UD05.2.POO.Herencia
Apuntes

DAM1-Programación 2024-25

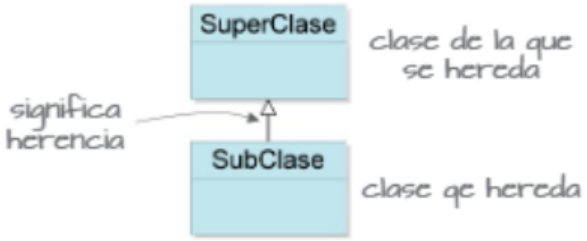
Otras fuentes:

- **Programación Java: Teoría**
  - Herencia en Java
  - Método abstracto
  - Polimorfismo. Concepto y ejemplo de Polimorfismo en Java
- **Universitat Politècnica de València - UPV**
  - La herencia en Java | | UPV
  - El polimorfismo en Java | | UPV
- **Relaciones entre clases: Asociación y Herencia:**
  - (Programación en Java: Fundamentos de programación y principios de diseño)
  - **Relaciones entre clases (diagramas de clases UML)**
  - Relación de ejercicios
- **draw.io**
- Creador gratuito de diagramas UML| Lucidchart
- Lucid para la Educación | Lucidchart

Herencia. Conceptos Básicos

La herencia es una de las grandes aportaciones de la POO y permite, igual que en la vida real, que las características pasen de padres a hijos. Cuando una clase hereda de otra, adquiere sus atributos y métodos visibles, permitiendo reutilizar el código y las funcionalidades, que se pueden ampliar o extender.

La clase de la que se hereda se denomina clase padre o **superclase**, y la clase que hereda es conocida como clase hija o **subclase**.



8.1. Subclase y superclase

Una subclase dispone de los miembros heredados de la superclase y, habitualmente, se amplía añadiéndole nuevos atributos y métodos. Esto aumenta su funcionalidad, a la vez que **evita la repetición innecesaria de código**. En la API, por ejemplo, la mayoría de las clases no se definen desde cero. Por el contrario, se construyen heredando de otras, lo que simplifica su desarrollo. En realidad, **todas las clases de Java heredan de la clase Object**, definida también en la API.

La forma de expresar cuál es la superclase de la que heredamos es mediante la palabra reservada **extends**, de la forma

```
class SubClase extends SuperClase (

}
```

Veamos un ejemplo: supongamos que disponemos de la clase **Persona** (nombre, edad y estatura) y necesitamos construir la clase **Empleado**. Un empleado, para nuestra aplicación, será una persona con un salario.

Vamos a definir Empleado heredando de Persona. Esto hará que adquiera todos sus miembros, que no es necesario escribir de nuevo. De momento añadiremos el atributo salario y un constructor.

```
class Persona {
    String nombre;
    byte edad;
    double estatura;
}

class Empleado extends Persona {
    double salario;

    Empleado (String nombre, byte edad, double estatura, double salario){

    }
}
```

Al crear un objeto de la clase Empleado disponemos de los atributos nombre, estatura y edad, además de los métodos que se hubieran definido en Persona y de los miembros propios salario y un constructor, añadidos en la definición de Empleado.

```
Empleado e = new Empleado ("Sancho", 25, 1.80, 1725.49);

System.out.println(e.nombre);  //muestra un atributo heredado
System.out.println(e.salario); //muestra un atributo propio
```

El mecanismo de la herencia puede continuar ampliando la biblioteca de clases a partir de las existentes. En nuestro ejemplo, podemos definir, a partir de **Empleado**, la clase **Jefe**, que no es más que un empleado con unas propiedades añadidas.

Existen lenguajes de programación, como C++, que permiten que una clase herede de más de una superclase, lo que se conoce como **herencia múltiple**. Java solo permite herencia simple, donde cada clase tiene como padre una única superclase, cosa que no impide que, a su vez, tenga varias clases hijas.

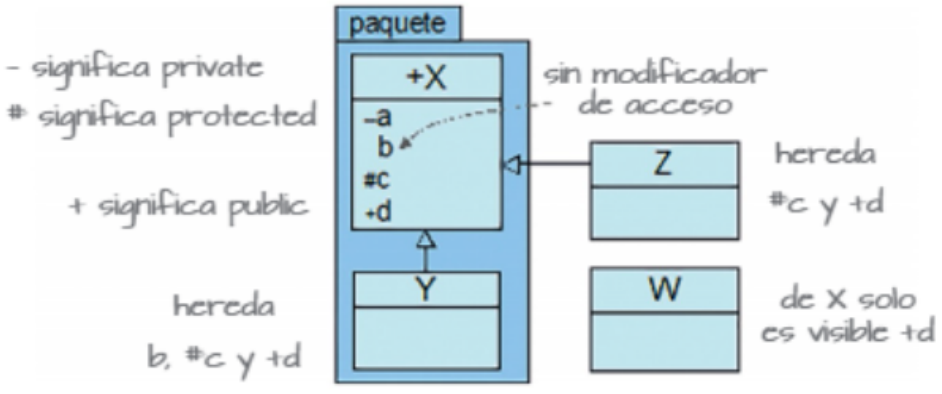
Los términos subclase y superclase son relativos. Una clase es subclase de otra si hereda de ella por medio de la palabra clave *extends* en su declaración. Automáticamente esta última es superclase de la primera. Una clase puede ser a la vez subclase de una clase y superclase de otras.

