

El lenguaje Java

Java es un lenguaje en el que se trabaja, preferentemente, sólo con clases y objetos. Esto implica que todas las variables representan instancias de objetos de alguna clase. De todas formas, los tipos primitivos como `int`, `float` y `double` se mantienen como opciones por eficiencia, aunque tienen su equivalente en forma de clase. En Java, incluso el programa en sí es un objeto. Enseguida presentamos el ejemplo de un programa en Java que escribe en la pantalla el mensaje "Hola Mundo!".

```
// Ejemplo simple
public class Ejemplo {

    // main es el metodo principal
    public static void main( String[] args ) {
        // imprime algo en la salida estandar
        System.out.println( "Hola mundo!" );
    }
}
```

Java es un lenguaje fuertemente tipado. Esto significa que para todo recurso que vayamos a utilizar, previamente, debemos definirle su tipo de datos.

Definición 1: llamamos “objeto” a toda variable cuyo tipo de datos es una “clase”.

Definición 2: llamamos “clase” a una estructura que agrupa datos más la funcionalidad necesaria para manipular dichos datos.

Las cadenas de caracteres, por ejemplo, son objetos de la clase `String`; por lo tanto, almacenan información (la cadena en sí misma) y la funcionalidad necesaria para manipularla. Veamos las siguientes líneas de código:

```
String s = "Hola Mundo";
int i = s.indexOf("M");
```

El objeto `s` almacena la cadena `"Hola Mundo"` y tiene la capacidad de informar la posición de la primera ocurrencia de un determinado carácter dentro de la cadena.

La figura II-1 ilustra la diferencia entre la arquitectura de ejecución de un programa escrito en C/C++ y la de un programa en Java:

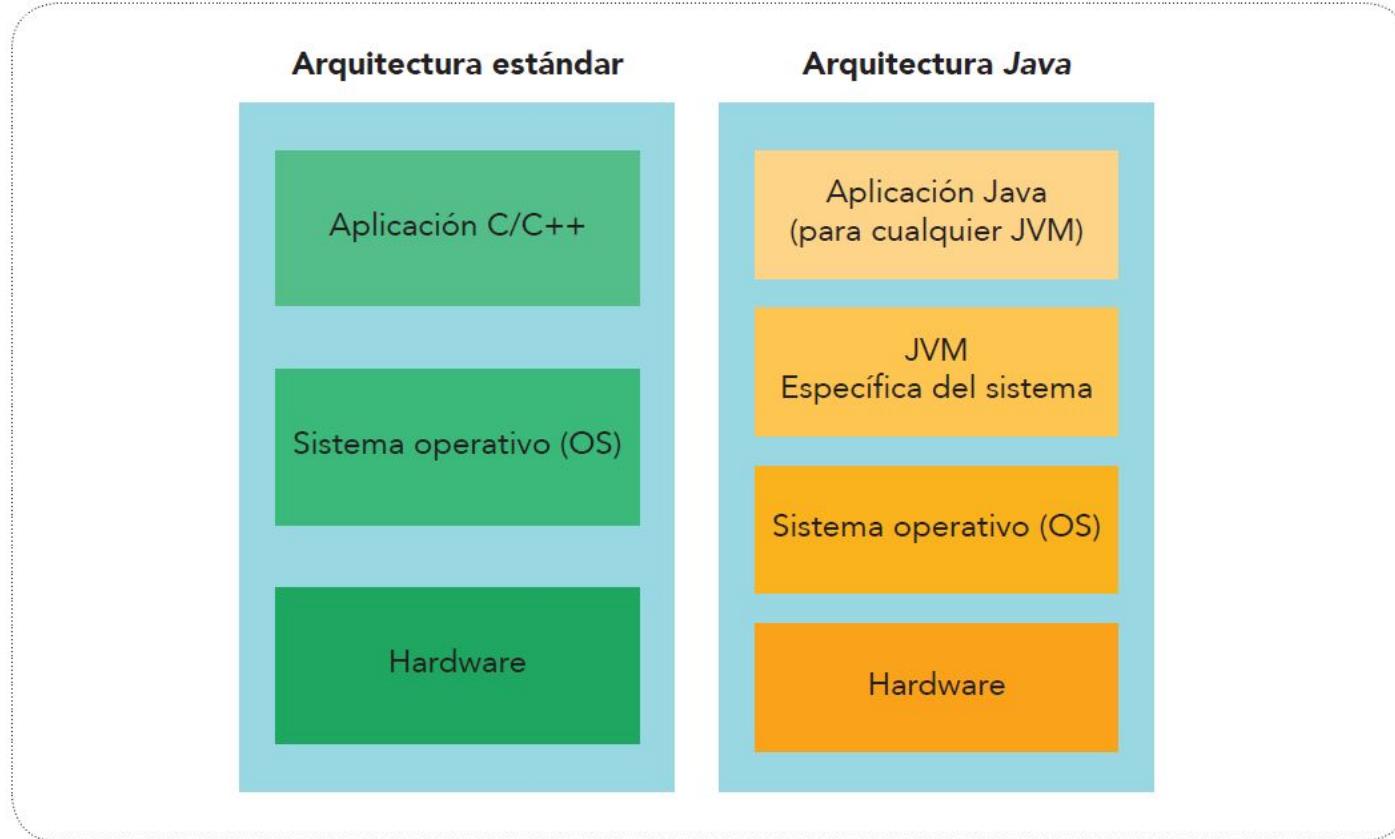


Figura II-1. Diferentes arquitecturas de ejecución

La figura II-2 muestra los productos parciales obtenidos después de cada etapa en la creación de software: tanto en una compilación estándar, como en una compilación Java. El código en Java compilado tiene la extensión `.class` y el archivo está escrito en *bytecode*. La máquina virtual de cada sistema operativo interpreta este bytecode ejecutable:

- Los programas fuente en Java tienen extensión `.java`
- Los archivos comprimidos tienen la extensión `.jar`

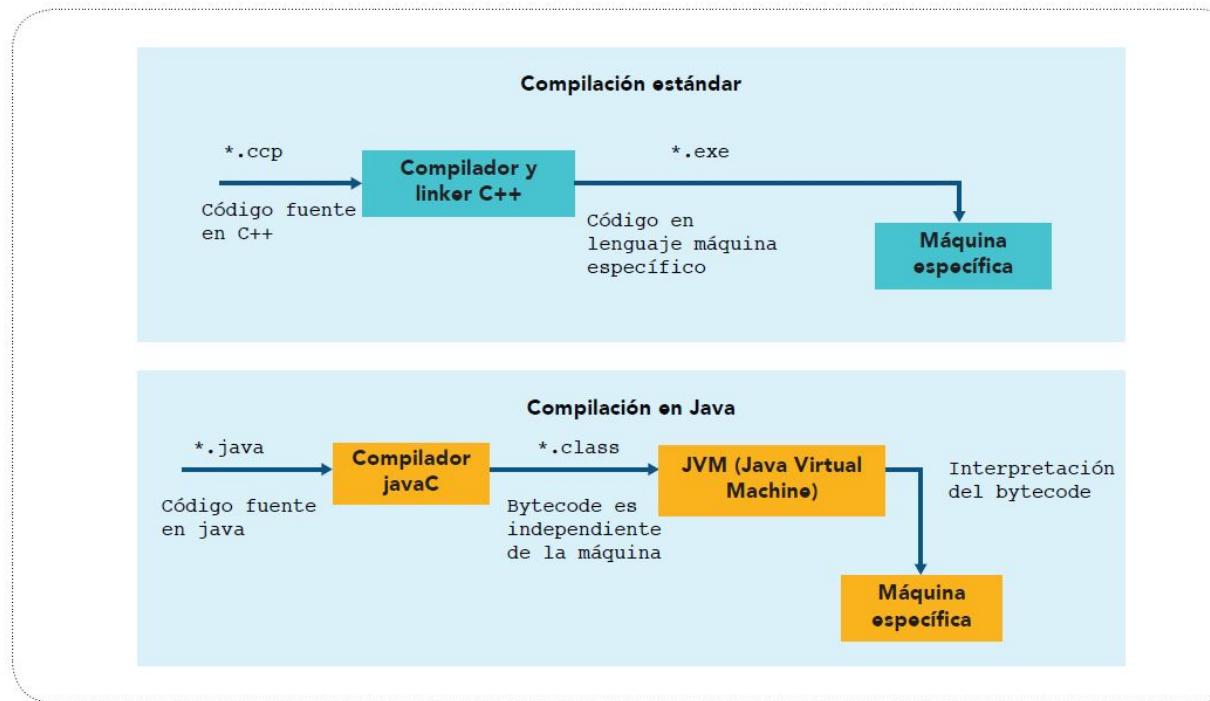


Figura II-2. Compilación estándar vs. compilación en Java

Clases en Java

La sintaxis para definir una clase en Java es la siguiente:

```
public class <NombreClase> {  
    <definicion de los atributos>  
    <definicion de los constructores>  
    <definicion de los metodos>  
}
```

Por convención, el nombre de una clase debe tener inicial mayúscula y el resto en minúsculas. En el caso de nombres compuestos, se utilizan las iniciales de cada palabra en mayúscula.

Los atributos

Los atributos definen la estructura de datos de la clase, los cuales, por omisión, son *públicos*, es decir, accesibles desde otras clases, lo que significa que se modifican desde afuera del objeto. Es altamente recomendable declarar todos los atributos con el modificador `private` y solamente cambiarlo a `public` o `protected` cuando sea absolutamente necesario. En los ejemplos de este libro se verá más claramente el uso de estos modificadores.

Por convención, los nombres de los atributos deben escribirse en minúsculas. En el caso de nombres compuestos, cada palabra intermedia debe iniciar con mayúscula. Por ejemplo:

```
private int promedio;  
public float saldoCuentaCredito;
```

La figura II-3 muestra la clase TarjetaCredito, que contiene varios ejemplos de atributos.

