# Programación orientada a objetos

## Capítulo 3

## Interacción de objetos

Tutor: Manuel Fernández Barcell
Centro Asociado de Cádiz
http://prof.mfbarcell.es

| Tema 3. Interacción de objetos. Semana 3 | |
|---|---|
| 1- Abstracción<br>2- Modularidad<br>3- Comparación de diagramas de clases con diagramas de objetos<br>4- Tipos primitivos y tipos objeto<br>5- Objetos que crean objetos<br>6- Constructores múltiples<br>7- Llamadas a métodos<br>a. Llamadas a métodos internos<br>b. Llamadas a métodos externos<br>8- Referencia a parámetros del propio objeto. La palabra reservada this<br>9- Depuración de código | 1- Estudiar el capítulo 3 del libro base para la Unidad Didáctica I<br>2- Leer el apéndice B del libro base para la Unidad Didáctica I.<br>3- Realizar los ejercicios en el entorno BlueJ sugeridos en el libro base<br>4- Definir los campos y los métodos necesarios para la resolución de la práctica. |

**Tema 3**. Se amplían los conceptos presentados en los temas anteriores al introducir la interacción entre objetos. Se profundiza en la colaboración entre objetos para conseguir un objetivo común. También se discute como un objeto puede crear otros objetos.

**Tema 3**:

1- Manejar operadores lógicos y matemáticos sobre tipos primitivos.
2- Ser capaz de implementar la creación de un objeto desde otra clase y llamar a sus métodos.
3- Ser capaz de realizar un diagrama de clases sencillo.

# 3.2 Abstracción y modularización

Abstracción

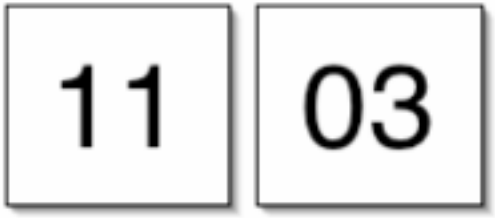
- Habilidad para ignorar/prescindir de los detalles de las partes para centrar la atención en un nivel más alto de un problema

**Concepto**

La **abstracción** es la habilidad de ignorar los detalles de las partes para centrar la atención en un nivel más alto de un problema.


¿Un visor de cuatro dígitos?

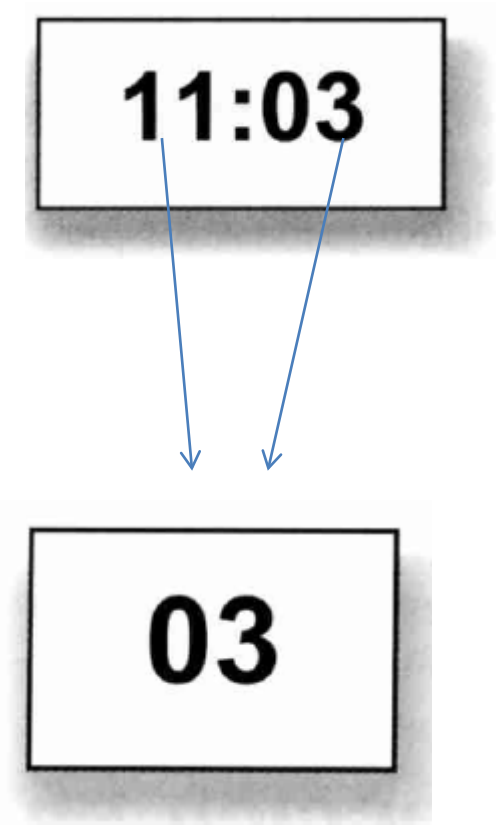
¿O dos visores de dos dígitos?



# 3.2 Abstracción y modularización

La Modularización

- Es el proceso de dividir un todo en partes bien definidas que pueden ser construidas y examinadas separadamente
- Las cuales interactúan de maneras bien definidas
  - Divide et vinces



