USERS

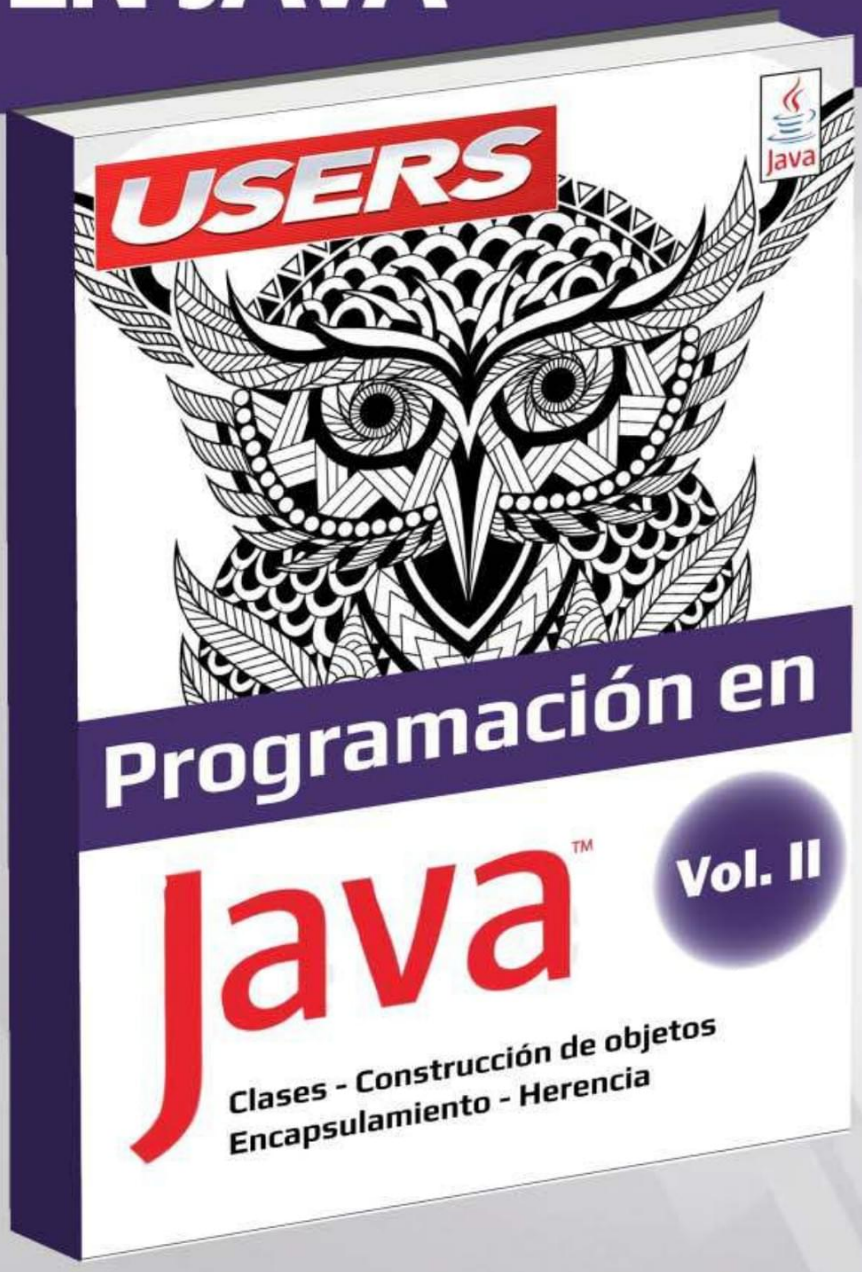
Programación en Java™

Vol. II

Clases - Construcción de objetos
Encapsulamiento - Herencia

CURSO DE

PROGRAMACION EN JAVA



Aprende a programar aplicaciones robustas y confiables. Escribe tus códigos una vez y ejecútalos en cualquier dispositivo.

Programación en Java™

Vol. II

Clases - Construcción de objetos
Encapsulamiento - Herencia

Título: Programación en Java II / **Autor**: Carlos Arroyo Díaz
Coordinador editorial: Miguel Lederkremer / **Edición**: Claudio Peña
Maquetado: Marina Mozzetti / **Colección**: USERS ebooks - LPCU289

PRÓLOGO

Java es un lenguaje maduro y robusto que, desde su nacimiento en el año 1995, ha demostrado que vino para quedarse y ha logrado evolucionar hasta convertirse en el lenguaje más utilizado en el mundo tecnológico.

Durante nuestro aprendizaje de cualquier lenguaje de programación podemos encontrarnos con variados libros, pero solo a través de la experimentación y escudriñando las entrañas de la Web (foros, blogs, under sites), podremos obtener los conocimientos necesarios para enfrentar el aprendizaje de mejor forma.

Mientras recorremos este camino, contar con un curso como este es fundamental pues se presenta como un compendio de todo lo que necesitamos para enfrentar un lenguaje de programación, permitiéndonos lograr, a través de ejemplos concretos, un mejor aprendizaje de los distintos temas necesarios para programar en Java.

Desde el primer momento somos optimistas en que este curso será para ustedes un desafío muy significativo y, por otro lado, logrará convertirse en una excelente guía del lenguaje Java.

Carlos Arroyo Díaz

Acerca del autor

Carlos Arroyo Díaz es programador de profesión, escritor especializado en tecnologías y docente por vocación. Se desempeña desde hace más de 20 años como docente en Informática General y en los últimos 10 años enseña programación en Java y Desarrollo Web. También se ha desempeñado como mentor docente en el área de Programación en varios proyectos del Ministerio de Educación de la Ciudad Autónoma de Buenos Aires, enseñando a programar a grupos de jóvenes.

ACERCA DE ESTE CURSO

Java es un lenguaje de programación que sigue afianzándose como un estándar de la web y, por eso, año tras año, aparece en el tope de las búsquedas laborales de programadores.

Es por esto que hemos creado este curso de **Programación en Java**, donde encontrarán todo lo necesario para iniciarse o profundizar sus conocimientos en este lenguaje de programación.

El curso está organizado en cuatro volúmenes, orientados tanto a quien recién se inicia en este lenguaje, como a quien ya está involucrado y enamorado de Java.

En el **primer volumen** se realiza una revisión de las características de este lenguaje, también se entregan las indicaciones para instalar el entorno de desarrollo y, posteriormente, se analizan los elementos básicos de la sintaxis y el uso básico de las estructuras de control.

En el **segundo volumen** se presentan las clases en Java, se realiza una introducción a los conceptos asociados a la Programación Orientada a Objetos y también se profundiza en el uso de la herencia, colaboración entre clases y polimorfismo.

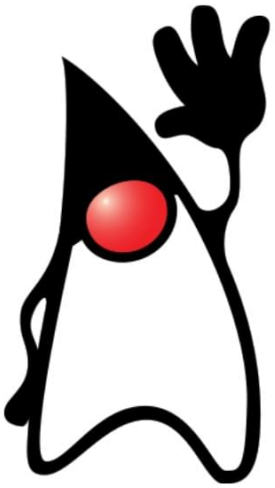
El **tercer volumen** contiene información sobre el uso de las clases abstractas e interfaces, el manejo de excepciones y la recursividad.

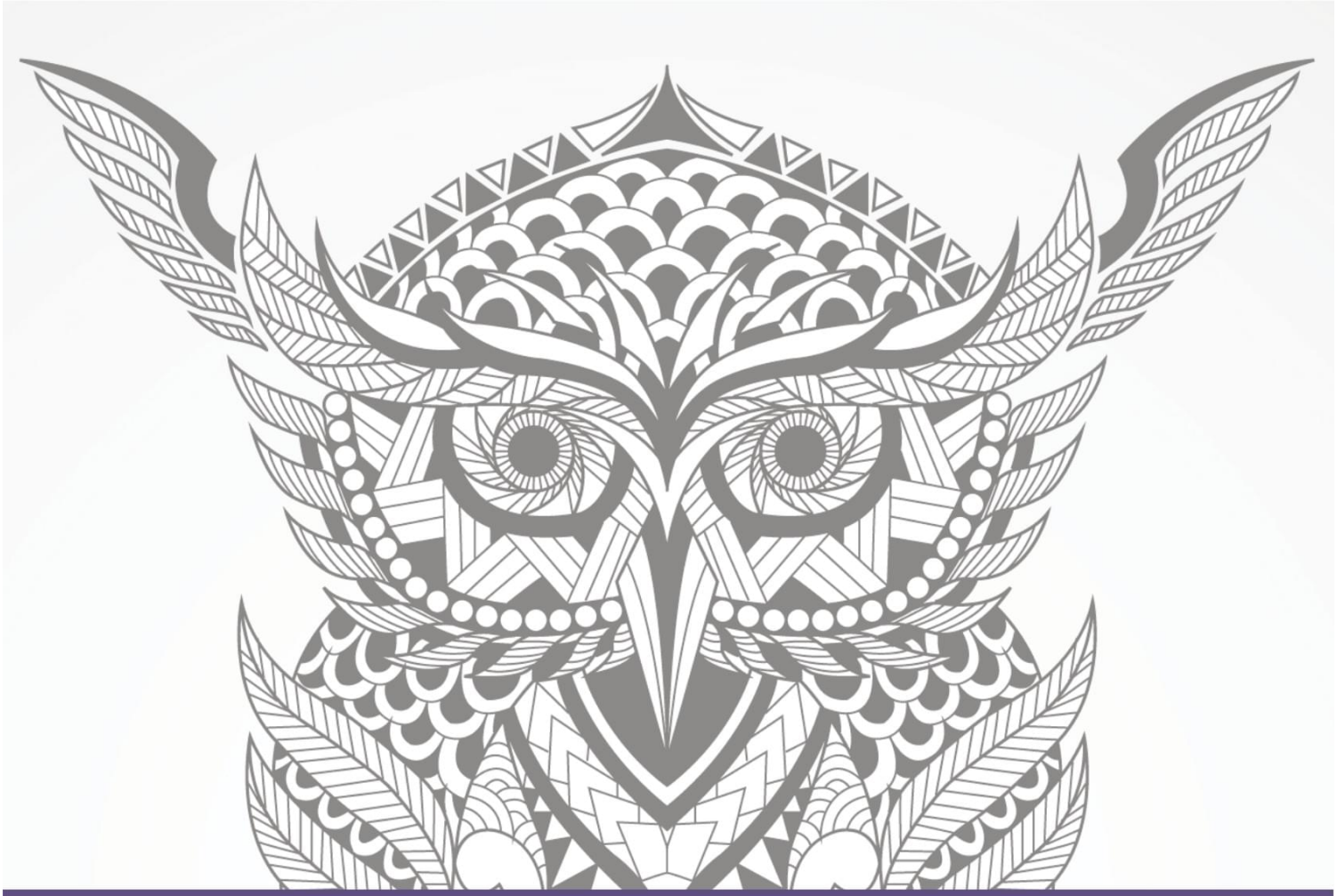
Finalmente, en el **cuarto volumen** se enseña el uso de las estructuras de datos dinámicas, el acceso a bases de datos y la programación Java para Android.

Sabemos que aprender todo lo necesario para programar en Java en tan solo cuatro volúmenes es un tremendo desafío, pero conforme vamos avanzando, el camino se va allanando y las ideas se tornan más claras.

¡Suerte en el aprendizaje!

SUMARIO DEL **VOLUMEN II**





Clases y objetos

Llegamos a un punto clave en nuestro aprendizaje de Java. Y, antes de comenzar a trabajar en la programación orientada a objetos, realizaremos una incursión en las clases y los objetos, pues se trata de elementos importantes en muchos lenguajes de programación modernos. En este capítulo, trabajaremos con clases y objetos de una manera individual, sin hacer uso de la colaboración entre las clases.

01

INTRODUCCIÓN A LAS CLASES

Una **clase** puede definirse como una especie de plantilla o molde. Aunque en el sentido estricto es una definición adecuada, intentaremos profundizar un poco más, para ello diremos que se trata de una **abstracción** de lo real (objetos), una forma de generalizar una idea para que podamos utilizarla en un programa.

En Java todo es clase y, también es objeto, ¿entonces?

Para ejemplificar esta idea, pensemos en un libro (abstracción), algunos pensaremos en un clásico; otros, en un libro de ficción, y otros, en un best-seller. Sin embargo, si solicitamos pensar en un libro de Borges, la situación cambia y vamos más allá; si pedimos pensar en El Aleph, ya tenemos algo concreto; de esta sencilla forma hemos creado un objeto de algo que se inició como una idea.

En Java todo es una clase y, al mismo tiempo, todo es un objeto. Por esta razón, cuando creamos un programa, Java nos invita a crear una clase pública que llevará el mismo nombre del archivo. La sintaxis de una clase es la siguiente:

```
public class Ejemplo {
    instrucciones…
}
```

Debemos utilizar el modificador para indicar que es pública, la palabra clave **class** y, luego, el nombre de la clase.

Las clases se heredan de una clase jerárquicamente superior llamada **Object**; sin embargo, cuando sea necesario heredar de otra clase, debemos utilizar la palabra clave **extends**. De esta forma, la sintaxis para una clase heredada será la siguiente:

```
public class ClaseHija extends ClasePadre{
    instrucciones…
}
```

Como vemos, una clase es una plantilla que define **atributos** (variables) y **métodos** (funciones). La clase se encarga de definir los

atributos y los métodos que son comunes a los objetos de ese tipo; pero luego, cada objeto adoptará sus propios valores, aunque todos compartirán las mismas funciones.

Al crear un objeto de una clase, podemos inferir que se crea una instancia de la clase o un objeto propiamente dicho.

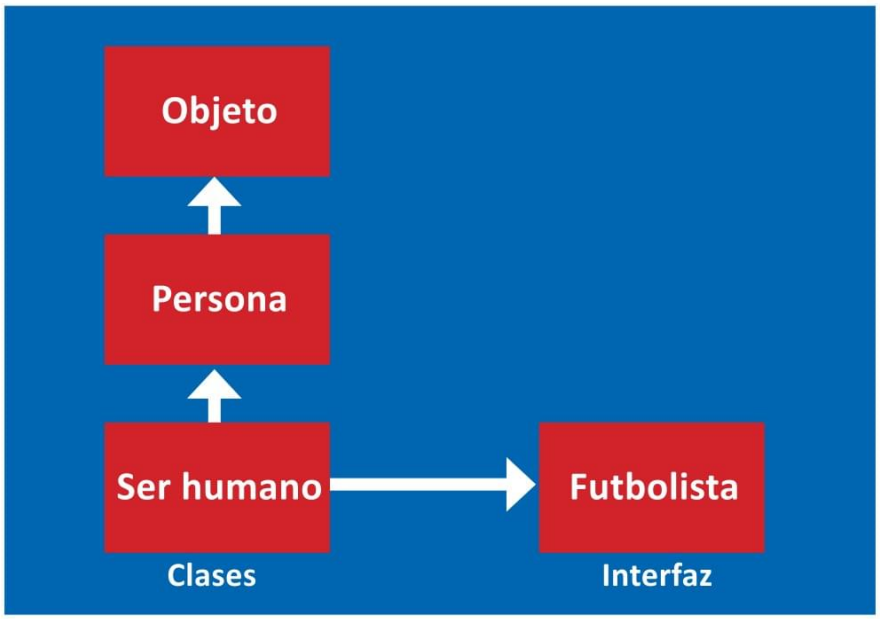


Figura 1. En este diagrama apreciamos la representación de una clase, una clase hija y una interfaz.

Veamos la sintaxis de la creación de una clase y los elementos que pertenecen a ella:

```
class [nombre de la clase] {
  [atributos o variables de la clase]
  [métodos o funciones de la clase]
  [main]
}
```

Para entender de mejor forma la creación e implementación de una clase, aplicaremos estos conceptos en la solución de un problema. Debemos crear una clase que permita realizar el ingreso de los datos, el código de un empleado y las horas trabajadas. Una vez ingresada esta información, deberán mostrarse los datos ingresados y, si el empleado trabajó más de 8 horas, mencionar que realizó horas extras. El código que necesitamos es el siguiente:

```
import java.util.Scanner;

public class Empleado {
     private Scanner teclado;
     private String nombre;
     private String apellido;
     private String codigo;
     private int hsTrab;

     public void inicializar() {
        teclado=new Scanner(System.in);
        System.out.print("Ingrese el nombre del empleado:
");
        nombre=teclado.next();
     System.out.print("Ingrese el apellido del
empleado:");
        apellido=teclado.next();
        System.out.print("Ingrese el código: ");
        codigo=teclado.next();
     System.out.print("Ingrese la cantidad de horas
trabajadas: ");
        hsTrab=teclado.nextInt();
         }
           public void imprimir() {
     System.out.println("Nombre: "+ nombre);
     System.out.println("Apellido: "+ apellido);
     System.out.println("Código: "+ codigo);
     System.out.println("Horas trabajadas: "+ hsTrab);
            }

       public void horasExtras() {
             if (hsTrab>8) {
      System.out.print(nombre+" "+apellido+ " trabajó
horas extras." );
             } else {
      System.out.print(nombre+" "+apellido+ " trabajó
horas normales." );
```

```
            }
        }

  public static void main(String[] ar) {
            Empleado empleado1;
            empleado1=new Empleado();
            empleado1.inicializar();
            empleado1.imprimir();
            empleado1.horasExtras();
        }
    }
```

Analizaremos ahora el código bloque a bloque, para verificar la participación de cada uno de los miembros de una clase.

En primer lugar, encontramos la clase que, en este ejemplo, se denomina **Empleado**:

```
public class Empleado {
}
```

Dentro de la misma clase definimos los atributos adecuados:

```
    private Scanner teclado;
    private String nombre;
    private String apellido
    private String codigo;
    private int hsTrab;
```

Crear una clase e instanciar un objeto

Ambas acciones van de la mano puesto que una depende de la otra. Instanciar es crear algo específico de algo general, otorgándole características propias. La clase es genérica, el objeto es específico.

Hemos definido todos los atributos con la cláusula del nombre clave **private**, por lo tanto, no nos permitirá el acceso desde otras clases o desde el **main**.

Para continuar, declaramos los métodos; la sintaxis es similar al **main**, aunque sin la palabra clave **static**:

```
  public void inicializar() {
          teclado=new Scanner(System.in);
         System.out.print("Ingrese el nombre del empleado: ");
          nombre=teclado.next();
  System.out.print("Ingrese el apellido del empleado:");
          apellido=teclado.next();
      System.out.print("Ingrese el código: ");
          codigo=teclado.next();
  System.out.print("Ingrese la cantidad de horas
trabajadas: ");
          hsTrab=teclado.nextInt();

      }
```

En el método **inicializar()**, que luego llamaremos desde el **main**, creamos el objeto de la clase **Scanner** y cargamos por teclado los atributos **nombre**, **apellido**, **código** y **hsTrab**; el método **inicializar()** puede acceder a los atributos de la clase **Empleado**.

Ahora pasemos al segundo método, que se encargará de imprimir el contenido de los atributos **nombre**, **apellido** y **hsTrab**, que fueron cargados previamente:

```
  public void imprimir() {
      System.out.println("Nombre: "+ nombre);
  System.out.println("Apellido: "+ apellido);
      System.out.println("Código: "+ codigo);
      System.out.println("Horas trabajadas: "+ hsTrab);
         }
```

El tercer método tiene como objetivo mostrar en un mensaje si el empleado trabajó o no horas extras:

```
  public void horasExtras() {
          if (hsTrab>8) {
      System.out.print(nombre+" "+apellido+ " trabajó horas
extras." );
          } else {
      System.out.print(nombre+" "+apellido+ " trabajó horas
normales." );
          }
```

Por último, en el método **main** declaramos un objeto de la clase **Empleado** y, luego, llamamos a los métodos creados antes en un orden adecuado:

```
  public static void main(String[] ar) {
        Empleado empleado1; //Declaración del objeto
        empleado1=new Empleado();//Creación del objeto
        empleado1.inicializar();//Llamada a un método
        empleado1.imprimir();
        empleado1.horasExtras();
      }
```

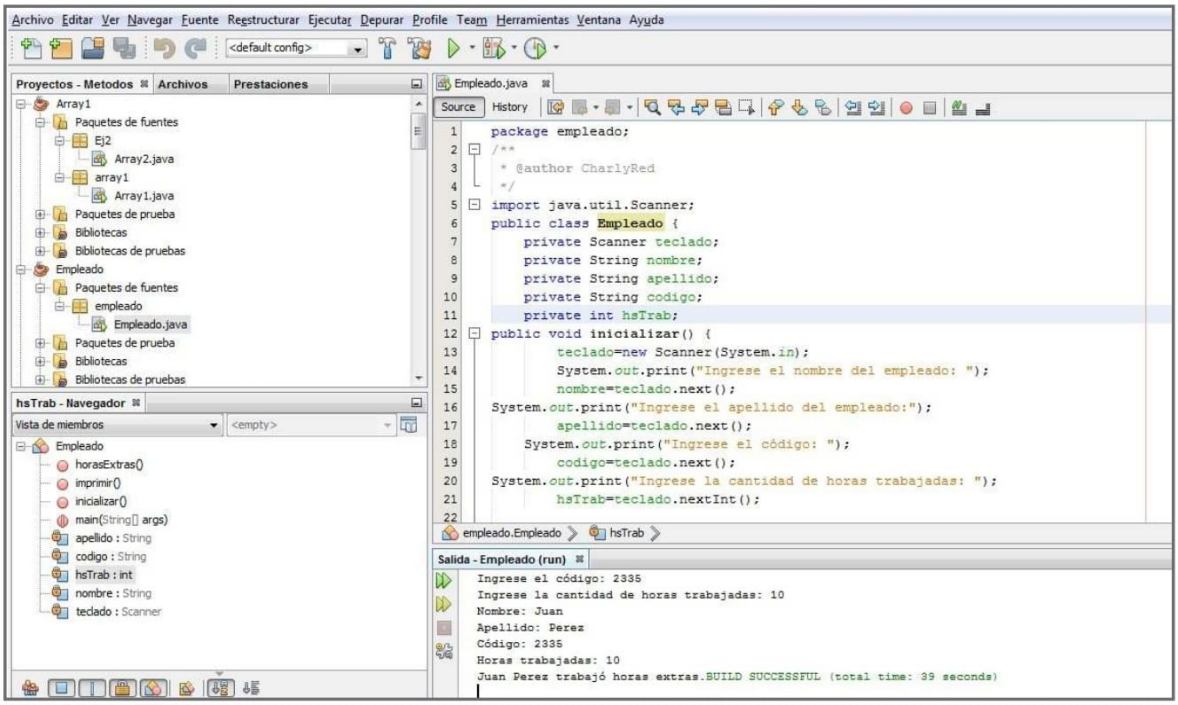


Figura 2. En el código mostrado en el IDE, observamos el listado de los miembros agrupados y en orden alfabético, no en orden de aparición.

Creación de los campos

Para ver el contenido de nuestra clase, debemos crear los diferentes campos de la clase; para lograrlo, nos basta con declarar variables en el interior del bloque de código de la clase e indicar la visibilidad de la variable, así como su tipo y su nombre. La sintaxis adecuada para la creación de los campos es la siguiente:

```
   [private | protected | public] tipoVariable
nombreVariable;
```

Para indicar la visibilidad de las variables podemos utilizar **private**, **protected** y **public**, o no especificar información.

ATRIBUTOS

Los métodos y los atributos que habitualmente utilizamos en una clase están asociados a las instancias de las clases o los objetos. Para simplificar esto podemos entender una **instancia de clase** como la concretización de una clase; de esta forma, los atributos vienen a ser las características de aquellos objetos instanciados.

Los atributos de una clase se definen según la siguiente sintaxis:

```
   [modificadorVisibilidad][modificadorAtributo]tipo
nombreVariable [=valorInicial];
```

La clase Object

Se trata de la clase raíz de todo el árbol jerárquico de clases de Java. Esta clase nos provee de una gran cantidad de métodos que serán utilizados por los objetos. En la documentación de la API de Java, encontraremos la lista completa de estos objetos; para acceder a ella hacemos uso de la siguiente dirección web: **https://docs.oracle.com/javase/10/docs/api/**.

En este código encontramos los siguientes elementos:

nombreVariable	Se trata del nombre de la variable.
modificadorVisibilidad	Indica desde qué parte del código se puede acceder a la variable, a esto lo llamaremos modificador de acceso a la variable.
valorInicial	Permite inicializar la variable con un valor.
modificadorAtributos	Se trata de las características específicas del atributo.

MODIFICADORES DE ACCESO

Los modificadores de acceso figuran en el encapsulamiento (tema que examinaremos en la introducción a la POO); sirven para controlar el acceso a los miembros de una clase además de vincular datos con el código que lo va a manipular, detalle muy importante en la seguridad de nuestros datos sensibles.

Por ahora presentaremos los tres modificadores: public, private y protected.:

public

Indica que se trata de un atributo accesible a través de una instancia del objeto, veamos un ejemplo:

```
public class Ejemplo{
    public String cadena = "Esto es un String";
    public int numero = 10;

    public void saludo(){
System.out.println("Este es un saludo de método");
     }
}
```

private

Indica que el atributo no es accesible. Al ser heredado, el atributo se convierte en inaccesible:

```
public class Ejemplo{
    private String cadena = "Esto es un String";
    private int numero = 10;

    private void saludo(){
System.out.println("Este es un saludo de método");
     }
}
```

protected

Indica que el atributo no es accesible, pero, al heredar, sí se puede usar desde la clase derivada. Veamos un código de ejemplo:

```
public class Ejemplo{
    protected String cadena = "Esto es un String";
    protected int numero = 10;

    protected void saludo(){
System.out.println("Este es un saludo de método");
     }
}
```

Sin especificar

También se conoce como *friendly*, sirve para indicar visibilidad de paquete, es decir, se puede acceder a través de una instancia, pero solo desde clases que se encuentren en el mismo paquete:

```
public class Ejemplo{
    String cadena = "Esto es un String";
    int numero = 10;
```

```
    void saludo(){
System.out.println("Este es un saludo de método");
     }
}
```

Modificador de atributos

Dependiendo de la pertenencia de algunos atributos, estos tendrán características propias en el objeto, en la vida del atributo o en la forma de acceso. Por esta razón, debemos tener en cuenta los siguientes modificadores:

static

Indica que el atributo pertenece a la clase, no a los objetos creados a partir de ella:

```
private String nombre;
 private int edad;
 static int contadorPersonas;
```

final

Indica que el atributo es una constante, en este caso debe presentar un valor inicial en forma obligatoria:

```
private static final float PI = 3.1416f;
```

transient

El atributo figurará como transitorio, por lo tanto, no podrá ser serializado:

```
private transient String password;
```

volatile

La forma de acceder a este atributo es mediante hilos o **threads** de forma asíncrona:

```
volatile int valor = 10;
```

¿QUÉ SON LOS OBJETOS?