8.2. Modificador de acceso para herencia

Una subclase hereda todos los miembros de su superclase salvo los *private*, que no son accesibles directamente en la subclase. No obstante, se puede acceder a ellos indirectamente con un método no privado.

Con la herencia aparece un nuevo modificador de acceso, **`protected`** (que siginifica «protegido»), pensado para facilitar la herencia.

Los miembros *protegidos* serán siempre visibles para las clases que hereden, independientemente de si la superclase y la subclase son vecinas o externas, aunque en este último caso habrá que importar la superclase.



Visibilidad de un miembro `protected`.

Un miembro protected es visible también en las clases vecinas.

	Visible desde...			
	la propia clase	clases vecinas	subclases	clases externas
private	✓			
sin modificador	✓	✓		
protected	✓	✓	✓	
public	✓	✓	✓	✓

8.3. Redefinición de miembros heredados

Cuando una clase hereda de otra, en alguna ocasión puede ocurrir que interese modificar algún atributo o redefinir un método. Este mecanismo se conoce como **ocultación** para los atributos y **sustitución** u ***overriding*** para los métodos. Consiste en declarar un miembro con igual nombre que uno heredado, lo que hace que este se oculte si es un atributo o se sustituya si es un método por el nuevo.

Veamos cómo sustituir un método. Partimos de la superclase

```
class Persona {
    String nombre;
    byte edad;
    double estatura;

    void mostrarDatos() {
        System.out.println(nombre);
        System.out.println(edad);
        System.out.println(estatura);
    }
}
```

A continuación definimos una nueva clase:

```
class Empleado extends Persona (//Empleado hereda de Persona
    double salario; //atributo propio
}
```

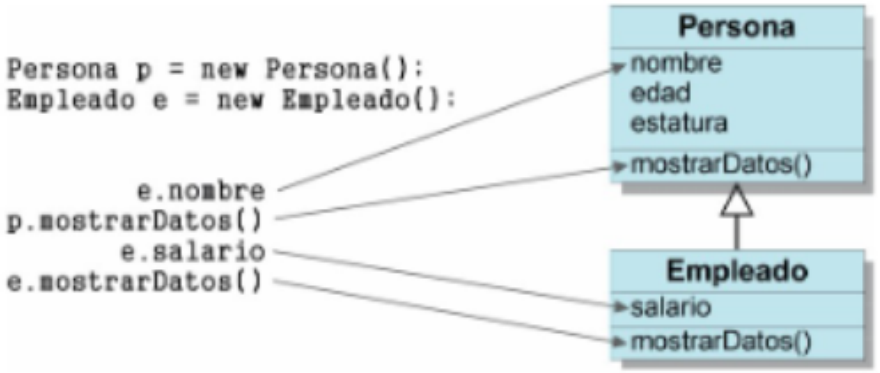
Nos encontramos que la clase `Empleado` dispone, heredado de `Persona`, del método `mostrarDatos()`, pero, en la práctica, este método no basta para mostrar la información de un empleado, ya que no muestra su salario. Una solución es redefinir el método en la clase Empleado.

Aunque es opcional, los métodos sustituidos en las subclases se suelen marcar con la anotación `@Override`, que indica que el método es una sustitución u *overriding* de un método de la superclase.

Para sobreescribir un método de la superclase, es imprescindible que el que lo sustituye en la subclase tenga el **mismo nombre, la misma lista de parámetros de entrada y el mismo tipo devuelto**. En otro caso, si la lista de parámetros no es la misma, se trataría de una sobrecarga, y si el valor devuelto es distinto, podría producir un error de compilación.

```
class Empleado extends Persona {
    double salario;

    @Override //significa: sustituye un método de la superclase
    void mostrarDatos() {
        System.out.println(nombre);
        System.out.println(edad);
        System.out.println(estatura);
        System.out.println(salario);
    }
}
```



Uso de miembros, heredados y propios

Veamos ahora un ejemplo de **ocultación**. Supongamos que la estatura de un empleado definida como una longitud no es un dato relevante para la empresa, pero sí es interesante conocer la estatura como talla del uniforme. Redefiniríamos el atributo como un String que contenga la talla del uniforme: «XXL», «XL», «L», etcétera.

```
class Empleado extends Persona (
    String estatura; //oculta a: la estatura de tipo byte

}
```

8.3.1. super y super()

- `super` se utiliza para hacer referencia a miembros de la superclase.
- `super()` para hacer referencia constructores de la superclase.

