# Manual de prácticas del laboratorio de Modelos de programación orientada a objetos

| Elaborado por: | Revisado por: | Autorizado por: | Vigente desde: |
|---|---|---|---|
| M.C. M. Angélica Nakayama C.<br>Ing. Jorge A. Solano Gálvez | **Laura Sandoval Montaño** | **Alejandro Velázquez Mena** | **14 de junio de 2018** |

# Índice de prácticas

# Guía práctica de estudio 01: Entorno y lenguaje de programación


***Elaborado por:***

M.C. M. Angélica Nakayama C.

Ing. Jorge A. Solano Gálvez

***Autorizado por:***

M.C. Alejandro Velázquez Mena

# Guía práctica de estudio 01: Entorno y lenguaje de programación

## Objetivo:

Identificar y probar el entorno de ejecución y el lenguaje de programación orientado a objetos a utilizar durante el curso.

## Actividades:

- Probar los conceptos básicos del entorno y el lenguaje.
- Revisar la instalación y configuración del entorno de ejecución.
- Realizar un programa en el lenguaje de programación a utilizar usando el entorno de ejecución, utilizando la sintaxis básica (notación, palabras reservadas, comentarios, etc.)

## Introducción

Para una mejor comprensión de la **programación orientada a objetos**, se deben practicar los conceptos aprendidos en la teoría implementándolos en algún lenguaje de programación.

Lo primero que se debe hacer para comenzar a programar en un lenguaje es conocer sus **fundamentos** y el **entorno de ejecución,** así como también las **herramientas** útiles con las que se cuenta para optimizar el desarrollo de programas.

Una vez que se han comprendido las bases del lenguaje, entorno y herramientas, se puede proceder a crear un programa sencillo, realizando todos los pasos necesarios desde la codificación en el lenguaje hasta la ejecución del mismo en el entorno.

**Nota: Elegir un lenguaje para aprender programación orientada a objetos es un tema de discusión entre programadores profesionales, ya que no existe un criterio unánime respecto a qué lenguaje es el ideal para aprender todos los conceptos necesarios. En estas**

**guías se tomará como caso de estudio el lenguaje de programación Java, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.**

## El entorno de ejecución

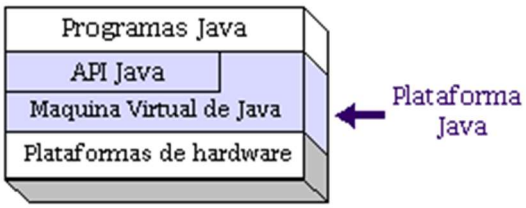
**La tecnología Java** es un lenguaje de programación y una plataforma comercializada por primera vez en 1995 por Sun Microsystems.

**El lenguaje de programación Java** fue originalmente desarrollado por James Gosling de Sun Microsystems (compañía que fue adquirida por Oracle) y publicado en 1995 como un componente fundamental de la plataforma Java de Sun Microsystems. Su sintaxis deriva en gran medida de C y C++, pero tiene menos utilidades de bajo nivel que cualquiera de ellos. El lenguaje de programación Java es un lenguaje de alto nivel, orientado a objetos. El lenguaje es inusual porque los programas Java son tanto compilados como interpretados.

**La plataforma Java** es una plataforma de software que se ejecuta sobre la base de varias plataformas de hardware. Está compuesto por la JVM (**Java Virtual Machine**) y la Interfaz de Programación de Aplicaciones Java o API (un amplio conjunto de componentes de software listos para usar, que facilitan el desarrollo y despliegue de aplicaciones).

Además de la API, toda implementación completa de la plataforma Java incluye:

- Herramientas de desarrollo para compilar, ejecutar, supervisar, depurar y documentar aplicaciones.
- Mecanismos estándar para desplegar aplicaciones para los usuarios.
- Kits de herramientas de interfaz de usuario que permiten crear interfaces gráficas de usuario (GUIs) sofisticadas.
- Bibliotecas de integración que permiten que los programas accedan a bases de datos y manipulen objetos remotos.

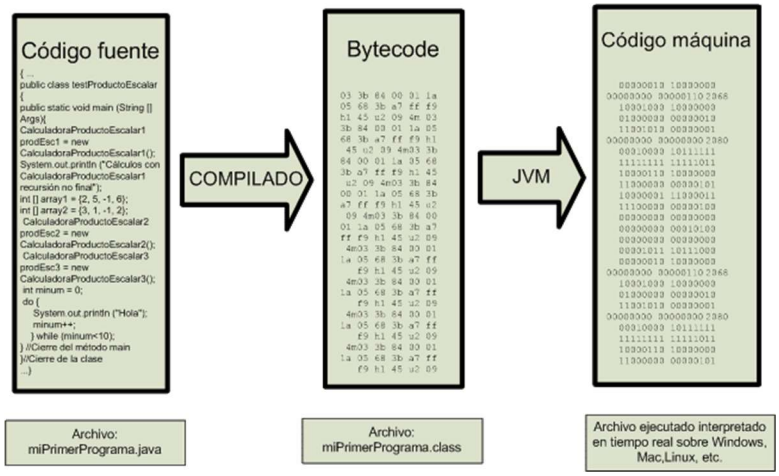



## La Máquina Virtual Java (JVM)

Java se hizo independiente del sistema operativo añadiendo un paso intermedio al proceso de compilación (traducir el código escrito en "Lenguaje entendible por humanos" a un código en "Lenguaje Máquina" que entienden las máquinas):

Los programas Java no se ejecutan en nuestra máquina real (en nuestra computadora) sino que Java simula una **"máquina virtual"** con su propio hardware y sistema operativo.

Entonces en Java el proceso es: del código fuente, se pasa a un código intermedio denominado habitualmente *"bytecode"* entendible por la máquina virtual Java. Y es esta máquina virtual simulada, denominada **Java Virtual Machine** o **JVM**, la encargada de interpretar el *bytecode* dando lugar a la ejecución del programa.

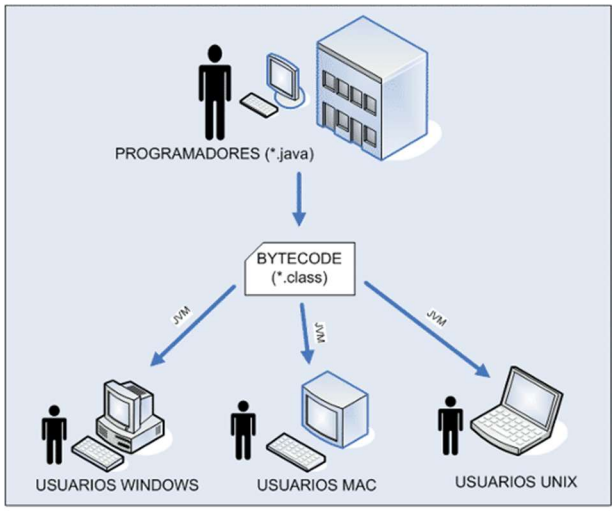



Esto permite que Java pueda ejecutarse en una máquina con sistema operativo Unix, Windows, Linux o cualquier otro, porque en realidad no va a ejecutarse en ninguno de los sistemas operativos, sino en su propia máquina virtual que se instala cuando se instala Java. El precio a pagar o desventaja de este esquema es que siempre que se quiera correr una aplicación Java se debe tener instalado el entorno de ejecución de Java, es decir, su máquina virtual.

Todo programa en Java está organizado en clases, éstas se codifican en archivos de texto con extensión **.java**. Cada archivo de código fuente **.java** puede contener una o varias clases, aunque lo normal es que haya un archivo por clase.

Cuando se compila un archivo **.java** se genera uno o varios archivos **.class** de código binario (*bytecodes*) que son independientes de la arquitectura. Esta independencia supone que los *bytecodes* no pueden ser ejecutados directamente por ningún sistema operativo.

Durante la fase de ejecución es cuando los archivos **.class** se someten a un proceso de interpretación, consistente en traducir los *bytecodes* a código ejecutable por el sistema operativo. Esta operación es realizada por la **JVM**.

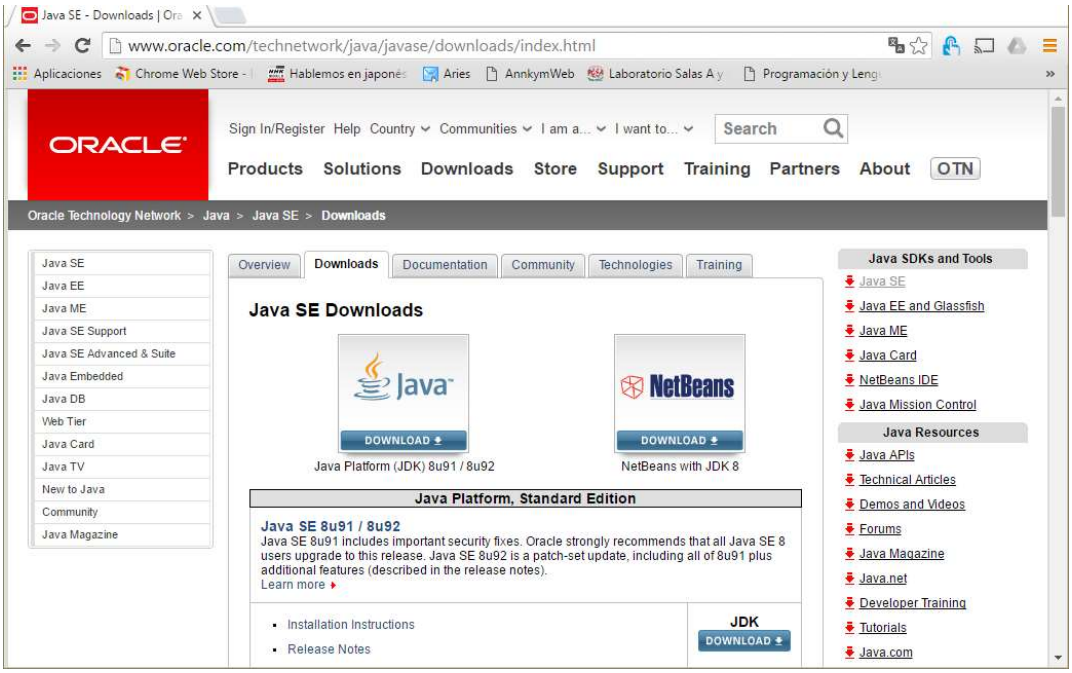


ESQUEMA COMPILADO – INTERPRETADO JAVA

## Herramientas de desarrollo (JDK)

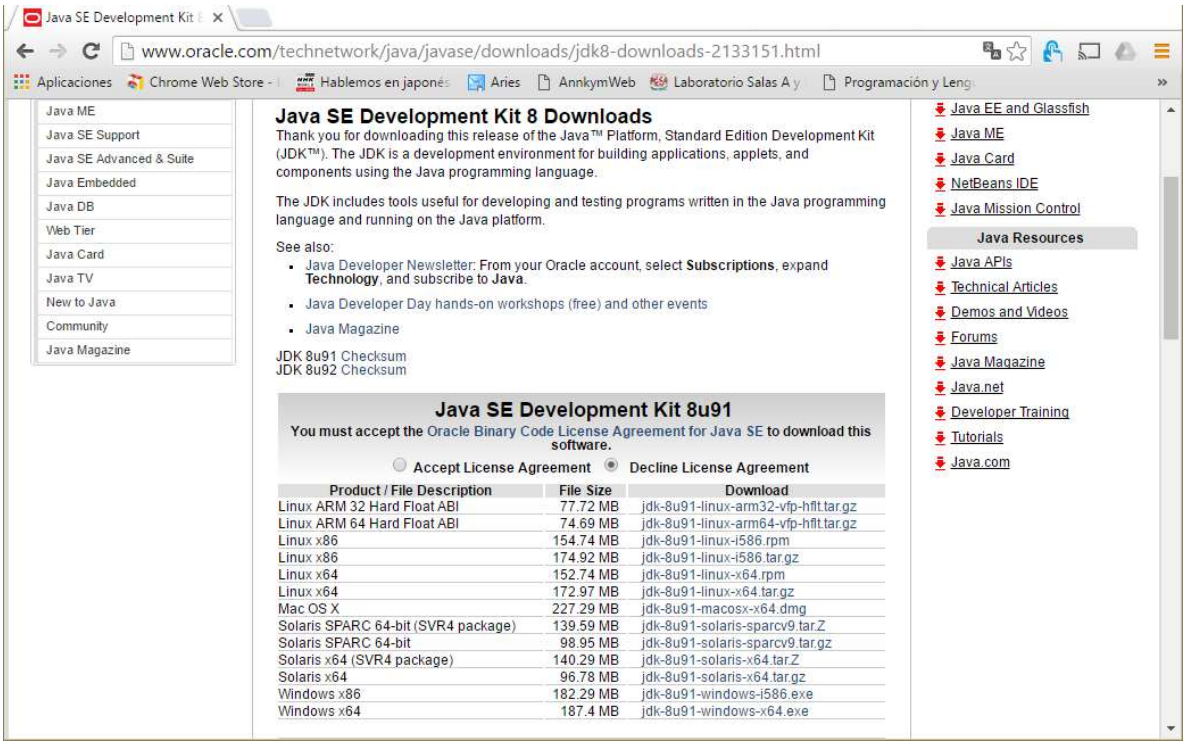
El **Java Development Kit (JDK)** proporciona el conjunto de herramientas básico para el desarrollo de aplicaciones con Java estándar. Se puede obtener de manera gratuita en internet, descargándola desde el sitio de Oracle.

http://www.oracle.com/technetwork/java/javase/downloads/index.html




En este sitio se pueden encontrar varios recursos relacionados con java, así como otras versiones y herramientas útiles.

Para este curso se utilizará la versión 8 de Java SE (Standard Edition).




Para obtener la descarga correcta se debe seleccionar el sistema operativo en donde se instalará y aceptar la licencia.

Las instrucciones de instalación en distintos sistemas operativos se encuentran disponibles en el sitio:

http://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html

# Programa en JAVA

Aunque aún no se tiene conocimiento del lenguaje, se puede crear un sencillo **programa en Java** que imprima un saludo. Este programa tiene como fin conocer el procedimiento general que se debe seguir para crear, compilar y ejecutar programas Java.

## Codificación

Utilizando cualquier **editor de texto** disponible en el sistema (Block de notas, notepad++, gedit, vi, etc.) se captura el siguiente código (teniendo en cuenta que Java es *case sensitive*, es decir, sensible a mayúsculas y minúsculas):

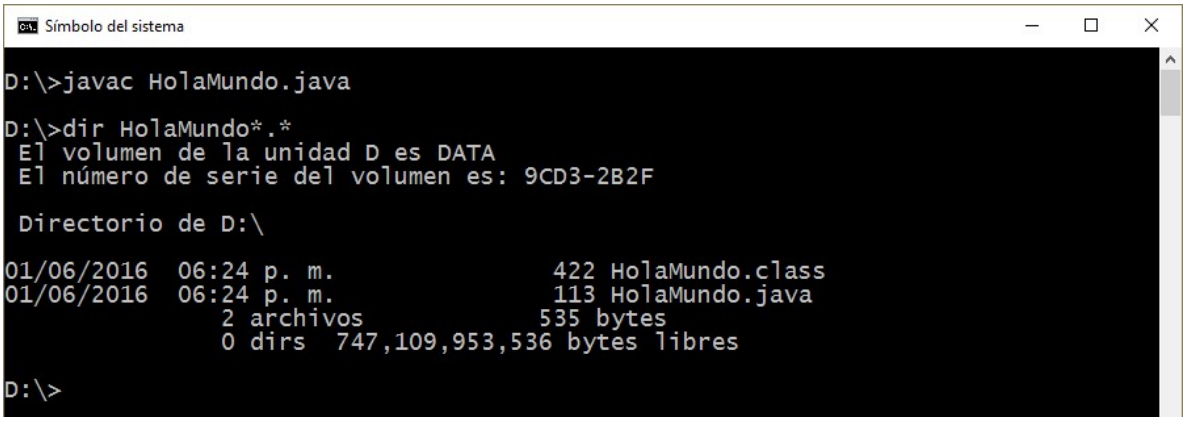
```
public class HolaMundo {
    public static void main(String[] args) {
        System.out.println("Hola Mundo");
    }
}
```

Después se guardar este programa en un archivo de texto llamado **HolaMundo.java** (el nombre del archivo debe ser el mismo que el de la clase)

## Compilación

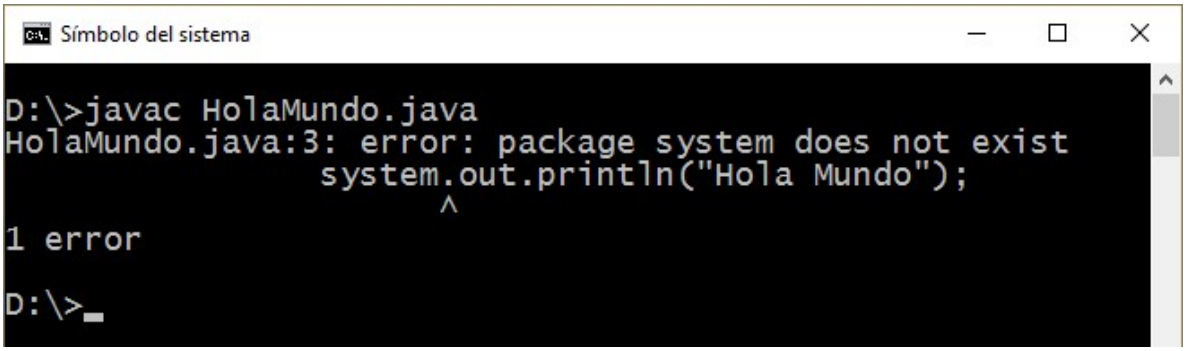
La compilación de un archivo de código fuente **.java** se realiza a través del comando **javac** del **JDK**. Si se ha instalado y configurado correctamente el **JDK**, entonces **javac** podrá ser invocado desde el directorio en el que se encuentre el archivo *HolaMundo.java* creado.

Tras ejecutar este comando, se generará un archivo **HolaMundo.class**.




En caso de que existieran errores sintácticos en el código fuente, el compilador informará de ello en la ventana de comandos y el archivo **.class** no se generaría.

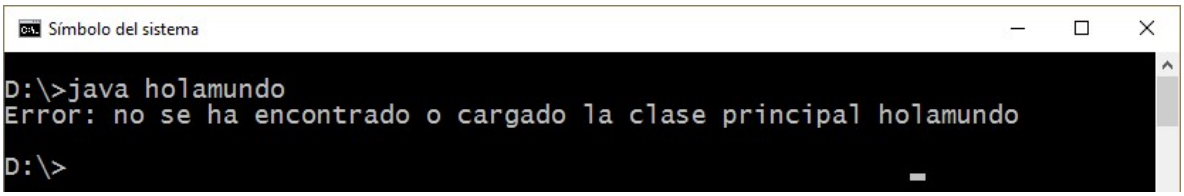
Por ejemplo, si en el código fuente escribimos **system** en vez de **System**, al intentar compilar se tendría la salida:



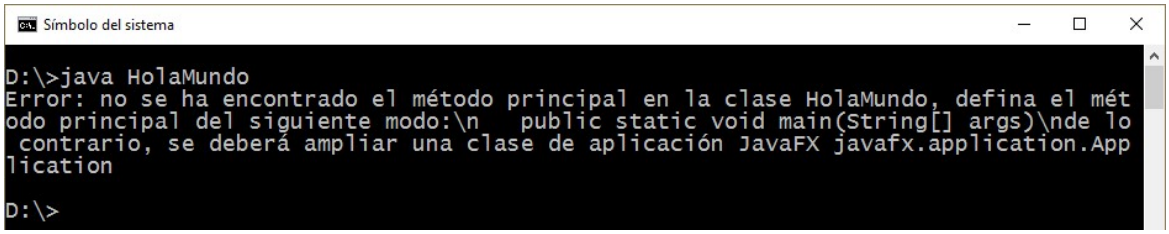
## Ejecución

Para ejecutar el programa (una vez compilado correctamente), se utiliza el comando **java** seguido del nombre de la clase que contiene el método *main()*. En este caso será **HolaMundo** ya que es la única clase existente.


La llamada al comando **java** insta a la máquina virtual a buscar en la clase indicada un método llamado *main( )* y procede a su ejecución. En caso que **java** no encuentre la clase ya sea porque el **JDK** esté mal configurado o porque el nombre de la clase sea incorrecto, se producirá un error como el siguiente:




Si el problema no es la configuración ni el nombre de la clase, si no el formato del método ***main( )*** no es correcto, el programa compilará correctamente pero se producirá un error al ejecutar el programa con java.



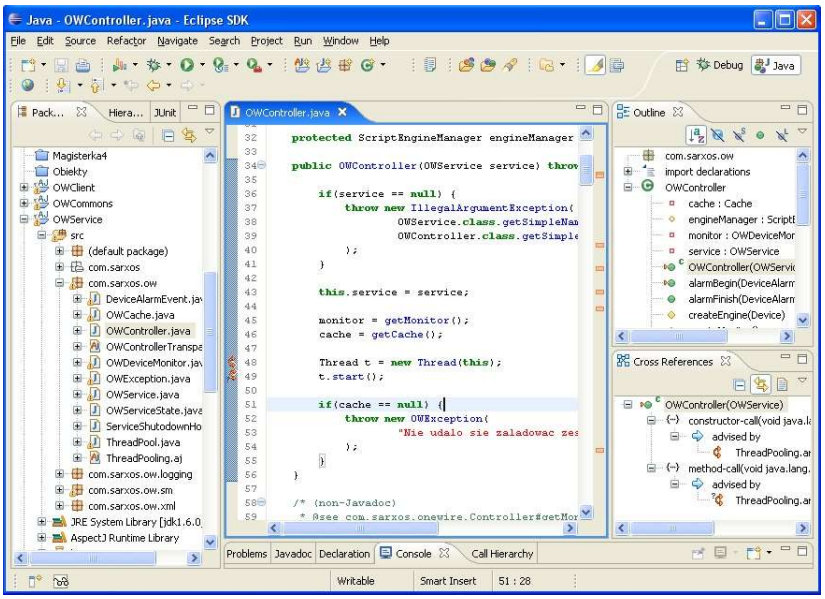

Este procedimiento para compilar y ejecutar la clase *HolaMundo* es el mismo para **cualquier programa en Java**.

