Informática

Todo lo básico que debería saber sobre

Programación orientada a objetos en Java®

Luis Fernando García Llinás

Ediciones Uninorte

Todo lo básico que debería saber sobre

Programación orientada a objetos en Java®

Todo lo básico que debería saber sobre

Programación orientada a objetos en Java®

LUIS FERNANDO GARCÍA LLINÁS

ediciones de la U

EDICIONES UNINORTE

2010

García Llinás, Luis Fernando.

Todo lo que debería saber sobre programación orientada a objetos en Java / Luis Fernando García Llinás. – Barranquilla : Ediciones Uninorte ; Grupo Editorial Ibánez, 2010.

258 p. ; 16 x 24 cm.
ISBN 978-958-741-062-4

1. Java (Lenguaje de programación de computadores). 2. Programación orientada a objetos (Computadores). I. Tít.

(005.117 G216 Ed. 22) (CO-BrUNB)

UNIVERSIDAD DEL NORTE
www.uninorte.edu.co
Km 5 vía a Puerto Colombia, A.A. 1569,
Barranquilla (Colombia)

ediciones de la U
http://edicionesdelau.com/
Calle 24A n.° 43-22
Bogotá (Colombia)

© Ediciones Uninorte, 2010
© Ediciones de la U, 2010
© Luis Fernando García Llinás, 2010

Coordinación editorial
Zoila Sotomayor O.

Diseño y diagramación
Nilson Ordoñez

Diseño de portada
Álvaro Bernal

Corrección de textos
Mercedes Castilla

Impreso y hecho en Colombia
X-press Proceso Gráfico
Bogotá
Printed and made in Colombia

CONTENIDO

1. TÓPICOS BÁSICOS 1

1.1. Sobre el paradigma de programación estructurada o procedimental 1

1.2. Objeto 5

1.3. Clase 6

1.4. Atributo 7

1.5. Instanciación 12

1.6. Método 22

Método constructor, 29

1.7. Encapsulamiento 39

1.8. Atributos finales 52

1.9. Atributos y métodos estáticos 54

1.10. Herencia 59

1.11. Métodos y clases abstractas 96

1.12. Casting 105

1.13. Polimorfismo 107

1.14. Métodos y clases finales 114

1.15. Herencia simple y múltiple 123

1.16. Interfaces 124

2. TÓPICOS AVANZADOS 135

2.1. Colecciones 136

Listas, 144. Conjuntos, 154. Mapas, 157. Genéricos, 159.

2.2. Manejo de excepciones 171

Generación y lanzamiento de excepciones, 174. Captura de excepciones, 177.

Definición de excepciones personales, 179.

3. CREACIÓN DE INTERFACES GRÁFICAS DE USUARIO 199

3.1. Componentes gráficos 201

3.2. Layouts 208

FlowLayout, 209. GridLayout, 211. BorderLayout, 213.

3.3. Bordes 221

3.4. Manejo de eventos 226

PREFACIO

El aprendizaje del paradigma de la programación orientada a objetos presenta considerables problemas a quienes están acostumbrados a trabajar bajo el paradigma estructurado, sobre todo cuando estos tienen varios años de experiencia. Quizás el cambio de mentalidad que se requiere o la gran cantidad de nociones que surgen acentúan esta situación.

Inicialmente la programación orientada a objetos puede llegar a verse como un paradigma complejo o complicado, aun cuando en realidad lo que busca es organizar de mejor manera el código de una aplicación desarrollada con el enfoque procedimental. Esto con el fin de facilitar su entendimiento y, por ende, si es necesario, su modificación.

La metodología que se emplea en este libro para introducir los tópicos consiste en plantear un problema sumamente sencillo, luego entrar a resolverlo empleando la programación estructurada y después de forma progresiva ir desarrollando la aplicación empleando la programación orientada a objetos. Durante este proceso se van introduciendo cada una de las nociones más importantes para adicionarlas al desarrollo de la solución (en algunos casos se presentan más ejemplos para lograr un mejor entendimiento de cada uno de los tópicos). Es realmente importante analizar cómo se va configurando la solución orientada a objetos y compararla con la solución estructurada.

Objetivos

- **Ser conciso.** Uno de los principales objetivos de este libro era escribirlo de manera sumamente concisa, es decir que no excediera las 250 páginas. Por eso únicamente se tocan los temas básicos de la codificación de aplicaciones orientadas a objetos empleando a Java como lenguaje de programación. Por consiguiente, se excluyen temas como el trabajo con redes, la interacción con bases de datos, el desarrollo de aplicaciones para la web, etc.

- **Servir como punto de partida.** En ningún momento este libro pretende reemplazar textos más robustos y completos que tratan estos y muchos otros tópicos. Más bien se presenta como un punto de partida para los programadores y un eficaz complemento de otros textos, pues en la medida en que se comprenda la totalidad de esta obra será mucho más sencillo entender cualquier otra que trate los mismos temas.

¿A quién va dirigido este libro?