Del mismo modo que la palabra reservada **this** se utiliza para indicar la propia clase, disponemos de **super** para hacer referencia a la superclase de aquella donde se usa.

Consideremos las siguientes clases:

```
class SuperClase {
    int a;
    int b;

    void mostrarDatos(){
    }
}

class SubClase extends SuperClase (
    String b;

    void mostrarDatos() {
    }
}
```

Como puede apreciarse, en `Subclase` se han redefinido el atributo `b` y el método `mostrarDatos ()`. Cada vez que se escriba `b` en el código de `SubClase` estaremos utilizando un String, pero si deseamos utilizar el atributo `b`, de tipo entero, de `Superclase`, escribiremos **`super.b`**.

Del mismo modo, para invocar el método mostrarDatos() de Superclase desde el código de Subclase escribiremos **`super.mostrarDatos()`**. Para el caso de Persona y Empleado, podríamos poner:

```
public class Persona {
    String nombre:
    byte edad,
    double estatura;

    void mostrarDatos() {
        System.out.println(nombre);
        System.out.println(edad);
        System.out.println(estatura);
    }
}

class Empleado extends Persona {
    double salario;

    @Override
    void mostrarDatos() {
```

```
        super.mostrarDatos(); // método de superclase, muestra atributos de Persona
        System.out.println(salario); // muestra el atributo añadido en Empleado
    }
}
```

Algo análogo ocurre con los constructores. **El método `super()` invoca un constructor de la superclase**. Desde el constructor de la subclase, podemos invocar uno de la superclase para inicializar los atributos heredados de ella. En nuestro ejemplo, quedaría:

```
public class Persona {
    String nombre:
    byte edad,
    double estatura:

    Persona (String nombre, byte edad, double estatura) {
        this.nombre = nombre;
        this.edad = edad;
        this.estatura = estatura;
    }
    //…
}

class Empleado extends Persona {
    double salario;

    Empleado (String nombre, byte edad, double estatura, double salario) {
        super (nombre, edad, estatura);  //constructor de Persona
        this.salario = salario;    //atributo propio de Empleado
    }
}
```

En caso de que el **constructor de la superclase** esté **sobrecargado**, podemos variar los parámetros de entrada de super() en número o tipo para hacerla coincidir con la versión que nos interese del constructor de la superclase.

Una restricción de **super()** es que, si lo utilizamos, **tiene que ser la primera instrucción** que aparezca en la implementación de un constructor.

Las subclases no heredan los **atributos privados**. Sin embargo, *están ahí y son accesibles indirectamente a través de métodos públicos heredados*. Además de esto, **deben ser inicializados** al crear un objeto de la subclase. Por ejemplo, si Persona tuviera el atributo privado nacionalidad, este tendría que ser inicializado de una forma u otra al crear un objeto de la clase Empleado, aunque esta no herede el atributo. Normalmente, nacionalidad aparecerá en la lista de parámetros del constructor de Persona y, en consecuencia, en el método super() cuando lo invoquemos desde el constructor de Empleado.

8.3.2. Selección dinámica de métodos

Cuando definimos una clase como subclase de otra, **los objetos de la subclase son también objetos de la superclase**. Por ejemplo, un objeto Empleado será, al mismo tiempo, un objeto de la clase Persona, ya que posee todos los miembros de Persona, además de otros específicos de Empleado. Esto no debe extrañar; ocurre lo mismo en el

mundo real: todo empleado es una persona. Por tanto, podemos referenciar un objeto Empleado usando una variable Persona. Por ejemplo

```
Empleado e new Empleado();
Persona p = e;
```

Al referenciar un objeto de una subclase, por ejemplo Empleado, **con una variable de la superclase**, por ejemplo de tipo Persona, **solo serán visibles los miembros**, tanto atributos como métodos, **definidos en la superclase**, es decir en la clase Persona.

Cuando hay ocultación de atributos, los atributos accesibles son los definidos en la clase de la variable. Por tanto, si usamos la variable de tipo Persona referenciando un objeto Empleado, no se produce la ocultación.

```
p.estatura  //atributo de Persona de tipo double
```

