



Tecnicatura Universitaria
en Programación

LABORATORIO DE COMPUTACIÓN III

Unidad Temática N°1:
Introducción al lenguaje Java

Material de Estudio
2° Año – 3° Cuatrimestre



Índice

ONBOARDING & SETUP 2

Control de versiones	2
JDK (Java Development Kit)	3
Herramientas de empaquetamiento (Construcción)	3
IDE (Integrated Development Environment)	5

INTRODUCCIÓN AL LENGUAJE JAVA 7

Lenguaje Java	7
Programa mínimo	9
Sintaxis de Java	11
Tipos de datos	13
Variables	17
Operadores	22
Estructuras de control	29
Entrada/Salida estándar	34

PROGRAMACIÓN ORIENTADA A OBJETOS 36

Introducción	36
Fundamentos de la programación orientada a objetos	36
Clases	37
Atributos	38
Propiedades (getters y setters)	39
Métodos	40
Constructores	41

BIBLIOGRAFÍA 42

ONBOARDING & SETUP

Como parte del desarrollo de la asignatura, vamos a aprender a instalar y usar un set de herramientas básicas que todo programador debe conocer y haremos foco en algunas que son específicas de la programación con Java.

Control de versiones

Con un sistema de control de versiones, los desarrolladores pueden mantener **un registro de todos los cambios que se realizan en el código fuente**, incluyendo **quién** hizo el cambio, **cuándo** se hizo y **por qué** se hizo. Esto permite a los desarrolladores trabajar juntos de manera más eficiente, ya que pueden colaborar en el mismo código fuente sin temor a sobrescribir el trabajo de otros o perder cambios importantes. Es importante que un desarrollador conozca sobre control de versiones porque le permite mantener un registro a lo largo del tiempo, lo que facilita la colaboración y la gestión de proyectos de software en general ya que permite a los desarrolladores crear ramas de desarrollo separadas para trabajar en nuevas funcionalidades o correcciones de errores, mientras que la rama principal del proyecto sigue siendo estable y funcional. También permite revertir cambios en el código fuente si se detectan errores o se desea volver a una versión anterior del software.

En resumen, conocer y utilizar un sistema de control de versiones es fundamental para cualquier desarrollador que desee trabajar y gestionar el código de manera eficiente y efectiva.

Para esta asignatura usaremos el sistema de control de versiones Git.

Git

Git es un sistema de **control de versiones distribuido de código abierto**, que fue creado por Linus Torvalds en 2005 para el desarrollo del kernel de Linux. Git es ampliamente utilizado en la comunidad de desarrollo de software para controlar y gestionar el código fuente de los proyectos.

A diferencia de los sistemas de control de versiones centralizados, en los que todos los desarrolladores trabajan en un mismo repositorio central, Git permite que cada desarrollador tenga una copia completa del repositorio en su propio equipo. Esto hace que Git sea más resistente a los fallos y más rápido en las operaciones de clonación y fusión de ramas de desarrollo.

Git utiliza un modelo de ramificación flexible que permite a los desarrolladores crear ramas de desarrollo independientes para trabajar en nuevas características o arreglar errores, sin afectar la rama principal del proyecto. Además, proporciona una

serie de herramientas para la resolución de conflictos y la colaboración entre desarrolladores.

Git se utiliza en una gran variedad de proyectos de software, desde pequeñas aplicaciones hasta proyectos empresariales de gran envergadura. También es compatible con una gran cantidad de plataformas de alojamiento de repositorios, como GitHub, GitLab o Bitbucket.

Los detalles sobre cómo instalar y usar esta herramienta, junto con los comandos más importantes, los encontrarán en el anexo **“TUP 3C LCIII ANEXO 1 Git”**.

JDK (Java Development Kit)

Es el kit de desarrollo de Java que incluye el compilador Java, la máquina virtual Java (JVM), y otras herramientas que permiten a los programadores escribir y ejecutar programas Java. Es importante mencionar que la JDK es necesaria para desarrollar aplicaciones Java y está disponible de forma gratuita en el sitio web de Oracle.

Más adelante hablaremos con más detalle sobre la JDK y su importancia para desarrollar programas en Java. En el anexo **“TUP 3C LCIII ANEXO 2 JDK”** esta el detalle de las alternativas de como hacer la instalación de esta herramienta de acuerdo al sistema operativo.

Herramientas de empaquetamiento (Construcción)

Las herramientas de empaquetamiento son herramientas de construcción de software que se utilizan para crear paquetes de distribución de una aplicación de software en un formato que pueda ser instalado y utilizado por los usuarios finales. Estas herramientas suelen ser utilizadas en proyectos de software de gran escala que tienen múltiples dependencias y componentes, y se utilizan para simplificar el proceso de distribución y la instalación del software.

Estas pueden crear paquetes en diferentes formatos, según las necesidades del proyecto. Algunos formatos comunes son:

- Archivos ZIP o JAR: Son archivos comprimidos que contienen todos los archivos y dependencias necesarias para ejecutar la aplicación de software.
- Archivos de instalación: Son programas de instalación que se utilizan para instalar y configurar la aplicación de software en el equipo del usuario final.

- Contenedores: Son entornos de ejecución virtualizados que contienen todos los componentes necesarios para ejecutar la aplicación de software, incluyendo el sistema operativo, las bibliotecas y las dependencias.

Las herramientas más populares para proyectos de Java son Maven y Gradle, que permiten empaquetar y distribuir aplicaciones en diferentes formatos. También existen otras herramientas de empaquetamiento que se utilizan en otros lenguajes de programación, como npm para proyectos de JavaScript, pip para proyectos de Python, y NuGet para proyectos de .NET.