Este libro fue ideado como texto guía para un curso de programación orientada a objetos donde llegan estudiantes con nociones básicas de programación y conocimiento de la teoría de algunas estructuras de datos. Por consiguiente, la obra se dirige especialmente a programadores que emplean el paradigma procedimental como principal aproximación a la codificación de soluciones (sobre todo empleando C como lenguaje de programación), y que desean aprender las nociones del desarrollo de aplicaciones orientadas a objetos empleando Java.

¿Qué se requiere antes de leerlo?

- Descargar e instalar la máquina virtual de Java que puede ser obtenida de forma gratuita de la web oficial de Sun (http://java.sun.com/javase/downloads/index.jsp)
- Leer del Tutorial oficial de Java (http://java.sun.com/tutorial) los apartados que llevan por títulos "About the Java Technology" (http://java.sun.com/docs/books/tutorial/getStarted/intro/definition.htm) y "Hello World! For Microsoft Windows" (http://java.sun.com/docs/books/tutorial/getStarted/cupojava/win32.html).
- Descargar e instalar un entorno de desarrollo (IDE) para Java. Esto facilitará en gran medida la codificación de los ejemplos. Existen muchos entornos gratuitos entre los que sobresalen: Netbeans (http://www.netbeans.org/), JDeveloper (http://www.oracle.com/technology/products/jdev/index.html), y Eclipse (http://www.eclipse.org/).

Organización

El libro está organizado en tres capítulos.

- En el primer capítulo se presentan los tópicos básicos de la programación orientada a objetos. También se analizan las nociones necesarias para la creación de aplicaciones orientadas a objetos sumamente básicas.
- En el segundo capítulo se presentan unos tópicos un poco más avanzados de la teoría de objetos. Aquí las nociones analizadas posibilitan la creación de aplicaciones mas robustas y completas orientadas a objetos.
- En el tercer capítulo se presentan las teorías básicas para diseñar y codificar interfaces gráficas de usuario sencillas y moderadamente atractivas.

Adicionalmente, en todos estos capítulos se incluye un conjunto de apartados donde se presentan explicaciones más amplias, aclaraciones pertinentes o simplemente ejemplos más completos de los tópicos que se abordan.

Retroalimentación

Cualquier comentario, observación, sugerencia o recomendación que se desee efectuar sobre la presente obra será bien recibida en las siguientes direcciones de correo electrónico: *garcialf@uninorte.edu.co* o *luisgarciallinas@gmail.com*

1. TÓPICOS BÁSICOS

1.1. Sobre el paradigma de programación estructurada o procedimental

Dentro del mundo del desarrollo de aplicaciones bajo el paradigma estructurado (también conocido como programación procedimental o tradicional) la idea general consiste en especificar el conjunto de instrucciones que brindan solución a un problema específico; este conjunto de instrucciones se definen al interior de las marcas de inicio y fin del bloque de codificación principal del algoritmo. Durante el aprendizaje de este paradigma de programación las recomendaciones iniciales son: entender el problema, definir las variables globales que harán parte de la solución, identificar aquellas secciones de código que conviene incluirlas dentro de funciones o subrutinas para futura reutilización, definir datos de entrada y de salida, entre otras.

Suponga que ha sido encargado del desarrollo de la aplicación de software a una compañía que la desea para efectuar el cálculo mensual de su nómina. Esta compañía contrata empleados a quienes les paga dependiendo del número de horas trabajadas y del valor por hora convenido previamente con cada uno. Como información básica de cada empleado debe registrarse el número de la cédula, su nombre y su apellido.

Aclaraciones:

- Tanto el valor del número de horas trabajadas por cada empleado como el valor de su sueldo por hora puede variar de un empleado a otro.
- Se supondrá que la aplicación solo se requiere para calcular el valor de la nómina de un único mes.

- Para efectos de mantener la simplicidad del ejemplo no se contemplan acciones para manejar la persistencia [1] de los datos.

A continuación una posible solución al problema planteado empleando el paradigma estructurado.

```
Inicio

   Entero: numeroEmpleados, i
   Caracteres: cedulas[50], apellidos[50], nombres[50]
   Real: horasTrabajadas[50], sueldoXHora[50]
   Caracteres: cedula, apellido, nombre
   Real: horas, sueldo
   Real: total <- 0

   Esc 'Digite número de empleados: '
   Lea numeroEmpleados

   Para i=0,numeroEmpleados-1,1
   Esc 'Digite la cédula del empleado: '
   Lea cedula
   Esc 'Digite el apellido del empleado: '
   Lea apellido
   Esc 'Digite el nombre del empleado: '
   Lea nombre
   Esc 'Digite número de horas trabajadas del empleado: '
   Lea horas
   Esc 'Digite valor de sueldo por hora del empleado:'
   Lea sueldo
   cedulas[i] <- cedula
   apellidos[i] <- apellido
   nombres[i] <- nombre
   horasTrabajadas[i] <- horas
   sueldoXHora[i] <- sueldo
   Fin-Para

   Para i=0, numeroEmpleados-1, 1
   total <- total + horasTrabajadas[i] * sueldoXHora[i]
   Fin-Para

   Esc 'La nómina total es: ' + total

Fin
```

Entender el anterior seudocódigo no debe presentar mayores problemas para cualquier programador. Sin embargo, es conveniente realizar las siguientes aclaraciones y comentarios:

- Aunque se puede condensar el código incluyendo las instrucciones del segundo 'Para' dentro del primero, de manera intencional se ha dejado así intencionalmente para delimitar funcionalmente cada bloque de código.