## Entorno de desarrollo integrado (IDE)

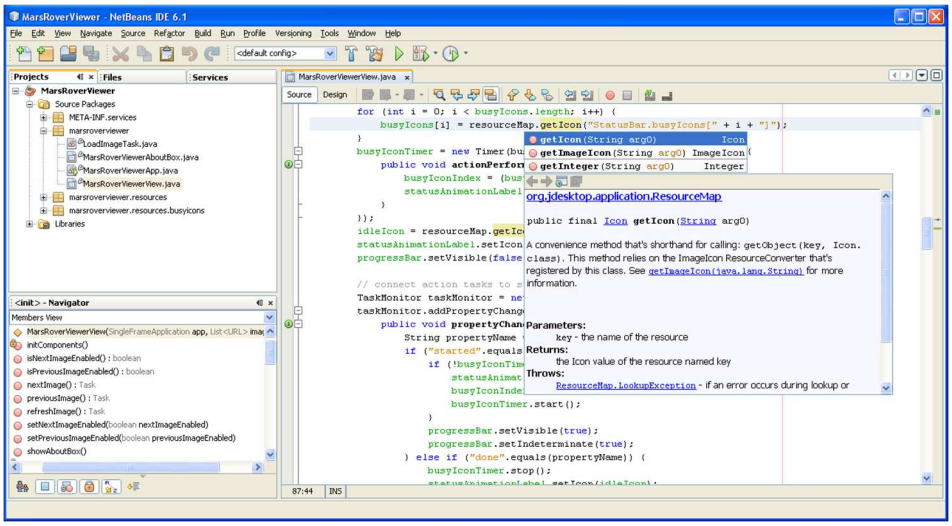
Para desarrollar un producto de software solo es necesario un editor de texto plano para capturar el código fuente y el compilador o el intérprete (según sea el caso) para transformar el lenguaje de alto nivel a lenguaje máquina. Sin embargo, también se puede hacer uso de una aplicación que contenga todas las herramientas en una interfaz, a este tipo de aplicaciones se les conoce como entorno de desarrollo integrado.

Un **entorno de desarrollo integrado** o **IDE** (**Integrated Development Environment**) es una aplicación que facilita el desarrollo de aplicaciones en algún lenguaje de programación. De manera general, un **IDE** es una interfaz gráfica de usuario diseñada para ayudar a los desarrolladores a construir aplicaciones de software proporcionando todas las herramientas necesarias para la codificación, compilación, depuración y ejecución.

Los **IDE** para Java utilizan internamente las herramientas básicas del **JDK** en la realización de estas operaciones, sin embargo, el programador no tendrá que hacer uso de la consola para ejecutar estos comandos, dado que el entorno ofrece una forma alternativa de utilización, basada en menús y barras de herramientas.



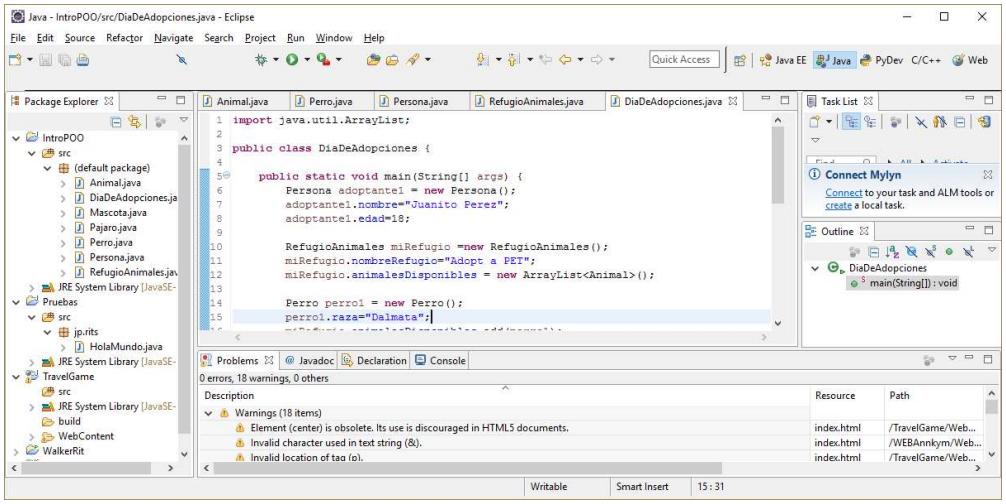

La escritura de código también resulta una tarea sencilla con un **IDE** ya que éstos suelen contar con un editor de código que resalta las palabras reservadas del lenguaje para distinguirlas del resto del código. Algunos incluso permiten autoescritura de instrucciones utilizando la técnica *Intellisense*, que consiste en mostrar la lista completa de métodos de un objeto según se escribe la referencia al mismo.



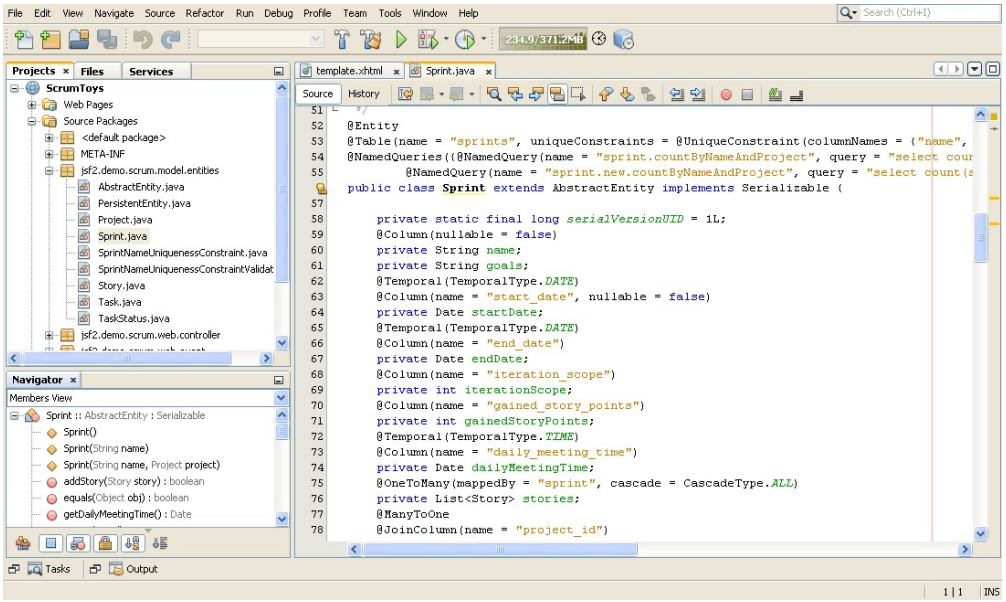
En el mercado existen diversos tipos de **IDE**, cada uno con características propias, empero, una constante es que permiten manejar las etapas para generar un programa dependiendo del tipo de lenguaje utilizado. La mecánica de utilización de estos programas es muy similar, todos ellos se basan en el concepto de proyecto como conjunto de clases que forman una aplicación.

Algunos de los IDE más populares para Java son:

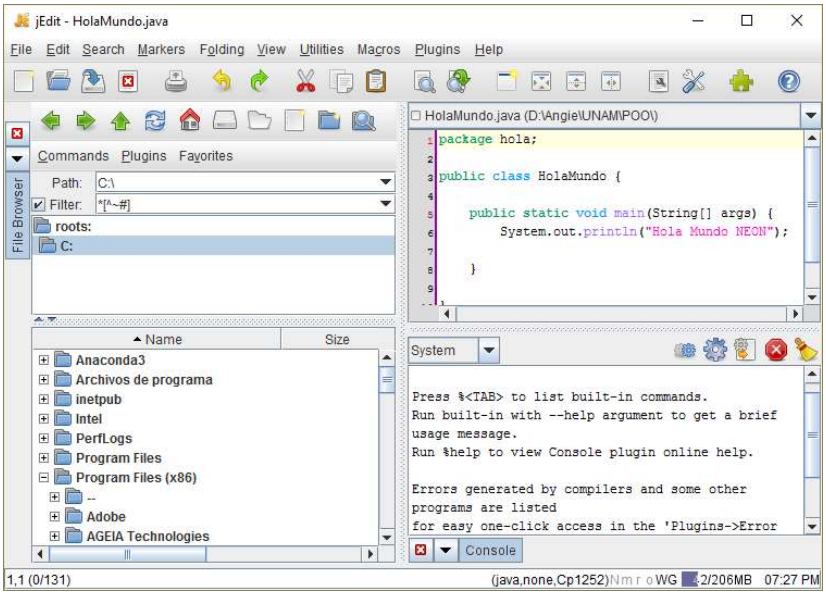
## Eclipse



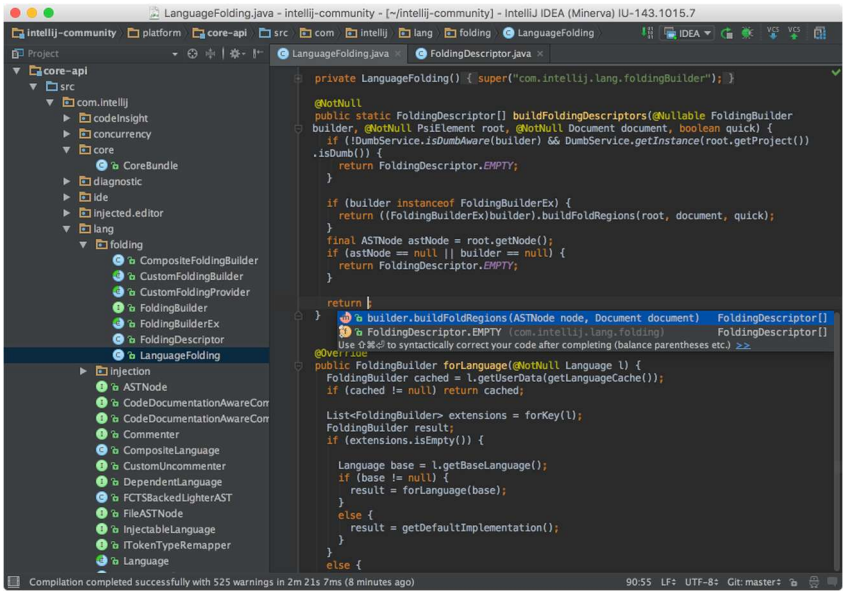

## NetBeans



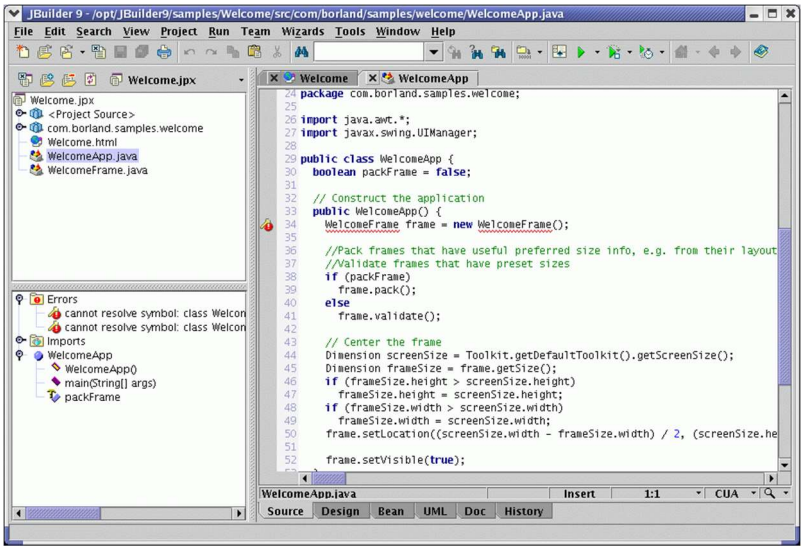
## jEdit



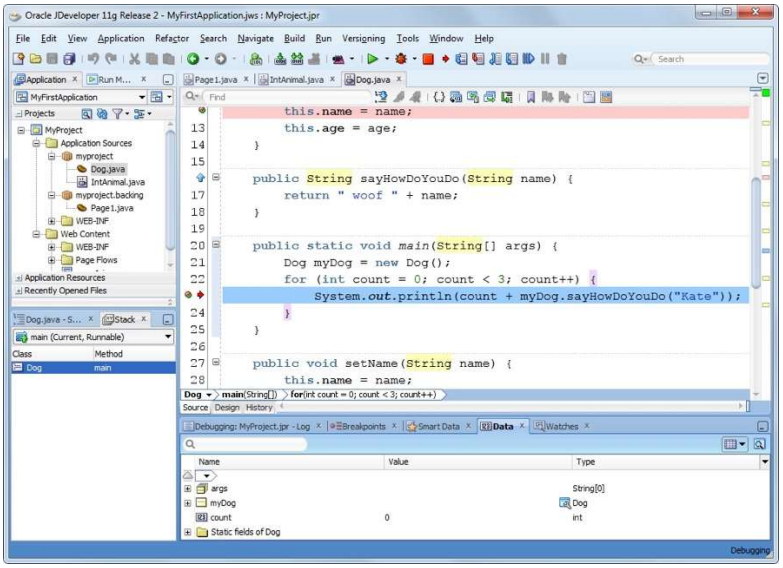

## IntelliJ IDEA



## JBuilder




## JDeveloper




Queda a juicio del programador elegir si utiliza un IDE o no y en caso de hacerlo, también decidir cuál de acuerdo a sus necesidades y gustos.

## Estructura general de un programa Java

Una de las principales características de Java es que es un lenguaje totalmente orientado a objetos. Como tal, todo programa Java debe estar escrito en una o varias **clases**, dentro de las cuales se podrá hacer uso del amplio conjunto de paquetes y clases prediseñadas.

Un programa en Java consta de una **clase** principal (que contiene el método **main)** y algunas clases de usuario (las específicas de la aplicación que se está desarrollando) que son utilizadas por el programa o clase principal.

La clase principal debe ser declarada con el modificador de acceso **public** y la palabra reservada **class**, seguida del nombre de la clase iniciando con mayúscula.

Un archivo fuente **(*.java**) puede contener más de una clase, pero sólo una puede ser **public**. El nombre del archivo fuente debe **coincidir** con el de la clase **public** (con la extensión *.java).

Ejemplo:

```
public class MiClase {
    public static void main(String[] args) {
        System.out.println("Esta es mi clase");
    }
}
```

El nombre del archivo tiene que ser **exactamente el mismo** que el de la clase, en este caso debe ser **MiClase.java**. Es importante que coincidan mayúsculas y minúsculas ya que *MiClase.java* y *miclase.java* serían clases diferentes para Java.

Normalmente, una aplicación está constituida por varios archivos ***.class**. Cada clase realiza funciones particulares, permitiendo construir las aplicaciones con gran modularidad e independencia entre clases. La aplicación se ejecuta por medio del nombre de la clase que contiene el método **main**( ) (sin la extensión *.class), invocando a la máquina virtual. Para el ejemplo:


El método main

En la clase principal debe existir una **función o método** estático llamado **main** cuyo formato debe ser:

*public static void main(String[ ] args)*

El método **main** debe cumplir las siguientes características:

- Debe ser un método púbico (**public**).
- Debe ser un método estático (**static**).
- No puede devolver ningún resultado (tipo de devolución **void**).
- Debe declarar un arreglo de cadenas de caracteres en la lista de parámetros o un número variable de argumentos).

El método **main** es el punto de arranque de un programa Java, cuando se invoca al comando **java** desde la línea de comandos, la **JVM** busca en la clase indicada un método estático llamado **main** con el formato indicado.

Dentro del código de **main** pueden crearse objetos de otras clases e invocar sus métodos. En general, se puede incluir cualquier tipo de lógica que respete la sintaxis y estructura de java.

# Bibliografía


*Martín, Antonio*
***Programador Certificado Java 2.***
*Segunda Edición.*
*México*
*Alfaomega Grupo Editor, 2008*

*Sierra Katy, Bates Bert*
***SCJP Sun Certified Programmer for Java 6 Study Guide***
*Mc Graw Hill*

*Dean John, Dean Raymond.*
***Introducción a la programación con Java***
*Primera Edición.*
*México*
*Mc Graw Hill, 2009*

# Guía práctica de estudio 02: Abstracción / modelado


***Elaborado por:***

M.C. M. Angélica Nakayama C.

Ing. Jorge A. Solano Gálvez

***Autorizado por:***

M.C. Alejandro Velázquez Mena

# Guía práctica de estudio 02: Abstracción / modelado

## Objetivo:

Obtener las características principales de un objeto a modelar, para plasmarlo en una clase a través de atributos y métodos.

## Actividades:

- Modelar un objeto, abstrayendo sus principales características.
- Implementar la clase modelada.

## Introducción

La programación orientada a objetos se basa en el hecho de que se debe dividir el programa, no en tareas, si no en modelos de objetos físicos o simulados. Si se escribe un programa en un lenguaje orientado a objetos, se está creando un modelo de alguna parte del mundo, esto es, se expresa un programa como un conjunto de objetos que colaboran entre ellos para realizar tareas.

Un **objeto** es, por tanto, la representación en un programa de un concepto, y contiene toda la información necesaria para abstraerlo: datos que describen sus atributos y operaciones que pueden realizarse sobre los mismos.

Los objetos pueden agruparse en categorías y una **clase** describe (de un modo abstracto) todos los objetos de un tipo o categoría determinada.

**NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.**

## Objeto

La idea fundamental de la programación orientada a objetos y de los lenguajes que implementan este paradigma de programación es combinar (encapsular) en una única unidad tanto los datos como las funciones que operan (manipulan) sobre los datos. Esta unidad de programación se denomina **objeto**.

Entonces, un **objeto** es una encapsulación genérica de datos y de los procedimientos para manipularlos. En otras palabras, un objeto no es más que un conjunto de **atributos** (variables o datos) y **métodos** (o funciones) relacionados entre sí.

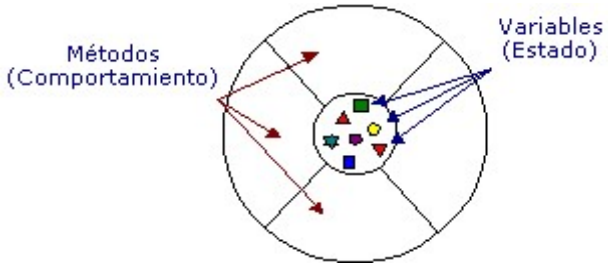



**Figura 1.** Conceptualización de un objeto en POO.

El **objeto** es el centro de la programación orientada a objetos. Un objeto es algo que se visualiza, se utiliza y que juega un papel o un rol en el dominio del problema del programa. La estructura interna y el comportamiento de un objeto, en consecuencia, no es prioritario durante el modelado del problema.

## Clase

En el mundo real existen varios objetos de un mismo tipo, o de una misma **clase**, por lo que una **clase** equivale a la generalización de un tipo específico de objetos. Así, la **clase** es una plantilla que define las variables y los métodos que son comunes para todos los objetos de un cierto tipo.

Por lo tanto, una **clase** es la implementación de un tipo abstracto de datos y describe no solo los **atributos** (datos) de un objeto sino también sus **operaciones** (comportamiento).

En Java para definir una clase se utiliza la palabra reservada **class** seguida del nombre de la clase.

***class*** *NombreDeLaClase*

## Instancia

Una vez definida la clase se pueden crear objetos a partir de ésta, a este proceso se le conoce como **crear instancias** de una clase o **instanciar** una clase. En este momento el sistema reserva suficiente memoria para el objeto con todos sus atributos.

Una **instancia** es un elemento de una clase (un objeto). Cada uno de los objetos o instancias tiene su propia copia de las variables definidas en la clase de la cual son instanciados y comparten la misma implementación de los métodos. Sin embargo, cada objeto asigna valores a sus atributos y es totalmente independiente de los demás.

En Java para crear una instancia se utiliza el operador **new** seguido del nombre de la clase y un par de paréntesis.

***new*** *NombreDeLaClase( );*

## Mensajes

Normalmente un único objeto por sí solo no es muy útil. Los objetos interactúan enviándose **mensajes** unos a otros. Tras la recepción de un **mensaje** el objeto actuará.

La acción puede ser el envío de otros mensajes, el cambio de su estado, o la ejecución de cualquier otra tarea que se requiera que haga el objeto. Los objetos de un programa interactúan y se **comunican** entre ellos por medio de **mensajes**.

Cuando un objeto A quiere que otro objeto B ejecute uno de sus métodos, el objeto A manda un mensaje al objeto B.

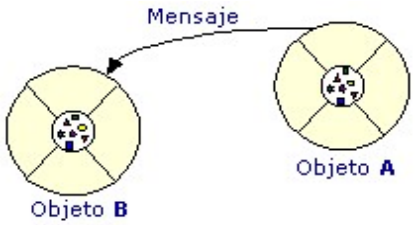



**Figura 2.** Mensajes entre objetos.

En ocasiones, el objeto que recibe el mensaje necesita más información para saber exactamente lo que tiene que hacer. Esta información se pasa junto con el mensaje en forma de parámetro.

En Java para que un objeto ejecute algún método se utiliza el operador punto:

*objeto.nombreDelMetodo( parametros );*

## Métodos

Los **métodos** especifican el comportamiento de la clase y sus instancias. En el momento de la declaración hay que indicar cuál es el tipo del parámetro que devolverá el método o *void* en caso de que no devuelva nada.

También se debe especificar el tipo y nombre de cada uno de los **parámetros** o **argumentos** del método entre paréntesis. Si un método no tiene parámetros el paréntesis queda vacío (no es necesario escribir *void*). Los parámetros de los métodos son variables locales a los métodos y existen sólo en el interior de estos.

Los argumentos pueden tener el mismo nombre que los atributos de la clase, de ser así, los argumentos *"ocultan"* a los atributos. Para acceder a los atributos en caso de ocultación se referencian a partir del objeto actual ***this***.

En Java se declara un método de manera similar a las funciones en C:

*tipoDato nombreDelMetodo( tipoDato parametro1, tipoDato parametro2, …)*

Ejemplo:

Se crea una clase Punto para representar un punto en el espacio 2D, cuyos atributos sean las coordenadas (x, y) y contiene un método que imprima los valores de dicha coordenada del punto.

```
public class Punto {
    int x,y;
    public void imprimePunto() {
        System.out.println("Punto [x=" + x + ", y=" + y + "]");
    }
}
```

Posteriormente se crea una clase de prueba para interactuar con la clase Punto. Dentro de esta clase se tiene un método *main* y se crean 2 instancias de la clase Punto. Una vez creadas se da valor a sus atributos y se manda ejecutar su método.

```
public class PruebaPunto {

    public static void main(String[] args) {
        Punto p = new Punto();
        p.x=5;
        p.y=8;
        p.imprimePunto();

        Punto x  = new Punto();
        x.x=7;
        x.x=2;
        x.imprimePunto();
    }

}
```

## Métodos de clase (static)

También puede haber métodos que no actúen sobre objetos concretos a través del operador punto. A estos métodos se les llama **métodos de clase** o static, estos pueden recibir argumentos, pero no pueden utilizar la referencia *this*.

Un ejemplo típico son los métodos matemáticos de la clase *java.lang.Math* (sin( ), cos( ), exp( ), pow( ), etc.).

Los métodos y variables de clase se crean anteponiendo la palabra **static**. Para llamarlos se suele utilizar el nombre de la clase, en vez del nombre de un objeto de la clase.

Ejemplo:

```
public class Circulo {
        static float PI = 3.14159f;
        private float radio;
        …
}
```

En otra clase se puede usar el método o variable:

*System.out.println(**Circulo.PI**);*

Los atributos static tendrán el mismo valor para todos los objetos que se creen de esa clase. Es decir si se modifica el valor de un atributo static, el cambio afectará a todos los objetos de esa clase.

## Constructores

Un **constructor** es un método que tiene el mismo nombre que la clase y cuyo propósito es **inicializar** los atributos de un nuevo objeto. **Se ejecuta automáticamente cuando se crea un objeto o instancia de la clase.**

Dependiendo del número y tipos de los argumentos proporcionados se llama al constructor correspondiente.