Un elemento fundamental a la hora de programar en Java son los **objetos**. Aunque en el **capítulo 6** de esta obra realizaremos un análisis exhaustivo de estos elementos y profundizaremos en la forma en que trabajan dentro de las clases, en esta ocasión presentaremos los fundamentos de los objetos dentro de un programa.

Partamos de la siguiente premisa que ya hemos mencionado: en Java todo es clase y todo es objeto. Teniendo esto en cuenta, podemos pensar en que, en el mundo real, los objetos se crean a partir de otros objetos (por ejemplo, un lápiz antes fue madera y, antes, fue un árbol). Cada objeto tiene sus propios estados o **atributos,** y comportamientos o **métodos**. Por ejemplo, una persona tiene sus propios atributos, como altura, peso, color de cabello, y también comportamientos: piensa, corre, estudia, duerme; es decir, se trata de acciones que van a determinar a esa persona u objeto.

Por esta razón, podemos deducir que los objetos se crean a partir de un modelo o clase, tienen características propias (atributos) y realizan determinadas acciones (métodos); pero también debemos considerar que un objeto se crea, vive y muere.

Creación de objetos

Para tener en cuenta lo que es una **clase de objetos**, pensemos en un molde para hacer objetos determinados (un juguete, una galletita); ese molde es nuestra clase y lo usamos para crear el objeto. El molde debe agrupar las propiedades comunes de todos los objetos, pero no todos los objetos tienen que ser necesariamente iguales.

Declaraciones import

Si se utiliza el nombre de clase completamente calificado cada vez que se usa el nombre de una clase en el código fuente, el compilador no requiere declaraciones **import** en el archivo fuente de Java. Sin embargo, la mayoría de los programadores prefiere utilizar declaraciones **import**.

Un objeto se crea en el momento en que se define una variable de dicha clase, veamos un ejemplo:

```
Objeto = new Ejemplo();
```

Con esto hemos creado un objeto de una clase **Ejemplo**. A la creación de un objeto se la llama **instancia**, por lo tanto, si decimos "instancia de la clase **Ejemplo**", nos estamos refiriendo a un objeto creado de la clase **Ejemplo**.

CONSTRUCTORES

Un **constructor de clase** debe poner cada objeto a disposición para ser usado, es lo que llamaremos **inicialización**. El constructor inicializa el objeto en un estado razonable.

Algunas características de los constructores son las siguientes:

- Debe llevar el mismo nombre de la clase.
- No siempre retorna valor.
- Toda clase tiene al menos un constructor; aun cuando no se haya definido ninguno, va crear uno de forma predeterminada.
- Podemos crear tantos constructores como deseemos.

Existen tres tipos de constructores: constructor por defecto, constructor de copia y constructor común, analicémoslos en detalle.

Constructor por defecto

Es un constructor que no fue definido antes; es muy común encontrarlos en los programas. Veamos un ejemplo:

```
public class Empleado {
        private String nombre;
        private String apellido;
```

```
        private int codigoEmpleado;
}
    Empleado empleado=new Empleado();
```

En el ejemplo anterior no fue creado un constructor, por lo tanto, Java crea uno de forma predeterminada.

El operador **new** indica que va a crear un objeto con ese constructor. Otra forma de definir un método constructor es la siguiente:

```
public class Empleado {
        private String nombre;
        private String apellido;
        private int codigoEmpleado;

        public Empleado(){

         }
}
    Empleado empleado=new Empleado();
```

Constructor de copia

Cuando necesitamos inicializar un objeto con otro objeto de la misma clase, usamos este tipo de constructores. Analicemos el siguiente ejemplo:

```
public class Empleado {
        private String nombre;
        private String apellido;
        private int codigoEmpleado;

        //Constructor por defecto
        public Empleado(){
         }
          //Constructor copia
      public Empleado(Empleado empleado){
```

```
        this.nombre=empleado.Nombre;
        this.apellido=empleado.Apellido;
        this.codigoempleado=empleado.CodigoEmpleado;
          }
  }
```

En el código anterior, podemos notar el cambio de nombres (entre mayúsculas y minúsculas) en los objetos, lo realizamos para evitar inconvenientes entre nombres que se refieren al mismo objeto.

Entonces, el método de implementación sería el siguiente:

```
    Empleado empleado=new Empleado ();
    Empleado empleadoCopia=new Empleado(empleado);
```

Constructor común (con argumentos)

Este tipo de constructor recibirá los parámetros y, luego, los pasará al objeto creado. Veamos el ejemplo para conocer cómo se implementa:

```
  public class Empleado {
          private String nombre;
          private String apellido;
          private int codigoEmpleado;

           //Constructor común
    public Empleado(String nom, String ape, int codigo){
                this.nombre=nom;
                this.apellido=ape;
                this.codigoEmpleado=codigo;
          }
  }
           Empleado empleadoCopia=new Empleado
("Juan","Pérez",2334);
```

Podemos observar que inicializamos los valores del empleado con los datos que hemos pasado por parámetro.

Declaración de métodos

Cuando se plantea una clase en lugar de especificar todo el algoritmo en un único método, se dividen todas las responsabilidades de las clases en un conjunto de métodos. Un método presenta la siguiente sintaxis:

```
public void nombre del método() {
  instrucciones
}
```

En la sintaxis ofrecida, se muestra el método predeterminado, es decir, **sin parámetros**.

Métodos con parámetros

Cuando un método tiene parámetros dentro de los paréntesis, se instancian las variables u otros argumentos:

```
 public void nombre del método(parámetros) {
  instrucciones…
}
```

A estos parámetros, podemos mencionarlos como **variables locales al método**, su valor se inicializa con datos que llegan cuando los llamamos.

Constructor vacío

Un **constructor** se denomina **vacío** cuando no lleva parámetros. Si no se define ningún constructor, el compilador creará uno por defecto; esto permite crear objetos con **New NombreClase()** sin argumentos, aunque no se haya definido ningún constructor. Muchos frameworks e IDE precisan la definición de objetos con constructores vacíos. Por esta razón, es común agregar un segundo constructor sin saber que el compilador creará un constructor vacío, y esto acarreará problemas y errores.

Veamos un ejemplo para clarificar el uso de los métodos dentro de las clases:

```
      int horas(int seg){
             int h = seg/3600;
             return h;
          }

    int minutos(int seg){
              int m = (seg%3600)/60;
              return m;
          }

    int segundos(int seg){
               int s = (seg%3600)%60;
               return s;
          }
```

Como podemos apreciar, dentro de los métodos **horas**, **minutos** y **segundos** figuran los parámetros que serán utilizados con posterioridad.

Métodos que retornan un dato

Cuando un método retorna un dato, en vez de indicar la palabra clave **void** previo al nombre del método, señalamos el tipo de dato que retorna. Luego, dentro del algoritmo, en el momento que queremos que este finalice y retorne el dato, empleamos la palabra clave **return** con el valor respectivo. Esta es la sintaxis adecuada:

```
  public tipo de dato nombre del método(parámetros) {
   instrucciones…
   return tipo de dato
}
```

Veamos un ejemplo para analizar la forma en que se implementan métodos que retornan datos. Pensemos en un programa que pide ingresar tres valores por teclado. Luego de esto, deberá mostrar el valor mayor y el menor.

Veamos el código que puede ayudarnos a calcular el mayor de tres valores:

```
      public int calcularMayor(int num1,int num2,int num3)
{
          int mayor;
          if(num1> num2 && num1> num3) {
            mayor= num1;
          } else {
              if(num2> num3) {
                  mayor= num2;
              } else {
                   mayor= num3;
              }
          }
          return mayor;
      }
```

Podemos observar que el método retorna un entero y recibe tres parámetros:

```
      public int calcularMayor(int num1,int num2,int num3)
{
  }
```

Dentro del método verificamos cuál de los tres parámetros almacena un valor mayor, guardamos este valor en una variable local llamada **mayor**, este valor lo retornamos con un **return**.

La llamada al método **calcularMayor()** la hacemos desde adentro de otro método, al que llamamos **cargarValores()**:

```
          mayor = calcularMayor(valor1,valor2,valor3);
```

Es necesario asignar a una variable el valor devuelto por el método **calcularMayor()**. Luego mostramos el contenido de la variable mayor:

```
          System.out.println("El valor mayor de los tres
es: " +mayor);
```

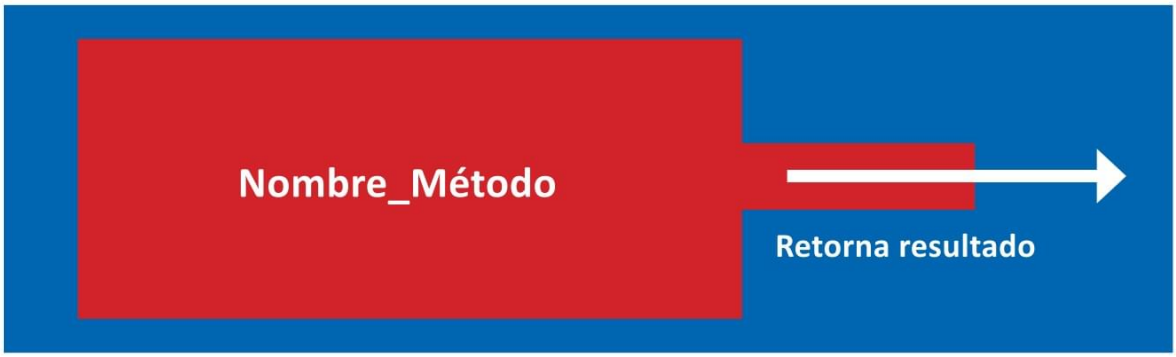


Figura 3. En la imagen podemos ver la forma en que se retorna un resultado ante la llamada al método.

El tipo de retorno puede ser cualquier tipo de dato, tipo básico del lenguaje o tipo objeto. Si el método no tiene información para devolver, usaremos la palabra clave **void** en lugar del tipo de retorno.

Creación de métodos

Los **métodos** son funciones definidas en el interior de una clase, que sirven para manejar los campos de dicha clase. La sintaxis de declaración de un método es la siguiente:

```
  modificadores tipoDeRetorno nombreDelMétodo
(listaParámetros)
       throws listaExcepción{
      instrucciones…
  }
```

Figura 4. Aquí podemos ver gráficamente la carga de varios parámetros y la llamada a un valor del método.

Entre los modificadores de métodos con que Java cuenta encontramos:

private	el método solo puede ser usado en la clase donde fue definido.
protected	el método solo se puede usar en la clase donde fue definido, en las subclases de esta clase y en las otras clases que pertenecen a un mismo paquete.
public	se puede utilizar el método desde cualquier otra clase.
	Si no se utiliza ninguna de estas palabras reservadas, entonces la visibilidad se limitará al paquete donde está definida la clase.

Es necesario tener en cuenta que existen otros modificadores que son utilizados para tareas específicas, por lo que no se encuentran en el ámbito del presente libro:

static	Indica que el método es un método de clase.
abstract	El método es abstracto y no contiene código. La clase donde está definido también debe ser abstracta.
final	El método no puede sobrescribirse en una subclase.
native	Usado cuando el método se encuentra en un archivo externo escrito en otro lenguaje.
synchronized	El método solo puede ser ejecutado por un único hilo a la vez.

Destructores en Java

En Java no existen los destructores, sino que la manera de limpiar la memoria es a través de su recolector de basura: **garbage collector**. Este se encarga de recolectar todas las variables u objetos que no se estén utilizando, al igual que aquellos para los que no haya referencias por una clase en ejecución. El programador puede decidir en qué momento pasará el recolector de basura. Para lograrlo, hay que escribir **System.gc()**.

USO DE THIS

Como sabemos, un **constructor** es un método especial que se ejecuta cuando creamos un objeto. A diferencia de un método común, no lleva un tipo de retorno, y su nombre es el mismo que el de la clase. Veamos la sintaxis de la implementación:

```
class Empleado{
    public Empleado(){
     instrucciones…
     }
    public Empleado(String nombre){
     instrucciones…
     }
}
```

Un método miembro es un método asociado al objeto, es decir, no se trata de un método estático y, cuando se está ejecutando, lo hace asociado a un objeto.

Ahora bien, ¿cómo podemos hacer una referencia al objeto que se esté ejecutando? En realidad, no existe una variable que refiera a dicho objeto, por esta razón entra en acción **this**, que actúa como una variable que referencia al objeto. En los ejemplos de la clase **Empleado**, podemos usar **this.nombre** para acceder al nombre del objeto asociado.

La palabra clave **this** funciona de la misma manera dentro de un constructor como dentro de un método miembro. En el siguiente ejemplo (propuesto antes), haremos un recorrido a modo de resumen de todo lo que hemos visto hasta este momento, incluyendo el uso de **this**:

```
package empleado;
public class Empleado {
    String nombre, apellido;
    int codigoEmpleado;

    public Empleado(){ //método sin parámetros
    this.nombre = "No informado";
```

```
      this.apellido = "No informado";
      this.codigoEmpleado = 0;
           }
  public Empleado(String nombre){ //con un parámetro
      this();
      this.nombre = nombre;
  }

  public Empleados(String nombre, String apellido, int
codigoEmpleado){ //método con varios parámetros
      this.nombre = nombre;
      this.apellido = apellido;
      this.codigoEmpleado = codigoEmpleado;
  }

  public String toString(){ //Método del objeto String
  return "Empleado = "+ this.nombre + " " + apellido + "
Código: " + codigoEmpleado;
  }
       public static void main(String[] args) {
       Empleado empleado1 = new Empleado();
       Empleado empleado2 = new Empleado("María");
       Empleado empleado3 = new Empleado("Juan","Pérez",
2335);
            System.out.println(empleado1.toString());
            System.out.println(empleado2.toString());
            System.out.println(empleado3.toString());
       }

  }
```

Cuando ejecutamos este programa, arrojará la siguiente información:

```
Empleado = No informado No informado Código: 0
Empleado = María No informado Código: 0
Empleado = Juan Pérez Código: 2335
```

Al crear el método sin parámetros, no arrojará información; cuando colocamos un solo parámetro (en este caso el nombre), solo arrojará el nombre del empleado que hemos llamado. Finalmente, si referenciamos a todos los objetos, observaremos que nos mostrará los datos del empleado en forma completa.

GARBAGE COLLECTOR Y EL MÉTODO FINALIZE()

Cuando creamos programas en Java, estos se almacenarán en la memoria principal, por esta razón, cuando creamos variables e instanciamos objetos, se ralentizará el programa ya que existirán elementos que no se están utilizando.

En este punto es importante el denominado **garbage collector**, se trata de una especie de limpiador que escanea dinámicamente en la memoria buscando objetos que ya no estén referenciados.

Para familiarizarnos con el uso y las características de este recolector de basura, podemos visitar la página oficial de Oracle, en la siguiente dirección: **https://docs.oracle.com/javase/10/docs/api/java/lang/System.html#gc()**.

Para profundizar en este recolector, debemos entender cómo funciona la memoria. Esta contiene dos elementos fundamentales: **stack** y **heap**, en el primero se guardan los valores primitivos (int, float, double, boolean, char, etcétera). En el segundo, se alojan todos los objetos creados en un programa.

Si tenemos una clase **Empleado**, esta creará una instancia de ella, o un objeto de clase en el heap, mientras que creará una referencia a ese objeto en el stack.

Garbage collector apunta a un espacio de memoria. Como cada espacio de esa memoria está reservado para alguna aplicación, rutina, etcétera, cuando creamos un objeto, estamos creando un **puntero**, que señalará ese espacio de la memoria. Por ejemplo cuando hacemos la instrucción **new Objeto()**, se reserva un espacio de la memoria para alojar el objeto que acabamos de instanciar.

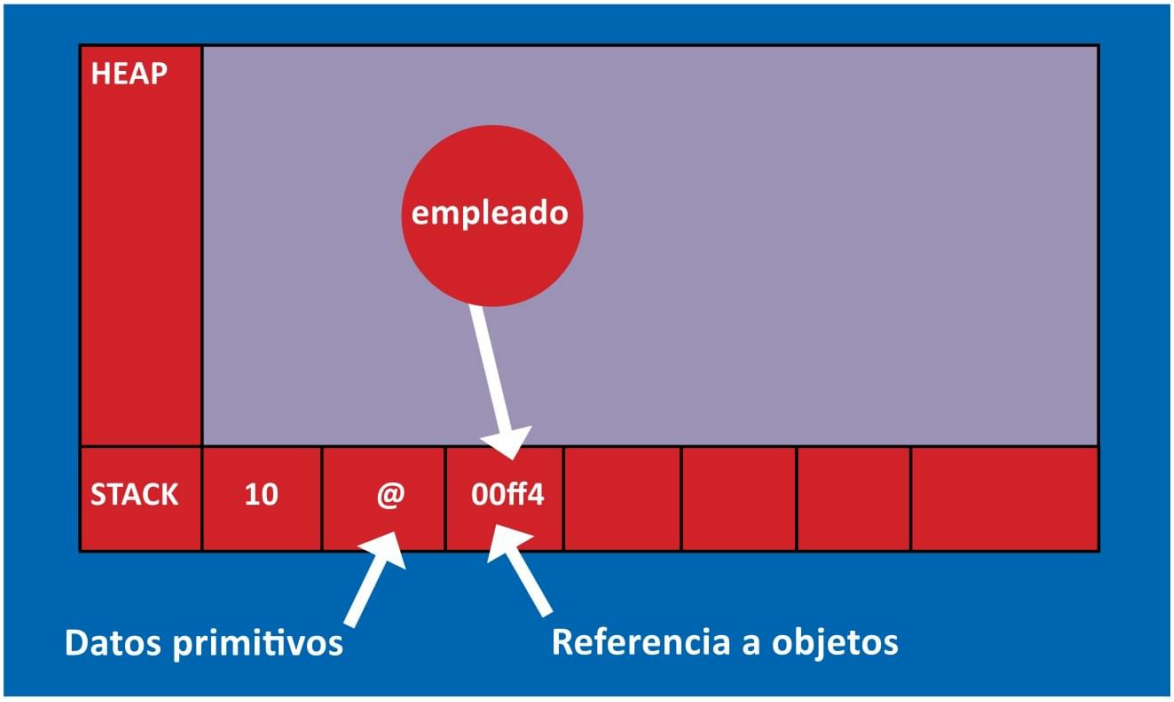


Figura 5. Este esquema muestra el funcionamiento de la memoria y el espacio asignado a cada uno de los objetos y tipos primitivos.

Veamos un ejemplo:

```
Empleado empleado1 = new Empleado();
Empleado empleado2 = new Empleado();
```

Hemos reservado dos espacios de memoria para los dos objetos referenciados: **empleado1** y **empleado2**. Si asignamos valores a los objetos antes creados, veremos lo que sucede:

```
Empleado empleado1 = new Empleado("Juan");
Empleado2 = empleado1;
```

A simple vista, estamos asignando los atributos del **empleado1** al **empleado2**, sin embargo, esto no es así y resulta muy importante reconocer este tipo de asignaciones, pues lo que realmente hacemos es que **empleado2** apunte al **empleado1**, es decir, a la misma ubicación de memoria a la que apunta **empleado1**. Esta es una de las situaciones que causan varios dolores de cabeza a los iniciados en este lenguaje, sin embargo, más adelante veremos cómo solucionarlo gracias a los **getters** y **setters**.

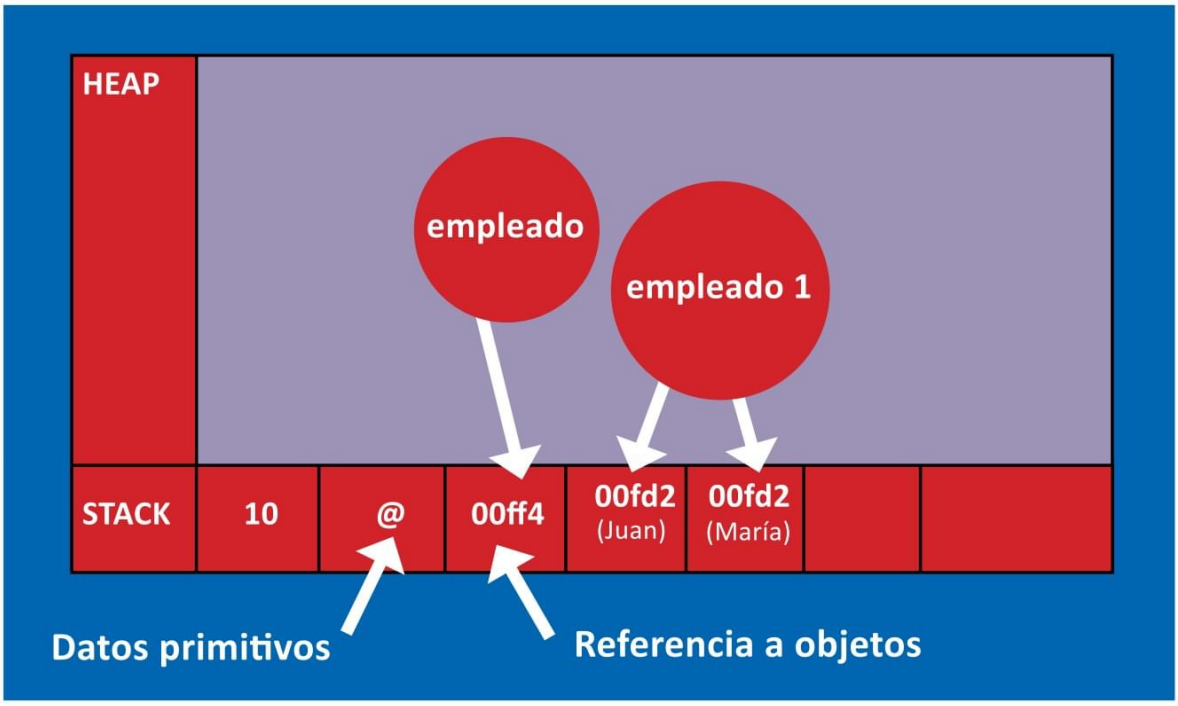


Figura 6. En este esquema se muestra cuándo dos objetos instanciados referencian al mismo espacio de memoria.

PAQUETES

Java provee una utilidad que nos permite almacenar códigos relacionados entre sí para poder utilizarlos en un nuevo programa, de tal manera que no sea necesario copiarlo o rehacerlo, al contrario, solo se precisa importarlo directamente y acceder a él.

Los denominados **paquetes** nos ayudan a organizar y administrar de una mejor manera nuestros proyectos. Otra de las ventajas de la utilización de los paquetes o **packages** es reducir los conflictos entre los nombres de las clases que se llaman de la misma manera, pero que se encuentran en paquetes diferentes, y también proteger el acceso a las clases. Un ejemplo que ya hemos visto es el uso de la clase **Scanner**, que se encuentra dentro del paquete **java.util**. Para tener acceso a cada uno de sus métodos y utilizarlos debemos escribir lo siguiente:

```
import java.util.Scanner;
```

Para referirnos a una clase de un paquete, es necesario anteponer el nombre de la clase al nombre del paquete, exceptuando aquellas ocasiones en que el paquete haya sido importando explícitamente. Veamos un ejemplo:

```
pqtMatriz.MatrizCuadrada
```

En esta ocasión, el paquete se llama **pqtMatriz** y contiene una clase denominada **MatrizCuadrada**. Entonces, para importarla en un proyecto, debemos usar la palabra reservada **import**, de esta manera accederemos a su contenido:

```
import pqtMatriz;
```

Ahora podemos referirnos a la clase **MatrizCuadrada** simplemente escribiendo su nombre sin necesidad de escribir qué paquete la contiene.

De la misma forma, es posible acceder a todos los paquetes que Java nos ofrece; si queremos conocer todos los que existen hasta la última versión, debemos visitar el sitio oficial de Oracle, en la siguiente dirección: **https://docs.oracle.com/javase/10/docs/api/overview-summary.htm**.

Creación de paquetes

Para crear un paquete en Java, necesitamos crear una carpeta (no importa por ahora el lugar), a la que llamaremos **misPaquetes**. Dentro de ella crearemos otra, a la que llamaremos **programas**, y otra carpeta a la que denominaremos **primeros**. Ahora escribiremos un sencillo programa de Java:

```
package misPaquetes.programas.primeros;
```

Le estamos indicando que nuestro programa va a quedar dentro de este paquete. Ahora definiremos nuestra clase:

```
public class Programita{
    public void imprimir(){
```

```
        System.out.println("\nEsto es un programa desde el
paquete primeros");
        }
   }
```

El modificador de acceso debe ser público tanto en la clase como en el método, de esta forma será posible acceder a la clase desde cualquier otro paquete. Cuando importemos la clase del paquete creado antes, debemos hacerlo de la siguiente manera:

```
   import misPaquetes.programas.primeros.*;
```

Con esto traeremos a las clases que hay dentro de él.