- En el algoritmo se captura información, como la cédula, el nombre y el apellido,

1 Acciones para preservar de forma permanente y para recuperar los datos, como, por ejemplo, guardarlos en un archivo en disco.

que no se utiliza; sin embargo esta información se mantiene porque posteriormente puede ser útil para ampliar la funcionalidad de la aplicación.

- Como la intención es que sea un ejemplo didáctico, inicialmente el algoritmo no contempla validaciones como impedir el doble ingreso de un mismo número de cédula.

CODIFICACIÓN EN JAVA DEL ALGORITMO PARA EL CÁLCULO DE LA NÓMINA EMPLEANDO PROGRAMACIÓN ESTRUCTURADA

Se presenta a continuación la codificación del ejemplo anterior aunque apenas se estén dando los primeros pasos en el aprendizaje del lenguaje de programación Java. Esto porque es más sencillo aprender a través de ejemplos y del establecimiento de analogías, asociaciones y comparaciones.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
class Nomina {
  public static void main(String[] args) throws Exception {

    int numeroEmpleados;
    String[] cedulas = new String[50];
    String[] apellidos = new String[50];
    String[] nombres = new String[50];
    double[] horasTrabajadas = new double[50];
    double[] sueldoXHora = new double[50];
    String cedula, apellido, nombre;
    double horas, sueldo;
    double total = 0;

    BufferedReader br = new BufferedReader(
      new InputStreamReader( System.in ) );

    System.out.print("Digite numero de empleados: ");
    numeroEmpleados = Integer.valueOf(br.readLine()).intValue();
    for(int i=0;i<numeroEmpleados;i++){
      System.out.print("\nDigite la cedula del empleado: " );
      cedula = br.readLine();
      System.out.print("Digite el apellido del empleado: " );
      apellido = br.readLine();
      System.out.print("Digite el nombre del empleado: " );
      nombre = br.      ();
      System.out.print("Digite num de horas trabajadas del empleado: ");
      horas = Double.valueOf(br.readLine()).doubleValue();
      System.out.print("Digite sueldo por hora del empleado: " );
```

```
      sueldo = Double.valueOf(br.readLine()).doubleValue();
      cedulas[i] = cedula;
      apellidos[i] = apellido;
      nombres[i] = nombre;
      horasTrabajadas[i] = horas;
      sueldoXHora[i] = sueldo;
    }

    for (int i=0;i<numeroEmpleados;i++)
      total = total + sueldoXHora[i]*horasTrabajadas[i];

