PROGRAMACIÓN

Desarrollo de Aplicaciones Multiplataforma

Manual del módulo

UD 7 - HERENCIA

Ander Gonzalez Ortiz

7.1. INTRODUCCIÓN

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 supere/ase, y la clase que hereda es conocida como clase hija o subclase. El siguiente diagrama de clases representa el concepto de herencia.

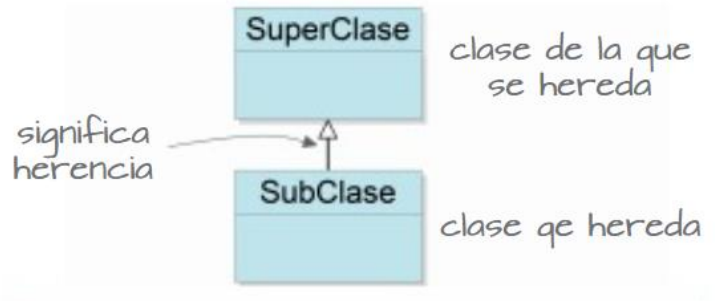

Figura 1. Herencia entre clases.

7.2. 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

```
public 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 (nombre, edad y estatura) 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.

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

public class Empleado extends Persona {
```

```
     double salario;

     public 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. Por ejemplo,

```
Empleado e = new Empleado("Sancho”, 25, 1.80, 1725.49);
Sytsem.out.println(e.nombre); //muestra un atributo heredado
System.ou.tprintln(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.

7.3. MODIFICADOR DE ACCESO PARA HERENCIA

Con la aparición de la herencia podemos plantearnos algunas cuestiones: ¿se heredan todos los miembros de una clase?; si no es así, ¿cuáles son los miembros que se heredan? Se heredan todos salvo los `private`, que no son accesibles directamente en la subclase. No obstante, se puede acceder a ellos indirectamente con un método no privado.

Por otra parte, junto con los tipos de visibilidad citados, para que un miembro sea accesible desde una subclase, con el fin de obtener una mayor flexibilidad, podemos hacer uso de un nuevo modificador de acceso, `protected` (que significa “protegido”), pensado para facilitar la herencia.

Funciona de forma muy similar a la visibilidad por defecto, con la diferencia de que 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.

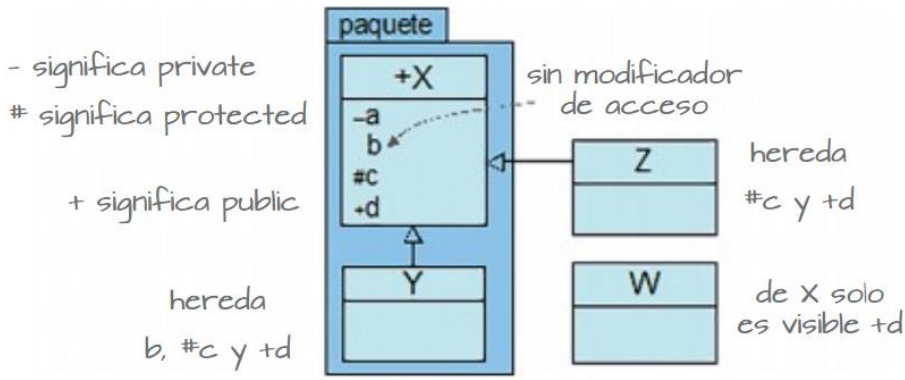

Figura 2. Visibilidad de un miembro protected.

En resumen, un miembro `protected` es visible en las clases vecinas, no es visible para las clases externas, pero siempre es visible, independientemente del paquete al que pertenezca, desde una clase hija.

La Tabla 8.1 muestra la visibilidad de un miembro `protected` junto al resto de los modificadores.

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

Veamos cómo se define la clase x utilizada en la Figura 2:

```
public class X {
     prívate int a; //invisible fuera de la clase
     int b; //visibildiad por defecto visible en el paquete
     protected int c; //visible en el paquete y para
     //las subclases (aunque sean externas)
     public int d; //visibilidad total
}
```

El atributo `a` es invisible desde fuera de la clase (aunque visible indirectamente desde una subclase); el atributo `b` es visible solo desde el mismo paquete, es decir, clases vecinas; `c` es accesible desde el mismo paquete y desde las subclases, y por último `d` es visible desde cualquier lugar, incluso para clases externas previa importación.

7.4. REDEFINICIÓN DE MIEMBROS HEREDADOS

Cuando una clase hereda de otra, en alguna ocasión puede ocurrir que interese modificar el tipo de 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.