Si no se ha escrito un constructor en la clase, el compilador proporciona un **constructor por defecto**, el cual no tiene parámetros e inicializa los atributos a su valor por defecto.

Las reglas para los constructores son:

- El constructor tiene el mismo nombre que la clase.
- Puede tener cero o más parámetros.
- No devuelve ningún valor (ni si quiera void).
- Toda clase debe tener al menos un constructor.

Cuando se define un objeto se pasan los valores de los parámetros al constructor utilizando una sintaxis similar a una llamada normal a un método.

Ejemplo:

Para la clase Punto se declara un constructor que reciba los valores de las coordenadas (x, y) y dichos valores se le asignan a los atributos correspondientes (x, y).

```
public class Punto {

    int x,y;

    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void imprimePunto() {
        System.out.println("Punto [x=" + x + ", y=" + y + "]");
    }
}
```

Al hacer esto, dado que ya se cuenta con un constructor que recibe dos parámetros, Java deja de agregar el **constructor por defecto** (el constructor sin parámetros) por lo cual la clase de prueba ahora marcará error.

```
public class PruebaPunto {

    public static void main(String[] args) {
        Punto p = new Punto();
        p.x=5;
        p.y=8;
        p.imprimePunto();

        Punto x = new Punto();
        x.x=7;
        x.y=2;
        x.imprimePunto();
    }

}
```

En este caso se pueden hacer dos cosas:

- Cambiar la creación de las instancias para que ahora reciban los valores (x, y) desde el momento de su creación.

```
public class PruebaPunto {

    public static void main(String[] args) {
        Punto p = new Punto(5, 8);
        p.imprimePunto();

        Punto x = new Punto(7, 2);
        x.imprimePunto();
    }

}
```

- Agregar un constructor por defecto (sin parámetros) el cual puede inicializar los valores a un valor por defecto (que se desee) o bien puede estar sin implementación (código).

```
public class Punto {

    int x,y;

    public Punto(){

    }
    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void imprimePunto() {
        System.out.println("Punto [x=" + x + ", y=" + y + "]");
    }
}
```

En este último caso, dado que se cuenta con 2 constructores distintos (diferenciados por el número y tipo de parámetros) se tendrían disponibles las dos formas de crear instancias, ya sea con los valores por defecto o pasándolos como parámetros.

```
public class PruebaPunto {

    public static void main(String[] args) {
        Punto p = new Punto(5, 8);
        p.imprimePunto();

        Punto x = new Punto();
        x.x=7;
        x.y=2;
        x.imprimePunto();
    }

}
```

## Destrucción de objetos (liberación de memoria)

Como los objetos se asignan **dinámicamente**, cuando estos objetos se destruyen será necesarios verificar que la memoria ocupada por ellos ha quedado liberada para usos posteriores. El procedimiento es distinto según el tipo de lenguaje utilizado. Por ejemplo, en C++ los objetos asignados dinámicamente se deben liberar utilizando un operador *delete* y en Java y C# se hace de modo automático utilizando una técnica conocida como **recolección de basura** (garbage collection).

Cuando no existe ninguna referencia a un objeto se supone que ese objeto ya no se necesita y la memoria ocupada por ese objeto puede ser recuperada (liberada), entonces el sistema se ocupa automáticamente de liberar la memoria.

Sin embargo, no se sabe exactamente cuándo se va a activar el **garbage collector**, si no falta memoria es posible que no se llegue a activar en ningún momento. Por lo cual no es conveniente confiar en él para la realización de otras tareas más críticas.

Java no soporta destructores pero existe el método *finalize( )* que es lo más aproximado a un destructor. Las tareas dentro de este método serán realizadas cuando el objeto se destruya y se active el **garbage collector**.

# UML

El **Lenguaje de Modelado Unificado** (**UML** - Unified Modeling Language) es un lenguaje gráfico que permite visualizar, especificar y documentar cada una de las partes que comprende el desarrollo de software.

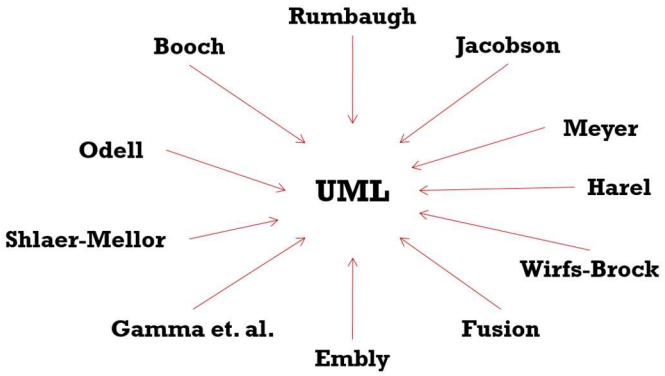



**Figura 1.** Representación de los lenguajes que componen a UML.

Los **diagramas UML** permiten modelar aspectos conceptuales como procesos de negocios o funcionalidades de un sistema, cumpliendo con los siguientes objetivos:

- *Visualizar:* expresar de forma gráfica la solución y/o flujo del proceso o sistema.
- *Especificar:* mostrar las características del sistema.
- *Construir:* generar soluciones de software.
- *Documentar:* especificar la solución implementada.

**UML** está compuesto por tres bloques generales de formas:

- *Elementos:* son representaciones de entes reales (usuarios) o abstractos (objetos, acciones, clases, etc.).
- *Relaciones:* es la unión e interacción entre los diferentes elementos.
- *Diagramas:* muestra a todos los elementos con sus relaciones.

## Diagrama de clases

Un **diagrama de clases** permite modelar las características de las clases que componen al sistema. Dentro de cada clase se pueden visualizar los atributos y métodos que contiene la clase. El conjunto de clases permite observar las relaciones que existen entre ellas dentro del sistema.

En UML una clase se representa de manera gráfica con 3 rectángulos dispuestos de manera vertical uno debajo de otro. En el primero se anota el nombre de la clase, en el segundo los atributos y en el tercero las operaciones, es decir:

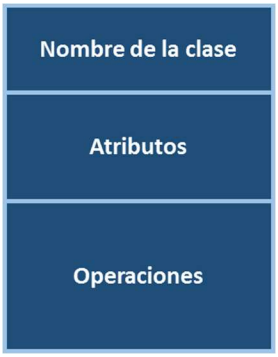



**Figura 3.** Diagrama de clase en UML.

Los diagramas de clase están compuestos por 3 elementos básicos:

- Clase(s)
- Cardinalidad
- Relación(es)

Los diagramas de clase permiten visualizar un **panorama general** del sistema, así como de la **comunicación** que se requiere entre las diferentes clases.

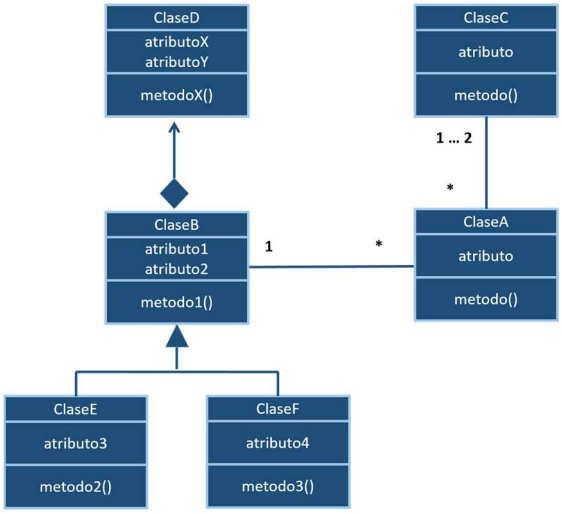



**Figura 4.** Diagrama de clases.

# Bibliografía


*Sierra Katy, Bates Bert*
***SCJP Sun Certified Programmer for Java 6 Study Guide***
*Mc Graw Hill*

*Martín, Antonio*
***Programador Certificado Java 2.***
*Segunda Edición.*
*México*
*Alfaomega Grupo Editor, 2008*

*Joyanes, Luis*
***Fundamentos de programación. Algoritmos, estructuras de datos y objetos.***
*Cuarta Edición*
*México*
*Mc Graw Hill, 2008*

# Guía práctica de estudio 03: Fundamentos y sintaxis del lenguaje


***Elaborado por:***

M.C. M. Angélica Nakayama C.

Ing. Jorge A. Solano Gálvez

***Autorizado por:***

M.C. Alejandro Velázquez Mena

# Guía práctica de estudio 03: Fundamentos y sintaxis del lenguaje

## Objetivo:

Implementar programas utilizando:

- Diversos tipos de datos.
- Expresiones (operadores, declaraciones, etc.)

## Actividades:

- Utilizar diferentes tipos de datos en un lenguaje de programación.
- Implementar expresiones entre diferentes tipos de datos.

## Introducción

Los lenguajes de programación tienen elementos básicos que se utilizan como bloques constructivos, así como reglas para que esos elementos se combinen. Estas reglas se denominan **sintaxis del lenguaje**. Solamente las instrucciones sintácticamente correctas pueden ser interpretadas por la computadora. Los programas que contengan errores de sintaxis son rechazados por la máquina.

La **sintaxis** de un lenguaje de programación se define como el conjunto de reglas que deben seguirse al escribir el código fuente de los programas para considerarse como correctos para ese lenguaje de programación.

Los elementos básicos constructivos de un programa son:

- Palabras reservadas (propias de cada lenguaje).
- Identificadores (nombres de variables, nombres de funciones, nombre del programa, etc.)
- Caracteres especiales (alfabeto, símbolos de operadores, delimitadores, comentarios, etc.)
- Expresiones.
- Instrucciones.

**NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.**

## Sintaxis básica

Una de las primeras cosas que hay que tener en cuenta es que Java es un lenguaje **sensitivo** a mayúsculas/minúsculas. El compilador Java hace distinción entre mayúsculas y minúsculas, esta distinción no solo se aplica a palabras reservadas del lenguaje sino también a nombres de variables y métodos.

La sintaxis de Java es muy parecida a la de C y C++, por ejemplo, las sentencias finalizan con '**;**', los bloques de instrucciones se delimitan con llaves **{** y **}**, etc. A continuación, se explican los puntos más relevantes de la sintaxis de Java.

## Comentarios

Los comentarios son muy útiles para poder entender el código utilizado, facilitando de ese modo futuras revisiones y correcciones. Además, permite que cualquier persona distinta al programador original pueda comprender el código escrito de una forma más rápida. Se recomienda acostumbrarse a comentar el código desarrollado. De esta forma se simplifica también la tarea de estudio y revisión posteriores.

En Java existen tres tipos de comentarios:

- Comentarios de una sola línea.

  *// Esta es una línea comentada.*

- Comentarios de bloques.

  */* Aquí el comentario por bloque*

  *y puede abarcar diferentes renglones */*

- Comentarios de documentación (**JavaDoc**).

  */** Los comentarios de documentación se realizan de este modo */*

## Identificadores

En Java los identificadores comienzan por una letra del alfabeto inglés, un guión bajo _ o el símbolo de dólar **$**, los siguientes caracteres del identificador pueden ser letras o dígitos (0-9). No se debe iniciar con un digito. No hay un límite en lo concerniente al número de caracteres que pueden tener los identificadores.

Las reglas del lenguaje respecto a los nombres de variables son muy amplias y permiten mucha libertad al programador, pero es habitual seguir ciertas normas que facilitan la lectura y el mantenimiento de los programas.

En Java es habitual utilizar nombres con minúsculas, con las excepciones que se indican en los puntos siguientes:

- Los nombres de **clases** e **interfaces** comienzan siempre por mayúscula. Ejemplos: *Geometria*, *Rectangulo*, *Dibujable*, *Graphics*, *Iterator*.
- Cuando un nombre consta de varias palabras se declaran una a continuación de otra, poniendo con mayúscula la primera letra de la palabra que sigue (**CammelCase**). Ejemplos: *elMayor( )*, *VentanaCerrable*, *RectanguloGrafico*, *addWindowListener( )*.
- Los nombres de **objetos**, **métodos, variables miembro** y **variables locales** de los métodos, comienzan siempre por minúscula. Ejemplos: *main( )*, *dibujar( )*, *numRectangulos*, *x*, *y*, *r*.
- Los nombres de las **variables finales**, es decir de las **constantes**, se definen siempre con mayúsculas. Ejemplo: *PI*

Sin embargo, éstas no son las únicas normas para la nomenclatura en Java, también existen las **convenciones de código**, las cuales son importantes para los programadores dado que mejoran la lectura del programa, permitiendo entender código nuevo mucho más rápidamente y más a fondo.

Para que funcionen, todos los programadores deben seguir la convención, por esta razón, es muy importante generar el hábito de aplicar convenciones y estándares de programación ya que de esta manera no solo se enfoca en las funcionalidades, sino que también se aporta a la calidad de los desarrollos.

Las convenciones de código o **Java Code Conventions**, pueden ser consultadas en el siguiente enlace: http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

### *Palabras reservadas*

Las siguientes son palabras reservadas utilizadas por Java y no pueden ser usadas como identificadores.

| | | | | | |
|---|---|---|---|---|---|
| abstract | boolean | break | byte | case | catch |
| char | class | const | continue | default | do |
| double | else | extends | final | finally | float |
| for | goto | if | implements | import | instanceof |
| int | interface | long | native | new | package |
| private | protected | public | return | short | static |
| strictfp | super | switch | synchronized | this | throw |
| throws | transient | try | void | volatile | while |
| assert | enum | | | | |

También existen las literales reservadas *null*, *true* y *false*, las cuales tampoco pueden ser usadas como identificadores.

## Tipos de datos

En Java existen dos grupos de tipos de datos: tipos primitivos y tipos referencia.

### *Tipos de dato primitivos*

Se llaman **tipos primitivos** a aquellos datos sencillos que contienen los tipos de información más habituales: valores booleanos, caracteres y valores numéricos enteros o de punto flotante.

Java dispone de ocho tipos primitivos:

| Tipo | Definición |
|---|---|
| boolean | `true o false` |
| char | Carácter Unicode de 16 bits |
| byte | Entero en complemento a dos con signo de 8 bits |
| short | Entero en complemento a dos con signo de 16 bits |
| int | Entero en complemento a dos con signo de 32 bits |
| long | Entero en complemento a dos con signo de 64 bits |
| float | Real en punto flotante según la norma IEEE 754 de 32 bits |
| double | Real en punto flotante según la norma IEEE 754 de 64 bits |

En Java al contrario que en C o C++ el tamaño de los tipos primitivos no depende del sistema operativo o de la arquitectura ya que en todas las arquitecturas y bajo todos los sistemas operativos el tamaño en memoria es el mismo.

Es posible recubrir los tipos primitivos para tratarlos como cualquier otro objeto en Java. Así, por ejemplo, existe una clase envoltura del tipo primitivo **int** llamado **Integer**. La utilidad de estas clases se explicará en otro momento.

### *Tipos de dato referencia*

Los **tipos de dato referencia** representan datos compuestos o estructuras, es decir, referencias a objetos. Estos tipos de dato almacenan las direcciones de memoria y no el valor en sí (similares a los apuntadores en C). Una referencia a un objeto es la dirección de un área en memoria destinada a representar ese objeto.

## Entrada y salida de datos por consola

Una de las operaciones más habituales que tiene que realizar un programa es intercambiar datos con el exterior. Para ello la **API** de java incluye una serie de clases que permiten gestionar la entrada y salida de datos en un programa, independientemente de los dispositivos utilizados para el envío/recepción de datos.

Para el envío de datos al exterior se utiliza un flujo de datos de impresión o **print stream**. Esto se logra usando la siguiente expresión:

```
System.out.println("Mi mensaje");
```

De manera análoga, para la recepción o lectura de datos desde el exterior se utiliza un flujo de datos de entrada o **input stream**. Para lectura de datos se utiliza la siguiente sintaxis:

```
Scanner sc = new Scanner(System.in);

String s = sc.next( );     //Para cadenas

int x = sc.nextInt( );     //Para enteros
```

Para usar la clase Scanner se debe incluir al inicio del archivo la siguiente línea:

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

Al finalizar su uso se debe cerrar el flujo utilizando el método **close**. Para este ejemplo:

```
sc.close( );
```

# Operadores

Java es un lenguaje rico en operadores, los cuales son casi idénticos a los de C/C++.

### *Operadores aritméticos:*

Son operadores binarios (requieren siempre dos operandos) que realizan las operaciones aritméticas habituales: suma (**+**), resta (**-**), multiplicación (*), división (**/**) y resto de la división o módulo (%).

### *Operadores de asignación*

Los operadores de asignación permiten asignar un valor a una variable. El operador de asignación por excelencia es el operador igual (**=**). La forma general de las sentencias de asignación con este operador es:

*variable = expression;*

Java dispone de otros operadores de asignación. Se trata de versiones abreviadas del operador (**=**) que realizan operaciones "acumulativas" sobre una variable.

| Operador | Utilización | Expresión equivalente |
|---|---|---|
| += | op1 += op2 | op1 = op1 + op2 |
| -= | op1 -= op2 | op1 = op1 - op2 |
| *= | op1 *= op2 | op1 = op1 * op2 |
| /= | op1 /= op2 | op1 = op1 / op2 |
| %= | op1 %= op2 | op1 = op1 % op2 |

### *Operadores unarios*

Los operadores más (**+**) y menos (**-**) unarios sirven para mantener o cambiar el signo de una variable, constante o expresión numérica. Su uso en Java es el estándar de estos operadores.

### *Operador instanceof*

El operador **instanceof** permite saber si un objeto **pertenece o no** a una determinada clase. Es un operador binario cuya forma general es:

*objectName* ***instanceof*** *ClassName*

Este operador devuelve **true** si el objeto pertenece a la clase o **false** en caso contrario.

### *Operador condicional*

El operador condicional ?, tomado de C/C++, permite realizar bifurcaciones condicionales sencillas. Su forma general es la siguiente:

*booleanExpression ? res1 : res2*

Donde se evalúa **booleanExpression** y se devuelve **res1** si el resultado es **true** y **res2** si el resultado es **false**. Es el único operador ternario (tres argumentos) de Java. Como todo operador que devuelve un valor puede ser utilizado en una expresión.

### *Operadores incrementales*

Java dispone del operador incremento (**++**) y decremento (**--**). El operador (**++**) incrementa en una unidad la variable a la que se aplica, mientras que (**--**) la reduce en una unidad. Estos operadores se pueden utilizar de dos formas:

- Precediendo a la variable (por ejemplo: **++i**). En este caso primero se incrementa la variable y luego se utiliza (ya incrementada) en la expresión en la que aparece.
- Siguiendo a la variable (por ejemplo: **i++**). En este caso primero se utiliza la variable en la expresión (con el valor anterior) y luego se incrementa.

La actualización de contadores en ciclos **for** es una de las aplicaciones más frecuentes de estos operadores.

### *Operadores relacionales*

Los operadores relacionales sirven para realizar **comparaciones** de igualdad, desigualdad y relación de menor o mayor. El resultado de estos operadores es siempre un valor **boolean** (**true** o **false**) según se cumpla o no la relación considerada.

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

Estos operadores se utilizan con mucha frecuencia en las estructuras de control.

### *Operadores lógicos*

Los operadores lógicos se utilizan para construir **expresiones lógicas**, combinando valores lógicos (**true** y/o **false**) o los resultados de los operadores relacionales.

| Operador | Nombre | Utilización | Resultado |
|---|---|---|---|
| && | AND | op1 && op2 | true si op1 y op2 son true. Si op1 es false ya no se evalúa op2 |
| \|\| | OR | op1 \|\| op2 | true si op1 u op2 son true. Si op1 es true ya no se evalúa op2 |
| ! | negación | ! op | true si op es false y false si op es true |
| & | AND | op1 & op2 | true si op1 y op2 son true. Siempre se evalúa op2 |
| \| | OR | op1 \| op2 | true si op1 u op2 son true. Siempre se evalúa op2 |

Debe notarse que en ciertos casos el segundo operando no se evalúa porque ya no es necesario (si ambos tienen que ser true y el primero es false, ya se sabe que la condición de que ambos sean true no se va a cumplir). Esto puede traer resultados no deseados y por eso se han añadido los operadores (&) y (|) que garantizan que los dos operandos se evalúan siempre.

### *Operador de concatenación de cadenas de caracteres*

El operador más (**+**) se utiliza también para concatenar cadenas de caracteres. Por ejemplo, para escribir una cantidad con un rótulo y valores puede utilizarse la sentencia:

*System.out.println("El total asciende a " + result + " unidades");*

Donde el operador de concatenación se utiliza dos veces para construir la cadena de caracteres que se desea imprimir por medio del método println( ). La variable numérica result es convertida automáticamente por Java en cadena de caracteres para poderla concatenar. En otras ocasiones se deberá llamar explícitamente a un método para que realice esta conversión.

### *Operadores a nivel de bits*

Java dispone también de un conjunto de operadores que actúan a nivel de bits. Las operaciones de bits se utilizan con frecuencia para definir señales o flags, esto es, variables de tipo entero en las que cada uno de sus bits indica si una opción está activada o no.

| Operador | Utilización | Resultado |
|---|---|---|
| >> | op1 >> op2 | Desplaza los bits de op1 a la derecha una distancia op2 |
| << | op1 << op2 | Desplaza los bits de op1 a la izquierda una distancia op2 |
| >>> | op1 >>> op2 | Desplaza los bits de op1 a la derecha una distancia op2 (positiva) |
| & | op1 & op2 | Operador AND a nivel de bits |
| \| | op1 \| op2 | Operador OR a nivel de bits |
| ^ | op1 ^ op2 | Operador XOR a nivel de bits (1 si sólo uno de los operandos es 1) |
| ~ | ~op2 | Operador complemento (invierte el valor de cada bit) |

| Operador | Utilización | Equivalente a |
|---|---|---|
| &= | op1 &= op2 | op1 = op1 & op2 |
| \|= | op1 \|= op2 | op1 = op1 \| op2 |
| ^= | op1 ^= op2 | op1 = op1 ^ op2 |
| <<= | op1 <<= op2 | op1 = op1 << op2 |
| >>= | op1 >>= op2 | op1 = op1 >> op2 |
| >>>= | op1 >>>= op2 | op1 = op1 >>> op2 |

### *Precedencia de operadores*

El **orden** en que se realizan las operaciones es fundamental para determinar el resultado de una expresión. La siguiente lista muestra el orden en que se ejecutan los distintos operadores en una sentencia, de mayor a menor precedencia:

| | |
|---|---|
| **postfix operators** | [ ] . (params) expr++ expr-- |
| **unary operators** | ++expr --expr +expr -expr ~ ! |
| **creation or cast** | new (type)expr |
| **multiplicative** | * / % |
| **additive** | + - |
| **shift** | << >> >>> |
| **relational** | < > <= >= instanceof |
| **equality** | == != |
| **bitwise AND** | & |
| **bitwise exclusive OR** | ^ |
| **bitwise inclusive OR** | \| |
| **logical AND** | && |
| **logical OR** | \|\| |
| **conditional** | ? : |
| **assignment** | = += -= *= /= %= &= ^= \|= <<= >>= >>>= |

En Java, todos los operadores binarios (excepto los operadores de asignación) se evalúan de **izquierda a derecha**. Los operadores de asignación se evalúan de derecha a izquierda, lo que significa que el valor de la derecha se copia sobre la variable de la izquierda.

