Tema 5

Objetos, clases y métodos

Cuando se comparan diferentes lenguajes de programación, como *C, Pascal, Fortran, C++, java*, etc, podemos reconocer unos elementos comunes a todos ellos, como variables, expresiones e instrucciones de asignación, condicionales y de repetición. Evidentemente, la existencia de esos aspectos comunes no implica que no haya diferencias entre unos lenguajes y otros. Por ejemplo, en algunos lenguajes deberemos declarar explícitamente las variables antes de utilizarlas, mientras que en otros no existe esa obligación. Así mismo, el repertorio de tipos de datos y operaciones disponibles o el repertorio de instrucciones de *E/S* también varía notablemente, al menos en apariencia, de unos lenguajes a otros. En cualquier caso, se trata de diferencias menores que no llevan consigo grandes cambios en la manera de abordar la escritura de programas.

Por lo que se refiere a los tipos de datos, en todos los lenguajes encontraremos todos o casi todos de los tipos de datos utilizados hasta ahora: números enteros, números en coma flotante, valores lógicos, caracteres y strings. Sin embargo, esas herramientas son insuficientes para manipular informaciones de otra naturaleza con un esfuerzo razonable, por lo que todos los lenguajes de programación incluyen mecanismos que permiten definir nuevos tipos de datos. A diferencia de lo que vemos al comparar los repertorios de instrucciones de diferentes lenguajes, cuando comparamos los mecanismos disponibles para definir nuevos tipos de datos encontramos diferencias importantes entre lenguajes como *C, Pascal, Fortran*, etc y otros como *Smalltalk, C++* o *java*.

Finalmente, es un hecho bien conocido que las aplicaciones informáticas, lo que llamamos el **software**, se hacen cada vez más sofisticadas y complejas: en la figura 5.1 puede verse el tamaño de sistemas operativos, aparecidos en diferentes fechas [1] o el tamaño de las últimas versiones de Eclipse [2]. También es un hecho bien conocido que las dificultades que plantea el desarrollo de una aplicación crecen extrardinariamente a medida que crece el número de líneas de código necesarias.

Para desarrollar una aplicación que sea de alguna utilidad, aunque sea de un tamaño mucho menor que las referidas, se necesitan mecanismos que nos permitan:

- **dividir** el código de la aplicación en **módulos** que puedan ser desarrollados, analizados

[1] D. A. Wheeler. More than a gigabuck: Estimating gnu/linux's size. Web report, June 2001. URL `http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html`

[2] URL `http://www.eclipse.org/org/press-release/20070627_europarelease.php`

Producto	Líneas de código (en miles)
MS-DOS 1.0 (1981)	4
Microsoft Windows 3.1	(1992) 3000
Microsoft Windows 95	15000
Microsoft Windows 98	18000
Microsoft Windows NT (1992)	4000
Microsoft Windows 2000	35000
Windows XP (2001)	45000
Red Hat Linux 6.2 (2000)	17000
Sun Solaris (1998-2000)	7000-8000
Eclipse (Calisto 2006)	7000
Eclipse (Europa 2007)	17000

Figura 5.1: Ejemplos de aumento del tamaño del software

y probados separadamente unos de otros

- **reutilizar** módulos desarrollados para una aplicación en posteriores desarrollos, sin recurrir al método obvio de *cut & paste* y sin que sea necesario disponer de su código fuente

En la forma de responder a esas necesidades es donde pueden verse las diferencias más importantes entre los lenguajes similares a *C* o *Pascal* de un lado, y los lenguajes de **Programación Orientada a Objetos** (P.O.O.), como *Smalltalk, C++* o *java*, de otro. Aunque no existe una varita mágica que permita resolver de manera totalmente satisfactoria esas dos necesidades, sí puede decirse que la P.O.O. aporta los conceptos y mecanismos que, a día de hoy, han demostrado mayor utilidad para atenuar las dificultades que plantea el desarrollo de programas no triviales.

5.1. Objetos, clases y mensajes

Desde el punto de vista de la P.O.O., la memoria del ordenador es algo más que una gigantesca maraña de bits. Es mas bien una especie de universo virtual poblado por representaciones virtuales de cosas o seres existentes en el mundo real, a las cuales llamamos **objetos**. A medida que progresa la ejecución de un programa se crean y destruyen personas, robots, coches, etc ¡un lenguaje de P.O.O. como *java* nos permite crear cualquier *objeto* que podamos imaginar.!

Cuando en P.O.O. hablamos de objetos, nos referimos siempre a objetos virtuales, que solamente existen en la memoria del ordenador, representados en forma de ceros y unos. No obstante, esos objetos tienen una característica muy importante, que es su capacidad de interaccionar unos con otros, enviándose mensajes de unos a otros. Al igual que una persona se puede dirigir a otra para preguntarle la hora o para darle una orden, por ejemplo, también uno cualquiera de esos objetos puede enviarle un mensaje a otro para obtener cierta información

o para conseguir que realice cierta tarea. De hecho, cada uno de los objetos creados por un programa, al margen de que represente una persona, un animal o una cosa, sirve para hacer ciertas tareas, que realizará cuando reciba el mensaje oportuno. En cierto modo, es como si los objetos creados durante la ejecución de un programa fuesen máquinas o instrumentos, cada uno con su propio repertorio de funciones, y el envío de un mensaje la manera de seleccionar una de ellas. Para hacer esto posible, los lenguajes de P.O.O. incluyen un nuevo tipo de instrucciones, cuyos detalles veremos más adelante.

En el ámbito de la P.O.O., todos los objetos creados durante la ejecución de un programa están agrupados o clasificados por categorías, que reciben el nombre de **clases**. De la misma manera que en el mundo real somos capaces de reconocer que varios vehículos distintos son ejemplares del mismo modelo (por ejemplo, `Seat Toledo 1600`), en P.O.O. cada objeto pertenece a una clase determinada. En la jerga de la P.O.O. se dice que cada objeto es una **instancia** de tal clase. Esta clasificación tiene una gran importancia. Por una parte, y al igual que en el mundo real, todos los objetos creados por un programa, y que pertenezcan a la misma clase, tienen las mismas características o prestaciones. Dicho de otra manera, si vemos esos objetos como máquinas o instrumentos virtuales, todos los objetos pertenecientes a la misma clase tienen las mismas funciones. Por otra parte, cuando se usa un lenguaje de P.O.O., uno no puede usar instrucciones para crear objetos de cierta clase, a menos que esa clase haya sido definida con todo detalle. Para eso es necesario escribir una pieza de código, llamada **clase**, que incluso suele almacenarse separadamente del texto del programa original. Esa pieza de código determina qué funciones tienen los objetos de la clase correspondiente, y también cómo se realiza cada una de ellas. La figura siguiente representa la relación existente entre los objetos creados en un programa y sus clases.

5.1.1. ¿Cómo se define una nueva clase de objetos?