RESUMEN CAPÍTULO 01

En este capítulo analizamos las clases, el control de accesos a los miembros, la creación de constructores y los atributos. Hicimos una incursión a los métodos, y analizamos de una manera práctica su uso dentro de las clases y la importancia de estos en el funcionamiento de los programas. Luego hicimos una referencia a objetos mediante el uso de this, y vimos cómo hace Java para recolectar la basura y liberar la memoria. Finalmente analizamos la creación y el acceso a los packages.

Actividades 01

Test de Autoevaluación

1. ¿Qué es una clase?
2. ¿A qué llamamos instancia de una clase?
3. ¿Qué es un objeto?
4. ¿Qué son los atributos? Cite tres ejemplos.
5. Explique la diferencia entre private y protected.
6. ¿Qué es un constructor?
7. ¿Cómo referenciamos los objetos a través de this?
8. ¿Existe un recolector de basura en Java?
9. ¿Qué función cumplen los paquetes?
10. Explique cómo se crean los paquetes.

Ejercicios prácticos

1. Cree una clase, luego instancie un objeto de ella.
2. Cree un método y provoque acceso a los miembros de la clase.
3. Cree tres métodos, uno sin parámetros, otro con un parámetro y otro con tres parámetros. Luego invoque a esos métodos.
4. Utilizando los conceptos de clases y objetos, genere un programa en donde se ingrese y almacene, por teclado, la información de varios alumnos: nombre, edad, grado. Luego haga el llamado de alguno de los datos a través de objetos.
5. Utilizando la importación de clases y paquetes, cree un programa que haga las operaciones aritméticas básicas, e incluya los métodos matemáticos pow y sqrt.

Programación orientada a objetos

La programación orientada a objetos (POO) es uno de los paradigmas más usados en la programación, por ello en esta ocasión veremos a fondo las clases y los objetos, cómo se crean, su tratamiento, la interacción con otro tipo de objetos y clases, y también algunos temas más profundos de este paradigma, como la herencia, el polimorfismo y el encapsulamiento.

02

INTRODUCCIÓN A LA POO

La **programación orientada a objetos** o **POO** es un paradigma de programación que define y organiza el software basándose en entidades que denominamos **objetos**. La POO se relaciona con la programación, base de datos, procesos de desarrollo, arquitectura de información y comunicaciones.

Java es 100% orientado a objetos, pues intenta trasladar la naturaleza de los objetos de la vida cotidiana a la programación, es decir, que los objetos tienen un estado, un comportamiento y propiedades.

Por ejemplo, llamemos objeto a un auto, una mesa o a seres vivos, como un perro o una persona. Tomemos como ejemplo el objeto auto y hagamos las siguientes preguntas:

¿Cuál es el estado del auto? Puede estar estacionado, detenido, circulando.

¿Qué propiedades tiene? Posee un color, un tamaño, un modelo.

¿Qué comportamiento tiene? Puede arrancar, frenar, acelerar, doblar, etcétera.

Los objetos combinan datos, comportamiento e identidad. En consecuencia, pensar de esta manera tiene sus ventajas pues dividimos a los objetos en elementos cada vez más pequeños. En el ejemplo que mencionamos, dividimos al auto en el motor, los asientos, las ruedas, el tablero de mando, etcétera.

El uso de la POO supone algunas ventajas, por ejemplo, la **reutilización de los códigos** (**herencia**); en el ejemplo mencionado, podemos utilizar los repuestos de un auto para otro del mismo tipo.

Por otra parte, si existiera un error en alguna línea de código, la ejecución puede continuar gracias al tratamiento de excepciones. Podría romperse alguna parte no fundamental del auto y este seguiría andando, ya que está dividido en partes o módulos.

Por último, es necesario mencionar que existen algunos objetos que no dependen directamente de otros, por ejemplo, el volante y la batería no dependen uno del otro. Sin embargo, estos objetos funcionan y logran que el todo marche correctamente, a esto se conoce como **encapsulamiento**.

Si tenemos en cuenta todo lo que hemos mencionado, un programa

en Java es un conjunto de clases unidas entre sí, de tal manera que funciona como una unidad. Hasta ahora, todos los programas que hemos desarrollado a lo largo de este libro, constaban de una única clase pues tenían como objetivo aprender la sintaxis y las características del lenguaje.

CLASES Y OBJETOS

Una **clase** es una forma generalizada de un modelo, en la que se describen las características más comunes de un grupo de objetos.

Si pensamos en una fábrica de autos, el plano y posteriormente el molde del cual se construirán los autos corresponderán a la clase, y cada uno de los autos que producirá la planta de montaje serán los objetos, que van a compartir una serie de características del molde original. Sin embargo, cada uno de ellos tendrá una identidad propia denominada **atributos** o propiedades y, también, **métodos** o comportamientos.

Básicamente, una clase genérica se compone de la **declaración** de la clase y el **cuerpo** de esta, que a su vez se puede dividir en la sección de la **declaración**, y, opcionalmente de la **inicialización** de los atributos o **variables miembro**, y la declaración e implementación de los métodos, llamados **funciones miembro**. De esta manera, la sintaxis adecuada es la siguiente:

Elementos de la POO

La POO ofrece características especiales que han cambiado de manera sustancial la forma de programar, por ejemplo, las **clases** y los **objetos**, elementos fundamentales en un programa de Java. Por otra parte, los **métodos** que se encargan de generar la comunicación entre las distintas clases. Asimismo, los objetos deben tener un **estado** y varios **atributos** que los identifican y diferencian respecto de otros objetos y, finalmente, la **herencia**, que nos permite reutilizar códigos, en tanto que el **polimorfismo** permite que los objetos se comporten de distintas maneras a lo largo de un programa.

```
DeclaraciónClase{
    Declaración Atributos;

    Declaración Métodos(){
}
}
```

Veamos un ejemplo en el que podremos analizar cómo se construye una clase y, dentro de ella, un conjunto de objetos.Para acceder a un atributo de un objeto, lo hacemos de la siguiente forma:

```
NombreObjeto.propiedad = [valor de la propiedad];
```

De la misma forma, para acceder al comportamiento de un objeto sería:

```
NombreObjeto.método();
```

PROPIEDADES DEL OBJETO

*Mercury.color="azul"
*Mercury.año=2015;
*Mercury.peso=1450;
*Mercury.puertas=4;

COMPORTAMIENTO DEL OBJETO

*Mercury.arranca();
*Mercury.frena();
*Mercury.gira();
*Mercury.acelera();

Figura 1. Pseudocódigo de la construcción del modelo de objeto de autos, en este caso tenemos las características y el comportamiento.

Objetos

Un **objeto** es una instanciación de una variable cuyo tipo de datos es una clase.

En sí, en Java todo es objeto, en consecuencia, todo es clase, puesto que un objeto es la concretización de la clase que lo precede.

Si pensamos en un perro (una clase), la idea es muy general; luego pensamos en nuestro perro, este tiene un nombre, unas características y sus comportamientos propios.

Creación de una clase

Para aprender la forma adecuada de crear una clase en NetBeans, comenzaremos creando la clase por la que se seguirán el resto de los elementos inherentes a ella.

Para lograrlo, debemos crear un nuevo proyecto haciendo clic en **Archivo/Proyecto nuevo**. A continuación, elegimos la carpeta **Java** y luego, en la opción **Java Application**, hacemos clic en el botón **Siguiente**.

Ahora es el momento en que podremos elegir el nombre del proyecto, la ubicación o también ubicar la clase dentro de un proyecto ya existente.

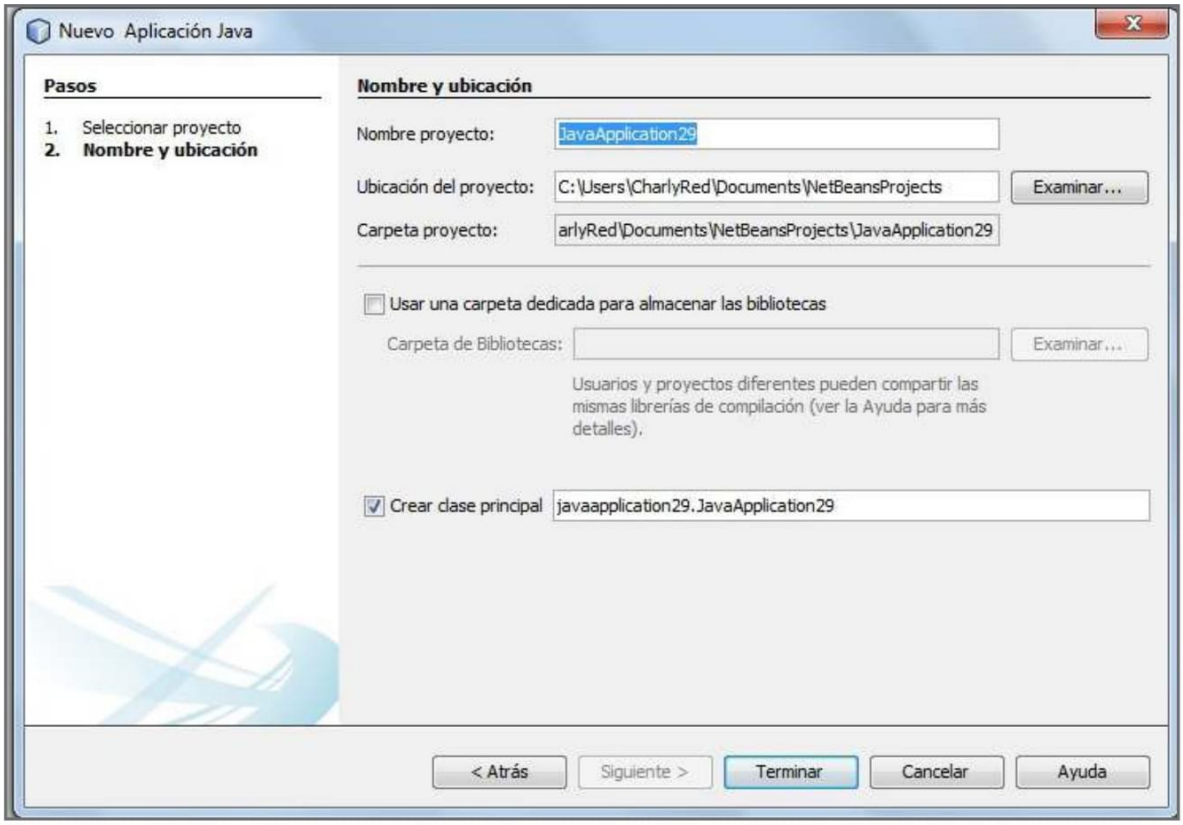


Figura 2. Una vez que hayamos elegido las opciones adecuadas para el nuevo proyecto, hacemos clic en Finalizar.

Ahora debemos eliminar la clase predeterminada para empezar todo desde cero. Crearemos una clase llamada **Auto**, para ello hacemos clic con el botón derecho del mouse en el nombre del paquete y elegimos la opción **Nuevo/Java class**. Esta clase será la base para el resto de los objetos que crearemos.

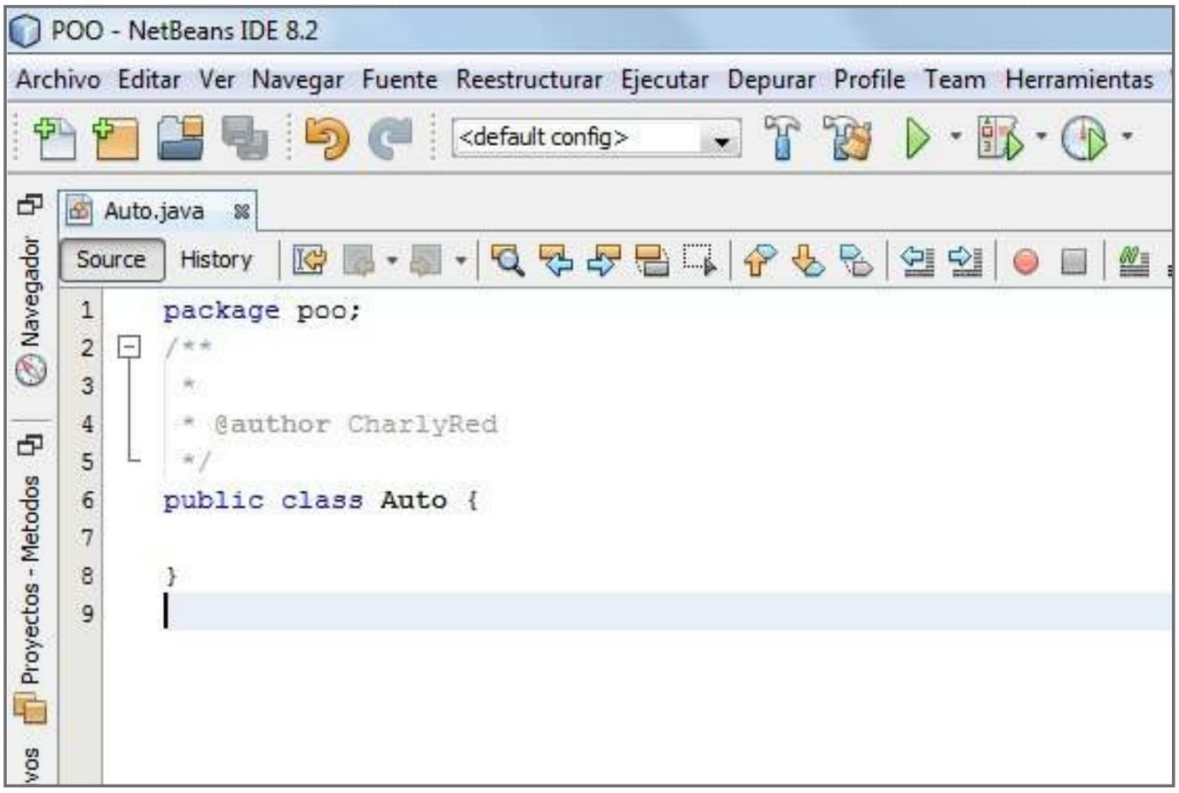


Figura 3. En esta imagen vemos la vista general de la creación de una clase y la estructura que existe dentro de ella.

Ahora nos dedicaremos a crear los atributos para nuestra clase **Auto**:

```
public class Auto {
int ruedas;
int ancho;
int largo;
int motor;
int peso;
}
```

Como ya tenemos las características o atributos de la clase **Auto**, le asignaremos los valores que tendrá cada una de ellas, recordemos que en el **capítulo 5** ya realizamos una incursión en este procedimiento. En esta ocasión vamos a realizar un **método constructor** que se encargará de otorgar un estado inicial a nuestro objeto. La manera de hacerlo es la siguiente:

```
public Auto() {//Constructor de la clase
        ruedas = 4;
        ancho = 300;
        largo = 2200;
        motor = 1600;
        peso = 600;
    }
```

Esto, de alguna forma, corresponde a inicializar los atributos creados en el código anterior. Ahora debemos crear una nueva clase, sin embargo, esta vez contendrá el método **main**, tal como vemos a continuación:

```
package poo;
public class Uso_Coche {

public static void main(String[] args) {

}
}
```

Lo que haremos es crear objetos que comparten la estructura común de la clase **Auto**; a esto lo denominamos **instanciar una clase**, un concepto que vamos a utilizar mucho en la programación con Java:

```
Auto mercury = new Auto();
```

Creamos un objeto de la clase **Auto**, en este caso lo llamamos **mercury**, luego escribimos la palabra clave **new** y el nombre del constructor de la clase, en este caso **Auto()**.

Cuando escribimos el nombre del objeto y luego un punto (.), el IDE nos mostrará las propiedades que pertenecen a la clase **Auto**, esto resulta bastante práctico a la hora de programar. Veamos el código completo:

```
  Public class Uso_Coche {

  Public static void main(String[] args) {
            Auto mercury = new Auto();
  System.out.println("Este coche tiene: " + mercury.ruedas
+ " ruedas");
      }
      }
```

En el código anterior, hemos instanciado la clase **Auto** mediante un constructor llamado **Auto()**, el que llama a la clase y permite utilizar las acciones que ella contenga. En este caso traeremos a los atributos que contenga el objeto **mercury** (por ejemplo las ruedas), aunque también podemos llamar a otros elementos de ese objeto.

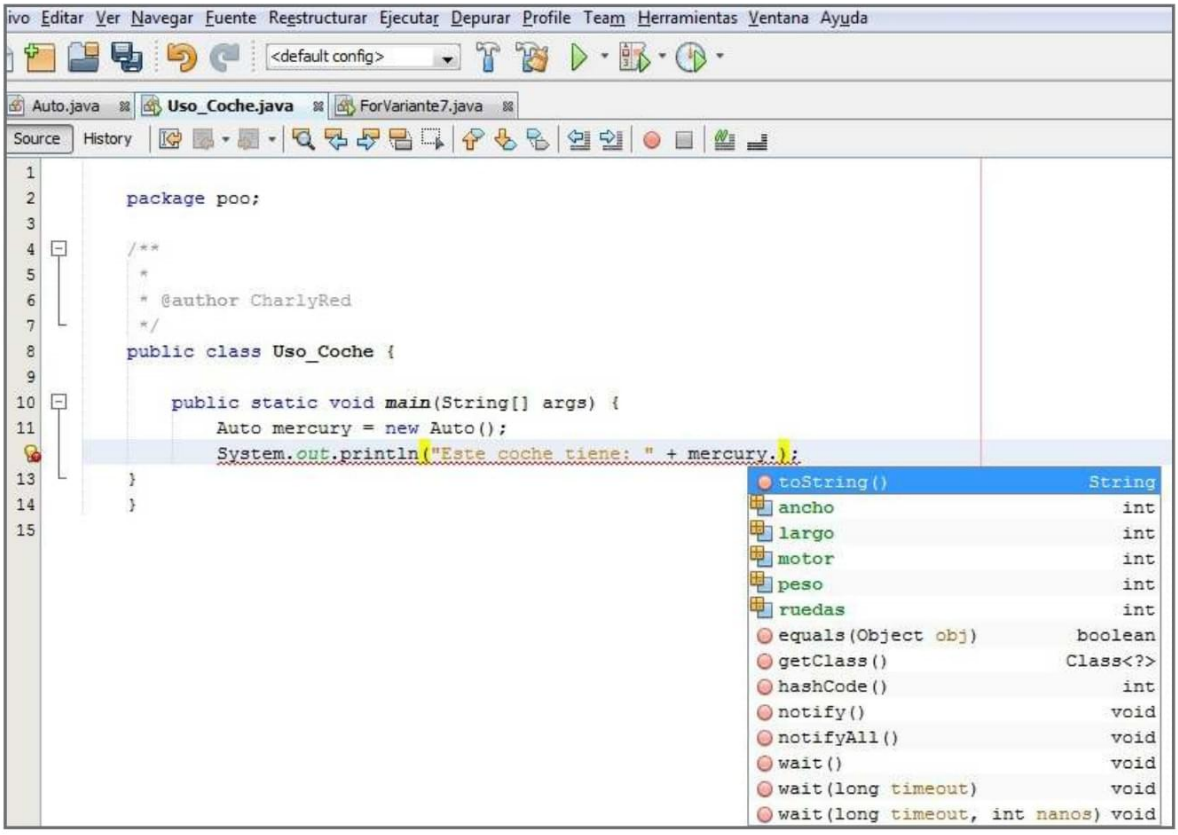


Figura 4. Notemos que, cuando escribimos el objeto y un punto (.), el IDE mostrará en forma automática los atributos generales creados en la clase, lo que nos resulta ventajoso y rápido.

MODULARIDAD Y ENCAPSULAMIENTO

Si no separamos en tareas más pequeñas nuestros programas, a futuro tendremos muchos problemas para detectar errores y depurarlos. Entonces nos encontramos con el concepto de **modularidad**.

La modularidad es la forma en que dividimos nuestro programa en tareas más pequeñas, al igual que un automóvil puede dividirse en partes más pequeñas, como chasis, ruedas, motor, asientos, etcétera, pero que, de alguna manera, todas se concatenan y hacen una unidad que tiene un propósito más general.

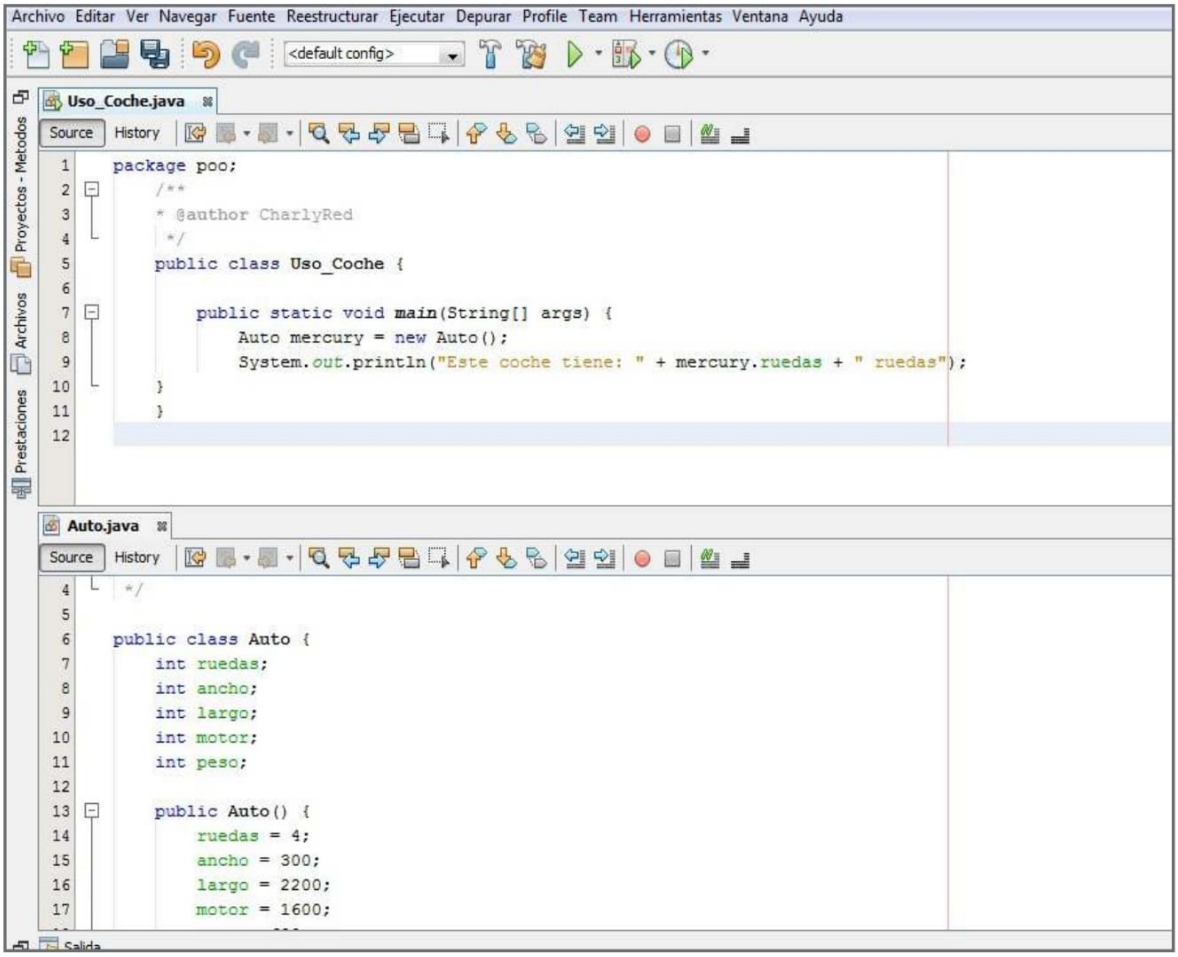


Figura 5. Observemos que se han ubicado las dos clases de manera paralela, de tal forma que podemos trabajar en conjunto y ver lo que hace una o la otra. Esto se puede realizar en forma vertical u horizontal.

En el ejemplo del auto, podemos ver que sus elementos se fusionan, ya que por separado serían inertes o, si no existieran, el programa comenzaría a fallar. Ahora bien, cuando en un programa tenemos varias clases, siempre una de ellas será la clase principal, es decir, la que provocará la ejecución del programa. Esta clase principal es aquella que contiene al método **main**, entonces, siempre existirá en un programa hecho en Java una clase que contenga a este método.

Encapsulamiento

Encapsular consiste en esconder los atributos de una manera que, tan solo con algunos métodos especiales, podamos verlos desde el lugar que los invoquemos. El uso de este concepto en el paradigma a objetos es sinónimo de seguridad. La encapsulación es muy conveniente y nos permite colocar en funcionamiento nuestro objeto en cualquier tipo de sistema, de una manera modular y escalable.

Existen ciertos modelos o reglas para poder encapsular:

Abierto	Hace que el miembro de la clase pueda ser accedido desde el exterior de una clase y desde cualquier parte del programa.
Protegido	Solo es accesible desde la clase y desde las clases que heredan (a cualquier nivel).
Cerrado	Solo es accesible desde las clases.

Si tenemos un programa que está compuesto por más de dos clases, necesitamos imperiosamente usar el encapsulamiento. Este concepto aparece cuando trabajamos con una cantidad de clases, y cada una de ellas posee dedicación a ciertas funcionalidades dentro del programa. Debería ser posible invocar las funcionalidades de una clase principal, pero no siempre esto sucede, ya sea por cuestiones de seguridad o por preferencias del programador, que puede requerir que determinadas acciones solo se puedan manejar desde su propio módulo, a esto se le llama **encapsulamiento**.

¿Cómo hacemos para implementarlo? Usemos el mismo ejercicio de la clase **Uso_Coche**:

```
mercury.ruedas =3;
```

Si ejecutamos el código anterior, veremos que dará como resultado que el auto posee 3 ruedas. Entonces, ¿para qué definimos en la clase **Auto** los atributos si podemos cambiarlos desde cualquier sección?

