```
    }
    ...
}
```

La llamada al constructor con los valores de los parámetros de entrada se hace por medio del operador `new`. Si deseamos crear un objeto `Persona` con los datos anteriores,

```
Persona p = new Persona("Claudia", 8, 1.20); //creamos el objeto
//y lo inicializamos mediante el constructor
```

Los atributos declarados como `final` también se pueden inicializar pasando sus valores como parámetros al constructor; no es necesario hacerlo en el sitio donde se declaran.

A la hora de sobrecargar un método tenemos que asegurarnos de que se pueda distinguir entre las distintas versiones mediante el número o el tipo de parámetros de entrada. La sobrecarga de constructores es útil cuando necesitamos inicializar objetos de varias formas. Hemos visto un constructor de `Persona` que permite asignar valores a todos los atributos. Podría darse el caso de que solo nos interesara pasar al constructor el nombre de la persona, dejando que el resto de los atributos se inicializaran con algunos valores arbitrarios.

```
clase Pereona {
   ...
   //constructor que asigna valores a todos los atributos
   Persona (String nombre, int edad, double estatura) {
      this.nombre = nombre;
      this.edad = edad;
      this.estatura = estatura;
   }

   //constructor sobrecargado: solo asigna el nombre
   Persona (String nombre) {
      this.nombre = nombre;
      estatura = 1.0; //valor arbitrario para la estatura
      //al no asignar la edad se inicializa por defecto: a 0
   }
```

```
Mascota perro = new Mascota();
```

Actividad resuelta 7.1

Diseñar la clase `CuentaCorriente`, que almacena los datos: DNI y nombre del titular, así como el saldo. Las operaciones típicas con una cuenta corriente son:

- Crear una cuenta: se necesita el DNI y nombre del titular. El saldo inicial será 0.
- Sacar dinero: el método debe indicar si ha sido posible llevar a cabo la operación, si existe saldo suficiente.
- Ingresar dinero: se incrementa el saldo.
- Mostrar información: muestra la información disponible de la cuenta corriente.

Solución

Clase `CuentaCorriente`

```
class CuentaCorriente {
   String dni; //del titular
   String nombre; //del titular
   double saldo; //efectivo disponible en la cuenta
   //Los parámetros de entrada: nombre y dni, ocultan a los atributos de la clase
   //con el mismo identificador. Para acceder a ellos hay que utilizar this.
   CuentaCorriente(String dni, String nombre) { //constructor
      this.dni = dni; //DNI pasado como parámetro
      this.nombre = nombre; //nombre pasado como parámetro
      saldo = 0; //asignamos el saldo por defecto
   }
   boolean egreso(double cant) { //sacar dinero de la cuenta corriente
      boolean operacionPosible;
      if (saldo >= cant) { //si disponemos de saldo suficiente
         saldo -= cant;
         operacionPosible = true;
      } else { //no hay saldo disponible
         System.out.println("No hay dinero suficiente");
         operacionPosible = false;
```

atributos. Podría darse el caso de que solo nos interesara pasar al constructor el nombre de la persona, dejando que el resto de los atributos se inicializaran con algunos valores arbitrarios.

```
class Persona {
   ...
   //constructor que asigna valores a todos los atributos
   Persona (String nombre, int edad, double estatura) {
      this.nombre = nombre;
      this.edad = edad;
      this.estatura = estatura;
   }

   //constructor sobrecargado: solo asigna el nombre
   Persona (String nombre) {
      this.nombre = nombre;
      estatura = 1.0; //valor arbitrario para la estatura
      //al no asignar la edad se inicializa por defecto: a 0
   }
}
```

Ahora disponemos de dos constructores, que se utilizan de la forma:

```
Persona a = new Persona("Pepe", 20, 1.90);
Persona b = new Persona("Dolores");
```

Cuando en una clase no se implementa ningún constructor, Java se encarga de crear uno que se denomina *constructor por defecto*. Este no usa parámetros de entrada e inicializa los atributos a cero, false o `null` según el tipo si no están ya inicializados en su declaración. No obstante, es conveniente implementar los constructores y no dejarlo en manos de Java. En cuanto se implementa un constructor en una clase, el constructor, por defecto, deja de estar disponible.

Solución

Clase `CuentaCorriente`

```
class CuentaCorriente {
   String dni; //del titular
   String nombre; //del titular
   double saldo; //efectivo disponible en la cuenta
   //Los parámetros de entrada: nombre y dni, ocultan a los atributos de la clase
   //con el mismo identificador. Para acceder a ellos hay que utilizar this.
   CuentaCorriente(String dni, String nombre) { //constructor
      this.dni = dni; //DNI pasado como parámetro
      this.nombre = nombre; //nombre pasado como parámetro
      saldo = 0; //asignamos el saldo por defecto
   }
   boolean egreso(double cant) { //sacar dinero de la cuenta corriente
      boolean operacionPosible;
      if (saldo >= cant) { //si disponemos de saldo suficiente
         saldo -= cant;
         operacionPosible = true;
      } else { //no hay saldo disponible
         System.out.println("No hay dinero suficiente");
         operacionPosible = false;
      }
      return (operacionPosible); //indica si ha sido posible realizar la operación
   }
   void ingreso(double cant) { //añadimos dinero a la cuenta corriente
      saldo += cant;
   }
   void mostrarInformacion() { //muestra el estado de la cuenta corriente
      System.out.println("Nombre: " + nombre);
      System.out.println("Dni: " + dni);
      System.out.println("Saldo: " + saldo + " euros");
   }
}
```

Programa principal

```
//Creamos un objeto CuentaCorriente para probar la clase y realizar algunas operaciones.
public class Main {
   public static void main(String[] args) {
      CuentaCorriente c;
      c = new CuentaCorriente("12345678A", "Pepe"); //Cuenta de Pepe con DNI 12.345.678-A
      c.ingreso(1000); // ingresamos 1000 euros
      c.egreso(300); // sacamos 300 euros. Quedarán 700 euros
      c.mostrarInformacion(); // mostramos
      System.out.println("Puedo sacar 700 euros: " + c.egreso(700)); //quedan 0 euros
      System.out.println("Puedo sacar 500 euros: " + c.egreso(500)); //no es posible
   }
}
```

7.7.1. this()

Cuando una clase dispone de un conjunto de constructores sobrecargados, es posible que un constructor invoque a otro y así reutilice su funcionalidad. Para eso se usa el constructor genérico `this()`, en lugar del constructor por su nombre. La forma de distinguir los distintos constructores, igual que en cualquier método sobrecargado, es mediante el número y el tipo de los parámetros de entrada.

Vamos a redefinir el constructor de `Persona` al que solo se le pasa el nombre usando `this()`:

```
class Persona {
   ...
   //constructor que asigna valores a todos los atributos
   Persona (String nombre, int edad, double estatura) {
      this.nombre = nombre;
      this.edad = edad;
      this.estatura = estatura;
   }

   //constructor sobrecargado que solo asigna el nombre
```

Solución a)

Clase `CuentaCorriente`

```
// Sobrecargamos los constructores
class CuentaCorriente {
   ... //resto de implementación
   CuentaCorriente(String dni, String nombre) { //constructor
      this.dni = dni; //DNI pasado como parámetro
      this.nombre = nombre; //nombre pasado como parámetro
      saldo = 0; //asignamos el saldo por defecto
   }
   CuentaCorriente(String dni, double saldo) { //constructor
      this.dni = dni;
      this.saldo = saldo;
      this.nombre = "Sin asignar"; //indicamos que no disponemos del nombre
   }
   CuentaCorriente(String dni, String nombre, double saldo) { //constructor
      this.dni = dni;
      this.nombre = nombre;
      this.saldo = saldo;
   }
}
```

Programa principal

```
//probamos los métodos
public class Main {
   public static void main(String[] args) {
      CuentaCorriente c;
      c = new CuentaCorriente("12345678-A", "Pepe");//crea un objeto con DNI y nombre
      c.ingreso(1000); // ingresamos 1000 euros
      c.egreso(300); // sacamos 300 euros. Quedarán 700 euros
      c.mostrarInformacion(); // mostramos
      System.out.println("Puedo sacar 700 euros: " + c.egreso(700)); //quedan 0 euros
      System.out.println("Puedo sacar 500 euros: " + c.egreso(500)); //no es posible
      //vamos a probar el constructor que utiliza el dni y el saldo:
      c = new CuentaCorriente("99755432-B", 2000); //c referencia al nuevo objeto ,
      //el anterior queda sin referencia a merced del recolector de basura
      c.mostrarInformacion();
   }
```

7.7.1. this()

Cuando una clase dispone de un conjunto de constructores sobrecargados, es posible que un constructor invoque a otro y así reutilice su funcionalidad. Para eso se usa el constructor genérico `this()`, en lugar del constructor por su nombre. La forma de distinguir los distintos constructores, igual que en cualquier método sobrecargado, es mediante el número y el tipo de los parámetros de entrada.

Vamos a redefinir el constructor de `Persona` al que solo se le pasa el nombre usando `this()`:

```
class Persona {
   ...
   //constructor que asigna valores a todos los atributos
   Persona (String nombre, int edad, double estatura) {
      this.nombre = nombre;
      this.edad = edad;
      this.estatura = estatura;
   }

   //constructor sobrecargado que solo asigna el nombre
   Persona (String nombre) {
      this(nombre, 0, 1.0); //invoca al primer constructor
      //la edad se pone a 0 y la estatura a 1.0
   }
}
```

Tenemos que tener presente que, en el caso de utilizar `this()`, tiene que ser siempre la primera instrucción de un constructor; en otro caso se producirá un error.

Actividad resuelta 7.2

En la clase `CuentaCorriente` sobrecargar los constructores para poder crear objetos.