Programa 1 Definición de una clase de objetos en *java*

```
1 package ehu.student;
2
3 /**
4  * Ejemplo de definicion de una clase de objetos.
5  *
6  * Los objetos de esta clase no sirven de mucho!!!
7  */
8 public class Inutil {
9 }
```

En el caso más sencillo, para definir una clase de objetos debemos indicar únicamente cuál es su nombre; por ejemplo, el programa 1 define una clase de objetos llamada `Inutil`. Vistos como máquinas virtuales, los objetos de esa clase se caracterizan por no poseer ninguna función que se pueda seleccionar; de ahí el nombre de la clase. Aunque crear objetos de esa clase no parece que tenga mucho interés, ese programa sirve para indicar que en *java* una pieza de texto con la estructura siguiente:

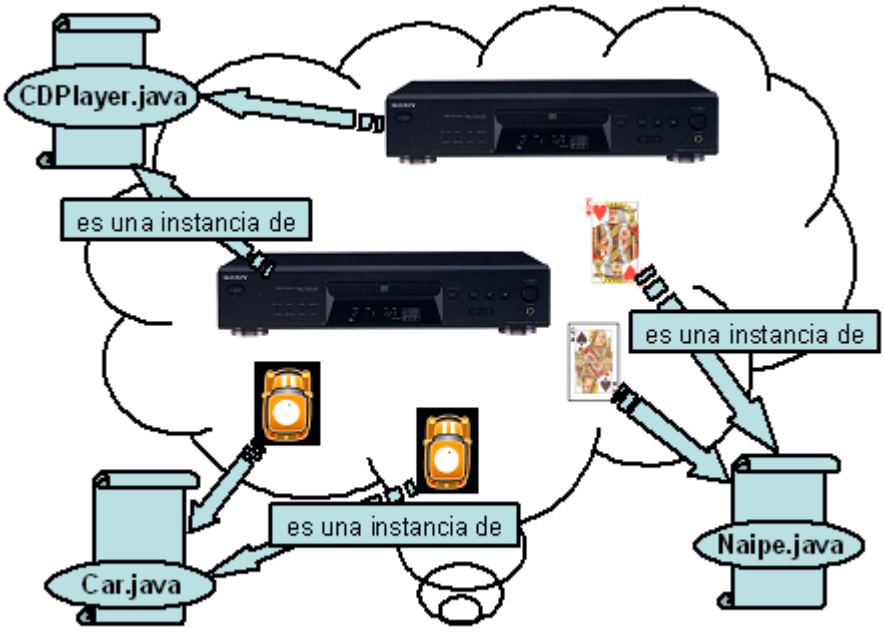


Figura 5.2: Relación entre objetos y clases

```
package Nombre de Paquete;
public class Nombre de clase {
        Declaraciones
}
```

define una clase de objetos. Las partes llamadas `Nombre de Paquete` y `Declaraciones` son partes opcionales que pueden no existir. Más adelante se verá cuál es su estructura y significado, caso de existir. Lo que sí puede decirse ya es que, incluso el típico programa para escribir `Hola mundo!`, como el siguiente: sirve, además, para definir una clase de objetos, llamada en este caso `HolaMundo`.

5.1.2. ¿Cómo se crean objetos?

Una vez definida una clase, podemos crear objetos de esa clase en otros programas. El programa 3 crea varios objetos de la clase `Inutil` que hemos definido antes, aunque su ejecución termina sin producir ningún resultado.

Programa 2 Definición de una clase de objetos en *java*

```
1  package ehu.student;
2
3  /**
4   * Ejemplo de definicion de una clase de objetos.
5   *
6   * Los objetos de esta clase no sirven de mucho!!!
7   */
8  public class HolaMundo {
9
10         public static void main(String[] args) {
11                 System.out.println("Hola, mundo!");
12         }
13
14 }
```

¿Cómo se crean objetos?

La ejecución de:

new `NombreDeClase` **(** `codigo` **)**

crea un *nuevo objeto* de la clase llamada

`NombreDeClase`

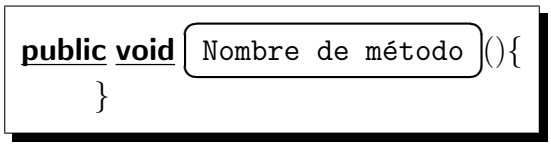
Más adelante se comentarán los detalles relacionados con la estructura del `codigo` entre paréntesis; en este punto, basta con decir que en muchas ocasiones existe. También veremos que para crear objetos de cierta clase no necesitamos tener disponible el código *java* de esa clase, sino que únicamente necesitamos haber instalado en nuestro entorno de desarrollo (por ejemplo, Eclipse) su traducción a *byte-codes*, es decir, el archivo *.class* generado por un compilador de *java*. Por el momento, no obstante, daremos por supuesto que estamos usando Eclipse, y que todas las clases forman parte del mismo proyecto y tienen el mismo `Nombre de Paquete`.

5.1.3. Métodos y mensajes

Si queremos crear objetos que nos sirvan realmente para algo, primero debemos identificar cuáles son sus funciones. Además, esa información debe incluirse en el texto de la clase correspondiente. El programa 4 define una nueva clase de objetos, que pretenden ser representaciones virtuales de reproductores *MP3* con dos funciones: `play` y `stop`. Como indica ese ejemplo, en el texto de una clase, la parte que se llamó antes `Declaraciones`, puede sustituirse por piezas de texto con la estructura siguiente:

Programa 3 Creación de objetos en *java*

```
1  package ehu.student;
2
3  /**
4   * Ejemplo de creacion de objetos.
5   *
6   * Se crean 1000 objetos de clase Inutil
7   */
8  public class CrearObjetosInutil {
9
10         public static void main(String[] args) {
11                 for (int i = 0; i < 1000; i++){
12                         new Inutil();
13                 }
14         }
15 }
```

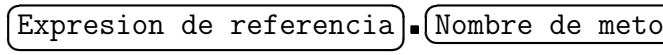


Cada una de esas piezas es una **declaración de método**, debe tener un nombre distinto a las demás, y representa tanto una función que los objetos de esa clase son capaces de realizar como un tipo de mensaje que son capaces de interpretar.

5.1.3.1. Envío de mensajes

Una vez definida la clase `MP3Inutil` podemos crear objetos de esa clase como se muestra en el programa 5. En la figura 5.3 se ha tratado de representar el estado de la memoria justo antes de terminarse la ejecución de ese programa, aunque los objetos `MP3Inutil` creados por el programa no son tan sofisticados como los que se muestran en la figura; al fin y al cabo ¡solo tienen las funciones `play` y `stop`!

Ya que los objetos creados por el programa 5 pretenden ser representaciones virtuales de reproductores MP3, parece razonable que deseemos hacer algo así como apretar alguno de sus botones; por ejemplo, el correspondiente a la función de reproducción, es decir, `play`. Dicho en la jerga de P.O.O., querríamos enviar el mensaje `play` a algunos de esos objetos; al creado en primer lugar, por ejemplo. Para hacer eso posible, todos los lenguajes de P.O.O.incluyen un nuevo tipo de instrucción: las instrucciones de **llamada a método**. Una instrucción de ese tipo sirve para enviar un mensaje a un objeto creado previamente y hace que el destinatario realice una de sus funciones. Su formato es muy parecido en casi todos los lenguajes de programación; en *java* es como sigue:



Por ejemplo:

```
unReproductor.play();
```

Programa 4 Una clase de objetos con dos funciones en *java*

```
1  package ehu.student;
2
3  /**
4   * Ejemplo de definicion de una clase de objetos.
5   *
6   * Los objetos de esta clase poseen varias funciones,
7   * pero no hacen nada
8   */
9  public class MP3Inutil {
10
11         public void play(){
12         }
13
14         public void stop(){
15         }
16 }
```

En los lenguajes de P.O.O., el símbolo • indica el envío de un mensaje a un objeto. La parte llamada `Expresion de referencia` sirve para decidir, en el momento de ejecutarse la instrucción, qué objeto es el destinatario del mensaje. Dicho de otra manera, si se han creado tres objetos `MP3Inutil`, como en la figura 5.3, poniendo en esa parte el código adecuado, podremos decidir a cuál de esos reproductores le damos la orden `play`. Por otra parte, en *java* se exige que el `Nombre de metodo` incluido después del símbolo • sea el nombre de alguna de los métodos incluidos en la clase del destinatario. Es decir, que se puede ejecutar la función `play` de un objeto `MP3Inutil` pero no de un objeto `Inutil`. En *java* el compilador rechazará una instrucción si con ella se pretende enviar a un objeto un mensaje que no se corresponde con ninguna de las funciones que puede realizar.