# 3.5 Implementación de la pantalla del reloj

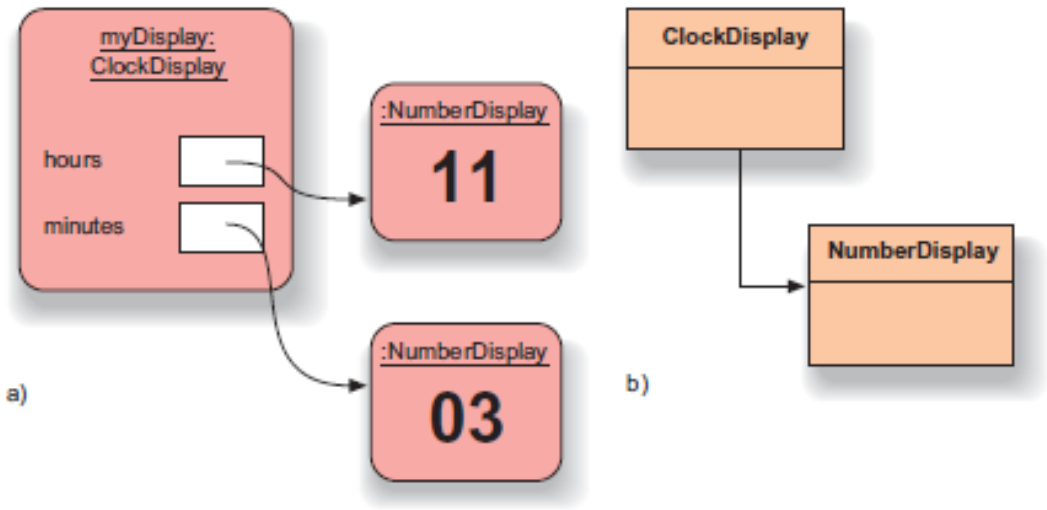
**Concepto**

**Las clases definen tipos**. El nombre de una clase puede ser usado como el tipo de una variable. Las variables cuyo tipo es una clase pueden almacenar objetos de dicha clase.

```
public class VisorDeNumeros
{
private int limite;
private int valor;
  Se omitieron los constructores y los métodos.
}
```

```
public class VisorDeReloj
{
private VisorDeNumeros horas;
private VisorDeNumeros minutos;
  Se omitieron los constructores y los métodos.
}
```

# 3.6 Diagramas de clases y diagramas de objetos




El **diagrama de clases** muestra las clases de una aplicación y las relaciones entre ellas. Da información sobre el código. Representa la vista estática de un programa.

El **diagrama de objetos** muestra los objetos y sus relaciones en un momento dado de la ejecución de una aplicación. Da información sobre los objetos en tiempo de ejecución. Representa la vista dinámica de un programa.

Cuando una variable almacena un objeto, el objeto no se almacena directamente en la variable, lo que **la variable contiene es una referencia al objeto**

**Concept:**

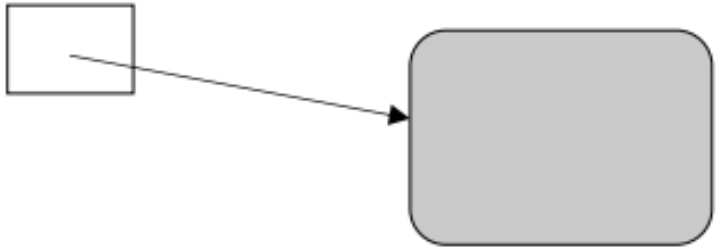
**Object references.** Variables of **object types** store references to objects.

# 3.7 Tipos primitivos y Tipos de objetos

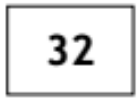
- Java reconoce dos clases de tipos muy diferentes:
  - Los tipos primitivos
  - Los tipos objeto
- Los tipos primitivos
  - Están todos predefinidos en el lenguaje Java
  - Son todos los tipos que **no son objetos**.
  - **No poseen métodos**
- Una diferencia radica en **cómo se almacenan los valores**.
  - **Los valores primitivos se almacenan directamente en una variable.**
  - **Los objetos** no se almacenan directamente en una variable sino que se **almacena una referencia al objeto**.

```
SomeObject obj;
```

Tipo de objeto



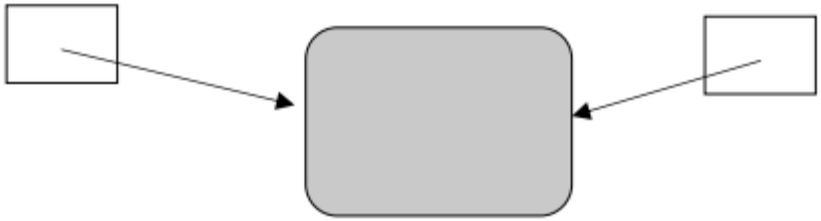
```
int i;
```




Tipo primitivo

```
ObjectType a;
```

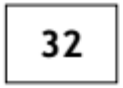
```
ObjectType b;
```



```
b = a;
```

```
int a;
```

```
int b;
```



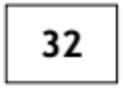



**Tabla 1: Tipos de datos primitivos en Java**

| Tipo primitivo | Descripción | Tamaño | Mínimo | Máximo | Envoltorio |
|---|---|---|---|---|---|
| boolean | Valor binario | - | - | - | Boolean |
| char | Carácter unicode | 16 bits | unicode 0 | unicode $2^{16}$-1 | Character |
| byte | Entero con signo | 8 bits | -128 | +127 | Byte |
| short | Entero con signo | 16 bits | $-2^{15}$ | $+2^{15}$-1 | Short |
| int | Entero con signo | 32 bits | $-2^{31}$ | $+2^{31}$-1 | Integer |
| long | Entero con signo | 64 bits | $-2^{63}$ | $+2^{63}$-1 | Long |
| float | Real simple precisión | 32 bits | $\pm 3.40282347\ e^{+38}$<br>$\pm 1.40239486\ e^{-45}$ | | Float |
| double | Real doble precisión | 64 bits | $\pm 1.79769313486231570\ e^{+308}$<br>$\pm 4.94065645841246544\ e^{-324}$ | | Double |
| void | - vacío - | - | - | - | Void |

# 3.8 ver código de ClokDisplay

- http://docs.oracle.com/javase/7/docs/api/
- http://docs.oracle.com/javase/8/index.html