- Con el DNI del titular de la cuenta y un saldo inicial.
- Con el DNI, nombre y el saldo inicial.

Escribir un programa que compruebe el funcionamiento de los métodos.

© Ediciones Paraninfo

```
      this.dni = dni;
      this.nombre = nombre;
      this.saldo = saldo;
   }
}
```

Programa principal

```
//probemos los métodos
public class Main {
   public static void main(String[] args) {
      CuentaCorriente c;
      c = new CuentaCorriente("12345678-A", "Pepe");//crea un objeto con DNI y nombre
      c.ingreso(1000); // ingresamos 1000 euros
      c.egreso(300); // sacamos 300 euros. Quedarán 700 euros
      c.mostrarInformacion(); // mostramos
      System.out.println("Puedo sacar 700 euros: " + c.egreso(700)); //quedan 0 euros
      System.out.println("Puedo sacar 500 euros: " + c.egreso(500)); //no es posible
      //vamos a probar el constructor que utiliza el dni y el saldo:
      c = new CuentaCorriente("98765432-Z", 2000); //c referencia al nuevo objeto ,
      //el anterior queda sin referencia a merced del recolector de basura
      c.mostrarInformacion();
   }
}
```

Solución b)

Clase `CuentaCorriente`

```
//Vamos a reutilizar los constructores mediante this(). Es habitual invocar el
//constructor con más parámetros y adaptar los valores por defecto.
class CuentaCorriente {
   ...
   // Reutilizamos el constructor: CuentaCorriente(dni, nombre, saldo)
   CuentaCorriente(String dni, String nombre) { //constructor
      this(dni, nombre, 0); //saldo inicial por defecto a 0
      //Una alternativa sería: usar this(dni, saldo) y posteriormente asignar el nombre
      //this(dni, 0);
      //this.nombre = nombre;
   }
```

Tabla 7.2. Alcance de la visibilidad según el modificador de acceso

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

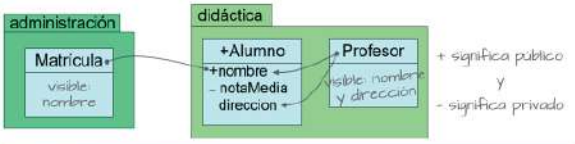

Figura 7.16. *Ejemplo de clases con visibilidad* `private`, `public` *y por defecto.*

Actividad resuelta 7.3

Modificar la visibilidad de la clase `CuentaCorriente` para que sea visible desde clases externas y la visibilidad de sus atributos para que:

- `saldo` no sea visible para otras clases.
- `nombre` sea público para cualquier clase.
- `dni` solo sea visible por clases vecinas.

Realizar un programa para comprobar la visibilidad de los atributos.

Solución

Clase `CuentaCorriente`

```
    //la clase CuentaCorriente
    c.dni = "11111111-T"; //al ser Main una clase vecina, dni es visible
    //en caso de acceder al dni desde una clase externa produciría un error
    c.nombre = "Antonio"; //nombre es visible desde cualquier clase
  }
}
```

7.9.3. Métodos get/set

Un atributo público puede ser modificado desde cualquier clase, lo que a veces tiene sus inconvenientes, ya que es imposible controlar los valores asignados, que pueden no tener sentido. Por ejemplo, nada impide que se asigne a un atributo edad un valor negativo.

Por este motivo, existe una convención en la comunidad de programadores que consiste en ocultar atributos y, en su lugar, crear dos métodos públicos: el primero —habitualmente llamado *set*— permite asignar un valor al atributo, controlando el rango válido de valores. Y el segundo —habitualmente llamado *get*— devuelve el atributo, lo que posibilita conocer su valor. Los métodos set/get hacen, en la práctica, que un atributo no visible se comporte como si lo fuera.

Estos métodos se identifican con set/get seguido del nombre del atributo. Para el atributo `edad` quedaría:

```
class Persona {
   private int edad;
   ...

   public void setEdad(int edad) {
      if (edad >= 0) //solo los valores positivos tienen sentido
         this.edad = edad;
      } // en caso contrario no se modifica la edad
   }

   public int getEdad() {
```

Figura 7.16. *Ejemplo de clases con visibilidad* `private`, `public` *y por defecto.*

Actividad resuelta 7.3

Modificar la visibilidad de la clase `CuentaCorriente` para que sea visible desde clases externas y la visibilidad de sus atributos para que:

- `saldo` no sea visible para otras clases.
- `nombre` sea público para cualquier clase.
- `dni` solo sea visible por clases vecinas.

Realizar un programa para comprobar la visibilidad de los atributos.

Solución

Clase `CuentaCorriente`

```
/* Marcamos la clase con public: para que sea visible desde clase externas donde es
 * posible usarla mediante importación. */
public class CuentaCorriente {
   String dni; //sin modificador, visibilidad por defecto. Solo visible por clases vecinas
   public String nombre; //visibilidad total
   private double saldo; //invisible para cualquier clase (vecina o externa)
   ...
}
```

Programa principal

```
// La clase Main es una clase vecina de CuentaCorriente
public class Main {
   public static void main(String[] args) {
      CuentaCorriente c;
      c = new CuentaCorriente("12345678-A", "Pepe"); //CuentaCorriente de ejemplo
      c.saldo = 2040; //produce un error, ya que el saldo no es visible desde fuera de
```


© Ediciones Paraninfo

su valor. Los métodos set/get hacen, en la práctica, que un atributo no visible se comporte como si lo fuera.

Estos métodos se identifican con set/get seguido del nombre del atributo. Para el atributo `edad` quedaría:

```
class Persona {
   private int edad;
   ...

   public void setEdad(int edad) {
      if (edad >= 0) //solo los valores positivos tienen sentido
         this.edad = edad;
      } // en caso contrario no se modifica la edad
   }

   public int getEdad() {
      return edad;
   }
}
```

Las ventajas de utilizar métodos set/get son que la implementación de la clase se encapsula, ocultando los detalles y, por otro lado, permite controlar qué atributos son accesibles para lectura y cuáles para escritura, así como los valores asignados. En nuestro ejemplo se ha limitado el uso a valores no negativos para la edad.

Actividad resuelta 7.4

Todas las cuentas corrientes con las que se va a trabajar pertenecen al mismo banco. Añadir un atributo que almacene el nombre del banco (que es único) en la clase `CuentaCorriente`. Diseñar un método que permita recuperar y modificar el nombre del banco (al que pertenecen todas las cuentas corrientes).

Solución

Clase `CuentaCorriente`

```
/* Vamos a añadir el atributo banco a la clase. Como el banco es el mismo para
 * todas las cuentas el atributo será estático. El atributo será privado y
 * escribiremos dos métodos estáticos para solicitar y modificar el nombre del banco.
*/
public class CuentaCorriente {
   ...
   static private String nombreBanco = "International Java Bank"; //valor por defecto
   //este valor se asigna antes de crear ningún objeto
   static void setBanco(String nuevoNombre) {
      nombreBanco = nuevoNombre;
   }
   static String getBanco() {
      return nombreBanco;
   }
}
```

Programa principal

```
// La clase Main y CuentaCorriente son vecinas (ubicadas en el mismo paquete).
public class Main {
   public static void main(String[] args) {
      CuentaCorriente c1, c2;
      c1 = new CuentaCorriente("12345678-A", "Pepe"); //CuentaCorriente para Pepe
      c2 = new CuentaCorriente("99999999-E", "Ana"); //cuenta de Ana
      c1.mostrarInformacion();
      CuentaCorriente.setBanco("Banco Central");
      c1.mostrarInformacion();
      CuentaCorriente.setBanco("Caja de Ahorros de Do-While");
      c1.mostrarInformacion();
      c2.mostrarInformacion();
   }
}
```

Solución

Clase `Gestor`

```
/* Para cumplir los requisitos:
 * -Todos los constructores usaran el nombre y el teléfono.
 * -El importe máximo autorizado tendrá un valor por defecto de 10000 euros.
 * -El teléfono será privado con un método get() para que se pueda consultar.
 * -El nombre será público y el importe máximo usará visibilidad por defecto. */
public class Gestor {
   public String nombre;
   private String tlf; //es un número con el que no se opera: es usual usar String
   double importeMax; //visibilidad por defecto
   public Gestor(String nombre, String tlf, double importeMax) {
      this.nombre = nombre;
      this.tlf = tlf;
      this.importeMax = importeMax;
   }
   public Gestor(String nombre, String tlf) {
      this(nombre, tlf, 10000.0); //asignamos el importe máximo por defecto:
                                  //10000 euros
   }
   String getTlf() { //al ser tlf privado permite consultar el teléfono de un gestor
      return tlf;
   }
   void mostrarInformacion() {
      System.out.println("Nombre: " + nombre);
      System.out.println("Teléfono: " + tlf);
      System.out.println("Importe máximo: " + importeMax + " euros");
   }
}
```

Clase `CuentaCorriente`

```
//Cada CuentaCorriente tendrá una referencia a un objeto de tipo Gestor.
public class CuentaCorriente {
... //resto de atributos y métodos
   Gestor gestor; //gestor que administra la cuenta
   CuentaCorriente(String dni, String nombre, Gestor gestor) { //sobrecargamos
      this(dni, nombre);
      this.gestor = gestor;
```

```
// La clase Main y CuentaCorriente son vecinas (ubicadas en el mismo paquete).
public class Main {
   public static void main(String[] args) {
      CuentaCorriente c1, c2;
      c1 = new CuentaCorriente("12345678-A", "Pepe"); //CuentaCorriente para Pepe
      c2 = new CuentaCorriente("999999999-E", "Ana"); //cuenta de Ana
      c1.mostrarInformacion();
      CuentaCorriente.setBanco("Banco Central");
      c1.mostrarInformacion();
      CuentaCorriente.setBanco("Caja de Ahorros de Do-While");
      c1.mostrarInformacion();
      c2.mostrarInformacion();
   }
}
```

Actividad resuelta 7.5

Existen gestores que administran las cuentas bancarias y atienden a sus propietarios.