En resumen, las herramientas de empaquetamiento son herramientas de construcción de software que se utilizan para crear paquetes de distribución de una aplicación de software en un formato que pueda ser instalado y utilizado por los usuarios finales.

Maven

Maven es una herramienta de construcción de proyectos de software que se utiliza principalmente en proyectos Java. Es un proyecto de software libre de la Apache Software Foundation y es una de las herramientas de construcción más utilizadas en la comunidad de desarrolladores de Java.

Maven se utiliza para automatizar el proceso de construcción de un proyecto de software. Esto incluye:

- la descarga de dependencias
- la compilación de código fuente
- la ejecución de pruebas
- la generación de documentación
- la empaquetación del proyecto en un formato distribuible.

Una de las características más importantes de Maven es su capacidad para gestionar las dependencias de un proyecto. Esto significa que, en lugar de descargar y mantener manualmente las bibliotecas y dependencias necesarias para un proyecto, Maven se encarga de ello automáticamente. Maven utiliza un archivo de configuración llamado "**pom.xml**" para gestionar las dependencias y configurar el proyecto. Otra característica importante es su compatibilidad con el repositorio central de Maven, que es un repositorio de artefactos de software mantenido por la comunidad de desarrolladores de Java. Esto significa que los desarrolladores pueden compartir y reutilizar bibliotecas y dependencias en diferentes proyectos de Java, lo que reduce la duplicación de código y simplifica la gestión de dependencias.

Maven también se integra bien con otras herramientas de desarrollo de software, como IDEs (entornos de desarrollo integrados) y herramientas de control de versiones como Git.

En resumen, Maven es una herramienta de construcción de proyectos de software muy popular en la comunidad de desarrolladores de Java que se utiliza para automatizar el proceso de construcción de un proyecto y gestionar sus dependencias. La capacidad de Maven para trabajar con el repositorio central de Maven y su integración con otras herramientas de desarrollo hacen que sea una herramienta muy poderosa y versátil.

Los detalles sobre cómo instalar y usar esta herramienta los encontrarán en el anexo “TUP_3C_LCIII_ANEXO_3_Maven”.

IDE (Integrated Development Environment)

IDE son las siglas en inglés de Integrated Development Environment, lo que se traduce como “Entorno de Desarrollo Integrado”. Es un tipo de software que proporciona a los programadores una plataforma completa y unificada para el desarrollo de software.

Un IDE típicamente incluye:

- un editor de código fuente
- herramientas de depuración, compilación y construcción
- herramientas de productividad y automatización que ayudan a los programadores a desarrollar software de manera más eficiente

Además, muchos IDEs incluyen características adicionales como:

- la integración con sistemas de control de versiones
- soporte para múltiples lenguajes de programación
- la capacidad de crear y administrar proyectos de software complejos

Los IDEs están diseñados para facilitar la tarea de desarrollo de software y ayudar a los programadores a desarrollar, probar y depurar aplicaciones de software por lo que son ampliamente utilizados en la industria del software. Algunos ejemplos de IDEs populares para Java son Eclipse, IntelliJ IDEA, Visual Studio y NetBeans.

En resumen, un IDE es un software que proporciona a los programadores una plataforma completa y unificada para el desarrollo de software, con herramientas para escribir, depurar, compilar y construir aplicaciones de software. Los IDEs son ampliamente utilizados en la industria del software para ayudar a los programadores a desarrollar software de manera más eficiente y productiva.

IntelliJ IDEA

IntelliJ IDEA (JetBrains) es uno de los IDE más populares para Java y se utiliza ampliamente en la industria del software para el desarrollo de aplicaciones empresariales, aplicaciones web, aplicaciones móviles y otros proyectos de software.

IntelliJ proporciona, además de las herramientas comunes de cualquier IDE, una serie de características avanzadas, como:

- el soporte para múltiples lenguajes de programación
- la integración con sistemas de control de versiones
- la generación automática de código
- el soporte para frameworks de desarrollo como Spring y Hibernate
- la finalización de código
- la corrección automática de errores
- la navegación de código y la refactorización
- integración con otras herramientas de desarrollo de software, como Maven, Gradle y Ant

lo que permite a los desarrolladores escribir código con mayor rapidez y precisión.

En resumen, IntelliJ IDEA es un IDE de Java y otros lenguajes de programación desarrollado por JetBrains que proporciona una amplia gama de herramientas para el desarrollo de software. Con su editor de código inteligente y sus características avanzadas de productividad, IntelliJ IDEA ayuda a los desarrolladores a escribir código de manera más rápida y eficiente.

INTRODUCCIÓN AL LENGUAJE JAVA

Lenguaje Java

Java es un lenguaje de programación de alto nivel, orientado a objetos y diseñado para ser portable y fácil de usar.

Es un lenguaje de programación de **alto nivel** porque fue diseñado para ser fácil de entender y utilizar por parte de los programadores. En contraposición, los lenguajes de programación de bajo nivel, como el lenguaje ensamblador, están diseñados para interactuar directamente con el hardware de la computadora, lo que los hace mucho más difíciles de aprender y utilizar. Java se centra en proporcionar abstracciones para el programador y en ocultar los detalles del hardware y la memoria de la computadora. Esta abstracción lo hace **portable**, ya que permite que los programas escritos en Java pueden ejecutarse en cualquier plataforma que tenga una máquina virtual Java (JVM) instalada. Esto se debe a que los programas escritos en Java se compilan en un código objeto llamado "bytecode" en lugar de un código de máquina específico de la plataforma. El bytecode es un formato de archivo binario que puede ser interpretado por cualquier máquina virtual Java, independientemente de la plataforma subyacente. Está **orientado a objetos** ya que se basa en el concepto de objetos, que pueden representar cosas del mundo real, como personas, animales, vehículos, entre otros. Los objetos tienen propiedades y métodos que se utilizan para describir su comportamiento y las acciones que pueden realizar.