    System.out.println("\nLa nómina total es: "+ total);
  }
}
```

Del código anterior vale la pena resaltar algunos puntos:

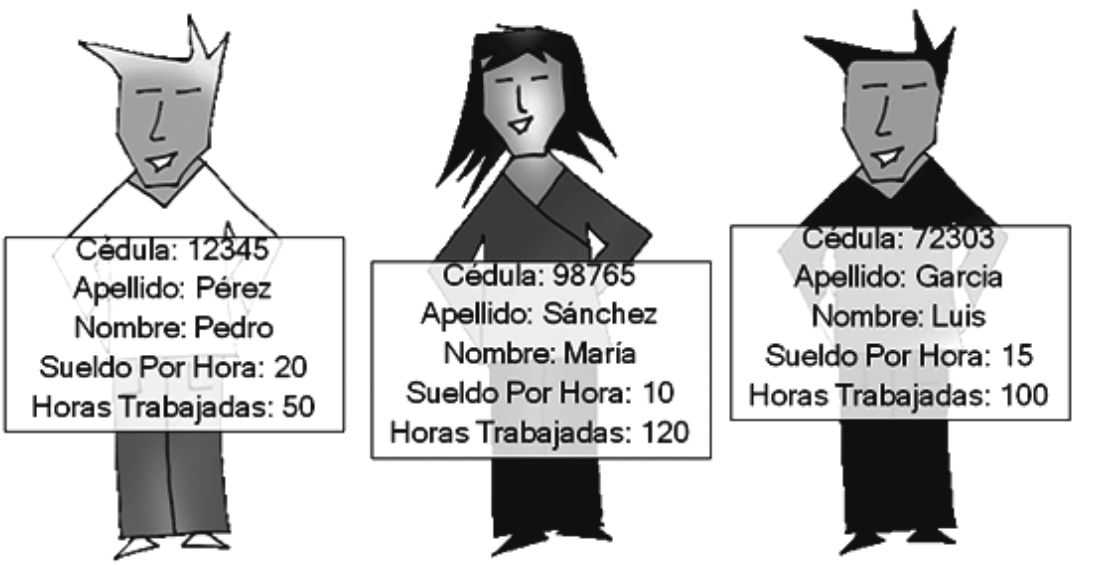
- Las dos primeras líneas donde aparecen instrucciones `import` sirven para importar componentes necesarios para la lectura de datos por pantalla.
- La instrucción `public static void main(String args[])` corresponde a la definición del método principal.
- El tipo de dato en Java `double` se corresponde con el tipo `Real` en el seudocódigo; el tipo de dato en Java `String` se corresponde con el tipo `Caracteres` en seudocódigo; y el tipo de dato `int` en Java se corresponde con el tipo de dato `Entero` en seudocódigo.
- La lectura de información por pantalla requiere la configuración de un componente de tipo `BufferedReader`.
- La instrucción que permite leer información de la pantalla (lo que en seudocódigo se escribe como la instrucción `Lea`, o en C la instrucción `cin`) corresponde a la función `readLine()`. Esta función siempre devuelve lo que digita el usuario como una cadena de caracteres; si se desea trabajar con un tipo de dato distinto, deben realizarse conversiones adicionales.
- La instrucción que permite escribir información en pantalla (lo que en seudocódigo se escribe como la instrucción `Esc`, o en C la instrucción `cout`) corresponde a la función `System.out.println()`. El carácter de escape '\n' corresponde al retorno de carro y nueva línea.
- El segundo bloque 'for' no define llave de apertura({) y de cierre(}), debido a que solo posee una línea de código.
- La concatenación de un número a una cadena de caracteres se realiza con el simple uso del operador '+'.

- Es conveniente identificar cómo es el manejo de los arrays en Java y cómo se realiza la conversión entre tipos de datos.
- En Java, los índices de los arrays comienzan en el valor 0; es por ello que los contadores de los bloques de código 'Para' se inicializan en dicho valor, y llegan hasta el valor menos 1.

1.2. Objeto

Dentro del paradigma de desarrollo de aplicaciones orientadas a objetos cambia el enfoque de la solución. Lo importante para el paradigma procedimental o estructurado es el bloque de código principal (instrucciones dentro de la marca de inicio y fin). Para el paradigma orientado a objetos lo principal es entender y modelar el problema, y luego sí definir el bloque de código principal que empleando el modelo definido brinde solución al problema específico.

La programación orientada a objetos requiere inicialmente identificar y modelar cada uno de los entes que hace parte del problema. Facilita la comprensión del tema hacerse una imagen mental de una posible situación; por ejemplo, para el caso del cálculo de la nómina suponga que la empresa cuenta únicamente con tres empleados cuya información se muestra en el siguiente gráfico.



Para un programador con poca experiencia en la orientación a objetos es moderadamente sencillo identificar que para el caso anterior los entes que toman parte del problema corresponden a los empleados de la empresa. En caso de que se necesite incluir un nuevo

empleado en la empresa, aparecerá otro objeto que seguramente poseerá las características descriptivas de cédula, apellido, nombre, sueldo por hora y horas trabajadas, y demás valores para dichas características.

Es conveniente presentar una definición para la noción de objeto.

Definición de objeto

Un objeto es un concepto, abstracción o cosa con límites bien definidos y con significado dentro del problema.

Como se puede apreciar la definición de objeto es realmente amplia pues en realidad cualquier cosa puede ser un objeto. De allí que puedan existir objetos que representen cosas concretas (como automóviles, casas, libros, etc.) y objetos que representan cosas abstractas (como pensamientos, ideas, etc.).

1.3. Clase

Para traducirlo a algún lenguaje de programación se requiere una estructura de datos a fin de almacenar la información de cada uno de los objetos del problema. Sin embargo, no tiene sentido definir una estructura de datos independiente para cada uno de los posibles objetos de tipo empleado (como, por ejemplo, EstructuraTrabajor1, EstructuraTrabajador2, etc.), es más conveniente definir una única estructura de datos que pueda servir para almacenar la información de cualquier objeto del mismo tipo. La siguiente tabla muestra la información que debe permitir registrar esta estructura genérica y su correspondiente tipo de dato.

Campo	Tipo dato	Descripción
cédula	Caracteres	En este campo se registra el número de la cédula de ciudadanía de un empleado.
apellido	Caracteres	En este campo se registra la cadena de caracteres que corresponde al apellido de un empleado.
nombre	Caracteres	En este campo se registra la cadena de caracteres que corresponde al nombre de un empleado.
horasTrabajadas	Real	En este campo se registra el número de horas trabajadas por un empleado; por ejemplo, el valor de 1.5 indica que el empleado ha trabajado una hora y media (90 minutos).
sueldoXHora	Real	En este campo se registra el valor que debe ser pagado a un empleado por cada hora de trabajo.

Definición de clase

Una clase describe a un conjunto de objetos que comparten una estructura y un comportamiento común.

Una clase es un molde o plantilla que indica cómo será un objeto de dicha clase. En el área de la construcción, una clase podría ser el plano de una casa que indica la estructura que debe tener cada una de las casas, y los objetos son la materialización de las casas construidas a partir de dicho plano. Es por ello que se define a un objeto como una instancia de una clase.

Para definir una clase en Java se utiliza la palabra reservada '`class`'. La sintaxis de dicha instrucción requiere además de la especificación de un nombre para la clase. La comunidad de programadores de Java maneja una convención de nombramiento para las clases (más que una regla es una simple sugerencia); dicha convención establece que el nombre de la clase debe escribirse todo en minúsculas a excepción de la primera letra del nombre de la clase. Si para establecer el nombre de la clase se requieren varias palabras se deben unir las letras de todas las palabras y la primera letra de cada palabra debe estar en mayúscula (por ejemplo, `EmpleadoDeEmpresa`). A continuación se presenta el código que permite definir en Java la clase para describir a objetos tipo empleado.