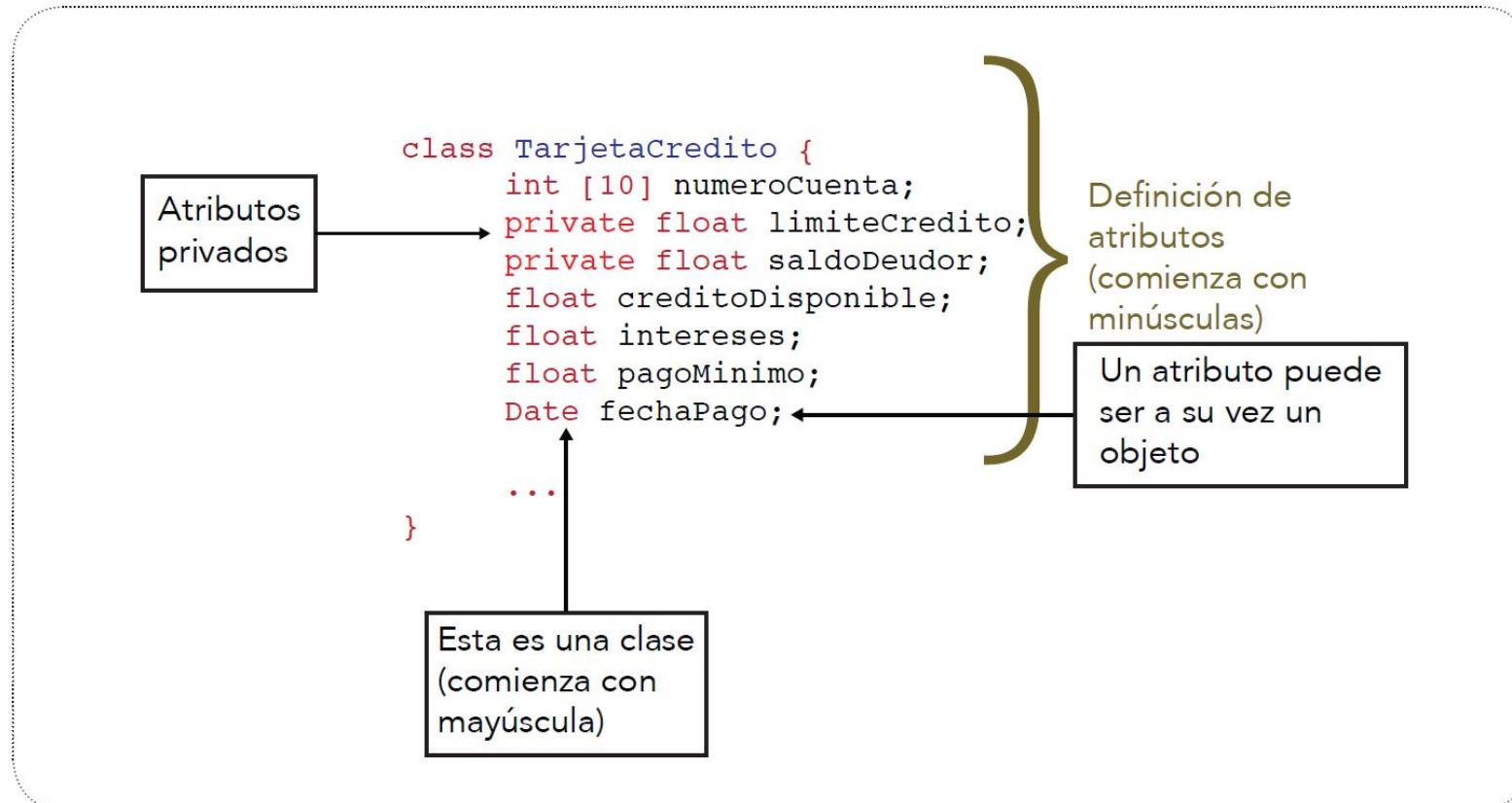


Figura II-3. Ejemplos de atributos de la clase TarjetaCredito

Los métodos

Los métodos constituyen el comportamiento de los objetos de la clase. La definición de un método es muy similar a la de una función. Los métodos públicos son las *operaciones* que los objetos externos realizan con el objeto en cuestión. Los métodos privados son las *operaciones internas* que no se pueden invocar desde el exterior, pero sí desde otro método dentro de la clase.

La sintaxis para escribir un método público es la siguiente:

```
public <TipoDatoRetorno> <NombreMetodo> ( <listaParametros> ) {  
    ...  
}
```

Y para escribir un método privado es similar, sólo cambia la primera palabra:

```
private <TipoDatoRetorno> <NombreMétodo> ( <listaParametros> ) {  
    ...  
}
```

Los métodos privados son auxiliares de los públicos. Se escriben para estructurar mejor el código de la clase. Por convención, los nombres de los métodos son verbos en infinitivo y escritos en minúsculas. En el caso de nombres compuestos, se usan mayúsculas para la inicial de cada palabra intermedia.

En el siguiente código se declara la clase `Ejemplo`, con los atributos `x` y `a`. Esta clase también contiene los métodos `hacerAlgo()`, `suma()` e `imprimir()`.

```
public class Ejemplo {  
    public int x;  
    public int a;  
  
    public void hacerAlgo() {  
        x = 1 + a * 3;  
    }  
  
    public int suma() {  
        return x + a;  
    }  
  
    public void imprimir() {  
        System.out.println( "x= " + x + " a= " + a + "\n" );  
    }  
}
```

Crear un objeto

Declaremos que los objetos `e` y `f` son de la clase `Ejemplo`:

```
Ejemplo e;  
Ejemplo f;
```

Declarar las variables `e` y `f` de la clase `Ejemplo` no implica que se crearon los objetos. En realidad, hasta aquí, sólo se crean apuntadores a objetos de esta clase. Para crear un objeto, se usa el operador `new`. El operador `new` crea un nuevo objeto de la clase especificada (alojando suficiente memoria para almacenar los datos del objeto) y regresa una *referencia* al objeto que se creó. Esta *referencia* es, en realidad, un *apuntador oculto*. En Java, el manejo de apuntadores y el desalojo de memoria se hacen automáticamente para evitar olvidos de los programadores y, por ende, el uso explícito de apuntadores se omite. El código para crear las instancias de los objetos de la clase `Ejemplo` sería:

```
Ejemplo e;  
Ejemplo f;  
e = new Ejemplo();  
f = new Ejemplo();
```

En general, la manera de crear un objeto de una clase X es la siguiente:

```
// primero declarar el objeto indicando su clase
<Clase> <nombreObjeto>;
// después crear el objeto
<nombreObjeto> = new <Clase>();
```

Lo anterior se crea en un solo enunciado:

```
<Clase> <nombreObjeto> = new <Clase>();
```

Los objetos son punteros. Al declarar un objeto, estamos declarando un puntero que, inicialmente, apuntará a una dirección de memoria nula (`null`). El objeto no se podrá utilizar mientras que no apunte a una dirección de memoria válida.

En el siguiente programa, declaramos un objeto e intentamos utilizarlo sin haberlo creado. Al ejecutarlo obtendremos un error de tipo `NullPointerException`.

La figura II-4 ilustra lo que es una *referencia* al objeto. Con `new` se deposita en las variables `e` y `f` un apuntador a la clase `Ejemplo`. Se dice que las variables `e` y `f` son una *referencia* al objeto o, dicho de otra manera, a la dirección de memoria que le fue asignada por el sistema operativo de la memoria *heap* (o pedacería de memoria).

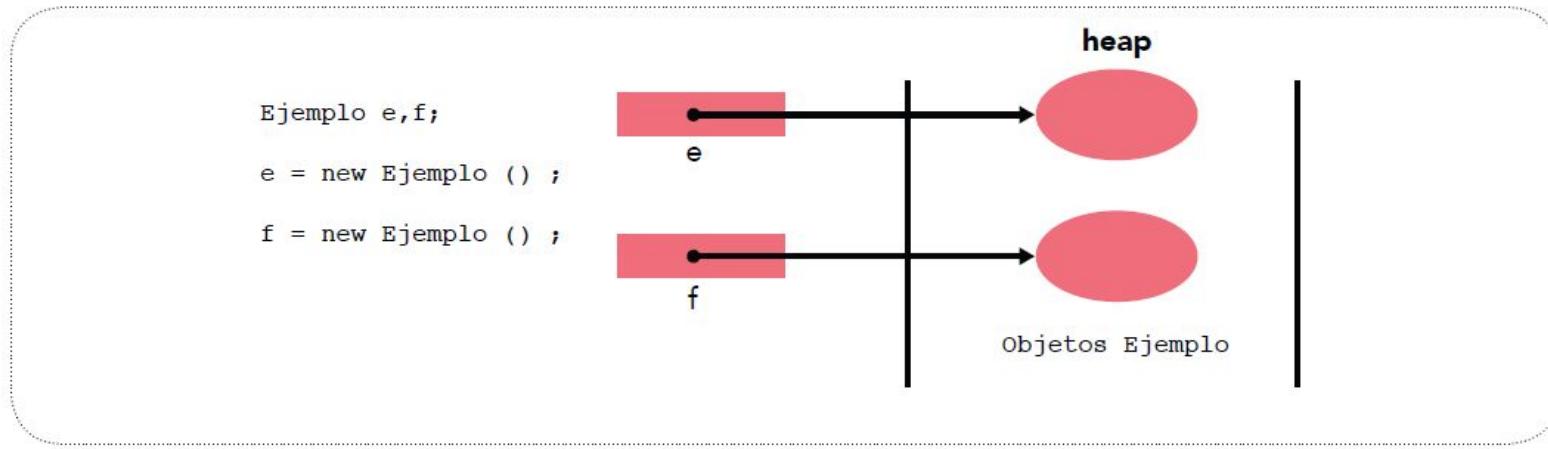


Figura II-4. Un objeto se crea con el operador new

En la práctica, sólo se dice que `e` y `f` son objetos de la clase `Ejemplo`. También podemos crear los objetos `e` y `f` en un solo paso:

```
Ejemplo e = new Ejemplo();
Ejemplo f = new Ejemplo();
```

Para tener acceso a los atributos y a los métodos del objeto, se utiliza el operador punto “.”. La sintaxis de la notación punto de Java para acceder a un método es la siguiente:

```
<referenciaObjeto>.<nombreMetodo>( <listaParametros> )
```

Entonces, la forma de ejecutar un método del objeto es:

```
nombreObjeto.nombreMetodo( );
```

Por ejemplo, para llamar al método `hacerAlgo()` del objeto `e` se escribe:

```
e.hacerAlgo();
```

A la clase que tiene el método main se le llama *clase principal*. En el siguiente ejemplo tenemos la clase PruebaEjemplo, cuyo método main() hace uso de dos objetos de la clase Ejemplo,

```
public class PruebaEjemplo {  
    public static void main( String[ ] args ) {  
        Ejemplo e;  
        Ejemplo f;  
        e = new Ejemplo(); // instancia "e" de la clase Ejemplo  
        f = new Ejemplo(); // instancia "f" de la clase Ejemplo  
        e.a = 7;  
        e.hacerAlgo();  
        f.x = e.suma();  
        f.a = f.x + e.a;  
        e.a = f.suma();  
        e.imprimir();  
        f.imprimir();  
    }  
}
```

El método `main()` debe ser siempre estático. La palabra `static` en la definición de un método implica que no es necesario crear el objeto para llamar al método. Así, la JVM puede invocarlo. Si queremos evitar que algunos datos de la clase se modifiquen desde afuera de ésta, conviene declarar a los atributos en cuestión como *privados*. La sintaxis para declarar un atributo como privado, es la siguiente:

```
private <tipo> <nombreAtributo>;  
  
public class Ejemplo {  
    int x;  
    private int a; // el atributo "a" es privado  
  
    public void hacerAlgo() {  
        x = 1 + a * 3;  
    }  
  
    public int suma() {  
        return x + a;  
    }  
  
    public void imprimir() {  
        System.out.println( "x= " + x + " a= " + a + "\n" );  
    }  
}
```

Entonces, no será posible tener acceso al atributo a desde el exterior de la clase y sólo los métodos de la clase tienen acceso a este atributo. Así es que, cuando los objetos e y f en la clase PruebaEjemplo tratan de leer o modificar el atributo a, ocurre un error de compilación.

```
public class PruebaEjemplo {  
    public static void main( String[ ] args ) {  
        Ejemplo e;  
        Ejemplo f;  
        e = new Ejemplo();  
        f = new Ejemplo();  
        e.a = 7; ←  
        e.hacerAlgo();  
        f.x = e.suma(); ←  
        f.a = f.x + e.a; ←  
        e.imprimir();  
        f.imprimir();  
    }  
}
```

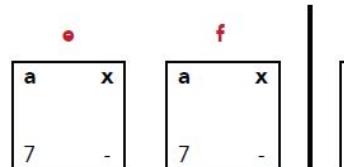
¡Error! El acceso al atributo a
no está permitido.

EJERCICIOS

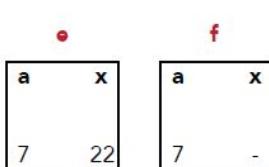
Ejercicio 1:

Ejercicio 2.1. ¿Qué es lo que imprimen los métodos `e.imprimir()` y `f.impri-
mir()` invocados desde la clase `PruebaEjemplo`?

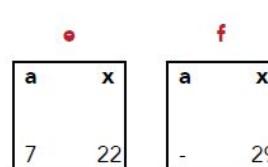
En la figura II-5 se muestra paso a paso lo que sucede con cada uno de los atributos de los objetos **e** y **f**.