El sitio [oracle.com](https://www.oracle.com) nos dice lo siguiente:

“Oracle Java es la plataforma número uno de lenguaje de programación y desarrollo. Reduce costos, acorta los plazos de desarrollo, impulsa la innovación y mejora los servicios de las aplicaciones. Java sigue siendo la plataforma de desarrollo preferida por empresas y desarrolladores, y cuenta con millones de desarrolladores que ejecutan más de 60 mil millones de máquinas virtuales Java en todo el mundo”.

Existen dos componentes claves del proceso de ejecución de un programa en Java, el compilador y el intérprete.

El **compilador** de Java se utiliza para convertir el código fuente escrito en el lenguaje de programación Java en bytecode, un formato de archivo binario que se puede ejecutar en cualquier plataforma con una máquina virtual Java (JVM) instalada. Por otro lado, el **intérprete** de Java se utiliza para cargar y ejecutar el bytecode generado por el compilador.

A continuación se describe el proceso de compilación e interpretación de Java:

1. El programador escribe el código fuente en el lenguaje de programación Java utilizando un editor de texto o un entorno de desarrollo integrado (IDE).
2. El compilador de Java toma el código fuente y lo compila en bytecode, que es un formato de archivo binario que puede ejecutarse en cualquier plataforma con una JVM instalada. El bytecode se almacena en un archivo con extensión ".class".
3. Cuando se ejecuta un programa Java, el intérprete de Java carga y ejecuta el bytecode generado por el compilador.
4. Durante la ejecución, el intérprete de Java convierte el bytecode en código de máquina específico de la plataforma subyacente (donde se está ejecutando la JVM). Esto se hace a través de un proceso llamado "just-in-time" (**JIT**) compilation, que convierte el bytecode en código de máquina a medida que se ejecuta el programa.

El intérprete de Java también realiza otras tareas, como la gestión automática de la memoria y la seguridad del programa, que son características clave de Java.

En resumen, el compilador de Java convierte el código fuente escrito en Java en bytecode, mientras que el intérprete de Java se encarga de cargar y ejecutar el bytecode en cualquier plataforma que tenga una JVM instalada. El proceso de JIT compilation convierte el bytecode en código de máquina específico de la plataforma a medida que se ejecuta el programa.

Java Development Kit

JDK (Java Development Kit) es un conjunto de herramientas que se utiliza para desarrollar aplicaciones en Java. Incluye un compilador Java, la máquina virtual Java (JVM), bibliotecas de clases Java, y otras herramientas que permiten a los programadores escribir y ejecutar programas Java.

A continuación, se detallan algunas de las herramientas que se encuentran en la JDK:

- Java Compiler: El compilador de Java.
- Java Virtual Machine (JVM): Responsable de ejecutar el bytecode generado por el compilador Java.
- Bibliotecas de clases Java: La JDK incluye una gran cantidad de bibliotecas de clases Java que proporcionan una amplia gama de funcionalidades para los programadores, como acceso a bases de datos, manipulación de archivos, gestión de ventanas y mucho más.
- Herramientas de desarrollo: La JDK también incluye una variedad de herramientas para desarrolladores, como el depurador de Java (JDB), el

monitor de comandos (JConsole), y el generador de documentación (Javadoc).

- Herramientas de empaquetamiento y distribución: La JDK incluye herramientas para empaquetar y distribuir aplicaciones Java, como el Archivo Java (JAR).

Programa mínimo

Todo programa Java debe poseer como mínimo las siguientes líneas de código:

```
public class HolaMundo {  
  
    no usages  
    public static void main(String[] args)  
    {  
        System.out.println("Hola mundo!!!");  
    }  
}
```

Imagen 1: Elaboración propia.

Como se evidencia el programa debe estar escrito dentro de un bloque de código cuya cabecera indica “class <NOMBRE_CLASE>”. La palabra class indica el inicio de una clase como en cualquier otro lenguaje orientado a objetos. El identificador “HolaMundo” a continuación define el nombre de dicha clase. En Java todo el código va a pertenecer a una clase aun cuando no se esté realizando programación orientada a objetos. En las próximas secciones se avanzará en esta temática.

Java exige que cada clase se encuentre almacenada en un archivo cuyo nombre coincida exactamente (incluyendo mayúsculas y minúsculas) con el nombre de la clase, por lo tanto el código anterior debe ser almacenado en un archivo HolaMundo.java.

Dentro de esta clase se encuentra un método denominado main, el cual indica el punto de entrada al programa, es decir, el inicio del mismo. Desde este método se pueden crear objetos y llamar a otros métodos. Cuando el método main finaliza, el programa se termina y se descarga la memoria.

Para presentar una línea de texto en la pantalla se utiliza la instrucción `System.out.println`, la cual recibe como parámetro una cadena con el texto que se desea presentar en la pantalla. El parámetro en el ejemplo anterior es una cadena fija (una constante de tipo `String`), pero también pueden pasarse variables para que se muestre su valor, o expresiones para que se muestre el resultado de las operaciones involucradas. Y si el tipo de dato de tales variables o expresiones no es `String`, también se muestra el dato sin necesidad de realizar operaciones de conversión.