Cada cuenta, en caso de tenerlo, cuenta con un único gestor. Diseñar la clase `Gestor` de la que interesa guardar su nombre, teléfono y el importe máximo autorizado con el que pueden operar. Con respecto a los gestores, existen las siguientes restricciones:

- Un gestor tendrá siempre un nombre y un teléfono.
- Si no se asigna, el importe máximo autorizado por operación será de 10 000 euros.
- Un gestor, una vez asignado, no podrá cambiar su número de teléfono. Y todo el mundo podrá consultarlo.

El nombre será público y el importe máximo solo será visible por clases vecinas.

Modificar la clase `CuentaCorriente` para que pueda disponer de un objeto `Gestor`. Escribir los métodos necesarios.

© Ediciones Paraninfo

```
   }
   String getTlf() { //al ser tlf privado permite consultar el teléfono de un gestor
      return tlf;
   }
   void mostrarInformacion() {
      System.out.println("Nombre: " + nombre);
      System.out.println("Teléfono: " + tlf);
      System.out.println("Importe máximo: " + importeMax + " euros");
   }
}
```

Clase `CuentaCorriente`

```
//Cada CuentaCorriente tendrá una referencia a un objeto de tipo Gestor.
public class CuentaCorriente {
... //resto de atributos y métodos
   Gestor gestor; //gestor que administra la cuenta
   CuentaCorriente(String dni, String nombre, Gestor gestor) { //sobrecargamos
      this(dni, nombre);
      this.gestor = gestor;
   }
   //permite asignar un nuevo objeto Gestor a la cuenta
   void setGestor(Gestor gestor) {
      this.gestor = gestor;
   }
   void mostrarInformacion() { //muestra el estado de la cuenta, incluido el gestor
      //No podemos usar directamente gestor.mostrarInformacion(), ya que puede
      //que el gestor sea null. Al intentar acceder a los miembros de un objeto
      //nulo se produce una excepción
      if (gestor == null) { //si la cuenta no está administrada por un gestor
         System.out.println("Cuenta sin gestor");
      } else {
         System.out.println("Información del gestor");
         gestor.mostrarInformacion(); //no es posible mostrar directamente sus
         //atributos, ya que algunos no son visibles
      }
```

```
        c1.mostrarInformacion();
        c2.mostrarInformacion();
        c3.mostrarInformacion();
        c1.setGestor(g1); //cambiamos de gestor
        c1.mostrarInformacion();
    }
}
```

Actividad resuelta 7.6

Escribir un programa que lea por teclado una hora cualquiera y un número *n* que representa una cantidad en segundos. El programa mostrará la hora introducida y las *n* siguientes, que se diferencian en un segundo. Para ello hemos de diseñar previamente la clase `Hora` que dispone de los atributos hora, minuto y segundo. Los valores de los atributos se controlarán mediante métodos set/get.

Solución

Clase `Hora`

```
/* La clase Hora es muy simple y dispone de los atributos: hora, minuto y segundo.
 * Estos serán privados y, para acceder a ellos, usaremos métodos set/get.
 * Internamente vamos a utilizar el tipo byte para almacenar los atributos, pero desde
 * fuera de la clase nos interesa dar la sensación de que los atributos son int:
 * estamos ocultando la verdadera implementación.
 * No escribimos ningún constructor. */
public class Hora {
    private byte hora; //atributos de tipo byte: más que suficiente para los valores
    private byte minuto; //que tenemos que guardar
    private byte segundo;
    public int getHora() {
        return hora; //devuelve la hora
    }
    public void setSegundo(int segundo) { //los segundos están comprendidos: 0..59
        if (0 <= segundo && segundo <= 59) {
            this.segundo = (byte) segundo;
        } else {
            this.segundo = 0; //si el valor está fuera de rango lo ponemos a 0
        }
    }
    public void incrementaSegundo() {
        segundo++; //incrementamos los segundos
        if (segundo == 60) { //si los segundo alcanza un valor de 60
            segundo = 0; //reiniciamos los segundos
            minuto++; //e incrementamos los minutos
            if (minuto == 60) { //si los minutos alcanza un valor de 60
                minuto = 0; //reiniciamos los minutos
                hora++; //e incrementamos las horas
                if (hora == 24) { //si la hora alcanza un valor 24
                    hora = 0; //reiniciamos las horas
                }
            }
        }
    }
}
```

Programa principal

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Hora h = new Hora(); //creamos un objeto Hora
        System.out.println("Hora: ");
        int valor = sc.nextInt(); //leemos un valor para la hora
        h.setHora(valor); //asignamos un valor para la hora
        System.out.println("Minuto: ");
        valor = sc.nextInt(); //leemos un valor para los minutos
        h.setMinuto(valor); //asignamos un valor a los minutos
        System.out.println("Segundo: ");
```


© Ediciones Paraninfo

```
        System.out.println("Cuántos segundos quiere mostrar: ");
        int numSegundos = sc.nextInt();
        for (int i = 0; i <= numSegundos; i++) {
            //mostramos la hora
            System.out.println(h.getHora() + ":" + h.getMinuto() + ":" + h.getSegundo());
            h.incrementaSegundo(); //incrementamos la hora actual en un segundo
        }
    }
}
```

7.10. Enumerados

Los tipos enumerados sirven para definir grupos de constantes como posibles valores de una variable. Por ejemplo, `DiaDeLaSemana` sería un tipo enumerado que puede tomar solo los valores constantes: LUNES, MARTES... DOMINGO. Se define de forma parecida a una clase:

```
enum DiaDeLaSemana {
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO
}
```

En la definición usamos la palabra clave `enum` y no `class`. En un programa se accede a sus valores de la forma `DiaDeLaSemana.LUNES`, `DiaDeLaSemana.MARTES`, etcétera.

Un tipo enumerado se puede implementar en un archivo aparte —normalmente dentro del mismo paquete, aunque no es obligatorio—, como si fuera una clase o bien dentro de la definición de la clase donde se va a usar. En el primer caso, en NetBeans pulsamos con el botón derecho sobre el nombre del paquete: *New/Java Enum*.

Ahora, si queremos guardar en una variable el día de la semana que tenemos inglés —los lunes— escribiremos:

```
DiaDeLaSemana ingles = DiaDeLaSemana.LUNES;
```

```
    Sexo sexo; //declaración de un atributo del tipo enumerado
    ...
}
```

Aquí, el tipo `Sexo` solo es accesible directamente desde dentro de la propia clase. Al escribir el constructor de `Cliente`, tenemos dos opciones para el parámetro de entrada del atributo `sexo`:

1. Definirlo de tipo `String` y convertirlo en `Sexo` dentro del código del constructor.

```
Cliente(..., String sexo) {
    ...
    this.sexo = Sexo.valueOf(sexo);
}
```

2. Definirlo de tipo Sexo directamente.

```
Cliente(..., Sexo sexo) {
    ...
    this.sexo = sexo;
}
```

En el primer caso, cuando se llame al constructor, se le pasa una cadena.

```
String sexoCliente = new Scanner(System.in).next();
Cliente c = new Cliente(..., sexoCliente);
```

En segundo caso, habrá que hacer la conversión de `String` a `Sexo` antes de llamar al constructor. Dicha conversión dependerá de dónde está definido el tipo enumerado.

a) Si está definido dentro de la clase Cliente, se accede a él con el nombre de la clase,

```
Cliente c = new Cliente(..., Cliente.Sexo.valueOf(sexoCliente));
```

b) Si se ha definido en un archivo propio, aunque dentro del mismo paquete,

```
Cliente c = new Cliente(..., Sexo.valueOf(sexoCliente));
```

Los tipos enumerados se pueden definir en paquetes distintos a donde se vayan a usar. En ese caso, habrá que definirlos `public` e importarlos igual que si fueran clases.

Nota técnica

de una variable. Por ejemplo, `DiaDeLaSemana` sería un tipo enumerado que puede tomar solo los valores constantes: LUNES, MARTES... DOMINGO. Se define de forma parecida a una clase:

```
enum DiaDeLaSemana {
   LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO
}
```

En la definición usamos la palabra clave `enum` y no `class`. En un programa se accede a sus valores de la forma `DiaDeLaSemana.LUNES`, `DiaDeLaSemana.MARTES`, etcétera.

Un tipo enumerado se puede implementar en un archivo aparte —normalmente dentro del mismo paquete, aunque no es obligatorio—, como si fuera una clase o bien dentro de la definición de la clase donde se va a usar. En el primer caso, en NetBeans pulsamos con el botón derecho sobre el nombre del paquete: *New/Java Enum*.

Ahora, si queremos guardar en una variable el día de la semana que tenemos inglés —los lunes— escribiremos:

```
DiaDeLaSemana ingles = DiaDeLaSemana.LUNES;
```

Normalmente, cuando tengamos que introducir por teclado un valor de tipo enumerado, escribiremos una cadena como «LUNES» y no «DiaDeLaSemana.LUNES». Para asignarlo a una variable de tipo `DiaDeLaSemana`, tendremos que convertirla en el valor enumerado correspondiente. Para eso se usa el método `valueOf()`, que convierte la cadena «LUNES» en el valor `DiaDeLaSemana.LUNES`.

