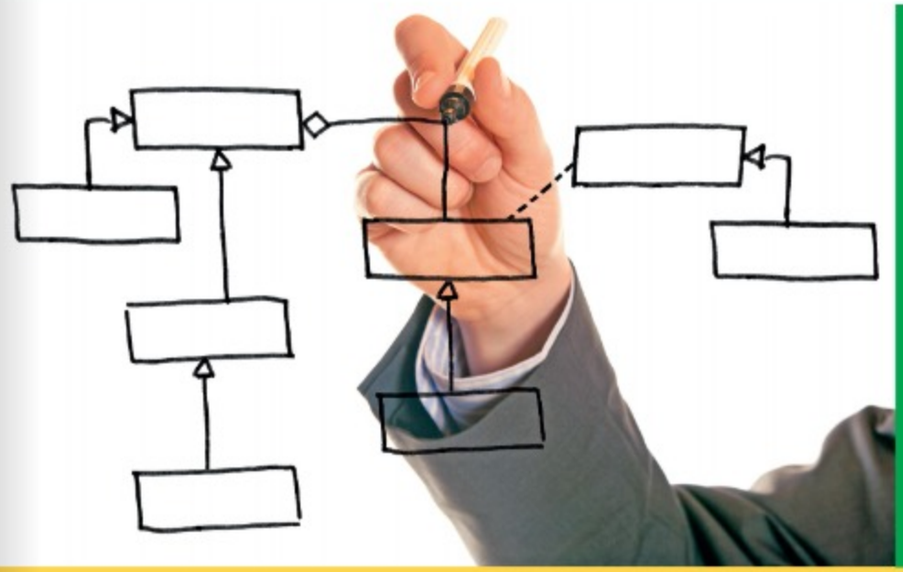


UNIDAD 8

Herencia

Objetivos

- Conocer la idea y la necesidad de la herencia entre clases.
- Saber cuáles son los conceptos de subclase y superclase.
- Utilizar la herencia como mecanismo de especialización.
- Conocer las limitaciones de acceso a los miembros de una superclase.
- Comprender y usar el mecanismo de la sustitución de métodos.
- Utilizar el acceso a miembros sustituidos de una superclase.
- Comprender y usar la selección dinámica de métodos en tiempo de ejecución.
- Reconocer las principales funcionalidades definidas en la clase `Object`.
- Usar la sustitución de los métodos de `Object`. En particular, la implementación de `toString()`, `equals()`.
- Conocer y utilizar las clases abstractas.
- Usar las clases abstractas para la selección dinámica de métodos.

Contenidos

- 8.1. Subclase y superclase
- 8.2. Modificador de acceso para herencia
- 8.3. Redefinición de miembros heredados
- 8.4. La clase Object
- 8.5. Clases abstractas

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 *superclase*, y la clase que hereda es conocida como clase *hija* o *subclase*. El diagrama de clases de la Figura 8.1 representa el concepto de herencia.

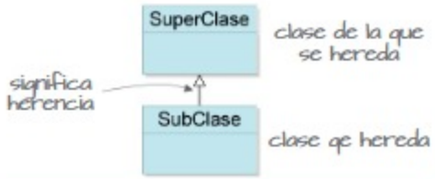


Figura 8.1. *Herencia entre clases.*

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 —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.

```
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`. Por ejemplo,

```
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.

Argot técnico

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 otra u otras.

8.2. 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, independien-

temente de si la superclase y la subclase son vecinas o externas, aunque en este último caso, habrá que importar la superclase.

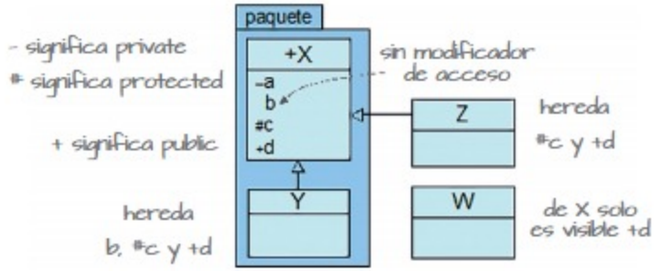


Figura 8.2. *Visibilidad de un miembro protected.*

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.

Tabla 8.1. Alcance de la visibilidad (incluida la herencia) según el modificador de acceso

	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 8.2:

```
public class X {
    private int a; //invisible fuera de la clase
    int b; //visibilidad 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.

INFORMÁTICA Y COMUNICACIONES

8.3. 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.

Argot técnico

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

```
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 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`:

```
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);
    }
}
```

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 8.3 muestra un ejemplo de qué miembros se usan.