5.1.3.2. Variables de tipo referencia

Habitualmente lo que hemos llamado arriba `Expresion de referencia` es simplemente un nombre de variable. En *java* pueden declararse, además de las variables de tipo entero o lógico, **variables de tipo referencia**. Dicho informalmente, una instrucción como la siguiente:

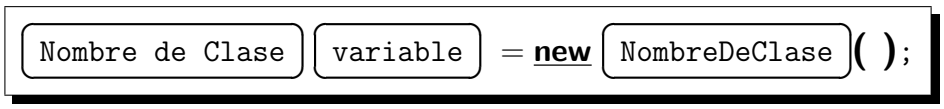
```
MP3Inutil miMp3 = new MP3Inutil( );
```

sirve para dos propósitos. Por una parte, se crea un nuevo objeto de la clase cuyo nombre aparece tras el operador `new`. Por otra parte, sirve para asociar el objeto creado con el nombre `miMp3`; es como si con esa instrucción dijéramos algo así como ¡eh, mira, en lo sucesivo voy a llamar `miMp3` a ese reproductor que acabo de crear!.

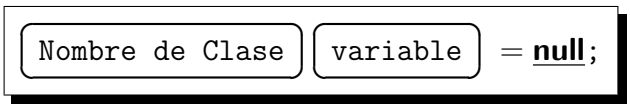
En *java* una instrucción como la anterior es, en definitiva, una declaración de variable y tiene un significado similar a las declaraciones de variables numéricas o lógicas. La diferencia es que una variable de ese tipo no tiene asociado un valor numérico o lógico sino un objeto. Dicho algo groseramene, almacena la **referencia** de un objeto, es decir, la dirección de memoria donde se encuentra ese objeto. Una variable de tipo referencia se declara como sigue:

Programa 5 Creación de objetos en *java*

```
1  package ehu.student;
2
3  /**
4   * Ejemplo de creacion de objetos.
5   *
6   * Se crean objetos de clase MP3Inutil
7   */
8  public class CrearObjetosMP3Inutil {
9
10         public static void main(String[] args) {
11                 new MP3Inutil();
12                 new MP3Inutil();
13                 new MP3Inutil();
14         }
15 }
```



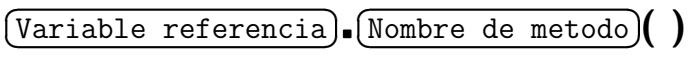
También puede declararse una variable sin crear ningún objeto:



En este caso, antes de que esa variable sea usada para enviar un mensaje, debe habérsele asignado la referencia de un objeto. De lo contrario, la ejecución del programa se abortará produciendo un error llamado `NullPointerException`.

El programa 6 muestra una versión diferente del programa 5. Además de crear varios objetos `MP3Inutil`, ahora se declaran otras tantas variables y a cada una de ellas se le asigna la referencia de uno de los objetos creados. En la figura 5.4 se representa gráficamente la correspondencia existente entre variables y objetos, después de crear esas variables, es decir, cuando se llega a la línea 15.

Las líneas 16 a 21 del programa 6 son instrucciones de envio de mensajes. Esas instrucciones tienen a menudo la estructura siguiente:



y sirven para que el objeto asociado con la variable referida ejecute la función indicada.

Aunque el programa 6 ha creado varios reproductores `MP3Inutil` y a pesar de que se han ejecutado las funciones disponibles en cada uno de ellos, la ejecución de ese programa termina sin que se produzca ningún resultado visible. A continuación veremos cómo se debe definir una clase de manera que los objetos creados en un programa exhiban un comportamiento más interesante.

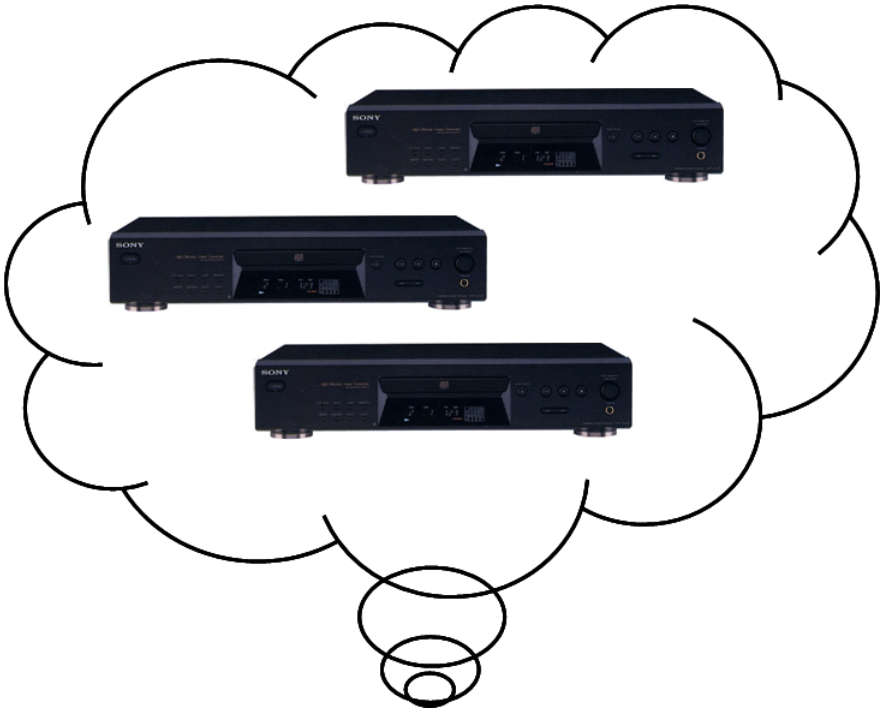


Figura 5.3: Objetos creados por `CrearObjetosMP3Inutil`

Programa 6 Creación de objetos y variables

```
1  package ehu.student;
2
3  /**
4   * Ejemplo de creacion de objetos y envio de mensajes.
5   *
6   * Se crean objetos de clase MP3Inutil
7   * y se ejecutan sus funciones.
8   */
9  public class SendMsgs2MP3Inutil {
10
11         public static void main(String[] args) {
12                 MP3Inutil unReproductor = new MP3Inutil();
13                 MP3Inutil otroReproductor = new MP3Inutil();
14                 MP3Inutil elReproductor = new MP3Inutil();
15
16                 unReproductor.play();
17                 unReproductor.stop();
18                 otroReproductor.play();
19                 otroReproductor.stop();
20                 elReproductor.play();
21                 elReproductor.stop();
22         }
23 }
```

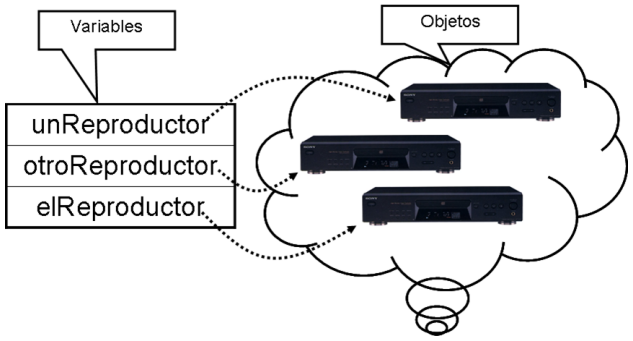


Figura 5.4: Objetos y variables

5.2. Métodos, instrucciones y parámetros

El programa 7 define una clase de *mayordomos* virtuales que son capaces de realizar dos funciones: `salutacion` y `despedida`. La estructura de ese programa es ligeramente distinta de la que tenía el programa 4. La declaración de cada método incluye ahora, además de su nombre, un bloque de instrucciones. Ese bloque de instrucciones es ejecutado cuando se hace una llamada al método correspondiente de un objeto `Mayordomo`.