```
Scanner sc = new Scanner(System.in);
String dia = sc.nextLine(); //introducimos LUNES
DiaDeLaSemana ingles = DiaDeLaSemana.valueOf(dia);
```

Si vamos a usar un tipo enumerado exclusivamente dentro de una clase, se puede definir dentro de ella. Por ejemplo, para añadir a la clase `Cliente` el atributo `sexo` con los valores posibles HOMBRE y MUJER, definimos el tipo enumerado `Sexo` dentro de la clase:


```
        this.sexo = sexo;
    }
```

En el primer caso, cuando se llame al constructor, se le pasa una cadena.

```
String sexoCliente = new Scanner(System.in).next();
Cliente c = new Cliente(..., sexoCliente);
```

En segundo caso, habrá que hacer la conversión de `String` a `Sexo` antes de llamar al constructor. Dicha conversión dependerá de dónde está definido el tipo enumerado.

a) Si está definido dentro de la clase Cliente, se accede a él con el nombre de la clase,

```
Cliente c = new Cliente(..., Cliente.Sexo.valueOf(sexoCliente));
```

b) Si se ha definido en un archivo propio, aunque dentro del mismo paquete,

```
Cliente c = new Cliente(..., Sexo.valueOf(sexoCliente));
```

Los tipos enumerados se pueden definir en paquetes distintos a donde se vayan a usar. En ese caso, habrá que definirlos `public` e importarlos igual que si fueran clases.

Nota técnica

En la Actividad resuelta 7.7 haremos uso de las clases `LocalDate` y `LocalDateTime`, que se usan para gestionar la fecha y la hora.

Debido a la extensión de la API de Java es imposible detallar todas y cada una de las clases que utilizamos. En la web de la Editorial Paraninfo puedes encontrar un anexo con el uso de las clases que manejan las fechas y horas en Java.

Actividad resuelta 7.7

Diseñar la clase `Texto` que gestiona una cadena de caracteres con algunas características:

- La cadena de caracteres tendrá una longitud máxima que se especifica en el constructor.

- Permite añadir un carácter al principio o al final, siempre y cuando no se exceda la longitud máxima, es decir, exista espacio disponible.
- Igualmente, permite añadir una cadena, al principio o al final del texto, siempre y cuando no se rebase el tamaño máximo establecido.
- Es necesario saber cuántas vocales (mayúsculas y minúsculas) hay en el texto.
- Cada objeto de tipo `Texto` tiene que conocer la fecha en la que se creó, así como la fecha y hora de la última modificación efectuada.
- Deberá existir un método que muestre la información que gestiona cada texto.

Solución

Clase `Texto`

```
import java.time.LocalDate;
import java.time.LocalDateTime;

/* La clase Texto contendrá:
 *  - un String (donde guardaremos la cadena de caracteres)
 *  - un número entero que indicará la longitud máxima del texto
 *  - fecha de creación del texto
 *  - y la fecha y hora de la última modificación */
public class Texto {
   private String cad; //cadena de caracteres
   LocalDate creacion;
   LocalDateTime ultimaModificacion;
   private final int LONGITUD_MAX; //del texto. Una vez asignado no varía
   static final String VOCALES = "aeiouáéíóúü"; //cadena constante y estática que
   //contiene todas las posibles vocales en minúsculas

   public Texto(int longitudMax) {
      cad = ""; //cad referencia un objeto String con valor "", no se puede usar
      //cad = null, en este caso cad no referencia ningún objeto y no es posible
```



```
//Añade un carácter al comienzo del texto, siempre y cuando quede sitio
public void addPrincipio(char c) {
   if (LONGITUD_MAX > cad.length()) {
      cad = c + cad;
      ultimaModificacion = LocalDateTime.now();
   }
}

//Añade una cadena al comienzo del texto, siempre y cuando quede sitio
public void addPrincipio(String c) {
   if (LONGITUD_MAX >= cad.length() + c.length()) {
      cad = c + cad;
      ultimaModificacion = LocalDateTime.now();
   }
}

public void mostrar() {
   System.out.println("Texto creado el " + creacion);
   System.out.println("Última modificación: " + ultimaModificacion);
   System.out.println(cad);
}

//Devuelve el número de vocales presentes en el texto
public int numVocales() {
   int voc = 0; // número de vocales del texto
   for (int i = 0; i < cad.length(); i++) {
      if (esVocal(cad.charAt(i))) {
         voc++;
      }
   }
   return (voc);
}

//Comprueba si el carácter pasado es una vocal: mayúscula/minúscula/acentuada
private boolean esVocal(char c) {
```

```
 *  - un número entero que indicará la longitud máxima del texto
 *  - fecha de creación del texto
 *  - y la fecha y hora de la última modificación */
public class Texto {
   private String cad; //cadena de caracteres
   LocalDate creacion;
   LocalDateTime ultimaModificacion;
   private final int LONGITUD_MAX; //del texto. Una vez asignado no varía
   static final String VOCALES = "aeiouáéíóúü"; //cadena constante y estática que
   //contiene todas las posibles vocales en minúsculas

   public Texto(int longitudMax) {
      cad = ""; //cad referencia un objeto String con valor "", no se puede usar
      // cad = null, en este caso cad no referencia ningún objeto y no es posible
      //usar sus métodos
      this.LONGITUD_MAX = longitudMax;
      creacion = LocalDate.now();
      ultimaModificacion = null; //aún no se ha modificado nada
   }

   //Añade un carácter al final del texto, siempre y cuando quede sitio
   public void addFinal(char c) {
      if (LONGITUD_MAX > cad.length()) {
         cad = cad + c; //concatena el carácter al final
         ultimaModificacion = LocalDateTime.now();
      }
   }

   //Añade una cadena al final del texto, siempre y cuando quede sitio
   public void addFinal(String c) {
      if (LONGITUD_MAX >= cad.length() + c.length()) {
         cad = cad + c;
         ultimaModificacion = LocalDateTime.now();
      }
   }
```


© Ediciones Paraninfo

```
   //Devuelve el número de vocales presentes en el texto
   public int numVocales() {
      int voc = 0; // número de vocales del texto
      for (int i = 0; i < cad.length(); i++) {
         if (esVocal(cad.charAt(i))) {
            voc++;
         }
      }
      return (voc);
   }

   //Comprueba si el carácter pasado es una vocal: mayúscula/minúscula/acentuada
   private boolean esVocal(char c) {
      boolean vocal = false;

      c = Character.toLowerCase(c); //convertimos el carácter a minúscula
      if (VOCALES.indexOf(c) != -1) {//buscamos el carácter (en minúscula) en
         vocal = true;              //las posibles vocales
      }
      return (vocal);
   }
}
```

Programa principal

```
// Creamos un objeto Texto para probar su funcionamiento.
public class Main {
   public static void main(String[] args) {
      Texto t = new Texto(5);
      t.addPrincipio("HO");
      t.addPrincipio(';');
      t.addFinal("LÁ");
      t.addFinal('X'); // este carácter no cabe en el texto. No se añade.
      t.mostrar();
      System.out.println("Número de vocales: " + t.numVocales());
   }
}
```

Actividad resuelta 7.8

Definir una clase que permita controlar un sintonizador digital de emisoras FM; concretamente, se desea dotar al controlador de una interfaz que permita subir (up) o bajar (down) la frecuencia (en saltos de 0,5 MHz) y mostrar la frecuencia sintonizada en un momento dado (display). Supondremos que el rango de frecuencias para manejar oscila entre los 80 MHz y los 108 MHz y que, al inicio, el controlador sintonice la frecuencia indicada en el constructor o 80 MHz por defecto. Si durante una operación de subida o bajada se sobrepasa uno de los dos límites, la frecuencia sintonizada debe pasar a ser la del extremo contrario. Escribir un pequeño programa principal para probar su funcionamiento.

Solución

Clase SintonizadorFM

```
/* La clase tiene un atributo real que almacena la frecuencia a la que estamos
 * sintonizando, junto a los métodos necesarios para utilizar el sintonizador. */
public class SintonizadorFM {
double frecuencia;
   // constructor que permite asignar una frecuencia inicial
   SintonizadorFM(double frecuenciaInicial) {
      // la frecuencia inicial debe encontrarse en el rango [80 - 108]
      if (frecuenciaInicial < 80) {
         frecuencia = 80; // MHz
      } else if (frecuenciaInicial > 108) {
         frecuencia = 108; //MHz
      } else {
         frecuencia = frecuenciaInicial;
      }
   }
   SintonizadorFM() { //constructor
      this(80); // MHz. Frecuencia inicial por defecto.
      // Otra forma sería inicializar el valor por defecto directamente:
      // frecuencia = 80;
   }
   public double down() {
      frecuencia -= 0.5; //bajamos la frecuencia 0.5 MHz
      comprobarRango(); //comprobamos si la nueva frecuencia está en el rango permitido
      return (frecuencia);
   }
   public double up() {
```

Programa principal

```
// Probamos el uso del sintonizador de FM
public class Main {
   public static void main(String[] args) {
      // ejemplo de funcionamiento
      SintonizadorFM a, b;
      a = new SintonizadorFM(107);
      a.up(); a.up(); a.up(); a.up(); // subimos un total de 2 MHz
      a.display(); // debe mostrar 80.5 MHz
      b = new SintonizadorFM(80.5);
      b.down(); b.down(); b.down(); //bajamos 1.5 MHz
      b.display(); // debe mostrar 107.5 MHz
      a = new SintonizadorFM(205); //frecuencia fuera de rango. Deba ajustarse
      a.display(); //debe mostrar 108.0 MHz
   }
}
```

Actividad resuelta 7.9

Modelar una casa con muchas bombillas, de forma que cada bombilla se pueda encender o apagar individualmente. Para ello, hacer una clase Bombilla con una variable privada que indique si está encendida o apagada, así como un método que nos diga el estado de una bombilla concreta. Además, queremos poner un interruptor general, de forma que si este se apaga, todas las bombillas quedan apagadas. Cuando el interruptor general se activa, las bombillas vuelven a estar encendidas o apagadas, según estuvieran antes. Cada bombilla se enciende y se apaga individualmente, pero solo responde que está encendida si su interruptor particular está activado y además hay luz general.

Solución

Clase Bombilla