Ejecución

Para poder ejecutar el programa se requiere realizar los pasos de compilación y ejecución. Para realizar la compilación y generación del código intermedio (los `bytecodes`) se invoca al programa compilador indicando el nombre del archivo de código fuente:

```
javac HolaMundo.java
```

Si la compilación finaliza sin errores, el compilador no muestra ningún mensaje y genera un archivo con el mismo nombre pero con extensión `.class`. Este archivo contiene el código de bytes y puede ser leído por la JVM para ejecutar el programa. La máquina virtual consiste en un programa llamado `java` el cual debe ser invocado indicando el nombre de la clase que contiene el punto de inicio, es decir, el método `main`:

```
java HolaMundo
```

Al ejecutarse el programa se presentan en la consola todos los datos impresos mediante el método `println`:

```
Hola mundo!!!
```

Sintaxis de Java

Java tiene una estructura de código clara y fácil de leer. Su sintaxis está basada en C y C++, pero es más simplificada y fácil de aprender. Java también cuenta con una amplia biblioteca de clases estándar que hacen que la programación sea más fácil y rápida.

Reglas de sintaxis

Al igual que otros lenguajes, no solo de programación, Java posee ciertas **reglas de sintaxis** que gobiernan cómo se escribe el código en este lenguaje y posee otras **convenciones** que hacen a las buenas prácticas de escritura de código en Java. Podríamos decir que **si una regla no se cumple, el programa no compilará**, pero sí una convención no se cumple si, aunque esto último no es aceptable por la comunidad de desarrolladores que trabajan en este lenguaje.

Nombre de archivos

Los archivos de código fuente llevan el sufijo “.java”, los archivos de bytecode llevan el sufijo “.class”

Nombres de variables, métodos, clases y constructores

Los nombres de las variables y métodos deben comenzar con una letra (**no pueden comenzar con un número**), seguida de letras, números o guiones bajos. Además, los nombres de las variables y métodos no pueden ser palabras reservadas en Java.

Los nombres de las clases deben comenzar con una letra mayúscula y deben seguir el estilo CamelCase.

Palabras reservadas

Java, al igual que los demás lenguajes, tiene una serie de palabras reservadas, como "class", "public", "static" y "void", que tienen un significado especial en el lenguaje. **Estas palabras no pueden utilizarse como nombres de variables o métodos.**

A continuación, se listan las palabras reservadas por el sistema:

abstract	boolean	break	byte	case
catch	char	class	continue	default
do	double	else	extends	false
final	finally	float	for	if
implements	import	instanceof	int	interface
long	native	new	null	package
private	protected	public	return	short
static	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while	var	rest
byvalue	cast	const	future	generic
goto	inner	operator	outer	strictfp

Tabla 1: Elaboración propia.

Símbolos de Puntuación

Java utiliza una serie de símbolos de puntuación, como llaves, paréntesis y puntos y comas, para indicar la estructura del código. **Estos símbolos deben colocarse en la posición correcta en el código para que el compilador de Java pueda analizar el programa correctamente.**

1. Cada instrucción debe finalizar con ; (símbolo de punto y coma)
2. Las estructuras de control (condicionales y repetitivas) no requieren punto y coma porque poseen un bloque delimitado por llaves {}

Mayúsculas y minúsculas

Java distingue entre mayúsculas y minúsculas, lo que significa que "MyVariable" y "myvariable" se consideran dos variables diferentes.

Comentarios

Java permite a los programadores agregar comentarios al código para hacerlo más legible y para explicar el funcionamiento del programa. Los comentarios se escriben utilizando "//" para comentarios de una sola línea y "/* */" para comentarios de varias líneas.

Convenciones

Naming Conventions

Los nombres de los métodos y las variables deben comenzar con una letra minúscula y seguir el estilo CamelCase.

Los nombres de las constantes deben escribirse en mayúsculas y separar las palabras con guiones bajos.

En el siguiente [link](https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html) están todas las convenciones de nombres que aplican a Java.

(<https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html>)

Indentación

Cuando se indica un bloque todo su contenido debería escribirse con mayor indentación o sangría.

Tipos de datos

Java posee dos categorías principales de tipos de datos, **tipos de datos primitivos** y **tipos de datos de referencia**.

Java ofrece representaciones de datos que permiten operar tanto con datos primitivos como de referencia, por ejemplo, el dato primitivo “*int*” tiene su contraparte de referencia “*Integer*”. En cuanto a la elección de utilizar tipos de datos primitivos o de referencia, no hay una respuesta única y definitiva ya que depende de cada caso en particular.

Por lo general, se recomienda utilizar tipos de datos primitivos cuando se trabaja con valores simples, como números enteros, decimales, caracteres, entre otros. Esto se debe a que los tipos de datos primitivos son más eficientes en términos de tiempo de ejecución y uso de memoria. Por otro lado, se recomienda utilizar tipos de datos de referencia cuando se trabaja con estructuras de datos más complejas, como arreglos, listas, objetos, entre otros. Esto se debe a que los tipos de datos de referencia proporcionan una mayor flexibilidad y funcionalidad que los tipos de datos primitivos. En cualquier caso, es importante tener en cuenta el contexto de la aplicación y evaluar los requisitos de rendimiento y uso de memoria antes de decidir qué tipo de datos utilizar. En muchos casos, el uso de tipos de datos primitivos o de referencia no tendrá un impacto significativo en el rendimiento de la aplicación, por lo que la elección puede basarse en la simplicidad y claridad del código.

Como recomendación general, siempre que necesitemos representar objetos complejos o variables de la aplicación, usaremos datos de referencia y cuando necesitemos hacer un uso puntual y momentáneo (como dentro de una estructura de control) usaremos datos primitivos.