# Clase “NumberDisplay”

```
public class NumberDisplay
{
   private int limit;
   private int value;

   /**
   * Constructor for objects of class NumberDisplay.
   * Set the limit at which the display rolls over.
   */
   public NumberDisplay(int rollOverLimit)
   {
      limit = rollOverLimit;
      value = 0;
   }
```

```
  /**
     * Return the current value.
     */
    public int getValue()
    {
       return value;
    }
```

```
public String getDisplayValue()
   {
      if(value < 10) {
         return "0" + value;
      }
      else {
         return "" + value;
      }
   }
   /**
    * Set the value of the display to the new specified value. If the new
    * value is less than zero or over the limit, do nothing.
    */
   public void setValue(int replacementValue)
   {
      if((replacementValue >= 0) && (replacementValue < limit)) {
         value = replacementValue;
      }
   }
  /**
   * Increment the display value by one, rolling over to zero if the
   * limit is reached.
   */
   public void increment()
   {
      value = (value + 1) % limit;
   }
}
```

**Operadores Lógicos**

Los operadores lógicos operan con valores booleanos (verdadero o falso) y producen como resultado un nuevo valor booleano. Los tres operadores lógicos más importantes son «y», «o» y «no». En Java se escriben:

**&&** (y)

**||** (o)

**!** (no)

La expresión

**a && b**

es verdadera si tanto **a** como **b** son verdaderas, en todos los otros casos es falsa. La expresión

**a || b**

es verdadera si alguna de las dos es verdadera, puede ser **a** o puede ser **b** o pueden ser las dos; si ambas son falsas el resultado es falso. La expresión

**!a**

es verdadera si **a** es falso, y es falsa si **a** es verdadera.

# Operadores relacionales

| Operador | Utilización | El resultado es true |
|---|---|---|
| > | op1 > op2 | si op1 es mayor que op2 |
| >= | op1 >= op2 | si op1 es mayor o igual que op2 |
| < | op1 < op2 | si op1 es menor que op2 |
| <= | op1 <= op2 | si op1 es menor o igual que op2 |
| == | op1 == op2 | si op1 y op2 son iguales |
| != | op1 != op2 | si op1 y op2 son diferentes |

## Concatenación de cadenas

El operador suma (+) tiene diferentes significados dependiendo del tipo de sus operandos. Si ambos operandos son números, el operador + representa la adición tal como esperamos. Por lo tanto,

```
42 + 12
```

suma esos dos números y su resultado es 54. Sin embargo, si los operandos son cadenas, el significado del signo más es la concatenación de cadenas y el resultado es una única cadena compuesta por los dos operandos. Por ejemplo, el resultado de la expresión

```
"Java" + "con BlueJ"
```

es una sola cadena que es

```
"Javacon BlueJ"
```

# El operador módulo

El último método de la clase `VisorDeNumeros` incrementa el valor del visor en 1 y cuida que el valor vuelva a ser cero cuando alcanza el límite:

```
public void incrementar()
      {
            valor = (valor + 1) % limite;
      }
```

Este método usa el operador *módulo* (%). El operador módulo calcula el resto de una división entera. Por ejemplo, el resultado de la división

```
27/4
```

puede expresarse en números enteros como

```
resultado = 6, resto = 3
```

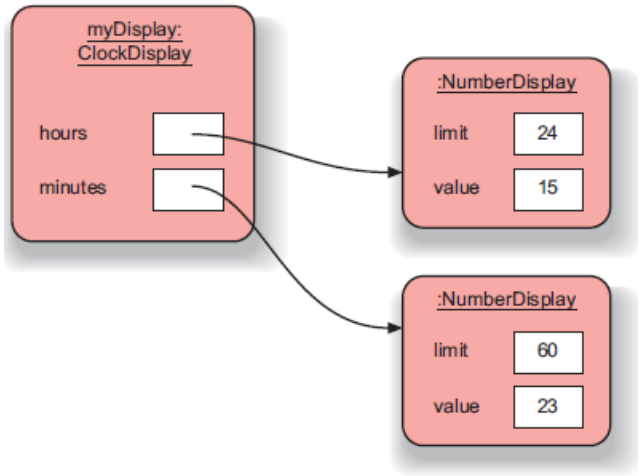
La operación módulo justamente devuelve el resto de la división, por lo que el resultado de la expresión (27%4) será 3.

# 3.9 Clase “ClockDisplay”

```
 */
public class ClockDisplay
{
   private NumberDisplay hours;
   private NumberDisplay minutes;
   private String displayString;    // simulates the
actual display

   /**
    * Constructor for ClockDisplay objects. This
constructor
    * creates a new clock set at 00:00.
    */
   public ClockDisplay()
   {
      hours = new NumberDisplay(24);
      minutes = new NumberDisplay(60);
      updateDisplay();
   }
```

```
   /**
    * Constructor for ClockDisplay objects. This
constructor
    * creates a new clock set at the time specified
by the
    * parameters.
    */
   public ClockDisplay(int hour, int minute)
   {
      hours = new NumberDisplay(24);
      minutes = new NumberDisplay(60);
      setTime(hour, minute);
   }
```