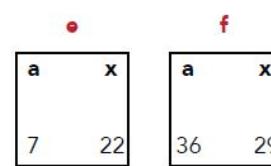
1)e.a=7



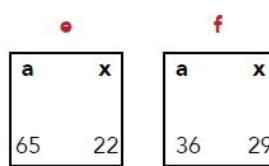
2)e.hacerAlgo()
e.x=1+a*3
e.x=1+7*3



3)f.x=e.sumar()
f.x=e.x+e.a
f.x=22+7=29



4)f.a=f.x+e.a
f.a=29+7



5)e.a=f.sumar()
e.a=f.x+f.a
e.a=29+36=65

Solucion Ejercicio 1

Figura II-5. Contenidos parciales de los atributos en los objetos **e** y **f**

El resultado de **e.imprimir()** y **f.imprimir()** es el contenido final de los objetos **e** y **f** de la figura II-5, es decir, para **e**: **a = 65, x = 22**; y para **f: a = 36, x = 29**.

Ejercicio 2.2. Declarar la clase `Cuenta` con los atributos `nombre`, `saldo`, `numero` y `tipo`, y con los métodos `depositar`, que recibe como parámetro la cantidad a depositar, y `retirar`, que recibe como parámetro la cantidad a retirar. Sólo se efectuará el retiro si el saldo es mayor o igual a la cantidad a retirar. Escribir un método que imprima los datos del objeto.

Solución Ejercicio 2.2:

La figura II-6 es la representación de la clase Cuenta.

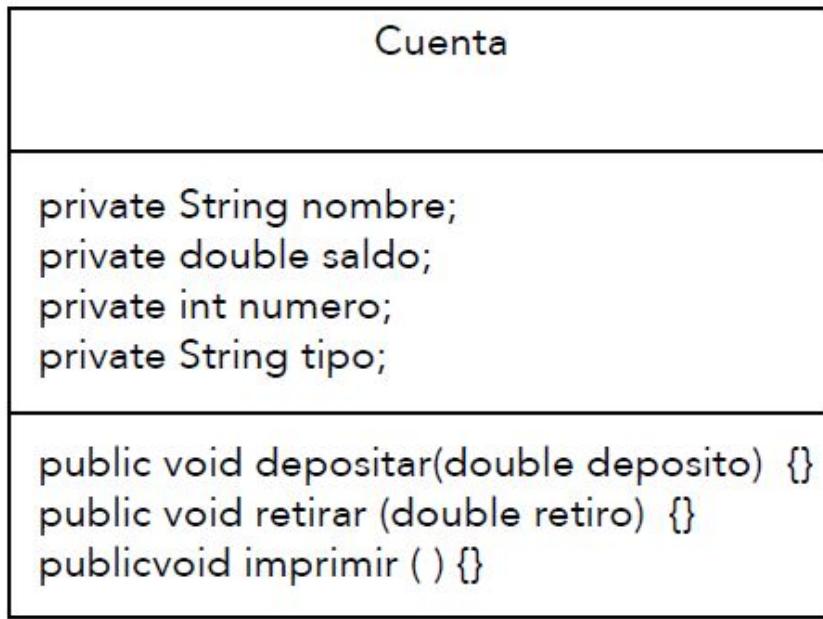


Figura II-6. Representación UML de la clase Cuenta

Solucion en JAVA:

```
public class Cuenta {  
    public String nombre;  
    public double saldo;  
    public int numero;  
    public String tipo;  
  
    public void depositar( double deposito ) {  
        saldo = saldo + deposito;  
    }  
  
    public void retirar( double retiro ) {  
        if ( saldo >= retiro ) {  
            saldo = saldo - retiro;  
        }  
    }  
  
    public void imprimir() {  
        System.out.println( "La cuenta es de: " + nombre  
                            + ", número: " + numero  
                            + ". Es una cuenta de " + tipo  
                            + ", con saldo: " + saldo + "\n" );  
    }  
}
```

Ejercicio 2.3. Crear los objetos `cuentaCredito` y `cuentaDebito` en la clase Principal. Al objeto `cuentaCredito` ponerlo a nombre de Pedro Sánchez, con saldo de 1500, con número de cuenta 244513, indicando que es una cuenta de crédito. Al objeto `cuentaDebito` ponerlo a nombre de Pablo García, con saldo de 7800, con número de cuenta 273516, indicando que es una cuenta de débito. ¿Qué es lo que se imprime?

```
public class Principal {
    public static void main( String[] args ) {
        Cuenta cuentaCredito;
        Cuenta cuentaDebito;

        // Creamos los objetos
        cuentaCredito = new Cuenta();
        cuentaDebito = new Cuenta();

        ...

        cuentaCredito.imprimir();
        cuentaDebito.imprimir();
    }
}
```

Solución:

```
public class Principal {
    public static void main( String[] args ) {
        Cuenta cuentaCredito;
        Cuenta cuentaDebito;

        // Creamos los objetos
        cuentaCredito = new Cuenta();
        cuentaDebito = new Cuenta();

        cuentaCredito.nombre = "Pedro Sanchez";
        cuentaCredito.saldo = 1500;
        cuentaCredito.numero = 244513;
        cuentaCredito.tipo = "crédito";

        cuentaDebito.nombre = "Pablo Garcia";
        cuentaDebito.saldo = 7800;
        cuentaDebito.numero = 273516;
        cuentaDebito.tipo = "débito";

        cuentaCredito.imprimir();
        cuentaDebito.imprimir();
    }
}
```

Solución:

```
public class Principal {  
    public static void main( String[] args ) {  
        Cuenta cuentaCredito;  
        Cuenta cuentaDebito;  
  
        // Creamos los objetos  
        cuentaCredito = new Cuenta();  
        cuentaDebito = new Cuenta();  
  
        cuentaCredito.nombre = "Pedro Sanchez";  
        cuentaCredito.saldo = 1500;  
        cuentaCredito.numero = 244513;  
        cuentaCredito.tipo = "crédito";  
  
        cuentaDebito.nombre = "Pablo Garcia";  
        cuentaDebito.saldo = 7800;  
        cuentaDebito.numero = 273516;  
        cuentaDebito.tipo = "débito";  
  
        cuentaCredito.imprimir();  
        cuentaDebito.imprimir();  
    }  
}
```

Ejercicio 2.3.a. Determina, sin ver la solución, cuál será la salida del programa anterior.

Solución:

La cuenta es de: Pedro Sanchez, número: 244513. Es una cuenta de crédito, con saldo: 1500.

La cuenta es de: Pablo Garcia, número: 273516. Es una cuenta de débito, con saldo: 7800.

Ejercicio 2.4. Declarar los atributos `saldo` y `numero` de la clase `Cuenta`, como privados. Declarar también los métodos necesarios para asignarles un valor y para obtener el valor de cada cual.

```
public class Cuenta {  
    public String nombre;  
    private double saldo;  
    private int numero;  
    public String tipo;  
  
    public void setSaldo( double s ) {  
        saldo = s;  
    }  
  
    public double getSaldo() {  
        return saldo;  
    }  
  
    public void setNumero( int num ) {  
        numero = num;  
    }  
  
    public int getNumero() {  
        return numero;  
    }  
  
    public void depositar( double deposito ) {  
        saldo = saldo + deposito;  
    }  
  
    public void retirar( double retiro ) {  
        if ( saldo >= retiro ) {  
            saldo = saldo - retiro;  
        }  
    }  
  
    public void imprimir() {  
        System.out.println( "La cuenta es de: " + nombre  
                           + ", número: " + numero  
                           + ". Es una cuenta de " + tipo  
                           + ", con saldo: " + saldo + "\n" );  
    }  
}
```

Ejercicio 2.5. Modificar el programa del ejercicio 2.3, teniendo en cuenta que ahora los atributos saldo y numCuenta son privados.

```
public class Principal {
    public static main(String args[]) {
        Cuenta cuentaCredito;
        Cuenta cuentaDebito;

        // Creamos los objetos
        cuentaCredito = new Cuenta();
        cuentaDebito = new Cuenta();

        cuentaCredito.nombre = "Pedro Sanchez";
        cuentaCredito.setSaldo(1500.0);
        cuentaCredito.setNumero(244513);
        cuentaCredito.tipo = "crédito";

        cuentaDebito.nombre = "Pablo Garcia";
        cuentaDebito.setSaldo(7800.0);
        cuentaDebito.setNumero(273516);
        cuentaDebito.tipo = "débito";

        cuentaCredito.imprimir();
        cuentaDebito.imprimir();
    }
}
```

La palabra reservada this y los métodos "getters" y "setters"

En Java se utiliza la palabra reservada `this` para denotar desde el interior de un objeto una referencia al propio objeto..

Cuando los atributos son privados, como en el caso de `saldo` y `numero`, entonces se codifican los métodos `getSaldo()` y `getNumero()` para que otros objetos tengan acceso a su valor. A este tipo de métodos se les conoce como **getters**. Si además queremos que los otros objetos también puedan modificar el valor de estos atributos, entonces usamos los métodos **setters**, que en este caso son `setSaldo()` y `setNumero()`:

```
public void setSaldo(double s) {  
    saldo = s;  
}  
  
public void setNumero(int num) {  
    numero = num;  
}
```

Nótese que para diferenciar entre el valor enviado como parámetro y el nombre del atributo usamos `s` y `num`, respectivamente. Declarar los atributos como privados permite *encapsular* los datos del objeto.

El encapsulamiento sirve para proteger los datos de los objetos y se logra declarando los atributos de una clase como **private** y codificando métodos especiales para controlar el acceso. La manera de acceder a los atributos desde afuera de la clase es por medio de los métodos *getter* y la manera de modificar los atributos desde afuera de la clase es usando los métodos *setter*.

Como el encapsulamiento es una práctica común, muchos ambientes de desarrollo orientado a objetos codifican automáticamente los métodos *getters* y *setters* con las siguientes reglas:

Como el encapsulamiento es una práctica común, muchos ambientes de desarrollo orientado a objetos codifican automáticamente los métodos *getters* y *setters* con las siguientes reglas:

1. Tanto *getters* como *setters* son públicos.
2. Los *getters* no reciben parámetros y el tipo de dato que regresan es el mismo que el del atributo correspondiente. Su nombre comienza con **get** seguido del nombre del atributo pero iniciando con mayúscula y regresan el valor del atributo. Por ejemplo:

```
public double getSaldo() {  
    return saldo;  
}
```

```
public int getNumero() {  
    return numero;  
}
```

3. Los *setters* reciben como parámetro el mismo tipo de dato que el del atributo. El nombre de los métodos *setters* se construye de forma análoga a la de los *getters*, pero iniciando con la palabra **set**, y asignan al atributo el valor del parámetro recibido. El parámetro recibido tiene el mismo nombre que el atributo. Para diferenciar entre el valor enviado como parámetro y el nombre del atributo se utiliza la palabra **this**, de tal forma que **this.nombreAtributo** se refiere al atributo del objeto. Por ejemplo:

```
public void setSaldo( double saldo ) {  
    // el parametro saldo se asigna al atributo this.saldo  
    this.saldo = saldo;  
}  
  
public void setNumero( int numero ) {  
    // el parametro numero se asigna al atributo  
    // this.numero  
    this.numero = numero;  
}
```

```
1 package Tema1;
2 public class Fecha {
3     private int dia;
4     private int mes;
5     private int anio;
6
7     public int getDia() {
8         // retorna el valor de la variable dia
9         return dia;
10    }
11    public void setDia(int dia) {
12        // asigna el valor del parametro a la variable dia
13        this.dia = dia;
14    }
15    public int getMes() {
16        return mes;
17    }
18    public void setMes(int mes) {
19        this.mes = mes;
20    }
21    public int getAnio() {
22        return anio;
23    }
24    public void setAnio(int anio) {
25        this.anio = anio;
26    }
27 }
```

Otro ejemplo usando Getters y Setters.