El programa 8 crea dos *mayordomos* virtuales. Además, en las líneas 15 y 16 se envían los mensajes `salutacion` y `despedida` al primero y al segundo de ellos, respectivamente. La figura 5.5 representa gráficamente el proceso de ejecución del programa 8. Con ella se quiere indicar que, cuando se ejecuta una llamada a un método, se pasa a la instrucción siguiente después de haberse completado el bloque de instrucciones de ese método.

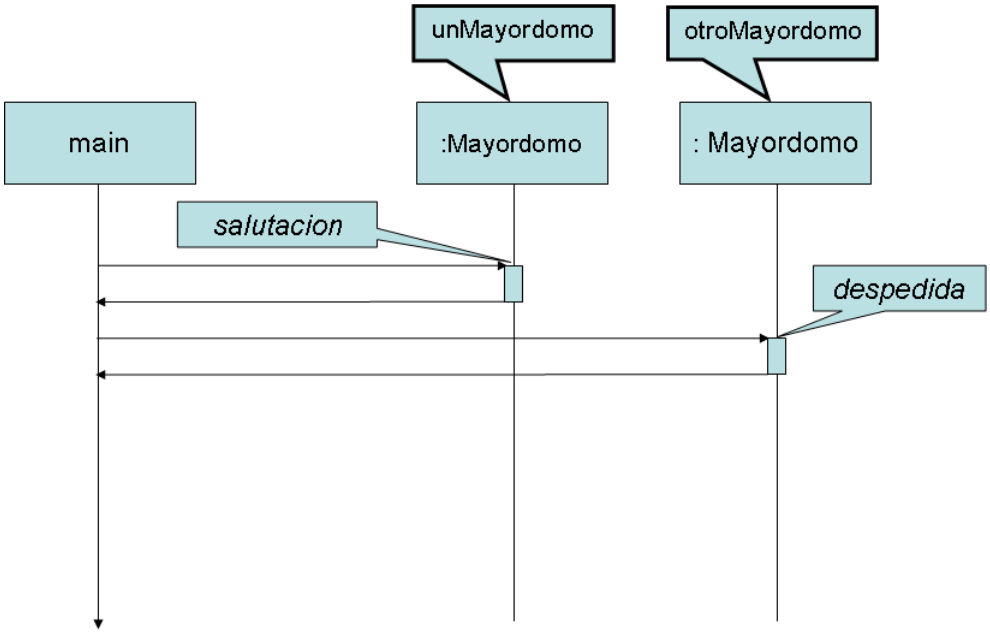


Figura 5.5: Envío de mensajes a mayordomos

Programa 7 Definición de una clase de objetos en *java*

```
1  package ehu.student;
2
3  /**
4   * Ejemplo de definicion de una clase de objetos.
5   *
6   * Los objetos de esta clase poseen varias funciones;
7   * escriben textos en la salida standard
8   */
9  public class Mayordomo {
10
11         public void salutacion(){
12                 System.out.println("Saludos, sir/madam");
13         }
14
15         public void despedida(){
16                 System.out.println("Goodbye!, sir/madam");
17         }
18 }
```

En el bloque de instrucciones de un método pueden incluirse instrucciones de cualquier tipo, conforme a las reglas habituales del lenguaje. Por ejemplo, el programa 9 define una clase de objetos que representan *tutores* virtuales para el aprendizaje de la tabla de multiplicar del cinco. Esos objetos solamente van a tener una función, que servirá para imprimir la tabla del cinco. Por eso la clase `TutorMate` incluye la declaración del método `printTablaDel5`. Como puede verse, el bloque de instrucciones de ese método incluye declaraciones de variables y una instrucción de repetición. Las variables declaradas en el bloque de instrucciones de un método se denominan **variables locales** y únicamente pueden usarse en el bloque de instrucciones en el cual se encuentran declaradas. Además, todas las variables locales de un método se destruyen cuando se completa la ejecución del bloque de instrucciones correspondiente. Así pues, a medida que progresa la ejecución de un programa y los objetos que ha creado van interpretando los mensajes recibidos, se van creando y destruyendo variables.

5.2.1. Métodos y parámetros formales

Los objetos de la clase `TutorMate` (ver programa 9,) no son muy útiles, incluso si solamente deben servir para enseñar tablas de multiplicar. Parece obvio que esos objetos también debieran ser capaces de escribir la tabla de multiplicar del uno, del dos, del tres, etc. También parece obvio que no es una buena solución incluir en la clase `TutorMate` una multitud de métodos, que son prácticamente copia del ya existente:

```
printTablaDel1  printTablaDel2  ...  printTablaDel9  ...
```

De alguna manera, lo que necesitamos es declarar un método de manera que algunos valores numéricos que aparecen incrustados en su bloque de instrucciones puedan ser *tuneados* cada vez que se hace una llamada a ese método. En otras palabras, si imaginamos que los mensajes enviados a los objetos son mensajes de correo electrónico, querríamos tener la posibilidad de enviar a un objeto `TutorMate` mensajes como los siguientes:

Programa 8 Creación de objetos en *java*

```
1  package ehu.student;
2
3  /**
4   * Ejemplo de creacion de objetos y envio de mensajes.
5   *
6   * Se crean objetos de clase Mayordomo
7   * y se ejecutan sus funciones.
8   */
9  public class SendMsgs2Mayordomo {
10
11         public static void main(String[] args) {
12                 Mayordomo unMayordomo = new Mayordomo();
13                 Mayordomo otroMayordomo = new Mayordomo();
14
15                 unMayordomo.salutacion();
16                 otroMayordomo.despedida();
17         }
18 }
```

Asunto	printTablaDel
Anexo 1	1

Asunto	printTablaDel
Anexo 1	5

El programa 10 define una clase similar a `TutorMate`, pero con un método para imprimir tablas de multiplicar más versátil que el método `printTablaDel5` incluido en `TutorMate`. Ahora, después del nombre del método, entre paréntesis, aparece un texto que recuerda una declaración de variable. Una variable como `x` que aparece declarada entre los paréntesis después del nombre de un método es un **parámetro formal**, o simplemente **parámetro**, del método. El parámetro `x` en la declaración del método `printTablaDel` indica que los objetos `TutorMatePlus` necesitan que se les proporcione un valor numérico para poder ejecutar la función referida.