# Bibliografía


*Martín, Antonio*
***Programador Certificado Java 2.***
*Segunda Edición.*
*México*
*Alfaomega Grupo Editor, 2008*

*Sierra Katy, Bates Bert*
***SCJP Sun Certified Programmer for Java 6 Study Guide***
*Mc Graw Hill*

*Dean John, Dean Raymond.*
***Introducción a la programación con Java***
*Primera Edición.*
*México*
*Mc Graw Hill, 2009*

# Guía práctica de estudio 04: Variables y arreglos


***Elaborado por:***

M.C. M. Angélica Nakayama C.

Ing. Jorge A. Solano Gálvez

***Autorizado por:***

M.C. Alejandro Velázquez Mena

# Guía práctica de estudio 04: Variables y arreglos

## Objetivo:

Utilizar variables y arreglos con tipos de datos primitivos y con bibliotecas propias del lenguaje.

## Actividades:

- Crear variables con diferentes tipos de datos.
- Crear arreglos con diferentes tipos de datos.

## Introducción

Los **tipos de datos** hacen referencia al tipo de información que se trabaja, donde la unidad mínima de almacenamiento es el dato, también se puede considerar como el rango de valores que puede tomar una variable durante la ejecución del programa. Un tipo de datos define un conjunto de valores y las operaciones sobre estos valores.

Casi todos los lenguajes de programación explícitamente incluyen la notación del tipo de datos, aunque lenguajes diferentes pueden usar terminologías diferentes. La mayor parte de los lenguajes de programación permiten al programador definir tipos de datos adicionales, normalmente combinando múltiples elementos de otros tipos y definiendo las operaciones del nuevo tipo de dato.

Los **valores** que pueden adquirir los **tipos de datos** se manipulan durante la ejecución de un programa a través de **variables** o **arreglos**.

**NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.**

## Variable

Una **variable** es un **nombre** que contiene un valor que **puede cambiar** a lo largo del programa. De acuerdo con el tipo de información que contienen, en Java hay dos tipos principales de variables:

1. Variables de tipos primitivos.
2. Variables de referencia.

Desde el punto de vista del papel o misión en el programa, las variables pueden ser:

- Variables **miembro** de una clase: Se definen en una clase, fuera de cualquier método; pueden ser tipos primitivos o referencias.
- Variables **locales**: Se definen dentro de un método o, de forma más general dentro de cualquier bloque entre llaves { }. Se crean en el interior del bloque y se destruyen al finalizar dicho bloque.

Una variable se define especificando el **tipo de dato** y el **nombre** de dicha variable. Las variables pueden ser tanto de tipos primitivos como referencias a objetos de alguna clase perteneciente al API de Java o creada por el usuario.

**tipoDeDato** *nombreVariable***;**

Ejemplos:

**int** *miVariable*;

**float** *area*;

**char** *letra*;

**String** *cadena*;

**MiClase** *prueba*;

Si no se especifica un valor en su declaración, las variables miembro primitivas se inicializan a **cero** (salvo boolean y char, que se inicializan a false y '\0', respectivamente). Así mismo,

las variables miembro de tipo referencia son inicializadas por defecto a un valor especial: **null**.

Es importante distinguir entre la referencia a un objeto y el objeto mismo. Una referencia es una variable que indica dónde está guardado un objeto en la memoria (a diferencia de C/C++, Java no permite acceder al valor de la dirección, pues en este lenguaje se han eliminado los apuntadores). Al declarar una referencia todavía no se encuentra "apuntando" a ningún objeto en particular (salvo que se cree explícitamente un nuevo objeto en la declaración), y por eso se le asigna el valor **null**.

Si se desea que esta referencia apunte a un nuevo objeto es necesario crear el objeto utilizando el operador **new**. Este operador reserva espacio en la memoria para el objeto (variables y funciones).

También es posible igualar la referencia declarada a otra referencia a un objeto existente previamente.

Ejemplo:

```
MyClass unaRef;
unaRef = new MyClass();
MyClass segundaRef = unaRef;
```

Un tipo particular de referencias son los **arrays** o **arreglos**, sean éstos de variables primitivas (por ejemplo, de enteros) o de objetos. En la declaración de una referencia de tipo **array** hay que incluir los **corchetes** [ ].

Ejemplo:

```
int [ ] vector;
vector = new int [10];
MyClass [ ] lista=new MyClass [5];
```

En Java todas las variables deben estar incluidas en alguna clase. En general las variables declaradas dentro de llaves **{ }**, es decir, dentro de un bloque, son visibles y existen dentro de estas llaves. Por ejemplo las variables declaradas al principio de un método existen mientras se ejecute el método; las variables declaradas dentro de un bloque if no serán válidas al finalizar las sentencias correspondientes a dicho if y las variables miembro de una

clase (es decir declaradas entre las llaves { } de la clase pero fuera de cualquier método) son válidas mientras existe el objeto de la clase.

## Constante

Una **constante** es una variable cuyo valor **no puede ser modificado**. Para definir una constante en Java se utiliza la palabra reservada **final**, antes de la declaración del tipo de dato, de la siguiente manera:

***final*** *tipoDato* nombreDeConstante = *valor;*

Ejemplo:

*final double PI = 3.1416;*

## Arreglo

Un **arreglo** es un objeto en el que se puede almacenar un conjunto de datos de un **mismo tipo**. Cada uno de los elementos del arreglo tiene asignado un **índice** numérico según su posición, siendo 0 el índice del primer elemento. Se declara de la siguiente manera:

*tipoDeDato [ ] nombreVariable;* o *tipoDeDato nombreVariable[ ];*

Como se puede apreciar, los corchetes pueden estar situados delante del nombre de la variable o detrás. Ejemplos:

*int [ ] k;* *String [ ] p;* *char datos[ ];*

Los arreglos pueden declararse en los mismos lugares que las variables estándar. Para asignar un tamaño al arreglo se utiliza la expresión:

*variableArreglo* = ***new*** *tipoDeDato[tamaño];*

También se puede asignar el tamaño al arreglo en la misma línea de declaración de la variable.

```
int [ ] k = new int[5];
```

Cuando un arreglo se dimensiona, todos sus elementos son inicializados explícitamente al **valor por defecto** del tipo correspondiente.

Para declarar, dimensionar e inicializar un arreglo en una misma sentencia se indican los valores del arreglo entre llaves y separados por comas. Ejemplo:

```
int [ ] nums = {10, 20, 30, 40};
```

El acceso a los elementos de un arreglo se realiza utilizando la expresión:

*variableArreglo[índice]*

Donde índice representa la posición a la que se quiere tener acceso y cuyo valor debe estar entre **0 y tamaño – 1**

Todos los objetos arreglo exponen un atributo publico llamado **length** que permite conocer el tamaño al que ha sido dimensionado un arreglo.

Ejemplo:

```
int [ ] nums = new int[10];
for(int i=0; i < nums.length; i++)
        nums[i] = i * 2;
```

Los arreglos al igual que las variables, se pueden usar como argumentos, así como también pueden ser devueltos por un método o función.

En Java se puede utilizar una variante del ciclo for llamado **for-each**, para facilitar el recorrido de arreglos y colecciones, recuperando su contenido y eliminando la necesidad de usar una variable de control que sirva de índice. Su sintaxis es:

```
for(tipoDato variable: variableArreglo){
        //instrucciones
}
```

Ejemplo:

```
int [ ] nums = { 4, 6 ,30, 15 };
for(int n : nums){
        System.out.println(n);
}
```

En este caso, sin acceder de forma explícita a las posiciones del arreglo, cada una de estas es copiada automáticamente a la variable auxiliar **n** al principio de cada iteración.

Los arreglos en Java también pueden tener más de una dimensión, al igual que en C/C++ para declarar un arreglo bidimensional ser tendrían que usar dos pares de corchetes y en general para cada dimensión se usa un nuevo par de corchetes.

Ejemplo:

```
int [ ] [ ] matriz;
```

## Argumentos por línea de comandos

Es posible suministrar parámetros al método **main** a través de la línea de comandos. Para ello, los valores a pasar deberán especificarse a continuación del nombre de la clase separados por un espacio:

```
>> java NombreClase arg1 arg2 arg3
```

Los datos llegarán al método **main** en forma de un arreglo de cadenas de caracteres.

Ejemplo:

```java
public class Ejemplo {

    public static void main(String[] args) {
        System.out.println(args[0]);
        System.out.println(args[1]);
        System.out.println(args[2]);
    }

}
```

Se ejecuta utilizando la siguiente expresión en la línea de comandos:


## API de JAVA

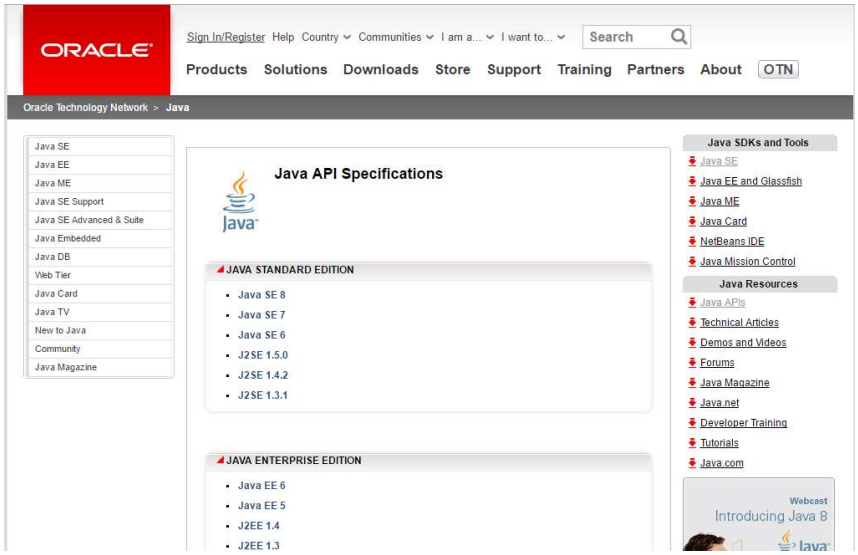
La API Java (**Application Programming Interface**) es una interfaz de programación de aplicaciones provista por los creadores del lenguaje, que da a los programadores los medios para desarrollar aplicaciones Java.

Como el lenguaje Java es un lenguaje orientado a objetos, la **API** de Java provee de un conjunto de clases utilitarias para efectuar toda clase de tareas necesarias dentro de un programa. La **API** Java está organizada en paquetes lógicos, donde cada paquete contiene un conjunto de clases relacionadas semánticamente.

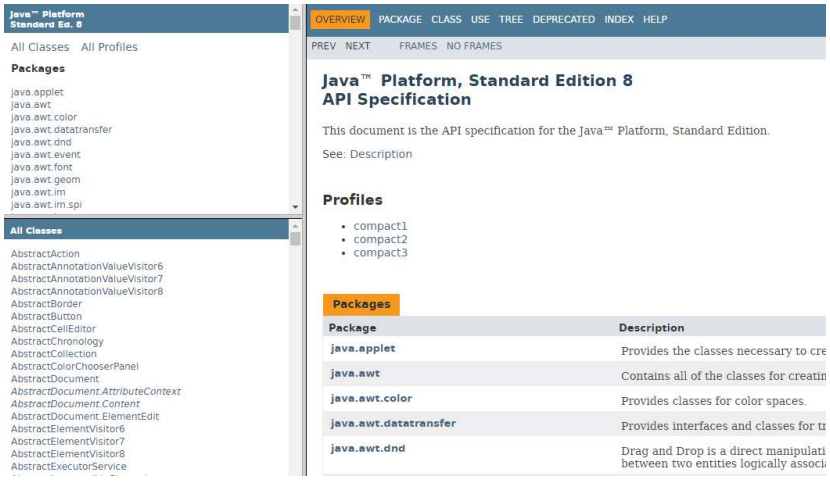
La información completa del **API** de java se denomina **especificación** y sirve para conocer cualquier aspecto sobre las clases que contiene la **API**. Esta especificación es de crucial importancia para los programadores ya que en ella se pueden consultar los detalles de alguna clase que se quiera utilizar y ya no sería necesario memorizar toda la información relacionada.

La especificación del **API** de java se encuentra en el sitio de Oracle: http://www.oracle.com/technetwork/java/api-141528.html.

En este sitio se puede consultar la especificación de las últimas versiones (ya que en cada versión se agregan o modifican algunas clases).




Una vez seleccionada la versión, se puede consultar el detalle de todas las clases que integran el API.



## Manejo de cadenas

En Java las cadenas de caracteres no son un tipo de datos primitivo, sino que son objetos pertenecientes a la clase **String**.

La clase **String** proporciona una amplia variedad de métodos que permiten realizar las operaciones de manipulación y tratamiento de cadenas de caracteres habituales en un programa.

Para crear un objeto **String** podemos seguir el procedimiento general de creación de objetos en Java, utilizando el operador **new**. Ejemplo:

*String s = new String("Texto de prueba");*

Sin embargo, dada la amplia utilización de estos objetos en un programa, Java permite crear y asignar un objeto String a una variable de la misma forma que se hace con los tipos de datos primitivos. Entonces el ejemplo anterior es equivalente a:

*String s = "Texto de prueba";*

Una vez creado el objeto y asignada la referencia al mismo a una variable, puede utilizarse para acceder a los métodos definidos en la clase String (se pueden revisar en la documentación del API com java.lang.String). Los más usados son: length, equals, charAt, substring, indexOf,replace, toUpperCase, toLowerCase, Split, entre otros.

Ejemplo:

```
s.length();          //Devuelve el tamaño de la cadena

s.toUpperCase();     //Devuelve la cadena en mayúsculas
```

Las variables de tipo String se pueden usar en una expresión que use el operador + para concatenar cadenas. Ejemplo:

*String s = "Hola";*

*String t = s + " qué tal";*

En Java las cadenas de caracteres son objetos **inmutables**, esto significa que una vez que el objeto se ha creado, no puede ser modificado.

Cuando escribimos una instrucción como la del ejemplo anterior, es fácil intuir que la variable s pasa a apuntar al objeto de texto *"Hola que tal"*. Aunque a raíz de la operación de concatenación pueda parecer que el objeto *"Hola"* apuntado por la variable **s** ha sido modificado en realidad no sucede esto, sino que al concatenarse *"Hola"* con *" que tal"* se está creando un nuevo objeto de texto *"Hola que tal"* que pasa a ser referenciado por la variable **s**. El objeto *"Hola"* deja de ser referenciado por dicha variable.

Java provee soporte especial para la concatenación de cadenas con las clases **StringBuilder** y **StringBuffer**. Un objeto **StringBuilder** es una secuencia de caracteres **mutable**, su contenido y capacidad puede cambiar en cualquier momento. Además, a diferencia de los **Strings**, los **builders** cuentan con una capacidad (capacity), la cantidad de espacios de caracteres asignados. Ésta es siempre mayor o igual que la longitud (length) y se expande automáticamente para acomodarse a más caracteres.

Los métodos principales de la clase **StringBuilder** son **append** e **insert**. Cada uno convierte un dato en **String** y concatena o inserta los caracteres de dicho String al **StringBuilder**. El método **append** agrega los caracteres al final mientras que **insert** los agrega en un punto específico.

Para hacer la misma concatenación que el ejemplo con String, quedaría:

*StringBuilder sb = new StringBuilder("Hola");*

*sb.append(" que tal");*

## Wrappers

Los **wrappers** o clases envoltorio son clases diseñadas para ser un complemento de los tipos primitivos. En efecto, los tipos primitivos son los únicos elementos de Java que no son objetos. Esto tiene algunas ventajas desde el punto de vista de la eficiencia, pero algunos inconvenientes desde el punto de vista de la funcionalidad.

Por ejemplo, los tipos primitivos siempre se pasan como argumento a los métodos por valor, mientras que los objetos se pasan por referencia. No hay forma de modificar en un método un argumento de tipo primitivo y que esa modificación se trasmita al entorno que hizo la llamada.

Una forma de conseguir esto es utilizar un **wrapper**, esto es un objeto cuya variable miembro es el tipo primitivo que se quiere modificar. Las clases **wrapper** también proporcionan métodos para realizar otras tareas con los tipos primitivos, tales como conversión con cadenas de caracteres en uno y otro sentido.

Existe una clase **wrapper** para cada uno de los tipos primitivos, las clases son: Byte, Short, Character, Integer, Long, Float, Double y Boolean (obsérvese que los nombres empiezan por mayúscula, siguiendo la nomenclatura típica de Java). Todas estas clases se encuentran en ***java.lang.***

Todas las clases **wrapper** permiten crear un objeto de la clase a partir de tipo básico.

```
int k =23;

Integer num = new Integer(k);
```

A excepción de Character, las clases **wrapper** también permiten crear objetos partiendo de la representación como cadena del dato.

```
String s = "4.65";

Float ft = new Float(s);
```

Para recuperar el valor a partir del objeto, las ocho clases **wrapper** proporcionan un método con el formato *xxxValue( )* que devuelve el dato encapsulado en el objeto donde xxx representa el nombre del tipo en el que se quiere obtener el dato.

```
float dato = ft.floatValue( );

int n=num.intValue( );
```

Las clases numéricas proporcionan un método estático *parseXxx(String)* que permite convertir la representación en forma de cadena de un numero en el correspondiente tipo numérico donde xxx es el nombre del tipo al que se va a convertir la cadena de caracteres.

```
String s1 = "25, s2="89.2";

int n = Integer.parseInt(s1);

double d = Double.parseDouble(s2);
```

## Auto boxing y unboxing

El **autoboxing** consiste en la encapsulación automática de un dato básico en un objeto wrapper mediante la utilización del operador de asignación.

Por ejemplo:

```
int p = 5;
Integer n = new Integer(p);
```

Equivale a:

```
int p = 5;
Integer n = p;
```

Es decir, la creación del objeto **wrapper** se produce implícitamente al asignar el dato primitivo a la variable objeto. De la misma forma, para obtener el dato básico a partir del objeto **wrapper** no será necesario recurrir al método *xxxValue( )*, esto se realizará implícitamente al utilizar la variable objeto en una expresión. A esto se le conoce como **autounboxing**. Para el ejemplo anterior:

```
int a = n;
```

# Bibliografía


*Martín, Antonio*
***Programador Certificado Java 2.***
*Segunda Edición.*
*México*
*Alfaomega Grupo Editor, 2008*

*Sierra Katy, Bates Bert*
***SCJP Sun Certified Programmer for Java 6 Study Guide***
*Mc Graw Hill*

*Dean John, Dean Raymond.*
***Introducción a la programación con Java***
*Primera Edición.*
*México*
*Mc Graw Hill, 2009*

# Guía práctica de estudio 05: Estructuras de selección


***Elaborado por:***

M.C. M. Angélica Nakayama C.

Ing. Jorge A. Solano Gálvez

***Autorizado por:***

M.C. Alejandro Velázquez Mena

# Guía práctica de estudio 05: Estructuras de selección

## Objetivo:

Implementar programas utilizando estructuras de selección en un lenguaje orientado a objetos.

## Actividades:

- Conocer la sintaxis para declarar diversas estructuras de selección.
- Implementar el uso de estructuras de selección en un programa.

## Introducción

Los lenguajes de programación tienen elementos básicos que se utilizan como bloques constructivos, así como reglas para que esos elementos se combinen. Estas reglas se denominan **sintaxis del lenguaje**. Solamente las instrucciones sintácticamente correctas pueden ser interpretadas por la computadora y los programas que contengan errores de sintaxis son rechazados por la máquina.

La **sintaxis** de un lenguaje de programación se define como el conjunto de reglas que deben seguirse al escribir el código fuente de los programas para considerarse como correctos para ese lenguaje de programación.

Las **estructuras de control** permiten modificar el flujo de ejecución de las instrucciones de un programa. Todas las estructuras de control tienen un único punto de entrada. Las estructuras de control se pueden clasificar en: secuenciales, transferencia de control e iterativas. Básicamente lo que varía entre las estructuras de control de los diferentes lenguajes es su sintaxis, cada lenguaje tiene una sintaxis propia para expresar la estructura.

**NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.**

# Estructuras de control

Las estructuras de programación o estructuras de control permiten **tomar decisiones** o **realizar un proceso repetidas veces**. En la mayoría de los lenguajes de programación, este tipo de estructuras son comunes en cuanto a concepto, aunque su sintaxis varía de un lenguaje a otro. La sintaxis de Java coincide prácticamente con la utilizada en C/C++, por lo que para un programador que maneja estos lenguajes no representa ninguna dificultad adicional.

### *Sentencias o expresiones*

Una expresión es un conjunto de variables unidos por operadores. Son órdenes que se envían a la computadora para que realice una tarea determinada. Una sentencia es una expresión que acaba en punto y coma (;). Se permite incluir varias sentencias en una línea, aunque lo habitual es utilizar una línea para cada sentencia.

Ejemplo:

```
i = 0; j = 5; x = i + j;      // Línea compuesta de tres sentencias
```

### *Estructuras de selección*

Las estructuras de selección o bifurcaciones permiten ejecutar una de entre varias acciones en función del valor de una expresión lógica o relacional. Se tratan de estructuras muy importantes ya que son las encargadas de controlar el **flujo de ejecución** de un programa.

Java posee las estructuras de selección: **if-else**, **operador ternario** y **switch**.

### *IF / IF-ELSE*

Esta estructura permite ejecutar un conjunto de sentencias en función del valor que tenga la expresión de comparación (se ejecuta si la expresión de comparación tiene valor **true**).

Las sentencias incluidas en el **else** se ejecutan en el caso de no cumplirse la expresión de comparación (**false**). Tiene la forma siguiente:

```
if (booleanExpression) {
        statements1;
} else {
        statements2;
}
```

```
/**
 * Se generan dos números aleatorios y se obtiene el valor absoulto
 * de la diferencia de los mismos.
 */
public class IfElse {
    public static void main (String []args){
        java.util.Random random = new java.util.Random();
        int one = random.nextInt();
        int two = random.nextInt();
        System.out.println("Los números generados son: ");
        System.out.println("one: " + one);
        System.out.println("two: " + two);
        if ((one-two) < 0)
            System.out.println("|" + one + " - " + two + "| = " + (one-two)*-1);
        else
            System.out.println("|" + one + " - " + two + "| = " + (one-two));
    }
}
```

Las llaves **{ }** sirven para agrupar en un bloque las sentencias que se han de ejecutar, y no son necesarias si sólo hay una sentencia dentro del bloque **if o else**.

Si se desea introducir más de una expresión de comparación se usa **if / else if**. Si la primera condición no se cumple, se compara la segunda y así sucesivamente. En el caso de que no se cumpla ninguna de las comparaciones se ejecutan las sentencias correspondientes al **else**.

```
if (booleanExpression1) {
        statements1;
} else if (booleanExpression2) {
        statements2;
} else if (booleanExpression3) {
        statements3;
} else {
        statements4;
}
```

```java
/**
 * Programa que genera un número aleatorio de 0 a 6, los cuales
 * tienen una correspondencia con un día de la semana
 */
public class IfElseAnidado {
    public static void main (String [] args){
        // 0-> Domingo, 1-> Lunes, 2-> Martes, 3-> Miércoles
        // 4-> Jueves, 5-> Viernes, 6-> Sábado
        java.util.Random random = new java.util.Random();
        int one = random.nextInt();
        if (one < 0)
            one *=  -1;
        one %= 7;
        System.out.println("One = " + one);
        if (one == 0 || one == 6)
            System.out.println("Día de descanso.");
        else if (one == 1)
            System.out.println("Inicio de semana.");
        else if (one == 5)
            System.out.println("Inicia el fín de semana.");
        else
            System.out.println("Entre semana.");
    }
}
```

### *SWITCH*

Se trata de una alternativa a la bifurcación **if /else if** cuando se compara la **misma expresión** con distintos valores.