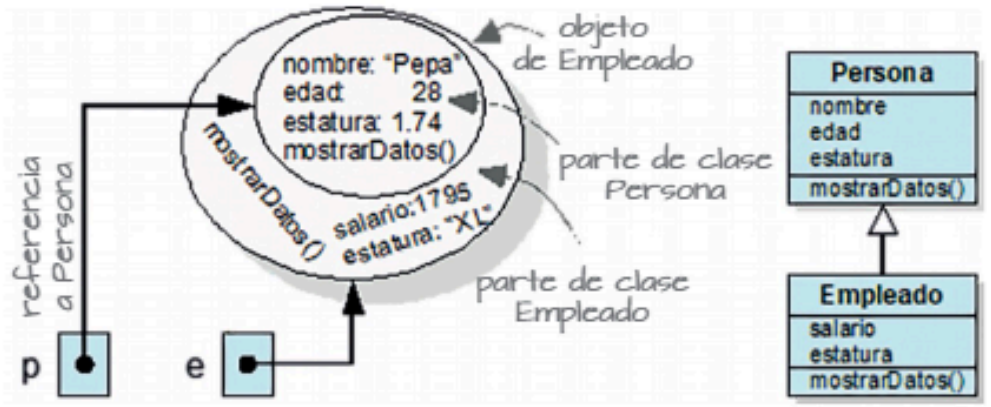


Figura 8.4. *Objeto de la clase Empleado.*

Si hubiéramos usado `e.estatura`, se estaría accediendo al atributos de `Empleado` de tipo `String`.

Pero, en cambio, con los métodos ocurre lo contrario. Se ejecuta la versión del objeto referenciado, es decir, de la subclase `Empleado`. Por tanto, sí funciona el overriding.

```
p.mostrarDatos(); //método de Empleado
```

En caso de usar `e.mostrarDatos()` se estaría ejecutando el mismo método. Esto proporciona una de las herramientas más potentes de que dispone Java para usar el polimorfismo: la selección de métodos en tiempo de ejecución. Por ejemplo, supongamos que una tercera clase `Cliente` hereda de `Persona`.

```
class Cliente extends Persona {
     ...
     @Override
     void mostrarDatos() {
             ...
     }
}
```

Si creamos una variable de tipo `Persona`, con ella podemos referenciar tanto objetos de clase `Empleado` como `Cliente` o `Persona`. Para todos ellos disponemos del método `mostrarDatos()`, pero se ejecutará una u otra versión, según el objeto referenciado, que puede cambiar en tiempo de ejecución.

```
Persona p;
p = new Persona();
p.mostarDatos(); //se ejecuta el método de Persona
p = new Empleado();
p.mostarDatos(); //se ejecuta el método de Empleado
p = new Cliente();
p.mostarDatos(); //se ejecuta el método de Cliente
```

Así, la misma línea de código, `p.mostrarDatos()`, ejecutará métodos distintos, según el tipo de objeto referenciado. Pero no debemos olvidar que, con una variable `Persona`, solo podemos acceder a métodos definidos en dicha clase.

Se llama *selección dinámica de métodos* al proceso por el cual se ejecuta un método u otro según la clase del método referenciado.

```
// Ejemplo de selección dinámica de métodos.

Persona[] p = new Persona[5];

p[0] = new Persona("Marta");
p[1] = new Empleado("Pepe", 26, 1.8, 25000);
p[2] = new Estudiante("Pablo", "DAM");
p[3] = new Empleado("Sandra", 29, 1.7, 34000);
p[4] = new Estudiante("Juan", "ASIR");

// En el bucle, Java "selecciona dinámicamente" el método más
for(int i = 0; i < 5; i++){
    p[i].mostrarDatos();
}
```

Crea las clases Empleado y Estudiante, subclases de Persona.

Empleado tendrá el atributo **salario** y los métodos

- constructor que permita indicar el salario.
- **toString()**, que imprimirá el nombre y el salario.
- **subirSalario(double porcentaje)**: incrementará el salario en el porcentaje indicado.

Estudiante tendrá los atributos, **nota1av**, **nota2av** y **nota3v** y los métodos:

- constructor que permita indicar y validar las notas.
- **toString()** que imprimirá el nombre y las notas de cada trimestre.
- **notaMedia()** que devolverá la media de las 3 notas.

Ejecuta el siguiente código para probar la selección dinámica de métodos (sustituyendo los puntos suspensivos por los valores adecuados para cada constructor):

```
Persona[] gente = new Persona[5];

gente[0] = new Persona(...);
gente[1] = new Empleado(...);
gente[2] = new Estudiante(...);

for(int i = 0; i < gente.length; i++){
     System.out.println(gente[i]);
}
```

La clase Object

Object (Java SE 19 & JDK 19 [build 1])

La clase `Object` del paquete `java.lang` es una clase especial de la que heredan, directa o indirectamente, todas las clases de Java. Es la **superclase** por excelencia, ya que se sitúa en la cúspide de la estructura de herencias entre clases.

Todas las clases que componen la API descienden de la clase `Object`. Incluso cualquier clase que implementemos nosotros hereda de `Object`. Esta herencia se realiza por defecto, sin necesidad de especificar nada. Por ejemplo, la definición de la clase `Persona`

```
class Persona {

}
```

es en realidad, equivalente a:

```
class Persona extends Object {

}
```

Y cualquier clase que herede de `Persona` está heredando, a su vez, de `Object`.