```
class Empleado{
   String cedula;
   String apellido;
   String nombre;
   double horasTrabajadas;
   double sueldoXHora;
}
```

Cada uno de elementos incluidos dentro de una clase recibe el nombre de atributos.

1.4. Atributo

Definición de atributo

Un atributo es una propiedad que ayuda a describir un objeto.

Es conveniente tener en cuenta lo siguiente:

- Hasta el momento y con las definiciones dadas, dos objetos distintos, pero de la misma clase tienen la misma estructura; es decir, comparten los mismos atributos, pero los valores para cada uno de los atributos son independientes.

- El orden de definición de los atributos es irrelevante.
- El concepto de atributo está estrechamente ligado al concepto de variable; en realidad todo atributo es un tipo de variable, sin embargo, no toda variable que pueda definirse en un programa en Java es un atributo.

Note que en Java para definir una variable se requiere, además de un nombre[2], su tipo de dato. La comunidad de programadores de Java maneja una convención de nombramiento para los atributos (más que una regla es una sugerencia), con la cual establece que el nombre del atributo debe escribirse en letras minúsculas. Si el nombre del atributo está compuesto por varias palabras, se unen las letras de todas las palabras y se colocan en mayúsculas las primeras letras de cada palabra a excepción de la primera letra del nombre del atributo (por ejemplo, horasTrabajadas).

Básicamente los tipos de datos para los atributos (y variables) pueden ser de dos clases:

- **Tipo de dato primitivo[3].** Corresponde a un tipo de dato predefinido por el lenguaje. Cuando se define un atributo (o variable) de este tipo, entonces hay que separar un espacio en memoria para guardar su valor. Los ocho tipos de datos primitivos son: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` y `char`. Se hace relativamente sencillo inferir el propósito de cada uno de ellos. Es conveniente resaltar que el tipo de dato `char` corresponde a un solo carácter, lo que significa que no existe un tipo de dato primitivo en Java para una cadena de caracteres. La buena noticia es que existe `String` que no corresponde a un tipo de dato primitivo, sino a un tipo de dato de referencia.
- **Tipo de dato de referencia.** Corresponde a un objeto de una clase (no obligatoriamente distinta a la definida). En este punto es donde radica gran parte de la importancia, flexibilidad y reutilización del paradigma de orientación a objetos; porque cuando se definen atributos cuyo tipo sea una clase, se amplía el espectro de posibilidades. Cuando se define un atributo de este tipo no se separa espacio en memoria para un nuevo objeto, sino que se define una referencia que apuntará a un espacio de memoria con la estructura definida en la clase.

2 Detalles sobre la convención de nombramiento de variables pueden ser consultados en la siguiente referencia: http://java.sun.com/docs/books/tutorial/java/nutsandbolts/variables.html

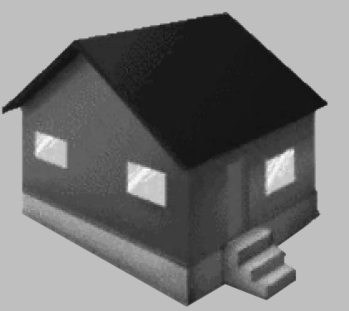
3 Mayor información sobre cada uno de estos tipos de datos puede ser consultada en la siguiente referencia: http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html

SOBRE LA COMPOSICIÓN DE OBJETOS

Uno de los pilares fundamentales de la programación orientada a objetos corresponde a la reutilización. Aquí la idea fundamental no es reinventar la rueda cada vez que se necesite, sino poder reutilizar la que ya esta inventada.

La siguiente podría ser una buena definición para la clase Casa.