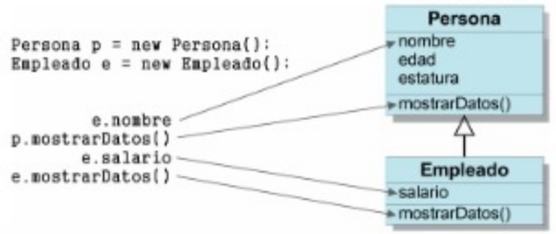


Figura 8.3. *Uso de miembros, heredados o propios.*

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 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
            ...
}
```

El código de la Figura 8.3 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.

8.3.1. 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:

```
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` 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;
    ...
    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 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;
    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 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`.

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 (véase Figura 8.4):

```
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
```


INFORMÁTICA Y COMUNICACIONES

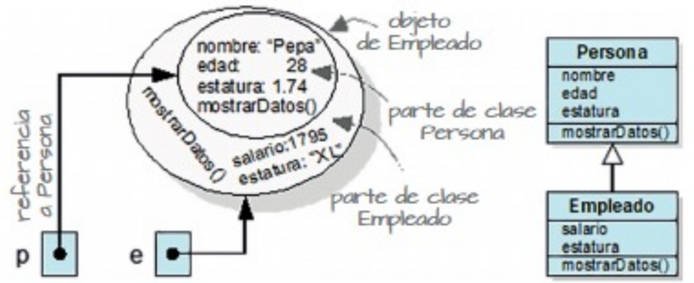


Figura 8.4. *Objeto de la clase Empleado.*

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.

Argot técnico

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.

8.4. 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`

```
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 8.3.2, 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.

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.

INFORMÁTICA Y COMUNICACIONES

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.

```
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());
```

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
```


INFORMÁTICA Y COMUNICACIONES

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
```

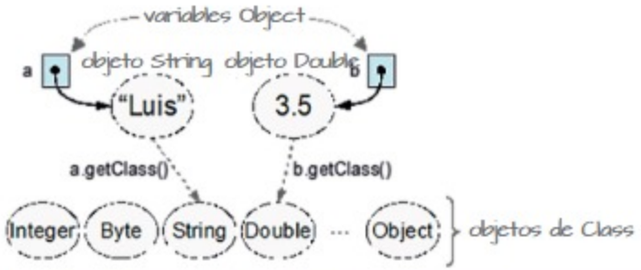


Figura 8.5. *Todas las clases existentes son objetos de la clase Class.*

Actividad resuelta 8.1

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 setMinutos(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.

Solución

```
public class Hora {
    protected int hora , minutos; //atributos protegidos, pensados para heredar
    Hora(int hora , int minutos) { //constructor
        this.hora = 0; //valores por defecto
        this.minutos = 0;
        if (!setHora(hora)) { //usamos métodos de asignación, que comprueban los
                              //valores
            System.out.println("La hora es incorrecta");
        }
        if (!setMinutos(minutos)) {
            System.out.println("Los minutos no son válidos");
        }
    }
    public void inc() { //incrementa la hora +1 minuto
        minutos++;
        if (minutos > 59) { //comprobamos si los minutos sobrepasan 59
            minutos = 0; //reiniciamos los minutos a 0
            hora++; //e incrementamos la hora
            if (hora > 23) { //si la hora es mayor a 23 (algo que no tiene sentido)
                hora = 0; //reiniciamos la hora a 0
            }
        }
    }
    public boolean setMinutos(int minutos) {
        boolean correcto = false;
        if (0 <= minutos && minutos < 60) { //solo modificamos si valor está en 0..59
            this.minutos = minutos;
            correcto = true;
        }
        return correcto;
    }
    public boolean setHora(int hora) {
        boolean correcto = false;
        if (0 <= hora && hora < 24) { //solo modificamos si el valor está en 0..23
            this.hora = hora;
            correcto = true;
        }
        return correcto;
    }
    @Override //indica que estamos sustituyendo (overriding) el método
    public String toString() {
        String result;
        result = hora + ":" + minutos;
        return result;
    }
}
```

Programa Principal