Su forma general es la siguiente:

```
switch (expression) {
        case value1:
                statements1;
                break;
        case value2:
                statements2;
                break;
        case value3:
                statements3;
                break;
        case value4:
                statements4;
                break;
        case value5:
                statements5;
                break;
        case value6:
                statements6;
                break;
        [default:
                statements7;]
}
```

Las características más relevantes de **switch** son las siguientes:

- Cada sentencia **case** corresponde con un único valor de **expression**. No se pueden establecer rangos o condiciones, sino que se debe comparar con valores concretos de tipo **int** (incluyendo a los que se pueden convertir a **int** como **byte, char y short**) y **enumeraciones**.
- No puede haber dos etiquetas **case** con el mismo valor.
- Los valores que no están comprendidos en alguna sentencia case se pueden gestionar en **default**, que es **opcional**.
- En ausencia de **break**, cuando se ejecuta una sentencia case se ejecutan también todas las case que van a continuación, hasta que se llega a un **break** o hasta que se termina el **switch**.

```
/**
 * Programa que genera un número aleatorio de 0 a 6, los cuales
 * tienen una correspondencia con un día de la semana.
 */
public class Switch {
    public static void main (String [] args){
        // 0-> Domingo, 1-> Lunes, 2-> Martes, 3-> Miércoles
        // 4-> Jueves, 5-> Viernes, 6-> Sábado
        java.util.Random random = new java.util.Random();
        int one = random.nextInt();
        if (one < 0)
            one *=  -1;
        one %= 7;
        System.out.println("One = " + one);
        switch (one) {
            case 0:
            case 6:
                System.out.println("Día de descanso.");
                break;
            case 1:
                System.out.println("Inicio de semana.");
                break;
            case 5:
                System.out.println("Inicia el fín de semana.");
                break;
            default:
                System.out.println("Entre semana.");
                break;
        }
    }
}
```

### *Operador ternario*

El operador ternario es una estructura parecida a if-else, para cuando el bloque de código está constituido por una sola instrucción. La sintaxis del operador ternario es la siguiente:

condicion_logica ? expresion1: expresion2

El operador ternario evalúa la condición lógica, si se cumple (true) se ejecuta la instrucción que está a la derecha del ? (expresion1); si no se cumple (false) se ejecuta la instrucción que está a la derecha de los : (expresion2).

```
/**
 * Se genera un valor aleatorio en valor absoluto para después
 * comparar su valor en alguna opción del switch
 */
public class Ternario {
    public static void main (String [] args){
        // 0-> Domingo, 1-> Lunes, 2-> Martes, 3-> Miércoles
        // 4-> Jueves, 5-> Viernes, 6-> Sábado
        java.util.Random random = new java.util.Random();
        int one = random.nextInt();
        one = one < 0 ? one*-1: one;
        one %= 7;
        System.out.println("One = " + one);
        switch (one) {
            case 0:
            case 6:
                System.out.println("Día de descanso.");
                break;
            case 1:
                System.out.println("Inicio de semana.");
                break;
            case 5:
                System.out.println("Inicia el fin de semana.");
                break;
            default:
                System.out.println("Entre semana.");
                break;
        }
    }
}
```

### *ENUMERADOR*

Los enumeradores son listas que permiten almacenar valores constantes. Todos los valores que le pertenecen a un enumerador son estáticos y finales.

Para definir una enumeración en java se utiliza la palabra reservada **enum**. La sintaxis es la siguiente:

```
[modificadorAcceso] enum Nombre {
        VALOR1, VALOR2, VALOR3, …
}
```

Debido a que los elementos de las enumeraciones son estáticos, el acceso a los mismos se realiza a través del nombre de la enumeración, seguido de punto y después el nombre del valor.

```
public enum Valores {
        VALOR1, VALOR2, VALOR3
}

Valores valor = Valores.VALOR1;
```

Los nombres de las enumeraciones siguen la misma convención que los nombres de las clases, es decir, ocupan notación Upper Camell Case. Así mismo, como los valores son constantes se escriben en mayúscula.

```java
/**
 * Se crea una enumeración para evaluar en el switch
 */
public class DiasSemana {
    public enum Dia {
        DOMINGO, LUNES, MARTES, MIERCOLES,  JUEVES, VIERNES, SABADO
    }

    public void describirDia(Dia diaElegido) {
        switch (diaElegido) {
            case LUNES:
                System.out.println("Inicio de semana.");
                break;
            case VIERNES:
                System.out.println("Inicia fin de semana.");
                break;
            case SABADO:
            case DOMINGO:
                System.out.println("Día de descanso.");
                break;
            default:
                System.out.println("Entre semana.");
                break;
        }
    }
}
```

```java
/**
 * Se prueban los valores de la enumeración
 */
public class PruebaDiasSemana {
    public static void main(String[] args) {
        DiasSemana dias = new DiasSemana();
        System.out.println("LUNES");
        dias.describirDia(DiasSemana.Dia.LUNES);
        System.out.println("MIÉRCOLES");
        dias.describirDia(DiasSemana.Dia.MIERCOLES);
        System.out.println("VIERNES");
        dias.describirDia(DiasSemana.Dia.VIERNES);
        System.out.println("SÁBADO");
        dias.describirDia(DiasSemana.Dia.SABADO);
    }
}
```

## Clases de utilerías

En Java existen algunas clases que sirven para apoyar el desarrollo de aplicaciones, dichas clases tienen funcionalidades generales como por ejemplo cálculos matemáticos, fechas, etc. Algunas de las clases más útiles son: Math, Date y Calendar.

### *Math*

Esta clase proporciona métodos para la realización de las **operaciones matemáticas** más habituales. Para utilizar sus métodos simplemente se utiliza el nombre de la clase Math seguida del operador punto y el nombre del método a utilizar.

Ejemplo:

```
Math.pow(5, 2);        //Eleva 5 a la potencia 2

Math.sqrt(25);         //Obtiene la raíz cuadrada de 25

Math.PI;               // Obtiene el valor aproximado de la constante PI
```

### *Date y Calendar*

En el paquete java.util se encuentran dos clases para el tratamiento básico de fechas: **Date** y **Calendar**.

Un objeto **Date** representa una fecha y hora concretas con precisión de un milisegundo. Esta clase permite manipular una fecha y obtener información de la misma de una manera sencilla.

Para crear un objeto de la clase **Date** con la fecha y hora actual se utiliza:

```
Date fecha = new Date( );
```

Usando el método *toString( )* se obtiene la representación en forma de cadena de la fecha:

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

A partir de la versión 1.1 se incorporó una nueva clase llamada **Calendar** que amplía las posibilidades a la hora de trabajar con fechas, por tanto, es una clase que surgió para cubrir las carencias de la clase **Date** en el tratamiento de las fechas. Para crear un objeto de **Calendar** se usa la siguiente sintaxis:

```
Calendar calendario = Calendar.getInstance( );
```

Utilizando el método *get( )* se puede recuperar cada uno de los campos que componen la fecha, para ello este método acepta un numero entero indicando el campo que se quiere obtener. La propia clase **Calendar** define una serie de **constantes** con los valores que corresponden a cada uno de los campos que componen una fecha y hora.

Ejemplo:

```
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;

public class Fechas {

    public static void main(String[] args) {
        Date fecha = new Date();
        System.out.println(fecha.toString());
        SimpleDateFormat formateador = new SimpleDateFormat("dd-MM-yyyy");
        System.out.println(formateador.format(fecha));

        Calendar calendario = Calendar.getInstance();
        String miFecha = "Hoy es día ";
        miFecha += calendario.get(Calendar.DAY_OF_MONTH) + " del mes ";
        miFecha += calendario.get(Calendar.MONTH)+ 1 + " de ";
        miFecha += calendario.get(Calendar.YEAR);
        System.out.println(miFecha);

    }

}
```

A partir de la versión **Java 8**, el manejo de las fechas y el tiempo ha cambiado en Java. Desde esta versión, se ha creado una nueva API para el manejo de fechas y tiempo en el paquete **java.time**, que resuelve distintos problemas que se presentaban con el manejo de fechas y tiempo en versiones anteriores.

Ejemplo:

```
LocalDate hoy = LocalDate.now();
System.out.println(hoy);
System.out.println(hoy.plusWeeks(1));
```

# Bibliografía


*Martín, Antonio*
***Programador Certificado Java 2.***
*Segunda Edición.*
*México*
*Alfaomega Grupo Editor, 2008*

*Sierra Katy, Bates Bert*
***SCJP Sun Certified Programmer for Java 6 Study Guide***
*Mc Graw Hill*

*Dean John, Dean Raymond.*
***Introducción a la programación con Java***
*Primera Edición.*
*México*
*Mc Graw Hill, 2009*

# Guía práctica de estudio 06: Estructuras de repetición



***Elaborado por:***

M.C. M. Angélica Nakayama C.

Ing. Jorge A. Solano Gálvez

***Autorizado por:***

M.C. Alejandro Velázquez Mena

# Guía práctica de estudio 06: Estructuras de repetición

## Objetivo:

Implementar programas utilizando estructuras de repetición en un lenguaje orientado a objetos.

## Actividades:

- Conocer la sintaxis para declarar diversas estructuras de repetición.
- Implementar el uso de estructuras de repetición en un programa.

## Introducción

Los lenguajes de programación tienen elementos básicos que se utilizan como bloques constructivos, así como reglas para que esos elementos se combinen. Estas reglas se denominan **sintaxis del lenguaje**. Solamente las instrucciones sintácticamente correctas pueden ser interpretadas por la computadora y los programas que contengan errores de sintaxis son rechazados por la máquina.

La **sintaxis** de un lenguaje de programación se define como el conjunto de reglas que deben seguirse al escribir el código fuente de los programas para considerarse como correctos para ese lenguaje de programación.

Las **estructuras de control** permiten modificar el flujo de ejecución de las instrucciones de un programa. Todas las estructuras de control tienen un único punto de entrada. Las estructuras de control se pueden clasificar en: secuenciales, transferencia de control e iterativas. Básicamente lo que varía entre las estructuras de control de los diferentes lenguajes es su sintaxis, cada lenguaje tiene una sintaxis propia para expresar la estructura.

**NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.**

# Estructuras de control

Las estructuras de programación o estructuras de control permiten **tomar decisiones** o **realizar un proceso repetidas veces**. En la mayoría de los lenguajes de programación, este tipo de estructuras son comunes en cuanto a concepto, aunque su sintaxis varía de un lenguaje a otro. La sintaxis de Java coincide prácticamente con la utilizada en C/C++, lo que hace que para un programador de C/C++ no represente ninguna dificultad adicional.

### *Estructuras de repetición*

Las estructuras de repetición, lazos, ciclos o bucles se utilizan para realizar un proceso **repetidas veces**. El código incluido entre las llaves **{ }** (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumplan determinadas condiciones.

Hay que prestar especial atención a los ciclos infinitos, hecho que ocurre cuando la condición de finalizar el ciclo no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores poco experimentados.

### *WHILE*

Las sentencias **statements** se ejecutan mientras **booleanExpression** sea **true**.

```
while (booleanExpression) {
        statements;
}
```

```
public class While {
    public static void main (String []args){
        java.util.Random random = new java.util.Random();
        int number = random.nextInt();
        System.out.println("number: " + number);
        while (number > 0) {
            number = random.nextInt();
            System.out.println("number: " + number);
        }
    }
}
```

### *DO-WHILE*

Es similar al ciclo **while** pero con la particularidad de que el control está **al final del ciclo** (lo que hace que el ciclo se ejecute **al menos una vez**, independientemente de que la condición se cumpla o no). Una vez ejecutados los **statements**, se evalúa la condición: si resulta **true** se vuelven a ejecutar las sentencias incluidas en el ciclo, mientras que si la condición se evalúa a **false** finaliza el ciclo.

*do {*
*statements*
*} while (booleanExpression);*

```
public class DoWhile {
    public static void main (String []args){
        java.util.Random random = new java.util.Random();
        int number;
        do {
            number = random.nextInt();
            System.out.println("number: " + number);
        } while (number > 0);
    }
}
```

### *FOR*

La forma general del **for** es la siguiente:

*for (initialization; booleanExpression; increment) {*
*statements;*
*}*

La sentencia o sentencias **initialization** se ejecutan al comienzo del **for**, e **increment** después de **statements**. La **booleanExpression** se evalúa al comienzo de cada iteración; el ciclo termina cuando la expresión de comparación toma el valor **false**. Cualquiera de las tres partes puede estar vacía. La **initialization** y el **increment** pueden tener varias expresiones separadas por comas.

```java
public class Argumentos {
    public static void main(String args[]){
        if ( args.length == 0 ){
            System.out.println("ERROR!, al ejecutar el programa, se deben");
            System.out.println("pasar argumentos de la siguiente manera:.");
            System.out.println("java Argumentos ARG1 ARG2 ...");
        }

        for ( int i = 0 ; i < args.length ; i++ ){
            System.out.println("Parametro " + (i+1) + ": " + args[i]);
        }
    }
}
```

### *BREAK y CONTINUE*

La sentencia **break** es válida tanto para las bifurcaciones como para los ciclos. Hace que se **salga inmediatamente** del ciclo o bloque que se está ejecutando, sin realizar la ejecución del resto de las sentencias.

```java
public class Break {
    public static void main (String []args){
        java.util.Random random = new java.util.Random();
        int number, limit = 5;
        while (limit > 0) {
            number = random.nextInt();
            System.out.println("number: " + number);
            if (number < 0){
                break;
            }
            limit--;
        }
        System.out.println("While done.");
    }
}
```

La sentencia **continue** se utiliza en los ciclos (no en bifurcaciones). Finaliza la iteración "i" que en ese momento se está ejecutando (no ejecuta el resto de sentencias que hubiera hasta el final del bucle). Vuelve al comienzo del bucle y comienza la **siguiente iteración** (i+1).

```java
public class Continue {
    public static void main (String []args){
        java.util.Random random = new java.util.Random();
        int number, limit = 5;
        do {
            number = random.nextInt();
            System.out.println("number: " + number);
            if (number < 0){
                continue;
            }
            limit--;
        } while (limit > 0);
        System.out.println("While done.");
    }
}
```

## Colecciones

Una **colección** es un objeto que almacena un conjunto de **referencias a objetos**, es decir, es parecido a un arreglo de objetos. Sin embargo, a diferencia de los arreglos, las colecciones son **dinámicas**, en el sentido de que no tienen un tamaño fijo y permiten añadir y eliminar objetos en tiempo de ejecución.

Java incluye un amplio conjunto de clases para la creación y tratamiento de colecciones. Todas ellas proporcionan una serie de métodos para realizar las operaciones básicas sobre una colección, como son:

- Añadir objetos a la colección.
- Eliminar objetos de la colección.
- Obtener un objeto de la colección
- Localizar un objeto en la colección.
- Iterar a través de una colección.

Los principales tipos de colecciones que se encuentran por defecto en el API Java son:

### Conjuntos

Un conjunto (**Set**) es una colección desordenada (no mantiene un orden de inserción) y no permite elementos duplicados.

Clases de este tipo: HashSet, TreeSet, LinkedHashSet.

### Listas

Una lista (**List**) es una colección ordenada (debido a que mantiene el orden de inserción) pero permite elementos duplicados.

Clases de este tipo: ArrayList y LinkedList

## Mapas

Un mapa (**Map** también llamado arreglo asociativo) es un conjunto de elementos agrupados con una llave y un valor: ***<llave, valor>***

Donde las llaves no pueden ser repetidas y a cada valor le corresponde una llave. La columna de valores sí puede repetir elementos.

Clases de este tipo: HashMap, HashTable, TreeMap, LinkedHashMap.

Algunas de las clases más usadas para manejo de colecciones son las siguientes (todas ellas se encuentran en el paquete **java.util**):

### *ArrayList*

Se basa en un arreglo redimensionable que aumenta su tamaño según crece la colección de elementos. Es la que mejor rendimiento tiene sobre la mayoría de situaciones. Para crear un objeto ArrayList se utiliza la siguiente sintáxis:

*ArrayList<TipoDato> nombreVariable = new ArrayList<TipoDato>( );*

Ejemplo:

*ArrayList<Integer> arreglo = new ArrayList<Integer>( );*

En este caso se creó un **ArrayList** llamado **arreglo**, el cual podrá contener elementos enteros (Integer).

Una vez creado, se pueden usar los métodos de la clase ArrayList para realizar las operaciones habituales con una colección, las más usuales son:

- **add(elemento)** – Añade un nuevo elemento al ArrayList y lo sitúa al final del mismo.
- **add(índice, elemento)** – Añade un nuevo elemento al ArrayList en la posición especificada por índice, desplazando hacia delante el resto de los elementos de la colección.
- **get(índice)** – Devuelve el elemento en la posición índice.

- **remove(índice) –** Elimina el elemento del ArrayList recorriendo los elementos de las posiciones siguientes. Devuelve el elemento eliminado.
- **clear( ) –** Elimina todos los elementos del ArrayList.
- **indexOf(elemento) –** Localiza en el ArrayList el elemento indicado devolviendo su posición o índice. En caso de que el elemento no se encuentre devuelve -1.
- **size( ) –** Devuelve el número de elementos almacenados en el ArrayList.

Ejemplo:

```
import java.util.ArrayList;

public class Colecciones {

    public static void main(String[] args) {
        ArrayList<Integer> arreglo = new ArrayList<Integer>();
        arreglo.add(1);
        arreglo.add(8);
        arreglo.add(5);
        arreglo.add(1, 9);
        System.out.println("Tamaño del array list " + arreglo.size());
        System.out.println("Elemento en la posicion 3: " + arreglo.get(3));
        for (Integer elemento : arreglo) {
            System.out.println(elemento);
        }

    }

}
```

### *Hashtable*

La clase Hashtable representa un tipo de colección basada en claves, donde los elementos almacenados en la misma (valores) no tienen asociado un índice numérico basado en su posición, sino una clave que lo identifica de forma única dentro de la colección. Una clave puede ser cualquier tipo de objeto.

La utilización de colecciones basadas en claves resulta útil en aquellas aplicaciones en las que se requiera realizar búsquedas de objetos a partir de un dato que lo identifica. La creación de un objeto Hashtable se realiza de la siguiente manera:

*Hashtable<TipoDatoClave, TipoDatoElemento> nombreVariable = new Hashtable<TipoDatoClave, TipoDatoElemento>( );*

Ejemplo:

*Hashtable<String, Integer> miHashTable = new Hashtable<String, Integer>( );*

Los principales métodos de la clase Hashtable para manipular la colección son los siguientes:

- **put(clave, valor)** – Añade a la colección el elemento **valor**, asignándole la **clave** especificada. En caso de que exista esa **clave** en la colección el elemento se sustituye por el nuevo **valor**.
- **containsKey(clave)** – Indica si la **clave** especificada existe o no en la colección. Devuelve un **boolean**.
- **get(clave)** – Devuelve el **valor** que tiene asociado la **clave** que se indica. En caso de que no exista ningún elemento con esa **clave** asociada devuelve **null**.
- **remove(clave)** – Elimina de la colección el **valor** cuya **clave** se especifica. En caso de que no exista ningún elemento con esa **clave** no hará nada y devolverá **null**, si existe eliminará el elemento y devolverá una referencia al mismo.
- **size( )** – Devuelve el número de elementos almacenados en el Hashtable.

Ejemplo:

```
import java.util.Hashtable;

public class Colecciones {

    public static void main(String[] args) {

        Hashtable<String, Integer> miTabla = new Hashtable<String, Integer>();
        miTabla.put("uno", 1);
        miTabla.put("dos",2);
        miTabla.put("cinco", 5);

        System.out.println("Contiene a cuatro? " + miTabla.containsKey("cuatro"));

        for (String clave : miTabla.keySet()) {
            System.out.println(clave);
        }
        for (Integer valor : miTabla.values()) {
            System.out.println(valor);
        }

    }

}
```

Al no estar basado en índices, un Hashtable no se puede recorrer totalmente usando el for con una sola variable. Esto no significa que no se pueda iterar sobre un Hashtable, se puede hacer a través de una enumeración.

Los métodos proporcionados por la enumeración (**Enumeration)** permiten recorrer una colección de objetos asociada y acceder a cada uno de sus elementos.

Así, para recorrer completamente el Hashtable, se obtienen sus claves usando el método *keys( )* y para cada una de las claves se obtiene su valor asociado usando *get(clave)*.

Ejemplo:

```
import java.util.Enumeration;
import java.util.Hashtable;

public class Colecciones {

    public static void main(String[] args) {

        Hashtable<String, Integer> miTabla = new Hashtable<String, Integer>();
        miTabla.put("uno", 1);
        miTabla.put("dos",2);
        miTabla.put("cinco", 5);

        String clave;
        Integer valor;
        Enumeration<String> claves = miTabla.keys();
        while(claves.hasMoreElements()){
            clave = claves.nextElement();
            valor = miTabla.get(clave);
            System.out.println("Clave : " + clave + "\tValor : " + valor);
        }

    }

}
```

# Bibliografía


*Martín, Antonio*
***Programador Certificado Java 2.***
*Segunda Edición.*
*México*
*Alfaomega Grupo Editor, 2008*

*Sierra Katy, Bates Bert*
***SCJP Sun Certified Programmer for Java 6 Study Guide***
*Mc Graw Hill*

*Dean John, Dean Raymond.*
***Introducción a la programación con Java***
*Primera Edición.*
*México*
*Mc Graw Hill, 2009*

# Guía práctica de estudio 07: Herencia (1ra parte)


***Elaborado por:***

M.C. M. Angélica Nakayama C.

Ing. Jorge A. Solano Gálvez

***Autorizado por:***

M.C. Alejandro Velázquez Mena

# Guía práctica de estudio 07: Herencia (1ra parte)

## Objetivo:

Implementar los conceptos de herencia en un lenguaje de programación orientado a objetos.

## Actividades:

- Implementar herencia en un lenguaje de programación orientado a objetos.
- Crear una jerarquía de clases.

## Introducción

En la programación orientada a objetos, la **herencia** está en todos lados, de hecho, se podría decir que es casi imposible escribir el más pequeño de los programas sin utilizar **herencia**. Todas las clases que se crean dentro de la mayoría de los lenguajes de programación orientados a objetos heredan implícitamente de la clase *Object* y, por ende, se pueden comportar como objetos (que es la base del paradigma).

La herencia permite crear nuevos objetos que asumen las propiedades de objetos existentes. Una clase que es usada como base para heredarse es llamada **súper clase** o **clase base**. Una clase que hereda de una súper clase es llamada **subclase** o **clase derivada**. La clase derivada hereda todas las propiedades y métodos visibles de la clase base y, además, puede agregar propiedades y métodos propios.

**NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.**

## Herencia

La **herencia** es el proceso que implica la creación de clases a partir de clases ya existentes, permitiendo, además, agregar más funcionalidades. Utilizando herencia la relación jerárquica queda establecida de manera implícita, partiendo de la clase más general (clase base) a la clase más específica (clase derivada).

Las dos razones más comunes para utilizar herencia son:

- Para promover la reutilización de código.
- Para usar polimorfismo.

Para heredar en Java se utiliza la palabra reservada *extends* al momento de definir la clase, su sintaxis es la siguiente:

*[modificadores] class NombreClaseDerivada* ***extends*** *NombreClaseBase*

Los objetos de las clases derivadas (subclases) se crean (instancian) igual que los de la clase base y pueden acceder tanto a atributos y métodos propios, así como a los de la clase base.

Existen lenguajes de programación que permiten heredar de más de una clase, lo que se conoce como multiherencia, sin embargo, **Java solo soporta herencia simple**.

Ejemplo:

Dada la siguiente jerarquía de clases:

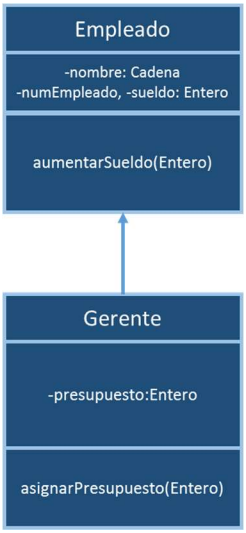



**Figura 1.** Jerarquía de clases de Empleado y Gerente.

Se crea la clase **Empleado** (clase base)

```
public class Empleado {
    private String nombre;
    private int numEmpleado, sueldo;

    public void setNombre(String nombre){
        this.nombre = nombre;
    }
    public String getNombre(){
        return this.nombre;
    }
    public void setNumEmpleado(int numEmpleado){
        this.numEmpleado = numEmpleado;
    }
    public int getNumEmpleado(){
        return this.numEmpleado;
    }
    public void setSueldo(int sueldo){
        if (sueldo >= 0) {
            this.sueldo = sueldo;
        }
    }
    public int getSueldo(){
        return this.sueldo;
    }
    public void aumentarSueldo(int porcentaje) {
        sueldo += (int)(sueldo * porcentaje / 100);
    }
}
```

Y la clase **Gerente** (subclase)

```
public class Gerente extends Empleado {
    private int presupuesto;
    public void setPresupuesto(int presupuesto){
        this.presupuesto = presupuesto;
    }
    public int getPresupuesto(){
        return this.presupuesto;
    }
    void asignarPresupuesto(int p) {
        presupuesto = p;
    }
}
```

Se crea también una clase de **PruebaEmpleado** para validar el comportamiento de las clases creadas y sus métodos.

```
public class PruebaEmpleado {
    public static void main (String [] args){
        Gerente gerente = new Gerente();
        // Gerente hereda todos los atributos y métodos
        // de la Empleado (reutiliza código)
        gerente.setNombre("Luis Aguilar");
        gerente.setNumEmpleado(8524);
        gerente.setSueldo(16000);
        System.out.println("Nombre: " + gerente.getNombre());
        System.out.println("Número de empleado: " + gerente.getNumEmpleado());
        System.out.println("Sueldo: " + gerente.getSueldo());
        gerente.aumentarSueldo(10);
        System.out.println("Nuevo sueldo: " + gerente.getSueldo());
        // Y tiene métodos y atributos propios
        gerente.setPresupuesto(50000);
        System.out.println("Presupuesto: " + gerente.getPresupuesto());
        gerente.asignarPresupuesto(65000);
        System.out.println("Nuevo presupuesto: " + gerente.getPresupuesto());
    }
}
```

## Relaciones IS-A y HAS-A

La **relación IS-A** (es un) se basa en la herencia y permite afirmar que un objeto es de un tipo (clase) en específico. Por ejemplo:

*public class Animal { }*
*public class Caballo extends Animal { }*
*public class Purasangre extends Caballo { }*
Dada la jerarquía de clases anterior se puede afirmar que:

> **Caballo hereda de Animal, lo que significa que Caballo IS-A (es un) Animal.**
> **Purasangre hereda de Caballo, lo que significa que Purasangre IS-A (es un) Caballo.**

La **relación HAS-A (tiene un)** es un concepto que tiene que ver con la Abstracción y el modelado, el cual se conoce como **composición**. Se basa en el uso más que en la herencia, es decir, una clase X HAS-A Y si el código en la clase X tiene como atributo una referencia de la clase Y, por ejemplo:

```
public class Animal { }
public class Caballo extends Animal {
        private SillaMontar miSilla;
}
```

Dada la jerarquía de clases anterior se puede afirmar que:

> **Un Caballo HAS-A (tiene una) SillaMontar, debido a que cada instancia de Caballo tendrá una referencia hacia una SillaMontar.**

## Clase Object

En Java todas las clases que se crean son una subclase de la clase **Object** (excepto, por supuesto, Object), es decir, cualquier clase que se escriba o que se use hereda de Object. Los métodos *clone*, *equals*, *hashCode*, *notify*, *toString*, *wait* y otros son declarados dentro de la clase Object y, por ende, todas las clases los poseen (los heredan).

Por otro lado, el operador *instanceof* es utilizado para verificar si la referencia de un objeto es de un tipo (clase) específico.

Ejemplo:

```
public class Instancias{
	public static void main (String [] args){
		Gerente gerente = new Gerente();
		// Instanceof permite validar si un objeto
		// es de un tipo específico
		if (gerente instanceof Gerente){
			System.out.println("Instancia de Gerente.");
		}
		// Como Gerente hereda de Empleado también es de tipo Empleado
		if (gerente instanceof Empleado){
			System.out.println("Instancia de Empleado.");
		}
		// Se verifica que cualquier objeto es una instancia de Object
		if (gerente instanceof Object){
			System.out.println("Instancia de Object.");
		}
	}
}
```

## Sobrescritura (overridding)

Como ya se mencionó, cuando una clase B hereda de otra A, la clase B puede acceder a todos los atributos y métodos visibles de la clase A, sin embargo, ¿qué pasa cuando el comportamiento de un método no es el adecuado o es parcialmente adecuado? La **sobrescritura** se refiere a la habilidad de **redefinir** el comportamiento de un método específico en una subclase, generando así un comportamiento acorde a las necesidades de cada clase.

El método *toString* es definido dentro de la clase *Object*. Este método es utilizado para mostrar información de un objeto. Por ejemplo, la clase *Empleado* posee los atributos *nombre*, *numEmpleado* y *sueldo*, los cuales se desean mostrar al momento de imprimir un objeto de esta clase, sin embargo, como el método *toString* está definido en la clase *Object* y ésta no conoce los atributos de *Empleado*, dichos atributos no se van a imprimir. Por lo tanto, para mostrar la información deseada es necesario sobrescribir el método.

Un método sobrescrito en una clase derivada debe seguir las siguientes reglas:

- Debe tener el mismo nombre.
- Debe tener el mismo tipo y número de parámetros.
- El tipo de nivel de acceso debe ser igual o más accesible.
- El valor de retorno debe ser del mismo tipo o un subtipo.

Por lo tanto, la **sobrescritura** solo tiene sentido en la herencia, es decir, no es posible sobrescribir un método dentro de la misma clase porque el compilador detectaría que existen dos métodos que se llaman igual y reciben el mismo número y tipo de parámetros.

Ejemplo:

```
public class Empleado {
    private String nombre;
    private int numEmpleado, sueldo;

    public void setNombre(String nombre){
        this.nombre = nombre;
    }
    public String getNombre(){
        return this.nombre;
    }
    public void setNumEmpleado(int numEmpleado){
        this.numEmpleado = numEmpleado;
    }
    public int getNumEmpleado(){
        return this.numEmpleado;
    }
    public void setSueldo(int sueldo){
        if (sueldo >= 0) {
            this.sueldo = sueldo;
        }
    }
    public int getSueldo(){
        return this.sueldo;
    }
    public void aumentarSueldo(int porcentaje) {
        sueldo += (int)(sueldo * porcentaje / 100);
    }
    public String toString (){
        return "Nombre: " + this.nombre +
                "\nNúmero: " + this.numEmpleado +
                "\nSueldo: " + this.sueldo;
    }
}
```

```
public class Gerente extends Empleado {
    private int presupuesto;
    public void setPresupuesto(int presupuesto){
        this.presupuesto = presupuesto;
    }
    public int getPresupuesto(){
        return this.presupuesto;
    }
    void asignarPresupuesto(int p) {
        presupuesto = p;
    }
    public String toString(){
        return super.toString() +
            "\nPresupuesto: " + this.presupuesto;
    }
}
```

```
public class PruebaEmpleado {
    public static void main (String [] args){
        Gerente gerente = new Gerente();
        gerente.setNombre("Luis Aguilar");
        gerente.setNumEmpleado(8524);
        gerente.setSueldo(16000);
        gerente.setPresupuesto(50000);
        // Se manda llamar el método toString, el cual
        // fue sobrescrito en la clase Empleado.
        System.out.println(gerente);
    }
}
```

Cuando se imprime el objeto gerente, implícitamente se manda llamar el método *toString*, como éste fue sobrescrito por la clase *Empleado*, esa información es la que se muestra. Sin embargo, para la clase *Gerente* el método es parcialmente correcto, ya que falta imprimir el atributo presupuesto propio de *Gerente*, por tanto, es necesario volver a sobrescribir el método *toString* para agregar dicho atributo.

## Bibliografía


*Barnes David, Kölling Michael*
***Programación Orientada a Objetos con Java.***
*Tercera Edición.*
*Madrid*
*Pearson Educación, 2007*

*Deitel Paul, Deitel Harvey.*
***Como programar en Java***
*Septima Edición.*
*México*
*Pearson Educación, 2008*

*Martín, Antonio*
***Programador Certificado Java 2.***
*Segunda Edición.*
*México*
*Alfaomega Grupo Editor, 2008*

*Dean John, Dean Raymond.*
***Introducción a la programación con Java***
*Primera Edición.*
*México*
*Mc Graw Hill, 2009*

# Guía práctica de estudio 08: Herencia (2da parte)


***Elaborado por:***

M.C. M. Angélica Nakayama C.

Ing. Jorge A. Solano Gálvez

***Autorizado por:***

M.C. Alejandro Velázquez Mena

# Guía práctica de estudio 08: Herencia (2da parte)

## Objetivo:

Implementar programas que permitan crear objetos con diferentes valores iniciales a lo largo de una jerarquía de clases.

## Actividades:

- Crear constructores sobrecargados.
- Implementar constructores en clases derivadas que utilicen los constructores definidos en la clase base.

## Introducción

Un **constructor** es un método que tiene como propósito **inicializar** los atributos de un nuevo objeto. **Se ejecuta automáticamente cuando se crea un objeto o instancia de la clase**.

Los constructores son métodos especiales que sólo existen cuando se crea un objeto, por lo tanto, dentro de los métodos de la clase no se pueden invocar. Dependiendo del número y tipos de los argumentos proporcionados se ejecutará al constructor correspondiente.

Si no se ha escrito un constructor en la clase, el compilador proporciona un **constructor por defecto**, el cual no tiene parámetros e inicializa los atributos a su valor por defecto.

**NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.**

## Constructores

Un método constructor permite reservar en memoria los atributos y métodos definidos en la clase. Las reglas para crear métodos constructores son:

- El constructor debe tener el mismo nombre que la clase.
- Puede tener cero o más parámetros.
- No devuelve ningún valor (ni si quiera void).
- Toda clase debe tener al menos un constructor.

Cuando se define un objeto se pasan los valores de los parámetros al constructor utilizando una sintaxis similar a una llamada normal a un método.

Ejemplo:

Para la clase Punto se declara un constructor que reciba los valores de las coordenadas (x, y) y dichos valores se le asignan a los atributos miembro correspondientes.

```
public class Punto {

    int x,y;

    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void imprimePunto() {
        System.out.println("Punto [x=" + x + ", y=" + y + "]");
    }
}
```

Cuando se definen constructores de manera explícita, en el ejemplo se crea el constructor que recibe dos parámetros, Java deja de agregar el **constructor por defecto** (el constructor sin parámetros) por lo cual si se intenta invocar al constructor por defecto, el compilador marcará error:

```
public class PruebaPunto {

    public static void main(String[] args) {
        Punto p = new Punto();
        p.x=5;
        p.y=8;
        p.imprimePunto();

        Punto x = new Punto();
        x.x=7;
        x.y=2;
        x.imprimePunto();
    }

}
```

En este caso se pueden hacer dos cosas:

- Cambiar la creación de las instancias para que ahora se envíen los valores (x, y) desde el momento de su creación.

```
public class PruebaPunto {

    public static void main(String[] args) {
        Punto p = new Punto(5, 8);
        p.imprimePunto();

        Punto x = new Punto(7, 2);
        x.imprimePunto();
    }

}
```

- Agregar un constructor por defecto (sin parámetros) el cual puede inicializar los valores a un valor por defecto (o al valor que se desee) o bien puede estar sin implementación (código).

```
public class Punto {

    int x,y;

    public Punto(){

    }
    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void imprimePunto() {
        System.out.println("Punto [x=" + x + ", y=" + y + "]");
    }
}
```

En este último caso, dado que se cuenta con 2 constructores distintos (diferenciados por el número y tipo de parámetros) se tendrían disponibles las dos formas de crear instancias, ya sea con los valores por defecto o pasándolos como parámetros, es decir, constructores sobrecargados.

```
public class PruebaPunto {

    public static void main(String[] args) {
        Punto p = new Punto(5, 8);
        p.imprimePunto();

        Punto x = new Punto();
        x.x=7;
        x.y=2;
        x.imprimePunto();
    }

}
```

## Constructores en la herencia

Como ya se mencionó, un constructor es un método que tiene el mismo nombre que la clase, que no define un valor de retorno y cuyo propósito es inicializar los atributos de un nuevo objeto. Se ejecuta automáticamente cuando se crea un objeto o instancia de la clase.

Cuando se crea un objeto de una clase derivada, implícitamente se crea un objeto de la clase base, el objeto de la súper clase se inicializa con el constructor por defecto aunque se puede invocar a cualquier constructor.

Por tanto, cuando se crea un objeto de una clase base, sin importar el constructor utilizado, se produce una llamada implícita al constructor sin argumentos de la clase base. Sin embargo, si se quiere utilizar constructores sobrecargados es necesario invocarlos explícitamente.

Como se mencionó anteriormente, los constructores son métodos especiales que sólo existen cuando se crea un objeto y no pueden ser invocados dentro de los métodos de la clase. Sin embargo, durante la creación de los objetos, todos los constructores existen, de tal manera que entre constructores sí se pueden invocar (ya sea constructores de la misma clase, con la palabra reservada **this**, o de la clase base, con la palabra reservada **super**).

Así mismo, dentro de un constructor se puede acceder a los elementos (atributos o métodos) de la clase derivada a través de la palabra reservada **this**, o acceder a los elementos de la clase base a través de la palabra reservada **super**.

Ejemplo

```
public class Empleado {
    private String nombre;
    private int numEmpleado, sueldo;

    public Empleado (String nombre, int sueldo, int numEmpleado) {
        this.nombre = nombre;
        this.sueldo = sueldo;
        this.numEmpleado = numEmpleado;
    }

    public void setNombre(String nombre){
        this.nombre = nombre;
    }
    public String getNombre(){
        return this.nombre;
    }
    public void setNumEmpleado(int numEmpleado){
        this.numEmpleado = numEmpleado;
    }
    public int getNumEmpleado(){
        return this.numEmpleado;
    }
    public void setSueldo(int sueldo){
        if (sueldo >= 0) {
            this.sueldo = sueldo;
        }
    }
    public int getSueldo(){
        return this.sueldo;
    }
    public void aumentarSueldo(int porcentaje) {
        sueldo += (int)(sueldo * porcentaje / 100);
    }
    @Override
    public String toString (){
        return "Nombre: " + this.nombre +
            "\nNúmero: " + this.numEmpleado +
            "\nSueldo: " + this.sueldo;
    }
}
```

```
public class Gerente extends Empleado {
    private int presupuesto;
    public Gerente (String nombre, int sueldo,
                    int numEmpleado, int presupuesto) {
        super(nombre, sueldo, numEmpleado);
        this.presupuesto = presupuesto;
    }
    public void setPresupuesto(int presupuesto){
        this.presupuesto = presupuesto;
    }
    public int getPresupuesto(){
        return this.presupuesto;
    }
    void asignarPresupuesto(int p) {
        presupuesto = p;
    }
    @Override
    public String toString(){
        return super.toString() +
            "\nPresupuesto: " + this.presupuesto;
    }
}
```

Como se puede observar en el ejemplo anterior, el constructor de la clase Gerente invoca directamente al constructor de la clase Empleado mediante la palabra reservada ***super***, pasando como argumentos nombre, sueldo y numEmpleado.

La llamada a otros constructores debe ser **la primera sentencia** dentro del constructor, por lo tanto, solo es posible hacer una llamada a otros constructores dentro del método constructor, es decir, o se utiliza **super** (para acceder a un constructor de la clase base) o se utiliza **this** (para acceder a un constructor de la subclase).

```
public class PruebaEmpleado {
    public static void main (String [] args){
        Gerente gerente = new Gerente("Luis Aguilar", 16000, 8524, 50000);
        System.out.println(gerente);
    }
}
```

En el ejemplo de la clase Gerente, debido a que el método *toString* de la clase *Empleado* muestra los atributos deseados, se invoca explícitamente y se concatena el atributo presupuesto. Por eso, el objeto *gerente* creado en la clase *PruebaEmpleado* muestra toda la información del gerente.

## Sobrecarga (overloading) vs Sobrescritura (overridding)

Una de las cosas que más confunden a los programadores novatos son las diferencias entre los conceptos de **sobrecarga** y **sobrescritura**.

La **sobrescritura** es un concepto que tiene sentido en la **herencia** y se refiere al hecho de **volver a definir** un método heredado.

La **sobrecarga** sólo tiene sentido en la clase misma y se refiere a la posibilidad de definir varios métodos con el **mismo nombre**, pero **con diferentes tipos y número de parámetros**.

## Destrucción de objetos (liberación de memoria)

Como los objetos se asignan **dinámicamente**, cuando estos objetos se destruyen será necesarios verificar que la memoria ocupada por ellos ha quedado liberada para usos posteriores. El procedimiento es distinto según el tipo de lenguaje utilizado. Por ejemplo, en C++ los objetos asignados dinámicamente se deben liberar utilizando un operador *delete* y en Java y C# se hace de modo automático utilizando una técnica conocida como **recolección de basura** (garbage collection).

Cuando no existe ninguna referencia a un objeto se supone que ese objeto ya no se necesita y la memoria ocupada por ese objeto puede ser recuperada (liberada), entonces el sistema se ocupa automáticamente de liberar la memoria.

Sin embargo, no se sabe exactamente cuándo se va a activar el **garbage collector**, si no falta memoria es posible que no se llegue a activar en ningún momento. Por lo cual no es conveniente confiar en él para la realización de otras tareas más críticas.

Java no soporta destructores pero existe el método *finalize( )* que es lo más aproximado a un destructor. Las tareas dentro de este método serán realizadas cuando el objeto se destruya y se active el **garbage collector**.

# Bibliografía


*Sierra Katy, Bates Bert*
***SCJP Sun Certified Programmer for Java 6 Study Guide***
*Mc Graw Hill*

*Martín, Antonio*
***Programador Certificado Java 2.***
*Segunda Edición.*
*México*
*Alfaomega Grupo Editor, 2008*

*Joyanes, Luis*
***Fundamentos de programación. Algoritmos, estructuras de datos y objetos.***
*Cuarta Edición*
*México*
*Mc Graw Hill, 2008*

# Guía práctica de estudio 9: Encapsulamiento (1ra parte)


***Elaborado por:***

M.C. M. Angélica Nakayama C.

Ing. Jorge A. Solano Gálvez

***Autorizado por:***

M.C. Alejandro Velázquez Mena

# Guía práctica de estudio 09: Encapsulamiento (1ra parte)

## Objetivo:

Aplicar el concepto encapsulamiento para proteger la información y ocultar la implementación.

## Actividades:

- Implementar el concepto de encapsulamiento en datos miembro.
- Implementar el concepto de encapsulamiento en métodos miembro.

## Introducción

El **encapsulamiento** sucede cuando algo es envuelto en una capa protectora. Cuando el **encapsulamiento** se aplica a los objetos, significa que los datos miembro están protegidos, **"ocultos"** dentro del objeto. Con los datos ocultos, ¿cómo puede el resto del programa acceder a ellos? (El acceso a los datos de un objeto se refiere a leerlos o modificarlos.) El resto del programa no puede acceder de manera directa a los datos de un objeto; lo tiene que hacer con ayuda de los métodos del objeto.

En el supuesto de que los métodos de un objeto estén bien escritos, éstos aseguran que se pueda acceder a los datos de manera adecuada. Al hecho de empaquetar o proteger los datos o atributos con los métodos se denomina **encapsulamiento**.

**NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.**

## Encapsulamiento

La **encapsulación** o **encapsulamiento** significa reunir en una cierta estructura a todos los elementos que en un cierto **nivel de abstracción** se pueden considerar pertenecientes a una misma entidad, y es el proceso de agrupamiento de datos y operaciones relacionadas bajo una misma unidad de programación, lo que permite aumentar la cohesión de los componentes del sistema.

El **encapsulamiento** oculta lo que hace un objeto de lo que hacen otros objetos y del mundo exterior por lo que se denomina también **ocultación de datos**. Un objeto tiene que presentar **"una cara"** al mundo exterior de modo que se puedan iniciar sus operaciones.

Por ejemplo, la televisión tiene un conjunto de botones para encender, apagar y cambiar canal. Una lavadora tiene un conjunto de mandos e indicadores que establecen la temperatura, el nivel de agua, etc. Los botones de la televisión y la lavadora constituyen la comunicación con el mundo interior, es decir son las **interfaces**.

La **interfaz** de una clase representa un **"contrato"** de prestación de servicios entre ella y los demás componentes del sistema. De este modo, los clientes solo necesitan conocer los servicios que este ofrece y no como están implementados internamente.

Por consiguiente, se puede modificar la implementación de una clase sin afectar a las restantes relacionadas con ella.

## Reglas de visibilidad

Las reglas de visibilidad complementan o refinan el concepto de **encapsulamiento**. Los diferentes niveles de visibilidad dependen del lenguaje de programación con el que se trabaje. Estos niveles de visibilidad son:

- El nivel más fuerte se denomina nivel **"privado"**; la sección privada de una clase es totalmente invisible para otras clases, solo miembros de la misma clase pueden acceder a atributos localizados en la sección privada.

- Es posible aliviar el nivel de ocultamiento situando algunos atributos en la sección **"protegida"** de la clase. Estos atributos son visibles tanto para la misma clase como para las clases derivadas de la clase. Para las clases restantes permanecen invisibles.
- El nivel más débil se obtiene situando los atributos en la sección **"pública"** de la clase con lo cual se hacen visibles a todas las clases.