¿Cuál es el objetivo de que todas las clases hereden de `Object`? Haciendo esto se consigue:

- Que todas las clases implementen un conjunto de métodos en `Object` solo se han definido métodos que son de uso universal en Java, como realizar comparaciones entre objetos, clonarlos o representar un objeto como una cadena. La función de estos métodos es ser reimplementados a la medida de cada clase.
- Poder referenciar cualquier objeto, de cualquier tipo, mediante una variable de tipo `Object`.

Podemos ver los métodos de `Object` que ha heredado una `Persona` escribiendo en el IDE una variable de tipo `Persona` seguida de un punto (.). Entonces se desplegarán todos los atributos y métodos disponibles: los propios más los heredados de `Object`.

8.4.1. Método toString()

Este método está pensado para que devuelva una cadena que represente al objeto que lo invoca con toda la información que interese mostrar.

Tiene el prototipo

```
public String toString()
```

Su implementación en la clase `Object` consiste en devolver el **nombre cualificado de la clase** a la que pertenece el objeto, seguida de una arroba (**@**) junto a la **referencia** del objeto. Para un objeto Persona devuelve algo similar a:

```
"paquete. Persona@2al 39a55"
```

Esta implementación por defecto no es útil para representar la mayoría de los objetos, por lo que nos vemos obligados a realizar un *overriding* de `toString()` en cada clase, que es donde se encuentra la información que queremos representar.

Por ejemplo, podemos reimplementar `toString()` en `Persona` para representar una personacon su nombre junto a la edad, omitiendo la estatura.

```
class Persona {
    //...
    @Override
    public String toString() {
        String cad;
        cad "Persona: " + nombre + " (" + edad + ")";
        return cad
    }
}
```

Debe declararse `public`, igual que en la clase `Object`, ya que **todo método que sustituye a otro tiene que tener, al menos, el mismo nivel de acceso.**

Ahora podemos mostrar por consola la información de un objeto `Persona`.

```
Persona = new Persona("Claudia", 8, 1.20);

System.out.println(p.toString());
```

En realidad, System.out.println() invoca por defecto el método toString(). Por tanto, solo será necesario escribir

```
System.out.println(p); //equivale a System.out.println(p.toString());
```

8.4.2. Método equals()

Compara dos objetos y decide si son iguales, devolviendo `true` en caso afirmativo y `false` en caso contrario. Su prototipo en la clase `Object` es:

```
public boolean equals (Object otro)
```

El operador es útil para comparar tipos primitivos, pero no sirve para comparar objetos, ya que en este caso compara sus referencias, sin fijarse en su contenido. Por ejemplo,

```
Persona a = new Persona("Claudia", 8, 1.20);
Persona b = new persona("Claudia", 8, 1.20);
System.out.println(a == b); //false
```

El resultado es `false` porque la comparación se hace atendiendo a las referencias de los objetos, que son distintas.

El prototipo de `equals()` tiene un parámetro de entrada de tipo `Object` para poder comparar objetos de cualquier clase. Este prototipo debe mantenerse al hacer overriding en cualquier subclase —de lo contrario no sería overriding, sino sobrecarga—. Pero, para acceder a los atributos del objeto pasado como parámetro, tenemos que informar al compilador de que, en realidad, es un objeto `Persona`. Esto se consigue por medio de un cast, como veremos a continuación.

Vamos a reimplementar `equals()` para comparar objetos de la clase `Persona`. Lo primero es decidir qué significa que dos personas sean iguales. Para este ejemplo, vamos a considerar dos personas iguales si tienen el mismo nombre y la misma edad.

```
@Override
public boolean equals(Object otro) { //compara this con otro
     Persona otraPersona = (Persona) otro; //este cast se explica más abajo
     boolean iguales;
     if (this.nombre.equals(otraPersona.nombre) && this.edad == otraPersona.edad) {
             iguales = true;
     } else {
             iguales = false;
     }
     return iguales;
}
```

Nota técnica

En la práctica, a la hora de comparar, se suelen utilizar atributos que identifiquen de forma única a cada objeto, como el DNI, el número de socio de una biblioteca, etcétera.

El cast siempre es necesario porque el prototipo de `equals()` tiene que ser el mismo que en la clase `Object`, donde el parámetro de entrada es de tipo `Object`. Pero para acceder a los atributos `nombre` y `edad` de la clase `Persona` necesitamos que la variable `otro` sea de tipo `Persona`. Esto nos obliga a realizar un cast en la asignación. Es una conversión de estrechamiento, que podemos hacer porque sabemos que el objeto pasado como parámetro es, en realidad, de la clase `Persona`, aunque esté referenciado con una variable de tipo `Object`.

Por otra parte, en la condición de la estructura `if`, hemos invocado la implementación de `equals()` de la clase `String` para comparar los nombres, ya que son cadenas. Pero hemos utilizado `==` para comparar la edad, ya que es de un tipo entero primitivo. Ahora podemos comparar

```
Persona a = new Persona("Claudia", 8, 1.20);
Persona b = new persona("Claudia", 8, 0.0);
Persona c = new Persona("Pepe", 24, 1.89);
System.out.println(a.equals(b)); //true
System.out.println(a.equals(c)); //false
```

Aquí `equals()` compara los atributos `nombre` y `edad`, no referencias. Obsérvese que la estatura no influye en el resultado de la primera comparación.

