

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	1/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Manual de prácticas del laboratorio de Programación orientada a objetos

Elaborado por:	Revisado por:	Autorizado por:	Vigente desde:
Jorge A. Solano	Laura Sandoval Montaño	Alejandro Velázquez Mena	20 de enero de 2017

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	2/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Índice de prácticas

No	Nombre
1	Entorno y lenguaje de programación
2	Fundamentos y sintaxis del lenguaje.
3	Utilerías y clases de uso general.
4	Clases y objetos.
5	Abstracción y Encapsulamiento
6	Organización de clases
7	Herencia
8	Polimorfismo
9	UML
10	Excepciones y errores
11	Manejo de archivos
12	Hilos
13	Patrones de diseño

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	3/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	4/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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 lenguaje.
- Revisar la instalación y configuración el entorno de ejecución.
- Realizar un programa en el lenguaje de programación 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 dicho 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 realizar un programa sencillo y realizar 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 esta

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	5/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

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.

El entorno de ejecución

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

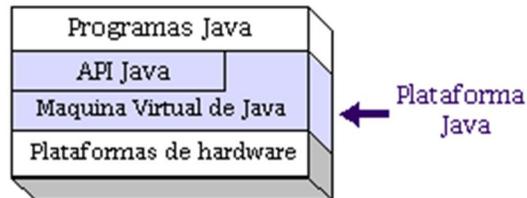
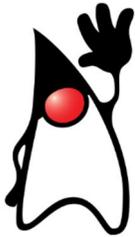
El **lenguaje de programación Java** fue originalmente desarrollado por James Gosling de Sun Microsystems (la cual fue adquirida por la compañía Oracle) y publicado en 1995 como un componente fundamental de la plataforma Java de Sun Microsystems. Su sintaxis deriva en gran medida de C y C++, pero tiene 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 Java, 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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	6/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			



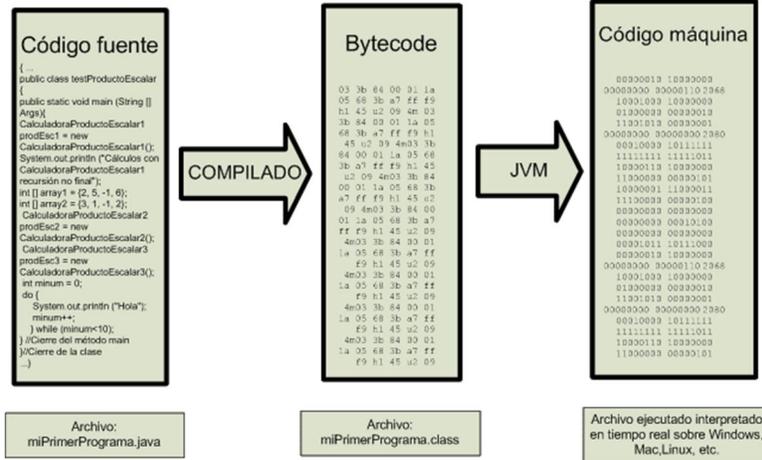
La Máquina Virtual Java (JVM)

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	7/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			



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 Java con su máquina virtual.

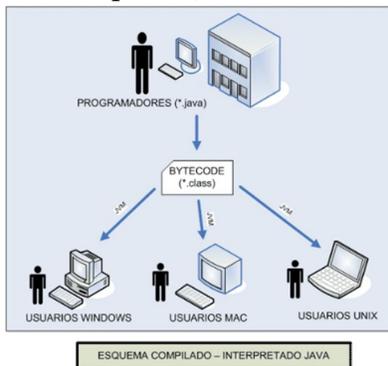
	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	8/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Programas en Java

Todo programa en Java está organizado en clases, estas 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 **.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**.

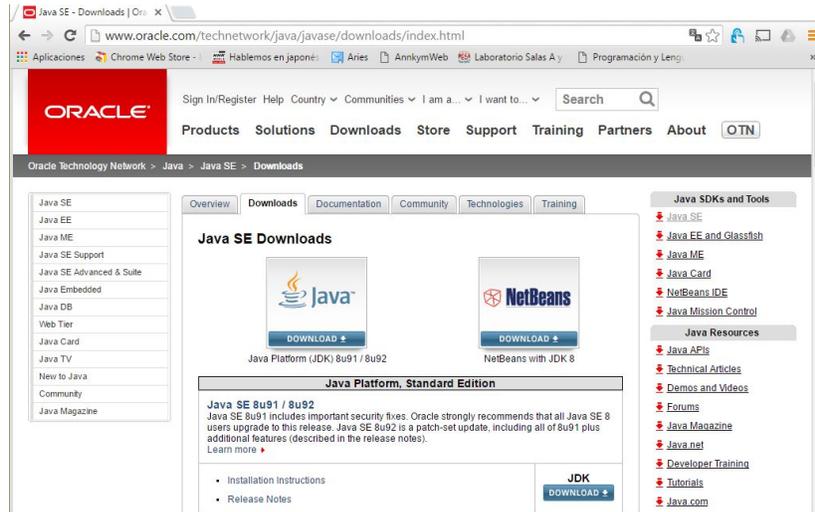


Herramientas de desarrollo (JDK)

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.

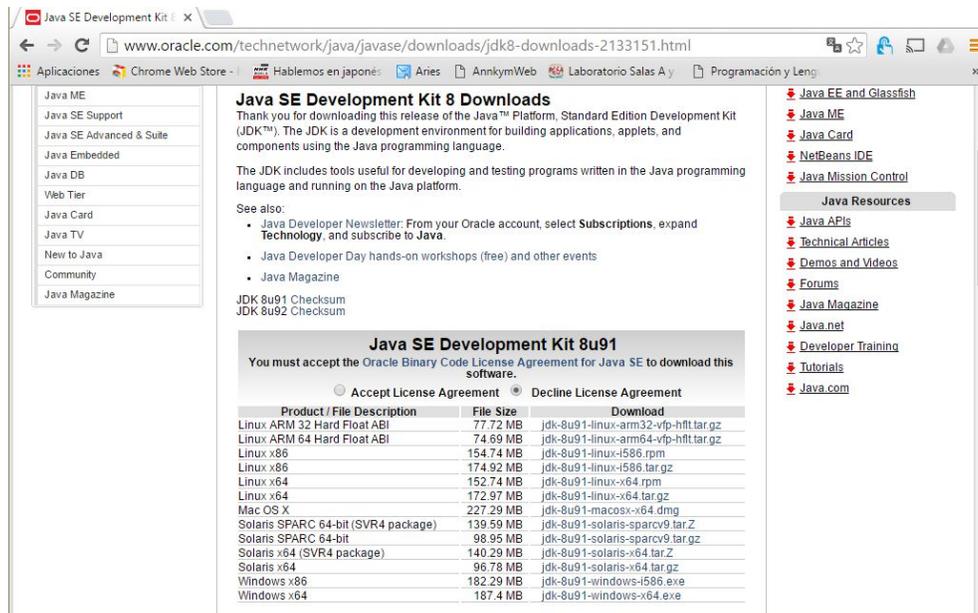
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	9/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			



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

En esta guía se utilizará la versión más reciente de Java SE (Standard Edition).



	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	10/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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

Instrucciones de instalación en distintos sistemas operativos en el sitio:

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

Programando en JAVA

Aunque aún carecemos del conocimiento del lenguaje, podemos iniciar con un sencillo **programa en Java** que imprima un saludo. Este programa nos va a servir para conocer el procedimiento general que se debe seguir para crear, compilar y ejecutar programas Java.

Codificación

Utilizando cualquier **editor de texto plano** (Block de notas, notepad++, gedit, vi, etc.) procedemos a capturar 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 procedemos a 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**.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	11/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

C:\> Símbolo del sistema
D:\>javac HolaMundo.java
D:\>dir HolaMundo*.*
El volumen de la unidad D es DATA
El número de serie del volumen es: 9CD3-2B2F

Directorio de D:\
01/06/2016  06:24 p. m.          422 HolaMundo.class
01/06/2016  06:24 p. m.          113 HolaMundo.java
                2 archivos             535 bytes
                0 dirs  747,109,953,536 bytes libres

D:\>

```

En caso de que existieran errores sintácticos en el código fuente, el compilador nos habría informado de ello y el archivo `.class` no se generaría.

Por ejemplo, si en el código fuente escribimos `system` en vez de `System`, al intentar compilar obtendríamos:

```

C:\> Símbolo del sistema
D:\>javac HolaMundo.java
HolaMundo.java:3: error: package system does not exist
    system.out.println("Hola Mundo");
    ^
1 error
D:\>_

```

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 nuestro caso será `HolaMundo` ya que es la única clase existente.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	12/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

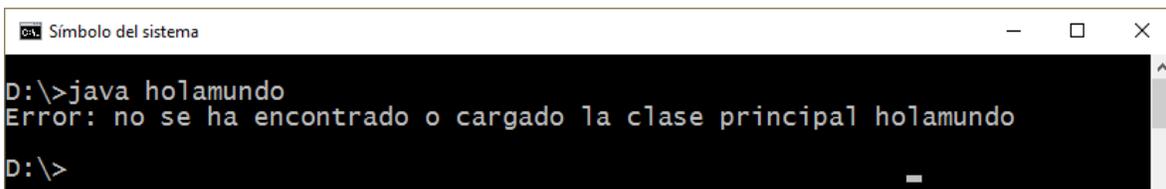


```

C:\> java HolaMundo
Hola Mundo
C:\>

```

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.

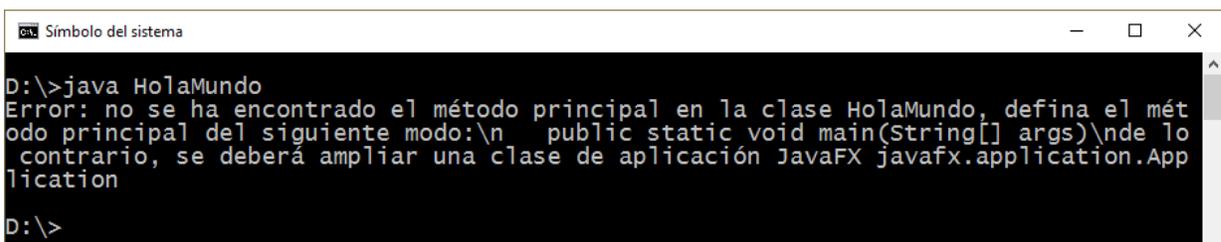


```

C:\> java holamundo
Error: no se ha encontrado o cargado la clase principal holamundo
C:\>

```

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



```

C:\> java HolaMundo
Error: no se ha encontrado el método principal en la clase HolaMundo, defina el método principal del siguiente modo:\n public static void main(String[] args)\nde lo contrario, se deberá ampliar una clase de aplicación JavaFX javafx.application.App
lication
C:\>

```

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

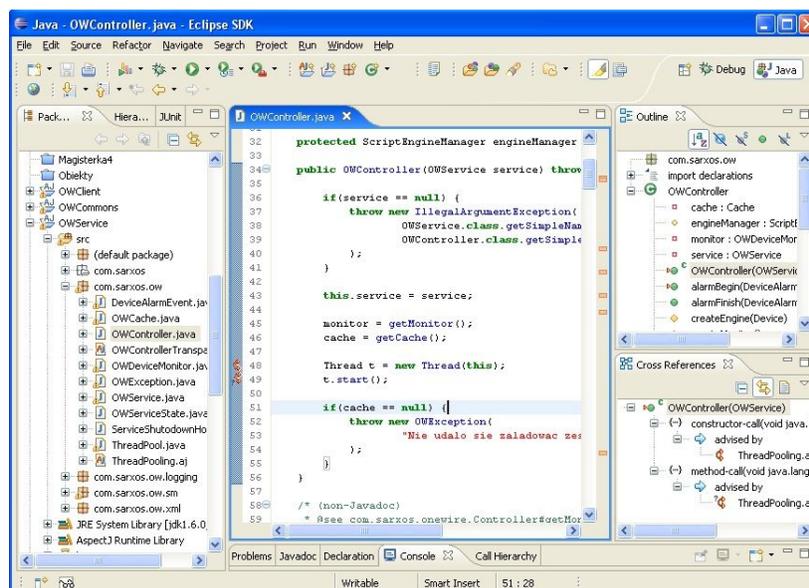
	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	13/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

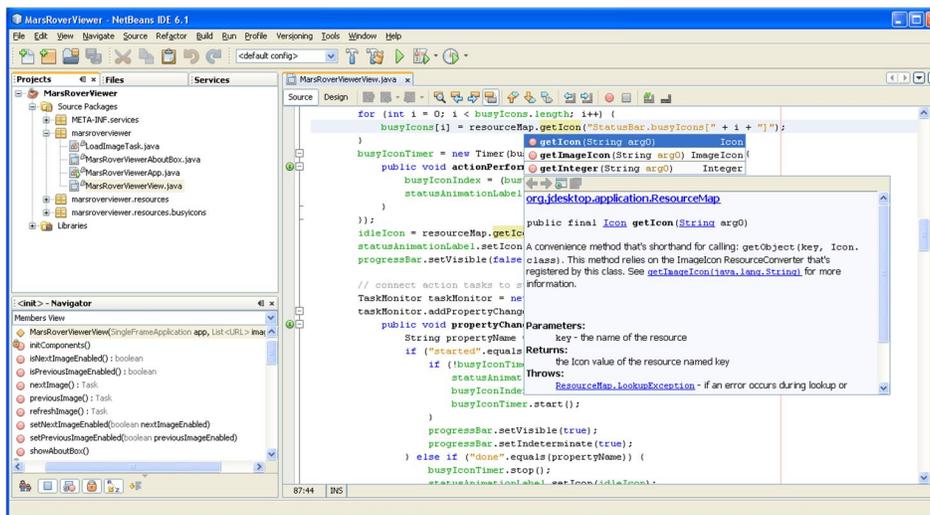
Un **entorno de desarrollo integrado** o **IDE (Integrated Development Environment)** es una aplicación que facilita el desarrollo de aplicaciones en algún lenguaje de programación. De manera general, un IDE es una interfaz gráfica de usuario diseñada para ayudar a los desarrolladores a construir aplicaciones 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 le ofrecerá una forma alternativa de utilización, basada en menús y barras de herramientas.



	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	14/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

La escritura de código también resulta una tarea sencilla con un IDE ya que estos suelen contar con un editor de código que resalta las palabras reservadas del lenguaje para distinguirlas del resto del código. Algunos incluso permiten auto escritura 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.

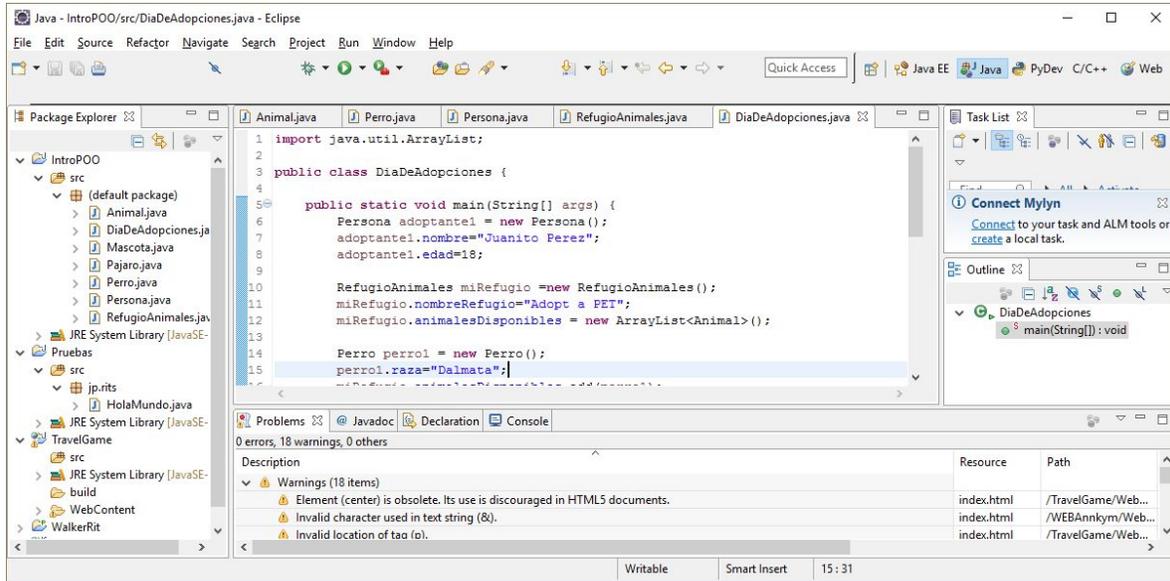


En el mercado existen diversos tipos de IDE, cada uno con características propias, empero, una constante es que permiten manejar las etapas para generar un programa dependiendo del tipo de lenguaje utilizado. La mecánica de utilización de estos programas es muy similar, todos ellos se basan en el concepto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

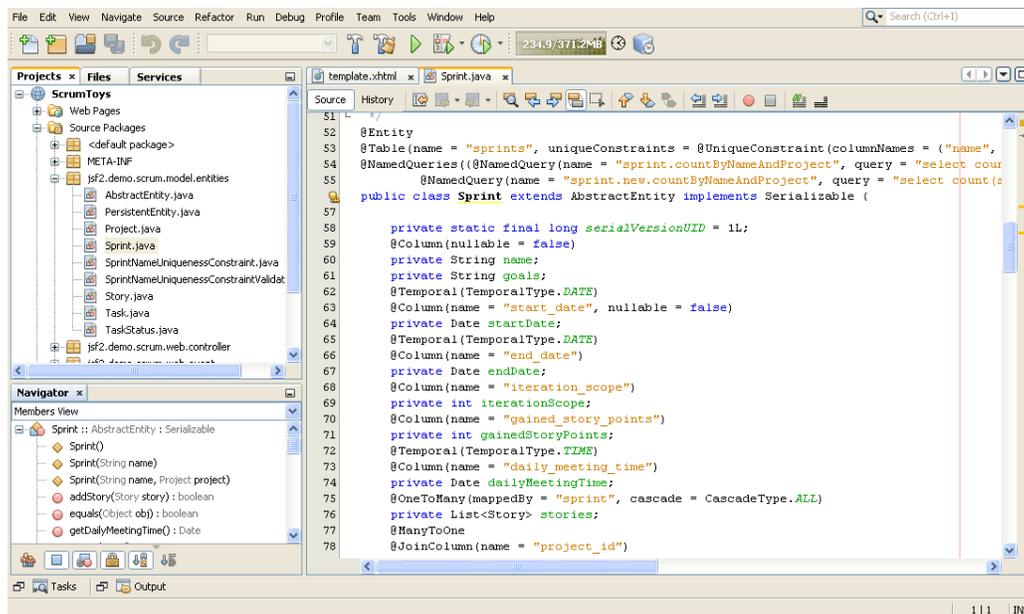
Algunos de los IDE más populares para Java son:

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	15/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Eclipse

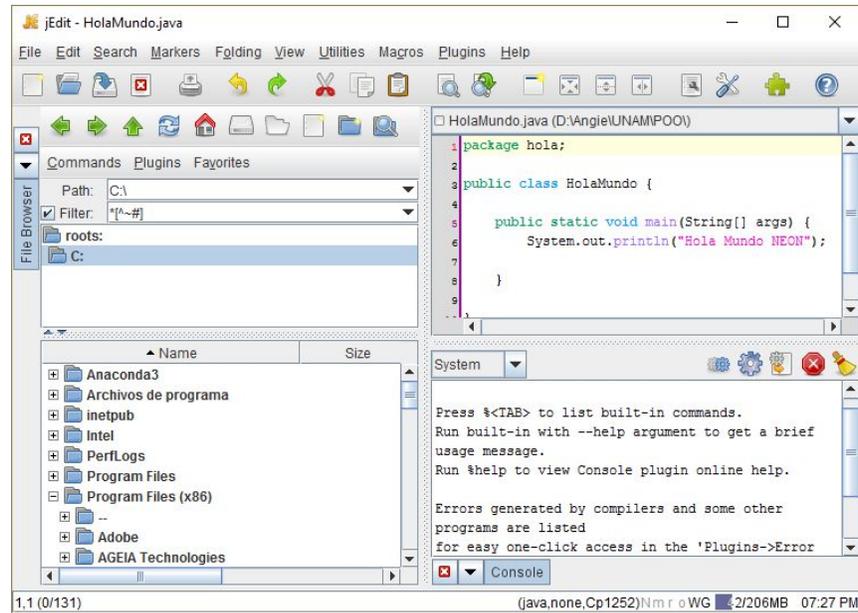


NetBeans

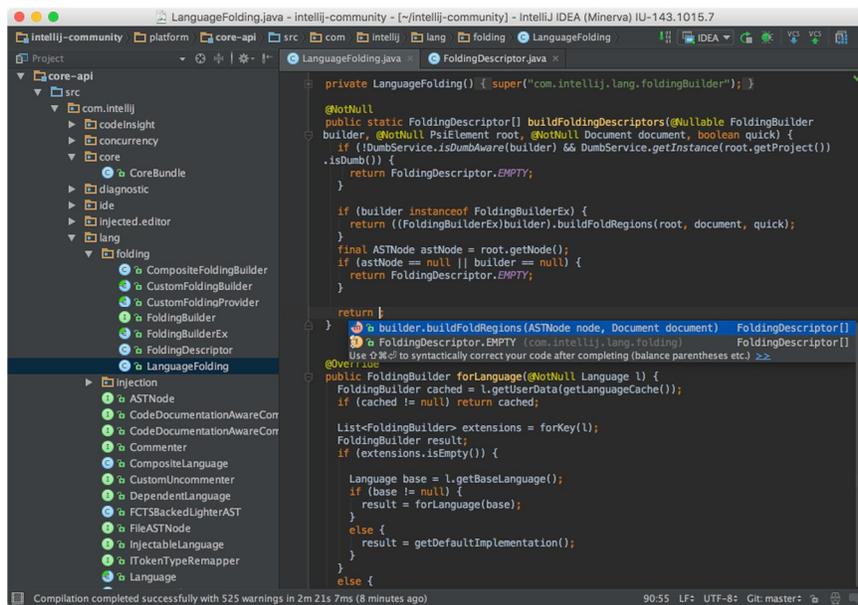


	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	16/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

jEdit

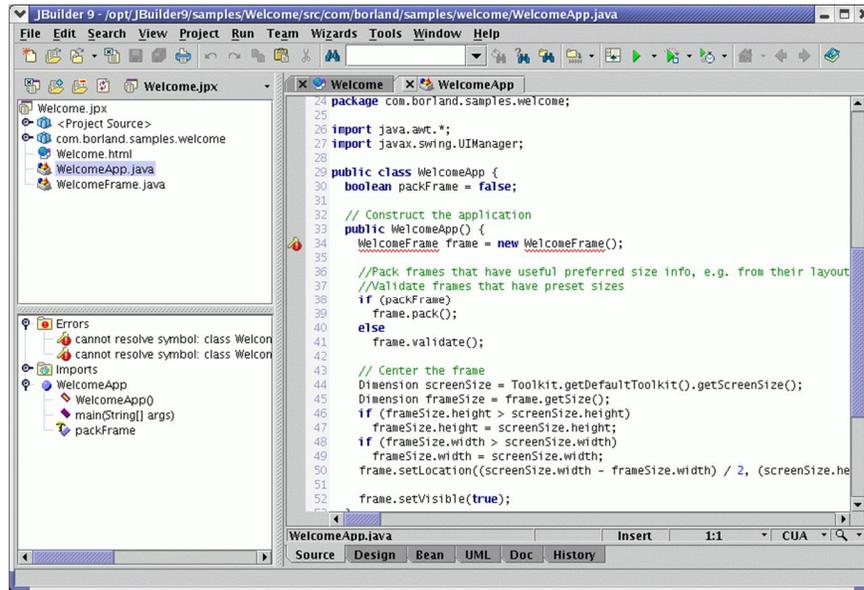


IntelliJ IDEA

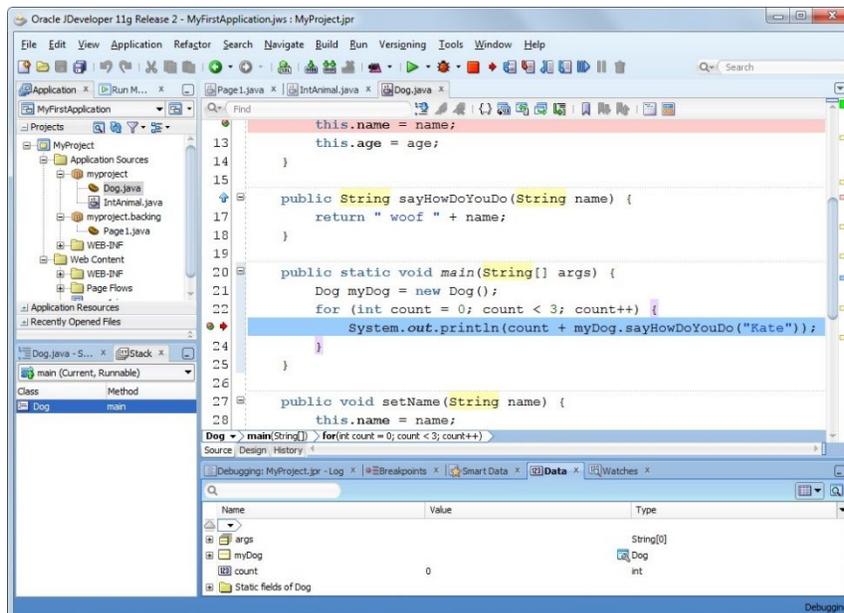


	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	17/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

JBuilder



JDeveloper



	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	18/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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).

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	19/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Para el ejemplo:



```
Símbolo del sistema
D:\>java MiClase
Esta es mi clase
```

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 público (**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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	20/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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 empieza el bloque comentado*

*y aquí acaba */*

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

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	21/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Identificadores

En Java los identificadores comienzan por una letra del alfabeto inglés, un subrayado `_` o el símbolo de dólar `$`, los siguientes caracteres del identificador pueden ser letras o dígitos. No se debe nunca iniciar con un dígito. 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 es habitual poner 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 las convenciones, 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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	22/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Las convenciones de código o **Java Code Conventions**, pueden ser consultadas en el siguiente link.

<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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	23/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	24/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 02: 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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	25/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 02: Fundamentos y sintaxis del lenguaje

Objetivo:

Crear programas que implementen variables y constantes de diferentes tipos de datos, expresiones y estructuras de control de flujo.

Actividades:

- Crear variables y constantes de diferentes tipos de datos.
- Crear diversas expresiones (operadores, declaraciones, etc.).
- Implementar estructuras de control de flujo (if/else, switch, for, while, etc.).

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	26/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

Además de estos elementos básicos, existen otros elementos que forman parte de los programas, cuya comprensión y funcionamiento será vital para la correcta codificación de un programa, por ejemplo, tipos de datos y estructuras de control de flujo.

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.

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	27/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	28/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

Variables

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 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 más en 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** y el **nombre** de dicha variable. Estas variables pueden ser tanto de tipos primitivos como referencias a objetos de alguna clase perteneciente al API de Java o generada por el usuario.

tipoDeDato *nombreVariable*;

Ejemplo:

int *miVariable*; **float** *area*; **char** *letra*; **String** *cadena*; **MiClase** *prueba*;

Si no se especifica un valor en su declaración, las variables primitivas se inicializan a **cero** (salvo boolean y char, que se inicializan a false y '\0'). Análogamente las variables de tipo referencia son inicializadas por defecto a un valor especial: **null**.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	29/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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 ese 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 una 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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	30/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Constantes

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

```
final tipoDato nombreDeConstante = valor;
```

Ejemplo:

```
final double PI = 3.1416;
```

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 el 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 usando el método **close**.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	31/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Para el ejemplo:

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

Estas clases y sus métodos se verán más a detalle en la práctica de Manejo de archivos.

Operadores

Java es un lenguaje rico en operadores, que 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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	32/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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** o **false** según el objeto pertenezca o no a la clase.

Operador condicional ? :

Este operador, 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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	33/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	34/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	35/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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	= += -= *= /= %= &x= ^= = <<= >>= >>>=

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	36/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

Sentencias o expresiones

Una expresión es un conjunto de variables unidos por operadores. Son órdenes que se le dan 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.

Existen dos bifurcaciones diferentes: **if** y **switch**.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	37/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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;
}
```

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 **if**.

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;
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	38/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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;]
}

```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	39/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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**).
- No puede haber dos etiquetas **case** con el mismo valor.
- Los valores no comprendidos en ninguna 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**.

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;
}
```

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	40/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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);
```

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.

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.

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).

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	41/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ejemplos usando estructuras de control:

Programa que suma los números pares comprendidos entre n1 y n2.

```
import java.util.Scanner;
public class SumaPares {
    public static void main(String[] args) {
        //Declaración de variables
        int n1 , n2;
        int suma = 0;
        int mayor, menor;
        Scanner sc = new Scanner(System.in);
        //Pedir datos al usuario
        System.out.println("Por favor introduzca un número entero");
        n1 = sc.nextInt();
        System.out.println("Introduzca otro número entero");
        n2 = sc.nextInt();
        //Validar cual es el número mayor y el menor
        if (n1 > n2){
            mayor = n1;
            menor = n2;
        } else {
            mayor = n2;
            menor = n1;
        }
        //Hacer un ciclo desde el menor hasta el mayor
        for(int i = menor; i <= mayor; i++){
            //Validar si es par para sumarlo
            if( i % 2 == 0){
                suma += i;
            }
        }
        //Imprimir el resultado
        System.out.println("La suma de los pares entre " + n1 + " y " + n2 +" es " + suma);
        sc.close();
    }
}
```

Programa que calcula el área de una figura geométrica dependiendo la opción seleccionada por el usuario en un menú que se repite hasta seleccionar la opción "Salir".

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	42/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

import java.util.Scanner;
public class FigurasGeometricas {
    public static void main(String[] args) {
        float area;
        int opcion;
        final float PI = 3.14159f;
        Scanner sc = new Scanner(System.in);
        do{
            System.out.println("Elige la opción");
            System.out.println("1.-Area de círculo");
            System.out.println("2.-Area de triángulo");
            System.out.println("3.-Area de cuadrado");
            System.out.println("4.-Salir");
            opcion = sc.nextInt();
            switch (opcion) {
                case 1:
                    //Círculo
                    System.out.println("Seleccionó el círculo");
                    float radio = 15;
                    area = PI * radio * radio;
                    break;
                case 2:
                    //Triángulo
                    System.out.println("Seleccionó el triángulo");
                    float base = 8, altura = 5;
                    area = ( base * altura ) / 2;
                    break;
                case 3:
                    //Cuadrado
                    System.out.println("Seleccionó el cuadrado");
                    float lado = 10;
                    area = lado * lado;
                    break;
                case 4:
                    //Salir
                    System.out.println("Hasta luego");
                    continue;
                default:
                    //Ninguno de los anteriores
                    System.out.println("Su elección no es correcta");
                    continue;
            }
            System.out.println("El area es: " + area);
        } while (opcion != 4);
        sc.close();
    }
}

```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	43/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	44/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 03: Utilerías y clases de uso general



Elaborado por:

M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:

M.C. Alejandro Velázquez Mena

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	45/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 03: Utilerías y clases de uso general

Objetivo:

Utilizar bibliotecas propias del lenguaje para realizar algunas tareas comunes y recurrentes.

Actividades:

- Conocer las bibliotecas del lenguaje.
- Utilizar algunas clases propias de la biblioteca del lenguaje.

Introducción

Al trabajar en un problema de programación, normalmente se debe verificar si hay clases pre-construidas que satisfagan las necesidades del programa. Si existen esas clases, entonces hay que utilizarlas: *“no tratar de reinventar la rueda”*.

Hay dos ventajas principales de usar clases pre-construidas: se puede ahorrar tiempo ya que no es necesario escribir nuevas; y el uso de clases pre-construidas también puede mejorar la calidad de los programas ya que han sido probadas completamente, depuradas y sometidas a un proceso de escrutinio para asegurar su eficiencia.

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	46/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Arreglos

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 primer índice. 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 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]

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	47/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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 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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	48/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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:

```
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:



```
Símbolo del sistema
D:\>java Ejemplo hola que tal
hola
que
tal
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	49/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

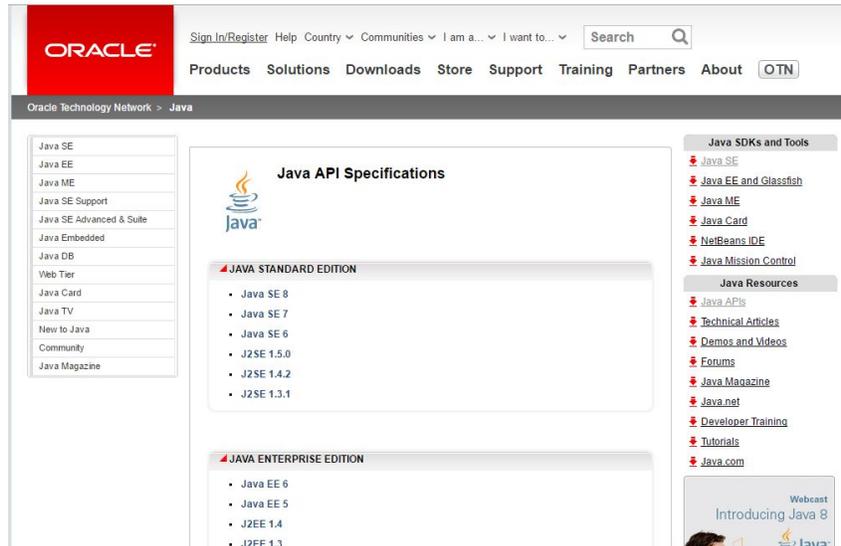
La información completa del **API** de java se denomina **especificación** y sirve para conocer cualquier aspecto sobre las clases que contiene la **API**. Esta especificación es de crucial importancia para los programadores ya que en ella se pueden consultar los detalles de alguna clase que se quiera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

La especificación del **API** de java se encuentra en el sitio de Oracle.

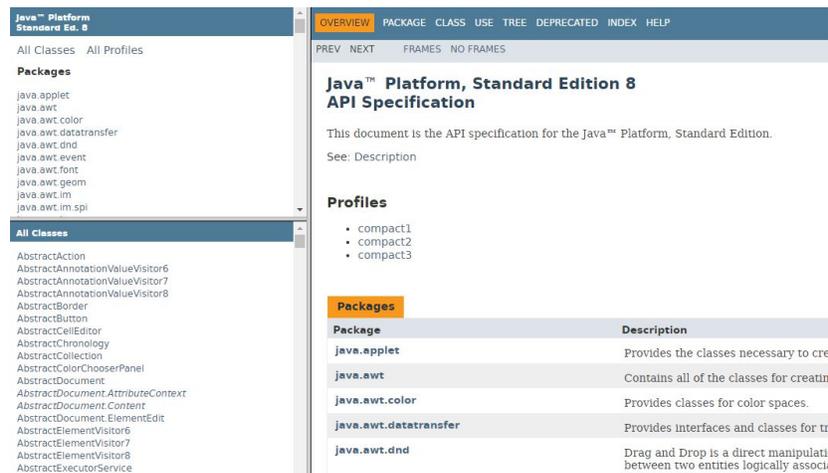
<http://www.oracle.com/technetwork/java/api-141528.html>

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	50/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			



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



	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	51/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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 con 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";
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	52/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

String t = s + " que 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");
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	53/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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: 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);
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	54/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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 número 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);
```

Autoboxing

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;
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	55/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Es decir, la creación del objeto **wrapper** se produce implícitamente al asignar el dato 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;
```

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	56/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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 **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 sintaxis:

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	57/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	58/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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 posición 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>();
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	59/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	60/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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);
        }
    }
}
```

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

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	61/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ejemplo:

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

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

Date y Calendar

En 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, sin embargo, 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.

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());

Calendar 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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	62/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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 introducción 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));
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	63/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	64/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 04: Clases y objetos



Elaborado por:

M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:

M.C. Alejandro Velázquez Mena

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	65/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 04: Clases y objetos

Objetivo:

Aplicar los conceptos básicos de la programación orientada a objetos en un lenguaje de programación.

Actividades:

- Crear clases.
- Crear objetos o instancias.
- Invocar métodos.
- Utilizar constructores.

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, es decir, 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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	66/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

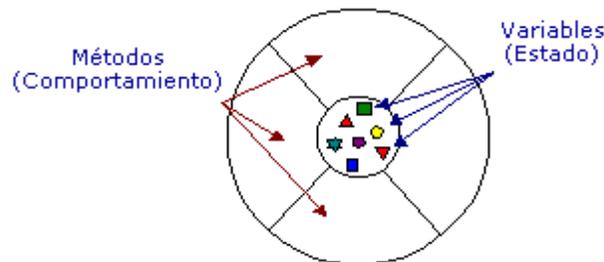


Figura 1. Conceptualización de un objeto en POO.

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. Una **clase** es una plantilla que define las variables y los métodos que son comunes para todos los objetos de un cierto tipo.

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).

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	67/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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

class NombreDeLaClase

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:

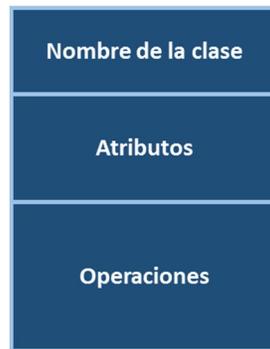


Figura 2. Diagrama de clase en UML.

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();

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	68/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

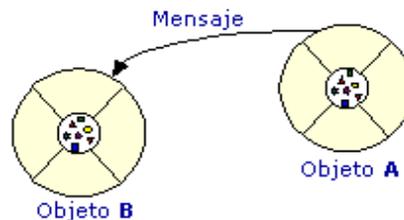


Figura 2. Mensajes entre objetos.

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	69/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.y=2;
        x.imprimePunto();
    }
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	70/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Sobrecarga de métodos (overload)

La **sobrecarga de métodos** permite usar el **mismo nombre** de un método en una clase, pero con **diferentes argumentos** (y, opcionalmente, con diferentes valores de retorno). Los métodos sobrecargados hacen más cómoda la implementación de un objeto y, por ende, la utilización de los métodos del mismo.

Los métodos sobrecargados:

- Deben tener el mismo nombre.
- Deben cambiar la lista de argumentos.
- Pueden cambiar el valor de retorno.
- Pueden cambiar el nivel de acceso.

Java permite métodos **sobrecargados** (overloaded), es decir, métodos distintos que tienen el mismo nombre, pero que se diferencian por el número y/o tipo de los argumentos.

Por ejemplo:

```
public void imprimePunto()
```

```
public void imprimePunto(int x, int y)
```

```
public void imprimePunto(float r, float th)
```

A la hora de llamar a un método sobrecargado, Java sigue algunas reglas para determinar el método concreto que debe llamar:

- Si existe el método cuyos argumentos se ajustan exactamente al tipo de los argumentos de la llamada (argumentos actuales), se llama ese método.
- Si no existe un método que se ajuste exactamente, se intenta promover los argumentos actuales al tipo inmediatamente superior (**cast implícito**) y se llama el método correspondiente.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	71/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

- Si sólo existen métodos con argumentos de un tipo más restringido, el programador debe hacer un **cast** explícito en la llamada, responsabilizándose de esta manera de lo que pueda ocurrir.
- El valor de retorno no influye en la elección del método sobrecargado. En realidad es imposible saber desde el propio método lo que se va a hacer con él. No es posible crear dos métodos sobrecargados, es decir con el mismo nombre, que sólo difieran en el valor de retorno.

Ejemplo:

```

public class Triangulo {
    float base, altura;

    public float area(){
        return (float)((this.base * this.altura)/2.0);
    }

    public float area(float base, float altura){
        return (float)((base * altura)/2.0);
    }

    public float area(int base, int altura){
        return (float)((base * altura)/2.0);
    }
}

public class PruebaTriangulo {
    public static void main (String [] args){
        Triangulo triangulo = new Triangulo();
        triangulo.base = 5;
        triangulo.altura = 8;
        System.out.println("base: " + triangulo.base);
        System.out.println("altura: " + triangulo.altura);
        // método área sobrecargado
        System.out.println("area() -> " + triangulo.area());
        System.out.println("area(6, 2) -> " + triangulo.area(6, 2));
        System.out.println("area(5.5f, 3.2f) -> " + triangulo.area(5.5f, 3.2f));
    }
}

```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	72/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	73/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	74/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

```
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).

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	75/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

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).

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	76/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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**.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	77/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	78/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 05: Abstracción y Encapsulamiento



Elaborado por:

M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:

M.C. Alejandro Velázquez Mena

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	79/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 05: Abstracción y encapsulamiento

Objetivo:

Aplicar el concepto de abstracción para el diseño de clases que integran una solución, utilizando el encapsulamiento para proteger la información y ocultar la implementación.

Actividades:

- Obtener las características y funcionalidades principales de un objeto dado.
- Utilizar diferentes niveles de acceso a las características y funcionalidades obtenidas.