Los miembros de una superclase se pueden redefinir en una subclase. Cuando se trata de un atributo, se habla de *ocultación*. Si es un método, se llama *sustitución* u *overriding*.

Veamos cómo sustituir un método. Partimos de la superclase

```
public class Persona {
    String nombre;
    byte edad;
    double estatura;
    public void mostrarDatos() {
```

```
        System.out.println(nmobre);
        System.out.println(edad);
        System.out.println(estatura);
    }
}
```

A continuación, definimos una nueva clase:

```
public 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 hacer *overriding* de un método de la superclase, es imprescindible que el que lo sustituye en la subclase tenga el mismo nombre y la misma lista de parámetros de entrada (el tipo devuelto deberá ser también el mismo; en caso contrario, se producirá un error de compilación).

Veamos cómo redefinir el método `mostrarDatos()` de la clase `Persona` en la subclase `Empleado`:

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

El método `mostrarDatos()` definido en `Empleado` sustituye al método, con el mismo nombre y los mismos parámetros, de `Persona`. Si la lista de parámetros no es la misma, no hay *overriding*. Estaríamos haciendo una sobrecarga del método `mostrarDatos()`. La Figura 3 muestra un ejemplo de qué miembros se usan.

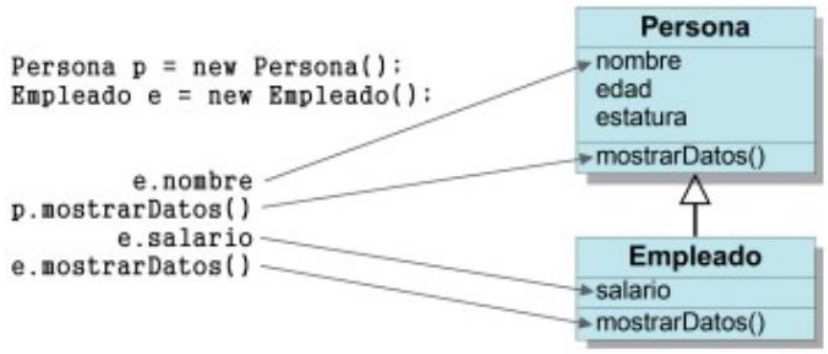

Figura 3. Uso de miembros, heredados o propios.

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 interesan te conocer la estatura como talla del uniforme. Redefiniríamos el atributo como un `String` que contenga la talla del uniforme: «XXL», «XL», «L», etcétera.

```
public class Empleadoextends Persona {
    String estatura;//oculta a: la estatura de tipo byte
}
```

El código de la Figura 3 muestra qué miembro es el que se utiliza en cada caso. De todas formas, el uso de la ocultación de atributos se desaconseja en la programación.

7.4.1. super y super()

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:

```
public class SuperClase {
   int a;
   int b;
   public void mostrarDatos() {
      ...
   }
}

public class Subclase extends SuperClase {
    String b;
    public 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` en el código de `Subclase`, 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;

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

public class Empleado extends Persona {
     double salario;
     @Override
     public void mostrarDatos() {
         super.mostrarDatos(); /*método de la superclase, muestra los
         atributos definidos en Persona*/
         System.out.println(salario); /*muestra el atributo añadido en
         Empleado*/
     }
}
```

Algo análogo ocurre con los constructores. Para ellos disponemos del método `super()`, que invoca un constructor de la superclase. Desde el constructor de la subclase, podemos invocar uno de la superclase con objeto de inicializar los atributos heredados de ella. En nuestro ejemplo, quedaría:

```
public class Persona {
    String nombre;
    byte edad;
    double estatura;
    public Persona (String nombre, byte edad, double estatura) {
        this.nombre = nombre;
        this.edad = edad;
        this.estatura = estatura;
    }
    ...
}

public class Empleado extends Persona {
    double salario;
    public Empleado(String nombre, byte edad, double estatura, double
    salario) {
        super(nombre, edad, estatrua); //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 forzosamente la primera instrucción que aparezca en la implementación de un constructor.

Aquí hay que hacer mención del caso de los atributos privados. Sabemos que 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`.

7.4.2. 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 (véase Figura 4):

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

¿Es lo mismo una variable Empleado que una variable Persona para referenciar un objeto Empleado?