Cuando la declaración de un método incluye también la declaración de un parámetro formal, las instrucciones de llamada correspondientes deben incluir el valor elegido para ese parámetro, como se hace, por ejemplo, en las líneas 14 a 16 del programa 11. Dicho de manera algo burda, una instrucción como:

```
miTutor.printTablaDel(2);
```

se ejecuta como sigue:

1. se examina la variable `miTutor` para determinar quién es el destiatario del mensaje
2. se crea una nueva variable `x` y se le asigna el valor encerrado entre paréntesis
3. se ejecutan las instrucciones del método `printTablaDel`
4. se destruyen las variables locales del método y también `x`

Programa 9 Definición de métodos

```
1  package ehu.student;
2
3  /**
4   * Ejemplo de definicion de una clase de objetos.
5   *
6   * Los objetos de esta clase poseen funciones con parametros.
7   */
8  public class TutorMate {
9
10         public void printTablaDel5(){
11                 int x = 5;
12                 int i = 0;
13                 while (i <= 10){
14                         int tmp = x * i;
15                         System.out.print(x);
16                         System.out.print(" x ");
17                         System.out.print(i);
18                         System.out.print(" = ");
19                         System.out.print(tmp);
20                 }
21         }
22 }
```

Este proceso se representa gráficamente en la figura 5.6.

En el bloque de instrucciones de un método, los parámetros formales se utilizan a todos los efectos como si de variables locales se tratase. En particular, pueden usarse como operandos en una expresión, como se hace en la línea 8 de la clase `TutorMatePlus` (programa 10.)

En *java*, como en cualquier otro lenguaje de programación, pueden declararse métodos con más de un parámetro. Por ejemplo, si queremos incluir en la clase `TutorMate` un método para imprimir las `n` primeras líneas de la tabla de multiplicar de un número `x`, podríamos hacerlo así:

```
1    public void printLineasDeTablaDe(int n, int x)
2    {
3      int i = 0;
4      while (i <= n){
5        int tmp = x * i;
6        System.out.print(x);
7        System.out.print(" x ");
8        System.out.print(i);
9        System.out.print(" = ");
10       System.out.print(tmp);
11     }
12   }
```

Cuando en un método se declaran varios parámetros formales, las instrucciones de llamada al mismo deben incluir el valor correspondiente a cada uno de ellos. Por ejemplo, el programa 11 podría modificarse como sigue para usar también el método `printLineasDeTablaDe` definido arriba (suponemos que ese método está incluido ya en la clase `TutorMate`):

Programa 10 Definición de métodos con parámetros

```
1  package ehu.student;
2
3  public class TutorMatePlus {
4
5          public void printTablaDel(int x){
6                  int i = 0;
7                  while (i <= 10){
8                          int tmp = x * i;
9                          System.out.print(x);
10                         System.out.print(" x ");
11                         System.out.print(i);
12                         System.out.print(" = ");
13                         System.out.print(tmp);
14                 }
15         }
16 }
```

```
1  public class SendMsgs2TutorMatePlus {
2
3    public static void main(String[] args) {
4      TutorMatePlus miTutor = new TutorMatePlus();
5
6      miTutor.printTablaDel(2);
7      miTutor.printLineasDeTablaDe(5, 7);
8    }
9  }
```

En *java* las llamadas a un método tienen la estructura siguiente:

$$\underline{\texttt{Expresión referencia}}.\underline{\texttt{Nombre de método}}(\ \underline{\texttt{exp}_1}, \underline{\texttt{exp}_2}, \underline{\texttt{exp}_3}, \ldots \underline{\texttt{exp}_{n-1}}, \underline{\texttt{exp}_n}\ );$$

y se ejecutan de la manera apuntada arriba; es decir:

1. se determina quién es el destinatario del mensaje, ejecutando el código en

   ```
   Expresion referencia
   ```

   como sabemos, eso se reduce a menudo a examinar el valor de una variable

2. se crea una nueva variable por cada parámetro formal declarado en el método y se le asigna el valor de la expresión correspondiente: el resultado de $\texttt{exp}_1$ se asigna al primer parámetro, el resultado de $\texttt{exp}_2$ al segundo, y así sucesivamente.
   Habitualmente, se dice que esas expresiones, y los resultados obtenidos al ejecutarse la instrucción, son los *argumentos reales* o simplemente **argumentos** de la llamada

3. se ejecutan las instrucciones del método indicado

4. se destruyen las variables locales del método; también las correspondientes a los parámetros formales

Programa 11 Llamadas a métodos con parámetros

```
1  package ehu.student;
2
3  /**
4   * Ejemplo de creacion de objetos y envio de mensajes.
5   *
6   * Se crean objetos de clase TutorMateConParam
7   * y se ejecutan sus funciones.
8   */
9  public class SendMsgs2TutorMatePlus {
10
11         public static void main(String[] args) {
12                 TutorMatePlus miTutor = new TutorMatePlus();
13
14                 miTutor.printTablaDel(2);
15                 miTutor.printTablaDel(5);
16                 miTutor.printTablaDel(7);
17         }
18 }
```

En la figura 5.7 se representa gráficamente este proceso para la instrucción:

```
miTutor.printLineasDeTablaDe(5, 7);
```

del programa anterior. En *java*, el compilador rechazará las llamadas a un método que no incluya una expresión por cada uno de los parámetros formales declarados en ese método. Además, también se exige que cada expresión tenga una estructura acorde con el tipo del parámetro correspondiente. Por ejemplo, no sería aceptable incluir las instrucciones siguientes en el programa anterior:

```
miTutor.printLineasDeTablaDe(5);        /* faltan argumentos */
miTutor.printLineasDeTablaDe(5, 7, 9); /* sobran argumentos */
miTutor.printLineasDeTablaDe(true, 7); /* argumento de tipo inadecuado */
```

5.2.2. Métodos con resultado: instrucciones return

En el mundo real nos dirijimos con frecuencia a otras personas solicitándoles alguna información. También estamos acostumbrados a usar instrumentos tales como relojes, termómetros, etc para saber la hora, la temperatura ambiente, etc. En la jerga de la P.O.O., diríamos que en cada una esas situaciones se ha enviado un mensaje a cierto objeto (una persona, un termómetro, un reloj, etc) y éste nos ha devuelto una respuesta con la información solicitada. Naturalmente, ese comportamiento se refleja también en el código que hay que escribir para definir la clase de objetos correspondiente, y en la manera de usar las funciones de esos objetos.

El programa 12 muestra la definición de una clase de objetos que representan termómetros. Los objetos de esa clase solamente sirven para una cosa, a saber, medir la temperatura ambiente. Así pues, la clase `TermometroCelsius` incluye la declaración del método `medirTemperatura`. Sin entrar en detalles, observaremos que la declaración de ese método

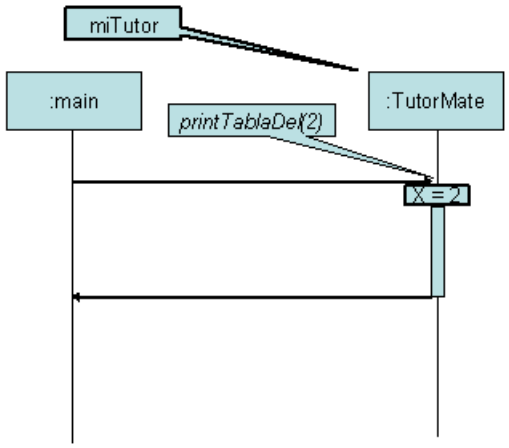
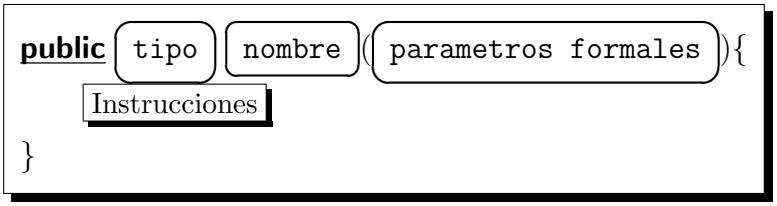


Figura 5.6: Ejecución de `miTutor.printTablaDel(2)`

presenta dos novedades. Por una parte, en la cabecera (ver línea 11) se ha sustituido la palabra **void** por **double**. Por otra parte, en el bloque de instrucciones (ver línea 12) aparece una instrucción nueva. Esas dos novedades indican que cada vez que enviemos a un objeto `TermometroCelsius` el mensaje `medirTemperatura`, ese objeto nos devolverá un número en coma flotante; que ese número sea siempre el valor `-1`, es un detalle sin importancia.