Recuerda

Los valores de tipo Double e Integer no se pueden comparar con el operador ==, aunque a veces funcione, ya que son objetos, no valores primitivos. Para ellos debe usarse el método equals(). Por otra parte, los datos de tipo `double` primitivo tampoco se deben comparar con `==` debido a problemas de precisión interna del ordenador. Por ejemplo,

```
System.out.println(5.6+5.8 == 5.7*2);
```

muestra `false` cuando debería mostrar `true`. Para cálculos de mayor precisión disponemos de la clase `BigDecimal`, que está definida en el paquete `java.math` y hereda de `Number`. Con ella podemos elegir la precisión (número de dígitos significativos) y el modo de redondeo que deseemos.

La mayoría de las clases de la API tienen su propia implementación de `equals()`, que permite comparar sus objetos entre sí. Sin embargo, las tablas, a pesar de ser objetos, no traen implementado el método `equal()`. Si queremos comparar dos tablas para ver si son iguales, tendremos que comparar elemento a elemento. Otra opción sería utilizar el método estático `equals()` de la clase `Arrays`, que devuelve `true` si las dos tablas tienen los mismos elementos en el mismo orden. Veamos un ejemplo:

```
int t1[] = {1, 2, 3, 4};
int t2[] = {1, 2, 3, 4};
int t3[] = {1, 4, 3, 2};
boolean iguales = Arrays.equals(t1, t2); //devuelve true
boolean iguales = Arrays.equals(t1, t3); //devuelve false
```

8.4.3. Método getClass()

Es común usar una variable `Object` para referenciar un objeto de cualquier clase que, como sabemos, siempre será una subclase de `Object`. A veces necesitamos saber cuál es esa clase.

Para eso está el método `getClass()`, definido en `Object` y heredado por todas las clases. Este método, invocado por un objeto cualquiera, devuelve su clase que, a su vez, es un objeto de la clase `Class`. Todas las clases de Java, incluidas `Object` y la propia `Class`, son objetos de la clase `Class`. Por ejemplo, si escribimos

```
Object a = "Luis";
System.out.println(a.getClass());
```

obtendremos por pantalla:

```
class java.lang.String
```

Es decir, la clase cuyo nombre cualificado es: `java.lang.String`. Podríamos haber puesto:

```
System.out.println(a.getClass().getName());
```

para obtener directamente el nombre: `java.lang.String`.

El método `getName()`, de la clase `Class`, devuelve el nombre cualificado de la clase invocante.

Por otra parte, a partir de una clase, podemos obtener su superclase por medio del método `getSuperclass()` de la clase `Class`. Por ejemplo,

```
Object b = Double.valueOf(3.5); //un objeto Double
Class clase = b.getClass(); //la clase de b: Double
Class superclase = clase.getSuperclass(); //superclase: class java.lang.Number
System.out.println(superclase.getName()); //nombre: java.lang.Number
```

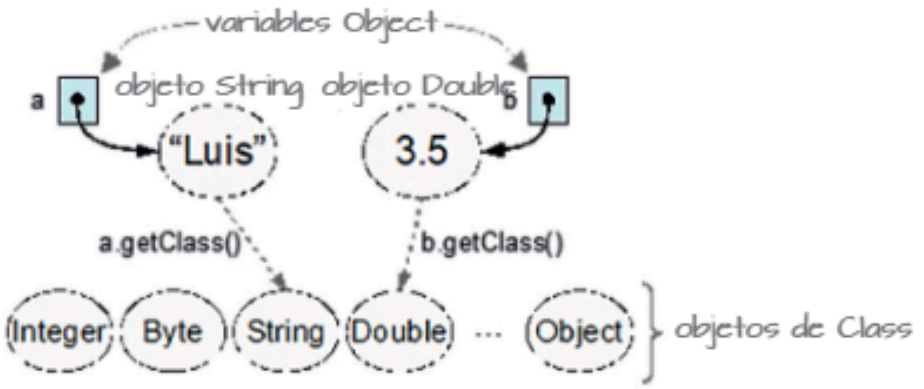


Figura 8.5. *Todas las clases existentes son objetos de la clase Class.*

El operador `instanceof` en Java se utiliza para verificar si un objeto es una instancia de una clase específica, una subclase de esa clase, o una clase que implementa una interfaz en particular. Devuelve un valor booleano: `true` si el objeto es una instancia de la clase o interfaz especificada, y `false` en caso contrario.

Sintaxis

```
objeto instanceof Clase
```

- objeto: Es la instancia del objeto que se desea verificar.
- Clase: Es la clase o interfaz con la que se desea comparar el objeto.