```
//vamos a probar la clase Hora
static public void main(String args[]) {
    Hora r = new Hora(11, 30); //las 11:30
    System.out.println(r);
    for (int i = 1; i <= 61; i++) { //incrementamos 61 minutos
        r.inc();
    }
    System.out.println(r); //mostramos
    System.out.println("Escriba una hora:");
    int hora = new Scanner(System.in).nextInt();
    boolean cambio = r.setHora(hora); //cambiamos la hora
    if (cambio) {
        System.out.println(r);
    } else {
        System.out.println("La hora no se pudo cambiar");
    }
}
```

Actividad resuelta 8.2

A partir de la clase `Hora` implementar la clase `HoraExacta`, que incluye en la hora los segundos. 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.
- `inc()`, que incrementa la hora en un segundo.

Solución

```
public class HoraExacta extends Hora { //heredamos de la clase Hora
    protected int segundos; //añadimos un atributo para los segundos
    public HoraExacta(int hora , int minutos, int segundos) {
        super(hora , minutos); //aprovechamos el constructor de la superclase
        //this.segundos = segundos; permitiría asignar cualquier valor a los
        //segundos
        if (!setSegundos(segundos)) { //mejor usar el método para asignar valores
            System.out.println("Segundos incorrectos ");
        }
    }
    //añadimos un método que asigna los segundos
    public boolean setSegundos (int segundos) {
        boolean correcto = false;
        if (0 <= segundos && segundos < 60) { //si está en un rango válido
            this.segundos = segundos; //modificamos los segundos
            correcto = true;
        }
        return correcto;
    }
    @Override //sustituimos el método para incrementar segundos en lugar de minutos
    public void inc() {
        segundos++;
        if (segundos > 59) { //si los segundos son mayores que 59
            segundos = 0; // inicializamos los segundos
            super.inc(); //+1 con el método inc() de la superclase, que
                         //incrementa minutos
        }
    }
    @Override //sustituimos toString() para mostrar los segundos
    public String toString() {
        String result = super.toString(); //utilizamos toString() de la superclase
        result += ":" + segundos; //añadimos los segundos
        return result;
    }
}
```

Programa Principal

```
static public void main(String args[]) {
    HoraExacta r = new HoraExacta (11, 15, 23); //¡hora del descanso!
    System.out.println(r);
    for (int i = 1; i <= 61; i++) {
        r.inc();
    }
    System.out.println(r);
    System.out.println("Escriba los segundos: ");
    int segundos = new Scanner(System.in).nextInt();
    if (r.setSegundos (segundos)) {
        System.out.println(r);
    } else{
        System.out.println("No es posible cambiar los segundos");
    }
}
```

Actividad resuelta 8.3

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.

Solución

```
public class HoraExacta extends Hora { //hereda de Hora
    //resto de implementación de la clase
    /*Reimplementaremos (overriding) el método equals() heredado de la clase
    Object, para comparar dos horas, que serán iguales si sus horas, minutos y
    segundos son iguales.
    La hora con la que tenemos que comparar se pasa como un objeto de la clase
    Object, que tendremos que convertir (cast) a HoraExacta.*/
    @Override
    public boolean equals(Object o) {
        HoraExacta otroReloj = (HoraExacta) o; //el mismo objeto está referenciado
        //como Object (con el parámetro o) y como HoraExacta (con la variable
        //otroReloj).
        boolean iguales;
        if (this.hora == otroReloj.hora //si las horas son iguales
            && this.minutos == otroReloj.minutos// y los minutos son iguales
            && this.segundos == otroReloj.segundos) {//y los segundos son iguales
                iguales = true; //son iguales
        } else {
            iguales = false; //no son iguales
        }
        return iguales;
    }
}
```

Programa principal

```
static public void main(String args[]) {
    HoraExacta a = new HoraExacta (1, 2, 3);
    HoraExacta b = new HoraExacta (1, 2, 3);
    HoraExacta c = new HoraExacta (10, 20, 30);
    System.out.println(a.equals(b)); //son iguales
    System.out.println(a.equals(c)); //son distintas
}
```

8.5. 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:

```
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
abstract class A {
    int x = 1;
    void metodo1() {  //método implementado y heredados por las subclases
            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 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.**

Actividad resuelta 8.4

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.