En este momento, podemos decir que existe información a la que llamaremos **sensible**, que deberá ser encapsulada. En el ejemplo que hemos desarrollado, las ruedas solo se pueden modificar en la clase a la que pertenecen o donde fueron construidas, para ello necesitamos usar los **modificadores de acceso**, en nuestro caso colocaremos antes de la variable el término **private**. Veamos ahora qué sucede si modificamos nuestro código y, a todos los atributos, les agregamos **private**:

```
public class Auto {
private int ruedas;
private int ancho;
private int largo;
private int motor;
private int peso;
…
}
```

Cuando vayamos a la clase **Uso_Coche**, veremos que el código nos mostrará un error, es decir, no será posible acceder al objeto que llamamos. Esto se debe a que hemos encapsulado el dato en la clase **Auto** y, por lo tanto, no será visible desde otra clase.

Reutilización de código

Una de las formas más útiles para generar código que sea reutilizable es que cada método se limite a realizar una tarea bien definida, y su nombre exprese esa tarea con efectividad, por ejemplo, **sumarCuadrados()**. Un método que lleva una tarea específica es más fácil de probar y depurar que uno más grande y con múltiples tareas. Una manera de darse cuenta de que no estamos modularizando correctamente es verificar que no podemos escribir un nombre conciso para un módulo o método.

La pregunta ahora es: ¿cómo podemos hacer para acceder a estas propiedades si permanecen invisibles? Necesitamos algunos elementos adicionales, en este caso, los métodos **getter** y **setter**.

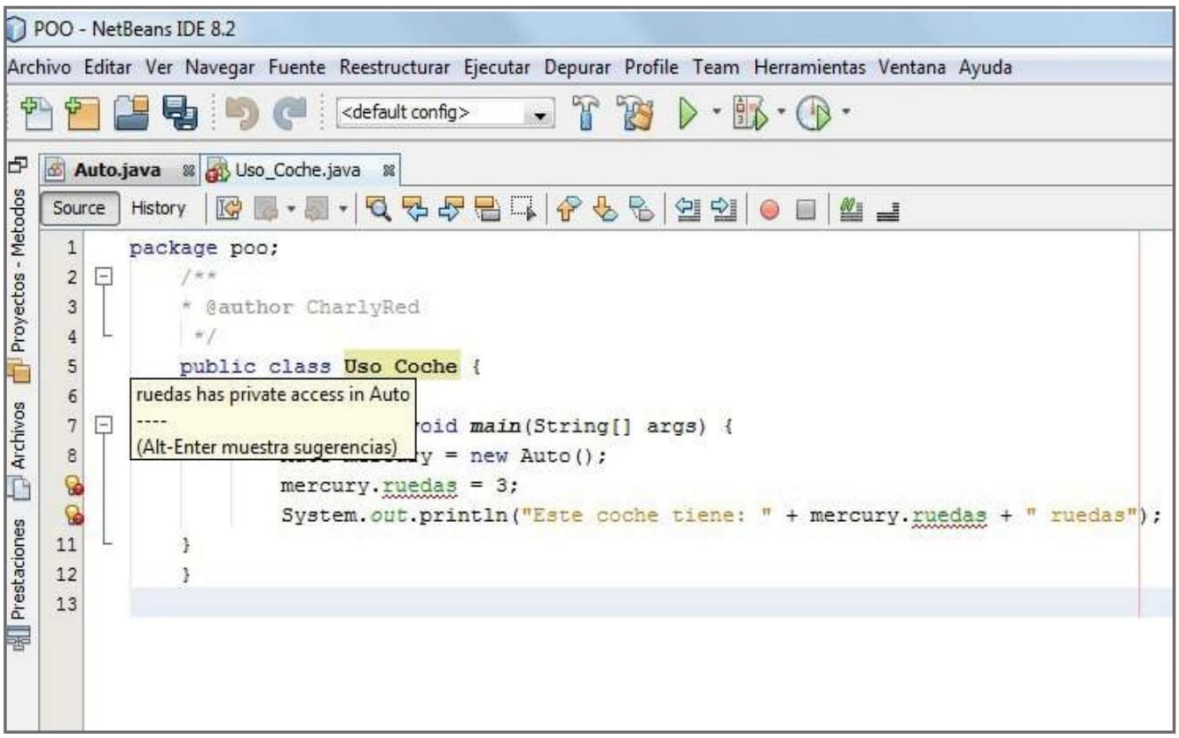


Figura 6. En el código de la imagen podemos observar que se presentan errores de acceso a los atributos de un objeto (ruedas).

Getter y setter

Para ver los datos que han sido encapsulados, debemos utilizar los métodos de acceso, esto es una forma de conectar los elementos que tenemos en las distintas clases.

Programación funcional en Java

Al igual que Scala o Haskell, con la incorporación de las expresiones Lambda, Java 8 hizo por fin el ingreso a la programación funcional. Por ejemplo, se trata de una manera bastante eficiente de agrupar una colección según el valor de un campo de una clase en particular.

Los métodos **getter** y **setter** son estructuras que sirven para obtener o almacenar algún valor de un atributo de una clase. Un **getter** (de *get*: 'obtener') indica que vamos a tomar un valor de un atributo, también se lo llama **método captador**.La sintaxis de un método de este tipo es la siguiente:

```
public [TipoDato] getNombreAtributo(){
     return nombreatributo;
}
```

Podemos notar que dentro de su instrucción hemos ubicado un **return**; esto es adecuado para que retorne un valor, en este caso el que corresponde a la variable.

En cuanto al método **setter** (de : 'poner'), está indicado para guardar lo obtenido anteriormente, también se lo denomina **definidor**. La sintaxis de un **setter** es la siguiente:

```
Public void setNombreAtributo([TipoDato] nombreatributo){
     this.nombreatributo = nombreatributo;
}
```

Podemos notar que, en la sintaxis del **setter**, se encuentra un **void**, lo que implica que no devolverá un valor.

La importancia de estos métodos radica en que podemos utilizar los valores de algún atributo sin la necesidad de hacerlo en la clase desde donde se originó.

Paradigmas de programación

En la programación existen varios paradigmas, por ejemplo, hay lenguajes más antiguos que estaban orientados a procedimientos (Fortran, Cobol, Basic) y también lenguajes más modernos que están orientados a objetos (C++, Java, Visual.Net). La diferencia se encuentra en la forma de interpretar la manera de programar. En los primeros, era bastante difícil corregir errores y reutilizar el código, mientras que en los segundos es más sencillo y pragmático, pues utilizamos el mismo comportamiento que tienen los objetos en la vida real.

Veamos ahora un ejemplo para entender la forma en que se implementan estos métodos siguiendo el trabajo que hemos realizado con la clase **Auto**:

```
public class Auto {
private int ruedas;
private int ancho;
   …

public Auto() {//Constructor de la clase
        ruedas = 4;
        ancho = 300;
        …
    }
    public String cantidad_ruedas(){//Getter
   return "la cantidad de ruedas es: " + ruedas;
    }
}
```

En el código anterior creamos el método **cantidad_ruedas()**, del tipo **String**, pues queremos que aparezca una cadena de caracteres en el resultado. Luego utilizamos la palabra clave **return** para que retorne lo que le encomendamos y, finalmente, concatenamos con la variable **ruedas**.

Ahora veamos cómo funcionará para que retorne un atributo en la clase donde ejecutaremos el programa, es decir, **Uso_Coche**:

```
System.out.println("Ruedas: " + mercury.cantidad_ruedas());
```

En este código hacemos una salida y, dentro de ella, concatenamos una cadena con el nombre del objeto previamente instanciado; podemos notar que, al poner el punto, aparecerán los métodos. En nuestro caso, solo debemos hacer clic en **cantidad_ruedas()**, que habíamos creado en el ejemplo anterior. Cuando ejecutamos el programa, veremos el siguiente resultado: **Ruedas: la cantidad de ruedas es: 4**

Ahora bien, agregaremos más atributos a nuestro programa, esta vez no usaremos el modificador de acceso **private**:

```
String color;
int peso_Total;
boolean asientosCuero, aireAcond;
```

Ahora crearemos un método **setter** para establecer una característica especial para un auto específico:

```
public void estableceColor(){ //Método setter
        color = "gris";
    }
```

En este caso hemos puesto el color gris. Para continuar, crearemos un método **getter** para obtener este valor en la clase principal:

```
public String unColor(){//Método getter
return "el color del coche es: " + color;
}
```

Ahora tendremos que instanciar un nuevo objeto en la clase principal y lo hacemos de la siguiente forma:

```
public static void main(String[] args) {
Auto miAuto = new Auto();
miAuto.estableceColor();
System.out.println(miAuto.unColor());

}
```

En el código anterior hemos instanciado a la clase en un objeto llamado **miAuto**.

Luego traemos el método que creamos en la clase **Auto**, escribimos **miAuto** y aparece la cantidad de métodos establecidos, entre ellos elegimos **estableceColor()**.

Luego, si queremos ver el resultado en pantalla, solo tenemos que poner el nombre del objeto y elegimos el método getter **unColor()**, con lo que obtendremos lo siguiente: **el color del coche es: gris**

Ahora bien, ¿qué pasaría si no escribiéramos la siguiente línea?

```
miAuto.estableceColor();
```

Probemos ejecutando el programa sin la línea anterior, veremos que nos arroja lo siguiente: **el color del coche es: null**.

Esto sucede porque no estamos utilizando el método **estableceColor()**, por lo tanto, aparecerá vacío, esto es porque no estamos invocando a un método existente.

Pasar parámetros

Una de las situaciones que mejorará la implementación de los métodos ya mencionados es el uso de parámetros. En el ejemplo de los getter y setter, vimos cómo hacíamos para pasar un aspecto del objeto, pero esto presentaba una complicación cuando necesitábamos pasar alguna otra propiedad, y nos dejaba limitados.

Ante esto, una de las soluciones es que, a la vez que llamamos alguna propiedad por un método, le pasamos junto con la llamada un parámetro o valor, de esta forma el método tomará ese valor y operará con él. Volvamos a los códigos para ver cómo implementar el paso de parámetros.

En la clase **Auto**:

```
public void estableceColor(Stringcolor_auto){
        color = color_auto;

    }
```

Agregamos al setter un argumento (**color_auto**) de tipo **String**, luego dentro del atributo **color** se iguala a la variable del parámetro.

Debemos acotar que se puede pasar más de un argumento en los parámetros de los métodos.

Constructor vacío

Las variables de instancia pueden declararse como public o como private y pueden pertenecer a cualquiera de los tipos de datos primitivos de Java. Los programadores suelen declarar variables de instancias private para probar, depurar y modificar sistemas. Entonces, si una subclase puede acceder a una instancia de variable private de su superclase, las que heredan de esa subclase también lo pueden hacer. Esto provocaría la privacidad de información, cosa que no está bien.

Ahora la clase **Uso_Coche**:

```
public static void main(String[] args) {
Auto miAuto = new Auto();
miAuto.estableceColor("Rojo");
System.out.println(miAuto.unColor());

}
```

Una vez instanciada la clase lo que hacemos es escribir **miAuto**, observamos que inmediatamente aparecerá la serie de elementos antes construidos. Elegiremos el método **estableceColor()**, dentro de él escribimos un **String** del color que deseemos.

Si ejecutamos el código deberíamos ver lo siguiente: **el color del coche es: Rojo**

Inicialmente habíamos establecido los atributos de las últimas tres variables sin una prohibición de acceso, esto atenta contra la regla principal de la encapsulación, veamos qué pasaría si escribimos la siguiente línea en la clase **Uso_Coche**:

```
miAuto.color= "verde";
```

Ahora veamos qué aparece cuando ejecutamos el código **el color del coche es: verde**

Esto no resulta una buena práctica, pues no nos serviría de nada todo lo creado antes. Nos encontramos ante un error conceptual, pues nunca debemos manipular directamente la propiedad de un objeto utilizando su propia instancia.

Referencias a objetos de la misma clase

Si un objeto de una clase hace referencia a otro objeto de la misma clase, el primer objeto puede acceder a todos los datos y métodos del segundo, no importa que alguno de estos sean private.

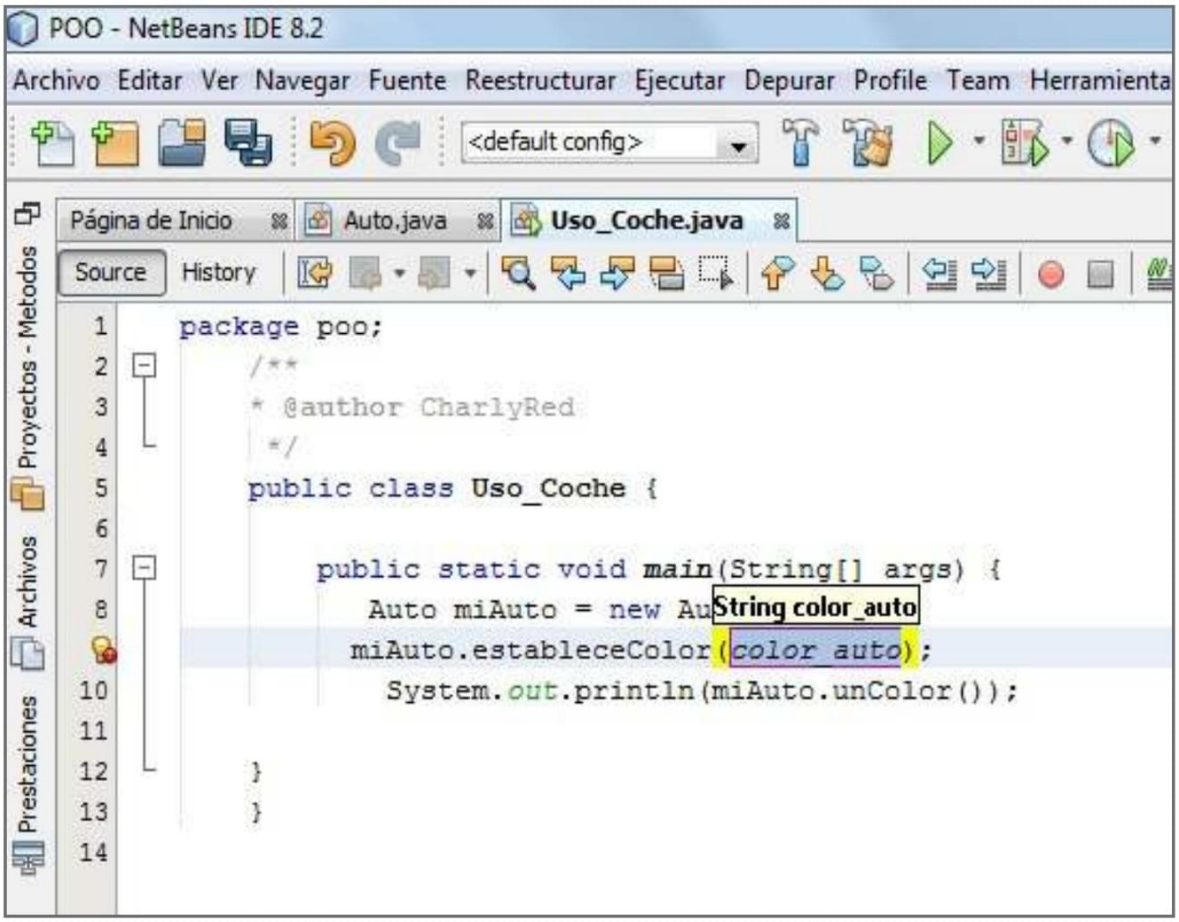


Figura 7. En la figura vemos que el IDE sugiere, dentro del elemento invocado, que el valor es de tipo String, esto se debe a que habíamos pasado el parámetro en el setter.

Ahora ponemos el modificador de acceso **private** en la siguiente variable:

```
private String color;
```

Esto provocara que la clase **Uso_Coche** nos marque un error de privacidad, por lo tanto, no podemos usar el atributo **color**:

```
miAuto.color= "verde";
```

Aquí vemos las consecuencias directas de lo que sucede si no encapsulamos nuestras variables.

Ahora crearemos un método setter para los asientos, para que podamos elegir si serán o no de cuero; cuando tengamos este tipo de disyuntivas, usaremos el tipo boolean:

```
public void estableceAsientos(StringasientosCuero){
//setter
if(asientosCuero==”si”){
this.asientosCuero = true;
        }else{
this.asientosCuero=false;
        }

    }
```

En este caso crearemos una estructura de decisión para resolver si el coche tendrá o no asientos de cuero. Para ello, utilizaremos la palabra reservada **this**:

```
public String tipoAsientos(){ //Getter
if(asientosCuero.equalsIgnoreCase(“si”)){
return “El coche lleva asientos de cuero”;
        }else{
return “el coche lleva asientos normales”;
        }
    }
```

Antes de continuar, debemos establecer que dentro de un **if** nunca se debe comparar el **String** con el signo **==**, en vez de ello, debemos hacerlo con el método **equals**.

Ahora iremos a la clase **uso_Coche** para ver si funciona lo que acabamos de construir, debemos establecer el método **setter estableceAsientos** y, dentro de él, ponemos un valor que determinará si lleva o no asientos de cuero:

```
miAuto.estableceAsientos(“si”);
System.out.println(miAuto.tipoAsientos());
    }
```

Al ejecutar el código anterior, vemos que funciona a la perfección, entregando el siguiente resultado: **El coche lleva asientos de cuero.**

Al utilizar el IDE, existe una forma automática de construir los getter y setter de todos los elementos creados; para lograrlo hacemos clic derecho en la línea donde vamos a incrustar los métodos y elegimos la opción **Insertar**, luego hacemos clic sobre **getter y setter**.

Una vez elegida la opción, veremos todas las variables disponibles, solo debemos marcar aquellas para las que construiremos el **getter** o el **setter**.

Una vez creados los métodos que necesitemos, el IDE nos ubicará dentro los elementos adecuados para seguir programando, esto genera una enorme ventaja ya que no es necesario escribir mucho código para implementarlo. La forma en que se verán es como muestra el siguiente ejemplo:

```
public int getRuedas() {
return ruedas;
    }
```

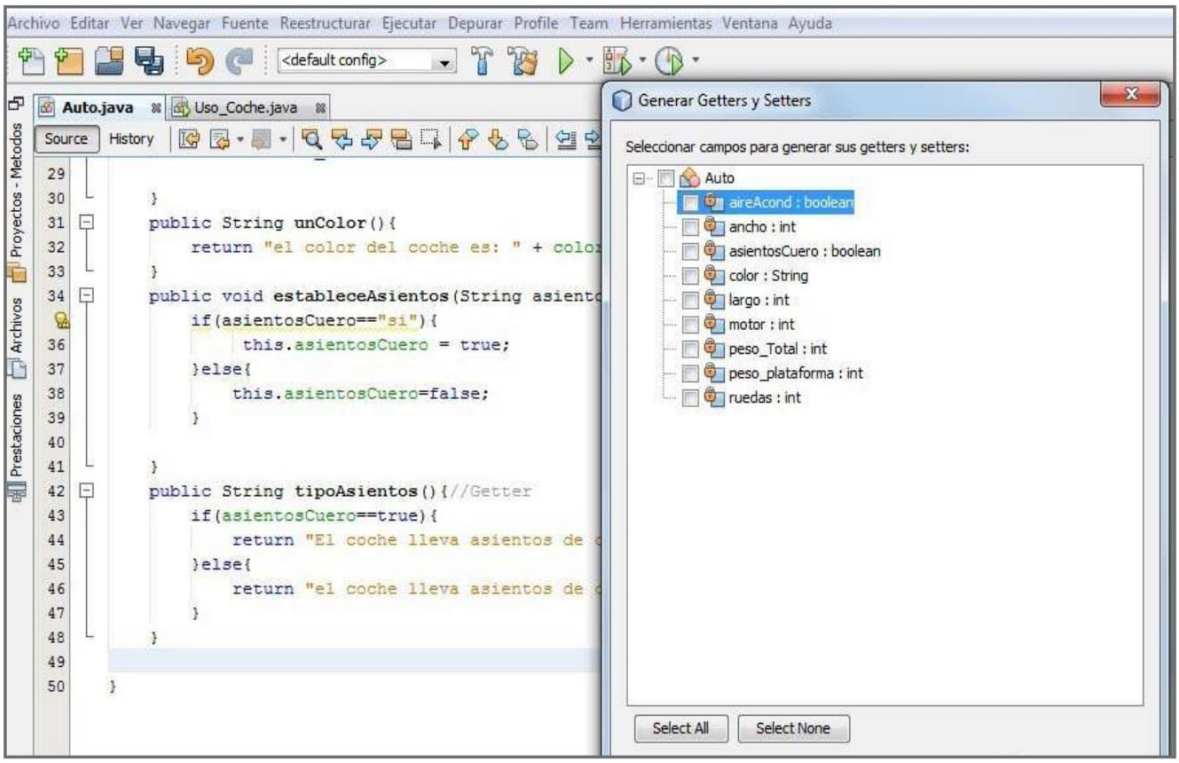


Figura 8. En la figura podemos ver cómo se pueden seleccionar los objetos que creamos convenientes y, a partir de eso, crear los getter y setter.

```
public void setRuedas(int ruedas) {
this.ruedas = ruedas;
    }
```

Se realizará lo mismo con cada una de las variables creadas. Por ejemplo, si tenemos 10 variables creadas, se crearán 10 getter y 10 setter, 20 métodos en total, todo en un segundo.

Al crear nuestros métodos **getter** y **setter**, puede que existan algunos que nunca usemos por lo que estaremos contradiciendo el ahorro de memoria, sin embargo, si tenemos en cuenta el ahorro de escritura de código, podemos considerarlo como una buena práctica.

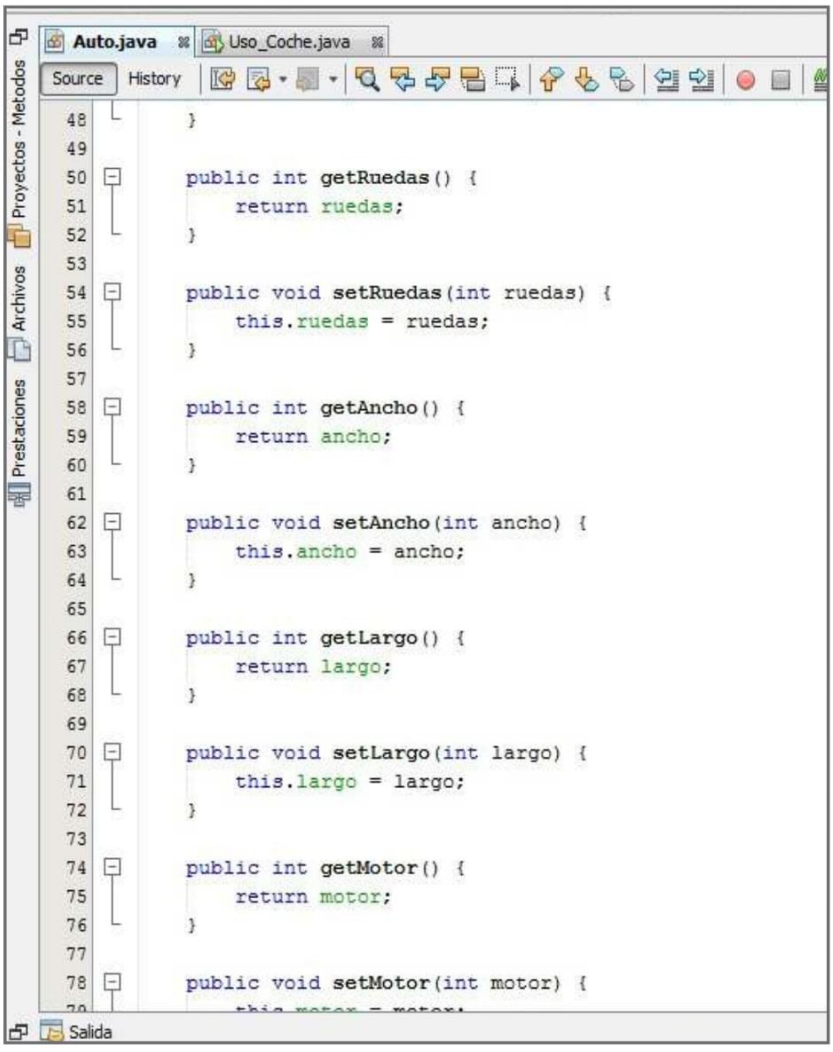


Figura 9. Podemos notar que el IDE ha ubicado, por cada elemento creado (variables), su respectivo getter y setter, con la codificación necesaria para continuar.

CONSTRUCCIÓN DE OBJETOS

Trabajaremos ahora con la construcción de objetos; en el ejemplo de la clase **Auto**, vamos a crear otro método, pero esta vez haremos que inicialice un **setter:**

```
public void configuraAire (String aireAcond){ //Setter
    if(aireAcond.equalsIgnoreCase("si")){
    this.aireAcond = true;
    }else{
    this.aireAcond = false;
}
}
```

Ahora que ya configuramos el estado del aire acondicionado del auto, haremos lo mismo con el **getter**:

```
public String tieneAireAcond(){ //Getter
    if(aireAcond ==true){
    return "El coche tiene aire acondicionado";
    }else{
    return "El coche no tiene aire acondicionado";
     }
}
}
```

Visibilidad de métodos y atributos

Cuando exponemos y, en contraposición, escondemos elementos, estamos haciendo referencia al nivel de visibilidad de los métodos y los atributos de las clases. Los atributos y los métodos que definamos como **public** serán visibles desde cualquier clase; por el contrario, aquellos a los que definamos como **private** estarán encapsulados y, por ende, solo podrán ser invocados y manipulados dentro de la misma clase.

A continuación, crearemos un método que sea **getter** y **setter** a la vez; aunque no sea una buena práctica este tipo de implementaciones, lo haremos para aprender un poco más acerca de estos conceptos. Básicamente, lo que logrará es hacer las dos cosas (**getter** y **setter**), pero no a la vez, sino dependiendo de lo que le pidamos.