Datos Primitivos

Los tipos de datos primitivos son los **tipos de datos básicos, que no están basados en objetos**, a diferencia de los tipos de datos de referencia que son objetos que se crean a partir de clases. Estos tipos de datos son simples y no tienen métodos. Estos son:

byte	Representa valores enteros de 8 bits. Su rango va desde -128 a 127.
short	Representa valores enteros de 16 bits. Su rango va desde -32,768 a 32,767
int	Valores enteros de 32 bits. Su rango va desde -2,147,483,648 a 2,147,483,647
long	Valores enteros de 64 bits. Va desde -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
float	Representa valores de punto flotante (Decimales) de 32 bits
double	Representa valores de punto flotante (Decimales) de 64 bits
char	Representa un solo carácter Unicode de 16 bits.
boolean	Representa valores verdadero/falso (true/false)

Tabla 2: Elaboración propia.

Cuando se utilizan tipos de datos primitivos en Java, **la variable que los contiene almacena directamente el valor en la memoria**, por ejemplo, un valor entero se almacena en la memoria como un byte, un short o un int, dependiendo de su tamaño.

Datos de Referencia

Los tipos de datos de referencia son tipos de datos complejos que refieren **a objetos que se crean a partir de clases**. Estos tipos de datos pueden tener métodos y pueden ser nulos (es decir, no tener un valor asignado). Los tipos de datos de referencia **no se almacenan directamente en la memoria**, sino que se crean a partir

de clases y **se almacenan en el heap de memoria**. Es decir que cuando se utilizan tipos de datos de referencia, la variable que los contiene **no almacena directamente el valor del objeto, sino una referencia a su ubicación en la memoria**; este espacio de memoria se llama "heap". Cuando se crea un objeto de una clase o un array, se reserva un espacio de memoria en el heap para almacenar los datos del objeto o del array. La variable que contiene una referencia al objeto o al array simplemente almacena la dirección de memoria donde se encuentra almacenado en el heap.

Tipo de dato String

Aunque el tipo String es muy utilizado en Java, en realidad no es un tipo de dato primitivo, sino un tipo de dato de referencia que representa una secuencia de caracteres. Se utiliza para almacenar texto y se implementa como una clase en Java. Cuando se crea un objeto de la clase String, se almacena en el heap de memoria y se hace referencia a través de una variable que contiene una referencia a su ubicación en la memoria.

A continuación, se presentan algunos detalles adicionales sobre el tipo de dato String en Java:

- Las cadenas String en Java **se crean utilizando comillas dobles**, por ejemplo: "Hola, mundo".
- Las **cadenas String son inmutables**, lo que significa que una vez creadas, no se pueden modificar. Si se necesita una cadena modificada, se debe crear una nueva cadena a partir de la original. Esto se debe a que los objetos de la clase String son inmutables en Java para garantizar su seguridad y evitar errores.
- Las cadenas String en Java tienen una **longitud máxima de $2^{31}-1$ caracteres**.
- Las cadenas String son **comparadas utilizando el método equals()**, que devuelve true si las cadenas son iguales en contenido y estructura. El operador == compara las referencias de los objetos de la clase String, no su contenido.

String Pool

El String Pool es un área especial de la memoria en Java donde se almacenan las cadenas String creadas en una aplicación. El propósito de este pool de cadenas **es optimizar el uso de memoria y mejorar el rendimiento de las aplicaciones**.

Cuando se crea un objeto String utilizando comillas dobles (""), Java busca en el String Pool si ya existe una cadena igual y, si la encuentra, devuelve una referencia a esa cadena. Si no encuentra una cadena igual, crea una nueva cadena en el String Pool. Esto permite ahorrar una cantidad significativa de memoria, especialmente cuando se utilizan cadenas idénticas en diferentes partes de una aplicación, en lugar de crear múltiples objetos String en memoria para cada instancia de la cadena, el String Pool reutiliza una única instancia de la cadena en todo el programa.

Es importante tener en cuenta que las cadenas creadas mediante el método "new" no siguen el proceso mencionado antes, sino que se crean como objetos separados en el heap de memoria. Esto significa que, en lugar de buscar en el String Pool, Java crea una nueva instancia de cadena cada vez que se utiliza el operador new.

```
String s1 = "Hello World";
String s2 = "Hello World";
if(s1 == s2) { // Esto es verdadero ya que ambas cadenas apuntan al mismo espacio de memoria
    System.out.println("Verdadero");
} else {
    System.out.println("Falso");
}

String s3 = new String( original: "Hello World");
if(s1 == s3) { // Esto es falso ya que Java asignó este valor en un espacio nuevo.
    System.out.println("Verdadero");
} else {
    System.out.println("Falso");
}

if(s1.equals(s3)) { // Esto es verdadero ya que el método equals esta preparado para comparar el contenido de la memoria.
    System.out.println("Verdadero");
} else {
    System.out.println("Falso");
}
```

Imagen 2: Elaboración propia.

Tipos de datos para representar fechas

Una de las características más importantes de cualquier lenguaje de programación es la capacidad de manejar y trabajar con fechas. En Java, existen varios tipos de datos que se utilizan para representar fechas y horas, cada uno con sus propias características y usos específicos.

Entre los tipos de datos más comunes para trabajar con fechas en Java se encuentran:

- **java.util.Date**: es una clase que representa una fecha y hora específicas en el tiempo, medida en milisegundos desde la medianoche del 1 de enero de 1970 UTC. A pesar de ser uno de los tipos de datos más antiguos de Java

para trabajar con fechas, se ha vuelto obsoleto y se recomienda utilizar otras alternativas más modernas.

- **java.time.LocalDate**: es una clase que representa una fecha sin hora ni zona horaria. Se utiliza principalmente para operaciones de fecha simples, como cálculos de días transcurridos entre dos fechas.
- **java.time.LocalDateTime**: es una clase que representa una fecha y hora, sin zona horaria. Se utiliza para operaciones de fecha y hora más complejas, como cálculos de diferencia de tiempo entre dos momentos específicos.
- **java.time.ZonedDateTime**: es una clase que representa una fecha y hora en una zona horaria específica. Se utiliza para trabajar con fechas y horas en diferentes zonas horarias.