```
/* La clase Bombilla se implementa con un indicador de estado (apagada/encendida), que
 * será individual para cada bombilla (para cada objeto bombilla). Además, el interruptor
 * general (que afecta a todas las bombillas) se implementa con un atributo estático,
 * cuyo valor será el mismo para todos los objetos de la clase. */
public class Bombilla {
```

```
SintonizadorFM(double frecuenciaInicial) {
   // la frecuencia inicial debe encontrarse en el rango [80 - 108]
   if (frecuenciaInicial < 80) {
      frecuencia = 80; // MHz
   } else if (frecuenciaInicial > 108) {
      frecuencia = 108; //MHz
   } else {
      frecuencia = frecuenciaInicial;
   }
}
SintonizadorFM() { //constructor
   this(80); // MHz. Frecuencia inicial por defecto.
   // Otra forma sería inicializar el valor por defecto directamente:
   // frecuencia = 80;
}
public double down() {
   frecuencia -= 0.5; //bajamos la frecuencia 0.5 MHz
   comprobarRango(); //comprobamos si la nueva frecuencia está en el rango permitido
   return (frecuencia);
}
public double up() {
   frecuencia += 0.5; //subimos la frecuencia
   comprobarRango(); //y comprobamos el rango
   return (frecuencia);
}
public void display() {
   System.out.println("Sintonizando: " + frecuencia + " MHz"); //mostramos
}
//método de uso interno que se encarga de comprobar que la frecuencia se encuentre
//en el rango 80..108. En caso de que la frecuencia esté fuera de rango la ajusta
private void comprobarRango() {
   if (frecuencia < 80) { //si al bajar la frecuencia es menor que el límite inferior
      frecuencia = 108; //asignamos el límite superior
   } else if (frecuencia > 108) { //si sobrepasamos el límite superior
      frecuencia = 80; //colocamos la frecuencia en el valor menor
   }
}
```


© Ediciones Paraninfo

Actividad resuelta 7.9

Modelar una casa con muchas bombillas, de forma que cada bombilla se pueda encender o apagar individualmente. Para ello, hacer una clase `Bombilla` con una variable privada que indique si está encendida o apagada, así como un método que nos diga el estado de una bombilla concreta. Además, queremos poner un interruptor general, de forma que si este se apaga, todas las bombillas quedan apagadas. Cuando el interruptor general se activa, las bombillas vuelven a estar encendidas o apagadas, según estuvieran antes. Cada bombilla se enciende y se apaga individualmente, pero solo responde que esté encendida si su interruptor particular está activado y además hay luz general.

Solución

Clase `Bombilla`

```
/* La clase Bombilla se implementa con un indicador de estado (apagada/encendida), que
 * será individual para cada bombilla (para cada objeto bombilla). Además, el interruptor
 * general (que afecta a todas las bombillas) se implementa con un atributo estático,
 * cuyo valor será el mismo para todos los objetos de la clase. */
public class Bombilla {
   public static boolean interruptorGeneral = true; // atributo estático
   private boolean interruptor; //interruptor (estado) que posee cada bombilla
   public Bombilla() {
      interruptor = false; // inicialmente la nueva bombilla está apagada
   }
   public void enciende() {
      interruptor = true; // activamos el interruptor (a true)
   }
   public void apaga() {
     interruptor = false; // desactivamos el interruptor
   }
   public boolean estado() {
      return interruptorGeneral && interruptor;
     //el estado es true si el interruptor de la bombilla y el general están activados
   }
  //Devuelve una cadena con el estado de la bombilla
```

```
    public String muestraEstado() {
        return estado() ? "Encendida" : "Apagada";
        //dependiendo del estado se devuelve la cadena "Encendida" o "Apagada"
    }
}
```

Programa principal

```
public class Main {
    public static void main(String[] args) {
        // vemos un ejemplo de funcionamiento
        Bombilla b1, b2;
        b1 = new Bombilla();
        b2 = new Bombilla();
        b1.enciende();
        b2.apaga();
        System.out.println("b1: " + b1.muestraEstado());
        System.out.println("b2: " + b2.muestraEstado());
        Bombilla.interrupcorGeneral = false; // cortamos la luz
        System.out.println("\nCortamos la luz general");
        System.out.println("b1: " + b1.muestraEstado());
        System.out.println("b2: " + b2.muestraEstado());
        Bombilla.interrupcorGeneral = true; // activamos la luz
        System.out.println("\nActivamos la luz general");
        System.out.println("b1: " + b1.muestraEstado());
        System.out.println("b2: " + b2.muestraEstado());
    }
}
```

Actividad resuelta 7.10

Hemos recibido el encargo de un cliente para definir los paquetes y las clases necesarias (solo implementar los atributos y los constructores) para gestionar una empresa ferroviaria, en la que se distinguen dos grandes grupos: el personal y la maquinaria. En el primero se ubican todos los empleados de la empresa, que se clasifican en tres grupos: los maquinistas, los mecánicos y los jefes de estación. De cada uno de ellos es necesario guardar:

- Maquinistas: su nombre, DNI, sueldo y el rango que tienen adquirido.

Todas las clases correspondientes al personal (`Maquinista`, `Mecanico` y `JefeEstacion`) serán de uso público. Entre las clases relativas a la maquinaria solo será posible construir, desde clases externas, objetos de tipo `Tren` y de tipo `Locomotora`. La clase `Vagon` será solo visible por sus clases vecinas.

Solución

Clase `Maquinista`

```
package personal;
public class Maquinista {
    String nombre;
    String dni;
    double sueldo;
    String rango;
    public Maquinista(String nombre, String dni, double sueldo, String rango) {
        this.nombre = nombre;
        this.dni = dni;
        this.sueldo = sueldo;
        this.rango = rango;
    }
}
```

Clase `Mecanico`

```
package personal;
public class Mecanico {
    String nombre;
    String telefono;
    enum Especialidad {FRENOS, HIDRAULICA, ELECTRICIDAD, MOTOR} //enumerado
    Especialidad especialidad;
    public Mecanico(String nombre, String telefono, String especialidad) {
        this.nombre = nombre;
        this.telefono = telefono;
        this.especialidad = Especialidad.valueOf(especialidad);//pasa de String a
                                                               //enumerado
    }
}
```

Clase `JefeEstacion`

```
        System.out.println("b2: " + b2.muestraEstado());
        Bombilla.interruptorGeneral = true; // activamos la luz
        System.out.println("\nActivamos la luz general");
        System.out.println("b1: " + b1.muestraEstado());
        System.out.println("b2: " + b2.muestraEstado());
    }
}
```

Actividad resuelta 7.10

Hemos recibido el encargo de un cliente para definir los paquetes y las clases necesarias (solo implementar los atributos y los constructores) para gestionar una empresa ferroviaria, en la que se distinguen dos grandes grupos: el personal y la maquinaria. En el primero se ubican todos los empleados de la empresa, que se clasifican en tres grupos: los maquinistes, los mecánicos y los jefes de estación. De cada uno de ellos es necesario guardar:

- Maquinistas: su nombre, DNI, sueldo y el rango que tienen adquirido.
- Mecánicos: su nombre, teléfono (para contactar en caso de urgencia) y en qué especialidad desarrollan su trabajo (esta puede ser: frenos, hidráulica, electricidad o motor).
- Jefes de estación: su nombre, DNI y la fecha en la que fue nombrado jefe de estación.

En la parte de maquinaria podemos encontrar trenes, locomotoras y vagones. De cada uno de ellos hay que considerar:

- Vagones: tienen un número que los identifica, una carga máxima (en kilos), la carga actual y el tipo de mercancía con el que están cargados.
- Locomotoras: disponen de una matrícula (que las identifica), la potencia de sus motores y una antigüedad (año de fabricación). Además, cada locomotora tiene asignado un mecánico que se encarga de su mantenimiento.
- Trenes: están formados por una locomotora y un máximo de 5 vagones. Cada tren tiene asignado un maquinista que es responsable de él.


```
    }
}
```

Clase `Mecanico`

```
package personal;
public class Mecanico {
    String nombre;
    String telefono;
    enum Especialidad {FRENOS, HIDRAULICA, ELECTRICIDAD, MOTOR} //enumerado
    Especialidad especialidad;
    public Mecanico(String nombre, String telefono, String especialidad) {
        this.nombre = nombre;
        this.telefono = telefono;
        this.especialidad = Especialidad.valueOf(especialidad);//pasa de String a
                                                               //enumerado
    }
}
```

Clase `JefeEstacion`

```
package personal;
import java.util.DateTime;

public class JefeEstacion {
    String nombre;
    String dni;
    DateTime nombramiento;

  public JefeEstacion(String nombre, String dni, DateTime nombramiento) {
        this.nombre = nombre;
        this.dni = dni;
        this.nombramiento = nombramiento;
    }
}
```

```
package maquinaria;
class Vagon { //visibilidad por defecto. Solo visible por clases vecinas
   int numIdentificativo;
   int cargaMax;
   int cargaActual;
   String mercancia;
   public Vagon(int numIdentificativo, int cargaMax, int cargaActual, String mercancia) {
      this.numIdentificativo = numIdentificativo;
      this.cargaMax = cargaMax;
      this.cargaActual = cargaActual;
      this.mercancia = mercancia;
   }
}
```

Clase `Locomotora`

