    // seguirá funcionando debido a Polimorfismo:
    static void afinar(Instrumento i) {
        // ...
        System.out.println(i.afinar());
    }

    //Llama al método estático afinar de esta clase que usa
    //el polimorfismo en el paso de parámetros
    static void afinarTodo(Instrumento[] e) {

        for(int i = 0; i < e.length; i++)
            afinar(e[i]);
    }


    public static void main(String[] args) {
        Instrumento[] orquesta = new Instrumento[4];
        int i = 0;

       orquesta[i++] = new Guitarra();
        orquesta[i++] = new Piano();
        orquesta[i++] = new Saxofon();
        orquesta[i++] = new Ukelele();
        afinarTodo(orquesta);
    }
}
```

Analizar la aplicación del polimorfismo en este código

Ejercicio II



- Crear los métodos estáticos siguientes en **main**:
  1. **polimorfismo()**:
     - Utilizando variables de instancia de tipo Asalariado, crear un objeto de cada clase hija y mostrar sus datos por pantalla.
     - *Reflexión*: A partir de cada instancia puedo acceder a unos métodos y a otros no, ¿por qué?
  2. **sin_polimorfismo()**:
     - Crear un objeto de cada clase (*la clase de la parte estática y de la parte dinámica deben ser el mismo*) y mostrar sus datos por pantalla.
     - *Reflexión*: Contrastar los métodos accesibles de las instancias respecto al método anterior.
  3. **polimorfismo_Arrays()**:
     - Crear un objeto de cada clase.
     - Crear un array de la superclase.
     - Almacenar los objetos creados en el array.
     - Recorrer el array y mostrar la información almacenada en cada objeto.
     - *Reflexión*: Todos los objetos almacenados en el array acceden a los mismos métodos, pero no todos tienen la misma implementación.
  4. **polimosfismo_Parametros()**:
     - Crear un objeto de cada clase sin usar polimorfismo.
     - Utilizar cada uno de los objetos creados como parámetro en un método llamado ***mostrarSalario()*** que muestra el salario de cada trabajador. La firma del método es ***public static void mostrarSalario(Asalariado asl).***
     - *Reflexión*: Analizar el paso por parámetros.

Casting / insteadof

- Los objetos, lo mismo que los tipos primitivos, permiten realizar la operación de **casting** siempre que los objetos pertenezcan a la misma rama de herencia.
- La mayor utilidad de los castings de objetos es permitir el acceso de una instancia a los métodos que le son propios y exclusivos.

```
Asalariado emplead  = new EmpleadoProduccion("Lurdes", "55333222L", 30, 2200, "tarde");
System.out.println(emplead.getNombre() + " tiene el turno de " + emplead.getTurno());

EmpleadoProduccion empP = (EmpleadoProduccion) emplead;
System.out.println(empP.getNombre() + " tiene el turno de " + empP.getTurno());
```

- Si no conozco previamente el tipo original de la instancia sobre la que tengo que hacer casting, deberé utilizar el operador **intanceof** para garantizar que la operación se pueda realizar.
- El operador **instanceof** nos indica si un objeto es de una clase concreta aunque esté almacenado en una variable de instancia de una superclase.

```
if (emplead instanceof EmpleadoProduccion) {
    EmpleadoProduccion empP = (EmpleadoProduccion) emplead;
    System.out.println(empP.getNombre() + " tiene el turno de " + empP.getTurno());
}else{
    System.out.println("No se puede hacer el casting");
}
```

Casting / insteadof. Ejemplo

```
public static void usoInstanceof() {
    Asalariado emplead1 = new Asalariado("Manuel Cortina", "12345678W", 28, 1200);
    EmpleadoProduccion emplead2 = new EmpleadoProduccion("Juan Mota", "55333222L", 30, 1200, "noche");
    EmpleadoDistribucion emplead3 = new EmpleadoDistribucion("Antonio Camino", "55333666P", 35, 1200,
"Granada");