Java instanceof Operator | Baeldung

E0801. Diseñar la clase `Hora`, que representa un instante de tiempo compuesto por la hora (de 0 a 23) y los minutos. Dispone de los métodos:

- `Hora (hora, minuto)`, que construye un objeto con los datos pasados como parámetros.
- `void inc()`, que incrementa la hora en un minuto.
- `boolean setMinuto(valor)`, que asigna un valor, si es válido, a los minutos. Devuelve true o false según haya sido posible modificar los minutos o no.
- `boolean setHora(valor)`, que asigna un valor, si está comprendido entre 0 y 23, a la hora. Devuelve `true` o `false` según haya sido posible cambiar la hora o no.

- `String toString()`, que devuelve un `String` con la representación de la hora.

E0802. A partir de la clase `Hora` implementar la clase `HoraExacta`, que incluye en la hora el `segundo`. Además de los métodos heredados de `Hora`, dispondrá de:

- `HoraExacta(hora, minuto, segundo)`, que construye un objeto con los datos pasados como parámetros.
- `setSegundo(valor)`, que asigna, si está comprendido entre 0 y 59, el valor indicado a los segundos.
- `void inc()`, que incrementa la hora en un segundo.

E0803. Añadir a la clase `HoraExacta` un método que compare si dos horas (la invocante y otra pasada como parámetro de entrada al método) son iguales o distintas.

Clases abstractas

Las **clases abstractas** son clases que tienen al menos un método sin implementar, denominado a su vez, método abstracto.

En la jerarquía de herencia de clases, cuanto más abajo, más específica y particular es la implementación de los métodos. Asimismo, cuanto más arriba, más general.

Hay métodos que no podemos implementar en una clase determinada por falta de datos, pero sí en sus subclases, donde se han añadido los atributos necesarios. La idea es implementarlos vacíos, solo con el prototipo, en la superclase, y hacer *overriding* en las subclases, donde ya disponemos de la información necesaria para implementar los detalles.

Un método definido en una clase, pero cuya implementación se delega en las subclases, se conoce como abstracto. Para declarar un método abstracto se le antepone el modificador **abstract** y se declara el prototipo, sin escribir el cuerpo de la función. Por ejemplo, para declarar un método abstracto que muestra información del objeto escribiremos:

```
abstract void mostrarDatos();
```

Las subclases deberán implementar el método mostrarDatos(), cada una con las parti-cularidades específicas de la clase, que no se conocen al nivel de la superclase.

Toda clase que tiene un método abstracto debe ser declarada, a su vez, **abstract**.

Las clases abstractas no son instanciables, es decir, no se pueden crear objetos de esa clase. Las clases abstractas **existen para ser heredadas por otras**, y no para ser instanciadas. Si una clase hereda de una abstracta, pero deja alguno de sus métodos abstractos sin implementar, será también abstracta. Sin embargo, una clase abstracta puede tener algún método implementado y algunos atributos definidos, que heredarán las subclases, pudiendo hacer sustitución u ocultación de ellos.

Vamos a ver todo esto por medio de un ejemplo. Definimos una clase abstracta A, donde declaramos e inicializamos una variable `x` entera. Asimismo, definimos e implementamos un método `metodo1()`. Tanto la variable como el método serán heredados tal cual por las subclases de A. Por otra parte, declaramos un método abstracto `metodo2()`

```
//clase abstracta, ya que uno de sus métodos, metodo2(), es abstracto
abstract class A {
    int x = 1;

    void metodo1() { //método implementado y heredados por las subclasen
        System.out.println("método1 definido en A");
    }
```

```
    abstract void metodo2(); //método abstracto para ser implementado por
las subclases
}
```

A continuación, definimos las clases B y C que heredan de A, e implementan el método `metodo2()`. Ambas clases heredan tanto la variable `x` como el método `metodo1()`, con su implementación.

```
class B extends A {
    //atributos y métodos propios de B

    void metodo2() {
        System.out.println("método2 implementado en B");
    }
}

class C extends A {
    //atributos y métodos propios de C
    void metodo2() {
        System.out.println("método2 implementado en C");
    }
}
```

Tanto B como C han heredado `metodo1()` tal como está implementado en A, pero cada una tiene su propia implementación de `metodo2()`.