Introducción

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

El **encapsulamiento** sucede cuando algo es envuelto en una capa protectora. Cuando el **encapsulamiento** se aplica a los objetos, significa que los datos del objeto 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, los métodos 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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	80/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Abstracción

La **abstracción** es la propiedad que considera los aspectos más significativos o notables de un problema y expresa una solución en esos términos. La **abstracción** posee diversos grados o **niveles de abstracción**, los cuales ayudan a estructurar la complejidad intrínseca que poseen los sistemas del mundo real.

En el análisis hay que concentrarse en **¿Qué hace?** y no en *¿Cómo lo hace?*

El principio de **abstracción** es más fácil de entender con una analogía del mundo real, por ejemplo, la televisión. Todos estamos familiarizados con sus características y su manejo manual o con el control remoto (encender, apagar, subir volumen, etc.) incluso sabemos cómo conectarlo con otros aparatos externos como altavoces o reproductores, o al internet. Sin embargo, no todos sabemos cómo funciona internamente, es decir, como recibe la señal, como la traduce y la visualiza en la pantalla, etc.

Normalmente sucederá que no sepamos cómo funciona el aparato de televisión pero si sabemos cómo utilizarlo. Esta característica se debe a que la televisión separa claramente su implementación interna de su **interfaz** externa. Interactuamos con la televisión a través de su **interfaz**: los botones de encendido, cambio de canal, volumen, etc. no conocemos el tipo de tecnología que utiliza, el método de generar la imagen en la pantalla o como funciona internamente, es decir, su implementación, ya que ello no afecta a su **interfaz**.

Encapsulamiento

La **encapsulación** o **encapsulamiento** significa reunir en una cierta estructura a todos los elementos que a 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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	81/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Por ejemplo, la televisión tiene un conjunto de botones para encender, apagar y/o 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, pero en general siguen el modelo de C++, 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 restantes clases 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. Ya que los atributos están **encapsulados** dentro de una clase, se necesitará definir cuáles son las clases que tienen acceso a visualizar y cambiar los atributos. Esta característica se conoce como **visibilidad de los atributos**. En general se recomienda visibilidad **privada** o **protegida** para los atributos.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	82/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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:

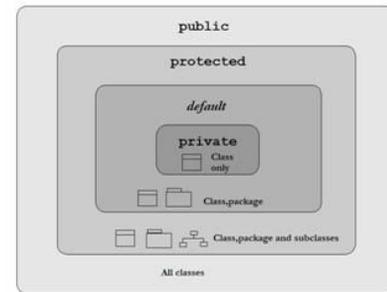
private String nombre;

public int operacion(int a, int b){

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	83/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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

Podemos 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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	84/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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:

Círculo
PI: REAL -radio: REAL
getRadio(): REAL setRadio(REAL) perimetro() area() toString(): CADENA

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	85/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

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 + "]";
    }
}

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 *setter*:

```

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 tener referencias a objetos de otras clases como miembros. A dicha capacidad se le conoce como **composición** y algunas veces como relación **“tiene un”**.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	86/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

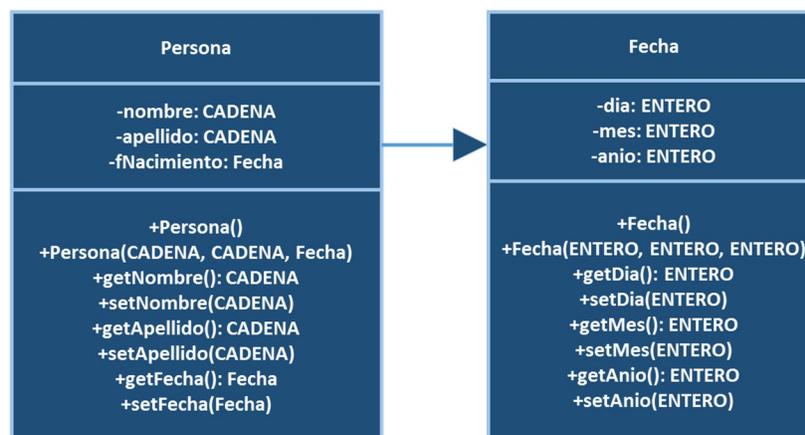
El concepto de **composición** no fue creado para la programación; suele usarse a menudo para objetos complejos en el mundo real. Toda criatura viviente y la mayor parte de los productos manufacturados están constituidos por partes. A menudo, cada parte es un subsistema que está integrado por su propio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





**Manual de prácticas del
Laboratorio de Programación
orientada a objetos**

Código:

MADO-22

Versión:

01

Página

87/208

Sección ISO

8.3

Fecha de
emisión

20 de enero de 2017

Facultad de Ingeniería

Área/Departamento:
Laboratorio de computación salas A y B

La impresión de este documento es una copia no controlada

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

    public Persona() {
    }

    public Persona(String nombre, String apellido, Fecha fNac:
        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;
    }

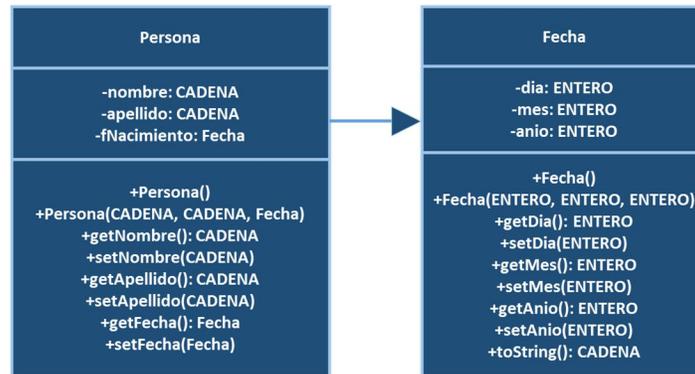
    public void setFNacimiento(Fecha fNacimiento) {
        this.fNacimiento = fNacimiento;
    }
}
```

```
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());
    }
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	88/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.





**Manual de prácticas del
Laboratorio de Programación
orientada a objetos**

Código:

MADO-22

Versión:

01

Página

89/208

Sección ISO

8.3

Fecha de
emisión

20 de enero de 2017

Facultad de Ingeniería

Área/Departamento:
Laboratorio de computación salas A y B

La impresión de este documento es una copia no controlada

```
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;
    }
}

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());
    }
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	90/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	91/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 06: Organización de clases



Elaborado por:

M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:

M.C. Alejandro Velázquez Mena

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	92/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 06: Organización de clases

Objetivo:

Organizar adecuadamente las clases según su funcionalidad o propósito bajo un *namespace* o paquete.

Actividades:

- Agrupar clase 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 requirieren utilizar.

Está bien que varias clases tengan el mismo nombre si están en paquetes distintos. Así, encapsular grupos pequeños de clases en paquetes individuales 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 este u otro lenguaje orientado a objetos.

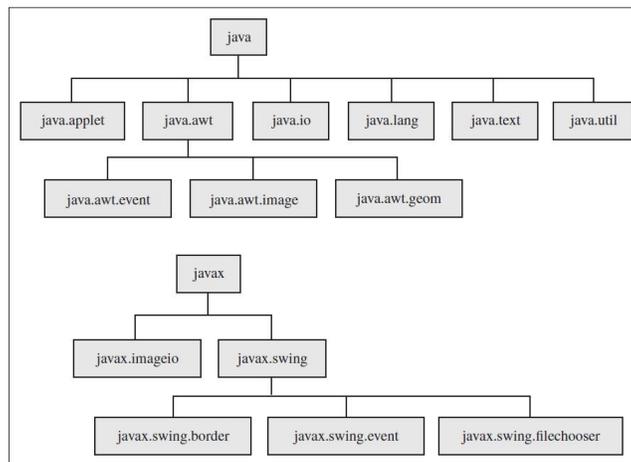
	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	93/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

A continuación, se muestra parte de la **jerarquía de paquetes** 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 diferenciarlos usando su nombre cualificado.
- 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 subpaquetes en donde se encuentra hasta llegar al paquete principal, separados por puntos.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	94/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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 se almacenan en la biblioteca de clases, pero no están disponibles automáticamente para su uso, tal como las otras clases del proyecto actual. 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 qualified 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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	95/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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*.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	96/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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 Internet. 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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	97/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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. (IDE típicas también proporcionan formas para hacer lo anterior).

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");
    }
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	98/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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:\>
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	99/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Distribución de aplicaciones

Una aplicación en Java está compuesta por varios archivos **.java**. Al compilarlos obtenemos varios archivos **.class** (uno por archivo **.java**), y no un único archivo ejecutable como ocurre en otros lenguajes. Además, a menudo la aplicación está formada no sólo por los archivos **.class** sino que requiere archivos adicionales (como archivos de texto, de configuración, iconos, etc.) lo que multiplica la cantidad de archivos que forman la aplicación compilada.

Todo esto hace que "*llevarse*" la aplicación para ejecutarla en una computadora diferente resulte un poco tedioso, sin mencionar que, olvidar cualquiera de los archivos que componen la aplicación significaría que ésta no va a funcionar correctamente.

Los archivos **JAR (Java ARchives)** permiten incluir en un sólo archivo varios archivos diferentes, almacenándolos en un formato **comprimido** para que ocupen menos espacio. Las siglas están deliberadamente escogidas para que coincidan con la palabra inglesa "**jar**" (tarro).

Es por tanto, algo similar a un archivo **.zip** pero con la particularidad de los archivos **.jar** **no necesitan ser descomprimidos para ser usados**, es decir que el intérprete de Java es capaz de ejecutar los archivos comprimidos en un archivo **JAR** directamente.

Los archivos **JAR**, contruidos sobre el formato de archivo **ZIP**, pueden recuperarse o desarrollarse desde cero utilizando los comandos y herramientas **JAR** proporcionadas por el **JDK**.

Un archivo **JAR** incluye una estructura de directorios con clases, lo anterior permite:

- Distribuir/utilizar clases de una manera eficiente a través de un solo archivo.
- Declarar dichas clases de una manera más eficiente en la variable CLASSPATH.
- En todo JDK se incluye el comando jar el cual permite generar, observar y descomprimir archivos JAR;

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	100/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Los archivos **JAR** contienen archivos de clases y recursos de la aplicación. En general un archivo **JAR** puede contener:

- Los archivos ***.class** que se generan a partir de compilar los archivos ***.java** que componen la aplicación.
- Los archivos de **recursos** que necesita la aplicación (Por ejemplo archivos de configuración, de texto, de sonido, imágenes, etc.)
- Opcionalmente se puede incluir los archivos de código fuente ***.java**
- Opcionalmente puede existir un archivo de configuración "**META-INF/MANIFEST.MF**".

Para que el archivo **JAR** sea ejecutable se debe incluir en el archivo MANIFEST.MF una línea indicando la clase que contiene el método estático *main()* que se usará para iniciar la aplicación. Este archivo se puede generar manualmente con cualquier editor de texto y es importante destacar que al final de la línea hay que agregar un salto de línea para que funcione. Si el archivo no se crea manualmente, se puede crear con la herramienta **JAR**.

Ejemplo:

Se tiene la clase *MiClase.java*, la cual se encuentra dentro del paquete *mx.unam.fi.poo*:

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

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	101/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Para compilarla y ejecutarla correctamente se utiliza lo siguiente:

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

Directorio de D:\pruebasJava

15/07/2016 06:26 p. m. <DIR>      .
15/07/2016 06:26 p. m. <DIR>      ..
15/07/2016 06:26 p. m.             146 MiClase.java
1 archivos                      146 bytes
2 dirs 710,417,502,208 bytes libres

D:\pruebasJava>javac -d . MiClase.java

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

Directorio de D:\pruebasJava

15/07/2016 06:26 p. m. <DIR>      .
15/07/2016 06:26 p. m. <DIR>      ..
15/07/2016 06:26 p. m.             146 MiClase.java
15/07/2016 06:26 p. m.             mx
1 archivos                      146 bytes
3 dirs 710,417,502,208 bytes libres

D:\pruebasJava>dir mx\unam\fi\poo
El volumen de la unidad D es DATA
El número de serie del volumen es: 9CD3-2B2F

Directorio de D:\pruebasJava\mx\unam\fi\poo

15/07/2016 06:26 p. m. <DIR>      .
15/07/2016 06:26 p. m. <DIR>      ..
15/07/2016 06:26 p. m.             440 MiClase.class
1 archivos                      440 bytes
2 dirs 710,417,502,208 bytes libres

D:\pruebasJava>java mx.unam.fi.poo.MiClase
Clase empaquetada
```

Sin embargo, se desea incluir esta clase en un archivo **JAR**. Para tal efecto, se utiliza la herramienta **jar** (incluida con el **JDK** al igual que **javac**).

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	102/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Además se desea que la herramienta genere el archivo MANIFEST.MF indicando que la clase principal (que contiene el *main*) es **mx.unam.fi.poo.MiClase**, entonces, se utiliza el siguiente comando:

```
D:\pruebasJava>jar -cvfe MiJar.jar mx.unam.fi.poo.MiClase mx/unam/fi/poo/*.class
manifiesto agregado
agregando: mx/unam/fi/poo/MiClase.class(entrada = 440) (salida = 303)(desinflado 31%)
D:\pruebasJava>dir
El volumen de la unidad D es DATA
El número de serie del volumen es: 9CD3-2B2F

Directorio de D:\pruebasJava

15/07/2016 06:32 p. m. <DIR> .
15/07/2016 06:32 p. m. <DIR> ..
15/07/2016 06:26 p. m. 146 MiClase.java
15/07/2016 06:32 p. m. 824 MiJar.jar
15/07/2016 06:26 p. m. <DIR> mx
2 archivos 970 bytes
3 dirs 710,417,498,112 bytes libres
```

Con esto se genera el archivo **MiJar.jar**, dentro del cual se encuentra la jerarquía de directorios correspondientes al paquete **mx.unam.fi.poo** con el archivo compilado **MiClase.class** y adicionalmente el archivo **MANIFEST.MF** dentro de un directorio **META-INF** (ambos creados por la herramienta) el cual contiene lo siguiente;

```
Manifest-Version: 1.0
Created-By: 1.8.0_45 (Oracle Corporation)
Main-Class: mx.unam.fi.poo.MiClase
```

Para verificar el contenido del archivo **JAR** se puede usar el siguiente comando:

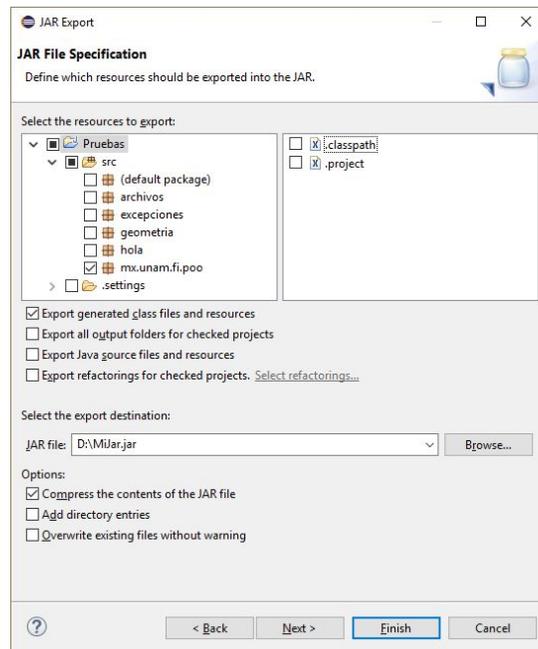
```
D:\pruebasJava>jar tf MiJar.jar
META-INF/
META-INF/MANIFEST.MF
mx/unam/fi/poo/MiClase.class
```

Y finalmente para ejecutar el método *main* de la clase principal (en este caso *MiClase*) se utiliza el comando **java -jar** seguido del nombre del archivo **JAR**:

```
D:\pruebasJava>java -jar MiJar.jar
Clase empaquetada
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	103/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Si se está utilizando algún **IDE**, la generación de los archivos **JAR** es más sencilla ya que generalmente se cuenta con un proceso guiado paso a paso (**wizard**).



La utilización de archivos **JAR** no solo facilita la distribución de clases, sino también su uso por parte de otras aplicaciones, ya que no es necesario descomprimir el archivo para que las clases contenidas en el puedan ser utilizadas por otras clases.

Una forma de lograr esto puede ser agregar el archivo **JAR** al **CLASPATH** del sistema, es decir, la ruta completa donde se encuentra el archivo y el nombre del mismo (por ejemplo, *D:\pruebasJava\MiJar.jar*) con lo cual ya se podrán utilizar las clases contenidas en el **JAR** importándolas con su **Fully Qualified Name**.

Otra opción es utilizar el **classpath en línea**, es decir, a la hora de compilar o ejecutar se indica con la opción **java -classpath** el jar que debe incluir.

Por ejemplo:

```
javac -classpath MiJar.jar OtraClase.java
```

```
java -classpath MiJar.jar OtraClase
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	104/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Documentación

La escritura de buena **documentación** de las definiciones de las clases y de las interfaces es un complemento importante para obtener código de buena calidad. La **documentación** le permite al programador comunicar sus intenciones a los lectores humanos en un lenguaje natural de alto nivel, en lugar de forzarlos a leer código de nivel relativamente bajo.

La **documentación** de los elementos **públicos** de una clase o de una interfaz tiene un valor especial, pues los programadores pueden usarla sin tener que conocer los detalles de su implementación.

Java cuenta con una herramienta de documentación llamada **javadoc** que se distribuye como parte del kit de desarrollo (**JDK**). Esta herramienta automatiza la **generación de documentación** de clases en formato **HTML** con un estilo consistente. El **API** de Java ha sido documentado usando esta misma herramienta y se aprecia su valor cuando se usa la biblioteca de clases.

Los comentarios **Javadoc**, están delimitados por `/**` y `*/`. Al igual que con los comentarios tradicionales, el compilador ignora todo el texto entre los delimitadores de los comentarios **Javadoc**.

Los elementos de una clase que se documentarán son:

- La definición de la clase
- Sus campos
- Sus constructores
- Sus métodos

Desde el punto de vista de un usuario, lo más importante de una clase es que tenga **documentación** sobre ella y sobre sus constructores y métodos públicos. Tendemos a no proporcionar comentarios del estilo de **javadoc** para los campos aunque recordamos que forman parte del detalle del nivel de implementación y no es algo que verán los usuarios.

Un **comentario** contendrá una descripción principal seguida por una sección de etiqueta, aunque ambas partes son opcionales.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	105/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

La **descripción principal** de una clase debiera consistir en una descripción del objetivo general de la clase. La descripción principal de un método debiera ser bastante general, sin introducir demasiados detalles sobre su implementación. En realidad, la descripción principal de un método generalmente consiste en una sola oración.

Ejemplo:

```
/**
 * Crea un nuevo pasajero con distintas ubicaciones de
 * salida y de destino.
 */
```

Las ideas esenciales debieran presentarse en la primera sentencia de la descripción principal de una clase, de una interfaz o de un método ya que es lo que se usa a modo de resumen independiente en la parte superior de la documentación generada.

Javadoc también soporta el uso de etiquetas HTML en sus comentarios.

A continuación de la descripción principal aparece la sección de **etiquetas**. (**Javadoc** reconoce alrededor de 20 etiquetas). Las etiquetas pueden usarse de dos maneras: en bloques de etiquetas o como etiquetas de una sola línea. Los bloques de etiquetas son los que se usan con mayor frecuencia. Para ver más detalles sobre las etiquetas de una sola línea y sobre las restantes etiquetas, puede recurrir a la sección **javadoc** de la documentación Tools and Utilities que forma parte del JDK.

(<http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html>)

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	106/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Las etiquetas más usadas son:

Etiqueta	Texto asociado
@author	nombre(s) del autor(es)
@param	nombre de parámetro y descripción
@return	descripción del valor de retorno
@see	referencia cruzada
@throws	tipo de excepción que se lanza y las circunstancias en las que se hace
@version	descripción de la versión

Las etiquetas *@author* y *@version* se encuentran regularmente en los comentarios de una clase y de una interfaz y no pueden usarse en los comentarios de métodos, constructores o campos. Ambas etiquetas pueden estar seguidas de cualquier texto y no se requiere ningún formato especial para ninguna de ellas. Ejemplos:

```
@author Hakcer T. LargeBrain
@version 2004.12.31
```

Las etiquetas *@param* y *@throws* se usan en métodos y en constructores, mientras que *@return* se usa sólo en métodos. Algunos ejemplos:

```
@param limite El valor máximo permitido.
@return Un número aleatorio en el rango 1 a limite (inclusive)
@throws IllegalLimitException Si el límite es menor que 1.
```

La etiqueta *@see* adopta varias formas diferentes y puede usarse en cualquier comentario de documentación. Proporciona un camino de referencia cruzada hacia un comentario de otra clase, método o cualquier otra forma de documentación. Se agrega una sección *See Also* al elemento que está siendo comentado.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	107/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Algunos ejemplos típicos:

@see "The Java Language Specification, by Joy et al"

@see The BlueJ web site

@see #estaVivo

@see java.util.ArrayList#add

La primera simplemente encierra un texto en forma de cadena sin un hipervínculo, la segunda es un hipervínculo hacia el documento especificado, la tercera es un vínculo a la documentación del método *estaVivo* de la misma clase, la cuarta vincula la documentación del método *add* con la clase *java.util.ArrayList*.

Para ejecutar **javadoc**, debe introducirse este comando

javadoc -d outputDirectory sourceFiles

La opción **-d outputDirectory** hace que la salida vaya a otro directorio. Si se omite la opción **-d**, por defecto la salida se dirige al directorio actual, pero no es una buena idea, ya que **javadoc** crea muchos archivos que pueden enredar el directorio actual. Es posible colocar documentación para más de una clase en el mismo directorio. Se usan espacios para separar múltiples nombres de archivos fuente con espacios.

Ejemplo:

Para generar la documentación de la clase *MiClase*, se debe editar su código fuente y agregarle los comentarios correspondientes. Una vez hecho esto se procede a generar la documentación con **javadoc**.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	108/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

package mx.unam.fi.poo;

/**
 * Esta es una clase de prueba
 * @author Angelica Nakayama
 * @version Julio-2016
 */
public class MiClase {

    /**
     * Metodo principal o main.
     * Imprime un mensaje indicando que la clase fue empaquetada
     * @param args Los argumentos de linea de comandos
     */
    public static void main(String[] args) {
        System.out.println("Clase empaquetada");
    }
}

```

```

D:\pruebasJava>javadoc -d documentacion MiClase.java
Loading source file MiClase.java...
Constructing Javadoc information...
Creating destination directory: "documentacion\"
Standard Doclet version 1.8.0_45
Building tree for all the packages and classes...
Generating documentacion\mx\unam\fi\poo\MiClase.html...
Generating documentacion\mx\unam\fi\poo\package-frame.html...
Generating documentacion\mx\unam\fi\poo\package-summary.html...
Generating documentacion\mx\unam\fi\poo\package-tree.html...
Generating documentacion\constant-values.html...
Building index for all the packages and classes...
Generating documentacion\overview-tree.html...
Generating documentacion\index-all.html...
Generating documentacion\deprecated-list.html...
Building index for all classes...
Generating documentacion\allclasses-frame.html...
Generating documentacion\allclasses-noframe.html...
Generating documentacion\index.html...
Generating documentacion\help-doc.html...

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

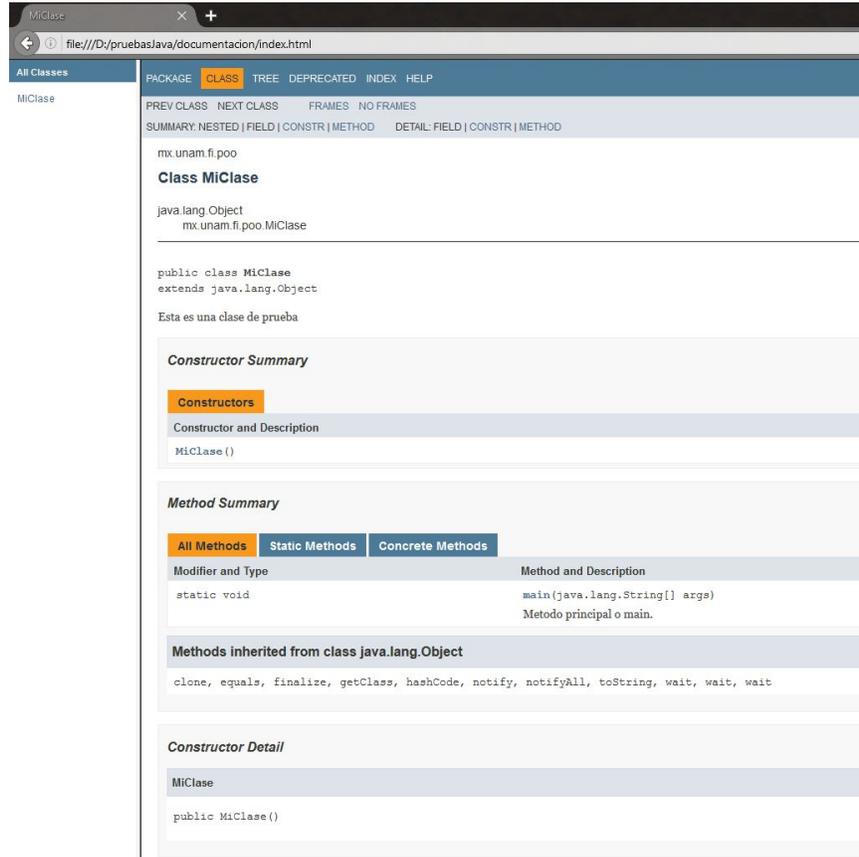
Directorio de D:\pruebasJava

20/07/2016  11:23 a. m.    <DIR>          .
20/07/2016  11:23 a. m.    <DIR>          ..
20/07/2016  11:22 a. m.    <DIR>          documentacion
15/07/2016  06:32 p. m.      104  MANIFEST.MF
20/07/2016  11:21 a. m.      409  MiClase.java
15/07/2016  06:32 p. m.      824  MiJar.jar
15/07/2016  06:26 p. m.    <DIR>          mx
                3 archivos      1,337 bytes
                4 dirs  710,416,855,040 bytes libres

```

La documentación generada estará en el directorio *documentacion* y se puede revisar abriendo el archivo *index.html* con cualquier navegador.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	109/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			



MiClase
 file:///D:/pruebasJava/documentacion/index.html
 All Classes
 PACKAGE CLASS TREE DEPRECATED INDEX HELP
 PREV CLASS NEXT CLASS FRAMES NO FRAMES
 SUMMARY NESTED FIELD CONSTR METHOD DETAIL FIELD CONSTR METHOD
 mx.unam.fi.poo
Class MiClase
 java lang Object
 mx.unam.fi.poo.MiClase

 public class MiClase
 extends java.lang.Object
 Esta es una clase de prueba

Constructor Summary
Constructors
 Constructor and Description
 MiClase ()

Method Summary
All Methods **Static Methods** **Concrete Methods**

Modifier and Type	Method and Description
static void	main(java.lang.String[] args) Metodo principal o main.

Methods inherited from class java.lang.Object
 clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail
MiClase
 public MiClase ()

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	110/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	111/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 07: Herencia



Elaborado por:

M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:

M.C. Alejandro Velázquez Mena

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	112/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 07: Herencia

Objetivo:

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

Actividades:

- Crear clases que implementen herencia.
- Generar 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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	113/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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 (el cuál se abordará en la siguiente práctica).

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**.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	114/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ejemplo:

Dada la siguiente jerarquía de clases:



Figura 1. Jerarquía de clases de Empleado y Gerente.

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	115/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

```

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;
    }
}

```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	116/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.**

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	117/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

La **relación HAS-A (tiene un)** es un concepto que se vio en la práctica de Abstracción y encapsulamiento 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. Este 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 por las referencias de los objetos para verificar si un objeto es de un tipo (clase) específico.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	118/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.");
        }
    }
}

```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	119/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Sobrescritura (overriding)

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	120/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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;
    }
}

```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	121/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

```

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.

Constructores en la herencia

Como se mencionó en la práctica de Clases y objetos, 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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	122/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Cuando se crea un objeto de una clase derivada se crea, implícitamente, un objeto de la clase base que se inicializa con su constructor correspondiente.

Si en la creación del objeto se usa el constructor sin argumentos (*constructor no-args*), entonces 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.

Así mismo, dentro de una clase derivada se puede acceder a los elementos de la clase base a través de la palabra reservada **super**.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	123/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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;
    }
}

```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	124/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

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, el constructor de Ejecutivo invoca directamente al constructor de Empleado mediante la palabra reservada super. La llamada al constructor de la superclase debe ser **la primera sentencia** del 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);
    }
}

```

Debido a que el método *toString* de la clase *Empleado* muestra los atributos deseados, se invoca explícitamente y se agrega el atributo presupuesto. La clase *PruebaEmpleado* ahora sí mostrará toda la información del gerente.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	125/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Sobrecarga (overloading) vs Sobrescritura (overriding)

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, se pueden definir varios métodos con el **mismo nombre**, pero **con diferentes tipos y número de parámetros** en ella.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	126/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	127/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 08: Polimorfismo



Elaborado por:

M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:

M.C. Alejandro Velázquez Mena

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	128/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 08: Polimorfismo

Objetivo:

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

Actividades:

- A partir de una jerarquía de clases, implementar referencias que se comporten como diferentes objetos.

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.

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 recibir utilizar todo tipo de

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	129/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

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**. Como se vio en la práctica de herencia, 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.

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 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 referencia, 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**).

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	130/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Dada la siguiente jerarquía de clases:

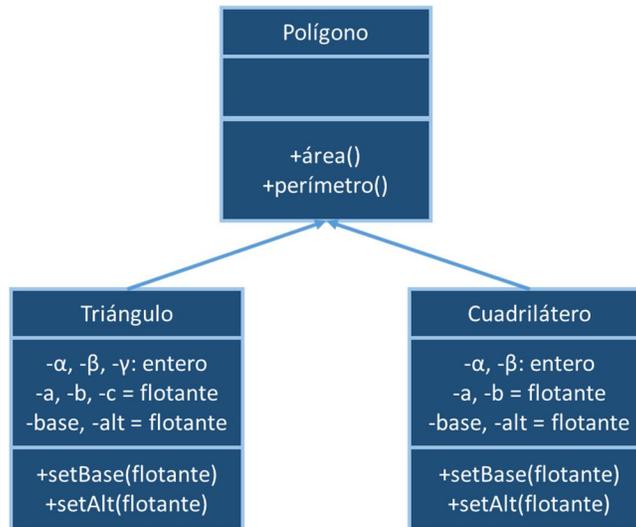


Figura 1. Jerarquía de clases

Ejemplo:

```

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

    public double perimetro(){
        return 0d;
    }

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	131/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

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

    public void setBase(){
    }

    public void setAltura(){
    }

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

public class Triangulo extends Poligono {
    private int alfa, beta, gama;
    private float a, b, c;
    private float base, altura;

    public void setBase(){
    }

    public void setAltura(){
    }

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

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();
        System.out.println(poligono);
        poligono = new Cuadrilatero();
        System.out.println(poligono);

        // Solo se pueden ejecutar los métodos que están definidos
        // en la referencia, sin embargo, se ejecutarán como están
        // implementados en la instancia.
        // El método toString se puede ejecutar porque está definido
        // en Polígono, sin embargo, se va a ejecutar como está
        // implementado en la instancia (Cuadrilátero en este caso).
        System.out.println(poligono);
        // El método setBase no está definido en Polígono, por lo tanto
        // la siguiente instrucción marcaría un error:
        // poligono.setBase(5.5);
    }
}

```

Así mismo, un método puede recibir cualquier tipo de dato como parámetro. 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. A esto se le conoce también como **polimorfismo**, pero en métodos. Como ya se mencionó en la práctica anterior, la palabra reservada *instanceof* permite identificar el tipo de objeto enviado, es decir, no revisa la referencia si no el objeto en sí.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	132/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Código (Métodos polimórficos)

```

public class MetodoPolimorfico {
    public static void main (String [] arg){
        Poligono poligono = new Poligono();
        getPoligono(poligono);
        poligono = new Triangulo();
        getPoligono(poligono);
        poligono = new Cuadrilatero();
        getPoligono(poligono);
    }

    public static void getPoligono(Poligono p){
        if (p instanceof Triangulo){
            System.out.println("El objeto es un triángulo.");
        } else {
            if (p instanceof Cuadrilatero){
                System.out.println("El objeto es un cuadrilátero.");
            } else {
                System.out.println("El objeto es un polígono.");
            }
        }
    }
}

```

Clases abstractas

Una clase **abstracta** es una clase de la que no se pueden crear objetos, debido a que define la existencia de métodos, pero no su implementación.

Las clases **abstractas** sirven como modelo para la creación de clases derivadas. Algunas características de éstas son:

- Pueden contener métodos abstractos y métodos concretos.
- Pueden contener atributos.
- Pueden heredar de otras clases.

Para declarar una clase **abstracta** en Java solo es necesario anteponer la palabra reservada **abstract** antes de palabra reservada *class*, es decir:

```

public abstract class Poligono {
    // Métodos abstractos o concretos
}

```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	133/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Una clase **abstracta** puede tener métodos declarados **abstractos**, en cuyo caso no se da definición del método (no se implementa). **Si una clase tiene algún método abstracto es obligatorio que la clase sea abstracta.** La sintaxis para declarar un método abstracto es:

```
public abstract double perimetro( );
```

La clase que hereda de una clase **abstracta** debe implementar los métodos abstractos definidos en la clase base:

```
public class Triangulo extends Poligono {
    public double perimetro() {
        // bloque de código para obtener el perímetro
    }
}
```

Es posible crear referencias de una clase abstracta:

```
Poligono figura;
```

Sin embargo, una clase **abstracta** **no se puede instanciar**, 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 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();
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	134/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ejemplo:

<pre>public abstract class Poligono { public abstract double area(); public abstract double perimetro(); public String toString(){ return "Polígono"; } }</pre>	<pre>public class Triangulo extends Poligono { private int alfa, beta, gama; private float a, b, c; private float base, altura; // La clase Triangulo está obligada a sobrescribir los métodos // abstractos que definió la clase abstracta de la que hereda. public double area(){ return (base*altura)/2; } public double perimetro(){ return a*b*c; } public String toString(){ return "Triángulo"; } }</pre>
<pre>public class Cuadrilatero extends Poligono { private int alfa, beta; private float a, b; private float base, altura; // La clase Cuadrilatero está obligada a sobrescribir los métodos // abstractos que definió la clase abstracta de la que hereda. public double area(){ return base*altura; } public double perimetro(){ return 2*a*b; } public String toString(){ return "Cuadrilátero"; } }</pre>	<pre>public class PruebaFigurasGeometricas{ public static void main (String [] args){ // No se pueden crear objetos de clases abstractas //Poligono poligono = new Poligono(); Poligono poligono; // Una referencia de una clase abstracta sí // puede almacenar un objeto de una clase concreta poligono = new Triangulo(); System.out.println(poligono); poligono = new Cuadrilatero(); System.out.println(poligono); } }</pre>

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	135/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Interfaces

El concepto de **Interfaces** lleva un paso más adelante la idea de las clases abstractas. Una **interfaz** es una clase abstracta pura, es decir, una clase donde **todos los métodos son abstractos** (no se implementa ninguno). Una interfaz es un contrato sobre *qué* puede hacer la clase, sin decir *cómo* lo va a hacer. Debido a que es una clase 100% abstracta, **no es posible crear instancias de una interfaz.**

Este tipo de diseños permiten establecer la forma de una clase (nombres de métodos, listas de argumentos y tipos de retorno), pero no bloques de código.

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

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

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

Las **interfaces** pueden ser **implementadas** por cualquier clase. **La clase que implementa una interfaz está obligada a definir (implementar) los métodos que la interfaz declaró y,** en ese sentido, adquieren una conducta o modo de funcionamiento particular (contrato).

La sintaxis para implementar una interfaz es la siguiente:

```
public class MiClase implements NombreInterfaz
```

El compilador se encargará de verificar que la clase efectivamente declare e implemente todos los métodos de la interfaz.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	136/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ejemplo:

```

public interface InstrumentoMusical {

    // Por defecto, todos los métodos definidos dentro de
    // una interfaz son públicos (public) y abstractos (abstract)

    void tocar();

    void afinar();

    String tipoInstrumento();
}

public class InstrumentoViento extends Object implements InstrumentoMusical {

    // Por defecto, todos los métodos de la interfaz son públicos,
    // por lo tanto, así se deben implementar

    public void tocar() {
        System.out.println("Tocando instrumento de viento.");
    }

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

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

public class Flauta extends InstrumentoViento {

    // La clase Flauta puede sobrescribir algún método

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

```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	137/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

public class PruebaInstrumento {

    public static void main (String [] args){

        //Se puede crear una referencia de una interfaz.
        InstrumentoMusical instrumento;

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

        // Una referencia a interfaz puede ser asignada cualquier
        // objeto que la implemente.

        instrumento = new Flauta();
        instrumento.tocar();
        System.out.println(instrumento.tipoInstrumento());
    }
}

```

Implementación múltiple

Una clase puede **implementar** cualquier cantidad de **interfaces**, la única restricción es que, dentro del cuerpo de la clase, se deben **implementar todos los métodos** de las interfaces que se implementen. La sintaxis es la siguiente la siguiente:

```

public class NombreClase implements Interfaz1, Interfaz2, ..., InterfazN {
    // Implementar métodos de la Interfaz1
    // Implementar métodos de la interfaz 2
    // ...
    // Implementar métodos de la interfaz N
}

```

El orden de la implementación de las interfaces y el orden en el que se implementan los métodos de las interfaces dentro de la clase no es importante.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	138/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

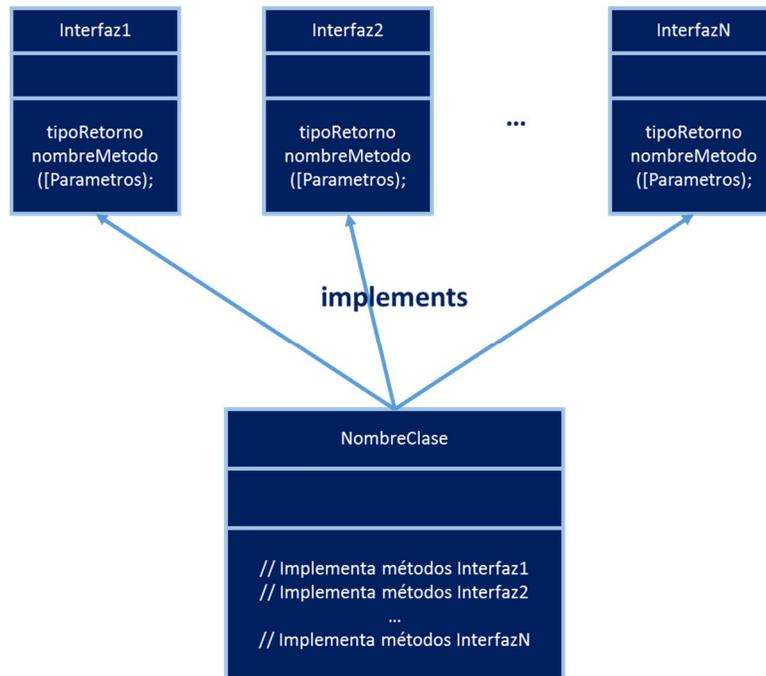


Figura 2. Implementación múltiple de interfaces en una clase.

Herencia múltiple entre interfaces

Las interfaces pueden heredar de otras interfaces y, a diferencia de las clases, **una interfaz puede heredar de una o más interfaces** (herencia múltiple), utilizando la siguiente sintaxis:

```
public interface HeredaInterfaz extends Interfaz1, Interfaz2, ..., InterfazN {
    tipoRetorno nombreMetodo([Parametros]);
}
```

En este caso, se aplican las mismas reglas que en la herencia, es decir, la interfaz que hereda de otras interfaces posee todos los métodos definidos en ellas. Por otro lado, la clase que implemente esta interfaz (la interfaz que hereda de otras interfaces) debe definir los métodos de todas las interfaces.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	139/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

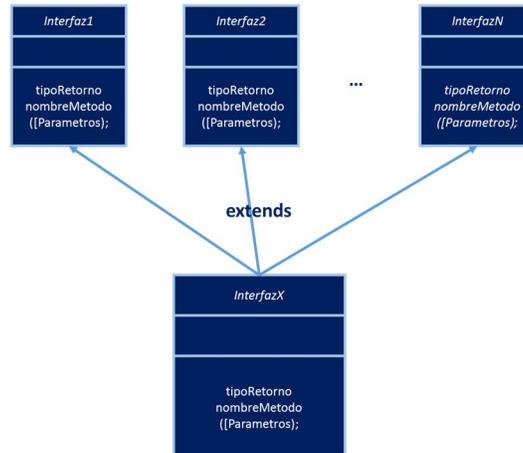


Figura 3. Herencia múltiple entre interfaces.

Atributos en las interfaces

Dado que, por definición, todos los datos miembros (atributos) que se definen en una interfaz son públicos, estáticos y finales y dado que las interfaces no pueden instanciarse, resultan una buena herramienta para implantar grupos de constantes que pueden ser llamados sin crear objetos.