Primero escribimos la clase con los atributos privados, los getters y los setters. Y en la siguiente placa escribimos el método main.

```
1 package Tema1;
2 public class TestFecha {
3     public static void main(String[] args) {
4         Fecha f = new Fecha();
5         f.setDia(2);
6         f.setMes(10);
7         f.setAnio(1970);
8         // imprimimos el dia
9         System.out.println("Dia=" + f.getDia());
10        // imprimimos el mes
11        System.out.println("Mes=" + f.getMes());
12        // imprimimos el anio
13        System.out.println("Anio=" + f.getAnio());
14        // imprimimos la fecha
15        System.out.println(f);
16    }
17 }
```

Contrariamente a lo que se podría esperar, la salida de: `System.out.println(f);` no será una fecha con el formato que, habitualmente, usamos para representarlas. La salida será algo así: `libro.cap14.fechas.Fecha@360be0`. ¿Por qué? Porque `System.out.println` no puede saber de antemano cómo queremos que imprima los objetos de las clases que nosotros programamos. Para solucionarlo debemos sobrescribir el método `toString` que heredamos de la clase `Object`.

Los métodos constructores

Un constructor es un método especial, que se invoca al momento de crear un objeto, es decir, cuando se instancia la clase. El método constructor *tiene exactamente el mismo nombre que el de la clase que construye*. Por ejemplo:

```
Reloj r = new Reloj();
```

Los métodos constructores tienen la tarea de dejar al objeto listo para su uso. Por ejemplo, en el reloj, el constructor debería por lo menos inicializar la hora a las 00:00:00. Además de la inicialización de variables (incluyendo el alojamiento de memoria dinámico para éstas), el constructor *debe encargarse de crear los objetos que forman parte del objeto*.

Todas las clases tienen un constructor por omisión, con el cuerpo vacío y sin parámetros. El constructor por omisión siempre existe y evita problemas cuando no se definen explícitamente constructores en una clase. Definamos un constructor para la clase **Cuenta**:

```
public class Cuenta {  
    String nombre;  
    double saldo;  
    int numero;  
    String tipo;  
  
    public Cuenta( String n, double s, int num, String t ) {  
        nombre = n;  
        saldo = s;  
        numero = num;  
        tipo = t;  
    }  
  
    public void depositar( double deposito ) {  
        saldo = saldo + deposito;  
    }  
  
    public void retirar( double retiro ) {  
        if ( saldo >= retiro ) {  
            saldo = saldo - retiro;  
        }  
    }  
  
    public void imprimir() {  
        System.out.println( "La cuenta es de: " + nombre  
                            + ", número: " + numero  
                            + ". Es una cuenta de " + tipo  
                            + ", con saldo: " + saldo + "\n" );  
    }  
}
```

Ahora, para construir un objeto de la clase **Cuenta** será necesario proporcionar los datos de la cuenta por medio de los parámetros del constructor. Por ejemplo, se crea el objeto **cuentaCredito**, desde la clase **Principal**, definiendo desde ese momento que dicha cuenta esté a nombre de Pedro Sánchez, con saldo de 1500, número de cuenta 244513 e indicando que es una cuenta de crédito. Otro objeto **cuentaDebito** puede crearse a nombre de Pablo García, con saldo de 7800, número de cuenta 273516 e indicando que es una cuenta de débito. Para lograr lo anterior, hacemos lo siguiente:

```
public class Principal {  
    public static void main(String[] args) {  
        Cuenta cuentaCredito;  
        Cuenta cuentaDebito;  
  
        // Creamos los objetos  
        cuentaCredito = new Cuenta( "Pedro Sanchez", 1500, 244513,  
                                    "crédito" );  
        cuentaDebito = new Cuenta( "Pablo Garcia", 7800, 273516,  
                                    "débito" );  
        ...  
    }  
}
```

this en los métodos constructores

La palabra reservada **this** también es útil para dar el mismo nombre de los atributos a los parámetros que recibe un método constructor. Enseguida presentamos el código del constructor de la clase **Cuenta** utilizando la palabra **this** para indicar que se trata del atributo de la clase:

```
public Cuenta( String nombre, double saldo, int numero,  
               String tipo) {  
    this.nombre = nombre;  
    this.saldo = saldo;  
    this.numero = numero;  
    this.tipo = tipo;  
}
```

Al utilizar **this.nombreAtributo**, evitamos las ambigüedades cuando los parámetros tienen el mismo nombre que los atributos.

```
package libro.cap14.fechas;

public class Fecha
{
    private int dia;
    private int mes;
    private int anio;

    // constructor
    public Fecha(int d, int m, int a)
    {
        dia = d;
        mes = m;
        anio = a;
    }

    // :
    // setters y getters...
}
```

Es importante tener en cuenta que al declarar, explícitamente, un constructor perdemos el constructor por defecto. Por lo tanto, ahora, el siguiente código no compilará:

```
// esto ahora no compila porque en la clase Fecha
// no existe un constructor que no reciba argumentos
Fecha f = new Fecha();
```

En cambio, podremos crear fechas especificando los valores de sus atributos.

```
// Ahora si... creamos la fecha del 2 de octubre de 1970
Fecha f = new Fecha(2, 10, 1970);
```

Herencia y Sobrescritura de Métodos

Una de las principales características de la programación orientada a objetos es la “herencia” que permite definir clases en función de otras clases ya existentes. Es decir, una clase define atributos y métodos y, además, hereda los atributos y métodos que define su “padre” o “clase base”.

Si bien este tema lo estudiaremos en detalle más adelante, llegamos a un punto en el que debemos saber que en Java, directa o indirectamente, todas las clases heredan de una clase base llamada `Object`. No hay que especificar nada para que esto ocurra. Siempre es así.

La herencia es transitiva. Sean las clases A, B y C, si A hereda de B y B hereda de C, entonces A hereda de C.

Pensemos en las clases `Empleado` y `Persona`. Evidentemente, un empleado primero es una persona ya que este tendrá todos los atributos de `Persona` (que podrían ser nombre, fechaNacimiento, DNI, etc.) y luego los atributos propios de un empleado (por ejemplo, matricula, sector, sueldo, etc.). Decimos entonces que la clase `Empleado` hereda de la clase `Persona` y (si `Persona` no hereda de ninguna otra clase) entonces `Persona` hereda de `Object`. Así, un empleado es una persona y una persona es un `object`; por lo tanto, por transitividad, un empleado también es un `object`.

Es muy importante saber que todas las clases heredan de la clase base `Object` ya que los métodos definidos en esta clase serán comunes a todas las demás (las que vienen con el lenguaje y las que programemos nosotros mismos).

En este momento nos interesa estudiar dos de los métodos que heredamos de `Object`: el método `toString` y el método `equals`.

```
package libro.cap14.fechas;

public class Fecha
{
    private int dia;
    private int mes;
    private int anio;

    public int getDia()
    {
        // retorna el valor de la variable dia
        return dia;
    }

    public void setDia(int dia)
    {
        // asigna el valor del parametro a la variable dia
        this.dia = dia;
    }

    public int getMes()
    {
        return mes;
    }

    public void setMes(int mes)
    {
        this.mes = mes;
    }

    public int getAnio()
    {
        return anio;
    }

    public void setAnio(int anio)
    {
        this.anio = anio;
    }
}
```

Ahora la clase provee métodos a través de los cuales podemos acceder a los atributos de sus objetos para asignar y/o consultar sus valores, como vemos en el siguiente código:

```
package libro.cap14.fechas;

public class TestFecha
{
    public static void main(String[] args)
    {
        Fecha f = new Fecha();
        f.setDia(2);
        f.setMes(10);
        f.setAnio(1970);

        // imprimimos el dia
        System.out.println("Dia="+f.getDia());

        // imprimimos el mes
        System.out.println("Mes="+f.getMes());

        // imprimimos el anio
        System.out.println("Anio="+f.getAnio());

        // imprimimos la fecha
        System.out.println(f);
    }
}
```

Contrariamente a lo que el lector podría esperar, la salida de:

```
System.out.println(f);
```

no será una fecha con el formato que, habitualmente, usamos para representarlas. La salida de esta línea de código será algo así:

```
libro.cap14.fechas.Fecha@360be0
```

¿Por qué? Porque `System.out.println` no puede saber de antemano cómo queremos que imprima los objetos de las clases que nosotros programamos. Para solucionarlo debemos sobrescribir el método `toString` que heredamos de la clase `Object`.

Sobrescribiendo El Método `toString`

Todas las clases heredan de `Object` el método `toString`; por esto, podemos invocar este método sobre cualquier objeto de cualquier clase. Tal es así que cuando hacemos:

```
System.out.println(obj);
```

siendo `obj` un objeto de cualquier clase, lo que implícitamente estamos haciendo es:

```
System.out.println( obj.toString() );
```

ya que `System.out.println` invoca el método `toString` del objeto que recibe como parámetro. Es decir, `System.out.println` “sabe” que cualquiera sea el tipo de datos del objeto, este seguro tendrá el método `toString`.

Por este motivo, en la clase `Fecha` podemos sobrescribir el método `toString` para indicar el formato con el que queremos que se impriman las fechas.

Decimos que sobrescribimos un método cuando programamos en una clase el mismo método que heredamos de nuestra clase base.

A continuación, veremos cómo sobrescribir el método `toString` en la clase `Fecha`.

```
package libro.cap14.fechas;

public class Fecha
{
    private int dia;
    private int mes;
    private int anio;

    // sobrescribimos el metodo toString (lo heredamos de Object)
    public String toString()
    {
        // retorna una cadena tal como queremos que se vea la fecha
        return dia+"/"+mes+"/"+anio;
    }

    /**
     * setters y getters...
     */
}
```

Ahora simplemente podremos imprimir el objeto `f` y la salida será la esperada: "2/10/1970" (considerando el ejemplo que analizamos más arriba).

Sobrescribiendo El método equals

Otro de los métodos que heredamos de la clase `Object` y que se utiliza para comparar objetos es `equals`. El lector recordará que utilizamos este método para comparar cadenas. Pues bien, la clase `String` lo hereda de `Object` y lo sobrescribe de forma tal que permite determinar si una cadena es igual a otra comparando uno a uno sus caracteres. En nuestro caso tendremos que sobrescribir el método `equals` para indicar si dos fechas son iguales o no.

```
package libro.cap14.fechas;

public class Fecha
{
    private int dia;
    private int mes;
    private int anio;

    // sobrescribimos el metodo equals que heredamos de Object
    public boolean equals(Object o)
    {
        Fecha otra = (Fecha)o;
        return (dia==otra.dia) && (mes==otra.mes) && (anio==otra.anio);
    }

    // :
    // setters y getters...
    // toString...
}
```

Repasso de lo visto hasta aquí:

Antes de seguir incorporando conceptos sería conveniente hacer un pequeño repaso de todo lo expuesto hasta el momento.

- Toda clase hereda, directa o indirectamente, de la clase base `Object`.
- Los métodos de la clase `Object` son comunes a todas las clases.
- De `Object` siempre heredaremos los métodos `toString` y `equals`.
- Podemos sobrescribir estos métodos para definir el formato de impresión de los objetos de nuestras clases y el criterio de comparación respectivamente.
- “Sobrescribir” significa reescribir el cuerpo de un método que estamos heredando sin modificar su prototipo.
- Los objetos no pueden ser utilizados hasta tanto no hayan sido creados.
- Para crear objetos utilizamos el constructor de la clase.
- Todas las clases tienen, al menos, un constructor.
- Podemos programar nuestro constructor o utilizar el constructor por defecto.
- Al programar explícitamente un constructor entonces “perdemos” el constructor nulo o por defecto.

Ejemplo: Compara dos fechas ingresadas por el usuario.