Al diseñar una nueva clase de objetos, *java* nos obliga a decidir cuáles de sus métodos producen una respuesta o resultado y cuáles no (en este caso también podemos decir que se produce una respuesta vacía.) Además, en *java* no se permite que la respuesta de un método sea unas veces un valor entero, otras un valor lógico, y otras que esa respuesta esté vacía. En otras palabras, al declarar un método estamos obligados a decidir qué tipo de respuesta produce, y esa información se incluye en su declaración, que tiene la estructura siguiente:



donde `tipo` debe sustituirse por:

- el nombre de alguno de los tipos de datos de *java*

  `int` `long` `double` ... `boolean`

- el nombre de una clase de objetos

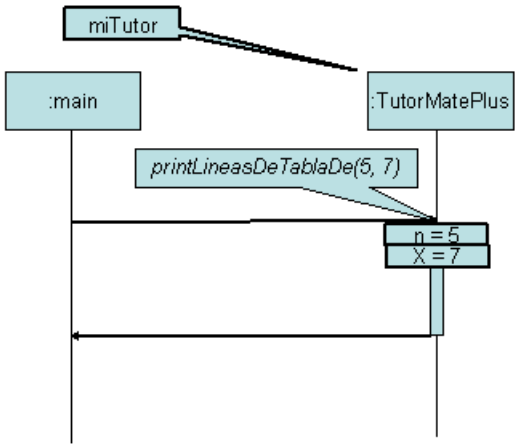


Figura 5.7: Ejecución de `miTutor.printLineasDeTablaDe(5, 7)`

```
MP3Inutil  TutorMate  ...
```

(veremos más adelante que a menudo conviene definir métodos cuyo resultado es la referencia de un objeto)

- la palabra **void** si el método no produce ningún resultado

Por lo general, cuando usamos un método de un objeto y ese método nos devuelve un resultado, es porque estamos interesados, de alguna manera, en ese resultado. El programa 13 muestra varios usos posibles del método `medirTemperatura` de un objeto `TermometroCelsius`. Como puede verse, podemos asignar ese resultado a una variable (ver línea 14), o bien usarlo como parte de una expresión más compleja (ver línea 19), o incluso olvidarnos de ese resultado (ver línea 23). En el segundo caso, el compilador de *java* comprobará que la llamada a un método se usa conforme al tipo de resultado que producirá ese método. Así, en el programa anterior sería ilegal incluir cosas como:

```
int v = termometro.medirTemperatura();        /* resultado no casa con int */
boolean flag = termometro.medirTemperatura();/* resultado no casa con ... */
```

5.2.2.1. La instrucción return

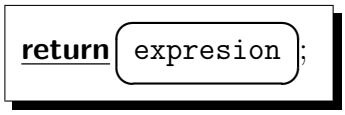
El método `medirTemperatura` de la clase `TermometroCelsius` (ver programa 12) solamente consta de una instrucción. Además, cuando un objeto `TermometroCelsius` ejecuta esa función siempre devuelve el mismo resultado. Por lo general, es necesario definir un bloque de instrucciones más complicado, para calcular el resultado deseado. El programa 14 define una clase de objetos que representan calculadores con dos funciones:

Programa 12 Definición de métodos con respuesta

```
1  package ehu.student;
2
3  /**
4   * Ejemplo de definicion de una clase de objetos.
5   *
6   * Los objetos de esta clase poseen funciones
7   * que devuelven un resultado.
8   */
9  public class TermometroCelsius {
10
11         public double medirTemperatura(){
12                 return -1;
13         }
14 }
```

maximoComunDivisor esPrimo

A diferencia de los métodos cuyo resultado es vacío, (es decir, **void**), un método que declara un tipo de resultado no vacío debe incluir una instrucción **return**. Una instrucción de este tipo tiene la estructura siguiente:



y hace que se termine la ejecución del bloque de instrucciones del método. El resultado devuelto es el valor de la `expresion`, y es ilegal una instrucción que no incluya una expresión del tipo adecuado, conforme al tipo de resultado declarado. Eso quiere decir que si el tipo de resultado de un método es `boolean` será ilegal la instrucción:

```
return 5;
```

Análogamente, si el tipo de resultado de un método es `int` será ilegal la instrucción:

```
return (5 < 3);
```

Como puede verse en el programa 14, un método puede incluir varias instrucciones de este tipo, aunque no es recomendable abusar de esa posiblidad, ya que se hace más difícil entender el comportamiento del programa. En cualquier caso, en *java* es ilegal cualquier método en el que se pueda llegar a la última instrucción sin ejecutar una instrucción de **return**, como podría ocurrir en el método `esPrimo` del programa 14 si se eliminase la línea 31.

5.3. Objetos con estado: atributos

En el mundo real, está claro que cualquier persona podrá, en principio, decirnos cuál es su nombre, edad, etc. Análogamente, en P.O.O. se considera que los objetos creados por un programa también almacenan o memorizan información. Además, todos los objetos de la misma clase almacenan el mismo tipo de información, aunque cada uno de ellos pueda

Programa 13 Uso de métodos con respuesta

```
1  package ehu.student;
2
3  /**
4   * Ejemplo de creacion de objetos y envio de mensajes.
5   *
6   * Se crean objetos de clase TermometroCelsius
7   * y se ejecutan sus funciones.
8   */
9  public class SendMsgs2TermometroCelsius {
10
11         public static void main(String[] args) {
12                 TermometroCelsius termometro = new TermometroCelsius();
13
14                 double t = termometro.medirTemperatura();
15                 System.out.println("Temperatura: " + t);
16
17                 String msg =
18                         "Delta: " +
19                         (t - termometro.medirTemperatura());
20                 System.out.println(msg);
21
22                 /* pasando del resultado! */
23                 termometro.medirTemperatura();
24         }
25 }
```

almacenar unos datos distintos, que pueden cambiar también durante la ejecución del programa. Dicho de otra manera, las instancias de una eventual clase `Persona`, podrían almacenar su nombre, edad y `NIF`, pero dos objetos de esa clase podrían tener distintos nombres, por ejemplo. En P.O.O.todo esto se traduce en que cada **cada instancia de una clase posee sus propias variables**. Los valores asignados en un momento dado a esas variables son los datos almacenados por su propietario en ese momento y definen el estado en que se halla el objeto.