```
class Casa{
  String colorTecho;
  String tipoTecho;
  double largoTecho;
  double anchoTecho;
  double altoTecho;
  String colorParedes;
  String tipoParedes;
  int numeroDeVentanas;
}
```



En la anterior definición el atributo `tipoTecho` hace referencia al material con que está construido el techo de la casa, que puede tomar los valores de: paja, cinc, teja, etc.; similarmente, el atributo `tipoParedes` hace referencia al material que compone las paredes de la casa, por ejemplo: ladrillo, madera, barro, etc. El resto de la anterior definición de la clase `Casa` no es demasiado compleja y la mayoría de los atributos se entienden fácilmente.

Una definición alternativa para modelar una casa podría ser la siguiente:

```
class Techo{
   String color;
   String tipo;
   int largo;
   int ancho;
   int alto;
}
```

```
class Pared{
   String tipo;
   String color;
}

class Casa{
   Techo elTecho;
   Pared lasParedes;
   int numeroDeVentanas;
}
```

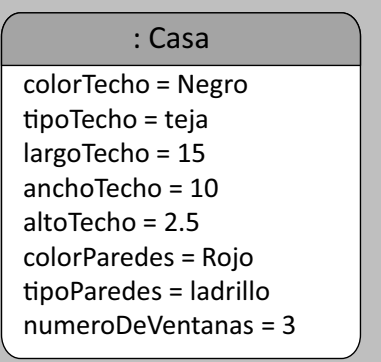
Note que en este caso para lograr la definición de la clase `Casa` ha sido necesaria la previa definición de las clases `Techo` y `Pared`. Adicionalmente cabe resaltar que en la clase `Casa` existe un atributo de tipo de referencia de clase `Techo` cuyo nombre es `elTecho`, y, además, existe un atributo de tipo de referencia de la clase `Pared` de nombre `lasParedes`.

¿Cuál de las dos definiciones es la mejor? Depende del problema. La primera definición de la clase `Casa` es simple y sencilla de trabajar, pero no muy reutilizable. La segunda es un poco más compleja, aunque más reutilizable (imagine que se necesita la clase `Edificio`, para la cual se podrían reutilizar algunas clases como `Pared` y `Techo`).

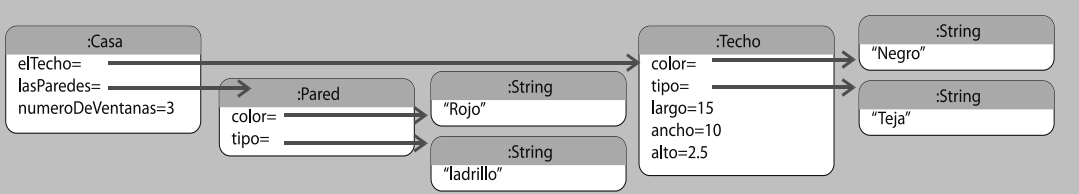
Sobre la definición y manipulación de los tipos de datos primitivos y los de referencia

Es realmente importante tener claridad sobre los tipos de datos de los atributos pues Java trata de forma muy distinta a cada uno de ellos.

Imagínese un objeto de la clase `Casa` donde solo se utilicen datos primitivos (la primera definición que aparece en el apartado **sobre la composición de objetos**). Java reservaría espacio en memoria de la siguiente manera (el texto en la parte superior del rectángulo es simplemente para denotar la clase del objeto):

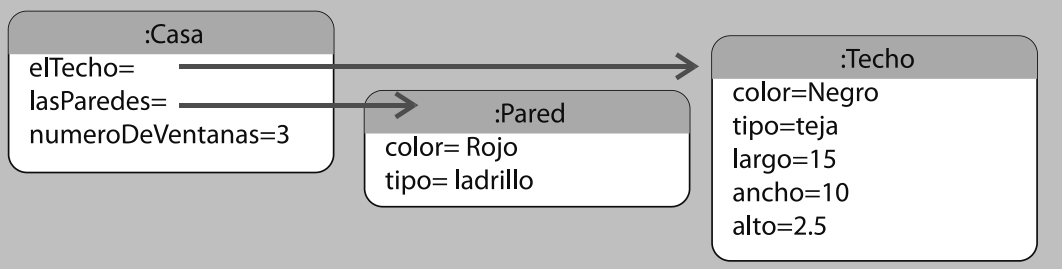


Para la segunda definición de la clase `Casa` (en el ejemplo donde se usa composición en el apartado sobre la composición de objetos), el mismo ejemplo anterior tendría la siguiente representación:

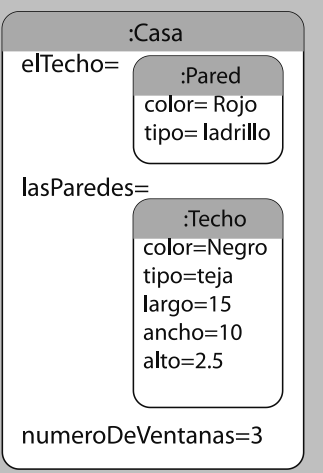


Como se mencionó previamente para los atributos cuyo tipo de dato son de referencia no se separan espacios en memoria, es decir, para atributos de tipo `Pared`, `Techo` y `String`. En vez de eso, se definen apuntadores a referencias de objetos de dichos tipos.

Simplemente por facilidad y para no complejizar innecesariamente las gráficas, a lo largo de este libro se obviará la definición de objetos para los tipos de datos de referencia `String`, es decir, que de ahora en adelante se presentarán los diagramas de la siguiente forma.



Simplemente para hacer claridad en este último punto, la estructura en memoria para el objeto de la clase `Casa` nunca es como se muestra en el siguiente gráfico.



La definición de clase hace referencia a dos puntos: primero, la estructura y segundo, el comportamiento. La definición de la estructura de los objetos de una clase se consigue a través del establecimiento de sus atributos.

1.5. Instanciación

Es conveniente introducir en estos momentos la instrucción que posibilita la instanciación de una clase, de tal manera que se pueda crear un nuevo objeto para el almacenamiento de valores en sus atributos. En Java, para crear un nuevo objeto es necesario utilizar el comando new, seguido del nombre de la clase que se desea instanciar[4]; entonces para crear un nuevo empleado se utiliza la siguiente instrucción