Para continuar con el ejemplo del **Auto**, incorporaremos el peso total del vehículo, sin embargo no sabemos si el auto va tener aire o asientos de cuero, por lo tanto, podría tener peso extra. Veamos el código:

```
public String pesoAuto(){
//Setter y Getter a la vez
    int peso_carroceria = 500;
    peso_Total = peso_plataforma + peso_carroceria;
    if(asientosCuero==true){
    peso_Total = peso_Total + 50;
//Estimamos este peso extra de los asientos de cuero
    }
    if(aireAcond ==true){
    peso_Total = peso_Total + 20;
//Estimamos este peso
    }
    return "el peso del coche es " + peso_Total;
}
```

Entonces, ¿por qué sería un método **getter** y **setter** a la vez? Por un lado establece, con una estructura de decisión, los pesos adicionales que tendrá el auto y, por otro lado, los obtiene a través del **return**.

La anotación @Override

Debemos declarar métodos sobre escritos cuando se va a asignar en tiempo de compilación y se hayan definido las firmas correspondientes al método. Aunque es mejor encontrar errores en tiempo de compilación que en ejecución.
La anotación @Override simplemente se utiliza para forzar al compilador a comprobar en tiempo de compilación que estamos sobrescribiendo correctamente un método, y de este modo, evitar errores en tiempo de ejecución.

Ahora declararemos un **getter** con el precio del coche, nuevamente tendremos que considerar que no es lo mismo que tenga aire o que cuente con asientos de cuero. Veamos el código:

```
  public int precioCoche(){ //Getter
      int precioFinal= 15000; //Precio estimado en dólares
      if(asientosCuero==true){
      precioFinal += 2000; //Precio estimado
      }
      if(aireAcond ==true){
      precioFinal += 1500; //Precio estimado
      }
      return precioFinal;
  }
  }
```

Ahora probamos este ejemplo en la clase **Uso_Coche**, recordemos que esta contiene el método **main** que llamará a los métodos creados:

```
  miAuto.estableceAsientos("no");
  System.out.println(miAuto.tipoAsientos());
  miAuto.configuraAire("si");
  System.out.println(miAuto.TieneAireAcond());
  System.out.println(miAuto.pesoAuto());
  System.out.println("el precio final del auto es: " + mi-
Auto.precioCoche());
```

Superpoblación de clases

En Java, todas las clases forman parte de una jerarquía de clases. Es bien sabido en la industria del software que se debe evitar la proliferación de clases, lo que provoca problemas de índole administrativa y obstaculiza la reutilización de software. Evitemos complejizar las jerarquías.

Ahora ejecutaremos el programa, para ello debemos probar cuando tiene asientos de cuero o cuando tiene aire, y cuando no presenta estas características. Hay que recordar que el precio establecido para este auto es de 15000 dólares, según ello, debe aumentar 2000 para los asientos de cuero y 1500 para el aire.

De la misma manera, debemos probar el peso, recordando que los asientos de cuero pesan más y si lleva aire, también.

Al ejecutar el programa podemos ver lo siguiente en la consola:

```
el color del coche es: Rojo
El coche lleva asientos de cuero
El coche tiene aire acondicionado
el peso del coche es 1170
el precio final del auto es: 3500
```

Está claro que el programa resulta un poco incómodo, pues debemos cambiar los valores de manera manual, la forma correcta es usando un cuadro de diálogo que nos pida los valores que deseemos para el auto. Esto puede solucionarse gracias al uso de la clase **Swing**, que conoceremos más adelante.

CLASE SWING

La clase **Swing** nos permite trabajar con elementos de entrada de datos, como caja de datos y mensajes. Para implementar esta clase debemos importarla:

```
import javax.swing.*;
```

De este paquete vamos a utilizar un elemento llamado **JOptionPane**:

```
Auto miAuto = new Auto();
            System.out.println(JOptionPane.
showInputDialog("Ingresa el color del auto: "));
```

```
            miAuto.estableceColor(miAuto.unColor());
          miAuto.estableceAsientos(JOptionPane.
showInputDialog("¿Tiene asientos de cuero?"));
  System.out.println(miAuto.tipoAsientos());
          miAuto.configuraAire(JOptionPane.
showInputDialog("¿Tiene aire acondicionado?"));
  System.out.println(miAuto.TieneAireAcond());
  System.out.println(miAuto.pesoAuto());
  System.out.println("el precio final del auto es: " + mi-
Auto.precioCoche());

      }
```

Implementamos el **JOptionPane** cuyo método **showInputDialog** nos permite efectuar la entrada de datos a través de un cuadro de diálogo. Por otro lado, las siguientes instrucciones permiten que los resultados puedan verse en la consola:

```
  miAuto.estableceColor(miAuto.unColor());
  System.out.println(miAuto.tipoAsientos());
  System.out.println(miAuto.TieneAireAcond());
```

Cuando respondemos que sí a alguno de los elementos extras que tendrá el auto, estos se sumarán al precio total y a su peso, mientras que, si respondemos en forma negativa, dejará todo como se estableció de manera predeterminada. Al ejecutarlo podemos verificar que el programa funciona a la perfección.

Java FX

Las bibliotecas de Java FX han venido en las últimas versiones a reemplazar a Java Swing en lo que se refiere a la GUI. Con el advenimiento de nuevos patrones de diseño cada vez más versátiles, como MVC, los programadores están viendo con mejores ojos esta nueva alternativa que ofrece Oracle.

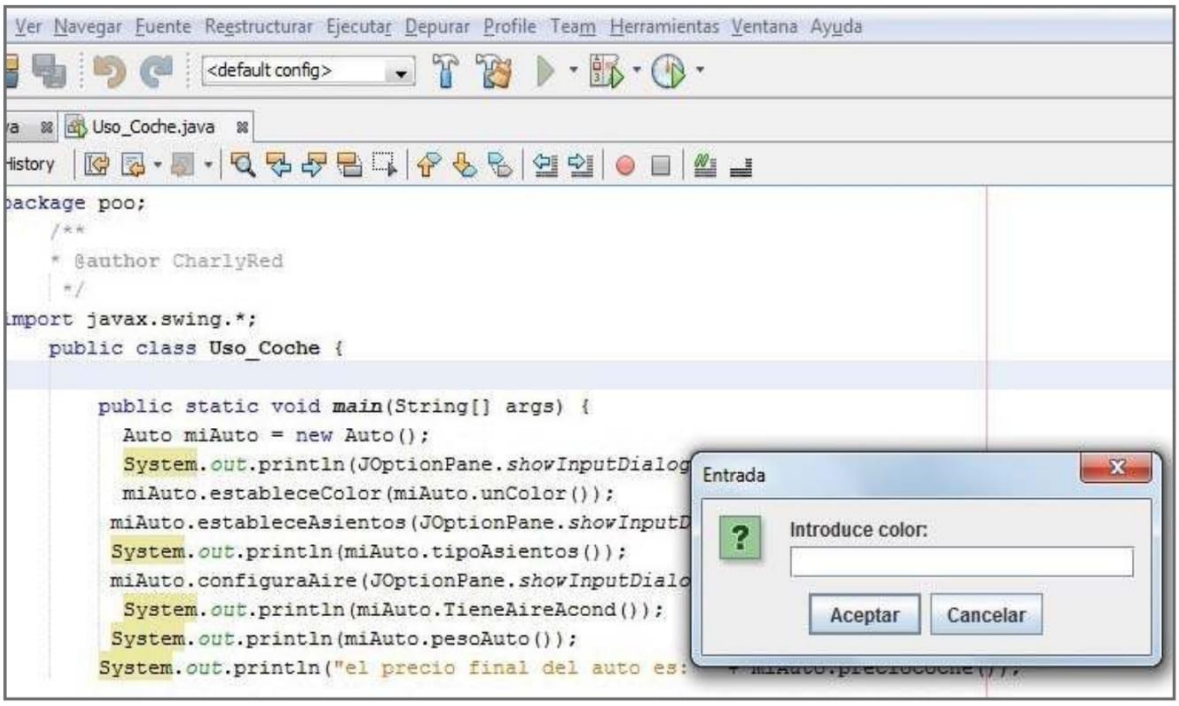


Figura 10. En esta imagen se muestra como Java Swing nos permite cargar información a través de un cuadro de diálogo.

RESUMEN CAPÍTULO 02

La programación orientada a objetos es uno de los paradigmas más utilizados hoy en día en el mundo de la programación, en este capítulo conocimos varias de sus principales características, aprendimos a construir una clase desde cero y también a crear los objetos que esta contiene, a esto lo llamamos instanciar una clase. Aprendimos a dividir estratégicamente nuestros programas mediante los métodos, revisamos la encapsulación de los atributos, también pasamos parámetros e invocamos a los métodos creados a través de los getter y setter.

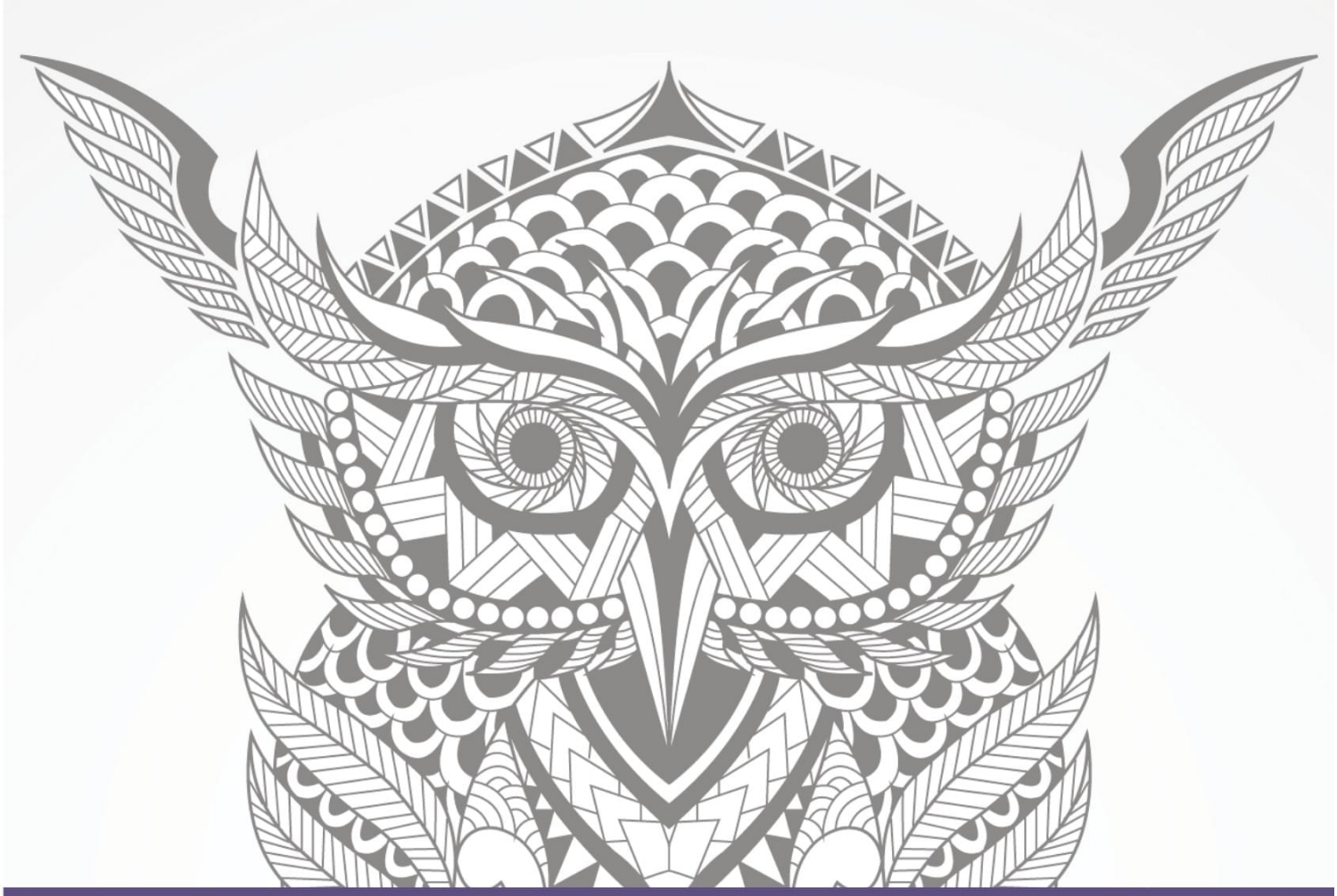
Actividades 02

Test de Autoevaluación

1. ¿Cómo se crea una clase?
2. ¿De qué forma se puede instanciar un objeto de clase?
3. Nombre los elementos fundamentales de la programación orientada a objetos.
4. ¿Qué función cumple el método constructor dentro de las clases?
5. ¿Cuál es el sentido de modularizar los programas?
6. ¿Qué significa el encapsulamiento?
7. ¿Qué es un método getter?
8. ¿Cómo pasamos parámetros en los métodos?
9. ¿Qué es un constructor de objeto?
10. ¿Qué es la clase Swing?

Ejercicios prácticos

1. Cree una clase e instancie los objetos que necesite para un programa en el que se calcularán los sueldos de los empleados, teniendo en cuenta las horas extra y los impuestos que se les descuenta.
2. Considerando el ejercicio anterior, debe encapsular las variables. Recuerde el motivo por el que se realiza esta acción.
3. Provoque los métodos getter y setter necesarios para invocar a las acciones realizadas en cada parte del programa.
4. Realice un programa calculadora, en el que dada cierta cantidad de números, se resuelvan operaciones básicas. Recuerde aplicar lo aprendido en este capítulo.



Elementos esenciales de la POO

En este capítulo revisaremos en forma más detallada algunos de los elementos que componen la POO. Veremos la colaboración de clases, aprenderemos sobre la herencia y, además, profundizaremos en el polimorfismo, es decir, lograremos que los métodos enviados a los distintos objetos sean interpretados de una manera inteligente.

03

COLABORACIÓN DE CLASES

Cuando dos o más clases se comunican entre sí y una de ellas intenta enviar un mensaje a través de los métodos, estamos frente a la **colaboración de clases**. Así como en la vida real es muy difícil que los objetos se encuentren aislados, en la POO también deben relacionarse.

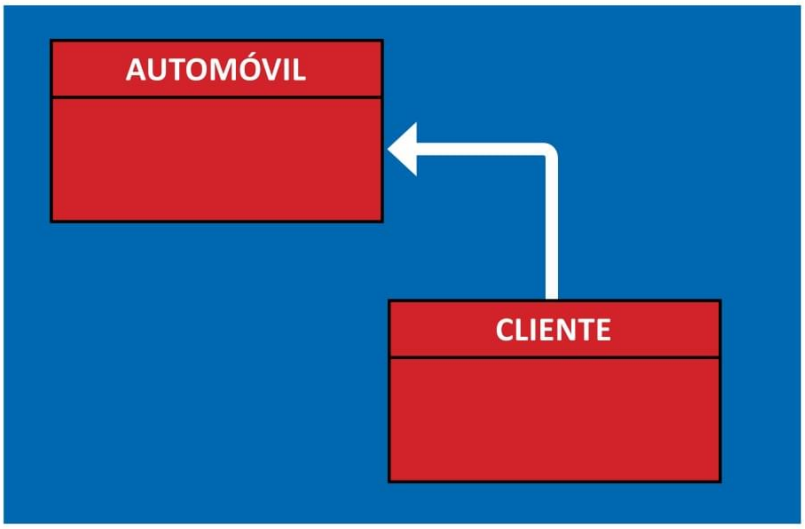


Figura 1. En el modelo se muestra cómo una clase intenta comunicarse con otra, esto se produce gracias a los métodos.

Ahora veremos un ejemplo que muestra cómo manejaríamos una caja menor, pues nos servirá para entender cómo funciona la colaboración de clases:

```
package cajamenor;

public class IngresosEgresos {
//Atributos
 private String nombre;
    private int caja;

    public IngresosEgresos(String nom) {
        nombre=nom;
        caja=0;
    }

    public void ingresar(int c) {
```

```
            caja=caja+c;
        }

        public void pagar(int c) {
            caja=caja-c;
        }

        public int retornarMonto() {
            return caja;
        }

        public void imprimirIngresos() {
            System.out.println(nombre +" pagó "+ caja);
        }
        public void imprimirEgresos() {
            System.out.println("A " + nombre +" se le pagó "+
-(caja));
        }
    }
```

En este código creamos la clase y las variables privadas o atributos, después el constructor de la clase, un método que pasa parámetros; luego creamos los getter y setter, finalmente el método **imprimirIngresos()** e **imprimirEgresos()** que devolverá los valores antes configurados.

Ahora debemos crear la clase **CajaMenor**, que contendrá la siguiente codificación:

```
    package cajamenor;
    public class CajaMenor {

        private IngresosEgresos cliente1, cliente2, cliente3,
cliente4,prov1,prov2,pago1;

        public CajaMenor() {
            cliente1=new IngresosEgresos("Juan");
            cliente2=new IngresosEgresos("Ana");
```

```
            cliente3=new IngresosEgresos("José");
            cliente4=new IngresosEgresos("Panadería");
            prov1=new IngresosEgresos("Lácteos");
            prov2=new IngresosEgresos("Bebidas");
            pago1 = new IngresosEgresos("Banco");

        }

        public void operaciones() {
            cliente1.ingresar(500);
            cliente2.ingresar(650);
            cliente3.ingresar(450);
            prov1.pagar(200);
            prov2.pagar(150);
            pago1.pagar(800);
        }

        public void Totales()
        {
      int total = cliente1.retornarMonto() + cliente2.
retornarMonto()+ cliente3.retornarMonto()+ prov1.
retornarMonto()+ prov2.retornarMonto()+ pago1.
retornarMonto();

    System.out.println ("El total de dinero en la caja es: "
+ total);
            cliente1.imprimirIngresos();
            cliente2.imprimirIngresos();
            cliente3.imprimirIngresos();
            prov1.imprimirEgresos();
            prov2.imprimirEgresos();
            pago1.imprimirEgresos();
        }

        public static void main(String[] ar) {
            CajaMenor caja=new CajaMenor();
```

```
            caja.operaciones();
            caja.totales();
        }
    }
```

Hasta aquí hemos creado e instanciado la clase **CajaMenor**, también definimos los atributos; mediante el constructor, le daremos el nombre tanto a los clientes como a los proveedores y el pago, ya sea el pago predeterminado o mediante el paso de parámetros.

A continuación, tenemos los métodos **operar()** en donde cargaremos el dinero que entra o sale de acuerdo al nombre instanciado.

En el método **Totales()** se imprimirá la información previamente calculada, por otra parte, en el método **main** haremos la llamada de todos los métodos creados tanto en esta clase como en la predecesora. Cuando ejecutemos el programa, deberá mostrar lo siguiente:

El total de dinero en la caja es: 450
Juan pagó 500
Ana pagó 650
José pagó 450
A Lácteos se le pagó 200
A bebidas se le pagó 150
A Banco se le pagó 800

Clases anidadas y métodos privados

Cuando tenemos tanto clases anidadas como internas, el miembro privado y el código que lo usa pueden estar en la misma clase, y al mismo tiempo, también están en diferentes clases. Como ejemplo, si tenemos dos clases anidadas en una clase de nivel superior, entonces el código en una de las clases anidadas puede ver un miembro privado de la otra clase anidada.

CONSTRUCTOR DE OBJETOS

Un **constructor de objetos** es un método que se llama automáticamente cuando se declara un objeto de esa clase y cuya función es inicializar el objeto. De hecho va a validar los valores que los objetos contendrán.

Recordemos la sintaxis de la construcción de un objeto:

```
// creamos un objeto a través del constructor por defecto
Nombre nom = new Nombre();
```

Como ya hemos visto, trabajaremos con un ejemplo al cuál le iremos agregando elementos teóricos conforme vayamos avanzando con el código.

Cambiaremos un poco la temática de los ejemplos para que no se torne monótona la forma de nuestro aprendizaje. Esta vez trabajaremos con una clase trabajador o empleado (para los efectos, da lo mismo).

El empleado va tener ciertos atributos (nombre, sueldo, fecha de ingreso) y un método al que llamaremos **aumentarSueldo()**.

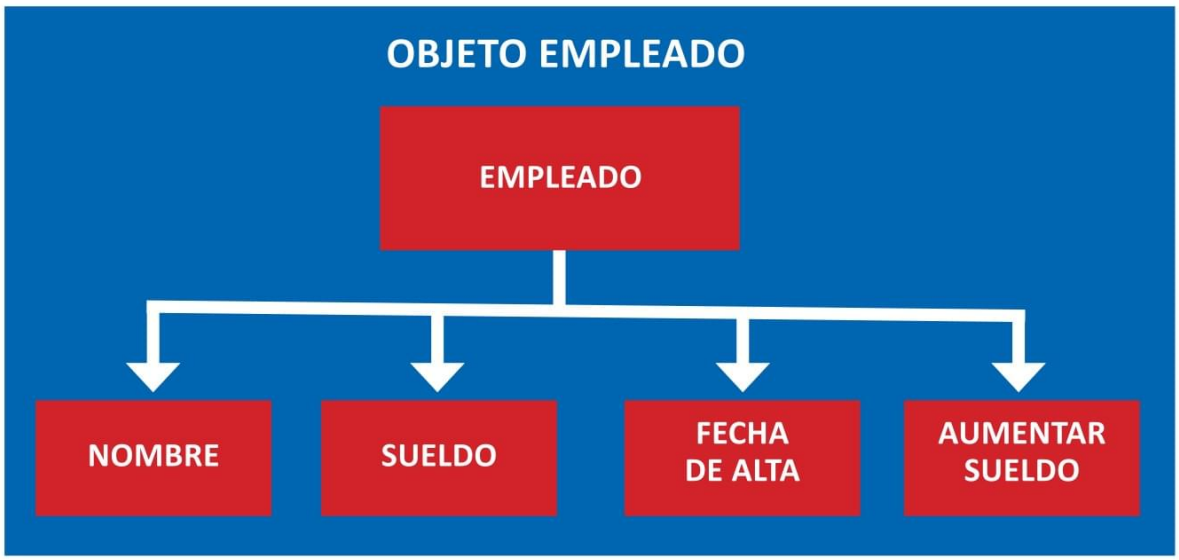


Figura 2. En el gráfico se muestra un objeto llamado empleado, los atributos que llevará y el método.

Nos remitiremos al IDE para crear una clase **Empleado**, pero esta vez trabajaremos en una misma clase. Esta clase contendrá varias clases adentro y, además, sabemos que esta clase tendrá el método **main**:

```
package empleado;
import java.util.Date; //importamos este paquete
public class UsoEmpleado {
    public static void main(String[] args) {

    }
 }
class Empleado{
    public Empleado(String nom, double sue, int año,int
mes, int dia){ //constructor de clase

}
// variables de clase
      private String nombre;
      private double sueldo;
      private Date altaIngreso;
}
```

Tengamos en cuenta que la clase **Empleado** no tiene el modificador de acceso **public**, pues cuando convive en una sola estructura debe haber una sola clase pública, de la misma manera que solo una de las clases debe contener un **main**, que es el que ejecuta la clase pública y todo lo que llamará según lo que se haya programado dentro.

El constructor de la clase va a recibir parámetros y debe llevar el mismo nombre de la clase que va a construir.

Tipos de datos objeto

Además de los tipos de datos primitivos (**int**, **char**, **boolean**) que conviven en la librería **java.lang**, a partir de la versión 5 de Java existen otros tipos de datos a los que llamaremos **tipos de objeto**. Estos en principio se escriben con mayúsculas, tenemos: **Boolean**, **Char**, **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **BigInteger**. En los tipos de datos objeto podemos envolver su respectivo tipo de dato primitivo para realizar las operaciones directamente sobre un objeto. De igual manera, de un tipo de dato objeto, podremos obtener su valor como tipo de dato primitivo.

En este momento debemos considerar a las **variables de clase** que, a diferencia de las **variables de instancia**, son propias de la clase que las contienen y no de las instancias de ellas.

Podemos notar que hemos puesto **private** en los modificadores de acceso (encapsulación) y, además, que **Date** es una variable de tipo de datos **objeto**. Otra de las cosas que notamos es que, al poner **Date**, nos aparece subrayado como si tuviese un error, entonces debemos importar puesto que no está dentro del paquete de **java.lang**. Debemos ir a la **API de Java** y verificar cuál es el paquete al que pertenece **Date**, que está dentro de **java.util.***; con esto el marcado del error desaparecerá.

Gregorian calendar

El **Gregorian calendar** es una subclase que hereda de la clase **Calendar** distintos elementos, como los métodos, que podemos encontrar en la API de Java, y que soporta un estándar de formatos de fechas y horas de todo el mundo. Se encuentra dentro del paquete **java.util.GregorianCalendar**. El constructor tiene los siguientes parámetros:

```
GregorianCalendar(int year, int month, int dayOfMonth)
```

Ahora bien, seguiremos con el ejemplo planteado donde trabajaremos dentro del constructor **public Empleado()**, igualaremos las variables de clase con el parámetro que hemos pasado al constructor.