```
package maquinaria;
import personal.Mecanico;
   public class Locomotora {
   String matricula;
   int potencia;
   int añoFabricacion;
   Mecanico mec;
   public Locomotora(String matricula, int potencia, int añoFabricacion, Mecanico mec) {
      this.matricula = matricula;
      this.potencia = potencia;
      this.añoFabricacion = añoFabricacion;
      this.mec = mec;
   }
}
```

Clase `Tren`

```
package maquinaria;
import personal.Maquinista;
public class Tren {
   Locomotora locomotora;
   Vagon vagones[];
   Maquinista responsable;
   private int numVagones; //número de vagones que forman el tren
   public Tren(Locomotora locomotora, Maquinista responsable) {
      this.locomotora = locomotora;
      this.responsable = responsable;
      vagones = new Vagon[5]; //creamos la tabla de tamaño 5, pero no se
```

Nota técnica

Los *wrappers* o envoltorios son clases que internamente contienen un dato de tipo primitivo, lo que proporciona una forma de trabajar con estos como objetos. Cada tipo primitivo tiene su correspondiente clase envoltorio: `Integer` para el tipo `int`, `Double` para el tipo `double`, `Character` para el tipo `char`, etcétera.

En la web de la Editorial Paraninfo puedes encontrar un anexo con la descripción de los *wrappers*.

Actividad resuelta 7.11

Las listas son estructuras dinámicas de datos donde se pueden insertar o eliminar elementos de un determinado tipo sin limitación de espacio.

Implementar la clase `Lista` correspondiente a una lista de números de la clase `Integer`. Los números se guardarán en una tabla que se redimensionará con las inserciones y eliminaciones, aumentando o disminuyendo la capacidad de la lista según el caso.

Entre los métodos de la clase, se incluirán las siguientes tareas:

- Un constructor que inicialice la tabla con un tamaño 0.
- Obtener el número de elementos insertados en la lista.
- Insertar un número al final de la lista.
- Insertar un número al principio de la lista.
- Insertar un número en un lugar de la lista cuyo índice, que es el de la tabla, se pasa como parámetro.
- Añadir al final de la lista los elementos de otra lista que se pasa como parámetro.
- Eliminar un elemento cuyo índice en la lista se pasa como parámetro.
- Obtener el elemento cuyo índice se pasa como parámetro.
- Buscar un número en la lista, devolviendo el índice del primer lugar donde se encuentre. Si no está, devolverá –1.
- Mostrar los elementos de la lista por consola.

Solución

```
    int potencia;
    int añoFabricacion;
    Mecanico mec;
    public Locomotora(String matricula, int potencia, int añoFabricacion, Mecanico mec) {
        this.matricula = matricula;
        this.potencia = potencia;
        this.añoFabricacion = añoFabricacion;
        this.mec = mec;
    }
}
```

Clase `Tren`

```
package maquinaria;
import personal.Maquinista;
public class Tren {
    Locomotora locomotora;
    Vagon vagones[];
    Maquinista responsable;
    private int numVagones; //número de vagones que forman el tren
    public Tren(Locomotora locomotora, Maquinista responsable) {
        this.locomotora = locomotora;
        this.responsable = responsable;
        vagones = new Vagon[5]; //creamos la tabla de tamaño 5, pero no se
                                //crea ningún objeto de tipo Vagón
        numVagones = 0; //por ahora no hay vagones enganchados al tren
    }
    /* Al ser la clase Vagon no visible por clases externas, será la clase Tren la
     * que se encargue de construir el objeto a partir de los datos que nos pasen. */
    public void enganchaVagon(int cargaMax, int cargaActual, String mercancia) {
        if (numVagones >= 5) {
            System.out.println("El tren no admite más vagones");
        } else {
            Vagon v = new Vagon(numVagones, cargaMax, cargaActual, mercancia);
            vagones[numVagones] = v; //el vagón pasado ocupa el último lugar
            numVagones++; //ahora tenemos un vagón más enganchado al tren
        }
    }
}
```


© Ediciones Paraninfo

Los números se guardarán en una tabla que se redimensionará con las inserciones y eliminaciones, aumentando o disminuyendo la capacidad de la lista según el caso.

Entre los métodos de la clase, se incluirán las siguientes tareas:

- Un constructor que inicialice la tabla con un tamaño 0.
- Obtener el número de elementos insertados en la lista.
- Insertar un número al final de la lista.
- Insertar un número al principio de la lista.
- Insertar un número en un lugar de la lista cuyo índice, que es el de la tabla, se pasa como parámetro.
- Añadir al final de la lista los elementos de otra lista que se pasa como parámetro.
- Eliminar un elemento cuyo índice en la lista se pasa como parámetro.
- Obtener el elemento cuyo índice se pasa como parámetro.
- Buscar un número en la lista, devolviendo el índice del primer lugar donde se encuentre. Si no está, devolverá –1.
- Mostrar los elementos de la lista por consola.

Solución

Clase `Lista`

```
import java.util.Arrays;

/* Implementamos las listas con tablas de tipo Integer, que iremos
 * redimensionando según vaya haciendo falta. El índice de un elemento en la
 * lista coincide con el índice del lugar que ocupa en la tabla. */
public class Lista {
    Integer[] tabla;

    public Lista() {
        tabla = new Integer[0];
    }
```

```
void insertarPrincipio(Integer nuevo) {
   tabla = Arrays.copyOf(tabla, tabla.length + 1);
   System.arraycopy(tabla, 0, tabla, 1, tabla.length - 1);
   tabla[0] = nuevo;
}

void insertarFinal(Integer nuevo) {
   tabla = Arrays.copyOf(tabla, tabla.length + 1);
   tabla[tabla.length - 1] = nuevo;
}

void insertarFinal(Lista otralista) {
   int tamIni = tabla.length;//tamaño inicial tabla
   tabla = Arrays.copyOf(tabla, tabla.length + otraLista.tabla.length);
   System.arraycopy(otraLista.tabla, 0, tabla, tamIni, otraLista.tabla.length);
}

//El primer parámetro es el índice del lugar donde queremos insertar
//el valor del segundo parámetro
void insertar(int posicion, Integer nuevo) {
   tabla = Arrays.copyOf(tabla, tabla.length + 1);
   System.arraycopy(tabla, posicion, tabla, posicion + 1,
      tabla.length - posicion - 1);
   tabla[posicion] = nuevo;
}

//Se elimina el elemento correspondiente a indice y se devuelve
Integer eliminar(int indice) {
   Integer eliminado = null;
   if (indice >= 0 && indice < tabla.length) {
      eliminado = tabla[indice];
      for (int i = indice + 1; i < tabla.length; i++) {
         tabla[i - 1] = tabla[i];
      }
      tabla = Arrays.copyOf(tabla, tabla.length - 1);
   }
   return eliminado;
}

/* Al siguiente método le pasaremos un índice y nos devolverá el elemento
   }

   //El número de elementos de la lista es el número de elementos de la tabla
   public int numeroElementos() {
      return tabla.length;
   }

   //Muestra por consola el contenido de la lista
   public void mostrar() {
      System.out.println("Lista: " + Arrays.toString(tabla));
   }
}
```

Programa principal

```
public class Main {
   //prueba de los métodos de la clase Lista
   public static void main(String[] args) {
      Lista l1 = new Lista();
      Lista l2 = new Lista();
      l1.insertarFinal(4);
      l1.insertarFinal(5);
      l1.insertarFinal(6);
      l1.mostrar();
      l1.insertarPrincipio(3);
      l1.insertarPrincipio(2);
      l1.insertarPrincipio(1);
      l1.mostrar();
      l1.insertar(2, 99);
      l1.mostrar();
      l1.eliminar(2);
      l1.mostrar();
      System.out.println(l1.buscar(4));
      l2.insertarFinal(10);
      l2.insertarFinal(20);
      l2.insertarFinal(30);
      l2.insertarFinal(40);
      l2.insertarFinal(50);
      l2.mostrar();
      l1.insertarFinal(l2);
      l1.mostrar();
```

```
        System.arraycopy(tabla, posicion, tabla, posicion + 1,
            tabla.length - posicion - 1);
        tabla[posicion] = nuevo;
    }

    //Se elimina el elemento correspondiente a indice y se devuelve
    Integer eliminar(int indice) {
        Integer eliminado = null;
        if (indice >= 0 && indice < tabla.length) {
            eliminado = tabla[indice];
            for (int i = indice + 1; i < tabla.length; i++) {
                tabla[i - 1] = tabla[i];
            }
            tabla = Arrays.copyOf(tabla, tabla.length - 1);
        }
        return eliminado;
    }

    /* Al siguiente método le pasaremos un índice y nos devolverá el elemento
       correspondiente de la tabla sin modificarla. En el caso de que el índice no
       sea válido, devolverá null, con lo cual evitamos que el programa aborte. */
    Integer get(int indice) {
        Integer resultado = null;
        if (indice >= 0 && indice < tabla.length) {//indice válido
            resultado = tabla[indice];
        }
        return resultado;
    }

    int buscar(Integer claveBusqueda) {
        int indice = -1;
        for (int i = 0; i < tabla.length && indice == -1; i++) {
            if (tabla[i].equals(claveBusqueda)) {//no vale tabla[i]==claveBusqueda
                indice = i;
            }
        }
        return indice;
```


© Ediciones Paraninfo

```
        l1.insertarFinal(6);
        l1.mostrar();
        l1.insertarPrincipio(3);
        l1.insertarPrincipio(2);
        l1.insertarPrincipio(1);
        l1.mostrar();
        l1.insertar(2, 99);
        l1.mostrar();
        l1.eliminar(2);
        l1.mostrar();
        System.out.println(l1.buscar(4));
        l2.insertarFinal(10);
        l2.insertarFinal(20);
        l2.insertarFinal(30);
        l2.insertarFinal(40);
        l2.insertarFinal(50);
        l2.mostrar();
        l1.insertarFinal(l2);
        l1.mostrar();
    }
}
```

Actividad resuelta 7.12

Añadir a la clase `Lista` el método estático:

```
Lista concatena(Lista l1, Lista l2)
```

que construye y devuelve una lista que contiene, en el mismo orden, una copia de todos los elementos de `l1` y `l2`.

Solución

Clase `Lista`

```
public class Lista {
    ... //resto de implementación de Lista
```

```
//Crearemos un objeto Lista donde insertaremos todos los elementos de l1 y l2.
static Lista concatena(Lista l1, Lista l2) {
   Lista resultado = new Lista(); //objeto Lista que contendrá la concatenación
   for (Integer e : l1.tabla) { //recorremos los elementos de l1 e insertamos
      resultado.insertarFinal(e); //insertamos al final para mantener el orden
   }
   for (Integer e : l2.tabla) { //hacemos lo mismo con l2.
      resultado.insertarFinal(e);
   }
   return resultado;
}
```

Programa principal

```
public class Main {
   //prueba del método estático concatena() de Lista
   public static void main(String[] args) {
      Lista l1 = new Lista();
      Lista l2 = new Lista();
      l1.insertarFinal(1); l1.insertarFinal(1); l1.insertarFinal(2);
      l1.insertarFinal(3);
      l2.insertarFinal(10); l2.insertarFinal(20); l2.insertarFinal(30);
      Lista concatenacion = Lista.concatena(l1, l2);
      concatenacion.mostrar();
   }
}
```

Actividad resuelta 7.13

Una pila es una estructura dinámica de datos donde los elementos se insertan (se apilan) y se retiran (se desapilan) siguiendo la norma de que el último que se apila será el primero en desapilarse, como ocurre con una pila de platos. Cuando vamos a retirar un plato de


```
   //desapilamos extrayendo el elemento de la cima. Si la pila está vacía, es
   //porque la lista también lo está y devuelve null
   Integer desapilar() {
      return lista.eliminar(lista.tabla.length - 1);
   }
   public void mostrar() {
      lista.mostrar();
   }
}
```

Programa principal

```
public class Main {
   //programa principal para probar la clase Pila
   public static void main(String[] args) {
      Pila p = new Pila();
      System.out.println(p.desapilar());//muestra null, ya que p está vacía
      for (int i = 0; i < 10; i++) { //apilamos los números del 0 al 9
         p.apilar(i);
      }
      Integer num = p.desapilar(); //desapilamos
      while (num != null) { //mientras la pila no esté vacía
         System.out.print(num + " "); //mostramos el elemento desapilado
         num = p.desapilar(); //y volvemos a desapilar
      }
   }
}
```

Actividad resuelta 7.14

Implementar el método no estático

```
void insertarFinal(int nuevo)
```

```
        Lista concatenacion = lista.concatena(l1, l2);
        concatenacion.mostrar();
    }
}
```

Actividad resuelta 7.13

Una pila es una estructura dinámica de datos donde los elementos se insertan (se apilan) y se retiran (se desapilan) siguiendo la norma de que el último que se apila será el primero en desapilarse, como ocurre con una pila de platos. Cuando vamos a retirar un plato de una pila a nadie se le ocurre tirar de uno de los de abajo; retiramos (desapilamos) el que está encima de todos, que fue el último en ser apilado. Se llama cima de la pila al último elemento apilado (o al primer elemento para desapilar). Los métodos fundamentales de una pila son `apilar()` y `desapilar()`.

Implementar la clase `Pila` para números `Integer`, donde se usa una lista (un objeto de la clase Lista implementada en la Actividad resuelta 7.11) para guardar los elementos apilados.

Solución

Clase `Pila`

```
/* Vamos a implementar una estructura de pila para Integer usando objetos de la clase
 * Lista para guardar los datos que se apilan. Por razón de eficiencia, la cima será el
 * final de la lista, evitando así mover los datos apilados previamente. */
public class Pila {
  private Lista lista; //objeto donde almacenaremos los datos
  public Pila() {
      lista = new Lista(); //creamos un objeto Lista
  }
  //apilamos añadiendo el elemento al final de la lista
  void apilar(Integer elemento) {
      lista.insertarFinal(elemento);
  }
```



```
        System.out.println(num + " "); //Mostramos el elemento desapilado
        num = p.desapilar(); //y volvemos a desapilar
    }
  }
}
```

Actividad resuelta 7.14

Implementar el método no estático

```
void insertarFinal(int nuevo)
```

que inserta un número entero al final de `tablaEnteros[]`, que es un atributo no estático de la clase `Main`. Escribir un programa que inicialice la tabla con los números del 1 al 10 y después la muestre por consola.

Solución

```
import java.util.Arrays;
/* La clase Main puede tener atributos y métodos no estáticos, aunque no podemos
 * invocarlos directamente desde el método main(), ya que este es static, pero sí a
 * través de un objeto de la propia clase Main. */
public class Main {
   int[] tablaEnteros = new int[0]; //atributo no estático de Main
   public static void main(String[] args) {
      Main m = new Main(); //creamos un objeto de la clase Main con el constructor
      //por defecto
      for (int i = 0; i < 10; i++) {
         m.insertarFinal(i + 1);
      }
      System.out.println("tabla: " + Arrays.toString(m.tablaEnteros));
   }
   void insertarFinal(int nuevo) {//método no estático de Main
      tablaEnteros = Arrays.copyOf(tablaEnteros, tablaEnteros.length + 1);
      tablaEnteros[tablaEnteros.length - 1] = nuevo;
   }
}
```

Actividades de comprobación

7.1. Dos clases se consideran vecinas siempre y cuando:

a) Sean visibles.
b) Ambas dispongan del mismo número de constructores.
c) Pertenezcan al mismo paquete.
d) Todo lo anterior ha de cumplirse para que dos clases sean vecinas.

7.2. Un miembro cuyo modificador de acceso es `private` será visible desde:

a) Todas las clases vecinas.
b) Todas las clases externas.
c) Es indistinto el paquete, pero será visible siempre que se importe la clase que lo contiene.
d) Ninguna de las respuestas anteriores.

7.3. Si desde un constructor queremos invocar a otro constructor de la misma clase, tendremos que usar:

a) `set()`.
b) `get()`.
c) `this()`.
d) `this`.

7.4. Si por error dejamos un objeto sin ninguna referencia, siempre podremos volver a referenciarlo mediante:

a) La referencia `this`.
b) La referencia `null`.
c) Utilizando `new`.
d) Es imposible.

7.5. ¿Qué hace el operador `new`?

a) Construye un objeto, invoca al constructor y devuelve su referencia.
b) Construye un objeto, comprueba que su clase esté importada y devuelve su referencia.
c) Busca en la memoria un objeto del mismo tipo, invoca al constructor y devuelve su referencia.
d) Busca en memoria un objeto del mismo tipo y devuelve su referencia.

tendremos que usar:

a) `set()`.
b) `get()`.
c) `this()`.
d) `this`.

7.4. **Si por error dejamos un objeto sin ninguna referencia, siempre podremos volver a referenciarlo mediante:**

a) La referencia `this`.
b) La referencia `null`.
c) Utilizando `new`.
d) Es imposible.

7.5. **¿Qué hace el operador `new`?**

a) Construye un objeto, invoca al constructor y devuelve su referencia.
b) Construye un objeto, comprueba que su clase esté importada y devuelve su referencia.
c) Busca en la memoria un objeto del mismo tipo, invoca al constructor y devuelve su referencia.
d) Busca en memoria un objeto del mismo tipo y devuelve su referencia.

7.6. **Cuando hablamos de miembros de una clase, nos estamos refiriendo a:**

a) Todos los atributos.
b) Todos los métodos.
c) Todos los atributos y métodos, indistintamente de los modificadores de acceso utilizados.
d) Todos los atributos y métodos que son visibles por sus clases vecinas.

7.7. **En la definición de una clase, los únicos modificadores de acceso que se pueden utilizar son:**

a) `public`.
b) `public` y el modificador de acceso por defecto.
c) `public`, el modificador de acceso por defecto y `private`.
d) El modificador `class`.

© Ediciones Paraninfo

7.8. ¿Qué diferencia un atributo estático definido en una clase de otro que no lo es?

a) El atributo estático es visible por todas las clases vecinas, mientras que el no estático solo será visible para las clases que usen importación.
b) Solo existe una copia del atributo estático en la clase, mientras que el atributo no estático tendrá una copia en cada uno de los objetos.
c) Existe una copia del atributo estático en todos y cada uno de los objetos, mientras que del atributo no estático solo existe una copia en la clase.
d) Ambos disponen de copias en cada objeto, pero el atributo no estático es accesible mediante la clase y el no estático es accesible mediante los objetos.

7.9. ¿Qué efecto tiene las siguientes líneas de código?