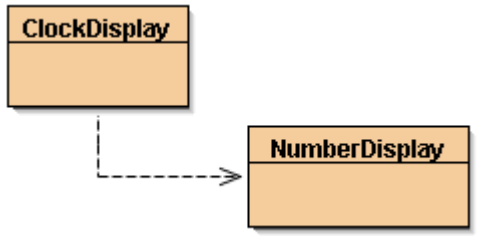


```
    /**
     * This method should get called once every minute
- it makes
     * the clock display go one minute forward.
     */
    public void timeTick()
    {
        minutes.increment();
        if(minutes.getValue() == 0) {  // it just rolled over!
            hours.increment();
        }
        updateDisplay();
    }


     /**
         * Update the internal string that represents the
     display.
         */
        private void updateDisplay()
        {
            displayString = hours.getDisplayValue() + ":" +
                        minutes.getDisplayValue();
        }
     }
```

```
     /**
         * Set the time of the display to the specified
hour and
         * minute.
         */
        public void setTime(int hour, int minute)
        {
            hours.setValue(hour);
            minutes.setValue(minute);
            updateDisplay();
        }

        /**
         * Return the current time of this display in
the format HH:MM.
         */
        public String getTime()
        {
            return displayString;
        }
```



# 3.10 Objetos que crean objetos

**Creación de objetos**. Los objetos pueden crear otros objetos usando el operador **new**.

```
public class VisorDeReloj
{
    private VisorDeNumeros horas;

    private VisorDeNumeros minutos;

    Se omitieron los restantes campos.

    public VisorDeReloj()
    {
        horas = new VisorDeNumeros(24);
        minutos = new VisorDeNumeros(60);
        actualizarVisor();
    }

    Se omitieron los métodos.
}
```

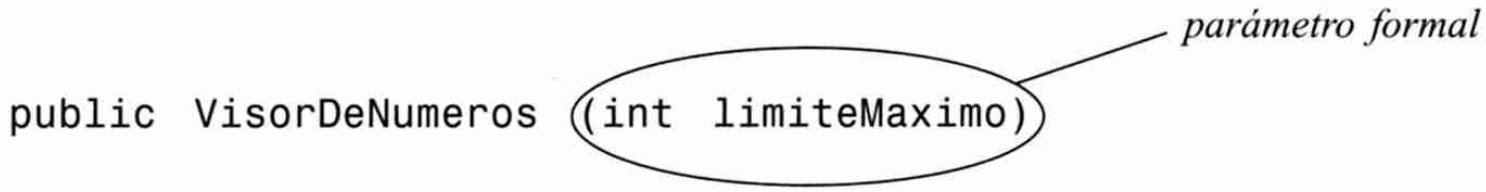
```
new NombreDeClase (lista-de-parámetros)
```

# Parámetro formal parámetro actual

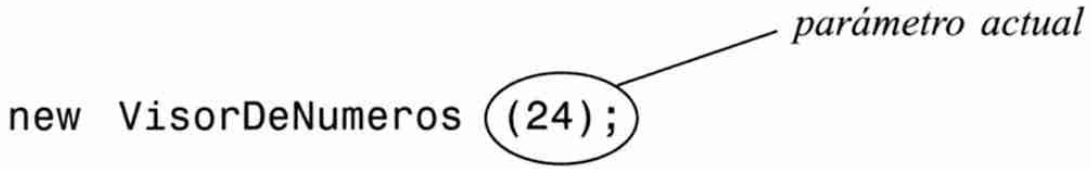
La operación new hace dos cosas:

1. Crea un nuevo objeto de la clase nombrada (en este caso, `VisorDeReloj`).
2. Ejecuta el constructor de dicha clase.

Si el constructor de la clase tiene parámetros, los parámetros actuales deben ser proporcionados en la sentencia new. Por ejemplo, el constructor de la clase `VisorDeNumeros` fue definido para esperar un parámetro de tipo entero:




Por lo tanto, la operación new sobre la clase `VisorDeNumeros` que invoca a este constructor, debe proveer un parámetro actual de tipo entero para que coincida con el encabezado que define el constructor:



```java
public class Cliente{

    private String nif;
    private String nombre;
    private String apellidos;

    public Cliente(String nif, String nombre, String apellidos){
        this.nif = nif;
        this.nombre = nombre;
        this.apellidos = apellidos;
    }

    public String getNIF() {
        return nif;
    }

    public void setNIF(String nif) {
        this.nif = nif;
    }
```

# 3.11 Constructores múltiples

**Sobrecarga**. Una clase puede contener más de un constructor o más de un método con el mismo nombre, siempre y cuando tengan distintos conjuntos de parámetros que se diferencien por sus tipos.

```
public class Arboles  {

    public Arboles() {
      System.out.println("Un árbol genérico");
    }

    public Arboles(String tipo) {
     System.out.println("Un árbol tipo " + tipo);
    }

    public Arboles(int altura) {
     System.out.println("Un árbol de " + altura + " metros");
   }

    public Arboles(int altura,String tipo) {
     System.out.println("Un " + tipo + " de " + altura + " metros");
   }

    public Arboles(int altura,String tipo,int lado) {
     System.out.println("Un " + tipo + " de " + altura + " metros "+ lado);
   }

    public static void main(String args[]) {
        Arboles arbol1 = new Arboles(4);
        Arboles arbol2 = new Arboles("Roble");
        Arboles arbol3 = new Arboles();
        Arboles arbol4 = new Arboles(5,"Pino");
        Arboles arbol5 = new Arboles(5,"Pino", 300);
     }
}
```

```
public class Coche {
  String matricula;
  String fabricante;
  String modelo;
  String color;

  public Coche (String matricula) {
    this.matricula = matricula;
  }

  public Coche (String matricula, String fabricante,
                String modelo,String Color) {
    this.matricula = matricula;
    this.fabricante = fabricante;
    this.modelo = modelo;
    this.color = color;
  }
}

public class PruebaCoche {
  public static void main (String[] args) {
    Coche c1 = new Coche ("3468 CXV");
    Coche c2 = new Coche ("4462 BIG", "BMW", "525i", "Negro");
    Coche c3 = new Coche ();   // ERROR!!
}
```