Es un buen momento para aprender a implementar una clase que viene con Java para la construcción de una fecha con el Gregorian calendar, la que también se encuentra dentro de la API de Java.

```
GregorianCalendar calendario = new GregorianCalendar(año,
mes-1, dia);
```

Este código nos indica que hemos creado un constructor, luego hemos instanciado el objeto **calendario** y le hemos pasado los parámetros que antes habíamos pasado. A la variable **mes**, le restamos **1**, pues se indexa desde cero y con esto no nos perdemos tanto al colocar los meses en enteros.

```
  public Empleado(String nom, double sue, int año,int mes,
int dia){
          nombre = nom;
          sueldo = sue;
          GregorianCalendar calendario = new
GregorianCalendar(año, mes-1, dia);
       altaIngreso = calendario.getTime(); //devuelve la
fecha
      }
```

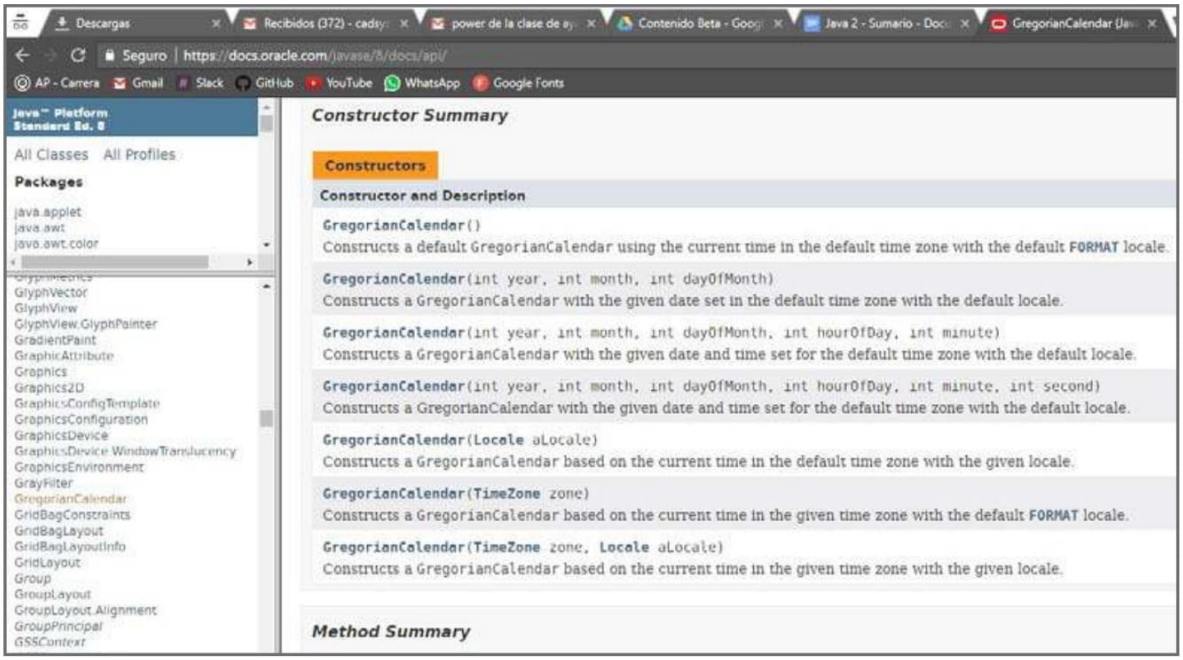


Figura 3. Aquí vemos la página oficial de Oracle en la que se encuentra la API Gregorian calendar. Existen varios constructores, usaremos el segundo en el que podremos personalizar nuestra propia fecha.

Ahora vamos a crear los **getter** que nos devolverán los valores que hemos almacenado:

```
      //GETTER
        public String unNombre(){
            return nombre;
        }
```

```
        public double unSueldo(){
            return sueldo;
        }
        public Date unaFechaAlta(){
            return altaIngreso;
        }
```

A continuación, crearemos un **setter** que nos va subir el sueldo. No llevará **getter** puesto que lo que hace es construir un método y no devolver algo que ya está almacenado:

```
  public void aumentaSueldo(double porcentaje){
         double aumento = (sueldo * porcentaje)/100;
         sueldo += aumento; //operador incremental
        }
```

Al crear este **setter** es obligatorio que lleve un parámetro y, dentro de él, haremos los cálculos respectivos para que se aumente un porcentaje al sueldo preestablecido.

Asimismo, hay que entender que el operador de incremento resume lo siguiente:

```
  sueldo = sueldo + aumento;
```

Una vez que hemos creado el **constructor de objeto** y los métodos inherentes para que todo esto sea posible, nos queda ir a la clase principal y crear las **instancias** de la clase **Empleado**. Esto lo haremos dentro del método **main**:

```
  Empleado empleado1 = new Empleado("Juan Pérez", 15000,
2000, 03, 18);
         Empleado empleado2 = new Empleado("Marta
Rodríguez", 12000, 2003, 05,22);
         Empleado empleado3 = new Empleado("Martín de
Robertis", 16000, 1998, 10,30);
```

Hemos instanciado tres objetos de la clase **Empleado** y dentro de los parámetros pasamos los argumentos necesarios para cada uno. Notemos que siempre usaremos la palabra **New** para crear instancias de la clase **Empleado**.

Debemos destacar que, si hemos creado el constructor con tres argumentos, debemos hacerlo con la misma cantidad, en caso contrario nos marcaría un error; lo mismo sucede con los tipos de datos, deben coincidir con los declarados antes.

Ahora utilizamos el método **aumentaSueldo()** para darles un aumento (por ejemplo **10%**) a cada uno de los empleados instanciados:

```
  empleado1.aumentaSueldo(10);
  empleado2.aumentaSueldo(10);
  empleado3.aumentaSueldo(10);
```

Ya hemos creado todo lo que necesitamos, ahora vamos a imprimir para poder ver los resultados por consola, para ello debemos escribir lo siguiente:

```
  System.out.println("| Nombre:"+empleado1.unNombre()+ "
| Sueldo: "+ empleado1.unSueldo()+ " | Fecha de Ingreso:
"+empleado1.unaFechaAlta());
  System.out.println("| Nombre: "+empleado2.unNombre()+
" | Sueldo: "+         empleado2.unSueldo()+ " | Fecha de
Ingreso: "+empleado2.unaFechaAlta());
          System.out.println("| Nombre: "+empleado3.
unNombre()+ " | Sueldo: " +empleado3.unSueldo()+ " | Fecha
de Ingreso: " + empleado3.unaFechaAlta());
```

Al ejecutar el programa veremos lo siguiente:

```
  | Nombre: Juan Pérez | Sueldo: 16500.0 | Fecha de
Ingreso: Sat Mar 18 00:00:00 ART 2000
  | Nombre: Marta Rodríguez | Sueldo: 13200.0 | Fecha de
Ingreso: Thu May 22 00:00:00 ART 2003
```

```
  | Nombre: Martín de Robertis | Sueldo: 17600.0 | Fecha de
Ingreso: Fri Oct 30 00:00:00 ART 1998
```

Podemos simplificar esta codificación a través de alguna de los temas que fuimos aprendiendo a lo largo del libro, para ello, implementaremos arreglos:

```
   Empleado[] misEmpleados  = new Empleado[3];
      misEmpleados[0] = new Empleado ("Juan Pérez", 15000,
2000, 03, 18);
      misEmpleados[1] = new Empleado ("Marta Rodríguez",
12000, 2003, 05,22);
      misEmpleados[2] = new Empleado ("Martín de Robertis",
16000, 1998, 10,30);

       for(int i =0;i<3;i++){
              misEmpleados[i].aumentaSueldo(10);
  }
```

Hemos creado un array **Empleado[]**, seguido del nombre del array (**misEmpleados**). De esta forma estamos instanciando dentro de la variable del array con el índice de cada uno de los elementos que creemos: **misEmpleados[0]**. Esto lo igualamos a **New** con los parámetros que consideremos convenientes.

Recordemos que debemos pasar la misma cantidad de argumentos de los creados y respetar el tipado. En el ejemplo tenemos tipos **String**, **double** e **int**.

Luego es necesario implementar un bucle **for** que recorra el arreglo y que sea menor a la cantidad de elementos creados. Cuando escribimos **misEmpleados[i].** aparecerán los distintos métodos que creamos, elegimos el que hace que aumente el sueldo y, como argumento el **10**, (que es el **10%**):

```
  for(int i =0;i<3;i++){
      System.out.println("Nombre: " + misEmpleados[i].
```

```
unNombre()+ " Sueldo: " + misEmpleados[i].unSueldo() + "
Fecha Ingreso: "+ misEmpleados[i].unaFechaAlta());
          }
```

Este bucle **for** sostendrá los elementos de impresión que, a diferencia del ejemplo que comentamos, debemos colocarle a **misEmpleados[i]**. También podemos implementar un **for each**:

```
      for(Empleado e: misEmpleados){ // una forma resumida
del for (con for each)
      System.out.println("Nombre: " + e.unNombre()+
" Sueldo: " + e.unSueldo() + " Fecha Ingreso: "+
e.unaFechaAlta());
       }
```

Cuando ejecutemos el programa, nos presentará el mismo resultado. Con esto tenemos un programa funcional, aunque la forma de implementarlo dependerá de nuestros gustos y necesidades.

USO DE CONSTANTES

Las **constantes** son elementos cuyo valor permanece inalterable a lo largo de un programa, y la manera de crearlas es muy parecida a la de las variables. Por convención deben escribirse en mayúsculas para diferenciarlas de las variables, además se debe agregar adelante el modificador **final**. En consecuencia, cada vez que veamos en una instrucción este modificador **final**, debemos asociarlo a una constante.

Su sintaxis es la siguiente: **static final NOMBRE_CONSTANTE = valor**. Veamos un ejemplo para aclarar este concepto: **static final PI = 3.14159**.

Podemos verificar que, antes del modificador **final**, está la palabra reservada **static**.

Ahora presentaremos un pequeño programa para ver en qué casos se pueden aplicar las constantes en Java:

```
  public class UsoConstantes {
       public static void main(String[] args) {
          Empleado empleado1 = new Empleado("Juan");
          Empleado empleado2 = new Empleado("Rosa");
          empleado1.cambiaArea("Administración");
          System.out.println(empleado1.devuelveDatos());
  System.out.println(empleado2.devuelveDatos());
      }
  }
    class Empleado{

    public Empleado(String nom){ //Constructor
         nombre = nom;
         areaTrabajo = "RRHH";
      }
   public void cambiaArea(String areaTrabajo){//SETTER
          this.areaTrabajo = areaTrabajo;
          }
   public String devuelveDatos(){  //GETTER
         return "El nombre es: "+ nombre + " el área es " +
areaTrabajo;
          }
          private String nombre;

   }
```

En principio, este código posee información de empleados que se inician en un área laboral, para ello hemos creado un constructor con los atributos de los objetos. Luego con un **setter cambiaArea()** y un **getter devuelveDatos()**, lograremos ciertos cambios en ellos.

En el método **main** instanciamos los objetos de la clase **Empleado** e incluimos información de dos empleados. Hasta aquí no hemos hecho nada que no sepamos, pues son métodos y clases que venimos trabajando en los últimos tres capítulos. Ahora bien, qué pasaría si creamos un **setter cambiaNombre()**:

```
   public void cambiaNombre(String nombre){ //SETTER
              this.nombre = nombre;
         }
```

Luego, en el **main** instanciamos al objeto:

```
empleado1.cambiaNombre("María");
```

Cuando ejecutemos este programa veremos que **empleado1**, al que inicialmente le pusimos por nombre **Juan**, pasará sin ningún problema a llamarse **María**.

Java no tiene ningún problema de hacerlo, pero esto no significa que sea lo correcto, puesto que existen reglas a la hora de codificar.

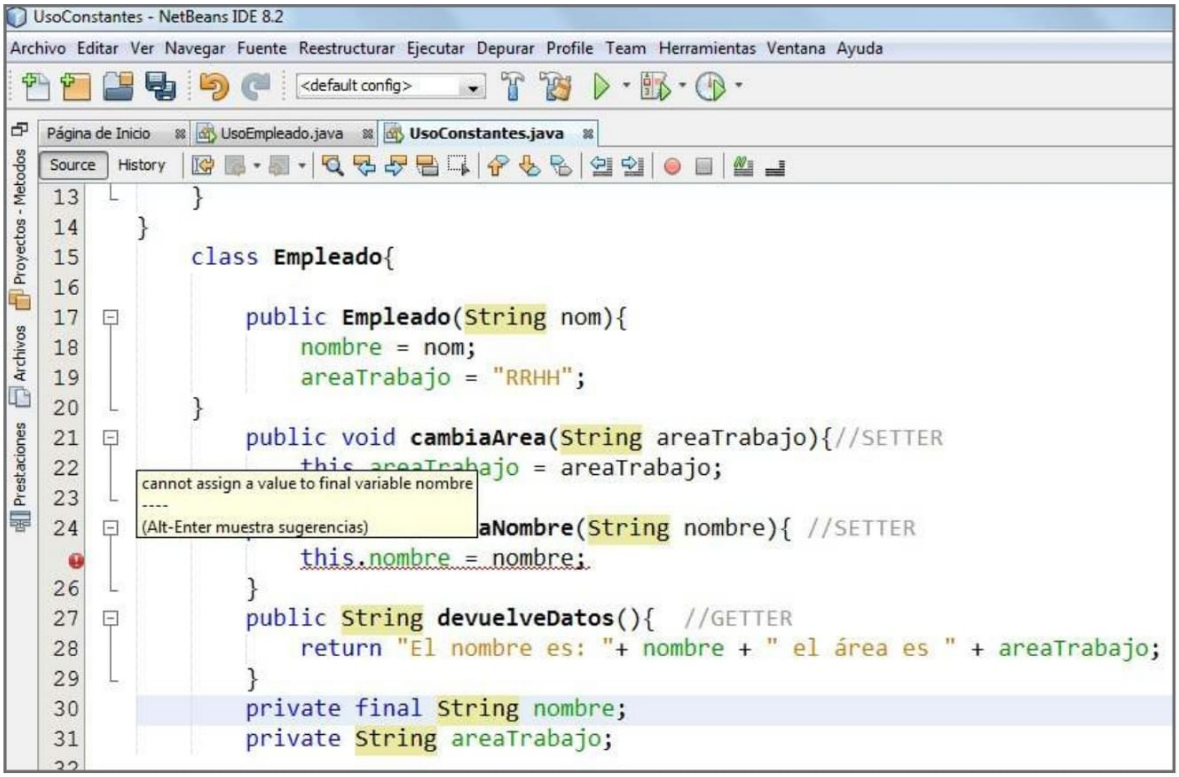


Figura 4. En la codificación, nótese que el IDE nos muestra cómo falla el programa si ponemos la palabra clave final antes del tipo de dato del atributo nombre.

Recordemos que, en el constructor de la clase, creamos dos variables con sus respectivos atributos:

```
private String nombre;
private String areaTrabajo;
```

A **nombre** le colocaremos la palabra reservada **final**, antes del tipo de dato. Con esto, esa variable (**nombre**) no puede ser modificada:

```
private final String nombre;
```

Podemos notar que, inmediatamente, el programa comienza a fallar en todos los lugares en donde se invoca a la variable. Si lo ejecutamos, nos mostrará el siguiente error:

```
Exception in thread "main" java.lang.RuntimeException:
Uncompilable source code - cannot assign a value to final
variable nombre
```

Trataremos las excepciones en profundidad en el **Capítulo 9** de este libro. Lo que haremos ahora es eliminar el **setter** que habíamos creado para cambiar el nombre, así como el llamado a él en el **main**.

USO DE STATIC

La palabra reservada **static** puede indicar que un atributo de una clase puede no pertenecer a una instancia de una clase, sino a la clase en sí. Por ello, cuando definimos un atributo como **static**, solemos llamarlo **atributos de la clase**.

Usemos el ejemplo anterior, en principio vamos a darle a cada empleado un **Id**, sabemos además que este número no puede repetirse.

Parametrización de métodos

Es recurrente producir errores lógicos cuando un método contiene un parámetro o una variable local con el mismo nombre que un campo de la clase. En esta situación, usaremos la referencia **this** si deseamos acceder al campo de la clase; en caso contrario, se hará referencia al parámetro o a la variable local del método. Por cierto estos errores son muy difíciles de localizar.

Ahora bien, usaremos la variable del tipo estática para ver de qué manera vamos a implementar dentro del ejemplo:

```
class Empleado{
  public Empleado(String nom, int Id){ //constructor
            nombre = nom;
            areaTrabajo = "RRHH";
            this.Id = Id;
    }
```

Dentro del constructor, agregamos la variable de tipo entero **Id** y luego con el **this** igualamos a **Id**.

Nuevamente, el **IDE** nos marcará error en los objetos instanciados en la clase principal (**main**), la que, si prestamos atención nos dirá que están preparados para recibir dos parámetros y solo les hemos pasado uno (recordemos que es un concepto que ya hemos introducido).

Luego debemos agregar lo siguiente para que no nos marque más este tipo de errores muy recurrentes al omitir el paso de parámetros:

```
Empleado empleado1 = new Empleado("Juan",1);
        Empleado empleado2 = new Empleado("Rosa",2);
```

Aquí hemos colocado a cada empleado un **Id** después de los nombres (**1** y **2**, respectivamente). Ahora que ya tenemos pasados los parámetros, falta aclararle al método **getter** (que es el encargado de devolver los valores) que ahora también devuelva los **Id** creados:

```
public String devuelveDatos(){  //GETTER
            return "El nombre es: "+ nombre + " el área
es " + areaTrabajo + " y el ID: " + Id;
        }
```

Esto funciona perfecto para fines educativos, pero, si queremos implementarlo para un entorno productivo, no sería muy práctico.

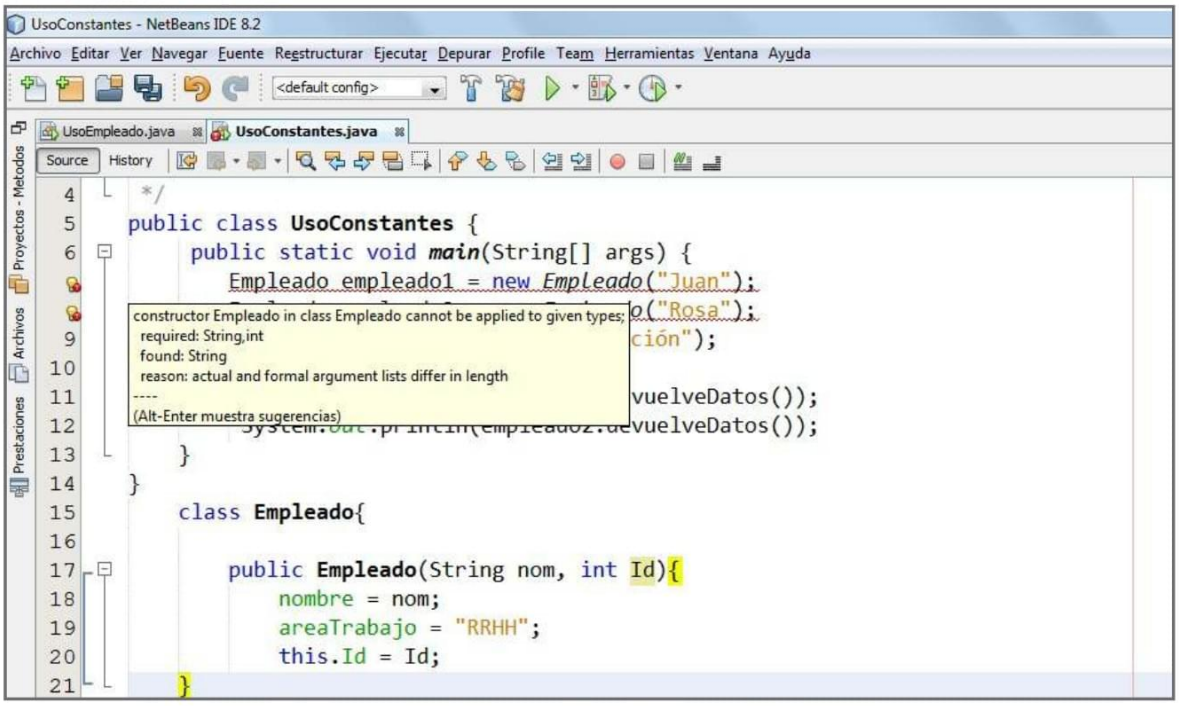


Figura 5. Notemos el error que nos muestra al haber ingresado un nuevo tipo de dato (int) en los parámetros del constructor, pues antes tenía String y básicamente estamos omitiendo el int.

Para no cometer errores a la hora de cargar información, por ejemplo, que se repitan **Id** se saltee un número correlativo, Java posee una manera de resolverlo de manera automática, aquí es donde debemos implementar la palabra clave **static**.

Sabemos que, cuando se instancia una clase y crea los objetos, este crea una copia de los atributos en cada uno de los objetos, lo que implica que cada objeto recibe una copia diferente (aunque en aspecto se parezcan). Para los efectos del ejercicio, este efecto pareciera no ser el indicado, pues no querríamos que ambos empleados compartieran el **Id** o algún otro atributo. Es como vamos a conseguir que cada uno de los objetos comparta una copia única de la variable **Id**. Esto lo hacemos poniendo el modificador **static** delante de la variable.

Ahora vamos a deshacer todo lo hecho en el punto anterior, y agregaremos al atributo **Id** el modificador **static**:

```
private final String nombre;
private String areaTrabajo;
public static int Id;
```

Podemos ver que el IDE pone nuestra variable **Id** en cursiva, esto se debe a que se ha convertido en una **variable de clase**. Ahora, en el constructor, vamos a asignar que **Id =1**:

```
Id =1;
```

Si ejecutamos el programa, notamos que, para ambos empleados, va a dar el mismo **Id** (**1**). Esto es muy sencillo de solucionar, usaremos un incrementador y lo vamos a colocar en:

```
System.out.println(empleado1.devuelveDatos());
          Empleado.Id++;
System.out.println(empleado2.devuelveDatos());
           Empleado.Id++;
```

Empleado.Id++ logrará que el **Id** de cada empleado aumente de a uno en uno. Hasta acá funciona de la manera esperada, pero aquí tenemos dos situaciones y es donde entra en escena la **encapsulación**, puesto que le habíamos puesto **public** a la variable **Id** (solo para que esto funcionara), pero, para efectos de buenas prácticas, esto no deberíamos hacerlo. El otro problema es que hay que incrementar en forma manual.

Al cambiar el modificador de acceso de **public** por **private**, automáticamente saltan los errores de acceso a los miembros.

En principio crearemos una variable auxiliar a la que llamaremos **IdSiguiente**, que implementaremos de la siguiente manera en el constructor:

```
public Empleado(String nom){
            nombre = nom;
            areaTrabajo = "RRHH";
            Id =IdSiguiente;
            IdSiguiente++;
    }
```

Luego, en el **getter** también debemos colocarlo:

```
…
   private int Id;
    private static int IdSiguiente=1;
```

Notemos que le hemos asignado un valor inicial de **1**, esto se debe a la indexación, así el primer **Id** no empieza en **0**. Ahora, cuando ejecutemos el programa, nos mostrará lo siguiente:

```
El nombre es: Juan el área es RRHH y el ID: 1
El nombre es: Rosa el área es RRHH y el ID: 2
El nombre es: Alberto el área es RRHH y el ID: 3
```

Y ya no tenemos que preocuparnos de hacer el incremento cada vez que incorporemos un nuevo empleado.

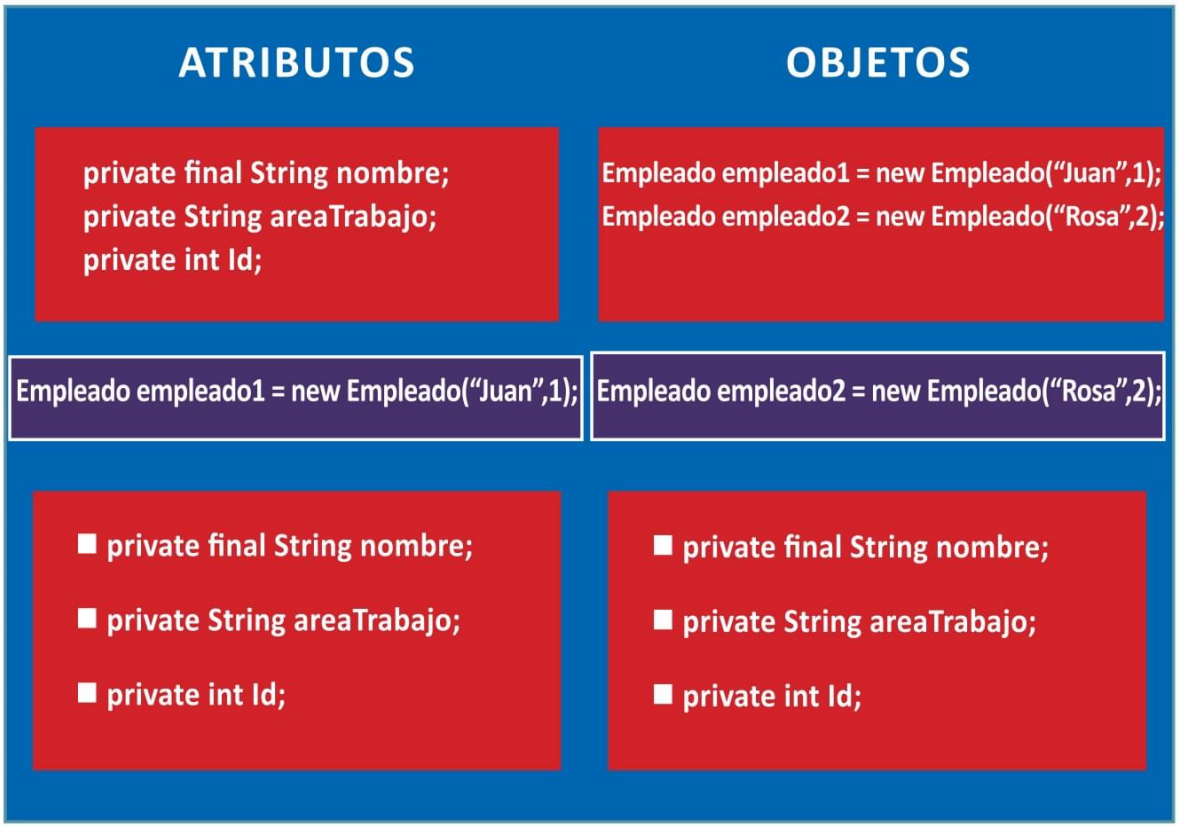


Figura 6. Gráfico del funcionamiento del static en Java. Notamos que cada instancia de objeto crea copia de los atributos de la clase (variables).