En el programa principal creamos sendos objetos de clase B y C (de la clase A no es posible, puesto que es abstracta) y ejecutamos los métodos `metodo1()` y `metodo2()` de cada uno de los dos objetos.

```
B b = new B();
C c = new C();

System.out.println("Valor de x en la clase B:” + b.x); //heredado de A

b.metodo1(); //método heredado directamento de A
b.metodo2(); //implementación del método2() abstracto de A

c.metodo1(); //método heredado de directamente A
c.metodo2(); //implementación del método2() abstracto de A
```

El resultado mostrado por consola será:

```
Valor de a en la clase B: 1
método1 definido en A
método2 definido en B
método1 definido en A
```

```
método2 definido en C
```

Aunque no se puedan crear objetos de clase A sí pueden existir variables de dicha clase. Una variable de clase A puede hacer referencia a cualquier objeto de una subclase de A que no sea abstracta, como B o C. Al código anterior le podemos añadir las siguientes líneas:

```
A a = b;
a.metodo2();
```

Como el objeto referenciado es de clase B, la versión de `metodo2()` ejecutada será la implementada en B. Si ahora asignamos a a la referencia de e de tipo C, se ejecutará la versión de `metodo2()` implementada en C.

```
a = c;
a.metodo2();
```

Como vemos, con la misma línea de código `a.metodo2()`, se ejecutan implementaciones distintas, es decir, código diferente. Esto es otro ejemplo de **selección dinámica de métodos**.

E0804. Crear la clase abstracta `Instrumento`, que almacena en una tabla las notas musicales de una melodía (dentro de una misma octava). El método `add()` añade nuevas notas musicales. La clase también dispone del método abstracto `interpretar()` que, en cada subclase que herede de `Instrumento`, mostrará por consola las notas musicales según las interprete. Utilizar enumerados para definir las notas musicales.

E0805. Crear la clase `Piano` heredando de la clase abstracta `Instrumento`. Crea las clases `Guitarra`, `Trompeta`, `Trompa`, `Flauta`, etc…

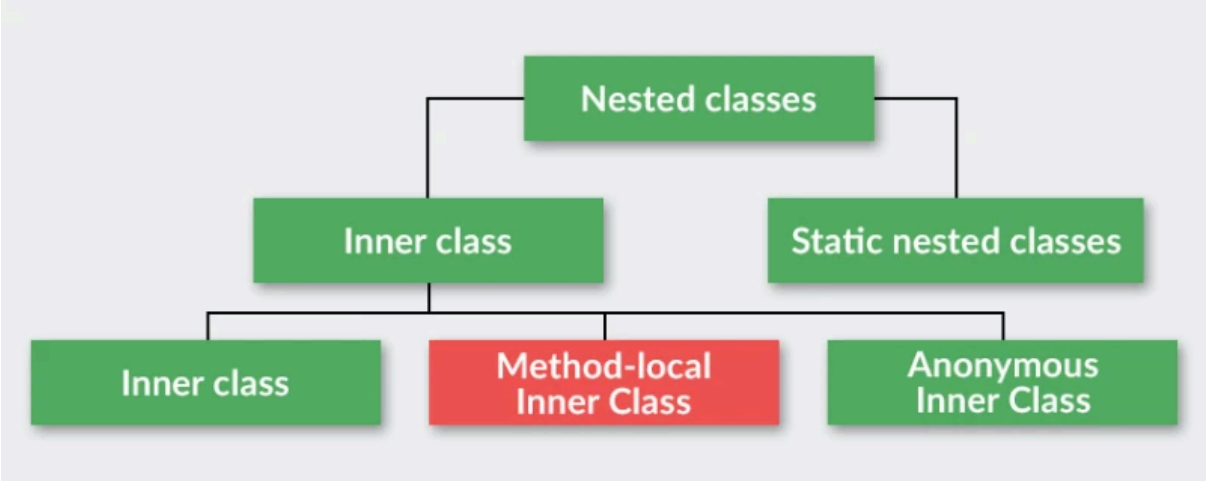
Implementa un programa principal que cree una orquesta con una colección de Instrumentos, cada uno con su melodía y que la interpreten conjuntamente.

Anexo. Clases Internas

- **Píldoras informáticas**
  - Curso Java. Interfaces y clases internas. Clases internas I. Vídeo 53
  - Curso Java. Interfaces y clases internas. Clases internas II. Vídeo 54
- Clases internas

Hay cuatro tipos de clases internas:

1. Clases anidadas estáticas (*static nested classes*). Sólo pueden ser creadas dentro de otra clase al máximo nivel. Se definen utilizando la palabra *static*.
2. **Clases internas miembro** (*inner nested classes*)**.** O simplemente clases internas miembro. Se definen en la clase contenedora sin la palabra *static*. Se utilizan para crear objetos internos a una clase (el motor de un coche por ejemplo), de modo que esos objetos pasan a ser atributos de la clase.
3. **Clases internas locales.** Se declaran dentro de un bloque de código, normalmente en un método, aunque también se pueden crear en un inicializador static o de objeto. Normalmente se usan cuando el método intenta solucionar un problema complicado y necesita apoyarse en una clase, pero no se necesita que esta clase esté disponible fuera. Son por tanto clases que quedan fuera del diseño. Las clases internas a un método sólo pueden ser instanciadas en el interior de dicho método
4. **Clases anónimas.** Similares a las clases internas locales, pero sin nombre. Sólo se puede crear un único objeto. Se utilizan con mucha frecuencia para definir clases y objetos que gestionen los eventos de los distintos componentes de la interface de usuario.



UD05.A.Clases Internas