Ejemplo:

```
public interface Meses {
    int UNO = 1, DOS = 2, TRES = 3, CUATRO = 4, CINCO = 5, SEIS = 6;
    int SIETE = 7, OCHO = 8, NUEVE = 9, DIEZ = 10, ONCE = 11, DOCE = 12;
    String [] NOMBRES_MESES = {"", "Enero", "Febrero", "Marzo",
        "Abril", "Mayo", "Junio", "Agosto", "Septiembre", "Octubre",
        "Noviembre", "Diciembre"};
}

public class PruebaMeses {
    public static void main (String [] args){
        // Se puede acceder a las variables de la interfaz sin crear instancias
        System.out.println("El mes " + Meses.DOS + " corresponde a:");
        System.out.println(Meses.NOMBRES_MESES[Meses.DOS]);
    }
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	140/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	141/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 09: UML



Elaborado por:

M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:

M.C. Alejandro Velázquez Mena

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	142/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 09: UML

Objetivo:

Utilizar UML como herramienta para diseñar soluciones de software para un lenguaje de programación orientado a objetos.

Actividades:

- Elegir el o los diagrama(s) necesarios para mostrar la solución de un problema.
- Crear diagramas UML para mostrar la solución de un problema.

Introducción

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.

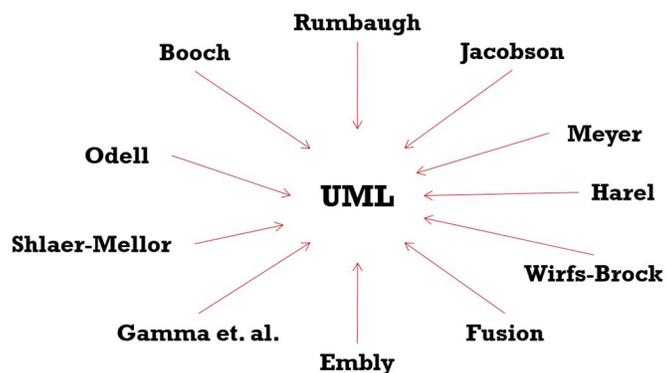


Figura 1. Representación de los lenguajes que componen a UML.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	143/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

Diseño estático o de estructura

Los **diagramas estáticos o estructurales** aportan una visión fija del sistema, los diagramas que permiten modelar estas características para un lenguaje orientado a objetos son:

- Diagrama de casos de uso
- Diagrama de clases
- Diagrama de objetos

Diseño dinámico o de comportamiento

Los **diagramas dinámicos o de comportamiento** permiten visualizar la comunicación entre elementos del sistema para un proceso específico, los diagramas que permiten modelar estas características para un lenguaje orientado a objetos son:

- Diagrama de estados
- Diagrama de actividades
- Diagrama de interacción

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	144/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Diagrama de casos de uso

Un **diagrama de caso de uso** define la manera en la que el usuario del sistema interactúa con éste. Los diagramas de casos de uso permiten modelar el comportamiento de un sistema, un subsistema o, incluso, una clase. Está compuesto por 3 elementos básicos:

- Actor(es).
- Caso(s) de uso.
- Relación(es) (uso, herencia y comunicación).

Por lo tanto, para generar un diagrama de casos de uso hay que identificar a los **actores** (o roles) que van a interactuar con el sistema, las **actividades** dentro del sistema, así como los **permisos** (relaciones) que tendrá cada actor con los casos de uso.

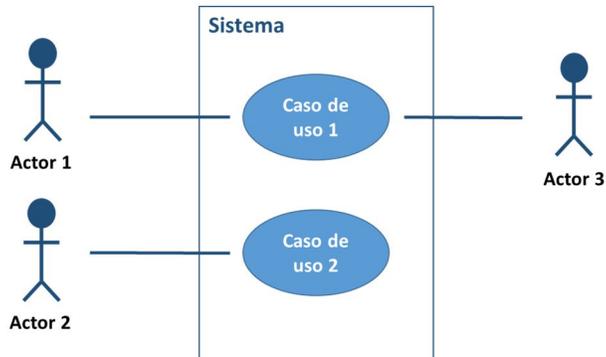


Figura 2. Diagrama de casos de uso.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	145/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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.

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.

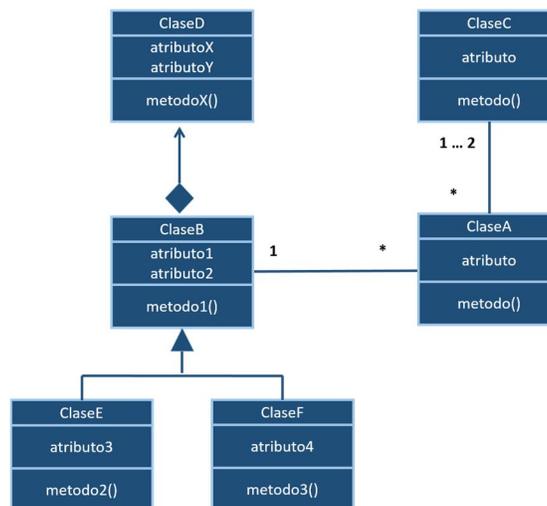


Figura 3. Diagrama de clases.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	146/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Diagrama de objetos

Los **diagramas de objetos** modelan las instancias generadas a través de las clases y se utilizan para describir al sistema en un instante de tiempo (o acción) en particular.

Permiten mostrar los objetos y las relaciones entre ellos en un momento dado, por lo tanto, representa la parte estática de la interacción entre objetos (una situación específica en un momento determinado).

Los elementos utilizados para generar este tipo de diagramas son:

- Objeto(s)
- Asociación(es)

Los diagramas de objetos permiten conocer los **valores** (estado) que pueden tener los objetos en un instante de tiempo, así como su **relación** con otros objetos.

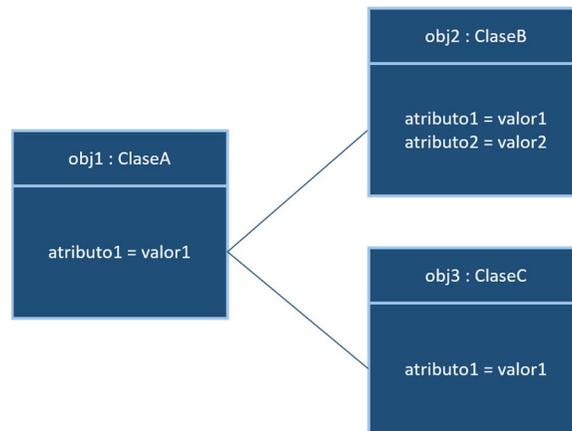


Figura 4. Diagrama de objetos.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	147/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Diagrama de estados

Los **diagramas de estados** describen las transiciones por las que puede pasar un objeto durante su tiempo de vida en la aplicación, así como la manera en la que cambia de estado el objeto.

Los elementos de un diagrama de estados son:

- Nodos de entrada y salida.
- Estado(s) del objeto.
- Transición(es) entre estados.

Un diagrama de estados permite modelar el **comportamiento** de un objeto de manera automática, viendo todas las posibles **transiciones** que puede generar.

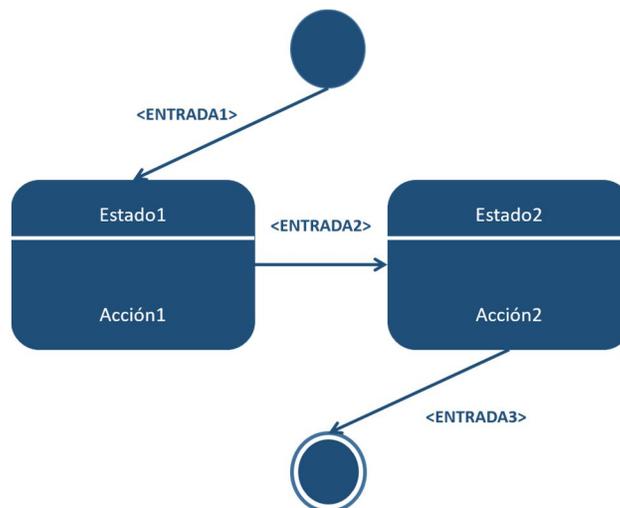


Figura 5. Diagrama de estados.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	148/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Diagrama de actividades

El **diagrama de actividades** muestra el flujo de acciones (operaciones que se ejecutan) y los objetos involucrados. Permite visualizar el orden en el que se van realizando tareas dentro de un sistema, así como los objetos involucrados en la comunicación.

Los componentes básicos de un diagrama de actividades son:

- Nodos de entrada y salida.
- Actividad(es).
- Transiciones.

Un diagrama de actividades permite ver la **comunicación** que tienen de manera interna las **actividades** que puede realizar un objeto ya sea con métodos propios del objeto (secuencial) o con otros objetos.

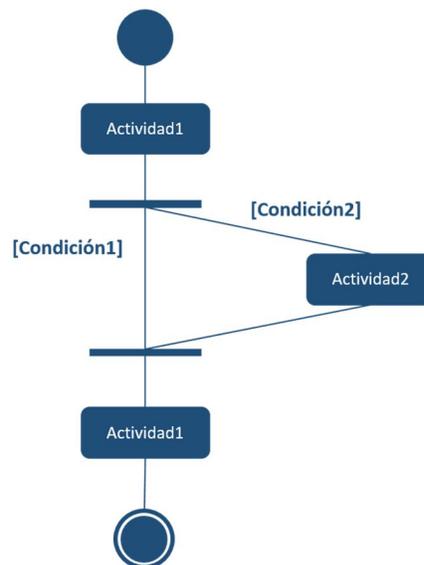


Figura 6. Diagrama de actividades.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	149/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Diagrama de interacción

Estos diagramas representan la comunicación que se lleva a cabo entre un cliente (actor) o un objeto (clase) cuando se ejecuta una acción en el sistema.

Los elementos básicos de los diagramas de interacción son:

- Objeto o actor.
- Enlace(s).
- Mensaje(s).

Los diagramas de interacción, a su vez, se dividen en dos tipos:

- Diagrama de secuencia.
- Diagrama de colaboración.

Diagrama de secuencia

Los **diagramas de secuencia** muestran una interacción ordenada de eventos, visualizando los objetos participantes en cada interacción, así como los mensajes que intercambian entre ellos.

Los diagramas de secuencia permiten ver la **interacción** que se genera al ejecutar una **acción**, así como los **objetos involucrados** en el proceso.

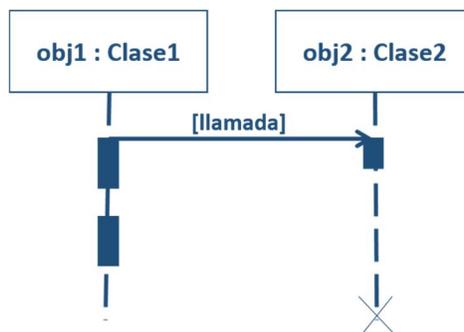


Figura 7. Diagrama de secuencia.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	150/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Diagrama de comunicación

Los **diagramas de comunicación** muestran la interacción entre varios objetos y el orden que existen entre ellos. La secuencia de los mensajes y los flujos de ejecución concurrentes se determinan mediante números secuenciales.

Un diagrama de comunicación permite visualizar la **interacción** que se presenta en un **proceso** en particular, permite identificar los objetos involucrados y el orden de invocación.

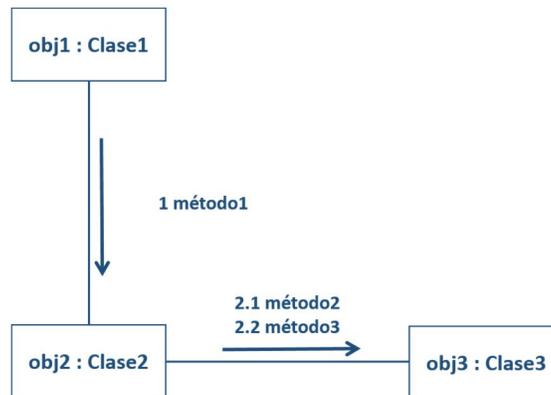


Figura 8. Diagrama de comunicación.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	151/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Bibliografía

MILES, Russ, HAMILTON, Kim

Learning UML 2.0

Boston

O Reilly Media, 2006

GOMAA, Hassan

Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures

Washington

Cambridge University Press, 2011

Schmuller, Joseph

Aprendiendo UML En 24 Horas

México

Prentice-Hall, 2000

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	152/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 10: Excepciones y errores



Elaborado por:

M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:

M.C. Alejandro Velázquez Mena

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	153/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 10: Excepciones y errores

Objetivo:

Identificar bloques de código propensos a generar errores y aplicar técnicas adecuadas para el manejo de situaciones excepcionales en tiempo de ejecución.

Actividades:

- Capturar excepciones de un bloque de código seleccionado.
- Capturar errores de un bloque de código seleccionado.

Introducción

La ley de Pareto, también conocida como la regla 80/20, establece que el 80 % de las consecuencias proviene del 20 % de las causas.

Una máxima en el desarrollo de software dicta que el 80 % del esfuerzo (en tiempo y recursos) produce el 20 % del código. Así mismo, en términos de calidad, el 80 % de las **fallas** de una aplicación son producidas por el 20 % del código. Por lo tanto, detectar y manejar errores es la parte más importante en una aplicación robusta.

Una aplicación puede tener diversos tipos de **errores**, los cuales se pueden clasificar en tres grandes grupos:

- **Errores sintácticos:** Son todos aquellos errores que se generan por **infringir las normas de escritura de un lenguaje**: coma, punto y coma, dos puntos, palabras reservadas mal escritas, etc. Normalmente son detectados por el compilador o el intérprete (según el lenguaje de programación utilizado) al procesar el código fuente.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	154/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

- **Errores semánticos (o lógicos):** Son errores más sutiles, se producen cuando la sintaxis del código es correcta, pero la **semántica o significado** no es el que se pretendía. Los compiladores e intérpretes sólo se ocupan de la estructura del código que se escribe y no de su significado, por lo tanto, un error semántico puede hacer que el programa termine de forma anormal, con o sin un mensaje de error.

No todos los errores semánticos se manifiestan de una forma obvia. Un programa puede continuar en ejecución después de haberse producido errores semánticos, pero su estado interno puede ser distinto del esperado. Quizá las variables no contengan los datos correctos, o bien es posible que el programa siga un camino distinto del pretendido. Eventualmente, la consecuencia será un resultado incorrecto. Estos errores se denominan lógicos, ya que, aunque el programa no se bloquea, la lógica que representan contiene un error.

- **Errores de ejecución:** Son errores que se presentan cuando la aplicación se está ejecutando. Su origen puede ser diverso, se pueden producir por un **uso incorrecto del programa** por parte del usuario (si ingresa una cadena cuando se espera un número), o se pueden presentar debido a **errores de programación** (acceder a localidades no reservadas o hacer divisiones entre cero), o debido a algún **recurso externo** al programa (al acceder a un archivo o al conectarse a algún servidor o cuando se acaba el espacio en la pila (stack) de la memoria).

Un **error en tiempo de ejecución** provoca que la aplicación termine abruptamente. Los lenguajes orientados a objetos proveen mecanismos para **manejar errores de ejecución**.

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	155/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Excepciones

El término **excepción** hace referencia a una condición excepcional que cuando ocurre **altera el flujo normal** del programa en ejecución. Estos errores pueden ser generados por la lógica del programa, como un índice de un arreglo fuera de su rango, una división entre cero, etc., o errores generados por los propios objetos que denuncian algún tipo de estado no previsto o condición que no pueden manejar.

Desde el punto de vista del tratamiento de una excepción dentro de un programa, hay que tener en cuenta que todas estas clases de excepción se dividen en dos grandes grupos:

- **Excepciones marcadas**
 - Aquellas cuya captura es obligatoria.
- **Excepciones no marcadas**
 - Las excepciones en tiempo de ejecución (RuntimeException y sus subclases). No es obligatorio capturarlas.

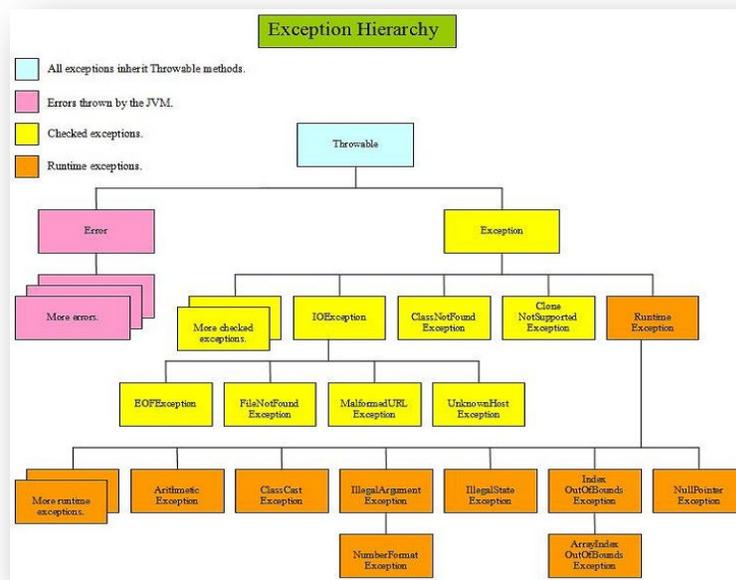


Figura 1. Jerarquía de clases de las Excepciones y Errores.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	156/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

En Java, cuando un evento excepcional ocurre se dice que se **lanza una excepción**. Las **excepciones** son el mecanismo mediante el cual se pueden controlar los errores producidos en tiempo de ejecución. El código responsable de hacer alguna acción cuando se arroja una excepción es llamado manejador de excepciones y lo que hace es **cachar la excepción lanzada**.

Manejo de Excepciones

Para **manejar una excepción** se utilizan las palabras reservadas **try** y **catch**. El bloque *try* es utilizado para definir el bloque de código en el cual una excepción pueda ocurrir. El o los bloques *catch* son utilizados para definir un bloque de código que maneje la excepción.

Las excepciones son **objetos** que contienen información del error que se ha producido. Heredan de la clase *Exception* que a su vez hereda de la clase *Throwable*.

Ejemplo:

```
public class TryCatchFinally {
    public static void main(String[] args) {
        try {
            String mensajes[] = {"Primero", "Segundo", "Tercero" };
            for (int i=0; i<=3; i++)
                System.out.println(mensajes[i]);
        } catch ( ArrayIndexOutOfBoundsException e ) {
            System.out.println("Error: apuntador fuera del rango del arreglo.");
        }
    }
}
```

Si no se captura la excepción interviene un manejador por defecto que normalmente imprime información que ayuda a encontrar dónde y cómo se produjo el error, sin embargo, como la excepción no es controlada, ésta se propaga hasta llegar al método principal y termina abruptamente el programa.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	157/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ejemplo:

```
public class TryCatchFinally {
    public static void main(String[] args) {
        String mensajes[] = {"Primero", "Segundo", "Tercero" };
        for (int i=0; i<=3; i++)
            System.out.println(mensajes[i]);
    }
}
```

La palabra reservada **finally** permite definir un tercer bloque de código dentro del manejador de excepciones. Este bloque le indica al programa las instrucciones a ejecutar de manera independiente de los bloques *try-catch*, es decir, si el código del bloque *try* se ejecuta de manera correcta, entra al bloque *finally*; si se genera un error, después de ejecutar el código del bloque *catch* ejecuta el código del bloque *finally*.

Ejemplo:

```
public class TryCatchFinally {
    public static void main(String[] args) {
        try {
            float equis = 5/0;
            System.out.println("Equis = " + equis);
        } catch ( ArithmeticException e ) {
            System.out.println("Error: división entre cero.");
        } finally {
            System.out.println("A pesar de todo, se ejecuta el bloque finally.");
        }
    }
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	158/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Sin generar excepción:

```
public class TryCatchFinally {
    public static void main(String[] args) {
        try {
            float equis = 5/2;
            System.out.println("Equis = " + equis);
        } catch ( ArithmeticException e ) {
            System.out.println("Error: división entre cero.");
        } finally {
            System.out.println("A pesar de todo, se ejecuta el bloque finally.");
        }
    }
}
```

Cómo ya se mencionó, el bloque *catch* permite capturar excepciones, es decir, manejar los errores que genere el código del bloque *try* en tiempo de ejecución impidiendo así que el programa deje de ejecutarse y posibilitando al desarrollador enviar el error generado.