# 3.12 Llamadas a métodos

## 3.11.1 Llamadas a métodos Internos

Los métodos pueden llamar a otros métodos de la misma clase como parte de su implementación. Esto se denomina ***llamada a método interno***.

```
actualizarVisor();
```

Esta sentencia es una *llamada a un método*. Como hemos visto anteriormente, la clase `VisorDeReloj` tiene un método con la siguiente signatura:

```
private void actualizarVisor()
```

La llamada a método que mostramos en la línea anterior, justamente invoca a este método. Dado que este método está ubicado en la misma clase en que se produce su llamada, decimos que es una *llamada a un método interno*. Las llamadas a métodos internos tienen la siguiente sintaxis:

```
nombreDelMétodo (lista-de-parámetros)
```

# 3.12.2 Llamada a métodos externos

Los métodos pueden llamar a métodos de otros objetos usando la notación de punto: se denomina ***llamada a método externo***.

```
public void ticTac()
      {
            minutos.incrementar();
            if(minutos.getValor() == 0) {   // ¡alcanzó el
límite!
                  horas.incrementar();
            }
            actualizarVisor();
      }
```

Si este visor se conectara a un reloj real, este método sería invocado una vez cada 60 segundos por el temporizador electrónico del reloj. Por ahora, lo llamamos nosotros para probar el visor.

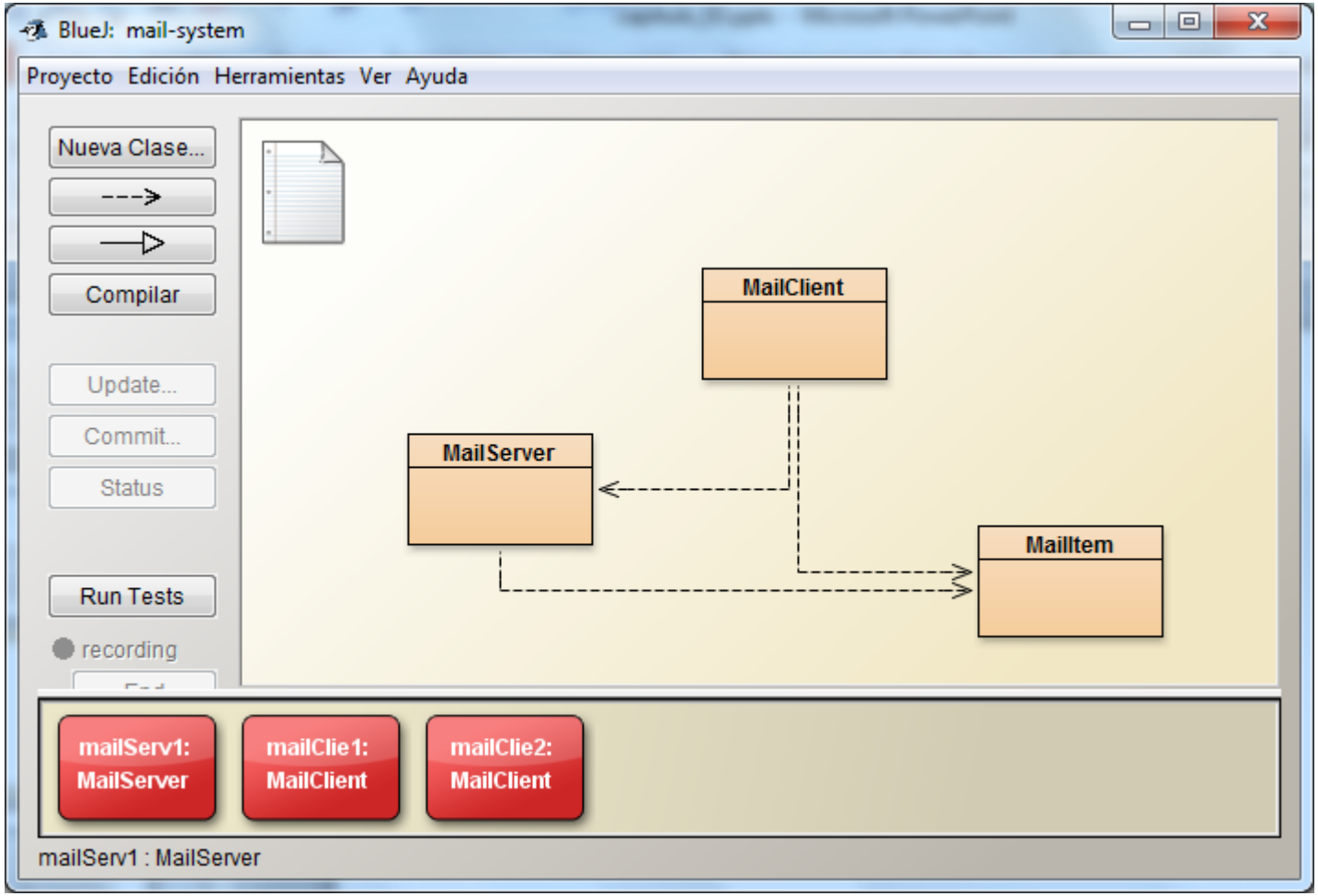
Cuando es llamado, el método `ticTac` ejecuta primero la sentencia

```
minutos.incrementar();
```