Solución

```
/* La clase abstracta Instrumento , básicamente contiene una tabla con una serie
de notas. Cada clase que herede de Instrumento, tendrá que implementar el método
interpretar() donde se decide de qué forma suenan las notas. Distinguiremos un
timbre de otro, por la forma en que escribamos las notas, por ejemplo: do, Do,
Dolooon, doooooooooo , etc. */
public abstract class Instrumento {
    protected Nota[] melodia; //tabla que almacena las notas a interpretar
    public Instrumento () {
        melodia = new Nota[0]; //creamos la tabla
    }
    //Usa el algoritmo de inserción no ordenada
    void add(Nota n) {
        melodia = Arrays.copyOf(melodia, melodia.length + 1); //redimensionamos
        melodia[melodia.length - 1] = n; //insertamos el nuevo elemento al final
    }
    abstract void interpretar (); //a implementar en cada subclase
}
```

Enumerado `Nota`

```
//Enumerado con las nota musicales
public enum Nota {DO, RE, MI, FA, SOL, LA, SI}
```

Actividad resuelta 8.5

Crear la clase `Piano` heredando de la clase abstracta `Instrumento`.

Solución

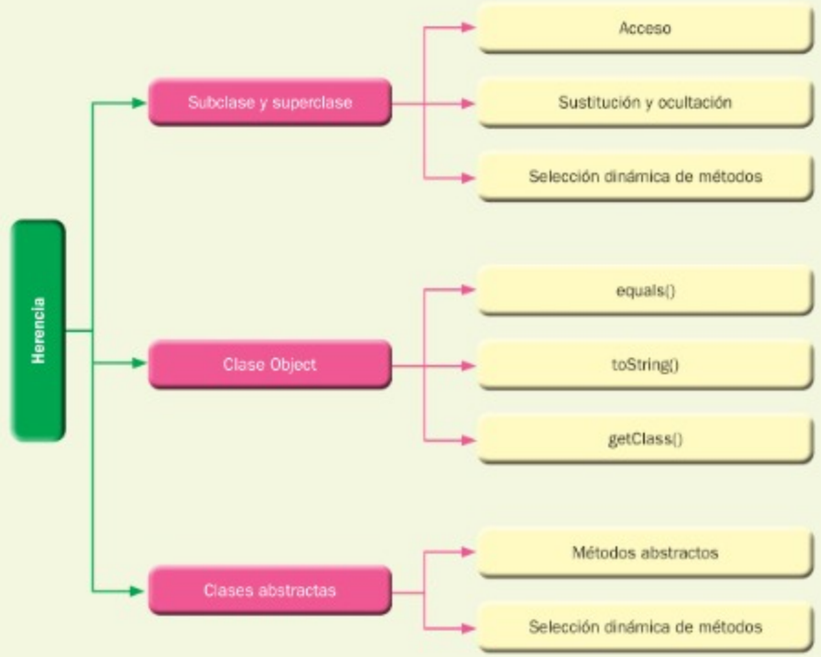
```
//Un piano es un instrumento que interpreta las notas con un timbre muy
//característico
public class Piano extends Instrumento {
    //podemos añadir tantos atributos y métodos como necesitemos
    //...
    public Piano() {
        super(); //constructor de la superclase
    }
    @Override //implementamos el método abstracto
    //recorreremos las notas y las interpretaremos de la forma específica del piano.
    public void interpretar () {
        for (Nota nota: melodia) {
            switch (nota) {
                case DO:
                    System.out.print("do ");
                    break;
                case RE:
                    System.out.print("re ");
                    break;
                case MI:
                    System.out.print("mi ");
                    break;
                case FA:
                    System.out.print("fa ");
                    break;
                case SOL:
                    System.out.print("sol ");
                    break;
                case LA:
                    System.out.print("la ");
                    break;
                case SI:
                    System.out.print("si ");
                    break;
            }
        }
        System.out.println("");
    }
}
```

Programa Principal

```
public static void main(String[] args) {
    Nota cancion[] = {Nota.DO, Nota.SI, Nota.SOL, Nota.RE, Nota.FA}; //notas
    Piano p = new Piano();
    for(Nota nota: cancion) { //añadimos las notas al piano
        p.add(nota);
    }
    p.interpretar ();
}
```


© Ediciones Paraninfo

MAPA CONCEPTUAL



ACTIVIDADES FINALES

Actividades de comprobación

8.1. Sobre una subclase es correcto afirmar que:
- a) Tiene menos atributos que su superclase.
- b) Tiene menos miembros que su superclase.
- c) Hereda los miembros no privados de su superclase.
- d) Hereda todos los miembros de su superclase.

8.2. En relación con las clases abstractas es correcto señalar que:
- a) Implementan todos sus métodos.
- b) No implementan ningún método.
- c) No tienen atributos.
- d) Tienen algún método abstracto.