El programa 15 define una nueva clase de objetos que representan *coches* y poseen las funciones:

`encenderLuces` `apagarLuces` `printInfo`

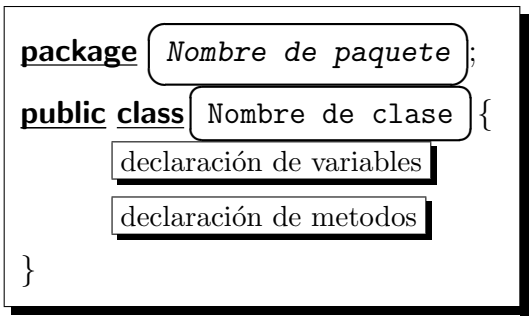
En el programa 16 se crean dos objetos `CocheConLuz`, se les envían mensajes para encender y apagar sus luces, y finalmente se les ordena escribir un texto indicando el estado en que se encuentran las luces de cada uno de ellos.

En la clase `CocheConLuz` (programa 15) podemos ver un elemento que no había aparecido hasta ahora. La línea 11 indica que cada objeto `CocheConLuz` poseerá una variable de nombre `cdgLuces`. La figura 5.8 representa la situación que se producirá después de ejecutarse las líneas 8 y 9 del programa 16.

En *java* la definición de una clase tiene la estructura siguiente:

Programa 14 Definición de métodos con resultado en *java*

```
1  package ehu.student;
2
3  /**
4   * Ejemplo de definicion de una clase de objetos.
5   *
6   * Los objetos de esta clase poseen funciones con parametros.
7   */
8  public class Calculador {
9
10         /**
11          * Devuelve el maximo comun divisor de dos valores dados.
12          */
13         public int maximoComunDivisor(int x, int y){
14                 while (x != y){
15                         int max = Math.max(x, y);
16                         int min = Math.max(x, y);
17                         x = max - min;
18                         y  = min;
19                 }
20                 return x;
21         }
22
23         /**
24          * Devuelve true si un valor dado es primo;
25          * false, en caso contrario.
26          */
27         public boolean esPrimo(int x){
28                 for (int i = 2; i < x; i++){
29                         if (x %i == 0) return false;
30                 }
31                 return true;
32         }
33 }
```



donde la parte declaración de variables, que es opcional, debe estar formada por declaraciones de variables como:

```
private int edad = 0;
private String nombre = "John";
```

Programa 15 La clase CocheConLuz

```
1  package ehu.student;
2
3  /**
4   * Ejemplo de definicion de una clase de objetos.
5   *
6   * Cada objeto de esta clase memoriza
7   * el estado de sus luces: encendidas o apagadas.
8   */
9  public class CocheConLuz {
10
11     private int cdgLuces = 0;
12
13     /**
14      * Las luces del coche quedan encendidas.
15      */
16     public void encenderLuces(){
17         cdgLuces = 1;
18     }
19
20     /**
21      * Las luces del coche quedan apagadas.
22      */
23     public void apagarLuces(){
24         cdgLuces = 0;
25     }
26
27     /**
28      * Se escribe un mensaje informando del estado de las luces.
29      */
30     public void printInfo(){
31         boolean flag;
32         if (cdgLuces == 1) flag = true;
33         else flag = false;
34         System.out.println("Luces encendidas? " + flag);
35     }
36
37 }
```

Las variables declaradas de esa manera se suelen denominar **variables de instancia** o **atributos**.

Las variables de instancia de una clase pueden utilizarse en los métodos de esa clase, como se hace por ejemplo en las líneas 32, 17 y 24 de CocheConLuz (programa 15). De hecho, esas variables se usan conforme a las mismas reglas que rigen el uso de las variables locales declaradas dentro de un método. Sin embargo, estas últimas se crean y destruyen en cada ejecución del método en el que estén declaradas. Por el contrario, cada vez que se crea una instancia de una clase se crean sus variables de instancia particulares, que existirán mientras exista la instancia a la cual pertenecen, y que solamente cambiarán de valor cuando ese objeto ejecute alguna de sus funciones. La figura 5.9 representa gráficamente la ejecución

Programa 16 Creación y uso de objetos `CocheConLuz`

```
1  package ehu.student;
2
3
4  public class CocheTest {
5
6      public static void main(String[] args)
7      {
8          CocheConLuz unCoche = new CocheConLuz();
9          CocheConLuz otroCoche = new CocheConLuz();
10
11         unCoche.encenderLuces();
12         otroCoche.encenderLuces();
13         otroCoche.apagarLuces();
14
15         unCoche.printInfo();
16         otroCoche.printInfo();
17     }
18 }
```

del programa 16 y el cambio de las variables de cada objeto.

Un aspecto crucial a la hora de definir una clase de objetos se refiere a la información que esos objetos necesitan poseer para cumplir su cometido. Es importante observar que, al elegir una familia de variables de instancia particular (en el programa 15, la variable `cdgLuces` de tipo `int`) estamos decidiendo **cómo representar** la información que posee cada objeto. Sin embargo, el aspecto más relevante no es *cómo* se representa esa información sino **qué información** almacena. Por ejemplo, en la clase `CocheConLuz`, también podríamos haber decidido representar el estado de las luces con una variable de tipo `boolean` o incluso de tipo `String`; otra cuestión es que esas elecciones sean más o menos afortunadas.

Por último, observaremos también las variables de instancia de una clase no siempre tendrán valores numéricos o lógicos. De la misma manera que un coche real está formado por multitud de componentes, un objeto puede estar formado, a su vez, por otros objetos. Eso quiere decir que las variables de instancia de una clase serán, a menudo, de tipo referencia. Por ejemplo, cabe imaginar una clase como la siguiente:

```
public class CocheConPiezas {
    private GPS elGPS = new GPS();
    private Motor elMotor = new Motor();

    /*
    * Aqui irian declaraciones de metodos
    */
}
```