En el siguiente programa, le pedimos al usuario que ingrese dos fechas y las compararemos utilizando su método `equals`.

```
1 package Tema3;
2 import java.util.Scanner;
3 public class TestFecha3 {
4     public static void main(String[] args) {
5         Scanner scanner = new Scanner(System.in);
6         // el usuario ingresa los datos de la fecha
7         System.out.print("Ingrese Fecha1 (dia, mes y anio): ");
8         int dia = scanner.nextInt();
9         int mes = scanner.nextInt();
10        int anio = scanner.nextInt();
11        // creamos un objeto de la clase Fecha
12        Fecha f1 = new Fecha(dia, mes, anio);
13        // el usuario ingresa los datos de la fecha
14        System.out.print("Ingrese Fecha2 (dia, mes y anio): ");
15        dia = scanner.nextInt();
16        mes = scanner.nextInt();
17        anio = scanner.nextInt();
18        // creamos un objeto de la clase Fecha
19        Fecha f2 = new Fecha(dia, mes, anio);
20        System.out.println("Fecha 1 = " + f1);
21        System.out.println("Fecha 2 = " + f2);
22        if (f1.equals(f2)) {
23            System.out.println("Son iguales!");
24        } else {
25            System.out.println("Son distintas...");
26        }
27    }
28 }
```

Agregaremos entonces en la clase Fecha un constructor que reciba tres enteros (día, mes y año) y los asigne a sus atributos.

```
package libro.cap14.fechas;

public class Fecha
{
    private int dia;
    private int mes;
    private int anio;

    // constructor
    public Fecha(int d, int m, int a)
    {
        dia = d;
        mes = m;
        anio = a;
    }

    // :
    // setters y getters...
    // toString...
    // equals...
    // :
}
```

Como podemos ver, en el constructor, recibimos los tres valores y los asignamos a los atributos correspondientes. También podríamos haber invocado, dentro del código del constructor, a los setters. Por ejemplo, en lugar de: dia = d, hacer: setDia(d).

14.2.8 Convenciones de nomenclatura

Antes de seguir avanzando considero conveniente explicar las convenciones que comúnmente son aceptadas y aplicadas a la hora de asignar nombres a las clases, métodos, atributos, constantes y variables.

14.2.8.1 Los nombres de las clases

Las clases siempre deben comenzar con mayúscula. En el caso de tener un nombre compuesto por más de una palabra, cada inicial también debe estar en mayúscula. Por ejemplo:

```
public class NombreDeLaClase
{
    // :
}
```

14.2.8.2 Los nombres de los métodos

Los métodos siempre deben comenzar en minúscula. En el caso de tener un nombre compuesto por más de una palabra, cada inicial debe comenzar en mayúscula salvo, obviamente, la primera.

```
public void nombreDelMetodo ()
{
    //...
}
```

14.2.8.3 Los nombres de los atributos

Para los atributos se utiliza la misma convención que definimos para los métodos: comienzan en minúscula y, si el nombre consta de más de una palabra, entonces cada inicial, salvo la primera, debe ir en mayúscula.

```
public class Persona
{
    private String nombre;
    private Date fechaDeNacimiento;

    // :
}
```

14.2.8.4 Los nombres de las variables de instancia

Las variables de instancia que no sean consideradas como atributos pueden definirse a gusto del programador siempre y cuando no comiencen con mayúscula. Algunos programadores utilizan la “notación húngara”, que consiste en definir prefijos que orienten sobre el tipo de datos de la variable. Así, si tenemos una variable `contador` de tipo `int`, según esta notación, la llamaremos `iContador` y si tenemos una variable `fin` de tipo `boolean` será `bFin`.

14.2.8.5 Los nombres de las constantes

Para las constantes se estila utilizar solo letras mayúsculas. Si el nombre de la constante está compuesto por más de una palabra, entonces debemos utilizar el “guión bajo” o *underscore* para separarlas.

```
public static final int NOMBRE_DE_LA_CONSTANTE = 1;
```

14.2.9 Sobrecarga de métodos

En el capítulo anterior, vimos que al método `indexOf` de la clase `String` le podemos pasar tanto un argumento de tipo `char` como uno de tipo `String`. Veamos:

```
String s = "Esto es una cadena";
int pos1 = s.indexOf("e"); // retorna 5
int pos2 = s.indexOf('e'); // retorna 5
```

En este código invocamos al método `indexOf` primero pasándole un argumento de tipo `String` y luego, uno de tipo `char`. Ambas invocaciones son correctas, funcionan bien y son posibles porque el método `indexOf` de la clase `String` está “sobrecargado”. Es decir, el mismo método puede invocarse con diferentes tipos y/o cantidades de argumentos.

Decimos que un método está sobrecargado cuando admite recibir más de una combinación de tipos y/o cantidades de argumentos. Esto se logra escribiendo el método tantas veces como tantas combinaciones diferentes queremos que el método admita.

A continuación,
veremos cómo
sobrecargar el
constructor de la clase
Fecha para poder
crear fechas especifi-
cando sus atributos o
bien sin especificar
ningún valor.

```
1 package Tema3;
2
3 public class Fecha {
4     private int dia;
5     private int mes;
6     private int anio;
7     // constructor
8     public Fecha(int d, int m, int a) {
9         dia = d;
10        mes = m;
11        anio = a;
12    }
13    // constructor vacio
14    public Fecha() {
15    }
16    // sobreescribimos el metodo toString()
17    public String toString() {
18        // retorna una cadena tal como queremos que se vea la fecha
19        return dia + "/" + mes + "/" + anio;
20    }
21    // sobrescribimos el metodo equals que heredamos de Object
22    public boolean equals(Object o) {
23        Fecha otra = (Fecha) o;
24        return (dia == otra.dia) && (mes == otra.mes) && (anio == otra.anio);
25    }
26    public int getDia() {
27        // retorna el valor de la variable dia
28        return dia;
29    }
30    public void setDia(int dia) {
31        // asigna el valor del parametro a la variable dia
32        this.dia = dia;
33    }
}
```

Luego, las siguientes líneas de código son correctas:

```
// creamos una fecha indicando los valores iniciales  
Fecha f1 = new Fecha(2,10,1970);  
  
// creamos una fecha sin indicar valores iniciales  
Fecha f2 = new Fecha();  
f2.setDia(4);      // asignamos el dia  
f2.setMes(6);      // asignamos el mes  
f2.setAnio(2008); // asignamos el anio
```

Es importante no confundir “sobrecarga” con “sobrescritura”:

- Sobrecargamos un método cuando lo programamos más de una vez, pero con diferentes tipos y/o cantidades de parámetros.
- Sobrescribimos un método cuando el método que estamos programando es el mismo que heredamos de su padre. En este caso, tenemos que respetar su encabezado (cantidades y tipos de parámetros y tipo del valor de retorno) ya que de lo contrario lo estaremos sobrecargando.

La propuesta ahora es hacer que la clase Fecha permita crear fechas a partir de una cadena de caracteres con este formato: “dd/mm/aaaa”. Para esto, tendremos que agregar (sobrecargar) un constructor que reciba como parámetro una cadena.

La estrategia será la siguiente: suponiendo que la cadena que recibimos como parámetro en el constructor es “15/06/1973”, entonces:

1. Ubicamos la posición de la primera ocurrencia de ' / ' (la llamaremos pos1).
2. Ubicamos la posición de la última ocurrencia de ' / ' (la llamaremos pos2).
3. Tomamos la subcadena ubicada entre 0 y pos1 (no inclusive), la convertimos a int y la asignamos al atributo dia.
4. Tomamos la subcadena ubicada entre pos1+1 y pos2 (no inclusive), la convertimos a int y la asignamos en el atributo mes.
5. Tomamos la subcadena ubicada a partir de pos2+1, la convertimos a int y la asignamos en el atributo anio.

```
1 package Tema4;
2 public class Fecha {
3     private int dia;
4     private int mes;
5     private int anio;
6
7     public Fecha(String s) {
8         // buscamos la primera ocurrencia de '/'
9         int pos1 = s.indexOf('/');
10        // buscamos la ultima ocurrencia de '/'
11        int pos2 = s.lastIndexOf('/');
12        // procesamos el dia
13        String sDia = s.substring(0, pos1);
14        dia = Integer.parseInt(sDia);
15        // procesamos el mes
16        String sMes = s.substring(pos1 + 1, pos2);
17        mes = Integer.parseInt(sMes);
18        // procesamos el anio
19        String sAnio = s.substring(pos2 + 1);
20        anio = Integer.parseInt(sAnio);
21    }
22    // constructor
23    public Fecha(int d, int m, int a) {
24        dia = d;
25        mes = m;
26        anio = a;
27    }
28    // constructor vacio
29    public Fecha() {
30    }
31
32    // sobreescribimos el metodo toString()
33    public String toString() {
34        // retorna una cadena tal como queremos que se vea la fecha
35        return dia + "/" + mes + "/" + anio;
36    }

```

```
1 package Tema4;
2 public class TestFecha {
3     public static void main(String[] args) {
4
5         // creamos una fecha a partir de los tres valores por separado
6         Fecha f = new Fecha(25, 10, 2004);
7         // creamos una fecha a partir de una cadena con formato dd/mm/aaaa
8         Fecha f2 = new Fecha("25/10/2004");
9
10        // imprimimos el dia
11        System.out.println("Dia=" + f.getDia());
12        // imprimimos el mes
13        System.out.println("Mes=" + f.getMes());
14        // imprimimos el anio
15        System.out.println("Anio=" + f.getAnio());
16
17        // imprimimos la fecha
18        System.out.println(f);
19        System.out.println(f2);
20        System.out.println("son iguales f1 y f2 ? " + f.equals(f2));
21    }
22 }
```

ENCAPSULAMIENTO

Uno de los objetivos que buscamos cuando programamos clases es encapsular la complejidad que emerge de las operaciones asociadas a sus atributos. Esto significa que debemos facilitarle la vida al programador que utilice objetos de nuestras clases exponiendo las operaciones que pueda llegar a necesitar, pero ocultando su complejidad.

Por ejemplo, una operación aplicable a una fecha podría ser sumarle o restarle días. Si definimos el método `addDias` en la clase `Fecha`, entonces quien utilice esta clase podrá sumarle días a sus fechas sin tener que conocer el algoritmo que resuelve el problema.

```
// creamos una fecha
Fecha f = new Fecha("23/12/1980");

// le sumamos 180 dias
f.addDias(180);

// mostramos como quedo la fecha luego de sumarle estos dias
System.out.println(f);
```

Para facilitar los cálculos y no perder el tiempo en cuestiones que aquí resultarían ajena s consideraremos que todos los meses tienen 30 días. Por lo tanto, en esta versión de la clase Fecha los años tendrán 360 días (12 meses de 30 días cada uno, $12 \times 30 = 360$).

Con estas consideraciones, el algoritmo para sumar días a una fecha consistirá en convertir la fecha a días, sumarle los días que corresponda y asignar los nuevos valores del día, mes y año en los atributos.