Cada uno de estos tipos de datos tiene sus propias características y se utiliza para diferentes propósitos. En general, se recomienda utilizar los tipos de datos más modernos y actualizados, como `LocalDate`, `LocalDateTime` y `ZonedDateTime`, ya que ofrecen una funcionalidad más completa y una mayor precisión en los cálculos de fechas y horas.

Variables

Java **es un lenguaje tipado** porque requiere que se especifique el tipo de datos que se almacena en cada variable. Esto significa que cada variable en Java debe ser declarada con un tipo específico antes de que se pueda utilizar en un programa. El uso de un sistema de tipos tiene varias ventajas.

- Ayuda a prevenir errores y fallos en el programa, ya que el compilador de Java puede detectar cualquier inconsistencia de tipos en el código. Por ejemplo, si se intenta asignar un valor de tipo "cadena de caracteres" a una variable de tipo "entero", el compilador de Java emitirá un error de tipo y evitará que el programa se compile.
- Permite una mejor optimización del código. Al conocer el tipo de datos que se almacena en una variable, el compilador de Java puede asignar la cantidad correcta de memoria para esa variable y optimizar la forma en que se accede a ella en el programa.

Las variables son las unidades mínimas de almacenamiento de datos. Cada una de ellas posee un tipo de datos, identificador y valor. En Java, una variable es una ubicación de memoria que se utiliza para almacenar un valor. Una variable se utiliza para representar un valor que puede cambiar durante la ejecución del programa.

Para declarar una variable en Java, se debe especificar su tipo y su nombre. Por ejemplo, para declarar una variable de tipo entero llamada "edad", se puede utilizar la siguiente sintaxis:

```
int edad;
```

Imagen 3: Elaboración propia.

Esto indica que la variable "edad" es de tipo entero y aún no tiene un valor asignado. Para asignar un valor a la variable, se puede utilizar el operador de asignación "=" y proporcionar el valor deseado. Por ejemplo, para asignar el valor 25 a la variable "edad", se puede utilizar la siguiente sintaxis:

```
edad = 25;
```

Imagen 4: Elaboración propia.

Es posible declarar y asignar un valor a una variable en una sola línea:

```
int edad = 25;
```

Imagen 5: Elaboración propia.

Ámbito de una variable

Java administra las variables utilizando un concepto llamado "ámbito" (alcance o scope). El ámbito de una variable se refiere a la parte del programa donde se puede acceder a ella. Una variable puede tener

- Ámbito de clase
- Ámbito de método
- Ámbito de parámetro
- Ámbito de bloque
- Ámbito de variable estática

Ámbito de Clase (Variables de Instancia)

En Java, las variables de ámbito de clase (también conocidas como "variables de instancia") son variables que se declaran en una clase y se utilizan en los métodos de esa clase. Estas variables son únicas para cada instancia de la clase y se almacenan en la memoria cuando se crea un objeto de la clase.

Las variables de ámbito de clase en Java tienen un alcance que es limitado a la clase en la que se declaran. Esto significa que no se pueden acceder a estas variables fuera de la clase, a menos que se declare un método público que permita el acceso a la variable. Además, cada objeto de la clase tiene su propia copia de las

variables de ámbito de clase. Esto significa que si se cambia el valor de una variable de ámbito de clase en un objeto, ese cambio no afectará a las variables de otros objetos de la misma clase.

Para declarar una variable de ámbito de clase en Java, se utiliza la palabra clave "private", "public" o "protected" (Más adelante hablaremos de los modificadores de acceso), seguida del tipo de datos de la variable y el nombre de la variable. Por ejemplo:

```
public class MiClase {  
    no usages  
    private int miVariable;  
    // otros métodos y variables de la clase  
}
```

Imagen 6: Elaboración propia.

En este ejemplo, la variable "miVariable" es de ámbito de clase y se puede acceder en todos los métodos de la clase "MiClase".

Ámbito de Método (Variables Local)

Las variables de ámbito de método (también conocidas como "variables locales") son variables que se declaran dentro de un método y se utilizan en ese método. Estas variables son únicas para ese método y se almacenan en la memoria mientras se ejecuta ese método.

Las variables de ámbito de método en Java tienen un alcance que es limitado al método en el que se declaran. Esto significa que no se pueden acceder a estas variables fuera del método, además, deben inicializarse antes de su uso. Si se intenta usar una variable de ámbito de método que no se ha inicializado, el compilador de Java mostrará un error.

Para declarar una variable de ámbito de método en Java, se utiliza el tipo de datos de la variable y el nombre de la variable dentro del método. Por ejemplo, en el siguiente código, la variable "miVariableDeMetodo" sólo es accesible dentro del bloque de código del método "miMetodo":

```
public void miMetodo() {  
    int miVariableDeMetodo = 10;  
    System.out.println(miVariableDeMetodo);  
    // otros códigos del método  
}
```


Imagen 7: Elaboración propia.

Si se intenta acceder la variable “miVariableDeMetodo” por fuera del scope (alcance) del método “local”, las IDE mostrarán un error y el compilador no podrá compilar el programa.