Es posible tener **más de un bloque *catch*** dentro del manejador de excepciones, pero cuidando el orden de captura, es decir, las excepciones se deben acomodar de las más específicas a las más generales, como se muestra a continuación:

```
catch (ClassNotFoundException e) {...}
catch (IOException e) {...}
catch (Exception e) {...}
```

Si se invierte el orden, se capturan las excepciones de lo más general a lo más específico, todas las excepciones caerían en la primera (*Exception* es la más general) y no habría manera de enviar un error del tipo *IOException* o *ClassNotFoundException*. Esta situación la detecta el compilador y no permite generar el *bytecode*, es decir, no compila.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	159/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ejemplo:

```
public class TryCatchFinally {
    public static void main(String[] args) {
        try {
            int equis = 5/0;
            System.out.println("Equis = " + equis);
        } catch ( ArithmeticException e ) {
            System.out.println("Error: división entre cero.");
        } catch ( Exception e ) {
            System.out.println("Error: excepción general.");
        } finally {
            System.out.println("El bloque finally siempre se ejecuta.");
        }
    }
}
```

Propagación de excepciones

No es obligatorio tratar las excepciones dentro de un bloque manejador de excepciones, pero, en tal caso, se debe indicar explícitamente a través del método. A su vez, el método superior deberá incluir los bloques *try/catch* o volver a **pasar la excepción**. De esta forma se puede ir propagando la excepción de un método a otro hasta llegar al último método del programa, el método *main*. Ejemplo:

```
public class PropagaExcepcion {
    public static int miMetodo(int a, int b){
        int c =a / b;
        return c;
    }
    public static void main(String[] args) {
        try{
            int division= miMetodo(10, 0);
            System.out.println(division);
        } catch(ArithmeticException e){
            System.out.println("Excepcion aritmetica arrojada: " );
            e.printStackTrace();
        }
    }
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	160/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Throws

Existen métodos que obligan a ejecutarse dentro de un manejador de excepciones (es decir, son marcadas), debido a que el método define que **va a lanzar** una excepción.

Para indicar que un método **puede lanzar una excepción** se utiliza la palabra reservada **throws** seguida de la o las excepciones que puede arrojar dicho método. La sintaxis es la siguiente:

```
[modificadores] valorRetorno nombreMetodo() throws Excepcion1, Excepcion2 {
    // Bloque de código del método
}
```

Esta sintaxis obliga que al utilizar el método se deba realizar dentro de un manejador de excepciones o dentro de un método que indique que va a arrojar la misma excepción.

```
public class PropagaExcepcion {
    public static void main(String[] args) {
        try{
            int division = miMetodo(10, 0);
            System.out.println(division);
        } catch (ArithmeticException e) {
            System.out.println("Excepcion aritmetica arrojada");
        }
    }
    public static int miMetodo(int a, int b) throws ArithmeticException{
        int c = a / b;
        return c;
    }
}
```

Throw

Una **excepción se puede lanzar** de manera explícita (sin necesidad de que ocurra un error) creando una instancia de la misma y arrojándola mediante la palabra reservada **throw**. Una excepción se debe arrojar dentro de un manejador de excepciones o dentro de un método que indique que se va arrojar dicha excepción (o cualquier subclase). La sintaxis es la siguiente:

```
throw new Excepcion();
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	161/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ejemplo:

```
public class PropagaExcepcion {
    public static int miMetodo(int a, int b) throws ArithmeticException{
        if(b == 0){
            throw new ArithmeticException();
        }
        int c = a / b;
        return c;
    }
    public static void main(String[] args) {
        try{
            int division= miMetodo(10, 0);
            System.out.println(division);
        } catch(ArithmeticException e){
            System.out.println("Excepcion aritmetica arrojada: " );
            e.printStackTrace();
        }
    }
}
```

Excepciones propias

Cuando se desarrollan aplicaciones es común requerir excepciones que no están definidas en el API de Java, es decir, **excepciones propias del negocio**. Es posible crear clases que se comporten como excepciones (que se puedan arrojar), para ello sólo es necesario heredar de *Exception* o de *Throwable*. Estas excepciones se invocan y se pueden arrojar de la misma manera en la que se utilizan las excepciones del API, pero pueden generar información más concisa del problema debido a que se están programando a la medida del negocio.

Ejemplo:

Para una aplicación bancaria se desea lanzar una excepción propia cuando el saldo de una cuenta sea insuficiente para realizar una operación (por ejemplo un retiro).

Se crea la clase de excepción propia *SaldoInsuficienteException*:

```
public class SaldoInsuficienteException extends Exception {
    public SaldoInsuficienteException() {
        super("Saldo insuficiente");
    }
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	162/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Se crea la clase *Cuenta*, la cual podrá lanzar una excepción de tipo *SaldoInsuficienteException* si se intenta retirar un monto mayor al saldo de la cuenta:

```
public class Cuenta {
    private double saldo;

    public Cuenta(){
        saldo=0;
    }

    public void depositar(double monto){
        System.out.println("Depositando " + monto);
        saldo += monto;
    }

    public void retirar(double monto) throws SaldoInsuficienteException{
        System.out.println("Retirando " + monto);
        if(saldo < monto)
            throw new SaldoInsuficienteException();
        else
            saldo -= monto;
    }

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

Finalmente, para probar el funcionamiento de la clase *Cuenta* y su correspondiente excepción, se crea una cuenta *Cajero* donde se emulan depósitos y retiros a una cuenta:

```
public class Cajero {
    public static void main(String[] args) {
        Cuenta cuenta = new Cuenta();
        try {
            cuenta.depositar(2000);
            cuenta.retirar(1000);
            cuenta.getSaldo();
            cuenta.retirar(1000);
            cuenta.getSaldo();
            cuenta.retirar(1000);
            cuenta.getSaldo();
            cuenta.depositar(200);
            cuenta.retirar(100);
        } catch (SaldoInsuficienteException e) {
            e.printStackTrace();
        }
    }
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	163/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Los métodos **getMessage** y **printStackTrace** se heredan de *Exception*. El método *getMessage* imprime la cadena que se envía al objeto cuando se construye. El método *printStackTrace* imprime la traza del error, es decir, el método donde se generó el problema y todos los métodos que se invocaron ante de éste.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	164/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	165/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 11: Manejo de archivos



Elaborado por:

M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:

M.C. Alejandro Velázquez Mena

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	166/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 11: Manejo de archivos

Objetivo:

Implementar el intercambio de datos (lectura y escritura) entre fuentes externas (archivos y/o entrada y salida estándar) y un programa (en un lenguaje orientado a objetos).

Actividades:

- Crear archivos de texto plano.
- Leer archivos de texto plano.
- Escribir en archivos de texto plano.

Introducción

Los programas necesitan comunicarse con su entorno, tanto para obtener datos e información que deben procesar, como para devolver los resultados obtenidos. El manejo de archivos se realiza a través de **streams** o **flujos de datos** desde una fuente hacia un repositorio. La fuente inicia el flujo de datos, por lo tanto, se conoce como flujo de datos de entrada. El repositorio termina el flujo de datos, por lo tanto, se conoce como flujo de datos de salida. Es decir, tanto la fuente como el repositorio son nodos de flujos de datos.



	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	167/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Archivos

Un **archivo** es un objeto en una computadora que puede almacenar información, configuraciones o comandos, el cual puede ser manipulado como una entidad por el sistema operativo o por cualquier programa o aplicación. Un archivo debe tener un nombre único dentro de la carpeta que lo contiene. Normalmente, el nombre de un archivo contiene un sufijo (extensión) que permite identificar el tipo del archivo.

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.

Flujos de datos

Las entradas y las salidas de datos en java se manejan mediante **streams** (flujos de datos). Un **stream** es una conexión entre el programa y la fuente (lectura) o el destino (escritura) de los datos. La información se traslada en serie a través de esta conexión.

En Java existen 4 jerarquías de clases relacionadas con los flujos de entrada y salida de datos.

- **Flujos de bytes:** las clases derivadas de *InputStream* (para lectura) y de *OutputStream* (para escritura), las cuales manejan los flujos de datos como stream de bytes.
- **Flujos de caracteres:** las clases derivadas de *Reader* (para lectura) y *Writer* (para escritura), las cuales manejan stream de caracteres.

Todas las clases de Java relacionadas con la entrada y salida se agrupan en el paquete *java.io*.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	168/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Clase File

La clase **File** permite manejar archivos o carpetas, es decir, crear y borrar tanto archivos como carpetas, entre otras funciones.

Cuando se crea una instancia de la clase *File* no se crea ningún archivo o directorio, solo se crea una referencia hacia un objeto de este tipo. La creación de archivos o carpetas se realizan de manera explícita, invocando a los métodos respectivos. A continuación se enlistan los métodos más útiles que posee la clase File:

- *exists()*
- *createNewFile()*
- *mkdir()*
- *delete()*
- *renameTo()*
- *list()*

Ejemplo:

```
import java.io. File;
import java.io. IOException;

class Escribe {

    public static void main(String [] args) {
        try {
            File archivo = new File("archivo.txt");
            System.out.println(archivo.exists());
            boolean seCreo = archivo.createNewFile();
            System.out.println(seCreo);
            System.out.println(archivo.exists());
        } catch(IOException e) { }
    }
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	169/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

FileOutputStream

La clase **FileOutputStream** permite crear y escribir un flujo de bytes en un archivo de texto plano. Esta clase hereda de la clase *OutputStream*. Sus constructores más comunes son:

- *FileOutputStream (String nombre)*
- *FileOutputStream (String nombre, boolean añadir)*
- *FileOutputStream (File archivo)*

FileInputStream

FileInputStream permite leer flujos de bytes desde un archivo de texto plano. Hereda de la clase *InputStream*. Sus constructores más comunes son:

- *FileInputStream(String nombre)*
- *FileInputStream(File archivo)*

Ejemplo:

Escritura usando *FileOutputStream*

```
import java.io.FileOutputStream;
import java.io.IOException;

public class ClaseFileOutputStream {

    public static void main (String [] args){
        FileOutputStream fos = null;
        byte[] buffer = new byte[81];
        int nBytes;
        try {
            System.out.println("Escribir el texto a guardar en el archivo:");
            nBytes = System.in.read(buffer);
            fos = new FileOutputStream("fos.txt");
            fos.write(buffer,0,nBytes);
        } catch (IOException ioe){
            System.out.println("Error: " + ioe.toString());
        } finally {
            try {
                if (fos != null) fos.close();
            } catch (IOException ioe) {
                System.out.println("Error al cerrar el archivo.");
            }
        }
    }
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	170/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Lectura usando *FileInputStream*

```
import java.io.FileInputStream;
import java.io.IOException;

public class ClaseFileInputStream {
    public static void main (String [] args){
        FileInputStream fis = null;
        byte[] buffer = new byte[81];
        int nbytes;
        try {
            fis = new FileInputStream("leer.txt");
            nbytes = fis.read(buffer, 0, 81);
            String texto = new String(buffer, 0, nbytes);
            System.out.println(texto);
        } catch (IOException ioe) {
            System.out.println("Error: " + ioe.toString());
        } finally {
            try {
                if (fis != null) fis.close();
            } catch (IOException ioe) {
                System.out.println("Error al cerrar el archivo.");
            }
        }
    }
}
```

FileWriter

La clase **FileWriter** hereda de *Writer* y permite escribir un flujo de caracteres en un archivo de texto plano.

BufferedWriter

La clase **BufferedWriter** también deriva de la clase *Writer* y permite crear un buffer para realizar una escritura eficiente de caracteres desde la aplicación hacia el archivo destino.

PrintWriter

La clase **PrintWriter**, que también deriva de *Writer*, permite escribir de forma sencilla en un archivo de texto plano. Posee los métodos *print* y *println*, idénticos a los de *System.out*, y el método *close()*, el cual cierra el stream de datos.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	171/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

FileReader

Las clases **Reader** se utilizan para obtener los caracteres ingresados desde una fuente. La clase **FileReader** hereda de *Reader* y permite leer flujos de caracteres de un archivo de texto plano.

InputStreamReader

InputStreamReader es una clase que deriva de *Reader* que convierte los streams de bytes a streams de caracteres. *System.in* es el objeto de la clase *InputStream* el cual recibe datos desde la entrada estándar del sistema (el teclado).

BufferedReader

La clase **BufferedReader**, que también deriva de la clase *Reader*, crea un buffer para realizar una lectura eficiente de caracteres. Dispone del método *readLine* que permite leer una línea de texto y tiene como valor de retorno un *String*.

Ejemplo:

Lectura usando *FileReader*

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ClaseFileReader{
    public static void main (String [] escribir){
        String texto = "";
        try {
            BufferedReader br;
            FileReader fr = new FileReader("leer.txt");
            br = new BufferedReader(fr);
            System.out.println("El texto contenido en el archivo leer.txt es:");
            String linea = br.readLine();
            while (linea != null ) {
                System.out.println(linea);
                linea = br.readLine();
            }
            br.close();
        } catch (IOException ioe){
            System.out.println("\n\nError al abrir o guardar el archivo:");
            ioe.printStackTrace();
        } catch (Exception e){
            System.out.println("\n\nError al leer de teclado:");
            e.printStackTrace();
        }
    }
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	172/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Escritura usando *FileWriter*

```

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.io.IOException;

public class ClaseFileWriter{
    public static void main (String [] leer){
        String texto = "";
        try{
            BufferedReader br;
            br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Escribir texto:");
            texto = br.readLine();
            FileWriter fw = new FileWriter("archivo.txt");
            BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter salida = new PrintWriter(bw);
            salida.println(texto);
            salida.close();
        } catch (IOException ioe){
            System.out.println("\n\nError al abrir o guardar el archivo:");
            ioe.printStackTrace();
        } catch (Exception e){
            System.out.println("\n\nError al leer de teclado:");
            e.printStackTrace();
        }
    }
}

```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	173/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Lectura de teclado

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class LeeTecladoBR {
    public static void main (String [] args){
        try {
            String texto = "";
            BufferedReader br;
            br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Escribir el texto deseado:");
            texto = br.readLine();
            System.out.println("El texto escrito fue: " + texto);
        } catch (IOException ioe){
            System.out.println("Error al leer caracteres: \n" + ioe);
        }
    }
}
```

StringTokenizer

La clase **StringTokenizer** permite separar una cadena de texto por palabras (espacios) o por algún otro carácter. La clase *StringTokenizer* pertenece al paquete *java.util*.

Ejemplo:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

public class LeeTecladoCompleto {
    public static void main (String [] leer){
        String texto = "";
        try{
            BufferedReader br;
            br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Escribir texto:");
            texto = br.readLine();

            System.out.println("\n\nEl texto separado por espacios es:");
            StringTokenizer st = new StringTokenizer(texto);
            while(st.hasMoreTokens()) {
                System.out.println(st.nextToken());
            }
        } catch (Exception e){
            System.out.println("\n\nError al leer de teclado:");
            e.printStackTrace();
        }
    }
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	174/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Scanner

La clase **Scanner** permite leer flujos de bytes desde la entrada estándar, pero también puede hacerlo desde otra fuente. Pertenece al paquete *java.util*.

Los métodos principales de esta clase son *next()* y *hasNext()*. El método *next()* obtiene el siguiente elemento del flujo de datos. El método *hasNext()* verifica si el flujo de datos todavía posee elementos, en caso afirmativo regresa *true*, de lo contrario regresa *false*. El delimitador de la clase *Scanner* (para obtener el siguiente elemento), por defecto, es el espacio en blanco, aunque es posible cambiar el delimitador utilizando el método *useDelimiter* que recibe como parámetro el delimitador (en forma de *String*).

Ejemplo:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class EjScanner {

    public static void main(String [] args) {
        File archivo = new File("archivo.txt");
        Scanner lector;
        try{
            lector = new Scanner(archivo);
            lector.useDelimiter("/");
            while(lector.hasNext()){
                System.out.println(lector.next());
            }
            lector.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	175/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Console

La clase **Console** permite recibir flujos de datos desde la **línea de comandos** (entrada estándar). Se encuentra dentro del paquete `java.io`. Entre los métodos importantes que posee se encuentran:

- **readLine()**: lee una cadena de caracteres hasta que encuentra el salto de línea (enter).
- **readPassword()**: lee una cadena de caracteres hasta que encuentra el salto de línea (enter), ocultando los caracteres como lo hace el sistema operativo que se utilice.

Ejemplo:

```
import java.io.Console;

public class EjConsole {

    public static void main(String [] args){
        Console con = System.console();
        System.out.print("Usuario: ");
        String s = con.readLine();
        System.out.println(s);
        System.out.print("Contraseña: ");
        char [] s2 = con.readPassword();
        System.out.println(s2);
    }
}
```

Serialización

La **serialización** es un mecanismo para guardar los objetos como una secuencia de bytes y poderlos reconstruir en un futuro cuando se necesiten, conservando su estado. Cuando un objeto se **serializa** solo los campos del objeto (atributos) son preservados.

Si un campo hace referencia a un objeto, esta referencia debe ser también **serializable**. Los objetos que no son **serializables** se deben declarar con la palabra reservada **transient** para evitar que se intenten serializar y se genere un error.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	176/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ejemplo:

Serializar un objeto Date

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.Date;

public class SerializeDate {

    SerializeDate() {
        Date d = new Date ();
        System.out.println(d);
        try {
            FileOutputStream f = new FileOutputStream ("date.ser");
            ObjectOutputStream s = new ObjectOutputStream (f);
            s.writeObject (d);
            s.close ();
        } catch (IOException e) {
            e.printStackTrace ();
        }
    }

    public static void main (String args[]) {
        new SerializeDate();
    }
}
```

Deserializar objeto Date

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Date;

public class DeSerializeDate {

    DeSerializeDate () {
        Date d = null;
        try {
            FileInputStream f = new FileInputStream ("date.ser");
            ObjectInputStream s = new ObjectInputStream (f);
            d = (Date) s.readObject ();
            s.close ();
        } catch (Exception e) {
            e.printStackTrace ();
        }
        System.out.println( "Deserialized Date object from date.ser");
        System.out.println("Date: "+d);
    }

    public static void main (String args[]) {
        new DeSerializeDate();
    }
}
```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	177/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Bibliografía

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

Deitel Paul, Deitel Harvey.
Como programar en Java
 Séptima 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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	178/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 12: Hilos



Elaborado por:

M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:

M.C. Alejandro Velázquez Mena

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	179/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 12: Hilos

Objetivo:

Implementar el concepto de multitarea utilizando **hilos** en un lenguaje orientado a objetos.

Objetivo:

- Implementar hilos utilizando la clase Thread.
- Implementar hilos utilizando la interfaz Runnable.

Introducción

Imaginemos una aplicación departamental que deba realizar varias operaciones complejas. Sus funciones son descargar el catálogo de precios de los productos nuevos, realizar la contabilidad del día anterior y aplicar descuentos a productos existentes.

En un flujo normal las tareas se realizarían de forma secuencial, es decir, las tareas se ejecutarán una después de la otra. Sin embargo, si la descarga de productos nuevos tarda demasiado, los descuentos no se podrían aplicar hasta que este proceso termine y, si se requiere un producto con descuento, éste no se podrá aplicar.

Lo ideal sería tener **varios flujos de ejecución** para poder realizar una tarea sin necesidad de esperar a las otras. Esto se puede lograr utilizando **hilos**, donde cada hilo representaría una tarea.

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.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	180/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Hilos (threads)

Un **hilo** es un único flujo de ejecución dentro de un proceso. Un proceso es un programa en ejecución dentro de su propio espacio de direcciones.

Por lo tanto, un **hilo** es una secuencia de código en ejecución dentro del contexto de un proceso, esto debido a que los **hilos** no pueden ejecutarse solos, requieren la supervisión de un proceso.

Hilos en Java

La Máquina Virtual Java (**JVM**) es capaz de manejar **multihilos**, es decir, puede crear varios flujos de ejecución de manera simultánea, administrando los detalles como asignación de tiempos de ejecución o prioridades de los **hilos**, de forma similar a como lo administra un Sistema Operativo múltiples procesos.

Java proporciona soporte para **hilos** a través de una interfaz y un conjunto de clases. La interfaz de Java y las clases que proporcionan algunas funcionalidades sobre hilos son:

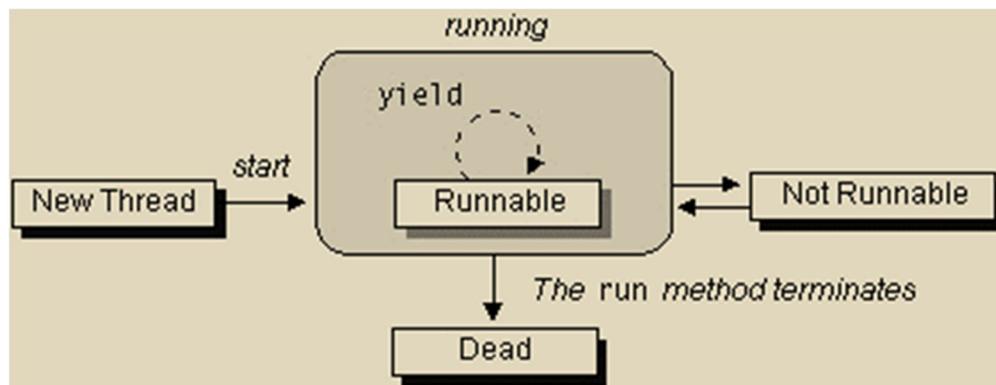
- *Thread*
- *Runnable* (interfaz)
- *ThreadDeath*
- *ThreadGroup*
- *Object*

Tanto las clases como la interfaz son parte del paquete básico de java (*java.lang*).