    //Array de tipo de la superclase Asalariado
    Asalariado[] array_asal = new Asalariado[3];

    array_asal[0] = emplead1;
    array_asal[1] = emplead2;
    array_asal[2] = emplead3;

    for (int i = 0; i < 3; i++) {
        System.out.println("El sueldo del trabajador " + i + " es "   + array_asal[i].getSalario());
    }

    /*Si quiero acceder al turno de EmpeladoProudccion y a la zona de EmpleadoDistribucio
    for (int i = 0; i < 3; i++) {
        if (array_asal[i] instanceof EmpleadoProduccion) {
            EmpleadoProduccion empP = (EmpleadoProduccion) array_asal[i];
            System.out.println(empP.getNombre() + " tiene el turno de " + empP.getTurno());
        } else if (array_asal[i] instanceof EmpleadoDistribucion) {
            EmpleadoDistribucion empD = (EmpleadoDistribucion) array_asal[i];
            System.out.println(empD.getNombre() + " trabaja en la zona " + empD.getRegion());
        }
    }
}
```

Con las llamadas polimórficas no se puede llamar a los métodos propios de cada clase, a no ser que estén sobrescritos

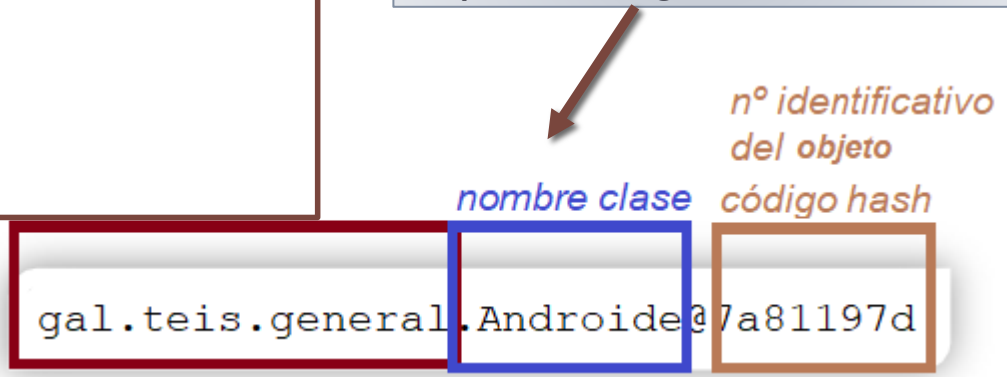
Si hago *casting* de los objetos del Array puedo convertir el objeto al tipo del objeto almacenado y así acceder a sus métodos propios

Método toSring() I

- El método **toSring()** es un método de la clase **Object**.
- Todos las clases heredan de **Object**, por lo tanto todas heredan en método **toString()**.
- Se puede invocar al método **toString()** desde cualquier variable de instancia, siempre que ésta no apunte a **null**.

```
public class Robot {
  public static void main(String args[]) {
         Androide droid = new Androide();
          System.out.println(droid.toString());
  }
}

class Androide {
  // Hereda el método toString()
  //de la clase Object.
}
```

El **String** que por defecto devuelve el método **toString()** está compuesto por los siguientes datos



Método toSring() II

- Existen otras formas de llamar al método **toString()** de manera implícita:
  - **String.valueOf(**variable_instancia**)**
  - **System.out.println(**variable_instancia**)**

```
class Androide {

    @Override
    public String toString() {
        String mensaje = "Este es un objeto "
                + "de la clase Androide";
        return mensaje;
    }
}
```

El método toString() se puede sobrescribir en cualquier clase

```
public class Robot {

    public static void main(String args[]) {

        Androide droid = new Androide();

        String uno = droid.toString();
        String dos = String.valueOf(droid);

        System.out.println(uno);
        System.out.println(dos);
        System.out.println(droid);
    }
}
```

```
Este es un objeto de la clase Androide
Este es un objeto de la clase Androide
Este es un objeto de la clase Androide
```

```
public class Robot {

    private String fecha;
    private String nombre;