Esta sentencia llama al método `incrementar` del objeto `minutos`. Cuando se llama a uno de los métodos del objeto `VisorDeReloj`, este método a su vez llama a un método de otro objeto para colaborar en la tarea. Una llamada a método desde un método de otro objeto se conoce como *llamada a un método externo*. La sintaxis de una llamada a un método externo es

```
objeto.nombreDelMétodo (lista-de-parámetros)
```

# Proyecto mail-system



# Palabra clave “this”

- El motivo por el que se usa esta construcción radica en que tenemos una situación que se conoce como sobrecarga de nombres.
- Significa que el mismo nombre es usado por entidades diferentes.

```
public class Correo{

    // El remitente del mensaje.
    private String para;
    // El destinatario del mensaje.
    private String de;
    // El texto del mensaje.
    private String mensaje;


    public Correo(String para,String de,String mensaje){
        this.para = para;//this.para es el campo y para el parámetro
        this.de = de;//this.de es el campo y de el parámetro
        this.mensaje = mensaje;//this.mensaje es el campo y mensaje el parámetro
    }
```

# this

- Lo que necesitamos es un mecanismo para acceder a un campo cuando existe una variable con el mismo nombre declarada más cerca de la sentencia que la usa.
- Este mecanismo es justamente lo que significa la palabra clave this.
  - La expresión this hace referencia al objeto actual.
- Ahora podemos leer la sentencia de asignación nuevamente:

  **this.para = para;**

  Como podemos ver, esta sentencia tiene el mismo efecto que la siguiente:

*campo de nombre*.`para` = *parámetro de nombre* `para`;

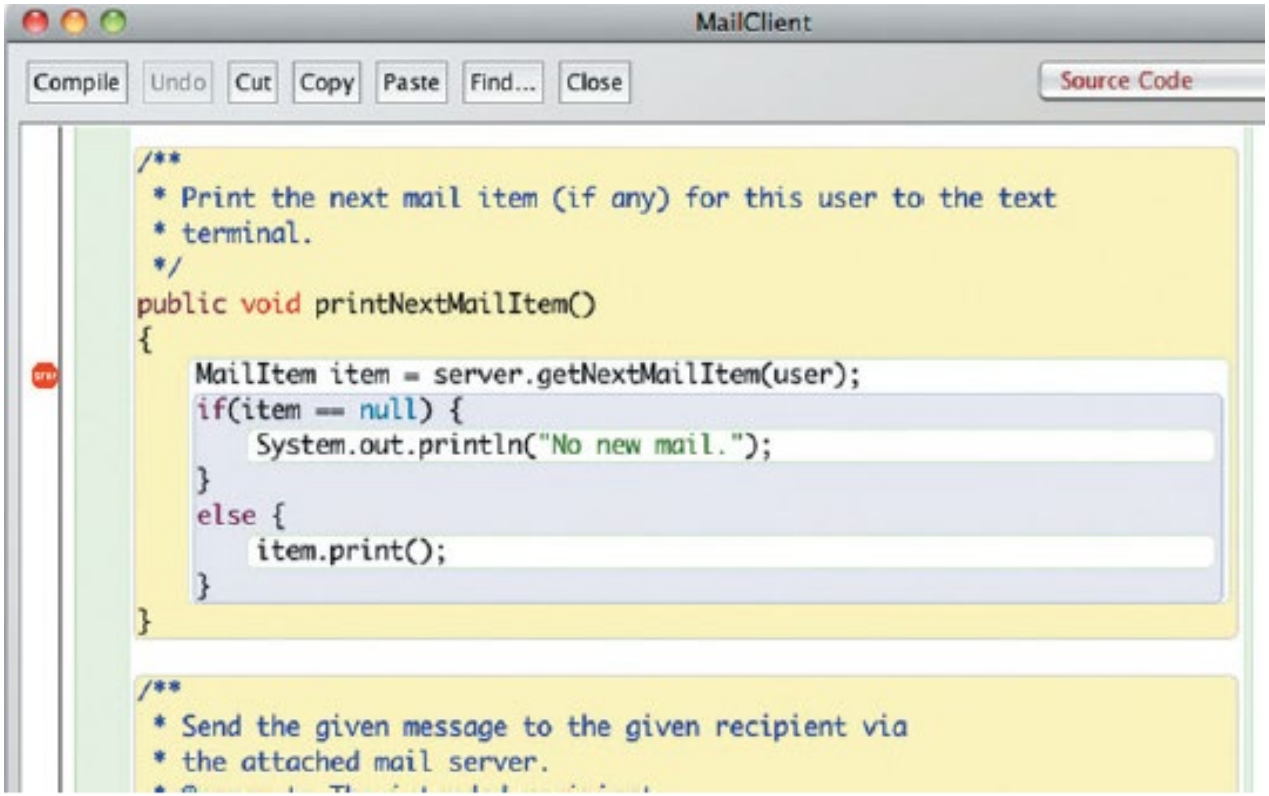
# Palabra clave “this”