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	181/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ciclo de vida de un hilo

El campo de acción de un **hilo** lo compone la etapa **runnable**, es decir, cuando se está **ejecutando** (corriendo) el proceso ligero.



Estado **new**

Un **hilo** está en el estado **new** (nuevo) la primera vez que se crea y hasta que el método **start** es llamado. Los hilos en estado **new** ya han sido inicializados y están listos para empezar a trabajar, pero aún no han sido notificados para que empiecen a realizar su trabajo.

Estado **runnable**

Cuando se llama al método **start** de un **hilo** nuevo, el método **run** es invocado, en ese momento el **hilo** entra en el estado **runnable** y por tanto, el hilo se encuentra en ejecución.

Estado **not running**

Cuando un **hilo** está **detenido** se dice que está en estado **not running**. Los **hilos** pueden pasar al estado **not running** por los métodos **suspend**, **sleep** y **wait** o por algún bloqueo de I/O.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	182/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Dependiendo de la manera en que el **hilo** pasó al estado **not running** es como se puede regresar al estado **runnable**:

- Cuando un hilo está **suspendido**, se invoca al método **resume**.
- Cuando un hilo está **durmiendo**, se mantendrá así el número de milisegundos especificado.
- Cuando un hilo está en **espera**, se activará cuando se haga una llamada a los métodos **notify** o **notifyAll**.
- Cuando un hilo está **bloqueado** por I/O, regresará al estado **runnable** cuando la operación I/O sea completada.

Estado **dead**

Un **hilo** entra en estado **dead** cuando ya no es un objeto necesario. Los **hilos** en estado **dead** no pueden ser resucitados, es decir, ejecutados de nuevo. Un **hilo** puede entrar en estado **dead** por dos causas:

- El método **run** termina su ejecución.
- Se realiza una llamada al método **stop**.

Planificador

Java tiene un **Planificador** (*Scheduler*) que controla todos los **hilos** que se están ejecutando en todos sus programas y decide cuáles deben ejecutarse (dependiendo de su prioridad) y cuáles deben encontrarse preparados para su ejecución.

Prioridad

Un **hilo** tiene una **prioridad** (un valor entero entre 1 y 10) de modo que cuanto mayor es el valor, mayor es la prioridad.

El **planificador** determina qué **hilo** debe ejecutarse en función de la **prioridad** asignada a cada uno de ellos. Cuando se crea un **hilo** en Java, éste hereda la prioridad de su padre. Una vez creado el **hilo** es posible modificar su **prioridad** en cualquier momento utilizando el método **setPriority**.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	183/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

El planificador ejecuta primero los **hilos** de **prioridad superior** y sólo cuando éstos terminan, puede ejecutar **hilos** de **prioridad inferior**. Si varios hilos tienen la misma prioridad, el planificador elige entre ellos de manera alternativa (forma de competición). Cuando pasa a ser “Ejecutable”, entonces el sistema elige a este nuevo hilo.

Clase Thread

Es la clase en Java responsable de producir **hilos** funcionales para otras clases. Para añadir la funcionalidad de **hilo** a una clase solo se hereda de ésta.

La clase **Thread** posee el método *run*, el cual define la acción de un hilo y, por lo tanto, se conoce como el cuerpo del hilo. La clase *Thread* también define los métodos *start* y *stop*, los cuales permiten iniciar y detener la ejecución del hilo.

Por lo tanto, para añadir la funcionalidad deseada a cada hilo creado es necesario redefinir el método *run*. Este método es invocado cuando se inicia el hilo. Un hilo se inicia mediante una llamada al método *start* de la clase *Thread*. El hilo se inicia con la llamada al método *run* y finaliza cuando termina este método llega a su fin.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	184/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ejemplo:

Clase *Hilo* que hereda de *Thread*

```
public class Hilo extends Thread {
    public Hilo (String nombre) {
        super(nombre);
    }
    public void run() {
        for(int i = 0 ; i < 5 ; i++) {
            System.out.println("IteraciOn " + (i+1) + " de " + getName());
        }
        System.out.println("Termina el " + getName());
    }
    public static void main(String[] args) {
        new Hilo ("Primer hilo").start();
        new Hilo ("Segundo Hilo").start();
        System.out.println("Termina el hilo principal");
    }
}
```

Interfaz Runnable

La interfaz **Runnable** permite producir hilos funcionales para otras clases. Para añadir la funcionalidad de hilo a una clase por medio de *Runnable*, solo es necesario implementar la interfaz.

La interfaz *Runnable* proporciona un método alternativo al uso de la clase *Thread*, para los casos en los que no es posible hacer que la clase definida herede de la clase *Thread*.

Las clases que implementan la interfaz *Runnable* proporcionan un método *run* que es ejecutado por un objeto *Thread* creado. Ésta es una herramienta muy útil y, a menudo, es la única salida que se tiene para incorporar hilos dentro de las clases.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	185/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ejemplo:

Clase *Hilo* que implementa *Runnable*

```
public class Hilo implements Runnable {
    public void run() {
        for(int i = 0 ; i < 5 ; i++) {
            System.out.println("IteraciOn " + (i+1) + " de " +
                Thread.currentThread().getName());
        }
        System.out.println("Termina el " +
            Thread.currentThread().getName());
    }
    public static void main(String[] args) {
        new Thread(new Hilo (), "Primer hilo").start();
        new Thread(new Hilo (), "Segundo Hilo").start();
        System.out.println("Termina el hilo principal");
    }
}
```

Clase *ThreadDeath*

ThreadDeath deriva de la clase *Error*, la cual proporciona medios para manejar y notificar errores.

Cuando el método *stop* de un hilo es invocado, una instancia de *ThreadDeath* es lanzada por el hilo como un error. Así, cuando se detiene al hilo de esta forma, se detiene de forma asíncrona. El hilo morirá cuando reciba realmente la excepción *ThreadDeath*.

Sólo se debe recoger el objeto *ThreadDeath* si se necesita para realiza una limpieza específica para la ejecución asíncrona, lo cual es una situación bastante inusual. Si se recoge el objeto, debe ser relanzado para que el hilo en realidad muera.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	186/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Clase ThreadGroup

ThreadGroup se utiliza para manejar un grupo de hilos de manera simplificada (en conjunto). Esta clase proporciona una manera de controlar de modo eficiente la ejecución de una serie de hilos.

ThreadGroup proporciona métodos como *stop*, *suspend* y *resume* para controlar la ejecución del grupo (todos los hilos del grupo).

Los hilos de un grupo pueden, a su vez, contener otros grupos de hilos permitiendo una jerarquía anidada de hilos. Los hilos individuales tienen acceso al grupo, pero no al padre del grupo.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	187/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ejemplo:
Grupo de hilos

```

public class EjThreadGroup extends Thread {

    public EjThreadGroup(ThreadGroup g, String n){
        super(g,n);
    }

    public void run(){
        for (int i = 0 ; i < 10 ; i++){
            System.out.print(getName());
        }
    }

    public static void listarHilos(ThreadGroup grupoActual) {
        int numHilos;
        Thread [] listaDeHilos;

        numHilos = grupoActual.activeCount();
        listaDeHilos = new Thread[numHilos];
        grupoActual.enumerate(listaDeHilos);
        System.out.println("\nNumero de hilos activos = " + numHilos + "\n");
        for (int i = 0 ; i < numHilos ; i++) {
            System.out.println("\nHilo activo " + (i+1) + " = "
                + listaDeHilos[i].getName());
        }
    }

    public static void main(String[] args) {
        ThreadGroup grupoHilos = new ThreadGroup("Grupo con prioridad normal");
        Thread hilo1 = new EjThreadGroup(grupoHilos, "Hilo 1 con prioridad maxima");
        Thread hilo2 = new EjThreadGroup(grupoHilos, "Hilo 2 con prioridad normal");
        Thread hilo3 = new EjThreadGroup(grupoHilos, "Hilo 3 con prioridad normal");
        Thread hilo4 = new EjThreadGroup(grupoHilos, "Hilo 4 con prioridad normal");
        Thread hilo5 = new EjThreadGroup(grupoHilos, "Hilo 5 con prioridad normal");
        hilo1.setPriority(Thread.MAX_PRIORITY);
        grupoHilos.setMaxPriority(Thread.NORM_PRIORITY);
        System.out.println("Prioridad del grupo = " +
            grupoHilos.getMaxPriority());

        System.out.println("Prioridad del Thread = " + hilo1.getPriority());
        System.out.println("Prioridad del Thread = " + hilo2.getPriority());
        System.out.println("Prioridad del Thread = " + hilo3.getPriority());
        System.out.println("Prioridad del Thread = " + hilo4.getPriority());
        System.out.println("Prioridad del Thread = " + hilo5.getPriority());

        hilo1.start();
        hilo2.start();
        hilo3.start();
        hilo4.start();
        hilo5.start();

        ListarHilos(grupoHilos);
    }
}

```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	188/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Métodos o bloques sincronizados

Los métodos **sincronizados** (*synchronizable*) son aquellos a los que es imposible acceder si un objeto está haciendo uso de ellos, es decir, dos objetos no pueden acceder a un método **sincronizado** al mismo tiempo.

Los métodos **sincronizados** son muy utilizados en hilos (*threads*) para evitar la pérdida de información o ambigüedades en la misma. Los hilos se ejecutan como **procesos paralelos** (independientes). Cuando se ejecutan varios hilos de manera simultánea, éstos pueden intentar acceder al mismo tiempo a un método, para, por ejemplo, obtener o modificar el valor de un atributo. Como los hilos se ejecutan en paralelo, el acceso al recurso no es controlado y, por lo tanto, puede haber pérdida de información.

Estas complicaciones se pueden evitar **bloqueando** el acceso al método mientras algún proceso lo esté ejecutando. Lo anterior se puede lograr haciendo que el método en cuestión sea **sincronizado** (*synchronizable*). Así, al estar sincronizados los métodos de una clase, si un recurso está ocupando un método, éste es inaccesible hasta que sea liberado.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	189/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Ejemplo:

```

public class Cuenta extends Thread {
    private static long saldo = 0;

    public Cuenta (String nombre){
        super(nombre);
    }

    public void run() {
        if (getName().equals("Deposito 1") ||
            getName().equals("Deposito 2")) {
            this.depositarDinero(100);
        } else {
            this.extraerDinero(50);
        }
        System.out.println("Termina el " + getName());
    }

    public synchronized void depositarDinero(int cantidad) {
        saldo += cantidad;
        System.out.println("Se depositaron " + cantidad + " pesos");
        notifyAll();
    }

    public synchronized void extraerDinero(int cantidad) {
        try {
            if (saldo <= 0){
                System.out.println(getName() + " espera depOsito"
                    + "\nSaldo = " + saldo);
                sleep(5000);
            }
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        saldo -= cantidad;
        System.out.println(getName() + " extrajo " + cantidad +
            " pesos.\nSaldo restante = "+ saldo);
        notifyAll();
    }

    public static void main(String[] args) {
        new Cuenta("Acceso 1").start();
        new Cuenta("Acceso 2").start();
        new Cuenta("Deposito 1").start();
        new Cuenta("Deposito 2").start();
        System.out.println("Termina el hilo principal");
    }
}

```

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	190/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Bibliografía

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

Deitel Paul, Deitel Harvey.
Como programar en Java
 Séptima 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

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	191/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 13: Patrones de diseño



Elaborado por:

M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:

M.C. Alejandro Velázquez Mena

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	192/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Guía práctica de estudio 13: Patrones de diseño

Objetivo:

Implementar una aplicación en un lenguaje orientado a objetos utilizando algún patrón de diseño.

Actividades:

- Conocer diferentes patrones de diseño.
- Implementar una aplicación utilizando algún patrón de diseño.

Introducción

En la ingeniería de software, un patrón de diseño es una solución repetible y general para problemas de ocurrencia cotidianos en el diseño de software. Es importante aclarar que un patrón de diseño no es un diseño de software terminado y listo para codificarse, más bien es una descripción o modelo (template) de cómo resolver un problema que puede utilizarse en diferentes situaciones.

Por lo tanto, los patrones de diseño permiten agilizar el proceso de desarrollo de solución debido a que proveen un paradigma desarrollado y probado.

Un diseño de software efectivo debe considerar detalles que tal vez no sean visibles hasta que se implemente la solución, es decir, debe anticiparse a los problemas y tratar de cubrir todos los resquicios. La reutilización de patrones de diseño ayuda a prevenir los detalles sutiles que provocarían problemas más grandes, además de ayudar a la legibilidad de código para programadores o analistas que estén familiarizados con patrones.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	193/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Patrones de software

Cada patrón de diseño describe un problema que ocurre una y otra vez de manera cotidiana y describe el núcleo de la solución, de tal manera que esa solución se puede utilizar muchas veces.

De manera general, un patrón de diseño posee 4 elementos esenciales:

1. El **nombre del patrón** se utiliza para describir el diseño del problema, su solución y sus consecuencias en una o dos palabras.
2. El **problema** describe cuando se debe aplicar el patrón. Explica el problema y su contexto. Puede describir problemas de diseño específico, como sería la representación de algoritmos como objetos. Puede describir estructuras de clases u objetos que son sintomáticos de un diseño inflexible, para evitar caer en ellos. Algunas veces el problema puede incluir una lista de condiciones que se deben cumplir para que el patrón tenga sentido y sea viable su implementación.
3. La **solución** describe los elementos que forman el diseño, su relación, responsabilidades y colaboraciones. La solución no describe de manera particular un diseño o implementación, más bien provee una descripción abstracta del diseño de un problema y cómo, de manera general, el acomodo de elementos (clases y objetos) resuelve el problema.
4. Las **consecuencias** son el resultado y las recompensas de haber aplicado el patrón. Aunque las consecuencias son a menudo poco mencionadas, cuando se describe el diseño de decisiones se vuelven críticas para evaluar el diseño alternativo y poder entender el costo-beneficio de aplicar el patrón. Las consecuencias en la ingeniería de software a menudo tienen que ver con compensaciones en espacio y tiempo. Debido a que la reutilización es un factor en el diseño orientado a objetos, las consecuencias de un patrón de diseño incluyen su impacto en la flexibilidad del sistema, su extensibilidad y su portabilidad.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	194/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería	Área/Departamento: Laboratorio de computación salas A y B		
La impresión de este documento es una copia no controlada			

Los puntos de vista afectan la interpretación de qué es y qué no es un patrón. Lo que para una persona puede ser un patrón de diseño para otra puede ser un bloque primitivo de construcción.

El diseño de patrones no establece un diseño como una lista ligada o una tabla hash que puede ser codificado en clases y reutilizado una y otra vez. Tampoco son tan complejos como para llegar a especificar un diseño en un dominio específico para una aplicación o sistema.

El diseño de patrones es, más bien, una descripción de la comunicación que existe entre objetos y clases, las cuales se pueden personalizar para resolver un problema de diseño general en un contexto en particular.

Se consideran tres niveles dentro de los patrones de software:

- Patrones arquitectónicos: Los cuales describen soluciones al más alto nivel de software y hardware. Normalmente soportan requerimientos no funcionales (criterios de evaluación).
- Patrones de diseño: Describen soluciones en un nivel medio de estructuras de software. Normalmente soportan requerimientos funcionales (especificaciones técnicas del sistema).
- Patrones de programación: Describen soluciones en el nivel de software más bajo, a nivel de clases y métodos. Normalmente soportan características específicas de un lenguaje de programación.

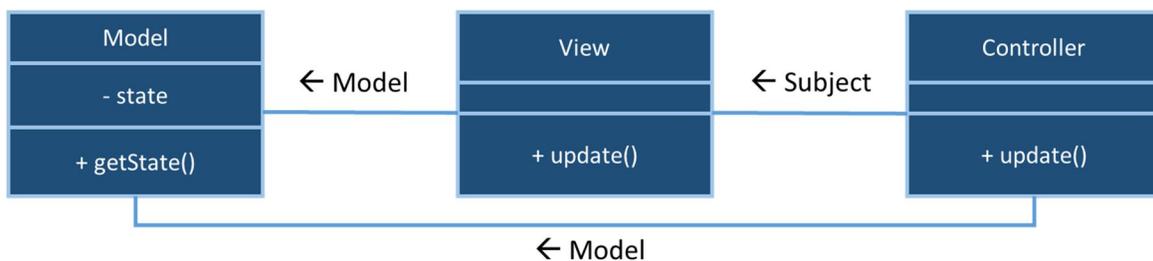
Así mismo, dentro del catálogo de los patrones de diseño existen tres grandes grupos: los creacionales, los estructurales y los de comportamiento.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	195/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

MVC

El patrón de diseño Modelo-Vista-Controlador (MVC) asigna objetos con alguno de los tres roles. El patrón define los roles que juegan los objetos en la aplicación, así como la manera en la que éstos se comunican entre sí. Dentro de una implementación MVC, la colección de objetos del mismo tipo forman lo que se conoce como capa, por ejemplo, la capa del modelo.

La implementación del patrón MVC provee varios beneficios. Los objetos tienden a ser más reutilizados, además, las interfaces creadas suelen estar mejor definidas y, en general, suelen ser aplicaciones fáciles de extender.



Modelo MVC

Modelo

Los objetos que forman parte del Modelo encapsulan la información de la aplicación y definen la lógica con la que se van manipular los datos. Así mismo, un objeto del Modelo puede tener relaciones uno a uno o uno a muchos con otros objetos del Modelo.

Debido a que los objetos del Modelo representan conocimiento y experiencia relacionada con un dominio específico del problema, éstos pueden ser en problemas con un dominio similar.

Idealmente, los objetos del Modelo no deben tener una conexión explícita con los objetos de la Vista, que son los que muestran la información y permiten al usuario modificarla. Las acciones de los usuarios en la capa de Vista son comunicadas al Modelo a través del Controlador. Así, cuando un objeto del Modelo cambia, éste notifica al objeto del Controlador para que actualice los objetos de Vista necesarios.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	196/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Vista

Un objeto de Vista es el objeto de la aplicación que el usuario puede ver. El objeto de Vista sabe cómo mostrarse y puede responder a las acciones del usuario. El propósito máximo de un objeto de Vista es mostrar la información del objeto del Modelo y habilitar la edición de información.

Los objetos de Vista se enteran de los cambios en la información a través de los objetos del Controlador y la comunicación con el usuario. Así mismo, envían la información recibida del usuario a los objetos del Controlador para que éste se los envíe a los objetos del Modelo.

Controlador

Un objeto del Controlador actúa como intermediario entre uno o más objetos de la Vista y uno o más objetos del Modelo. Por lo tanto, los objetos del Controlador son un conducto a través del cual los objetos de la Vista se enteran de los cambios de los objetos del Modelo y viceversa. Los objetos del Controlador también pueden establecer y coordinar tareas de la aplicación y administrar el ciclo de vida de los objetos.

En general, los objetos del Controlador interpretan las acciones realizadas por los usuarios en los objetos de la Vista y comunican estas acciones hacia la capa del Modelo. Así mismo, cuando un objeto del Modelo cambia, el objeto del Controlador envía la información a los objetos de la Vista para que ésta pueda ser mostrada.

Nombre: Modelo-Vista-Controlador (MVC).

Problema:

Se requieren mostrar varias pantallas de manera gráfica (Vista) con la información que se encuentra en un repositorio de datos (Modelo). Debe existir un mediador (Controlador) entre las pantallas y el repositorio de datos para facilitar la comunicación y el paso de información.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	197/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Solución.

Se debe desacoplar la información (Modelo de datos) del código de la interfaz gráfica que construye la vista, permitiendo así construir varias vistas.

Cuando se ejecute la aplicación se puede registrar cada vista con el modelo de información correspondiente a través de un controlador de paso de información (Observador).

Consecuencias

MVC desacopla vistas y modelos estableciendo un protocolo de suscriptor/notificador entre ellos, de tal manera que se debe asegurar que la vista refleje el estado del modelo. Así, si la información del modelo cambia, el modelo debe notificar a la vista que depende de él.