```
new Empleado();
```

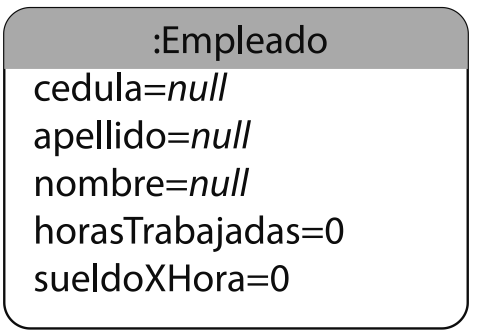
4 Cuando se llegue al concepto de método constructor se redefine esta afirmación. Por ahora, el objetivo es mantener la explicación lo más simple posible.

Con esta instrucción Java separa un espacio en memoria con la estructura definida para la clase `Empleado` (es decir, debe haber un espacio `cedula` para almacenar la cédula del empleado, un espacio `apellido` para almacenar el apellido del empleado, etc.).

La pregunta que debe surgir en estos momentos es, si con la instrucción `new Empleado()` se separa espacio en memoria para crear un objeto de la clase `Empleado`, ¿con qué valores inician los atributos de este nuevo objeto? La respuesta, Java inicializa los valores de los atributos con valores por defecto de la siguiente manera:

- Para atributos cuyo tipo sea de referencia, el valor por defecto es `null`; es decir, la referencia no apunta a ningún objeto.
- Para atributos cuyo tipo sea un tipo de dato primitivo, el valor depende del tipo de dato: si es un valor numérico (`byte`, `short`, `int`, `long`, `float` y `double`), su valor inicial es 0; para tipo `boolean`, el valor inicial es falso (`false`), y, para tipo `char`, el valor inicial es '\u0000', que corresponde al primer carácter que puede ser representado.

Es decir, después de ejecutar la instrucción `new Empleado();` se crea un objeto de la siguiente manera:



Luego de haber instanciado la clase, puede surgir la siguiente pregunta: ¿cómo se le asignan valores a los atributos del objeto (para poder cambiar los valores por defecto con los que se inicializa)? Como el objeto se crea en un espacio de la memoria, es necesario obtener la referencia a esa posición en memoria y para poder hacerlo se define una variable, a la que se le asigna dicha posición en memoria. Se requiere modificar la anterior instrucción por las siguientes:

```
Empleado elEmpleado;
elEmpleado = new Empleado();
```

O se puede simplificar en una sola instrucción:

```
Empleado elEmpleado = new Empleado();
```

El entendimiento de esta definición suele causar problemas a algunos programadores. La siguiente idea puede ayudar a clarificar la situación; al trabajar en cualquier lenguaje de programación a la hora de definir una variable se emplea una instrucción semejante a `int a = 50`, es decir, primero se define el tipo de dato, luego el nombre de la variable y posteriormente su valor. Si se analiza con detenimiento esta última instrucción y se compara con la de creación de una instancia de `Empleado`, se obtiene lo siguiente:

Aquí se mencionan los siguientes puntos:

- La primera parte de la definición indica el tipo de dato de la variable (que puede ser un tipo de dato primitivo o de referencia). Esta parte cobra gran importancia al trabajar el tópico de herencia.
- La segunda parte de la definición corresponde al nombre que identificará a la variable. Para la primera definición, el nombre de la variable es `elEmpleado;` para la segunda, simplemente el carácter `a`.
- La tercera parte de la definición especifica el valor que le será asignado a la variable.
- Es importante considerar que aunque se ha intentado realizar una comparación de ambas instrucciones, por definir variables de tipo de datos distintos, la manipulación interna que hace el lenguaje de programación de estas variables es también distinto. Recuerde que en la primera instrucción la variable `elEmpleado` corresponde a una referencia de una posición en memoria, mientras que la segunda variable efectivamente guarda el valor que se le asigna.

Una vez se tiene una referencia a un objeto (a través de la definición de una variable), es posible consultar o modificar el valor de cualquiera de sus atributos. Para hacer cualquiera de estas dos operaciones basta con generar una instrucción donde se especifique la referencia al objeto y al atributo del objeto que se desea manipular; la delimitación entre ambos se logra empleando el carácter punto ('.'); por ejemplo, suponiendo que la referencia a un objeto de la clase Empleado se llame `elEmpleado`,

- Se puede cambiar el valor del atributo `horasTrabajadas` empleando la siguiente instrucción: `elEmpleado.horasTrabajadas = 1.5;`
- Se puede cambiar el valor del atributo sueldoXHora empleando la siguiente instrucción: `elEmpleado.sueldoXHora = 100;`
- Se puede consultar el valor del atributo cédula empleando la siguiente instrucción: `elEmpleado.cedula`

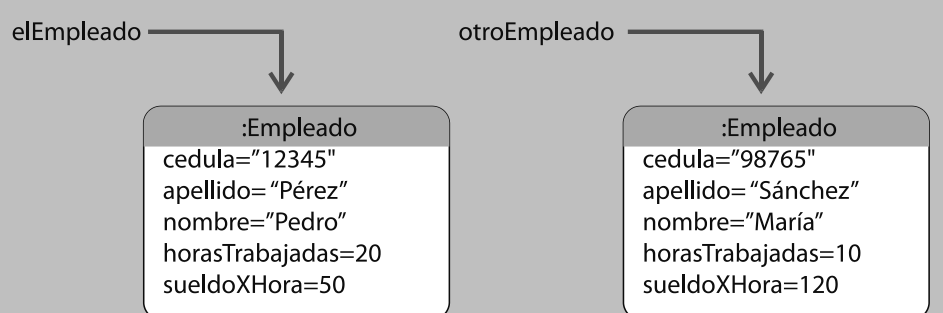
UN EJEMPLO AMPLIADO SOBRE LA INSTANCIACIÓN

Considere el siguiente código:

```
Empleado elEmpleado = new Empleado();
elEmpleado.cedula = "12345";
elEmpleado.apellido = "Pérez";
elEmpleado.nombre = "Pedro";
elEmpleado.sueldoXHora = 50;
elEmpleado.horasTrabajadas = 20;

Empleado otroEmpleado = new Empleado();
otroEmpleado.cedula = "98765";
otroEmpleado.apellido = "Sánchez";
otroEmpleado.nombre = "María";
otroEmpleado.sueldoXHora = 120;
otroEmpleado.horasTrabajadas = 10;
```

Después de ejecutar este código en Java se obtiene una configuración en memoria similar a la siguiente:



La anterior gráfica debe entenderse de la siguiente manera: existen dos variables de tipo `Empleado` que referencian a objetos de la clase `Empleado`; la primera referencia tiene el nombre de `elEmpleado` y apunta a un objeto cuyo valor para el atributo `cedula` es `12345`; la segunda referencia tiene el nombre de `otroEmpleado` y apunta a un objeto cuyo valor para el atributo `cedula` es `98765`. Es recomendable tener pendiente que los valores para los atributos de un objeto (por lo menos de la manera en que han sido definidos hasta el momento) son independientes de los valores que pueda tener cualquier otro objeto.

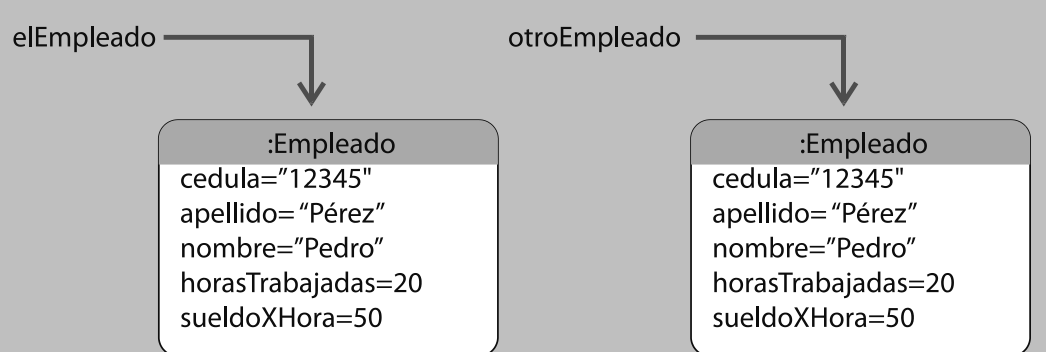
Sobre el operador lógico de comparación y su funcionamiento sobre las variables de tipo primitivo y de referencia

El lector perspicaz debe estarse preguntando lo siguiente, si los valores de los atributos de un objeto son totalmente independientes de los valores de los atributos de otros objetos de la misma clase, ¿qué evita que se puedan crear dos objetos con exactamente los mismos valores para sus atributos? La respuesta a este cuestionamiento es: nada. Considere el siguiente código:

```
Empleado elEmpleado = new Empleado();
elEmpleado.cedula = "12345";
elEmpleado.apellido = "Pérez";
elEmpleado.nombre = "Pedro";
elEmpleado.sueldoXHora = 50;
elEmpleado.horasTrabajadas = 20;

Empleado otroEmpleado = new Empleado();
otroEmpleado.cedula = "12345";
otroEmpleado.apellido = "Pérez";
otroEmpleado.nombre = "Pedro";
otroEmpleado.sueldoXHora = 50;
otroEmpleado.horasTrabajadas = 20;
```

En memoria se obtendría la siguiente configuración:



Es conveniente analizar el comportamiento de las variables al trabajar con el operador lógico de comparación que provee Java (este operador se denota empleando el símbolo del doble igual '==')