Los **atributos privados** están contenidos en el interior de la clase, ocultos a cualquier otra clase. Debido a que los atributos están **encapsulados** dentro de una clase, se necesitará definir cuáles son las clases que tienen acceso a visualizar y modificar el valor de los atributos. Esta característica se conoce como **visibilidad de los atributos**. En general se recomienda visibilidad **privada** o **protegida** para los atributos.

## Modificadores de acceso

Los modificadores de acceso se utilizan para definir la visibilidad de los miembros de una clase (atributos y métodos) y de la propia clase. En Java existen tres modificadores de acceso:

- **public**
- **protected**
- **private**

Sin embargo, existen cuatro niveles de acceso. Cuando no se especifica ninguno de los tres modificadores anteriores se tiene el nivel de acceso por defecto, que es el nivel de paquete.

La sintaxis para los modificadores de acceso es simplemente anteponerlos a la declaración de atributos y métodos.

***Modificador*** *tipoDato nombreVariable;*

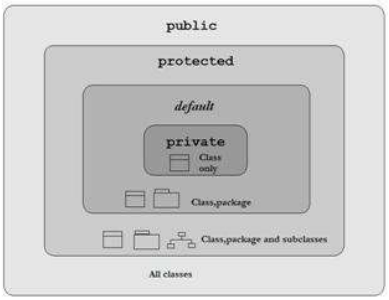
***Modificador*** *tipoDato nombreMetodo(parámetros…)*

Ejemplo:

```
private String nombre;
public int operacion(int a, int b){
```

A continuación se muestra el acceso permitido para cada modificador.

| Visibilidad | Public | Protected | Default | Private |
|---|---|---|---|---|
| Desde la misma Clase | SI | SI | SI | SI |
| Desde cualquier Clase del mismo Paquete | SI | SI | SI | NO |
| Desde una SubClase del mismo Paquete | SI | SI | SI | NO |
| Desde una SubClase fuera del mismo Paquete | SI | SI, a través de la herencia | NO | NO |
| Desde cualquier Clase fuera del Paquete | SI | NO | NO | NO |




Un principio fundamental en la programación orientada a objetos es la **ocultación de la información**, que significa que determinados datos del interior de una clase no pueden ser accedidos por funciones externas a la clase, esto sugiere que solamente la información sobre lo que puede hacer una clase debe estar visible desde el exterior, pero no cómo lo hace. Esto tiene una gran ventaja: si ninguna otra clase conoce cómo está almacenada la información entonces se puede cambiar fácilmente la forma de almacenarla sin afectar otras clases.

## Acceso a miembros

Se puede reforzar la separación del qué hacer del cómo hacerlo, declarando los campos como **privados** y usando un **método de acceso** para acceder a ellos.

Los **métodos de acceso** son el medio de acceder a los atributos privados del objeto. Son métodos públicos del objeto y pueden ser:

- **Métodos modificadores**: llamamos métodos modificadores a aquellos métodos que dan lugar a un cambio en el valor de uno o varios de los atributos del objeto.
- **Métodos consultores u observadores**: son métodos que devuelven información sobre el contenido de los atributos del objeto sin modificar los valores de estos atributos.

Cuando se crea una clase es frecuente que lo primero que se haga sea establecer métodos para consultar sus atributos y estos métodos suelen ir precedidos del prefijo **get** (getNombre, getValor, etc.) por lo que muchas veces se alude coloquialmente a ellos como **"métodos get"** o **"getters"**. Los **métodos get** son un tipo de **métodos consultores**, porque solo consultan y devuelven el valor de los atributos de un objeto.

Se suele proceder de igual forma con métodos que permitan establecer los valores de los atributos. Estos métodos suelen ir precedidos del prefijo **set** (setNombre, setValor, etc.) por lo que muchas veces se alude coloquialmente a ellos como **"métodos set"** o **"setters"**. Los **métodos set** son un tipo de **métodos modificadores**, porque cambian el valor de los atributos de un objeto.

Ejemplo:

```java
public class Circulo {

    static float PI = 3.14159f;
    private float radio;

    public float getRadio() {
        return radio;
    }
    public void setRadio(float radio) {
        this.radio = radio;
    }
    public float perimetro(){
        return 2 * PI * radio;
    }
    public float area(){
        return PI * radio * radio;
    }
    public String toString() {
        return "Circulo [radio=" + radio + "]";
    }
}
```

```java
public class PruebaFiguras {

    public static void main(String[] args) {
        Circulo cir=new Circulo();
        cir.setRadio(7.2f);
        System.out.println("El area es " + cir.area());
    }

}
```

Parece ser que proporcionar herramientas para establecer y obtener es esencialmente lo mismo que hacer las variables de instancia *public*. Si una variable de instancia se declara como *public*, cualquier método que tenga una referencia a un objeto que contenga esta variable de instancia podrá leer o escribir en ella. Si una variable de instancia se declara como *private*, un método set *public* evidentemente permite a otros métodos el acceso a la variable, pero el método set puede controlar la manera en que el cliente puede tener acceso a la variable.

Para el mismo ejemplo, se puede validar que el radio que se le asigna a un círculo nunca sea negativo modificando su método setter:

```java
public void setRadio(float radio) {
    if(radio < 0){
        radio = 0;
    }
    this.radio = radio;
}
```

Entonces, aunque los métodos **set** y **get** proporcionan acceso a los datos privados, el programador restringe su acceso mediante la implementación de los métodos.

## Composición

Una clase puede definir como datos miembro a objetos de otras clases. A dicha capacidad se le conoce como **composición** o como relación **"tiene un"** (has a)**.**

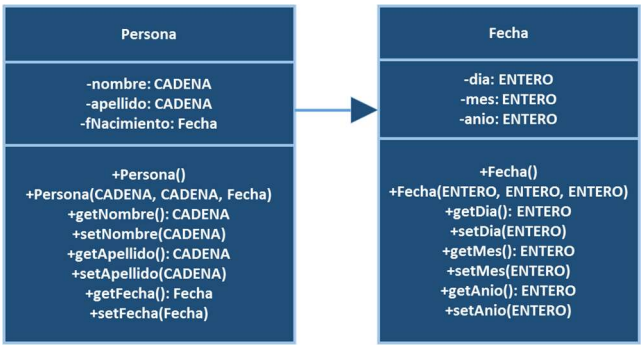
El concepto de **composición** no fue creado para la programación; suele usarse a menudo para objetos complejos en el mundo real. Toda criatura viviente y la mayor parte de los productos manufacturados están constituidos por partes. A menudo, cada parte es un subsistema que está integrado por su propio conjunto de sub-partes. Junto, todo el sistema forma una jerarquía de composición.

Por ejemplo, el cuerpo humano está compuesto por varios órganos: cerebro, corazón, estómago, huesos, músculos, etc. A su vez, cada uno de estos órganos está compuesto por muchas células, y cada una de estas células está compuesta por muchos orgánulos, como el núcleo (el "cerebro" de una célula) y las mitocondrias (los "músculos" de una célula). Cada orgánulo está compuesto por muchas moléculas. Y finalmente, cada molécula orgánica está compuesta por muchos átomos.

En una jerarquía de **composición** (así como en una jerarquía de agregación), la relación entre una clase contenedora y una de sus clases parte se denomina relación tiene-un. Por ejemplo, cada cuerpo humano tiene un cerebro y tiene un corazón.

La **composición** es una forma de **reutilización de software**, en donde una clase tiene como miembros referencias a objetos de otras clases.

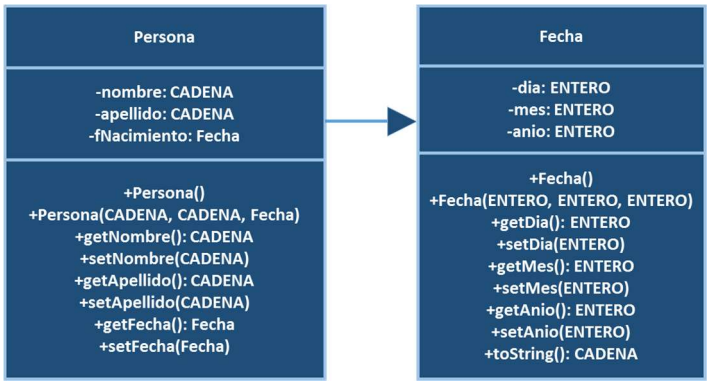
Ejemplo:



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

    public Fecha() {

    }
    public Fecha(int dia, int mes, int anio) {
        setDia(dia);
        setMes(mes);
        setAnio(anio);
    }
    public int getDia() {
        return dia;
    }
    public void setDia(int dia) {
        if(dia > 0 && dia < 32){
            this.dia = dia;
        } else {
            System.out.println("Día no valido");
        }
    }
    public int getMes() {
        return mes;
    }
    public void setMes(int mes) {
        if(mes>0 && mes < 13){
            this.mes = mes;
        } else {
            System.out.println("Mes no valido");
        }
    }
    public int getAnio() {
        return anio;
    }
    public void setAnio(int anio) {
        if(anio>0){
            this.anio = anio;
        } else {
            System.out.println("El año no puede ser negativo");
        }
    }

}
```

```java
public class Persona {
    private String nombre;
    private String apellido;
    private Fecha fNacimiento;

    public Persona() {
    }
    public Persona(String nombre, String apellido, Fecha fNacimiento) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.fNacimiento = fNacimiento;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getApellido() {
        return apellido;
    }
    public void setApellido(String apellido) {
        this.apellido = apellido;
    }
    public Fecha getFNacimiento() {
        return fNacimiento;
    }
    public void setFNacimiento(Fecha fNacimiento) {
        this.fNacimiento = fNacimiento;
    }
}


public class PruebaPersona {

    public static void main(String[] args) {
        Persona per1 = new Persona();
        Fecha nac = new Fecha();
        per1.setNombre("Juan");
        per1.setApellido("Perez");
        nac.setDia(15);
        nac.setMes(8);
        nac.setAnio(1950);
        per1.setFNacimiento(nac);
        System.out.println("Nombre : " + per1.getNombre());
        System.out.println("Apellido : " + per1.getApellido());
        System.out.println("Fecha Nacimiento : " + per1.getFNacimiento().getDia() +
                "/"+per1.getFNacimiento().getMes()+"/"+per1.getFNacimiento().getAnio());
    }

}
```

Sin embargo, se pueden modificar las clases para que no tenga que instanciarse la clase Fecha fuera de la clase Persona, quedando así un mejor diseño de clases.



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

    public Fecha() {

    }
    public Fecha(int dia, int mes, int anio) {
        setDia(dia);
        setMes(mes);
        setAnio(anio);
    }
    public int getDia() {
        return dia;
    }
    public void setDia(int dia) {
        if(dia > 0 && dia < 32){
            this.dia = dia;
        } else {
            System.out.println("Día no valido");
        }
    }
    public int getMes() {
        return mes;
    }
    public void setMes(int mes) {
        if(mes>0 && mes < 13){
            this.mes = mes;
        } else {
            System.out.println("Mes no valido");
        }
    }
    public int getAnio() {
        return anio;
    }
    public void setAnio(int anio) {
        if(anio>0){
            this.anio = anio;
        } else {
            System.out.println("El año no puede ser negativo");
        }
    }
    public String toString() {
        return dia + "/" + mes + "/" + anio;
    }

}
```

```java
public class Persona {
    private String nombre;
    private String apellido;
    private Fecha fNacimiento;

    public Persona() {
        fNacimiento = new Fecha();
    }
    public Persona(String nombre, String apellido, int fDia, int fMes, int fAnio) {
        this.nombre = nombre;
        this.apellido = apellido;
        fNacimiento.setDia(fDia);
        fNacimiento.setMes(fMes);
        fNacimiento.setAnio(fAnio);
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getApellido() {
        return apellido;
    }
    public void setApellido(String apellido) {
        this.apellido = apellido;
    }
    public void setFNacimiento(int dia, int mes, int anio){
        fNacimiento.setDia(dia);
        fNacimiento.setMes(mes);
        fNacimiento.setAnio(anio);
    }
    public Fecha getFNacimiento(){
        return fNacimiento;
    }
}

public class PruebaPersona {

    public static void main(String[] args) {
        Persona per1 = new Persona();
        per1.setNombre("Juan");
        per1.setApellido("Perez");
        per1.setFNacimiento(15, 8, 1950);
        System.out.println("Nombre : " + per1.getNombre());
        System.out.println("Apellido : " + per1.getApellido());
        System.out.println("Fecha Nacimiento : " + per1.getFNacimiento());
    }

}
```

# Bibliografía


*Barnes David, Kölling Michael*
***Programación Orientada a Objetos con Java.***
*Tercera Edición.*
*Madrid*
*Pearson Educación, 2007*

*Deitel Paul, Deitel Harvey.*
***Como programar en Java***
*Septima Edición.*
*México*
*Pearson Educación, 2008*

*Joyanes, Luis*
***Fundamentos de programación. Algoritmos, estructuras de datos y objetos.***
*Cuarta Edición*
*México*
*Mc Graw Hill, 2008*

*Dean John, Dean Raymond.*
***Introducción a la programación con Java***
*Primera Edición.*
*México*
*Mc Graw Hill, 2009*

# Guía práctica de estudio 10: Polimorfismo (1ra parte)


***Elaborado por:***

M.C. M. Angélica Nakayama C.

Ing. Jorge A. Solano Gálvez

***Autorizado por:***

M.C. Alejandro Velázquez Mena

# Guía práctica de estudio 10: Polimorfismo (1ra parte)

## Objetivo:

Implementar el concepto de polimorfismo en un lenguaje de programación orientado a objetos.

## Actividades:

- Crear una jerarquía de clases.
- Crear una referencia de la clase base más general.
- Crear instancias de diferentes clases derivadas.

## Introducción

El término **polimorfismo** es constantemente referido como uno de los pilares de la programación orientada a objetos (junto con la Abstracción, el Encapsulamiento y la Herencia).

El término **polimorfismo** es una palabra de origen griego que significa **muchas formas.** En la programación orientada a objetos se refiere a la **propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos**.

El **polimorfismo** consiste en conseguir que un objeto de una clase se comporte como un objeto de cualquiera de sus subclases. Se puede aplicar tanto a métodos como a tipos de datos. Los métodos pueden evaluar y ser aplicados a diferentes tipos de datos de manera indistinta. Los tipos polimórficos son tipos de datos que contienen al menos un elemento cuyo tipo no está especificado.

**NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.**

## Polimorfismo

**Polimorfismo** se refiere a la habilidad de **tener diferentes formas**. El término **IS-A** se refiere a la pertenencia de un objeto con un tipo, es decir, si se crea una instancia de tipo A, se dice que el objeto creado es un A.

El polimorfismo se puede clasificar en dos grandes grupos:

- Polimorfismo dinámico (o paramétrico): es aquel en el que **no se especifica el tipo de datos sobre el que se trabaja** y, por ende, se puede utilizar todo tipo de datos compatible. Este tipo de polimorfismo también se conoce como programación genérica.
- Polimorfismo estático (o ad hoc): es aquel en el que **los tipos de datos que se pueden utilizar deben ser especificados** de manera explícita antes de ser utilizados.

En Java, cualquier objeto que pueda comportarse como más de un **IS-A** (es un) puede ser considerado **polimórfico**. Por lo tanto, todos los objetos en Java pueden ser considerados polimórficos porque todos se pueden comportar como objetos de su propio tipo y como objetos de la clase *Object*.

Por otro lado, debido a que la única manera de acceder a un objeto durante su tiempo de vida es a través de su referencia, existen algunos puntos clave que se deben recordar sobre las mismas:

- Una referencia puede ser solo de un tipo y, una vez declarado, el tipo no puede ser cambiado.
- Una referencia es una variable, por lo tanto, ésta puede ser reasignada a otros objetos (a menos que la referencia sea declarada como *final*).
- El tipo de una referencia determina los métodos que pueden ser invocados del objeto al que referencía, es decir, solo se pueden ejecutar los métodos definidos en el tipo de la referencia.
- A una referencia se le puede asignar cualquier objeto que sea del mismo tipo con el que fue declarada la referencia o de algún subtipo (**Polimorfismo**).

Dada la siguiente jerarquía de clases:

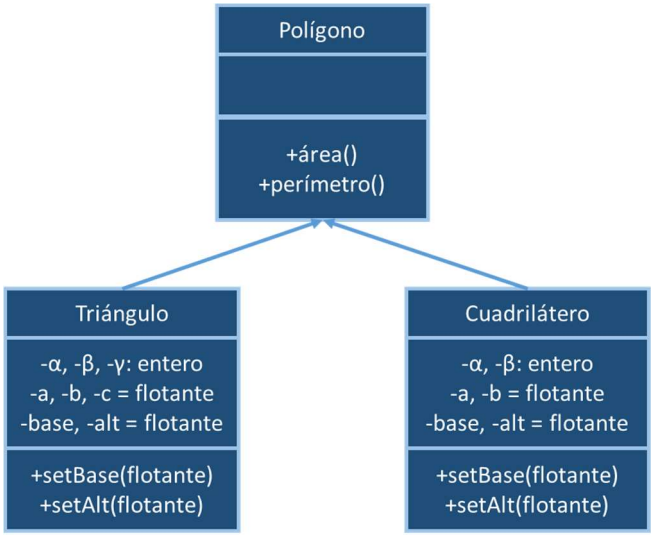



**Figura 1.** Jerarquía de clases

Su codificación sería la siguiente:

```
public class Poligono {
        public double area(){
            return 0d;
        }

        public double perimetro(){
            return 0d;
        }

        public String toString(){
            return "Polígono";
        }
}
```

```java
public class Cuadrilatero extends Poligono {
        private int alfa, beta;
        private float a, b;
        private float base, altura;

        @Override
        public String toString(){
                return "Cuadrilátero";
        }

        public Boolean tieneLadosParalelos(){
            return true;
        }
}
```

```java
public class Triangulo extends Poligono {
        float sideA, sideB, sideC;

        @Override
        public String toString(){
                return "Triángulo";
        }

        public Boolean tieneLadosParalelos(){
            return false;
        }
}
```

La jerarquía de clases anterior se puede probar a través de la siguiente clase:

```java
public class PruebaFigurasGeometricas {
    public static void main (String [] args){
        Poligono poligono = new Poligono();
        // Polígono puede comportarse como Objeto
        Object objeto = poligono;
        System.out.println(objeto);

        // Una referencia puede ser reasignada a otros objetos
        poligono = new Triangulo();
        // Solo se pueden ejecutar los métodos que están definidos
        // en la referencia, sin embargo, se ejecutarán como están
        // implementados en la instancia.
        System.out.println(poligono);

        poligono = new Cuadrilatero();
        // El método toString se puede ejecutar porque está definido
        // en Polígono, sin embargo, se va a ejecutar como está
        // implementado en la instancia (Cuadrilatero o Triangulo).
        System.out.println(poligono);

        // El método tieneLadosParalelos no está definido en Polígono,
        // por lo que la siguiente instrucción marcaría un error:
        //System.out.println(poligono.tieneLadosParalelos());
    }
}
```

## Cast o moldeo de objetos

Es posible convertir un objeto de un tipo más general a uno más específico mientras exista una relación de herencia a través de un cast o moldeo. La única restricción es que el objeto sea de un tipo más específico.

Para validar si el objeto al que apunta la referencia es de un tipo en específico se utiliza la palabra reservada instanceof de la siguiente manera:

```
ref instanceof Triangulo
```

En este caso se verifica si la referencia ref está apuntando en el heap a un objeto del tipo Triangulo. La validación anterior devuelve verdadero o falso.

Existen dos tipos de cast, el cast hacia arriba (up-casting) y el cast hacia abajo (down-casting). El up-casting es explícito, es decir, no es obligatorio indicar a qué tipo de dato se quiere convertir el objeto, se infiere a partir de la referencia. Por otra parte, el down-casting debe ser implícito, es decir, se debe indicar a qué tipo se quiere moldear el objeto.

Por ejemplo, para la jerarquía de clases de Polígono, Triángulo y Cuadrilátero se tiene:

```
          Polígono
          /      \
Triángulo      Cuadrilátero
```

A una referencia tipo Polígono se puede asignar una instancia de Polígono, de Triángulo o de Cuadrilátero sin necesidad de hacer un casting explícito, es decir:

```
Poligono p = new Poligono();
Poligono fig = new Triangulo();
Poligono pol = new Cuadrilatero();
```

Sin embargo, a partir de estas referencias creadas, para recuperar los objetos se debe asignar la referencia tipo polígono a una referencia más específica, realizando un cast de manera explícita, es decir:

Triangulo t = (Triangulo) fig;
Cuadrilatero c = (Cuadrilatero) pol;

```
public class PruebaFigurasGeometricas {
    public static void main (String [] args){
        System.out.println("\nPoligono p = new Triangulo();\n");
        Poligono p = new Triangulo();
        System.out.println(p.toString());
        if (p instanceof Triangulo){
            // Cast de Polígono a Triángulo
            System.out.println("\nTriangulo t = (Triangulo)p\n");
            Triangulo t = (Triangulo)p;
            System.out.println(t.tieneLadosParalelos());
        }

        System.out.println("\np = new Cuadrilatero();\n");

        p = new Cuadrilatero();
        System.out.println(p.toString());
        if (p instanceof Cuadrilatero){
            // Cast de Polígono a Cuadrilátero
            System.out.println("\nCuadrilatero q = (Cuadrilatero)p\n");
            Cuadrilatero q = (Cuadrilatero)p;
            System.out.println(q.tieneLadosParalelos());
        }
    }
}
```

## Polimorfismo en métodos

Un método puede recibir cualquier tipo y número de datos como parámetros y puede devolver cualquier tipo de dato. A este concepto se le conoce también como **polimorfismo en métodos**.

Cuando el parámetro definido es una referencia a una clase, el método es capaz de recibir un objeto de ese tipo o de **cualquier subtipo** de esa clase. Cuando el valor de retorno definido es una referencia a una clase, el método es capaz de devolver un objeto de ese tipo o de **cualquier subtipo** de esa clase.

```java
public class PolimorfismoMetodos {
    public static void main (String [] args){
        Poligono fig1 = crearFigura(1);
        Poligono fig2 = crearFigura(2);
        Poligono fig3 = crearFigura(3);
        imprimirTipoPoligono(fig3);
        imprimirTipoPoligono(fig2);
        imprimirTipoPoligono(fig1);
    }

    // Crear figura recibe como parámetro un entero para
    // indicar el tipo de objeto que se va a crear:
    // 1 polígono, 2 triángulo, 3 cuadrilátero
    public static Poligono crearFigura(int id){
        Poligono pol = null;
        switch(id) {
            case 1:
                pol = new Poligono();
                break;
            case 2:
                pol = new Triangulo();
                break;
            case 3:
                pol = new Cuadrilatero();
                break;
        }
        return pol;
    }

    public static void imprimirTipoPoligono(Poligono p){
        if (p instanceof Triangulo){
            System.out.println("p es una instancia de Triángulo");
        } else {
            if (p instanceof Cuadrilatero){
                System.out.println("p es una instancia de Cuadrilatero");
            } else {
                System.out.println("p es una instancia de Polígono");
            }
        }
    }
}
```

## Bibliografía


*Barnes David, Kölling Michael*
***Programación Orientada a Objetos con Java.***
*Tercera Edición.*
*Madrid*
*Pearson Educación, 2007*

*Deitel Paul, Deitel Harvey.*
***Como programar en Java***
*Septima Edición.*
*México*
*Pearson Educación, 2008*

*Martín, Antonio*
***Programador Certificado Java 2.***
*Segunda Edición.*
*México*
*Alfaomega Grupo Editor, 2008*

*Dean John, Dean Raymond.*
***Introducción a la programación con Java***
*Primera Edición.*
*México*
*Mc Graw Hill, 2009*

# Guía práctica de estudio 11: Polimorfismo (2da parte)


***Elaborado por:***

M.C. M. Angélica Nakayama C.