Entonces serán tres los métodos que vamos a programar:

- El método `addDias` será el método que vamos a “exponer” para que los usuarios de la clase puedan invocar y así sumarle días a sus fechas.
- Desarrollaremos también el método `fechaToDias` que retornará un entero para representar la fecha expresada en días. Este método no lo vamos a “exponer”. Es decir, no será visible para el usuario: será `private` (privado).
- Por último, desarrollaremos el método inverso: `diasToFecha` que, dado un valor entero que representa una fecha expresada en días, lo separará y asignará los valores que correspondan a los atributos `dia`, `mes` y `anio`. Este método también será `private` ya que no nos interesa que el usuario lo pueda invocar.

```
1 package Tema5;
2 public class Fecha {
3     private int dia;
4     private int mes;
5     private int anio;
6     // retorna la fecha expresada en dias
7     private int fechaToDias() {
8         // no requiere demasiada explicacion...
9         return anio * 360 + mes * 30 + dia;
10    }           // asigna la fecha expresada en dias a los atributos
11    private void diasToFecha(int i) {
12        // dividimos por 360 y obtenemos el año
13        anio = (int) i / 360;
14        // del resto o residuo de la division anterior
15        // podremos obtener el mes y el dia
16        int resto = i % 360;
17        // el mes es el resto dividido 30
18        mes = (int) resto / 30;
19        // el resto de la division anterior son los dias
20        dia = resto % 30;
21        // ajuste por si el dia quedo en cero
22        if (dia == 0) {
23            dia = 30;
24            mes--;
25        // ajuste por si el mes quedo en cero
26        if (mes == 0) {
27            mes = 12;
28            anio--;
29        }
30        public void addDias(int d) {
31            // convertimos la fecha a dias y le sumamos d
32            int sum = fechaToDias() + d;
33            // la fecha resultante (sum) se separa en dia, mes y año
34            diasToFecha(sum);
35        }
36        public Fecha(String s) {
37            // convertimos la cadena s a un numero de dias
38            int dias = Integer.parseInt(s);
39            addDias(dias);
40        }
41    }
42}
```

```
1 package Tema5;
2 import java.util.Scanner;
3
4 public class TestFecha {
5     public static void main(String[] args) {
6
7         Scanner scanner = new Scanner(System.in);
8         // el usuario ingresa los datos de la fecha
9         System.out.print("Ingrese Fecha (dd/mm/aaaa): ");
10        // leemos la fecha (cadena de caracteres) ingresada
11        String sFecha = scanner.nextLine();
12        // creamos un objeto de la clase Fecha
13        Fecha f = new Fecha(sFecha);
14        // lo mostramos
15        System.out.println("La fecha ingresada es: " + f);
16        // el usuario ingresa una cantidad de dias por sumar
17        System.out.print("Ingrese dias a sumar (puede ser negativo): ");
18        int diasSum = scanner.nextInt();
19        // le sumamos esos dias a la fecha
20        f.addDias(diasSum);
21        // mostramos la nueva fecha (con los dias sumados)
22        System.out.println("sumando " + diasSum + " dias, queda: " + f);
23    }
24 }
```

VISIBILIDAD DE LOS METODOS Y LOS ATRIBUTOS

En el ejemplo anterior, hablamos de “exponer” y “ocultar”. Esto tiene que ver con el nivel de visibilidad que podemos definir para los métodos y los atributos.

Aquellos métodos y atributos declarados como `public` (públicos) serán visibles desde cualquier otra clase. Por el contrario, los métodos y atributos declarados como `private` (privados) estarán encapsulados y solo podrán ser invocados y manipulados dentro de la misma clase.

En el caso de la clase `Fecha`, los atributos `dia`, `mes` y `anio` son `private`; por lo tanto, no pueden ser manipulados desde el programa principal.

```
public class ProgramaPrincipal
{
    public static void main(String args)
    {
        Fecha f = new Fecha();

        // error de compilacion ya que el atributo es private
        f.dia = 21;
    }
}
```

Lo mismo pasa con los métodos `fechaToDias` y `diasToFecha`. Estos métodos son privados y no se pueden invocar desde afuera de la clase.

```
public class ProgramaPrincipal
{
    public static void main(String args)
    {
        Fecha f = new Fecha("25/2/1980");

        // el metodo fechaToDias es private, error de compilacion
        int dias = f.fechaToDias();
    }
}
```

En cambio, podremos invocar cualquier método `public` como pueden ser los constructores, los métodos `toString`, `equals`, `addDias`, etcétera. Si la clase tuviese atributos declarados `public` entonces podríamos manipularlos en cualquier método de cualquier otra clase.

En general, se estila que los atributos sean `private` y los métodos `public`. Si para desarrollar un método complejo tenemos que dividirlo (estructurarlo) en varios métodos más simples, estos probablemente deban ser `private` para prevenir que el usuario los pueda invocar. Con esto, evitaremos confundir al programador que usa nuestras clases.

Existen otros dos niveles de visibilidad: `protected` y `friendly` pero los estudiaremos más adelante.

PAQUETES (Packages)

Los paquetes proporcionan un mecanismo que permite organizar las clases en función de un determinado criterio. Además, constituyen un *namespace* (espacio de nombres) que posibilita que varias clases tengan el mismo nombre siempre y cuando estén ubicadas en paquetes diferentes.

Para que una clase se ubique en un determinado paquete, la primera línea de código del archivo que la contiene debe ser `package`, seguida del nombre del paquete.

¿Qué hubiera sucedido si a nuestra clase `Fecha` la hubiésemos llamado `Date`? No sucedería absolutamente nada. Java trae por lo menos dos clases llamadas `Date`; la nuestra hubiera sido la tercera. Siendo así, ¿cómo pueden convivir tres clases con el mismo nombre? Es simple, están ubicadas en diferentes paquetes. Java provee las clases: `java.util.Date` y `java.sql.Date`. La nuestra, de haberla llamado `Date`, sería: `libro.cap14.fechas.Date`.

Físicamente, los paquetes son directorios o carpetas. Para ubicar una clase dentro del paquete `x`, el `.class` debe estar grabado en la carpeta `x` y el código fuente debe comenzar con la línea `package x`.

ESTRUCTURA DE PAQUETES

Los proyectos Java deben estar ubicados dentro de una carpeta que será la base de las carpetas que constituyen los paquetes. A esta carpeta la llamaremos *package root* (raíz de los paquetes).

Por ejemplo, la clase `Fecha` está en el paquete `libro.cap14.fechas`. Si consideramos como *package root* a la carpeta `c:\misproyectos\demolibro`, entonces la estructura del proyecto será la siguiente:

```
c:\  
  |_misproyectos\  
    |_demolibro\  
      |_src\  
        |_libro\  
          |_cap14\  
            |_fechas\  
              |_Fecha.java  
              |_TestFecha.java  
              |_ :  
  
      |_bin\  
        |_libro\  
          |_cap14\  
            |_fechas\  
              |_Fecha.class  
              |_TestFecha.class  
              |_ :  
            :
```

14.2.14 Las APIs (Application Programming Interface)

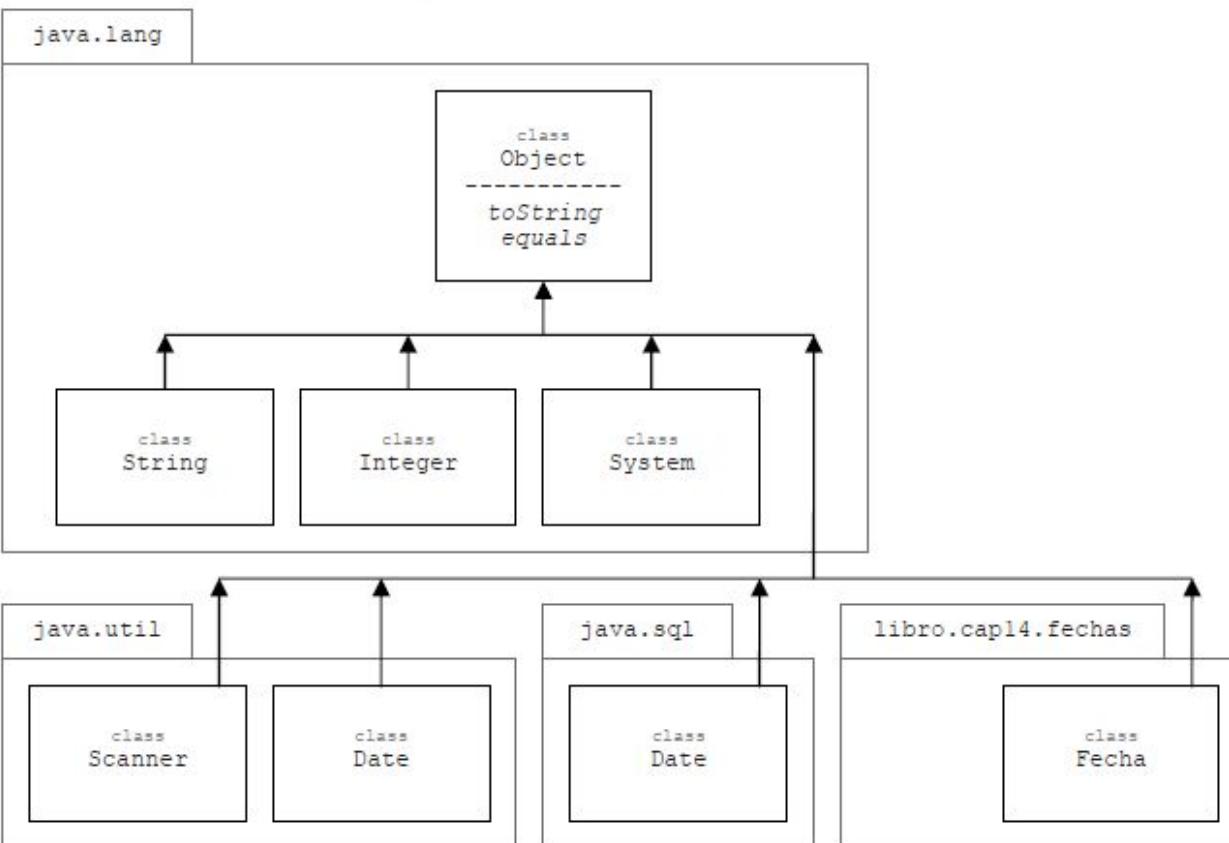
Dado que los paquetes agrupan clases funcionalmente homogéneas, es común decir que determinados paquetes constituyen una API.

Llamamos API al conjunto de paquetes (con sus clases y métodos) que están disponibles para utilizar en nuestros programas.

Todos los paquetes que provee Java constituyen la API de Java, pero podemos ser más específicos y, por ejemplo, decir que el paquete `java.net` constituye la API de *networking* y también podemos decir que el paquete `java.sql` constituye la API de acceso a bases de datos (o la API de JDBC).

En resumen, una API es un paquete o un conjunto de paquetes cuyas clases son funcionalmente homogéneas y están a nuestra disposición.

En este caso utilizaremos un diagrama de clases y paquetes para representar, gráficamente, lo que estudiamos más arriba. Los paquetes se representan como "carpetas" y las clases se representan en "cajas". Las flechas indican una relación de herencia entre clases. Si de una clase sale una flecha que apunta hacia otra clase será porque la primera es una subclase de la segunda.



Representación gráfica UML

14.2.16 Importar clases de otros paquetes

Nuestras clases pueden utilizar otras clases, independientemente del paquete en el que estén contenidas; simplemente “las importamos” y las usamos. Para esto, se utiliza la sentencia `import`.

Si observamos bien, en todos los programas que hemos desarrollado utilizamos la sentencia `import java.util.Scanner`. Esto nos permitió utilizar la clase `Scanner` que, evidentemente, nosotros no desarrollamos.

Sin embargo, también utilizamos otras clases que no necesitamos importar. Por ejemplo: `String`, `System`, `Integer`, `Object`. Todas estas clases, y muchas más, están ubicadas en el paquete `java.lang`. Este paquete se importa solo; no es necesario importarlo explícitamente.

14.3 Herencia y polimorfismo

Como mencionamos anteriormente la herencia permite definir clases en función de otras clases ya existentes. Diremos que la “clase derivada” o la “subclase” hereda los métodos y los atributos de la “clase base”. Esto posibilita redefinir el comportamiento de los métodos heredados y/o extender su funcionalidad.

En la sección anterior, trabajamos con la clase `Fecha`. Supongamos que no tenemos acceso al código de esta clase. Es decir, podemos utilizarla pero no la podemos modificar porque, por ejemplo, fue provista por terceras partes. Hagamos de cuenta que no la desarrollamos nosotros. De ese modo, supongamos que, aunque la clase `Fecha` nos resulta útil, funciona bien y es muy práctica, queremos modificar la forma en que una fecha se representa a sí misma cuando invocamos su método `toString`.