No. Hay una sutil pero importante diferencia. En primer lugar, solo serán visibles los miembros (tanto atributos como métodos) definidos en la clase Persona. Sin embargo, cuando hay ocultación de atributos o sustitución de métodos en la subclase, ¿a qué versión accedemos, la de la variable o la del objeto referenciado? Depende, 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
```

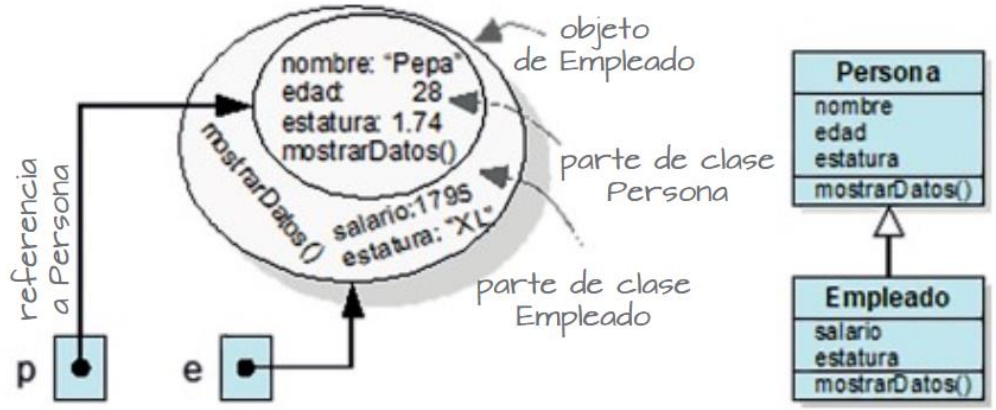

Figura 4. Objeto de la clase Empleado.

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

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`.

```
public class Cliente extends Persona {
    ...
    @Override
    public 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, en tiempo de ejecución, usando una misma variable, se ejecuta un método u otro, según la clase del objeto referenciado.”

7.5. LA CLASE OBJECT

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`

```
public 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.
- Como se ha visto en el Apartado 7.4.2, poder referenciar cualquier objeto, de cualquier tipo, mediante una variable de tipo `Object`.

Si queremos ver los métodos de `Object` que ha heredado `Persona`, escribiremos en NetBeans una variable de tipo `Persona`, seguida de un punto (.) Se desplegarán todos los atributos y métodos disponibles: los propios (en negrita) más los heredados de `Object`.

Veamos los métodos más importantes de `Object`, heredados por todas las clases de Java.

7.5.1. 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@2a139a55”
```

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.

Vamos a reimplementar `toString()` en `Persona`; podemos elegir cómo queremos representar una persona, pero en este caso decidimos que una representación adecuada consiste en el nombre junto a la edad, omitiendo la estatura.

```
public class Persona {
    ...
    @Override
    public String toString() { //siempre utilizar public
        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 p = 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());
```

7.5.2. 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;
}
```

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.

La mayoría de las clases de la API tienen su propia implementación de `equals()`, que permite comparar sus objetos entre sí. Sin embargo, los Arrays, a pesar de ser objetos, no traen implementado el método `equals()`. 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:

```
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
```

7.5.3. 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 `Obejct` 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
Syste.mout.println(superclase.getName()); //nombre: java.lang.Number
```

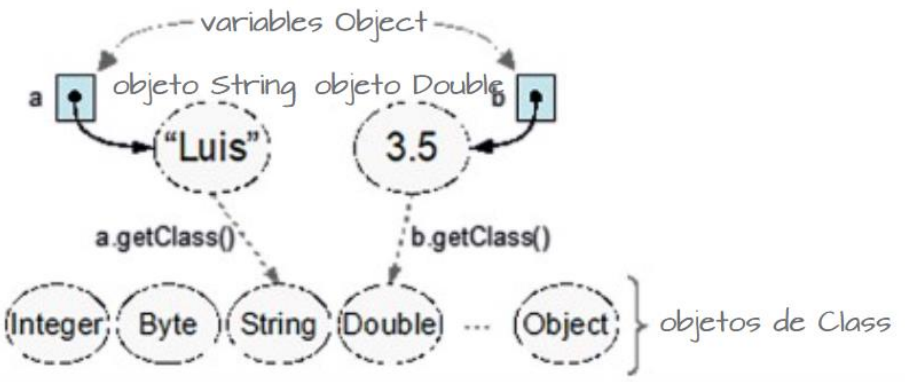

Figura 5. Todas las clases existentes son objetos de la clase Class.

7.6. CLASES ABSTRACTAS

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:

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

Las subclases deberán implementar el método `mostrarDatos()`, cada una con las particularidades 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
public abstract class A {
    int X = l;
    public void metodo1() { //método implementado y heredados por subclases
        System.out.println("método1 definido en A");
    }
    public 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.

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

El que no se puedan crear objetos de clase `A` no significa que no puedan 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 `c` 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.**