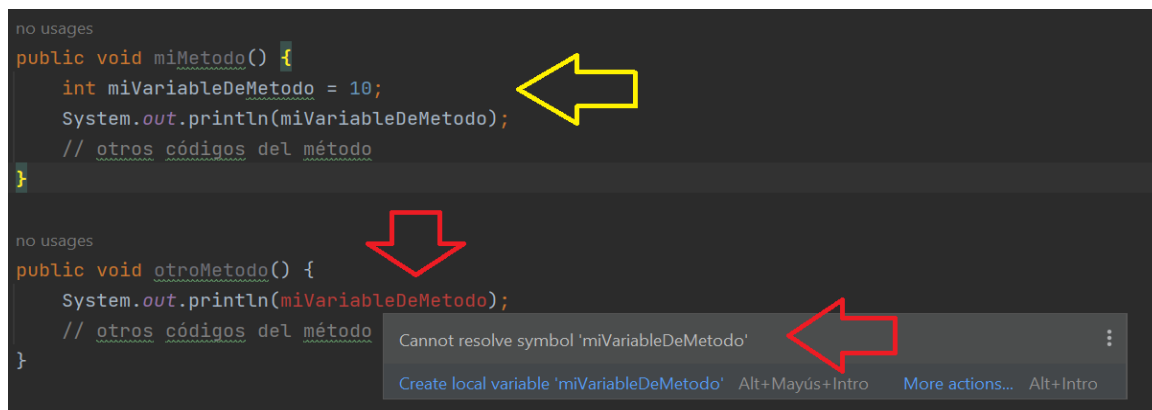


Imagen 8: Elaboración propia.

Ámbito de Parámetro (Variables de métodos)

Los parámetros son variables que se pasan a un método cuando se lo llama. Estas variables tienen un ámbito de método y son accesibles dentro del cuerpo del método. Los parámetros se declaran en la definición del método y pueden ser de cualquier tipo de datos. **Cuando se llama a un método en Java, se pasan argumentos como parámetros.** Estos parámetros pueden ser de cualquier tipo de datos que se pueda pasar como argumento, como tipos primitivos o tipos de referencia.

Es importante tener en cuenta que, aunque los parámetros de un método se definen como variables, su valor inicial se establece cuando se llama al método y se proporcionan los argumentos. Además, cualquier cambio que se haga en el valor de un parámetro dentro del cuerpo del método solo afecta a esa instancia del parámetro, no a la variable original fuera del método.

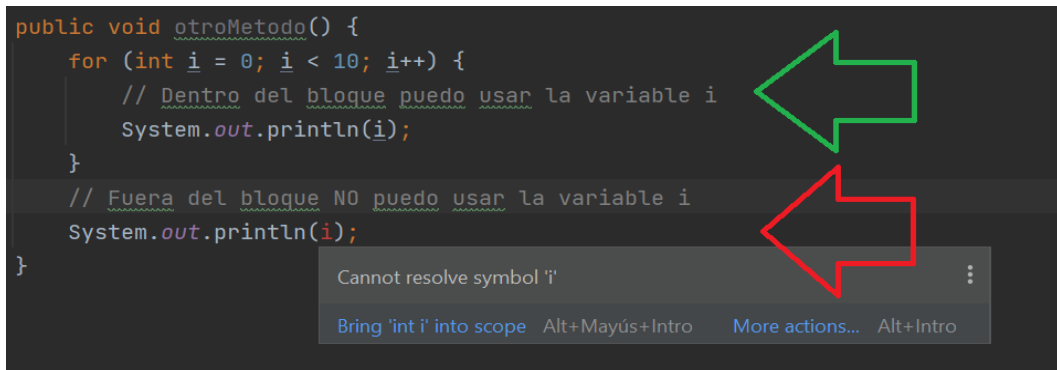
```
public void miMetodoConParametros(int parametro) {  
    System.out.println(parametro);  
}
```

Imagen 9: Elaboración propia.

Ámbito de Bloque (Variables de bloque)

Las variables que se declaran dentro de un ámbito de una estructura de control (como un “if”, “while”, “for”, entre otras) se denominan variables de ámbito de bloque o variables locales de bloque.

Estas variables son similares a las variables de ámbito de método en que su alcance está limitado al bloque o estructura de control en el que se declaran. Además, deben inicializarse antes de usarse. Un ejemplo de una variable de ámbito de bloque en un bucle “for” sería:



```
public void otroMetodo() {  
    for (int i = 0; i < 10; i++) {  
        // Dentro del bloque puedo usar la variable i  
        System.out.println(i);  
    }  
    // Fuera del bloque NO puedo usar la variable i  
    System.out.println(i);  
}
```

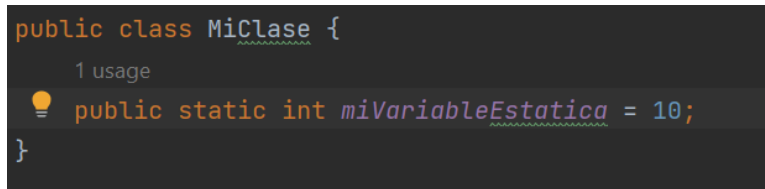
Imagen 10: Elaboración propia.

En este ejemplo, la variable “i” es una variable de ámbito de bloque y solo es accesible dentro del bucle “for”. Si se intenta acceder a la variable “i” fuera del bucle, se producirá un error de compilación.

Ámbito de variables estáticas (Variables estáticas)

Las variables estáticas se declaran con la palabra clave “static” y **se asocian con la clase en lugar de con una instancia de la clase**. Estas variables tienen un ámbito de clase y se pueden acceder desde cualquier parte del programa (de acuerdo a su accesibilidad). Estas son útiles para compartir datos comunes entre diferentes instancias de la misma clase.