8.3. ¿En qué consiste la sustitución u overriding?
- a) En sustituir un método heredado por otro implementado en la propia clase.
- b) En sustituir un atributo por otro del mismo nombre.
- c) En sustituir una clase por una subclase.
- d) En sustituir un valor de una variable por otro.

8.4. Sobre la clase `Object` es cierto indicar que:
- a) Es abstracta.
- b) Hereda de todas las demás.
- c) Tiene todos sus métodos abstractos.
- d) Es superclase de todas las demás clases.

8.5. ¿Cuál de las siguientes afirmaciones sobre el método `equals()` es correcta?
- a) Hay que implementarlo, ya que es abstracto.
- b) Sirve para comparar solo objetos de la clase `Object`.
- c) Se hereda de `Object`, pero debemos reimplementarlo al definirlo en una clase.
- d) No hay que implementarlo, ya que se hereda de Object.

8.6. ¿Cuál de las siguientes afirmaciones sobre el método `toString()` es correcta?
- a) Sirve para mostrar la información que nos interesa de un objeto.
- b) Convierte automáticamente un objeto en una cadena.
- c) Encadena varios objetos.
- d) Es un método abstracto de `Object` que tenemos que implementar.

8.7. ¿Cuál de las siguientes afirmaciones sobre el método `getClass()` es correcta?
- a) Convierte los objetos en clases.
- b) Obtiene la clase a la que pertenece un objeto.
- c) Obtiene la superclase de una clase.
- d) Obtiene una clase a partir de su nombre.

8.8. Una clase puede heredar:
- a) De una clase.
- b) De dos clases.
- c) De todas las clases que queramos.
- d) Solo de la clase `Object`.

8.9. La selección dinámica de métodos:

a) Se produce cuando una variable cambia de valor durante la ejecución de un programa.
b) Es el cambio de tipo de una variable en tiempo de ejecución.
c) Es la asignación de un mismo objeto a más de una variable en tiempo de ejecución.
d) Es la ejecución de distintas implementaciones de un mismo método, asignando objetos de distintas clases a una misma variable, en tiempo de ejecución.

8.10. ¿Cuál de las siguientes afirmaciones sobre el método `super()` es correcta?

a) Sirve para llamar al constructor de la superclase.
b) Sirve para invocar un método escrito más arriba en el código.
c) Sirve para llamar a cualquier método de la superclase.
d) Sirve para hacer referencia a un atributo de la superclase.

Actividades de aplicación

8.11. Crea la clase `Campana` que hereda de `Instrumento` (definida en la Actividad resuelta 8.4).

8.12. Las empresas de transporte, para evitar daños en los paquetes, embalan todas sus mercancías en cajas con el tamaño adecuado. Una caja se crea expresamente con un ancho, un alto y un fondo y, una vez creada, se mantiene inmutable. Cada caja lleva pegada una etiqueta, de un máximo de 30 caracteres, con información útil como el nombre del destinatario, dirección, etc. Implementa la clase `Caja` con los siguientes métodos:

- `Caja(int ancho, int alto, int fondo, Unidad unidad)`: que construye una caja con las dimensiones especificadas, que pueden encontrarse en «cm» (centímetros) o «m» (metros).
- `double getVolumen()`: que devuelve el volumen de la caja en metros cúbicos.
- `void setEtiqueta(String etiqueta)`: que modifica el valor de la etiqueta de la caja.
- `String toString()`: que devuelve una cadena con la representación de la caja.

8.13. La empresa de mensajería BiciExpress, que reparte en bicicleta, para disminuir el peso que transportan sus empleados solo utiliza cajas de cartón. El volumen de estas se calcula como el 80 % del volumen real, con el fin de evitar que se deformen y se rompan. Otra característica de las cajas de cartón es que sus medidas siempre están en centímetros. Por último, la empresa, para controlar costes, necesita saber cuál es la superficie total de cartón utilizado para construir todas las cajas.
Escribe la clase `CajaCarton` heredando de la clase `Caja`.

8.14. Reimplementa la clase `Lista` de la Actividad resuelta 7.11, sustituyendo el método `mostrar()` por el método `toString()`.

8.15. Escribe en la clase `Lista` un método `equals()` para compararlas. Dos listas se considerarán iguales si tienen los mismos elementos (incluidas las repeticiones) en el mismo orden.