```
Cliente c;
c.nombre = "Pepita";
```

a) Inicializa el atributo nombre de `Cliente` con el valor «Pepita».
b) Invoca al constructor y posteriormente asigna el valor «Pepita» al atributo nombre, siempre y cuando este sea público.
c) Si el atributo `nombre` es público, se le asigna un valor, pero si el atributo es privado, producirá un error.
d) Siempre produce un error.

7.10. La ocultación de atributos puede definirse como:

a) El proceso en el que un atributo pasa de ser público a privado.
b) El proceso en el que se define una variable local (en un método) con el mismo identificador que un atributo.
c) El proceso en el que un atributo estático deja de serlo.
d) Todas las respuestas anteriores son correctas.

ACTIVIDADES FINALES 7. CLASES

7.14. Crea una clase que sea capaz de mostrar el importe de un cambio, por ejemplo, al realizar una compra, con el menor número de monedas y billetes posibles.

7.15. Diseña la clase `Calendario` que representa una fecha concreta (año, mes y día). La clase debe disponer de los métodos:

- `Calendario(int año, int mes, int dia)`: que crea un objeto con los datos pasados como parámetros, siempre y cuando, la fecha que representen sea correcta.
- `void incrementarDia()`: que incrementa en un día la fecha del calendario.
- `void incrementarMes()`: que incrementa en un mes la fecha del calendario.
- `void incrementarAño(int cantidad)`: que incrementa la fecha del calendario en el número de años especificados. Ten en cuenta que el año 0 no existió.
- `void mostrar()`: muestra la fecha por consola.
- `boolean iguales(Calendario otraFecha)`: que determina si la fecha invocante y la que se pasa como parámetro son iguales o distintas.

Por simplicidad, solo tendremos en consideración que existen meses con distinto número de días, pero no tendremos en cuenta los años bisiestos.

7.16. Escribe la clase `Punto` que representa un punto en el plano (con un componente x y un componente y), con los métodos:

- `Punto(double x, double y)`: construye un objeto con los datos pasados como parámetros.
- `void desplazaX(double dx)`: incrementa el componente x en la cantidad `dx`.
- `void desplazaY(double dy)`: incrementa el componente y en la cantidad `dy`.
- `void desplaza(double dx, double dy)`: desplaza ambos componentes según las cantidades `dx` (en el eje x) y `dy` (en el componente y).
- `double distanciaEuclidea(Punto otro)`: calcula y devuelve la distancia euclídea entre el punto invocante y el punto `otro`.

producirá un error.

d) Siempre produce un error.

7.10. **La ocultación de atributos puede definirse como:**

a) El proceso en el que un atributo pasa de ser público a privado.

b) El proceso en el que se define una variable local (en un método) con el mismo identificador que un atributo.

c) El proceso en el que un atributo estático deja de serlo.

d) Todas las respuestas anteriores son correctas.

Actividades de aplicación

7.11. Escribe la clase `MarcaPagina`, que ayuda a llevar el control de la lectura de un libro. Deberá disponer de métodos para incrementar la página leída, para obtener información de la última página que se ha leído y para comenzar desde el principio una nueva lectura del mismo libro.

7.12. Implementa una clase que permita resolver ecuaciones de segundo grado. Los coeficientes pueden indicarse en el constructor y modificarse *a posteriori*. Es fundamental que la clase disponga de un método que devuelva las distintas soluciones y de un método que nos informe si el discriminante es positivo.

7.13. En el momento de decorar una casa, una habitación o cualquier objeto, se plantea el problema de elegir la paleta de colores que vamos a utilizar en nuestra decoración. Existe una solución, algo atrevida, que consiste en utilizar colores al azar.

Diseña la clase `Colores`, que alberga por defecto una serie de colores (mediante una cadena), aunque es posible añadir tantos como necesitemos. La clase tendrá un método que devuelve una tabla con los *n* colores que necesitemos elegidos al azar sin repeticiones.

7.16. Escribe la clase `Punto` que representa un punto en el plano (con un componente x y un componente y), con los métodos:

- `Punto(double x, double y)`: construye un objeto con los datos pasados como parámetros.
- `void desplazaX(double dx)`: incrementa el componente x en la cantidad `dx`.
- `void desplazaY(double dy)`: incrementa el componente y en la cantidad `dy`.
- `void desplaza(double dx, double dy)`: desplaza ambos componentes según las cantidades `dx` (en el eje x) y `dy` (en el componente y).
- `double distanciaEuclidea(Punto otro)`: calcula y devuelve la distancia euclídea entre el punto invocante y el punto `otro`.
- `void muestra()`: muestra por consola la información relativa al punto.

7.17. El cifrado César es una forma sencilla de modificar un texto para que no sea entendible a quienes no conocen el código. Este cifrado consiste en modificar cada letra de un texto por otra que se encuentra en el alfabeto *n* posiciones detrás.

Por ejemplo, para un valor de *n* igual a 3, la letra *a* se codifica con la *d*, y la letra *q* se codifica con la *x*. En el caso de que una letra exceda a la *z*, seguiremos de forma circular utilizando la *a*. Solo se cifrarán las letras, mayúsculas o minúsculas.

Realiza una clase que, mediante un método estático, devuelva cifrado el texto que se le pasa con un paso de *n* letras.

7.18. Una cola es otra estructura dinámica como la pila, donde los elementos, en vez de apilarse y desapilarse, se encolan y desencolan. La diferencia con las pilas es que se desencola el primer elemento encolado, ya que así es como funcionan las colas del autobús o del cine. El primero que llega es el primero que sale de la cola (vamos a suponer que nadie se cuela). Por tanto, los elementos se encolan y desencolan en extremos opuestos de la estructura, llamados *primero* (el que está primero y será el próximo en abandonar la cola) y *último* (el que llegó último). Implementa la clase `Cola` donde los elementos `Integer` encolados se guardan en una tabla.

7.19. Implementa la clase `Fila` para números `Integer`, usando directamente una tabla para guardar los elementos apilados.

7.20. Repite la Actividad de aplicación 7.18, usando una `Lista` para guardar los elementos encolados.

7.21. Un conjunto es una estructura dinámica de datos como la lista, con dos diferencias: en primer lugar, en una lista puede haber elementos repetidos, mientras que en un conjunto, no. Además, en una lista el orden de inserción de los elementos puede ser relevante y debemos tenerlo en cuenta, mientras que en un conjunto solo interesa si un elemento pertenece o no al conjunto y no el lugar que ocupa. Se pide implementar la clase Conjunto utilizando una lista para almacenar números de tipo `Integer`. Implementa los siguientes métodos:

- Un constructor sin parámetros.
- `int numeroElementos()`: devuelve el número de elementos del conjunto.
- `boolean insertar(Integer nuevo)`: inserta un nuevo elemento en el conjunto.
- `boolean insertar(Conjunto otroConjunto)`: añade al conjunto los elementos del conjunto `otroConjunto`.
- `boolean eliminarElemento(Integer elemento)`: en caso de pertenecer al conjunto, elimina `elemento`.
- `boolean eliminarConjunto(Conjunto otroConjunto)`: elimina del conjunto invocante los elementos del conjunto que se pasa como parámetro.
- `boolean pertenece(Integer elemento)`: indica si el elemento que se le pasa como parámetro pertenece o no al conjunto.
- `muestra()`: muestra el conjunto por consola.

De forma general, los métodos que devuelven un booleano indican con él si el conjunto se ha modificado.

ACTIVIDADES FINALES 7. CLASES

Actividades de ampliación

7.23. Busca en internet información sobre qué son y para qué se usan las colecciones. ¿Qué opinión te merecen? Razona dónde se podrían utilizar.

7.24. Realiza una investigación sobre los tipos abstractos de datos (TAD) que se usan en lenguajes que no disponen de POO. Enumera las ventajas e inconvenientes de los TAD frente a la POO. Justifica tu respuesta.

7.25. Familiarízate con la documentación de Java de Oracle, donde están disponibles las principales clases de la API, así como sus atributos y métodos.

7.26. Existen lenguajes orientados a objetos que utilizan, al igual que Java, constructores para inicializar los objetos recién creados; pero además, implementan el uso de destructores: métodos que se ejecutan justo antes de que un objeto se elimine de la memoria. Lee sobre los destructores y sus usos.

7.27. La POO se basa en una serie de conceptos que son utilizados por los lenguajes de programación. Algunas de las características más importantes de la POO son la abstracción, el encapsulamiento, el principio de ocultación y la herencia. Realiza una investigación sobre estos conceptos y comparte tus conocimientos con tus compañeros.

7.28. La POO está muy vinculada a otro paradigma de programación denominado *programación orientada a eventos*. Busca en internet en qué consiste y cómo podría usarse para implementar una aplicación con interfaz gráfica de usuario.

conjunto `otroConjunto`.

- `boolean eliminarElemento(Integer elemento)`: en caso de pertenecer al conjunto, elimina `elemento`.
- `boolean eliminarConjunto(Conjunto otroConjunto)`: elimina del conjunto invocante los elementos del conjunto que se pasa como parámetro.
- `boolean pertenece(Integer elemento)`: indica si el elemento que se le pasa como parámetro pertenece o no al conjunto.
- `muestra()`: muestra el conjunto por consola.

De forma general, los métodos que devuelven un booleano indican con él si el conjunto se ha modificado.

7.22. Añade a la clase `Conjunto` los siguientes métodos estáticos:

- `static boolean incluido(Conjunto c1, Conjunto c2)`: que devuelve `true` si `c1` está incluido en `c2`, es decir, si todos los elementos de `c1` están también en `c2`.
- `static Conjunto union(Conjunto c1, Conjunto c2)`: devuelve un nuevo conjunto con todos los elementos que están en `c1`, en `c2` o en ambos (elementos comunes y no comunes).
- `static interseccion(Conjunto c1, Conjunto c2)`: que devuelve un nuevo conjunto con todos los elementos que están en `c1` y en `c2` a la vez (elementos comunes).
- `static diferencia(Conjunto c1, Conjunto c2)`: que devuelve un nuevo conjunto con todos los elementos que están en `c1`, pero no en `c2`.

7.27. La POO se basa en una serie de conceptos que son utilizados por los lenguajes de programación. Algunas de las características más importantes de la POO son la abstracción, el encapsulamiento, el principio de ocultación y la herencia. Realiza una investigación sobre estos conceptos y comparte tus conocimientos con tus compañeros.

7.28. La POO está muy vinculada a otro paradigma de programación denominado *programación orientada a eventos*. Busca en internet en qué consiste y cómo podría usarse para implementar una aplicación con interfaz gráfica de usuario.