La solución será crear una nueva clase que herede de `Fecha` y que modifique la manera en que esta se representa como cadena. Esto lo podremos hacer sobrescribiendo el método `toString`. Llamaremos a la nueva clase `FechaDetallada` y haremos que se represente así: “25 de octubre de 2009”.

```
1 package Tema6;
2 [-] import Tema5.Fecha;
3 public class FechaDetallada extends Fecha{
4     private static String meses[] = {"Enero",
5         "Febrero",
6         "Marzo",
7         "Abril",
8         "Mayo",
9         "Junio",
10        "Julio",
11        "Agosto",
12        "Septiembre",
13        "Octubre",
14        "Noviembre",
15        "Diciembre"};
16    // constructor nulo
17    [-] public FechaDetallada() {
18    }
19    // constructor que recibe dia, mes y anio
20    [-] public FechaDetallada(int d, int m, int a) {
21        setDia(d);
22        setMes(m);
23        setAnio(a);
24    }
25    [-] public String toString() {
26        return getDia() + " de " + meses[getMes() - 1] + " de " + getAnio();
27    }
28}
```

La clase `FechaDetallada` hereda de la clase base `Fecha` y sobrescribe el método `toString` para retornar una representación con más nivel de detalle que la que provee la implementación de su padre.

Para indicar que una clase hereda de otra, se utiliza la palabra `extends`. Decimos entonces que `FechaDetallada` “extiende” a `Fecha`. Otras expresiones válidas son:

- `FechaDetallada` “hereda” de `Fecha`
- `FechaDetallada` “es una especie” de `Fecha`
- `FechaDetallada` “es hija” de `Fecha`
- `FechaDetallada` “es una subclase” de `Fecha`
- `FechaDetallada` “subclasea” a `Fecha`

La lógica de programación que utilizamos para resolver el método `toString` en la clase `FechaDetallada` es simple: definimos un `String[]` que contenga los nombres de los meses, luego retornamos una cadena de caracteres concatenando el día, seguido del nombre del mes y el año.

El array `meses` fue definido como `static`. Esto significa que es una “variable de clase”, pero este tema lo estudiaremos más adelante.

Notemos también que para acceder a los atributos `dia`, `mes` y `anio` que la clase `FechaDetallada` hereda de `Fecha` fue necesario utilizar los `getters`. Esto se debe a que los atributos son privados y, si bien existen en la clase derivada, no son accesibles sino a través de sus métodos de acceso.

Lamentablemente, los constructores no se heredan. Por este motivo, los desarrollamos explícitamente ya que, de no hacerlo así, el único constructor disponible sería el constructor nulo o por defecto.

Probemos la clase FechaDetallada con el siguiente programa donde creamos un objeto tipo FechaDetallada, le asignamos valor a sus atributos y lo imprimimos.

```
1 package Tema6;  
2  
3 public class TestFechaDetallada {  
4  
5     public static void main(String[] args) {  
6         FechaDetallada f = new FechaDetallada();  
7         f.setDia(25);  
8         f.setMes(10);  
9         f.setAnio(2009);  
10        System.out.println(f);  
11    }  
12 }
```

En el siguiente diagrama, reflejamos las relaciones que existen entre los objetos del programa anterior.

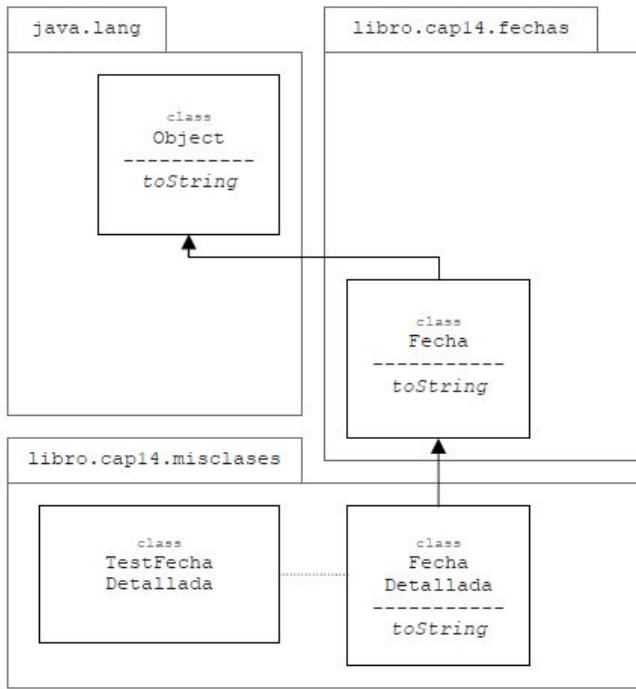


Fig. 14.2 Representación de las clases Fecha y FechaDetallada.

El diagrama representa la relación de herencia que existe entre las clases `FechaDetallada`, `Fecha` y `Object`, cada una en su propio paquete. También refleja que las clases `Fecha` y `FechaDetallada` sobreescriben el método `toString` que heredan de `Object`. Por último, muestra que la clase `TestFechaDetallada` “usa” la clase `FechaDetallada`.

Obviamente, la clase `TestFechaDetallada` también hereda de `Object`. Sin embargo, esta relación no es relevante en nuestro ejemplo, por lo que decidimos no reflejarla.

POLIMORFISMO

Los objetos nunca dejan de reconocerse como miembros de una determinada clase. Por tal motivo, independientemente de cual sea el tipo de datos de la variable que los contenga, ante la invocación de cualquiera de sus métodos siempre reaccionan como su propia clase lo define.

Para terminar de comprender la idea pensemos en una clase con un método que permita imprimir un conjunto de objetos. Este método recibirá un `Object []` (léase “object array”) para recorrerlo y mostrar en la consola la representación de cada uno de sus elementos.

```
1 package Tema7;  
2  
3 public class MuestraConjunto {  
4  
5     public static void mostrar(Object[] arr) {  
6         for (int i = 0; i < arr.length; i++) {  
7             System.out.println("arr[" + i + "] = " + arr[i]);  
8         }  
9     }  
10 }
```

Como podemos ver, dentro del método mostrar no conocemos el tipo de datos de los objetos que contiene el array. Como arr es un Object[] y todas las clases heredan de Object, entonces arr[i] puede ser un objeto de cualquier tipo.

Sin embargo, esto no nos impide imprimirlos ya que cada objeto arr[i] acudirá a su propia clase cuando System.out.println invoque a su método toString.

Veamos ahora el programa principal donde invocamos al método mostrar de la clase MuestraConjunto.

```
1 package Tema7;
2 import Tema5.Fecha;
3 import Tema6.FechaDetallada;
4
5 public class TestMuestraConjunto {
6     public static void main(String[] args) {
7         Object[] arr = {new Fecha(2, 10, 1970),
8                         new FechaDetallada(23, 12, 1948),
9                         new String("Esto es una cadena"),
10                        new Integer(34)};
11    // como el metodo es estatico lo invocamos a traves de la clase
12    MuestraConjunto.mostrar(arr);
13 }
14 }
```

Efectivamente, arr contiene objetos de tipos de datos diferentes: Fecha, FechaDetallada, String e Integer. Aun así, cuando se les invoca el método `toString` cada uno reacciona como su propia clase lo define.

La salida de este programa será la siguiente:

```
arr[0] = 2/10/1970  
arr[1] = 23 de Diciembre de 1948  
arr[2] = Esto es una cadena  
arr[3] = 34
```

Como comentamos más arriba, un objeto nunca se olvida de su clase. Así, *por polimorfismo*, se invocará al método `toString` definido en la clase a la que este pertenece. (Recordemos que `System.out.println(x)` imprime la cadena que retorna el método `toString` del objeto x).

Polimorfismo es la característica fundamental de la programación orientada a objetos. Este tema será profundizado a lo largo del presente capítulo. Por el momento basta con esta breve explicación.

Antes de terminar debemos mencionar que el método `mostrar` de la clase `MuestraConjunto` es estático. Esto lo convierte en un “método de la clase”. Si bien este tema lo analizaremos más adelante, podemos ver que en el método `main` lo invocamos directamente sobre la clase haciendo `MuestraConjunto.mostrar`. Es el mismo caso que el de `Integer.parseInt` que utilizamos para convertir cadenas de caracteres a números enteros.

Lo primero que hace el constructor de una clase derivada es invocar al constructor de su clase base. Si esto no se define, explícitamente, (programándolo) entonces se invocará al constructor nulo de la clase base y si este no existe, tendremos un error de compilación.

En la clase `FechaDetallada`, programamos los siguientes constructores:

```
public FechaDetallada()
{
}

public FechaDetallada(int d,int m,int a)
{
    setDia(d);
    setMes(m);
    setAnio(a);
}
```

Como en ninguno hacemos referencia explícita al constructor de la clase base, implícitamente, cuando invoquemos a cualquiera de estos constructores, Java invocará al constructor nulo de `Fecha`.

En este punto le recomiendo al lector comentar, momentáneamente, el código del constructor nulo de la clase `Fecha` y volver a compilar la clase `FechaDetallada`. Así, obtendrá el siguiente error de compilación:

```
Implicit super constructor Fecha() is undefined
      for default constructor. FechaDetallada.java (line 6)
```

Este mensaje indica que no está definido el constructor nulo (o sin argumentos) en la clase `Fecha`.

Constructores de SubClase

```
package libro.cap14.misclases;

public class FechaDetallada extends Fecha
{
    // :
    // definicion del array meses...
    // :

    public FechaDetallada(int dia, int mes, int anio)
    {
        // invocamos al constructor del padre
        super(dia,mes,anio);
    }

    public FechaDetallada(String s)
    {
        // invocamos al constructor del padre
        super(s);
    }

    public FechaDetallada()
    {
        // invocamos al constructor del padre
        super();
    }

    // :
    // metodo toString...
    // :
}
```

Como en la clase *Fecha* definimos tres constructores, lo razonable será programar estos mismos constructores también en la clase *FechaDetallada*.

Con esto, en la clase *FechaDetallada*, tendremos los mismos constructores que ofrece la clase *Fecha*.

En cada uno de estos constructores, invocamos al constructor del padre a través de la palabra `super`. Esta palabra representa al constructor del padre que, en este caso, es el constructor de la clase *Fecha*.

Ahora bien, ¿qué ocurrirá si en alguno de estos constructores no invocamos, explícitamente, al constructor del padre? Sucederá lo siguiente:

1. Por omisión se invocará al constructor nulo de la clase Fecha, pero como este existe no tendremos ningún error de compilación.
2. Como el constructor nulo de Fecha no hace nada, quedarán sin asignar los atributos dia, mes y anio.
3. Cuando imprimamos una fecha con System.out.println se invocará al método `toString` de FechaDetallada, que accederá al array meses [mes-1], pero como el atributo mes no tiene valor se generará un error en tiempo de ejecución, un `ArrayIndexOutOfBoundsException`.

Le sugiero al lector tratar de modificar el constructor que recibe tres enteros anulando la llamada al constructor del padre de la siguiente manera:

```
public FechaDetallada(int dia, int mes, int anio)
{
    // super(dia,mes,anio);
}
```

Luego, si hacemos:

```
public static void main(String args[])
{
    FechaDetallada f = new FechaDetallada(25,10,2009);
    System.out.println(f);
}
```

tendremos el siguiente error en tiempo de ejecución:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
at libro.cap14....FechaDetallada.toString(FechaDetallada.java:42)
at java.lang.String.valueOf(String.java:2577)
at java.io.PrintStream.print(PrintStream.java:616)
at java.io.PrintStream.println(PrintStream.java:753)
at lib....TestFechaDetallada.main(TestFechaDetallada.java:15)
```

Para exemplificar su uso, extenderemos todavía más la funcionalidad de nuestras clases desarrollando la clase FechaHora como una subclase de FechaDetallada. La clase FechaHora permitirá también representar una hora (hora, minutos y segundos).

```
package libro.cap14.misclases;

public class FechaHora extends FechaDetallada
{
    private int hora;
    private int minuto;
    private int segundo;

    public FechaHora(String sFecha, int hora, int min, int seg)
    {
        super(sFecha);
        this.hora = hora;
        this.minuto = min;
        this.segundo = seg;
    }

    public String toString()
    {
        // invocamos al metodo toString de nuestro padre
        return super.toString()+" ("+hora+":"+minuto+":"+segundo+") ";
    }

    /**
     * otros constructores...
     * setters y getters...
     */
}
```

Esta clase extiende la funcionalidad de `FechaDetallada` proveyendo, además, la capacidad de almacenar la hora.

En el método `toString` (que sobrescribimos) utilizamos la palabra `super` para invocar al método `toString` de `FechaDetallada`. A la cadena que obtenemos le concatenamos otra cadena representando la hora, y eso es lo que retornamos.

Ahora veremos un programa donde utilizamos un objeto tipo `FechaHora` y lo imprimimos.