8.16. Diseña la clase `Pila` heredando de `Lista` (ver Actividad resuelta 7.13).

8.17. Escribe la clase `Cola` heredando de `Lista` (ver Actividad final 7.18).

8.18. Diseña la clase `ColaDoble`, que hereda de `Cola` para enteros, añadiendo los siguientes métodos:

- `void encolarPrincipio()`, que encola un elemento al principio de la cola.
- `Integer desencolarFinal()`, que desencola un elemento del final de la cola.

8.19. Un conjunto es un objeto similar a las listas, capaz de guardar valores de un tipo determinado, con la diferencia de que sus elementos no pueden estar repetidos. Escribe la clase `Conjunto` para enteros heredando de `Lista` y reimplementando los métodos de inserción para evitar las repeticiones.

8.20. Implementa el método `equals()` en la clase `Conjunto`. Dos conjuntos se consideran iguales si tienen los mismos elementos, no importa en qué orden.

8.21. Implementa los siguientes métodos:

- `static boolean esNumero(Object ob)`, que nos dice si su parámetro de entrada es de tipo numérico (`Integer, Double, Long, Float. . .`).
- `boolean sumar(Object a, Object b)`, que muestra por consola la concatenación de los parámetros de entrada, si ambos son cadenas, o muestra su suma convertida al tipo `Double`, si ambos son de tipo numérico. En cualquier otro caso, muestra el mensaje «No sumables».

8.22. La clase `Object` dispone del método `finalize()`, que se ejecuta justo antes de que el recolector de basura destruya un objeto. Escribe un programa que, mediante la creación masiva de objetos no referenciados y el overriding del método `finalize()`, compruebe el funcionamiento del recolector de basura.

8.23. Implementa la clase abstracta `Poligono`, con los atributos `base` y `altura`, de tipo double y el método abstracto `double area()`.

8.24. Heredando de `Poligono`, implementa las clases no abstractas `Triangulo` y `Rectangulo`.

Actividades de ampliación

8.25. Define la clase `Punto`, que tiene como atributos las coordenadas `x` e `y`, de tipo entero, que lo sitúan en el plano. Además del constructor, implementa el método

```
double distancia(Punto otroPunto),
```

que devuelve la distancia a otro punto que se le pasa como parámetro.
A partir de `Punto`, por herencia, implementa la clase `Punto3D`, que representa un punto en tres dimensiones y necesita una coordenada adicional `z`. Reimplementa el método `distancia()` para puntos 3D.

8.26. A partir de la clase `Calendario`, implementada en la Actividad de aplicación 7.15, escribe la clase `CalendarioExacto`, que determina un instante de tiempo exacto formado por un año, un mes, un día, una hora y un minuto. Implementa los métodos `toString()`, `equals()` y aquellos necesarios para manejar la clase.

8.27. Implementa el método `equals()` para las clases `Punto` y `Punto3D`, teniendo en cuenta que dos puntos son iguales solo si tienen todas sus coordenadas iguales.

8.28. Implementa la clase `Suceso`, que hereda de `Punto3D`. Un suceso está caracterizado de forma única por el lugar y el instante en que ocurre (el atributo `tiempo` de tipo int). Añade un atributo `descripcion` de tipo `String`.
Implementa el método `equals()` para sucesos.

8.29. Calcula la raíz cuadrada de 2 con 100 cifras significativas usando objetos de la clase `BigDecimal`.

UNIDAD 9

Interfaces

Objetivos

- Conocer la idea y la necesidad de las interfaces y distinguirlas de las clases abstractas.
- Definir métodos abstractos en una interfaz.
- Implementar una o más interfaces en una clase.
- Implementar métodos de extensión en una interfaz.
- Implementar métodos privados como auxiliares en una interfaz.
- Diseñar interfaces para operaciones específicas en clases diversas.
- Utilizar variables de tipo interfaz para conseguir la selección dinámica de métodos.
- Conocer qué son la herencia simple y múltiple de interfaces.
- Conocer algunas interfaces importantes de la API.
- Saber implementar clases anónimas.
- Implementar criterios de comparación y aplicarlos a procesos de búsqueda y ordenación.

Contenidos

- 9.1. Concepto de interfaz
- 9.2. Atributos de una interfaz
- 9.3. Métodos implementados en una interfaz
- 9.4. Herencia
- 9.5. Variables de tipo interfaz
- 9.6. Clases anónimas
- 9.7. Acceso entre miembros de una interfaz
- 9.8. Sintaxis general
- 9.9. Un par de interfaces de la API