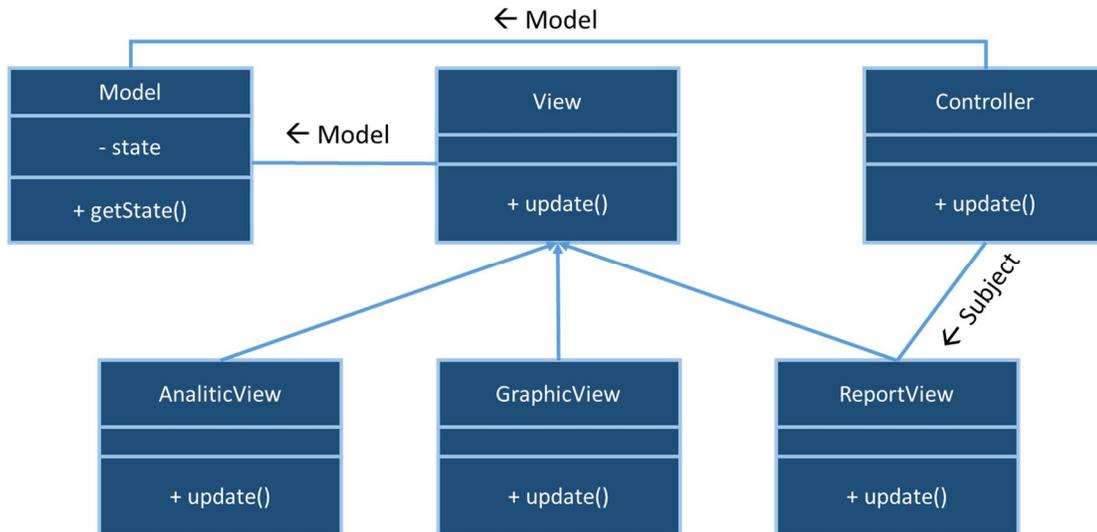
Se pueden acoplar varias vistas a un modelo, para mostrar diferente información. También se pueden crear nuevas vistas sin necesidad de reescribir el modelo.

Permite manejar las entradas del usuario sin necesidad de cambiar la presentación (Vista), ya que la capacidad de responder ante un estímulo de la vista se encapsula en el controlador, el cual puede implementar diversas respuestas dependiendo del estímulo enviado por la vista, sin modificar ésta.

Ejemplo:

Una aplicación debe mostrar información analítica (información numérica) y gráfica (gráfica de barras) a partir de una base de datos. Así mismo, la aplicación puede presentar un reporte de la información mostrada.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	198/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			



Patrón de diseño creacional

Este tipo de diseño abstrae el proceso de instanciación (creación de objetos), permitiendo con esto que el sistema sea independiente de cómo sus objetos son creados, se componen y se representan. Un patrón de diseño creacional de clases utiliza la herencia para variar la clase que crea una instancia, mientras que un patrón de diseño creacional de objetos delegará la instanciación a otro objeto.

Los patrones creacionales se vuelven importantes como sistemas que evolucionan para depender más de la composición de objetos que de la herencia de clases. A medida que esto sucede, el énfasis se desplaza lejos de un código duro y hacia la definición de un conjunto fijo de comportamientos con desempeños fundamentales que pueden ser parte de conjuntos más complejos. Así, la creación de objetos con un comportamiento particular requiere más que simplemente instanciar una clase.

En estos patrones existen dos temas recurrentes. El primero, se encapsula todo el conocimiento sobre una clase en específico que utiliza el sistema. El segundo, se oculta la manera en la que las instancias de las clases del sistema se crean y se unen entre sí. Por ende, los patrones creacionales proporcionan mucha flexibilidad en lo que se crea, quién lo

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	199/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

crea, cómo lo crea y cuando lo hace. Esto permite configurar un sistema objetos producto que pueden variar ampliamente en su estructura y funcionalidad.

Es importante recalcar que algunas veces los patrones relacionales pueden competir entre sí por ser igual de rentables. Algunas otras veces se pueden complementar.

Los patrones creacionales más conocidos son:

- Singleton
- Factory Method
- Abstract Factory
- Builder
- Prototype

Siendo uno de los más utilizados y conocidos el patrón de diseño creacional MVC.

Patrón de diseño creacional Singleton

Nombre: Singleton.

Problema:

En la aplicación debe existir únicamente una instancia de una clase y ésta (la instancia) debe ser accesible para los clientes a través de un punto de acceso bien conocido.

Para algunas aplicaciones es importante tener solamente una instancia en ejecución. Por ejemplo, en un sistema de impresoras se debe tener una sola cola de impresión.

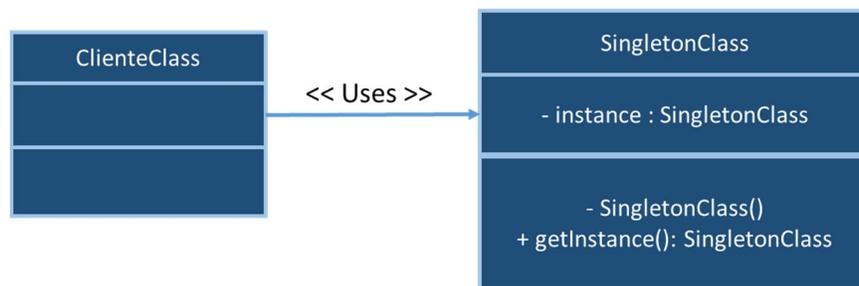
Solución.

La clase debe ser la responsable de hacer el seguimiento para una única instancia. La clase puede asegurar que no se pueda crear otra instancia, así como proveer una ruta para acceder a dicho objeto.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	200/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Por lo tanto, lo que se debe hacer es una clase que:

- a) Mantenga su constructor privado.
- b) Mantenga una instancia como privada y estática.
- c) Provea un método público y estático para acceder a la instancia.



Modelo Singleton

Consecuencias

Los beneficios de utilizar el patrón de diseño Singleton son:

- Controlar el acceso a la única instancia. Debido a que se encapsula la única instancia, se posee un estricto control sobre cómo y cuándo se puede acceder a ella.
- Reducir el espacio de nombres. Evita llenar el espacio de nombres con variables globales, debido a que se almacena una única instancia.
- Refinar las operaciones y su representación. Una clase puede tener subclasses y se puede configurar la aplicación para que se tenga una instancia de las subclasses. Se puede, por tanto, crear una instancia de alguna subclase en tiempo de ejecución.
- Permitir un número variable de instancias. El patrón permite modificar para crear más de una instancia de la clase, así mismo se pueden controlar todas las instancias de manera general, lo único que hay que modificar son las operaciones que permiten acceso a la instancia.
- Es más flexible que las operaciones de clase. Se pueden empaquetar las funcionalidades del patrón mediante métodos de clase.

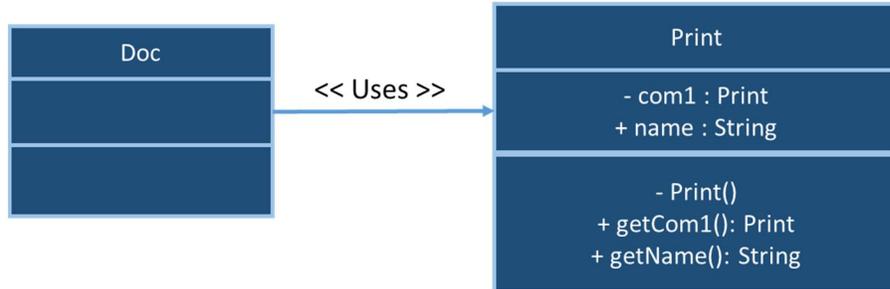
	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	201/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

El patrón de diseño creacional Singleton se puede usar cuando:

- Debe existir exactamente una instancia de una clase y ésta debe ser accesible desde un punto de acceso bien conocido.
- Cuando una única instancia deba existir para una subclase y los clientes deban ser capaces de utilizar una subclase sin modificar su código.

Ejemplo:

Una impresora dentro de una compañía debe manejar todos los documentos que se envíen para mantener un orden y un registro, ese es un buen lugar para proveer una clase que mantenga una sola instancia de la impresora para todos los objetos que la utilicen. Por lo tanto, se puede usar el patrón Singleton para manejar las impresiones.



Patrón de diseño estructural

A los patrones de diseño estructural les interesa como están compuestas clases y objetos para formar estructuras grandes.

El patrón estructural de clases usa la herencia para componer la implementación de múltiples clases base. Esto es, cuando se mezclan varias clases base dentro de una (herencia múltiple), la clase resultante combina las propiedades de todas las clases base, generando con ello una estructura grande. Este patrón es muy útil cuando se quiere hacer que clases desarrolladas de manera independiente y en diferentes bibliotecas trabajen unidas.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	202/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

El patrón estructural de objetos describe la manera en la que se componen los objetos para reaccionar ante una nueva funcionalidad. Los objetos poseen una composición flexible, esto es, tienen la habilidad de cambiar su forma en tiempo de ejecución, lo cual no es posible con una composición estática.

Los patrones estructurales más conocidos son:

- Composite
- Proxy
- Flyweight
- Facade
- Bridge

El patrón estructural de objetos Composite (compuesto) describe como crear una jerarquía de clases a partir de clases con dos tipos de elementos: primitivos y compuestos. Los objetos compuestos permiten crear instancias de ambos elementos dentro de una estructura compleja.

El patrón Proxy actúa como un sustituto de otro objeto, el cual puede ser utilizado de diferentes maneras. Puede actuar como una representación local para un objeto dentro de un espacio remoto. Puede representar un objeto grande que se carga sobre demanda. Protege el acceso a objetos con información sensible.

El patrón Flyweight (peso mosca) define una estructura para compartir objetos. Los objetos se pueden compartir por, al menos, dos razones: eficiencia y consistencia. Flyweight se enfoca en compartir un espacio eficiente, para aquellas aplicaciones que utilizan muchos objetos, ya que se puede ahorrar bastante espacio compartiendo objetos en lugar de replicarlos. La información adicional que necesitan los objetos para realizar sus tareas se les pasa sobre demanda (cuando la necesitan).

El patrón Facade permite crear un solo objeto que representa un subsistema completo. Está compuesto por un conjunto de objetos y, para llevar a cabo sus tareas, envía mensajes a las instancias para comunicarles sus responsabilidades.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	203/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

El patrón Bridge (puente) separa la abstracción de un objeto de su implementación, de tal manera que se puede modificar el comportamiento de manera independiente.

El patrón Decorator (decorador) describe la posibilidad de agregar funcionalidades a los objetos de manera dinámica. Crea los objetos de manera recursiva, de tal forma que permite agregar un número ilimitado de funciones adicionales. Por ejemplo, un objeto Decorador contiene un componente de interfaz de usuario, se puede agregar un decorador al componente como puede ser un borde o un sombreado o, incluso, se puede agregar una funcionalidad como una barra de desplazamiento o un zoom.

Patrón de diseño estructural Composite

Nombre: Composite.

Problema:

Se requiere notificar a un conjunto variado de objetos que un evento ha ocurrido en la aplicación.

Una modificación en un objeto debe desencadenar una serie de cambios en otros objetos, sin embargo, no se tiene la certeza del número de objetos que este cambio afectará.

Solución.

Crear una clase abstracta *Subject* la cual contenga un conjunto de observadores, de tal manera que cuando un cambio ocurra en *Subject*, el observador le informará a todos los objetos de su conjunto de dicho cambio.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	204/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

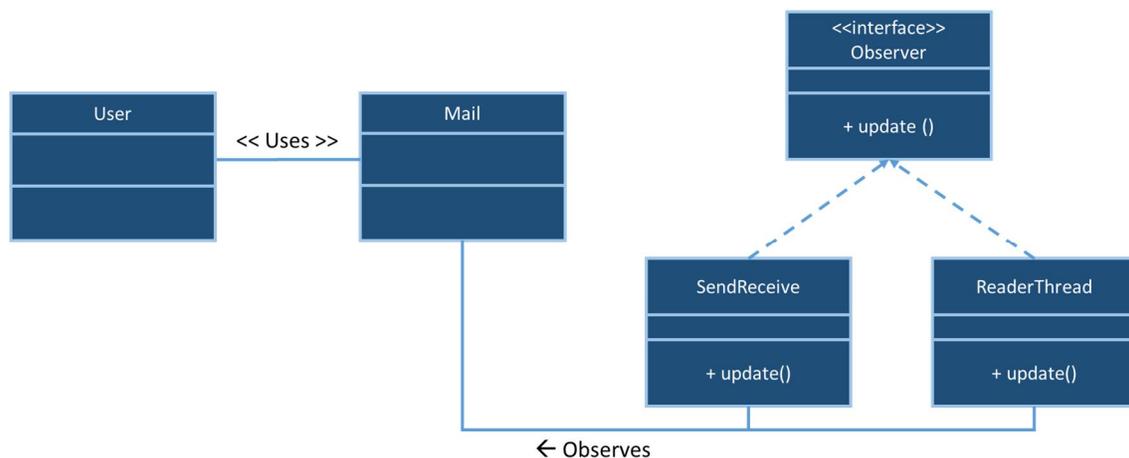
Consecuencias

Los beneficios de utilizar el patrón de diseño Composite son:

- Definir una jerarquía de clases que consiste en objetos primitivos y compuestos, donde los objetos primitivos pueden componer objetos más complejos, que a su vez pueden componer objetos más robustos y así de manera recursiva.
- Hacer la tarea del cliente simple, ya que puede tratar con estructuras compuestas u objetos de manera uniforme.
- Hacer fácil la creación de nuevas funcionalidades, debido a que los cambios se ven reflejados en las clases compuestas, el cliente no se ve obligado a agregar código y sí puede utilizar las nuevas funcionalidades.
- La creación de nuevas funcionalidades de manera sencilla provoca que sea difícil restringir los componentes nuevos, por la misma facilidad para agregar.

Ejemplo:

Una aplicación de correo debe estar actualizando constantemente su bandeja en busca de nuevo correo. Además, el usuario puede cerciorarse de manera manual de la recepción de nuevo correo. En ambos casos, la bandeja de entrada se debe actualizar si es que existe nuevo correo. Un observador se encarga de actualizar la información en la bandeja de entrada cuando es necesario en ambos casos.



	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	205/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Patrón de diseño de comportamiento

Los patrones de diseño de comportamiento están relacionados con los algoritmos y la asignación de responsabilidades entre objetos, es decir, no sólo describen los patrones de objetos o clases, también se encargan de los patrones de comunicación entre ellos. Este patrón se caracteriza por un complejo control de flujo, el cual es difícil manejar en tiempo de ejecución, por lo tanto, provoca que se cambie el enfoque lejos del control de flujo para concentrarse exclusivamente en la manera en la que los objetos se comunican.

Los patrones de comportamiento más conocidos son:

- Método Template
- Interpreter
- Mediator
- Chain of responsibility
- Observer
- Strategy
- Command
- State

El patrón de comportamiento de clases utiliza la herencia para distribuir el comportamiento entre clases. Dentro de los patrones de este tipo se encuentran: el método Template (modelo) y el Interpreter (intérprete).

El método Template es un patrón muy utilizado y fácil de implementar, consiste en una definición abstracta de un algoritmo, paso a paso, donde en cada paso se invoca ya sea a una operación abstracta o a una operación primitiva.

El Interpreter define un conjunto de normas y reglas como una jerarquía de clases, e implementa un intérprete como una operación en las instancias de estas clases.

El patrón de comportamiento de objetos se basa la composición de objetos más que en la herencia. Algunos describen cómo un grupo de dos objetos cooperan para desarrollar una tarea que no podría realizar un objeto solo. Un detalle que sobresale es cómo el par de

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	206/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

objetos se conocen para cooperar entre sí, ya que entonces los objetos deberían tener una referencia uno hacia el otro, lo que incrementaría el acoplamiento. Además, en un caso extremo, cada objeto debería poseer una referencia hacia todos los otros objetos.

El patrón Mediator (mediador) permite optimizar las referencias de los pares de objetos utilizando un objeto mediador entre ellos, y como los objetos no necesitan este objeto mediador se elimina el acoplamiento.

El patrón de Chain of responsibility (cadena de responsabilidades) también permite bajar el acoplamiento entre objetos enviando solicitudes a un objeto utilizando una cadena de objetos candidatos, donde cualquier candidato podría cumplir la solicitud dependiendo de las condiciones de ejecución. El número de candidatos es ilimitado y se puede seleccionar qué objetos van a ser parte de la cadena de candidatos en tiempo de ejecución.

El patrón Observer (observador) define, mantiene y administra una dependencia entre objetos, informando en tiempo real sobre los cambios que se presentan en uno u otro objeto.

El patrón Strategy (estratégico) encapsula un algoritmo dentro de un objeto, haciendo con esto fácil de especificar y cambiar el algoritmo que utiliza un objeto.

El patrón Command (comando) encapsula una solicitud dentro de un objeto, de tal manera que éste se puede pasar como parámetro, guardar en un histórico o manipular de alguna otra manera.

El patrón State (estado) encapsula el estado de un objeto, de tal forma que el objeto puede cambiar su comportamiento cuando el estado del objeto cambie.

	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	207/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Patrón de diseño de comportamiento

Nombre: Observer.

Problema:

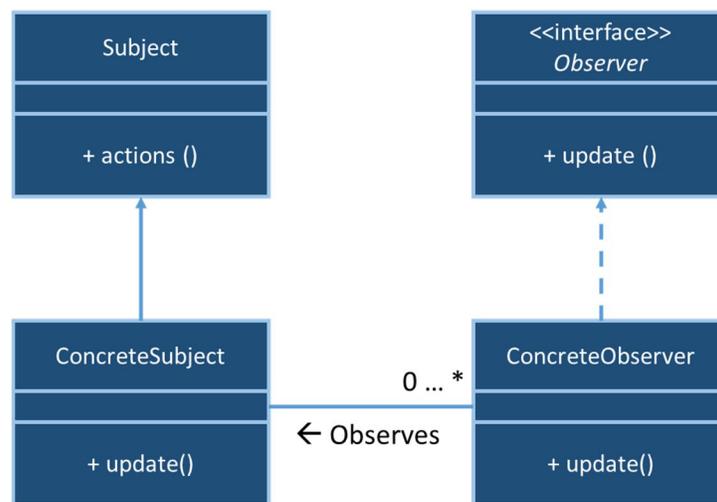
Se necesita notificar a un grupo de diferentes objetos que un evento ha ocurrido en la aplicación.

Un cambio en el estado de un objeto requiere que se cambie el estado de otros objetos, sin saber cuántos son los objetos que deben reflejar el cambio.

Solución.

Se crea una clase abstracta la cuál será la encargada de manejar un conjunto de objetos observadores en la aplicación.

Cuando un cambio ocurra en el sujeto observado, se debe notificar a todos los objetos observadores dentro del conjunto.



Modelo Singleton

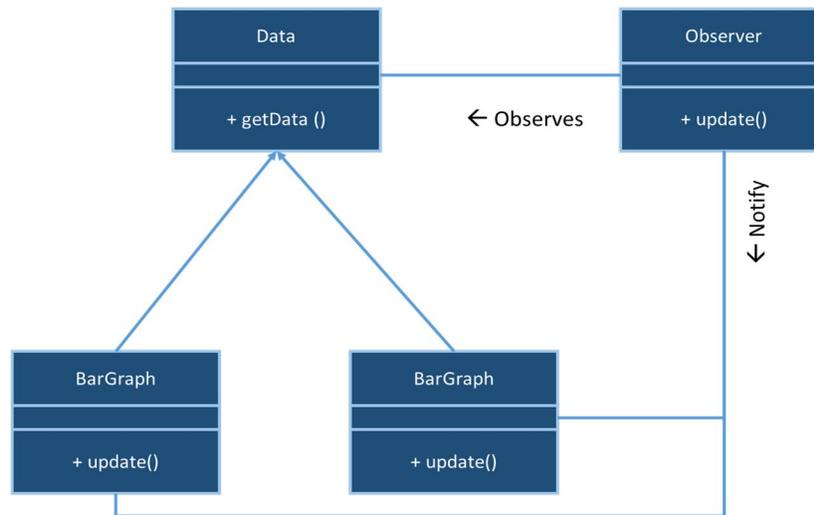
	Manual de prácticas del Laboratorio de Programación orientada a objetos	Código:	MADO-22
		Versión:	01
		Página	208/208
		Sección ISO	8.3
		Fecha de emisión	20 de enero de 2017
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Consecuencias

El patrón de diseño de comportamiento Observer permite partir un sistema en una colección de clases cooperativas, generando con esto consistencia de información entre los objetos relacionados. Permite que la clase tenga un bajo acoplamiento y se pueda reutilizar.

Ejemplo:

Se desean crear gráficas de una aplicación gráfica que permita separar el aspecto (la presentación) de la información subyacente. Además, se desea que los datos y la presentación de los mismos se puedan reutilizar de manera independiente y trabajar en conjunto.



Bibliografía

Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Addison Wesley Professional, 1994