```
package libro.cap14.misclases;

public class TestFechaHora
{
    public static void main(String[] args)
    {
        FechaHora fh = new FechaHora("25/2/2006",14,30,10);
        System.out.println(fh);
    }
}
```

La salida de este programa será:

25 de Febrero de 2006 (14:30:10)

14.3.4 La referencia this

Así como `super` hace referencia al constructor del padre, la palabra `this` hace referencia a los otros constructores dentro de una misma clase.

Para exemplificar el uso de `this` replantearemos el desarrollo de los constructores de la clase `FechaDetallada` de la siguiente manera:

```
package libro.cap14.misclases;

public class FechaDetallada extends Fecha
{
    // :
    // definicion del array meses...
    // :

    public FechaDetallada(int dia, int mes, int anio)
    {
        super(dia,mes,anio);
    }

    public FechaDetallada()
    {
        // invocamos al constructor de tres int pasando ceros
        this(0,0,0);
    }

    // :
    // contructor que recibe un String
    // metodo toString...
    // :
```

Como vemos, desde el constructor que no recibe parámetros invocamos al constructor que recibe tres enteros y le pasamos valores cero. Con esto, daremos valores iniciales (aunque absurdos) a la fecha que está siendo creada.

También, `this` puede ser usado como referencia a “nosotros mismos”. Esto ya lo utilizamos en la clase `Fecha` cuando en los `setters` hacíamos:

```
public void setDia(int dia)
{
    this.dia = dia;
}
```

Dentro de este método asignamos el valor del parámetro `dia` al atributo `dia`. Con `this.dia` nos referimos al atributo `dia` (que es un miembro o variable de instancia de la clase) y lo diferenciamos del parámetro `dia`, que simplemente es una variable automática del método.

Los conceptos de “instancia”, “atributo” y “variable de instancia” los estudiaremos en detalle más adelante, en este mismo capítulo.

14.3.5 Clases abstractas

En ocasiones reconocemos la existencia de objetos que, claramente, son elementos de una misma clase y, sin embargo, sus operaciones se realizan de manera muy diferente. El caso típico para estudiar este tema es el de las figuras geométricas.

Nadie dudaría en afirmar que “toda figura geométrica describe un área cuyo valor se puede calcular”. Sin embargo, para calcular el área de una figura geométrica será necesario conocer cuál es esa figura. Es decir: no basta con saber que se trata de una figura geométrica; necesitamos también conocer de qué figura estamos hablando.

Una clase abstracta es una clase que tiene métodos que no pueden ser desarrollados por falta de información concreta. Estos métodos se llaman “métodos abstractos” y deben desarrollarse en las subclases, cuando esta información esté disponible.

En nuestro ejemplo, `FiguraGeometrica` será una clase abstracta con un único método abstracto: el método `area`.

```
package libro.cap14.figuras;

public abstract class FiguraGeometrica
{
    // metodo abstracto
    public abstract double area();

    public String toString()
    {
        return "area = " + area();
    }
}
```

Las clases abstractas se definen como `abstract class`. El método `area` es un método abstracto, lo definimos como `abstract` y lo dejamos sin resolver finalizando su declaración con ; (punto y coma).

Las clases abstractas no pueden ser instanciadas. Es decir, no podemos crear objetos de clases declaradas como abstractas porque, por definición, hay métodos que aún no han sido implementados.

No podemos hacer esto:

```
// MAL, esto no compila
FiguraGeometrica fg = new FiguraGeometrica();
```

En general, las clases abstractas deben ser “subclaseadas”. Toda clase que herede de una clase abstracta tiene que sobrescribir los métodos abstractos de su padre o bien deberá ser declarada abstracta y, por lo tanto, no se podrá instanciar.

Ahora podemos pensar en figuras geométricas tales como el círculo, el rectángulo y el triángulo. Serán subclases de `FiguraGeometrica` en las cuales tendremos la información suficiente como para sobrescribir el método `area` ya que estaremos hablando de figuras concretas. Por ejemplo, el área de un círculo se calcula como $\pi \times \text{radio}^2$ (léase “pi por radio al cuadrado”) y el de un rectángulo será `base x altura` mientras que el de un triángulo será `base x altura / 2`. Es decir, el área se calcula en función de la figura geométrica y de sus atributos.

Figura	Atributos	Área
Círculo	<code>radio</code>	$\pi \times \text{radio}^2$
Rectángulo	<code>base, altura</code>	<code>base x altura</code>
Triángulo	<code>base, altura</code>	<code>base x altura / 2</code>

```
package libro.cap14.figuras;

public class Rectangulo extends FiguraGeometrica
{
    private double base;
    private double altura;

    public Rectangulo(double b, double h)
    {
        base = b;
        altura = h;
    }

    public double area()
    {
        return base * altura;
    }

    /**
     * setters y getters
     */
}
```

Como vemos, en la clase `Rectangulo`, definimos los atributos que caracterizan a dicha figura y sobrescribimos, adecuadamente, el método `area` retornando el producto de sus atributos `base` y `altura`. Lo mismo faremos con las clases `Circulo` y `Triangulo`.

```
package libro.cap14.figuras;

public class Circulo extends FiguraGeometrica
{
    private int radio;

    public Circulo(int r)
    {
        radio = r;
    }

    public double area()
    {
        // retornamos "PI por radio al cuadrado"
        return Math.PI*Math.pow(radio,2);
    }
}
```

El número π está definido como una constante estática en la clase `Math`. Para acceder a su valor, lo hacemos a través de `Math.PI`. Para calcular la potencia $radio^2$ utilizamos el método `pow`, también estático y definido en la misma clase `Math`.

El concepto de “estático” lo estudiaremos más adelante pero, como comentamos anteriormente, sabemos que los métodos estáticos pueden invocarse directamente sobre la clase sin necesidad de instanciarla; es el caso de `Math.pow`, `System.out.println`, `Integer.parseInt`, etcétera.

```
package libro.cap14.figuras;

public class Triangulo extends FiguraGeometrica
{
    private double base;
    private double altura;

    public Triangulo(int b, int h)
    {
        base = b;
        altura = h;
    }

    public double area()
    {
        return base*altura/2;
    }
}
```

Pensemos ahora en un programa que utilice estas clases:

```
package libro.cap14.figuras;

public class TestFiguras
{
    public static void main(String[] args)
    {
        Circulo c = new Circulo(4);
        Rectangulo r = new Rectangulo(10,5);
        Triangulo t = new Triangulo(3,6);

        System.out.println(c);
        System.out.println(r);
        System.out.println(t);
    }
}
```

La salida será:

```
area = 50.26548245743669  
area = 50.0  
area = 9.0
```

Este resultado demuestra lo siguiente:

1. Las clases Circulo, Rectangulo y Triangulo heredan de FiguraGeometrica el método `toString`. Recordemos que dentro de este método invocábamos al método abstracto `area`.
2. Cuando en `toString` invocamos al método `area` en realidad estamos invocando a la implementación concreta del método `area` del objeto sobre el cual se invocó a `toString`. Por este motivo, resulta que el cálculo del área es correcto para las tres figuras que utilizamos en el `main`.

14.3.6 Constructores de clases abstractas

Que una clase abstracta no pueda ser instanciada no significa que no pueda tener constructores.

¿Qué sentido tiene declarar un constructor en una clase que no podremos instanciar? El sentido es el de “obligar” a las subclases a *settear* los valores de los atributos de la clase base.

Agregaremos el atributo nombre en clase FiguraGeometrica y también un constructor que permita especificar su valor. Así, cada figura podrá guardar su nombre y proveer información más específica cuando invoquemos a su método `toString`.

```
package libro.cap14.figuras;

public abstract class FiguraGeometrica
{
    private String nombre;

    // metodo abstracto
    public abstract double area();

    // agregamos un constructor
    public FiguraGeometrica(String nom)
    {
        nombre = nom;
    }

    // ahora en el tostring mostramos tambien el nombre
    public String toString()
    {
        return nombre + " (area = "+ area()+" ) ";
    }

    public String getNombre()
    {
        return nombre;
    }

    public void setNombre(String nombre)
    {
        this.nombre = nombre;
    }
}
```

Ahora tenemos que modificar las subclases e invocar explícitamente al constructor definido en la clase base.

```
package libro.cap14.figuras;

public class Rectangulo extends FiguraGeometrica
{
    private double base;
    private double altura;

    public Rectangulo(double b, double h)
    {
        super("Rectangulo"); // constructor del padre
        base = b;
        altura = h;
    }

    /**
     * metodo area
     * setters y getters...
     */
}
```

Lo primero que debemos hacer en el constructor de la clase derivada es invocar al constructor del padre. Como el constructor de `FiguraGeometrica` espera recibir el nombre de la figura, le pasamos como argumento “nuestro” propio nombre que, en este caso, es “Rectangulo”.

Algún lector con experiencia podrá pensar que estamos “hardcodeando” el nombre “Rectangulo”, pero no es así. Se trata del nombre de la figura y este nunca podrá cambiar arbitrariamente.

Apliquemos los cambios en Circulo y Triangulo.

```
package libro.cap14.figuras;

public class Circulo extends FiguraGeometrica
{
    private int radio;

    public Circulo(int r)
    {
        super("Circulo");
        radio = r;
    }
}
```

```
package libro.cap14.figuras;

public class Triangulo extends FiguraGeometrica
{
    private double base;
    private double altura;

    public Triangulo(int b, int h)
    {
        super("Triangulo");
        base = b;
        altura = h;
    }
}
```

Ahora al ejecutar el programa principal obtendremos la siguiente salida:

Circulo (area = 50.26548245743669)

Rectangulo (area = 50.0)

Triangulo (area = 9.0)

Por último, informalmente, dijimos que los métodos estáticos pueden invocarse directamente sobre las clases sin tener que instanciarlas. Podríamos definir un método estático en la clase `FiguraGeometrica` que permita calcular el área promedio de un conjunto de figuras.

```
package libro.cap14.figuras;

public abstract class FiguraGeometrica
{
    private String nombre;

    // metodo abstracto
    public abstract double area();

    public static double areaPromedio(FiguraGeometrica arr[])
    {
        int sum=0;
        for( int i=0; i<arr.length; i++ )
        {
            sum += arr[i].area();
        }
        return sum/arr.length;
    }

    // :
    // constructor
    // setters y getters...
    // :
}
```

Es fundamental notar la importancia y el poder de abstracción que logramos al combinar métodos abstractos y polimorfismo.

En el método `areaPromedio`, recorremos el conjunto de figuras y sobre cada elemento `arr[i]` invocamos al método `area` sin preocuparnos por conocer cuál es la figura concreta.

El simple hecho de no poder instanciar a `FiguraGeometrica` nos garantiza que `arr[i]` únicamente podrá tener objetos de alguna de las implementaciones concretas de esta clase. Luego, por polimorfismo, `arr[i].area()` llamará al método `area` de la clase concreta, nunca al de la clase abstracta porque esta no puede ser instanciada.

Veamos un programa donde calculamos el área promedio de las figuras geométricas contenidas en un array.

```
package libro.cap14.figuras;

public class TestAreaPromedio
{
    public static void main(String[] args)
    {
        FiguraGeometrica arr[] = { new Circulo(23),
                                    , new Rectangulo(12, 4)
                                    , new Triangulo(2, 5) };

        double prom = FiguraGeometrica.areaPromedio(arr);

        System.out.println("Promedio = " + prom);
    }
}
```