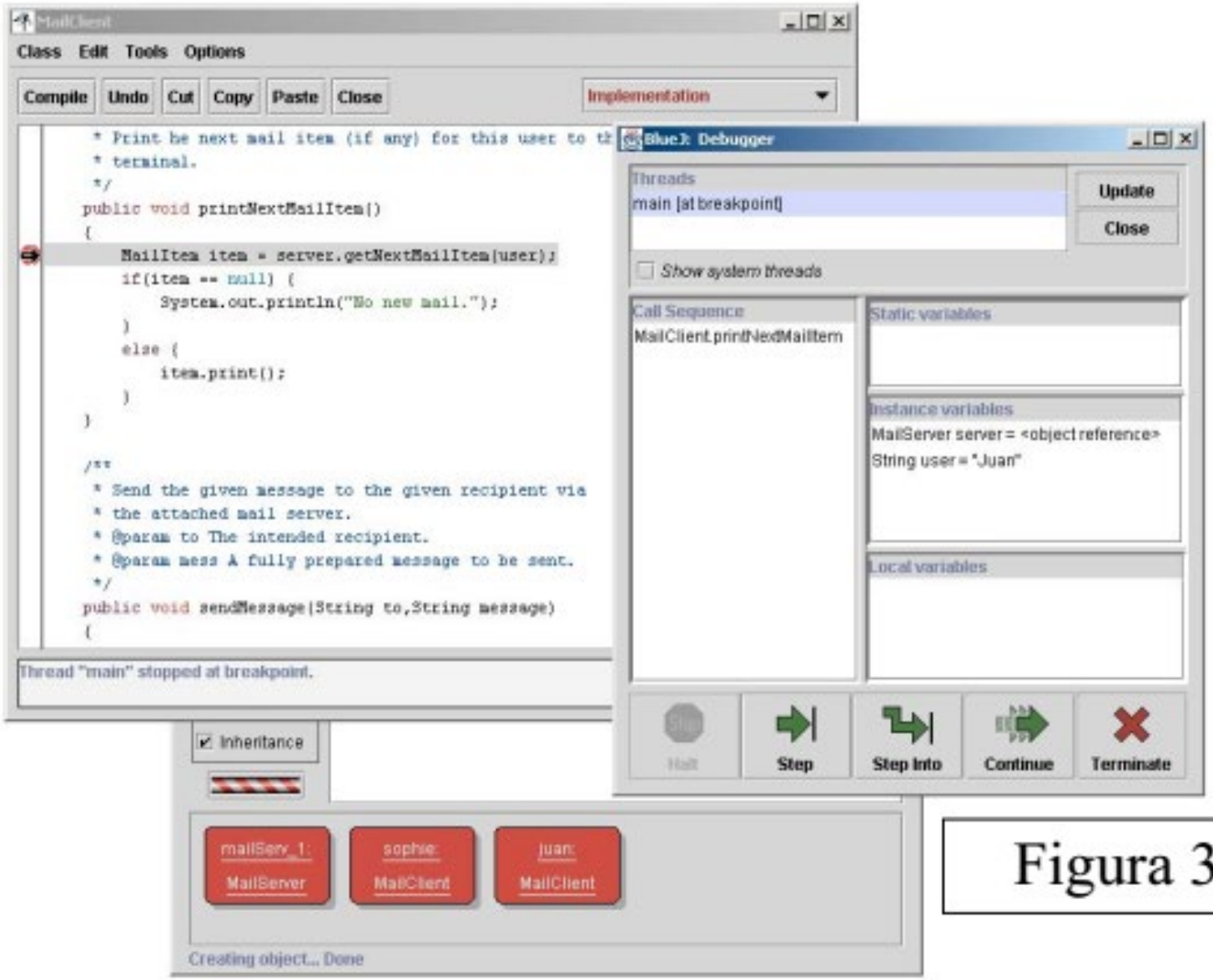
```
public class Mensaje
{
    // El remitente del mensaje.
    private String de;
    // El destinatario del mensaje.
    private String para;
    // El texto del mensaje.
    private String texto;
    /**
     * Crea un mensaje de correo del remitente para un
destinatario
     * dado, que contiene el texto especificado.
     * @param de      El remitente de este mensaje.
     * @param para    El destinatario de este mensaje.
     * @param texto El texto del mensaje que será enviado.
     */
    public Mensaje(String de, String para, String texto)
    {
        this.de = de;
        this.para = para;
        this.texto = texto;
    }
    Se omitieron los métodos.
}
```

`this.de = de;` *campo de nombre* “de” = *parámetro de nombre* “de”;

# 3.13 Uso del depurador

- Establecemos el punto de interrupción, pulsando sobre el número de la línea






Un **depurador** es una herramienta de software que ayuda a examinar cómo se ejecuta una aplicación. Puede usarse para encontrar problemas.