MÉTODOS STATIC

Este tipo de métodos pertenecen solo a la clase y no a un objeto, para llamarlo no podemos utilizar ningún objeto y solo tiene acceso a los atributos **static** de la clase, por lo cual no es necesario instanciar un objeto para invocarlo. La sintaxis de los métodos **static** es la siguiente:

```
NombreClase.metodo()
```

Veamos, sobre el ejemplo anterior, cómo podemos implementar este tipo de métodos en Java. Dentro de la clase **Empleado** vamos a crear el siguiente método:

```
…
public static String darIdSiguiente(){
       return "El ID siguiente es : "+ IdSiguiente;
      }
```

El método **static** obligatoriamente tendrá un **return** y regresará, en este caso, un atributo de la clase. Otro detalle importante es que los métodos de esta índole no acceden a los campos de clase (sean constantes o variables), sino que actúan sobre **IdSiguiente** que es estático. Para verificar esto realicemos el siguiente cambio:

```
public static String darIdSiguiente(){
            return "El ID siguiente es : "+ Id;
        }
```

Vemos que **Id** es una variable de campo **non-Static**, y, automáticamente el IDE nos mostrará el siguiente error:

```
Exception in thread "main" java.lang.RuntimeException:
Uncompilable source code - non-static variable Id cannot be
referenced from a static context
```

Para continuar, en el **main** vamos a trabajar con la salida de información:

```
   …
   System.out.println(empleado1.devuelveDatos()+"\
n"+empleado2.devuelveDatos()+"\n"+empleado3.
devuelveDatos()+"\n");
            System.out.println(Empleado.darIdSiguiente());
```

Como vemos, hemos trabajado sobre la clase ya que el método, al ser estático, no permite trabajar sobre objetos instanciados.

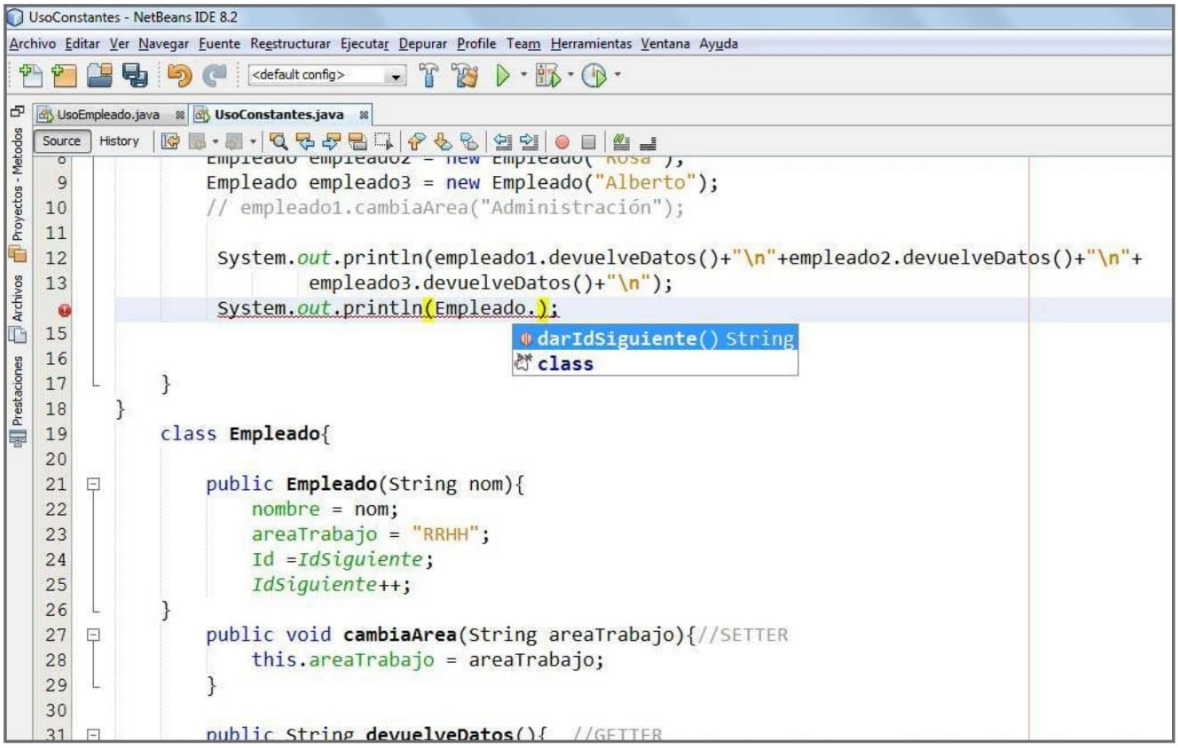


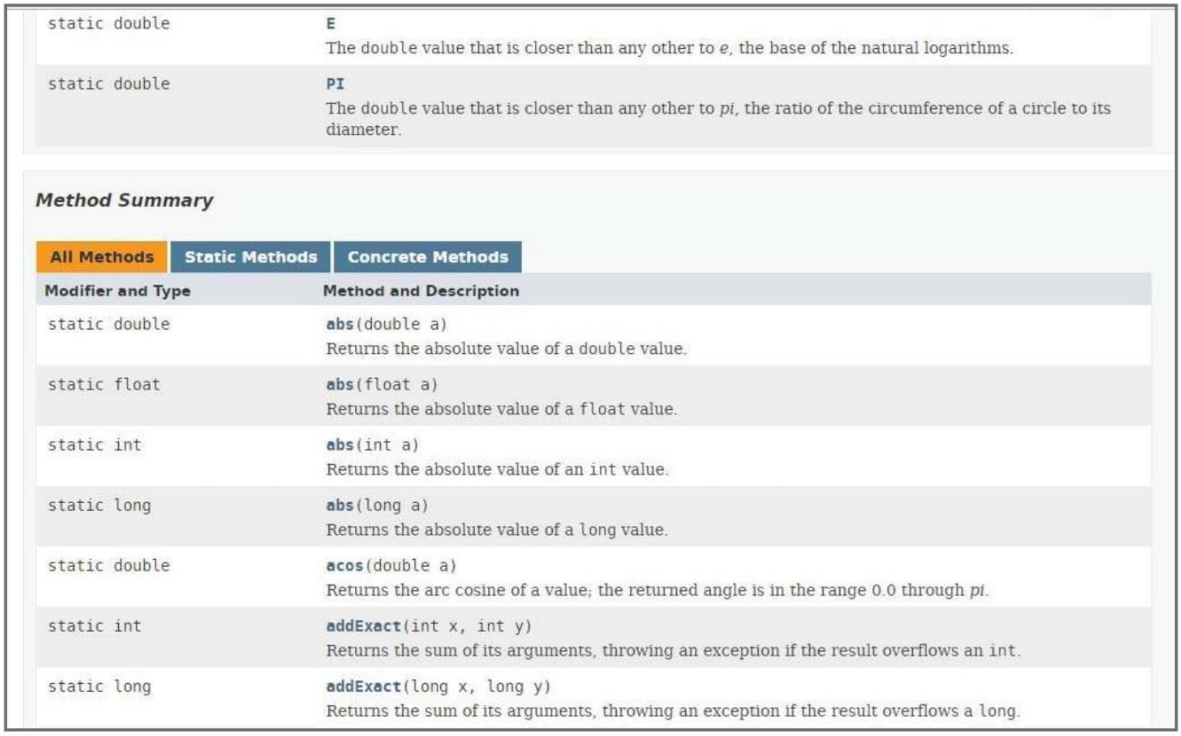
Figura 7. En la imagen podemos ver que solo al escribir la clase nos muestra el método estático, ya que conceptualmente estos métodos no trabajan sobre los objetos.

Métodos estáticos de la API

Cuando trabajamos con clases de la API de Java y utilizamos algunos de sus métodos, sin darnos cuenta también lo hacemos con los llamados **métodos estáticos**. Por ejemplo, con la clase **Math**. Veamos cómo:

```
public class CalculosBasicos{
     Math.sqrt(9); //Halla la raíz cuadrada de 9
     Math.pow(2,3); //Halla la potencia de 2 elevado al cubo
}
```

En esos ejemplos observamos que los dos métodos, tanto **sqrt** como **pow**, son estáticos, pues actúan sobre una clase (**Math**) y no sobre objetos.



static double	**E** The double value that is closer than any other to *e*, the base of the natural logarithms.
static double	**PI** The double value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

Method Summary

All Methods | Static Methods | Concrete Methods

Modifier and Type	Method and Description
static double	**abs**(double a) Returns the absolute value of a double value.
static float	**abs**(float a) Returns the absolute value of a float value.
static int	**abs**(int a) Returns the absolute value of an int value.
static long	**abs**(long a) Returns the absolute value of a long value.
static double	**acos**(double a) Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*.
static int	**addExact**(int x, int y) Returns the sum of its arguments, throwing an exception if the result overflows an int.
static long	**addExact**(long x, long y) Returns the sum of its arguments, throwing an exception if the result overflows a long.

Figura 8. La API de Java en la clase Math. Verificamos que todos los métodos que pertenecen a esta clase son del tipo static y que pueden devolver un int, float, double o long.

Alias de tipos

En Java, no podemos definir alias de tipo en un nivel superior, pero podemos hacerlo por el alcance de una clase, o un método. Consideremos esto con los nombres de Integer, Long, si le ponemos nombres más cortos: I y L.:

```
class Test<I extends Integer> {
   <L extends Long> void x(I i, L l) {
      System.out.println( i.intValue() + ", " + l.longValue()); }}
```

El método main

Otro método **static**, que trabajamos desde el comienzo, es el **main**, cuya sintaxis es la siguiente:

```
public static void main(String[] args) {
    instrucciones…
}
```

El contenido de la instrucción es **public**, para que su contenido pueda ser utilizado dentro de la clase y de otras; **void**, pues no retorna ningún valor y, finalmente, **static** ya que este método no actúa sobre ningún objeto y, cuando empezamos a utilizarlo, no hay ningún objeto construido, de hecho es este método el encargado de construir algún objeto.

Por otro lado, este método recibe parámetros, para ser exactos un array (llamado **args**) de tipo **String**.

SOBRECARGA DE CONSTRUCTORES

Sabemos que un constructor es un método que le va a dar una instancia inicial a los objetos. Debemos entender que las declaraciones de las clases pueden tener versiones alternativas de constructores o métodos, que proporcionen varias maneras de llevar a cabo una tarea en particular mediante diferentes conjuntos de parámetros. A esto que lo llamamos **sobrecarga de un constructor** o **constructores múltiples** (se puede crear un número grande de ellos, no hay límite).

Veamos el ejemplo de **UsoEmpleado** para entender cómo podemos sobrecargar métodos. En principio, recordemos que ya teníamos un constructor al que le pasamos algunos parámetros:

```
public Empleado(String nom, double sue, int año, int mes,
int dia){
    …
}
```

Plantearemos un caso en el que ingresa un nuevo empleado del que solo conocemos el nombre. Sabemos que, si no le colocamos los parámetros a un método, no podemos darles un estado inicial a los objetos que construyamos, porque nos daría un error. Entonces vamos a sobrecargar el método, para que le dé un estado inicial a algún atributo; esto lo haremos dentro de la clase **Empleado**:

```
//sobrecarga de constructores
      public Empleado(String nom){
          nombre = nom;
      }
```

Observamos que hemos pasado tan solo un parámetro, pues es solo ese valor el que conocemos. Ahora vamos a la clase principal (donde está el **main**) e instanciaremos a un nuevo objeto. Notemos que nos va a sugerir qué constructor es el que necesitamos (de los dos que tenemos):

```
Empleado[] misEmpleados  = new Empleado[4];
      misEmpleados[0] = new Empleado ("Juan Pérez", 15000,
2000, 03, 18);
      misEmpleados[1] = new Empleado ("Marta Rodríguez",
12000, 2003, 05,22);
      misEmpleados[2] = new Empleado ("Martín de Robertis",
16000, 1998, 10,30);
      misEmpleados[3] = new Empleado("José Coronado"); //
nuevo objeto pero con distinto constructor
```

El último empleado que ingresamos tiene un solo parámetro, pues pertenece al método que acabamos de crear, y recordemos que solo va admitir un único argumento, en caso contrario nos marcaría error.

Vamos a ejecutar el programa y nos mostrará lo siguiente:

```
…
Nombre: José Coronado Sueldo: 0.0 Fecha Ingreso: null
```

Hasta acá, como no teníamos información acerca del sueldo ni de la fecha de ingreso del empleado **José Coronado**, nos dará sueldo **0.0** y fecha **null**.

Recordemos que la fecha no era un tipo primitivo, sino una variable de objeto (**Date**), por eso nos dará como resultado un **null**.

Nos preguntamos qué pasaría si, en vez de obtener ese tipo de resultados inciertos para los empleados de quienes no tengamos suficiente información, nos diera algo predeterminado (por ejemplo, un sueldo mínimo y una fecha base):

```
   //sobrecarga de constructores
   public Empleado(String nom){
        // nombre = nom;
         this(nom,8000,2000,01,01); // llama al otro
constructor

          }
```

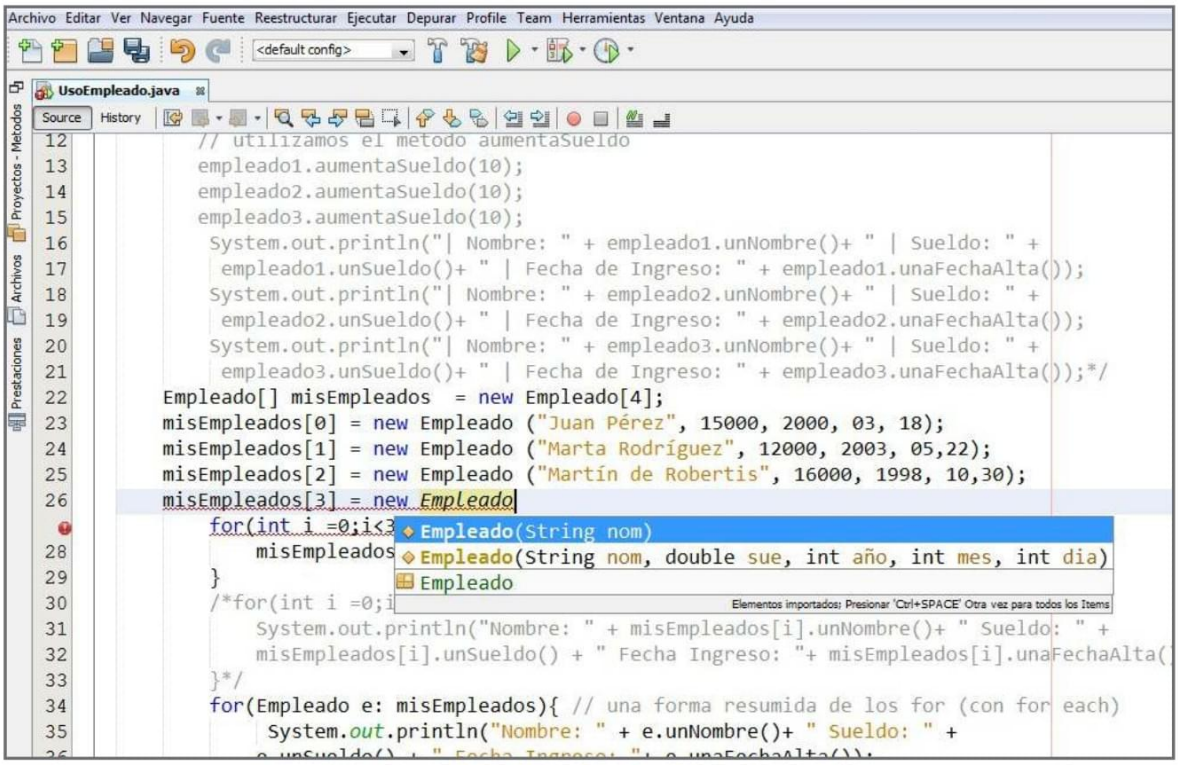


Figura 9. Cuando instanciamos Empleado en el código, el IDE nos sugiere a los dos constructores, entonces, debemos elegir con cuántos datos contamos para cargarle la información necesaria.

HERENCIA

La **herencia** es una de las características más sobresaliente en el paradigma orientado a objetos, pues va a permitir que una clase herede los atributos y los métodos de otra clase, esto significa que podremos reutilizar códigos. Con la herencia, las clases tienen un sistema de jerarquías, en consecuencia, cada una de ellas presenta una **superclase**, llamada también clase **padre** o clase **base** y, a su vez, puede tener **subclases**, denominadas clases **hija** o clases **extendidas**.

Un dato para tener en cuenta es que, en Java, cada clase puede tener solo una clase padre (**herencia simple**), y las clases hijas podrían tener sus propios atributos y métodos, además de los heredados; estos a su vez pueden ser modificados.

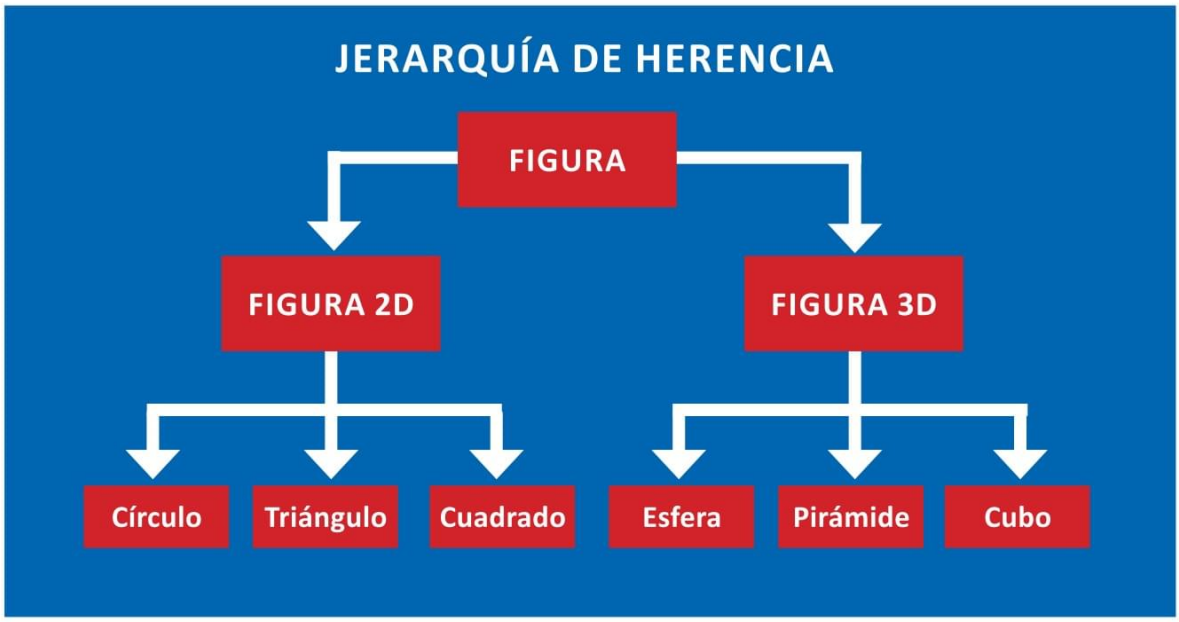


Figura 10. Con este gráfico podemos darnos una idea de la existencia de jerarquías de herencia en Java a través de las figuras geométricas de 2 y 3 dimensiones.

Traslademos este concepto a la vida real, imaginemos que alguno de nosotros heredáramos, por ejemplo, una casa, en el supuesto caso que esta tuviera una deuda, también la heredaríamos y, de hecho, deberíamos hacernos cargo de ella.

En este momento es necesario que utilicemos el proyecto **POO** que contenía las clases **Auto** y **Uso_Auto**. Para utilizar la herencia y entender cómo funciona este concepto, crearemos una clase nueva, que será

Furgoneta, ya que este tipo de automóvil, a simple vista, no posee las características de los coches normales, pero sí comparte algunas de ellas, como las ruedas, las puertas, el motor, el peso, el color, etcétera. Además, presenta características propias que un coche no tiene, por ejemplo, una capacidad de carga superior o que puedan viajar más personas. A esto lo llamaremos **características extras** (**atributos**) que el objeto furgoneta presentará.

Necesitamos crear la clase **Furgoneta**, que tendrá ciertos atributos que ya vimos en la clase **Auto**, es decir, que van a ser compartidos, y además otros propios. Haremos que esta clase herede, de la clase **Auto**, alguna de sus características, así ahorramos la necesidad de codificar cosas que ya teníamos hechas y, con esto, aplicamos el término de **reutilización de código**. Ahora hay que agregar que no solo heredaremos atributos, sino también métodos (si tiene aire acondicionado, asientos normales o de cuero, etcétera).

Para crear una nueva clase, debemos hacerlo desde el paquete abierto, como dijimos antes, se llamará **Furgoneta**:

```
package poo;
/**
 * @author Carlos Arroyo
 */
public class Furgoneta{

}
```

La primera línea es el paquete al que nos referíamos (**poo**), luego están los comentarios (datos del autor en este caso) y el nombre de la clase.

Para que esta clase herede de otra, debemos agregarle la palabra reservada **extends** después de la clase y, luego, el nombre de la clase padre (**Auto**):

```
public class Furgoneta extends Auto{

}
```

Si hablamos de **jerarquías**, la clase **Auto** es una superclase, y la clase **Furgoneta** se acaba de convertir en una subclase.

Yendo un poco más a fondo en este concepto, la clase **Furgoneta** tendrá atributos y métodos propios, además que va a heredar de la clase padre otros. Sin embargo, una subclase no puede heredar de otras clases, aunque en la vida real esto pueda suceder, en POO en Java no existe la **herencia múltiple**, concepto que sí existe en otros lenguajes de programación.

Ahora empezamos a codificar nuestra nueva clase:

```
  public class Furgoneta extends Auto{

      private int capacidadCarga;
      private int asientosExtra;

      public Furgoneta(int asientosExtra, int
capacidadCarga){ //constructor de la clase
          super();  //llama al constructor de la
superclase→Auto
          this.capacidadCarga = capacidadCarga;
          this.asientosExtra = asientosExtra;
      }
  }
```

En principio hemos creado dos atributos propios de la clase **Furgoneta**, de la misma manera los encapsulamos usando **private**.

Herencia múltiple

En el paradigma orientado a objetos existe la posibilidad de heredar características y comportamientos de una clase a otra, sin embargo, esto solo se puede hacer desde una única clase, a esto lo llamamos **herencia simple**. En Java no existe la herencia múltiple, es decir, que una clase pueda heredar atributos y métodos de varias clases, esto sí sucede en otros lenguajes, como C++. Pero, a cambio, dispone de la construcción denominada **Interface**, que permite una forma de simulación o implementación limitada de la herencia múltiple.

Luego escribimos el método constructor de la clase y le pasamos dos parámetros (con los mismos nombres de los atributos).

Debemos tener en cuenta que **super()** es una palabra reservada que significa que va a llamar al constructor de la clase padre. Luego, vamos a crear los **this** de cada uno de los argumentos ya que nos indicará que el elemento al que hace alusión se encuentra en la misma clase, es decir, que les está dando un estado inicial a las dos variables; además, vemos que se repiten los mismos valores luego del signo igual, a los que no hay que confundir, veamos:

```
this.capacidadCarga = capacidadCarga;
```

El primero pertenece a la **variable de la clase**, mientras que el segundo corresponde al argumento del **constructor de la clase**. Hay que decir que se pudo haber omitido el **this** dándole un nombre distinto a los argumentos. A esto le agregaremos un método **getter**:

```
//GETTER
    public String datosFurgoneta(){
        return " La capacidad de carga es: " +
capacidadCarga +" kg. "+ " y los lugares son: " +
asientosExtra;
    }
```

El método **getter** retornará un **String** con los valores de la capacidad de carga y concatenamos con los asientos extras que tenga el vehículo.

Antes teníamos una clase llamada **Uso_Auto**, sin embargo, para evitar confusiones, le vamos a cambiar el nombre por el de **Uso_Vehiculo**, pues hemos agregado una clase **Furgoneta**. Esta acción también se puede hacer con las teclas **Ctrl + R**.

Una vez dentro de la clase **Uso_Vehiculo**, escribiremos el siguiente código:

```
…
Auto miAuto1 = new Auto();
       miAuto1.estableceColor("Azul");
```

```
Furgoneta miFurgo1= new Furgoneta();
            …
```

Notemos que, en la tercera línea, surge un error; si observamos en el marcador, que nos da sugerencias y errores, veremos indicado que faltan argumentos y, de hecho, dos del tipo enteros (**int**), recordemos que en la clase **Furgoneta** el constructor tenía dos argumentos. Corrijamos de la siguiente forma:

```
Furgoneta miFurgo1= new Furgoneta(8,600);
```

Entonces **8** es la capacidad extra de asientos y **600** la capacidad de carga. Ahora pasaremos al nuevo objeto **miFurgo1** algunos estados o métodos:

```
…
    miFurgo1.estableceColor("blanco");
    miFurgo1.estableceAsientos("si");
    miFurgo1.configuraAire("Si");
```

Cuando escribimos el objeto instanciado y el punto, el IDE nos propone una cierta cantidad de atributos y métodos, pero esta vez, al ser una clase que hereda de otra, también se verán los de la clase padre, es decir, de **Auto**.

Y este es el punto al que queríamos llegar, la gran ventaja de la herencia, la **reutilización de código**, ya que no hemos tenido la necesidad de programar de nuevo.