    Robot(String nombre) {
        setFecha();
        this.nombre = nombre;
    }

    private void setFecha() {
        LocalDate date = LocalDate.now();
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd  MM yyyy");
        fecha = date.format(formatter);
    }

    public String getFecha() {
        return fecha;
    }

    @Override
    public String toString() {
        return "Robot creado el " + fecha + ", responde por " + nombre + ".";
    }
}
```

Ejemplo de uso de toString() en una clase

Comparación de objetos (==)

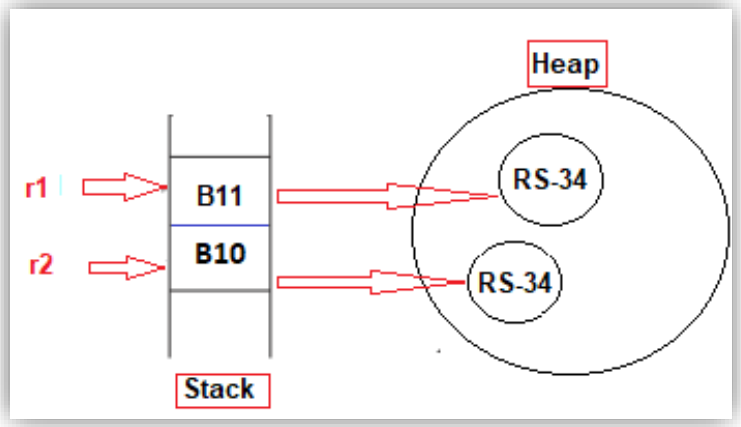
- **Cuando comparamos con ==**

  - **Tipos básicos** → El operador lógico == compara dos variables o constantes de tipos básicos y devuelve true si son iguales y false en caso contrario.

```
void mirarEdad (int edad){
    if(edad == 20){
        System.out.println("Tienes 20 años");
    }else{
        System.out.println("No tienes 20 años");
    }
}
```

  - **Objetos** → Si comparamos dos objetos se comparan las direcciones de memoria almacenadas en el **stack**, no se compara ninguna información que el objeto posea.

```
Robot r1= new Robot("RS-34");
Robot r2= new Robot("RS-34");

System.out.println(r1 == r2);
```



El resultado de la comparación es *false* pues las direcciones que almacenan en el *stack* son diferentes

Comparación de objetos (equals()/hashCode())

- Los parámetros para comparar dos objetos deben ser definidos por el programa, es decir, se puede determinar que dos objetos de tipos **Persona** con iguales cuando tienen el mismo nombre y los mismos apellidos, o bien, si tienen el mismo NIF.
- El método **equals()** es un método de **Object** que puede ser sobrescrito para adaptarlo a los requisitos de funcionamiento.
- El método **hashCode()** es un método de **Object** que puede ser sobrescrito para adaptarlo a los requisitos de funcionamiento. Devuelve un código hexadecimal único para cada objetos.
- Se debe cumplir que, si dos objetos con iguales según el método **equals**(), deben generar el mismo entero al llamar a **hashCode()**.

```java
public class Robot {

    private String fecha;
    private String nombre;

    Robot(String nombre) {
        setFecha();
        this.nombre = nombre;
    }

    private void setFecha() {
        LocalDate date = LocalDate.now();
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd  MM yyyy");
        fecha = date.format(formatter);
    }

    public String getFecha() {
        return fecha;
    }

    @Override
    public String toString() {
        return "Robot creado el " + fecha + ", responde por " + nombre + ".";
    }
}
```

```java
@Override
public int hashCode() {
    return nombre.hashCode() + fecha.hashCode();
}

@Override
public boolean equals(Object obj) {
    Robot r = (Robot) obj;
    return r.nombre.equals(this.nombre) &&
                    r.fecha.equals(this.fecha);
}
```

Saber más

Métodos a agregar a la clase si se desea que dos objetos de tipo Robot sean iguales cuando su nombre y la fecha de creación es la misma es el mismo

Implementar un proyecto para comprobar el funcionamiento de **equals**()