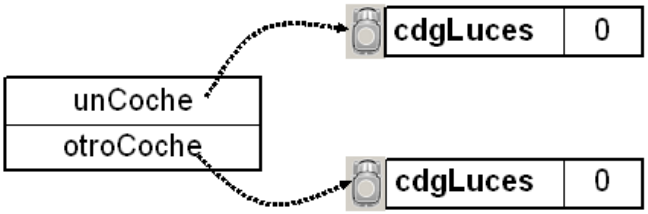


Figura 5.8: Dos objetos con variables

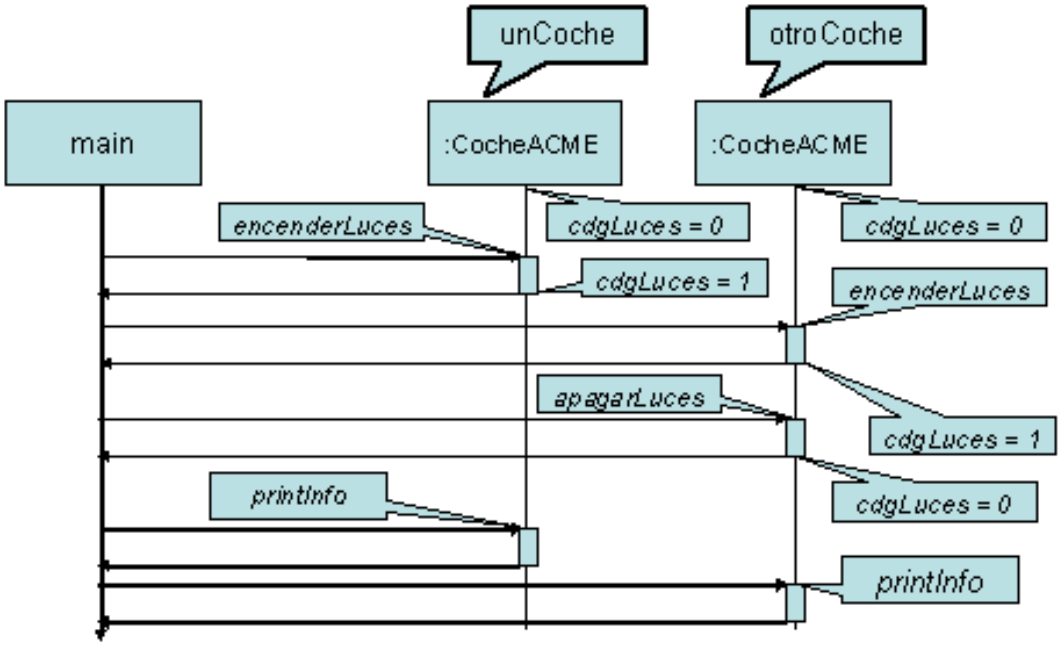


Figura 5.9: Evolución de dos objetos `CocheConLuz`

Programa 17 La clase `CartaBaraja`

```
1  package ehu.student;
2
3  /**
4   * Ejemplo de definicion de una clase de objetos.
5   *
6   * Los objetos de esta clase reprsentan cartas de la baraja.
7   *
8   * Los palos y las figuras de cada palo se suponen numerados consecutivamente:
9   *      palos: 0 (Oros), 1 (Copas), 2 (Espadas), 3 (Bastos)
10  *      figuras: 0 (As), 1 (Dos), ... 8 (Caballo), 9 (Rey)
11  */
12 public class CartaBaraja
13 {
14         private int suit = 0;       /* el codigo del palo */
15         private int figure = 0;     /* el codigo de la figura */
16
17
18         public int getSuit() {
19                 return suit;
20         }
21
22         public int getFigure() {
23                 return figure;
24         }
25
26         public void setSuit(int cdg) {
27                 suit = cdg;
28         }
29
30         public void setFigure(int cdg) {
31                 figure = cdg;
32         }
33
34         public String toString(){
35                 String s =
36                         "[Palo: " + suit + "; Figura: " + figure + "]";
37                 return s;
38         }
39 }
```

5.4. Clases y `constructoras`: objetos inmutables

El programa 27 define una nueva clase de objetos que representan *naipes* de la baraja española. Cada naipe va a almacenar la información correspondiente a su palo y su figura; eso justifica la declaración de variables de instancia en las líneas 14 y 15. Además, cada naipe tiene funciones para poder saber cuál es su palo y su figura:

`getSuit` `getFigure`

Ahora, imaginemos que se crean objetos `CartaBaraja` como sigue:

```
public static void main(String[] args){
  CartaBaraja Ascopas = New CartaBaraja();
  CartaBaraja reyBastos = new CartaBaraja();

  System.out.println(asCopas.toString());
  System.out.println(reyBastos.toString());
}
```

Sin embargo, y a pesar de que los nombres de las variables sugieren otra cosa, los dos objetos creados son iguales, en el sentido de que los valores de sus variables son iguales. Básicamente, ese es el motivo por el cual la clase `CartaBaraja` incluye también los métodos:

`setSuit` `setFigure`

Con esos métodos podemos cambiar los valores asignados por defecto a las variables de instancia de cada `CartaBaraja`, como se muestra en el programa 18.

Programa 18 Manipulación de objetos `CartaBaraja`

```
1  package ehu.student;
2
3  public class CartaBarajaTest {
4
5          public static void main(String[] args) {
6                  CartaBaraja asCopas = new CartaBaraja();
7                  CartaBaraja reyBastos = new CartaBaraja();
8
9                  asCopas.setSuit(1);
10                 asCopas.setFigure(0);
11                 reyBastos.setSuit(3);
12                 reyBastos.setFigure(9);
13
14                 System.out.println(asCopas.toString());
15                 System.out.println(reyBastos.toString());
16         }
17
18 }
```

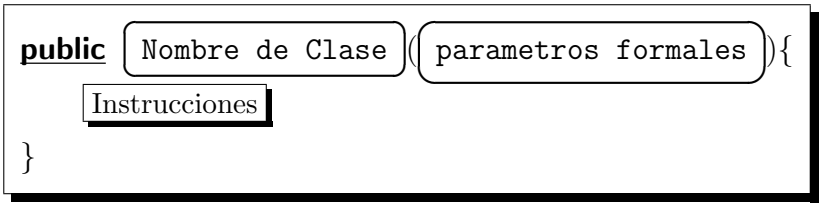
La necesidad de escribir instrucciones como las incluidas entre las líneas 9 a 12 del programa 18, acaba por ser algo molesta. Peor todavía es que esa necesidad es el motivo por el cual se han incluido en la clase `CartaBaraja` los métodos `setSuit` y `setFigure`. Es muy posible que, desde el punto de vista de la estructura de nuestra aplicación, no sea buena idea permitir que puedan cambiarse cosas como el palo o la figura de un naipe, una vez creado. Por todo ello, los lenguajes de P.O.O. permiten incluir en la definición de una clase un tipo particular de métodos, que se suelen llamar **constructores**. El aspecto más relevante de esos métodos es que su bloque de instrucciones se ejecuta automáticamente, cada vez que se crea una instancia de la clase correspondiente. Un *constructor* sirve, por tanto, para establecer el estado inicial adecuado de cada instancia.

El programa 19 muestra otra manera de definir objetos que representen naipes. Por una parte, han desaparecido los métodos `setSuit` y `setFigure`. Por otra parte, entre las líneas 9 a 12 se ha incluido la declaración de uno de esos métodos que llamamos *constructores*.

Programa 19 La clase `CartaBarajaInmutable`

```
1  package ehu.student;
2
3  public class CartaBarajaInmutable
4  {
5          private final int suit;       /* el codigo del palo */
6          private final int figure;     /* el codigo de la figura */
7
8
9          public CartaBarajaInmutable(int cdgSuit, int cdgFigure) {
10                 suit = cdgSuit;
11                 figure = cdgFigure;
12         }
13
14         public int getSuit() {
15                 return suit;
16         }
17
18         public int getFigure() {
19                 return figure;
20         }
21
22         public String toString(){
23                 String s =
24                         "[Palo: " + suit + "; Figura: " + figure + "]";
25                 return s;
26         }
27 }
```

En *java*, la definición de un constructor tiene el formato siguiente:



y conviene tener en cuenta que en el bloque de instrucciones *no puede* incluirse ninguna instrucción `return`.

Cuando en una clase se incluye una constructora con parámetros, el operador `new` *debe* incluir los argumentos correspondientes, conforme a la declaración de parámetros formales de la constructora. Eso quiere decir que podemos crear objetos `CartaBarajaInmutable` como sigue:

```
CartaBarajaInmutable asCopas = new CartaBarajaInmutable(1, 0);
CartaBarajaInmutable reyBastos = new CartaBarajaInmutable(3, 9);
```

Además, también quiere decir que no está permitido usar el operador `new` sin los argumentos necesarios; es decir, que lo siguiente:

```
CartaBarajaInmutable asCopas = new CartaBarajaInmutable();
CartaBarajaInmutable reyBastos = new CartaBarajaInmutable();
```

será ilegal, a menos que en la clase `CartaBarajaInmutable` se incluya también una constructora sin parámetros.

Siempre que sea posible, es recomendable definir y usar clases como `CartaBarajaInmutable` o `String`, cuyas instancias están obligadas a mantener su estado inalterado durante toda la ejecución del programa. En P.O.O. se usa en ocasiones el término **objeto inmutable** para referirse a objetos con esa característica.