Ing. Jorge A. Solano Gálvez

***Autorizado por:***

M.C. Alejandro Velázquez Mena

# Guía práctica de estudio 11: Polimorfismo (2da parte)

## Objetivo:

Implementar el concepto de polimorfismo desde un nivel muy general y hasta un nivel muy específico en un lenguaje de programación orientado a objetos.

## Actividades:

- Crear una jerarquía de clases desde un nivel muy abstracto hasta un nivel muy específico.
- Crear una referencia de la clase base más general.
- Crear instancias de diferentes clases derivadas.

## Introducción

Una jerarquía de clases determina la relación que existe entre clases, determinando diversos niveles dentro de la estructura. La relación que existe entre las clases se puede modelar utilizando una estructura de datos no lineal tipo grafo, separados por niveles, donde los nodos en el nivel más alto representan los elementos más generales o abstractos y los nodos en el nivel más bajo representan los elementos más específicos.

Las clases más generales (las del nivel más alto) tienen por objeto determinar un comportamiento estándar para toda la jerarquía de clases. Por otro lado, las clases más específicas (las del nivel más bajo) determinan el último eslabón de la jerarquía, lo que implica que la jerarquía ya no puede hacerse más especializada y, por ende, ha llegado a su fin.

Cada lenguaje de programación modela de manera diferente el elemento más específico y el elemento más general, pero estos conceptos son soportados por la mayoría de los lenguajes.

**NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.**

## Clases abstractas

Las clases abstractas sirven como modelo para la creación de otras clases (clases derivadas), es decir, definen las clases más generales de la jerarquía de clases.

Una clase abstracta permite definir la existencia de métodos, pero no su implementación, es decir, permite identificar métodos comunes para la toda jerarquía de clase sin especificar cómo se deben realizar las acciones de dichos métodos.

Las características principales de las clases abstractas son:

- Pueden definir métodos abstractos y métodos concretos (con o sin implementación).
- Pueden definir atributos.
- Pueden heredar de otras clases.

Una clase abstracta se declara simplemente con el modificador abstract, adicionalmente, se puede usar un modificador de acceso. La sintaxis de una clase abstracta es la siguiente:

```
[modificadores] abstract class NombreClase {
        [modificadores] abstract float metodo();
}
```

Un método abstracto se declara de utilizando el modificador abstract, pero no se define una implementación, es decir, sólo se define la firma de la función que deben respetar todas las clases que sobrescriban dicho método.

La clase que herede de una clase abstracta debe declarar e implementar el comportamiento de los métodos abstractos (a menos que ésta, la que hereda, sea abstracta). De no declararse (implícitamente), el compilador generará un error indicando que no se han implementado

todos los métodos abstractos y que, o bien se implementen, o bien se declare la clase como abstracta.

Dada la siguiente jerarquía de clases:

Polígono
|
Triángulo

La implementación de la clase Polígono define la base de todas las clases que deriven de ella, porque todas heredan de ella. Por lo tanto, es un buen lugar para definir la existencia de métodos que se deseen tengan todas las clases derivadas, en este caso, área y perímetro.

```
public abstract class Poligono {
        public abstract double area();

        public abstract double perimetro();

        @Override
        public String toString(){
            return "Polígono";
        }
}
```

```java
public class Triangulo extends Poligono {
    protected float base, altura, ladoA, ladoB, ladoC;

    public Triangulo() {}

    public Triangulo(float base, float altura,
            float ladoA, float ladoB, float ladoC) {
        this.base = base;
        this.altura = altura;
        this.ladoA = ladoA;
        this.ladoB = ladoB;
        this.ladoC = ladoC;
    }

    public Boolean tieneLadosParalelos(){
        return false;
    }

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

    @Override
    public double perimetro() {
        return ladoA + ladoB + ladoC;
    }

    @Override
    public String toString(){
            return "Triángulo";
    }
}
```

Cuando se crean objetos, la parte izquierda de la ecuación se conoce como referencia. Es posible crear referencias de una clase abstracta:

*Poligono figura;*

Sin embargo, una clase **abstracta <u>no se puede instanciar</u>**, es decir, no se pueden crear objetos de una clase abstracta, la siguiente línea de código generaría un error en tiempo de compilación:

*Poligono figura = new Poligono();*

El que una clase **abstracta** no se pueda instanciar es coherente con la definición, dado que este tipo de clases no tiene completa su implementación y encaja bien con la idea de que un ente abstracto no puede materializarse. Sin embargo, una referencia abstracta sí puede contener un objeto concreto, es decir:

*Poligono figura = new Triangulo();*

Ejemplo:

```
public abstract class Poligono {
        public abstract double area();

        public abstract double perimetro();

        @Override
        public String toString(){
            return "Polígono";
        }
}
```

```java
public class Triangulo extends Poligono {
    protected float base, altura, ladoA, ladoB, ladoC;

    public Triangulo() {}

    public Triangulo(float base, float altura,
            float ladoA, float ladoB, float ladoC) {
        this.base = base;
        this.altura = altura;
        this.ladoA = ladoA;
        this.ladoB = ladoB;
        this.ladoC = ladoC;
    }

    public Boolean tieneLadosParalelos(){
        return false;
    }

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

    @Override
    public double perimetro() {
        return ladoA + ladoB + ladoC;
    }

    @Override
    public String toString(){
            return "Triángulo";
    }
}
```

```java
public class Cuadrilatero extends Poligono {
    protected float ladoA, ladoB;
    protected float base, altura;

    public Cuadrilatero() {}

    public Cuadrilatero(float base, float altura,
            float ladoA, float ladoB) {
        this.base = base;
        this.altura = altura;
        this.ladoA = ladoA;
        this.ladoB = ladoB;
    }

    public Boolean tieneLadosParalelos(){
        return true;
    }

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

    @Override
    public double perimetro() {
        return ladoA + ladoB;
    }

    @Override
    public String toString(){
            return "Cuadrilátero";
    }
}
```

```java
public class PruebaFigurasGeometricas {
    public static void main (String [] args){
        // No se pueden crear objetos de clases abstractas
        // Poligono poligono = new Poligono()

        // Sí se pueden crear referencias de clases abstractas
        Poligono poligono;

        // Las referencias abstractas pueden ser asignadas
        // con objetos concretos mientras estén en la misma
        // jerarquía de clases
        poligono = new Triangulo();
        System.out.println(poligono.area());
        poligono = new Cuadrilatero(2, 4, 2, 4);
        System.out.println(poligono.area());
    }
}
```

## Clase final

Una clase que representa el final de la jerarquía de clases, es decir, es un elemento suficientemente específico que no necesita modificarse. Cuando esto ocurre el funcionamiento de la clase se puede proteger evitando ser modificada por otras clases.

En java para evitar que una clase pueda ser modificada se agrega la palabra reservada final a la declaración de la misma, consiguiendo que ésta no pueda ser heredada y, por ende, modificada.

```
public final class TrianguloIsosceles extends Triangulo {
        @Override
        public double perimetro(){
                return 3 * ladoA;
        }
}
```

```
cannot inherit from final TrianguloIsosceles
----
(Alt-Enter shows hints)
class HeredaTrianguloIsosceles extends TrianguloIsosceles {

}
```

## Interfaces

De igual manera, debido a que una interfaz es una clase abstracta pura (todos los métodos son abstractos).

Para crear una interfaz, se utiliza la palabra reservada interface en lugar de la palabra reservada class. La interfaz puede definirse publica o sin modificador de acceso, y tiene el mismo significado que para las clases. La sintaxis general para declarar una interfaz es la siguiente:

```
interface NombreInterfaz {
        tipoRetorno nombreMetodo([Parametros]);
}
```

Todos los métodos que declara una interfaz son siempre públicos y abstractos. Una interfaz puede contener atributos, pero estos son siempre públicos, estáticos y finales.

Las interfaces son implementadas por las clases. Del mismo modo que con las clases abstractas, la clase que implemente una o varias interfaces, está obligada a darle un comportamiento a los métodos que defina la(s) interfaz(ces). La sintaxis para que una clase pueda implementar una interfaz es la siguiente:

```
class NombreClase implements NombreInterfaz {
        // definición de los métodos que define la interfaz
}
```

Al igual que con las clases abstractas, es posible crear referencias de una interfaz:

*InstrumentoMusical instrumento;*

Sin embargo, **no se puede instanciar**, es decir:

*InstrumentoMusical instrumento = new InstrumentoMusical();*

Lo que sí se puede hacer, como en las clases abstractas, es crear una referencia abstracta que contenga un objeto concreto, es decir:

*InstrumentoMusical instrumento = new Flauta();*

Ejemplo:

```
public interface InstrumentoMusical {
    void tocar();

    void afinar();

    String tipoInstrumento();
}
```

```
public class InstrumentoViento extends Object implements InstrumentoMusical {
    // Por defecto, todos los métodos declarados en una interfaz son
    // públicos, por tanto, se deben implementar con el mismo nivel de acceso
    @Override
    public void tocar() {
        System.out.println("Tocando un instrumento de viento");
    }

    @Override
    public void afinar() {
        System.out.println("Afinando un instrumento de viento");
    }

    @Override
    public String tipoInstrumento() {
        return "Instrumento de viento";
    }

}
```

```java
public class Flauta extends InstrumentoViento {
    // La clase Flauta puede modificar el comportamiento
    // de alguno de los métodos heredados

    @Override
    public String tipoInstrumento() {
        return "Flauta";
    }
}


public class PruebaInstrumento {
    public static void main (String [] args) {
        // Se puede crear una referencia del tipo interfaz
        InstrumentoMusical instrumento;

        // Pero no se posible crear una instancia de una interfaz
        // instrumento = new InstrumentoMusical();

        // Una referencia del tipo interfaz, puede ser asignada
        // a cualquier instancia que la implemente
        instrumento = new Flauta();
        instrumento.tocar();
        System.out.println(instrumento.tipoInstrumento());
    }
}
```

## Bibliografía


*Barnes David, Kölling Michael*
***Programación Orientada a Objetos con Java.***
*Tercera Edición.*
*Madrid*
*Pearson Educación, 2007*

*Deitel Paul, Deitel Harvey.*
***Como programar en Java***
*Septima Edición.*
*México*
*Pearson Educación, 2008*

*Martín, Antonio*
***Programador Certificado Java 2.***
*Segunda Edición.*
*México*
*Alfaomega Grupo Editor, 2008*

*Dean John, Dean Raymond.*
***Introducción a la programación con Java***
*Primera Edición.*
*México*
*Mc Graw Hill, 2009*

# Guía práctica de estudio 12: Encapsulamiento (2da parte)


***Elaborado por:***

M.C. M. Angélica Nakayama C.

Ing. Jorge A. Solano Gálvez

***Autorizado por:***

M.C. Alejandro Velázquez Mena

# Guía práctica de estudio 12: Encapsulamiento (2da parte)

## Objetivo:

Aplicar el concepto encapsulamiento para proteger la información dentro de una organización de clases.

## Actividades:

- Agrupar clases en paquetes.
- Importar clases de diversos paquetes.

## Introducción

Las clases de las bibliotecas estándar del lenguaje están organizadas en **jerarquías de paquetes**. Esta organización en jerarquías ayuda a que las personas encuentren clases particulares que requieren utilizar de manera fácil.

Está bien que varias clases tengan el mismo nombre si están en paquetes distintos. Así, encapsular grupos pequeños de clases en paquetes individuales según su funcionalidad permite reusar el nombre de una clase dada en diversos contextos.

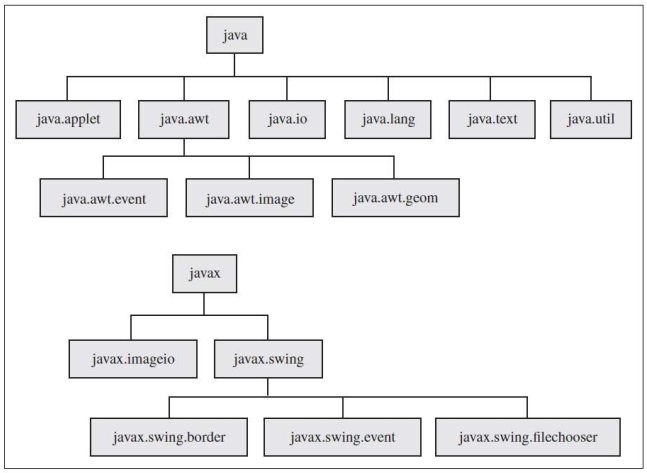
**NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.**

## Paquetes

Un **paquete** o **package** es una agrupación de clases. La API de Java cuenta con muchos **paquetes** que contienen clases agrupadas bajo un mismo propósito. Dado que la biblioteca de Java contiene miles de clases, es necesaria alguna estructura en la organización de la biblioteca para facilitar el trabajo con este enorme número de clases.

Java utiliza **paquetes** para acomodar las clases de la biblioteca en grupos que permanecen juntos. Las clases del API están organizadas en **jerarquías de paquetes**. Esta organización en jerarquías ayuda a encontrar clases particulares de forma eficiente.

A continuación, se muestra parte de la **jerarquía de paquetes** de la API de Java.




Los **paquetes** se utilizan con las finalidades siguientes:

- Para agrupar clases relacionadas.
- Para evitar conflictos de nombres. En caso de conflicto de nombres entre clases el compilador obliga a diferenciarlos usando su nombre calificado.
- Para ayudar en el control de la accesibilidad de clases y miembros.

El nombre completo o nombre calificado (**Fully Qualified Name**) de una clase debe ser único y está formado por el nombre de la clase precedido por los nombres de los sub-paquetes en donde se encuentra hasta llegar al paquete principal, separados por puntos.

Ejemplo:

*java.util.Random* es el **Fully Qualified Name** de la clase *Random* que se encuentra en el paquete *java.util*

## Importar paquetes

Las clases de Java que se almacenan en la biblioteca de clases, no están disponibles automáticamente para su uso. Para poder disponer de alguna de estas clases, se debe indicar en el código que se va usar una clase de la biblioteca usando su **fully qualfied name**.

Ejemplo:

```
public class PruebaPaquetes {

    public static void main(String[] args) {
        java.util.Random rnd = new java.util.Random();
        System.out.println(rnd.nextInt(100));
    }

}
```

Para hacer que las clases en un paquete particular estén disponibles para el programa que se está escribiendo, es necesario **importar** ese paquete, esto permite abreviar los nombres de las clases, variables y métodos, evitando el tener que escribir continuamente el nombre completo de la clase.

La sentencia **import** tiene la forma general:

***import*** *fullyQualifiedName;*

Estas sentencias deben ir antes de la declaración de la clase. Ejemplo:

```
import java.util.Random;

public class PruebaPaquetes {

    public static void main(String[] args) {
        Random rnd = new Random();
        System.out.println(rnd.nextInt(100));
    }

}
```

Java también permite importar paquetes completos con sentencias de la forma

***import*** *nombreDePaquete.*;*

Por ejemplo, la siguiente sentencia importaría todas las clases del paquete *java.util*:

***import*** *java.util.*;*

El **importar** un paquete no hace que se carguen todas las clases del paquete, sino que sólo se cargarán las clases **public** del paquete.

Al **importar** un paquete **no se importan los sub-paquetes**. Éstos deben ser importados explícitamente, pues en realidad son paquetes distintos. Por ejemplo, al importar *java.awt* no se importa *java.awt.event*.

Algunas clases se usan tan frecuentemente que casi todas las clases debieran importarlas. Estas clases se han ubicado en el paquete **java.lang** y este paquete se **importa automáticamente** dentro de cada clase. La clase *String* es un ejemplo de una clase ubicada en *java.lang*.

## Paquetes propios

El lenguaje Java permite crear sus **propios paquetes** para organizar clases definidas por el programador en jerarquías de paquetes.

Para que una clase pase a formar parte de un **paquete** hay que introducir en ella la sentencia:

***package*** *nombreDelPaquete;*

La cual debe ser la **primera sentencia del archivo** sin contar comentarios y líneas en blanco.

Los nombres de los **paquetes** se suelen escribir con minúsculas, para distinguirlos de las clases, que empiezan por mayúscula. El nombre de un paquete puede constar de varios nombres unidos por puntos.

Todas las clases que forman parte de un **paquete** deben estar en el mismo directorio. Los nombres compuestos de los paquetes están relacionados con la jerarquía de directorios en que se guardan las clases. Es recomendable que los nombres de las clases sean únicos en toda la biblioteca. Es el nombre del paquete lo que permite obtener esta característica. Una forma de conseguirlo es incluir el nombre del dominio.

Por ejemplo, la clase:

*mx.unam.fi.poo.MiClase.class*

Debería estar en:

*CLASSPATH\mx\unam\fi\poo\MiClase.class*

Ejemplo:

```
package mx.unam.fi.poo;

public class MiClase {
    public static void main(String[] args) {
        System.out.println("Clase empaquetada");
    }

}
```

## Compilación y ejecución de clases en paquetes

Se puede solicitar al compilador de Java que coloque en forma automática el archivo compilado .class en la ruta destino correspondiente. Para hacer lo anterior, debe invocar al compilador desde línea de comandos con la opción –d, como sigue:

***javac -d*** *rutaOrigen archivoFuente*

El nombre de ruta completo del directorio que obtiene el código compilado es *rutaOrigen/rutaDelPaquete.*

Si el directorio destino ya existe, entonces el archivo generado **.class** va a ese directorio. Si el directorio destino no existe, el compilador crea en forma automática el directorio requerido y luego inserta ahí el archivo generado **.class**. Por tanto, no es necesario crear explícitamente la estructura de directorios, se puede dejar que el compilador lo haga.

Ejemplo:

Se desea poner la clase *HolaMundo* en un paquete llamado *hola*, para tal efecto, se modifica el código fuente de la siguiente manera:

```
package hola;
public class HolaMundo {
    public static void main(String[] args) {
        System.out.println("Hola Mundo");
    }
}
```

Para que se genere el archivo **HolaMundo.class** dentro del directorio hola, se compila con la opción –d y la ruta origen sería el directorio actual, es decir punto ( . ):

```
D:\>dir hola*
 El volumen de la unidad D es DATA
 El número de serie del volumen es: 9CD3-2B2F

 Directorio de D:\

21/06/2016  09:43 p. m.               130 HolaMundo.java
               1 archivos            130 bytes
               0 dirs  741,121,339,392 bytes libres

D:\>javac -d . HolaMundo.java

D:\>dir hola*
 El volumen de la unidad D es DATA
 El número de serie del volumen es: 9CD3-2B2F

 Directorio de D:\

21/06/2016  09:47 p. m.    <DIR>          hola
21/06/2016  09:43 p. m.               130 HolaMundo.java
               1 archivos            130 bytes
               1 dirs  741,121,339,392 bytes libres
```

En este caso se generó un directorio llamado hola, dentro del cual se generó el archivo **HolaMundo.class** correspondiente a la clase compilada.

```
D:\>dir hola
 El volumen de la unidad D es DATA
 El número de serie del volumen es: 9CD3-2B2F

 Directorio de D:\hola

21/06/2016  09:47 p. m.    <DIR>          .
21/06/2016  09:47 p. m.    <DIR>          ..
21/06/2016  09:47 p. m.               427 HolaMundo.class
               1 archivos            427 bytes
               2 dirs  741,121,339,392 bytes libres

D:\>
```

Para poder ejecutar correctamente esta nueva clase compilada, se debe hacer usando su **Fully Qualified Name** desde la ruta origen, ya que si solo se invoca el intérprete **java** con el nombre de la clase la ejecución fallará dado que no reconoce la clase.

```
D:\>java HolaMundo
Error: no se ha encontrado o cargado la clase principal HolaMundo

D:\>java hola.HolaMundo
Hola Mundo

D:\>
```

## Encapsulamiento con paquetes

Como se mencionó en la práctica anterior, el nivel de **acceso por defecto** cuando no se especifica alguno es el de paquete, esto es, los atributos que no tienen asignado un nivel de acceso solo pueden ser accedidos por clases que se encuentran **dentro del mismo paquete**. Por otro lado, el nivel de acceso protegido (**protected**) permite que un elemento sea **accedido fuera del paquete** mientras exista una **relación de herencia** entre la clase que contiene al elemento y la clase que lo intenta acceder. A esto se le conoce como encapsulamiento en paquetes, ya que también se puede proteger la información negando el acceso a los datos según se requiera.

Ejemplo

Dada la siguiente disposición de clases:


La clase A se encuentra dentro del paquete paquete.abecedario y tiene dos atributos, uno protegido y otro de paquete (sin modificador).

```java
package paquete.abecedario;

public class A {
    protected int uno;
    int dos;
}
```

La clase Alfa se encuentra dentro del paquete paquete.abecedario y hereda de A. Por el hecho de heredar tiene acceso al atributo protegido (uno), pero no al atributo de paquete (dos).

```java
package paquete.alfabeto;
import paquete.abecedario.A;


public class Alfa extends A {
    public static void main(String[] args) {
        Alfa alfa = new Alfa();
        System.out.println(alfa.uno);
        System.out.println(alfa.dos);
    }
}
```

Al compilar la clase se genera la siguiente salida:

```
paquete/alfabeto/Alfa.java:16: error: dos is not public in A; cannot be accessed from outside package
        System.out.println(alfa.dos);
                           ^
1 error
```

La clase Beta se encuentra dentro del paquete paquete.abecedario (igual que Alfa), pero no hereda de nadie, por lo tanto, no tiene acceso a ninguna de las variables de A, ya que solo podría acceder a variables públicas de A.

```
package paquete.alfabeto;

public class Beta {
    public static void main(String[] args) {
        Beta beta = new Beta();
        System.out.println(beta.uno);
        System.out.println(beta.dos);
    }
}
```

Al compilar la clase se genera la siguiente salida:

```
paquete/alfabeto/Beta.java:12: error: cannot find symbol
        System.out.println(beta.uno);
                           ^
  symbol:   variable uno
  location: variable beta of type Beta
paquete/alfabeto/Beta.java:13: error: cannot find symbol
        System.out.println(beta.dos);
                           ^
  symbol:   variable dos
  location: variable beta of type Beta
2 errors
```

# Bibliografía


*Barnes David, Kölling Michael*
***Programación Orientada a Objetos con Java.***
*Tercera Edición.*
*Madrid*
*Pearson Educación, 2007*

*Deitel Paul, Deitel Harvey.*
***Como programar en Java***
*Septima Edición.*
*México*
*Pearson Educación, 2008*

*Joyanes, Luis*
***Fundamentos de programación. Algoritmos, estructuras de datos y objetos.***
*Cuarta Edición*
*México*
*Mc Graw Hill, 2008*

*Dean John, Dean Raymond.*
***Introducción a la programación con Java***
*Primera Edición.*
*México*
*Mc Graw Hill, 2009*