Figura 3.6

- Cinco zonas de visualización:
  - Variables:
    - Estáticas,
    - De variable (instancia),
    - Locales al método actual)
  - Pila de llamadas a métodos (Call Sequence)
  - Threads (para programas concurrentes).
- Botones para controlar la ejecución:
  - Halt:
    - Activado cuando se está ejecutando el programa. Permite parar la ejecución.
  - Step:
    - Ejecución de la sentencia actual. Se para justo al completarla.
  - Step Into:
    - Si la sentencia es de llamada a un método, se introduce en éste y se para en la primera sentencia del mismo.
  - Continue:
    - Para seguir la ejecución normal del programa hasta llegar a otro breakpoint o que se pulse Halt.
  - Terminate:
    - Acaba la ejecución del programa

# Ejecución paso a paso

- Cuando la ejecución se detiene en un punto de interrupción, al hacer clic sobre el botón **Step** se ejecuta una sola línea de código y luego se detiene nuevamente

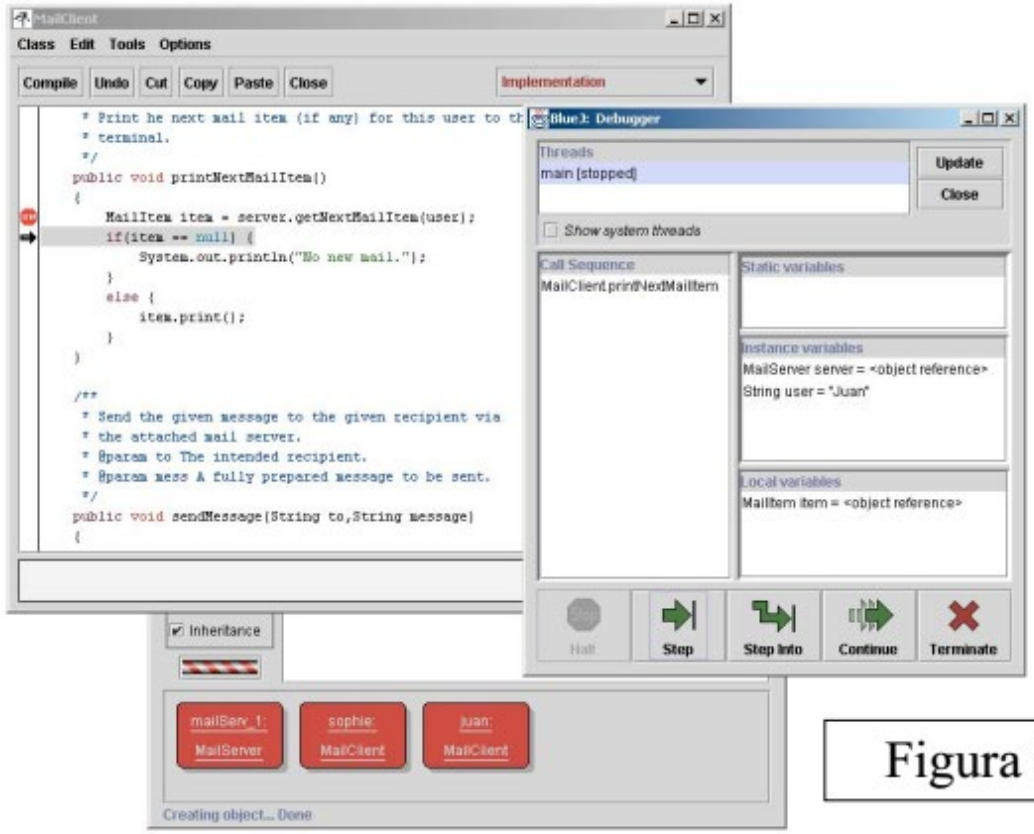



Figura 3.7

# Glosario

- **abstracción** La abstracción es la habilidad de ignorar los detalles de las partes para enfocar la atención en un nivel más alto de un problema.
- **modularización** La modularización es el proceso de dividir una totalidad en partes bien definidas que podemos construir y examinar separadamente y que interactúan de maneras bien definidas.
- **las clases definen tipos** Puede usarse un nombre de clase para el tipo de una variable. Las variables que tienen una clase como su tipo pueden almacenar objetos de dicha clase.
- **diagrama de clases** Los diagramas de clases muestran las clases de una aplicación y las relaciones entre ellas. Dan información sobre el código. Representan la vista estática de un programa.
- **diagrama de objetos** Los diagramas de objetos muestran los objetos y sus relaciones en un momento dado, durante el tiempo de ejecución de una aplicación. Dan información sobre los objetos en tiempo de ejecución. Representan la vista dinámica de un programa.
- **referencias a objetos** Las variables de tipo objeto almacenan referencias a los objetos.

- **tipo primitivo** Los tipos primitivos en Java no son objetos. Los tipos `int`, `boolean`, `char`, `double` y `long` son los tipos primitivos más comunes. Los tipos primitivos no tienen métodos.
- **creación de objetos** Los objetos pueden crear otros objetos usando el operador `new`.
- **sobrecarga** Una clase puede contener más de un constructor o más de un método con el mismo nombre, siempre y cuando tengan un conjunto de tipos de parámetros que los distinga.
- **llamada a método interno** Los métodos pueden llamar a otros métodos de la misma clase como parte de su implementación. Esto se denomina llamada a método interno.
- **llamada a método externo** Los métodos pueden llamar a métodos de otros objetos usando la notación de punto. Esto se denomina llamada a método externo.
- **depurador** Un depurador es una herramienta de software que ayuda a examinar cómo se ejecuta una aplicación. Puede usarse para encontrar errores.