Sobrecargas de métodos

El lenguaje Java no permite que dos métodos sean "equivalentes de anulación" dentro de la misma clase, independientemente de sus cláusulas potenciales o tipos de devolución potencialmente diferentes.

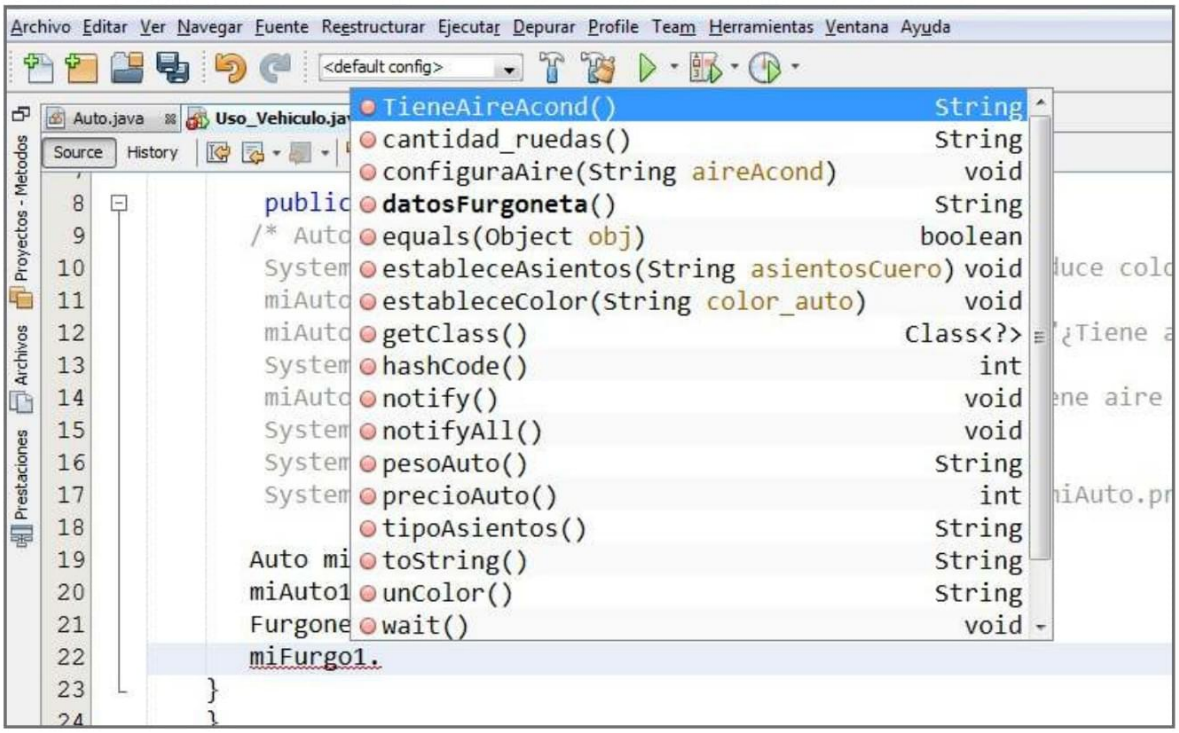


Figura 11. En esta imagen se muestra que, al poner el punto luego de escribir el objeto instanciado, los atributos y los métodos tanto de Furgoneta como de Auto también serán visibles.

En la clase **Auto** agregaremos un **getter** (luego de los atributos):

```
      public String datosGenerales(){ //Getter
              return "La plataforma del vehículo tiene "+ ruedas
+ " ruedas"+", mide: "+ largo/1000+ " mts con un ancho de: "+
ancho+" cms y un peso de plataforma de " + peso_plataforma+ " kg.";
            }
```

Este **getter** en sí tiene como datos generales ciertos estados de la clase **Auto**, pero al usar la herencia cuando hagamos el objeto **miFurgo1** invocamos el **getter**, veamos lo que sucede:

```
    System.out.println(miFurgo1.datosGenerales());
```

Cuando lo ejecutamos obtenemos lo siguiente:

```
  La plataforma del vehículo tiene 4 ruedas, mide: 2 mts
con un ancho de: 300 cms y un peso de plataforma de 600 kg.
```

Entonces, por la herencia, nos traerá características propias de la clase **Auto** (que tiene 4 ruedas, el largo y el ancho), en consecuencia, arraigará esas características para sí. Ahora traeremos los datos propios de la furgoneta, para ello lo vamos a concatenar con el código anterior:

```
System.out.println(miFurgo1.datosGenerales()+ miFurgo1.
datosFurgoneta());
```

Esta vez la salida, luego de la ejecución, será la siguiente:

```
La plataforma del vehículo tiene 4 ruedas, mide: 2 mts
con un ancho de: 300 cms y un peso de plataforma de 600 kg.
La capacidad de carga es: 600 kg. y los lugares son: 8
```

Por otro lado, si queremos ingresar los métodos al objeto de clase **miAuto1** y colocamos el punto para verificar qué tipo de métodos nos sugiere el IDE, verificamos que no aparece ninguno de los métodos propios de la clase **Furgoneta**, y esto es lógico, ya que son métodos propios de ella, por lo tanto, el otro objeto no los podría ver y mucho menos invocar.

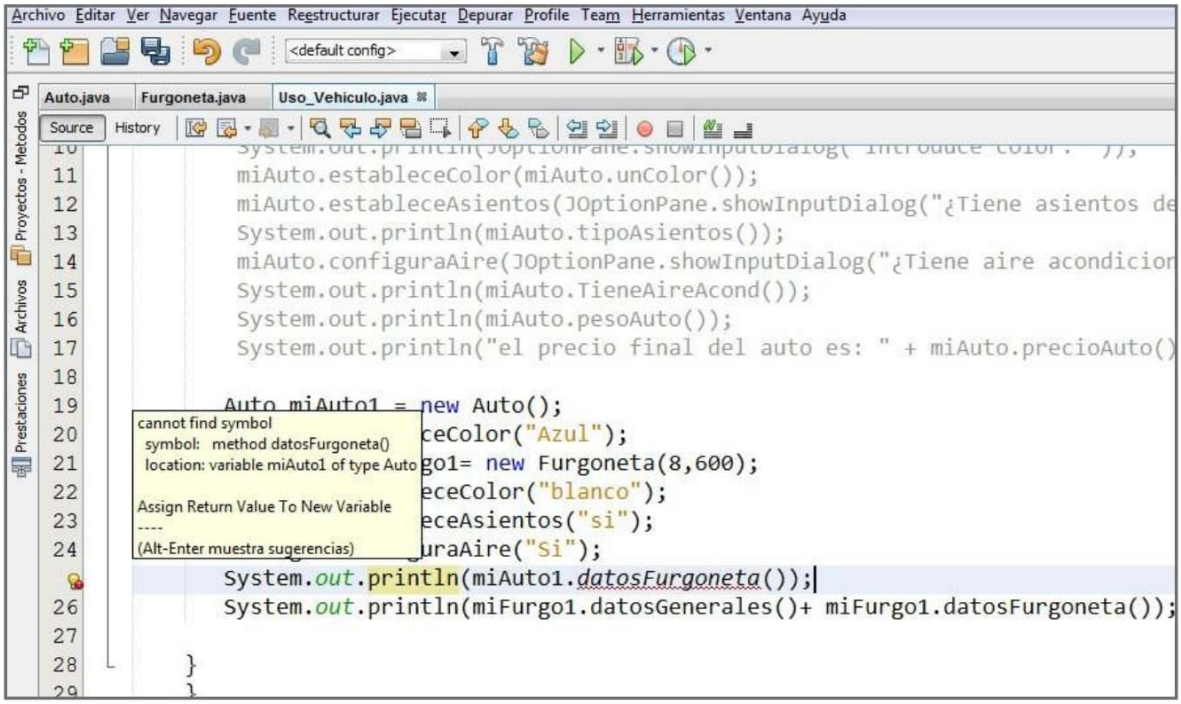


Figura 12. En la figura se verifica que, a pesar de que el método datosFurgoneta() de la clase Furgoneta fue invocado por el objeto miAuto1, este dará error.

Regla de diseño

Esta regla consiste en aplicar el término **"Es un..."** a nuestro diseño de clases. En el ejemplo tenemos que la clase **Furgoneta** hereda de la clase **Auto**:

```
public class Furgoneta extends Auto{
…
}
```

Según esta regla, nos preguntamos: "¿Una furgoneta **es un** auto?". Y claramente la respuesta es **No**. Pues una furgoneta es un tipo de vehículo; en cambio, un vehículo no siempre es una furgoneta.

Para concluir: pudimos haber programado una clase más bien general llamada **Vehiculo** de la cual van a heredar características el resto de las clases. A esto lo llamamos "hacer una planificación correcta de nuestros programas".

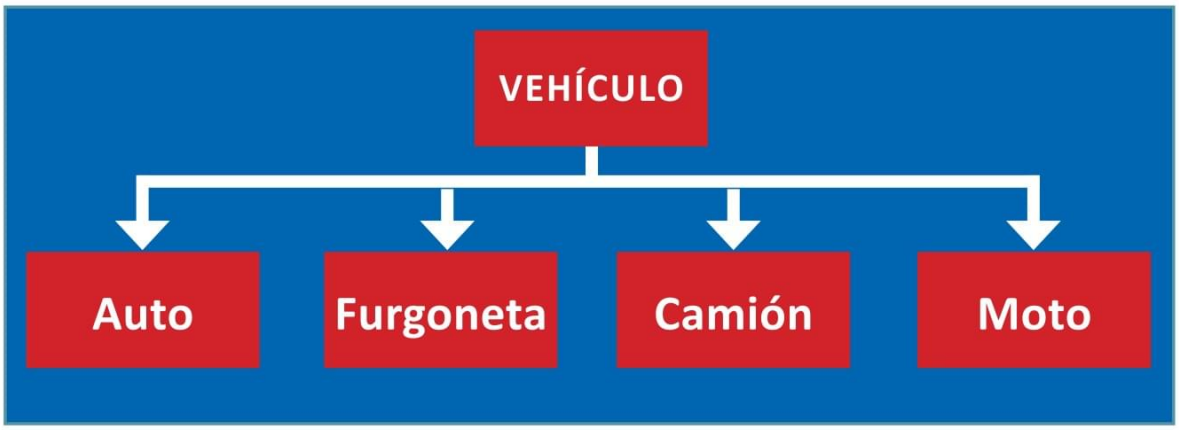


Figura 13. La figura nos muestra la herencia y cómo debería respetarse la regla de su diseño.

EL POLIMORFISMO

Este término está muy ligado en Java al concepto de herencia, nos dice que podemos programar de una manera más general en vez de una forma específica, esto nos permite escribir programas que procesen objetos que compartan la misma superclase, de una manera directa o indirecta, como si todos fueran parte de ella, simplificando de esta forma enormemente la programación de objetos.

Veamos un ejemplo de la vida real para comprender este concepto tan abstracto. Supongamos que queremos programar varios tipos de vehículos: un automóvil, un vehículo que va por agua-tierra (anfibio) y otro que, a la vez, despliegue alas y logre volar. Todos estos vehículos en sí pueden tener tanto comportamientos propios y distintos como comportamientos en común, por ejemplo, el de moverse.

Esto lo programaremos en la superclase **Vehículo**. Cuando enviamos el mensaje mover, cada vehículo sabe cómo hacerlo, puesto que fue programado para eso. Nosotros confiamos en que lo hacen, independientemente de cómo lo hagan. Esta respuesta al mismo mensaje de cada uno de los tres objetos es el concepto clave del **polimorfismo**, pues este mismo mensaje tiene diversas formas de resultados.

Veamos un caso para poder implementar este concepto, para ello trabajaremos sobre el ejemplo de **Uso_Empleado**, pero esta vez crearemos una clase más que se llamará **Jefe**. Esta clase será una subclase de **Empleado**. De tal forma que implementaremos de una manera correcta el uso de la herencia. Nos hacemos la pregunta: "¿Un Jefe **es un** empleado? La respuesta es **sí**. Ahora lo hacemos al revés: "¿Un empleado **es un** jefe?". La respuesta es **no siempre**, entonces, vamos a dejar que la clase **Jefe** sea una subclase de **Empleado**.

```
   public class Jefe extends Empleado{

      public Jefe(String nom, double sue, int año, int mes,
int dia){
           super(nom, sue, año, mes, dia);
      }
```

En el código de la clase **Jefe**, que va a heredar de la clase **Empleado**, le creamos un constructor con cinco argumentos, luego, en **super** colocaremos también cinco argumentos.

Crearemos un campo de clase a la que llamaremos **incentivo**, recordemos que para el ejemplo solo los jefes recibirán un incentivo:

```
  private double incentivo; //campo de clase
```

Ahora un **setter** para que reciba el parámetro con el valor que le pasemos por el incentivo:

```
public void ganaIncentivo(double m){ //SETTER
    incentivo = m;
}
```

Le pasamos a la variable **incentivo** el parámetro **m** que habíamos pasado por argumento en el **setter**. Un tema que no podemos olvidar es que, si bien en la clase **Jefe** es posible heredar el sueldo del empleado, en la práctica esto no sucede así. Aquí debemos sobrescribir el método **unSueldo()** que es el que nos va traer el sueldo heredado:

```
public double unSueldo(){
    double sueldoJefe = unSueldo();
    return sueldoJefe + incentivo;
}
```

Sobrescribimos el método y adentro le cargamos información propia de la clase **Jefe**. El IDE nos marcará que estamos sobrescribiendo el método y que escribamos **@Override** antes del método sobrescrito, a manera de sugerencia.

Si nos localizamos en la otra clase, también existirá una marca a la altura del método **public double unSueldo()**, que nos advierte que este método se encuentra redefinido. Por lo tanto, en la clase donde lo sobrescribimos, invalida al método de origen.

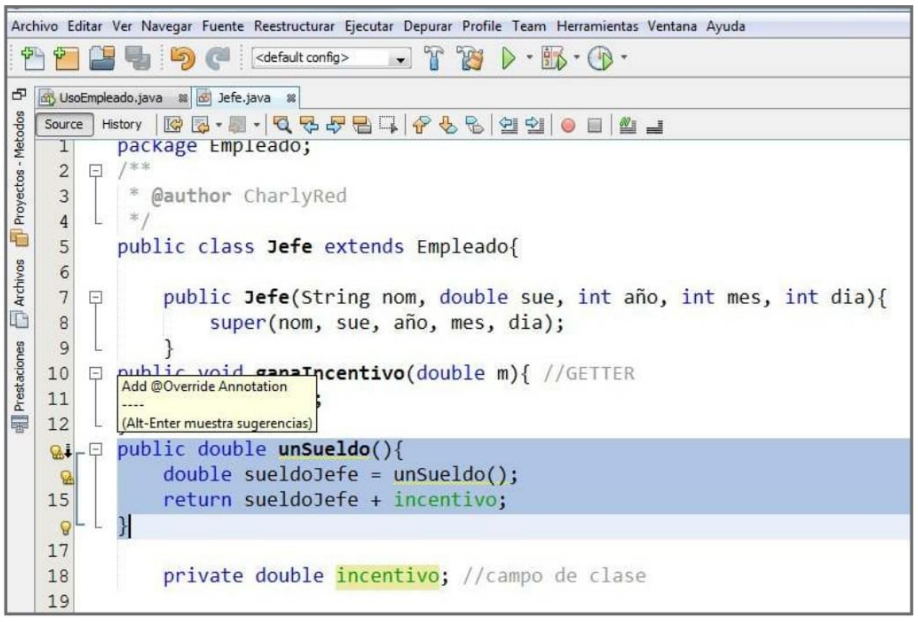


Figura 14. El IDE nos señala en los marcadores que estamos sobrescribiendo un método y nos sugiere que escribamos @Override.

Ahora corregiremos algo que no está bien del todo en el código en el que hemos trabajado. Al sobrescribir un método de otra clase, no se garantiza que traiga los elementos que requerimos, porque le pedimos que almacene en **sueldoJefe** lo que devuelve el método **unSueldo**. Debería quedar de la siguiente forma:

```
double sueldoJefe = super.unSueldo();
```

Al utilizar **super** le decimos que no llame al método de la clase **Jefe**, sino al de la clase padre.

En la clase **Uso_Empleado** escribiremos el código para instanciar el nuevo objeto de uno de los jefes, en este caso un jefe de RRHH:

```
Jefe jefeRRHH = new Jefe("Manuel
Fernández",50000,1998,04,21);
      jefeRRHH.ganaIncentivo(6000);
      Empleado[] misEmpleados  = new Empleado[5];
```

Una vez instanciado, le cargamos el método **ganaIncentivo()** con un valor **double**. Y luego, como habíamos trabajado con arrays, debemos agregar un índice más (**5**). Ahora es cuando entra en escena el concepto de polimorfismo:

```
misEmpleados[4] = jefeRRHH; //polimorfismo
```

Opera el **principio de sustitución**, porque el objeto espera un objeto del tipo **Empleado**, sin embargo, estamos poniendo un objeto de tipo **Jefe**, con lo que utilizamos un objeto de la subclase aun cuando el programa espere a uno de la superclase. Agreguemos un elemento más

Referencia a Superclase

Cuando referenciamos a una superclase, lo podemos hacer de una manera polimórfica a cualquier método declarado en la superclase y sus superclases. Una variable de un tipo de interfaz debe hace referencia a un objeto para llamar a los métodos y a todos los objetos de la clase Object.

en el array **[6]**, de tal forma que lo instanciemos e inicialicemos en una misma línea de código, el programa no debería mostrar un error:

```
  misEmpleados[5] = new Jefe("José López", 66000,
2001,10,27);
```

Recordemos que habíamos utilizado un **foreach** para que hiciera el recorrido por el array:

```
  …
          for(Empleado e: misEmpleados){
              e.aumentaSueldo(10);
  }
  for(Empleado e: misEmpleados){
               System.out.println("Nombre: " +
e.unNombre()+ " Sueldo: " +
              e.unSueldo() + " Fecha Ingreso: "+
e.unaFechaAlta());
          }
  …
```

Aquí es donde se ve la potencia del polimorfismo, puesto que la variable **e** pertenece a la clase **Empleado**, para el ejemplo nuestro los empleados no tienen incentivo; pero cuando ejecutamos el programa:

```
  Nombre: Manuel Fernández Sueldo: 61000.0 Fecha Ingreso:
Tue Apr 21 00:00:00 GMT-03:00 1998
```

Al sueldo base de **50.000** le agregamos un **10%** (**5000**) y le sumamos un incentivo de **6000**: esto nos dará **61000**.

Descendentes y ascendentes

Es importante considerar que cuando realizamos la tarea de asignar una variable de la superclase a una variable de la subclase (sin una conversión descendente explícita) veremos que se provocará un error de compilación. Por esta razón debemos poner atención a este tipo de asignación.

Esto quiere decir que el programa se da cuenta de que el empleado **Manuel Fernández** viene de la clase **Jefe** y le aplica el incentivo que solo tiene esa clase. Es decir, la variable **e** se comporta de una u otra forma, de acuerdo con el lugar de donde venga el objeto al que le envía el mensaje, esto se denomina **enlazado dinámico**, es decir, la forma cómo la máquina virtual de Java entiende y decide a quién enviarle un mensaje A y a quién enviarle un mensaje B.

RESUMEN CAPÍTULO 03

En este capítulo profundizamos en algunos aspectos de la programación orientada a objetos, trabajamos con la colaboración entre clases y la manera en que ellas se comunican a través de mensajes (métodos); aprendimos a construir objetos, trabajamos con el uso de constantes y final. También revisamos el uso de static y los métodos estáticos, conocimos los distintos constructores y la sobrecarga de métodos en sus distintas versiones. Finalmente vimos la herencia y el polimorfismo.

Actividades 03

Test de Autoevaluación

1. ¿Cómo funciona la colaboración entre clases?
2. ¿Qué es un constructor de objetos? ¿Qué función cumple dentro de una clase?
3. ¿Qué es una variable de clase?
4. ¿Qué es un tipo de datos de objeto? Nombre algunas diferencias de los tipos de datos primitivos.
5. ¿Cómo se implementa una constante?
6. ¿Qué es la palabra clave static?
7. ¿A qué llamamos la sobrecarga de un constructor? Muestre un par de ejemplos.
8. ¿Qué es la herencia?
9. ¿Qué es una superclase? ¿Y una subclase?
10. ¿Qué es el polimorfismo?

Ejercicios prácticos

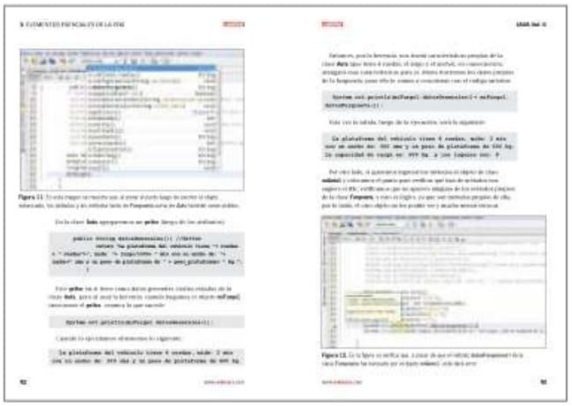
1. Cree tres clases: Alumno, Profesor y Materias. Mediante la colaboración de clases, estas deben comunicarse a través de métodos.
2. A la clase Profesor, del ejercicio anterior, agréguele dos constructores que tengan que ver con las materias y provoque la sobrecarga de estos, cuando existan materias comunes para distintos grados.
3. Cree una clase que oficiará de clase padre y tres clases hijas que hereden algunos atributos y métodos de la superclase. Debe lograr que la estructura de estas cumpla con la regla de oro del diseño de herencia.

USERS

Programación en Java™ Vol. II

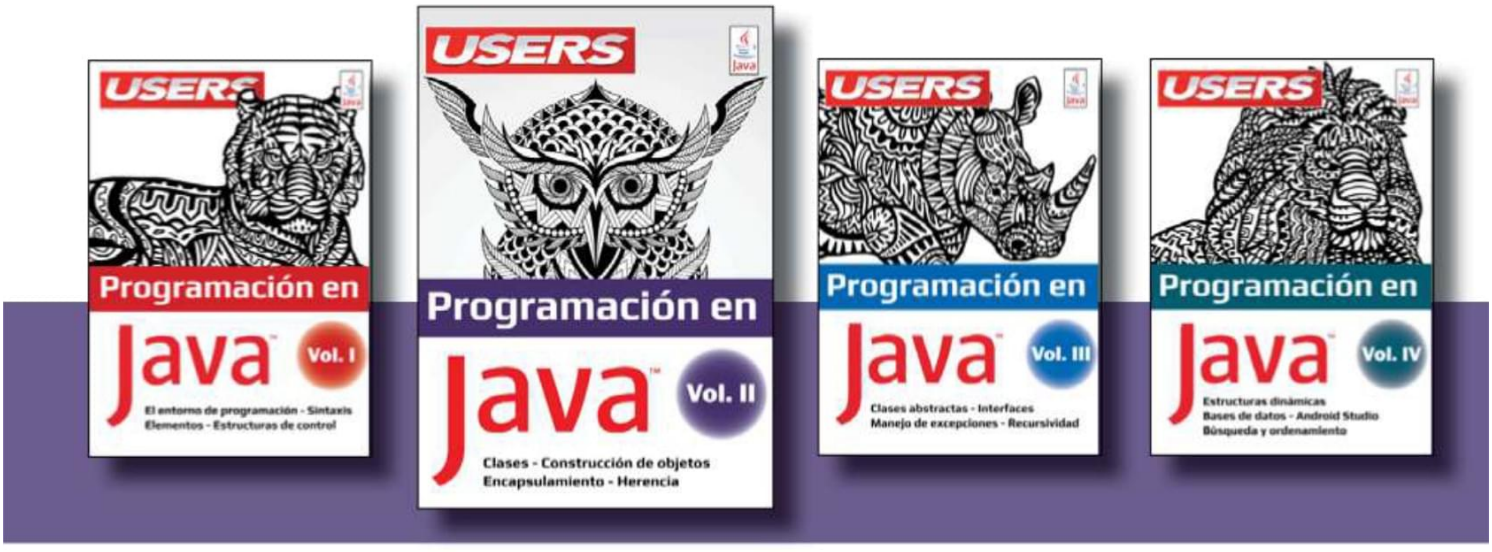
ACERCA DE ESTE CURSO

Java es uno de los lenguajes más robustos y populares en la actualidad, existe hace más de 20 años y ha sabido dar los giros adecuados para mantenerse vigente. Este curso de Programación en Java nos enseña, desde cero, todo lo que necesitamos para aprender a programar y, mediante ejemplos prácticos, actividades y guías paso a paso, nos presenta desde las nociones básicas de la sintaxis y codificación en Java hasta conceptos avanzados como el acceso a bases de datos y la programación para móviles.



ACERCA DE ESTE VOLUMEN

En este **volumen** se realiza el tratamiento de uno de los paradigmas más utilizados en la actualidad, la Programación Orientadas a Objetos (POO). Aprenderemos a instanciar objetos, veremos el acceso a los datos y sus restricciones a través de los modificadores y el uso de los constructores.



SOBRE EL AUTOR

Carlos Arroyo Díaz es programador, escritor y contenidista especializado en tecnologías y educación. Se ha desempeñado como consultor técnico en INET, y actualmente trabaja como mentor docente en el área de Programación en varios proyectos del Ministerio de Educación de la Ciudad Autónoma de Buenos Aires.

RedUSERS

En nuestro sitio podrá encontrar noticias relacionadas y participar de la comunidad de tecnología más importante de América Latina.

PREMIUM.REDUSERS.COM

RedUSERS PREMIUM la biblioteca digital de USERS. Accederás a cientos de publicaciones: Informes; eBooks; Guías; Revistas; Cursos. Todo el contenido está disponible online - offline y para cualquier dispositivo. Publicamos, al menos, una novedad cada 7 días.

ISBN 978-987-4958-07-5