Una variable estática **se crea cuando se carga la clase** y persiste hasta que finaliza la ejecución del programa. Se puede acceder a ella incluso cuando no se ha creado una instancia de la clase. **Cada instancia de la clase comparte la misma variable estática**, lo que significa que si se cambia el valor de en una instancia, ese cambio se reflejará en todas las demás instancias de la clase.



```
public class MiClase {  
    1 usage  
    public static int miVariableEstatica = 10;  
}
```

Imagen 11: Elaboración propia.

```
public class App
{
    no usages
    public static void main( String[] args ) {
        System.out.println(MiClase.miVariableEstatica);
    }
}
```

Imagen 12: Elaboración propia.

Operadores

Los operadores en Java **son símbolos que realizan una operación sobre uno o más operandos para producir un resultado**. Los operadores en Java se pueden clasificar en los siguientes tipos:

- Operadores aritméticos
- Operadores de asignación
- Operadores de comparación
- Operadores lógicos
- Operadores de incremento y decremento
- Operadores ternarios

Operadores aritméticos

Son los operadores que se utilizan para realizar operaciones aritméticas, como sumar, restar, multiplicar, dividir y obtener el resto de una división.

- Suma (+): se utiliza para sumar dos valores.
- Resta (-): se utiliza para restar un valor de otro.
- Multiplicación (*): se utiliza para multiplicar dos valores.
- División (/): se utiliza para dividir un valor por otro. Si ambos operandos son enteros, la división es entera.
- Módulo (%): se utiliza para obtener el resto de la división de un valor por otro.

```
public class App
{
    no usages
    public static void main( String[] args ) {
        int a = 10, b = 5, c = 2;

        // Suma
        int suma = a + b + 7;
        System.out.println(suma); // 22

        // Resta
        int resta = a - b;
        System.out.println(resta); // 5

        // Multiplicación
        int producto = b * b;
        System.out.println(producto); // 25

        // División
        int divisionEntera = a / b; // 2
        double divisionDecimal = b / c; // 2.0
        System.out.println(divisionEntera);
        System.out.println(divisionDecimal);

        // Modulo
        int modulo = b % c;
        System.out.println(modulo); // 1
    }
}
```

Imagen 13: Elaboración propia.

Operadores de asignación

Son los operadores que se utilizan para asignar un valor a una variable. El operador de asignación básico en Java es el signo igual (=), pero existen otros:

- Asignación simple (=): se utiliza para asignar el valor de la expresión a la variable.
- Asignación de suma (+=): se utiliza para sumar un valor a una variable y asignar el resultado a la variable.
- Asignación de resta (-=): se utiliza para restar un valor de una variable y asignar el resultado a la variable.
- Asignación de multiplicación (*=): se utiliza para multiplicar una variable por un valor y asignar el resultado a la variable.
- Asignación de división (/=): se utiliza para dividir una variable por un valor y asignar el resultado a la variable.
- Asignación de módulo (%=): se utiliza para obtener el resto de la división de una variable por un valor y asignar el resultado a la variable.

```
public class App
{
    no usages
    public static void main( String[] args ) {

        // Asignación Simple
        int a = 10, b = 5, c = 2;
        System.out.println(a); // 10
        System.out.println(b); // 5
        System.out.println(c); // 2

        // Asignación de suma
        a += b;
        System.out.println(a); // 15

        // Asignación de Resta
        b -= c;
        System.out.println(b); // 3

        // Asignación de Multiplicación
        c *= c;
        System.out.println(c); // 4

        // Asignación de División
        a /= b;
        System.out.println(a); // 5

        // Módulo
        a %= b;
        System.out.println(a); // 2
    }
}
```

Imagen 14: Elaboración propia.

Operadores de comparación

Los operadores de comparación en Java se utilizan para comparar dos valores y producir un resultado booleano (true o false) que indica si la comparación es verdadera o falsa.

- Igual que (==): comprueba si dos valores son iguales.
- Distinto de (!=): comprueba si dos valores son diferentes.
- Mayor que (>): comprueba si un valor es mayor que otro.
- Mayor o igual que (>=): comprueba si un valor es mayor o igual que otro.
- Menor que (<): comprueba si un valor es menor que otro.
- Menor o igual que (<=): comprueba si un valor es menor o igual que otro.

```
public class App
{
    no usages
    public static void main( String[] args ) {
        int a = 10, b = 5, c = 10;
        // Igual que
        System.out.println(a == b); // false
        System.out.println(a == c); // true
        System.out.println(a == (b * 2)); // true
        // Distinto de
        System.out.println(a != b); // true
        System.out.println(a != c); // false
        System.out.println(a != (b * 2)); // true
        // Mayor que
        System.out.println(a > b); // true
        System.out.println(a > c); // false
        System.out.println(b > a); // false
        // Mayor o igual que
        System.out.println(a >= b); // true
        System.out.println(a >= c); // true
        System.out.println(b >= a); // false
        // Menor que
        System.out.println(a < b); // false
        System.out.println(a < c); // false
        System.out.println(b < a); // true
        // Menor o igual que
        System.out.println(a <= b); // false
        System.out.println(a <= c); // true
        System.out.println(b <= a); // true
    }
}
```

Imagen 15: Elaboración propia.

Operadores lógicos

Son los operadores que se utilizan para combinar dos o más expresiones booleanas y devolver un valor booleano que indica si la combinación es verdadera o falsa.

- AND lógico (&): evalúa si ambas expresiones que se comparan son verdaderas. Si ambas expresiones son verdaderas, el resultado de la expresión es verdadero. Si alguna de las expresiones es falsa, el resultado es falso.
- OR lógico (||): evalúa si al menos una de las expresiones que se comparan es verdadera. Si alguna de las expresiones es verdadera, el resultado de la

expresión es verdadero. Solo si ambas expresiones son falsas, el resultado es falso.

- Negación lógica (!): niega una expresión booleana. Si la expresión es verdadera, la negación lógica la convierte en falsa, y si la expresión es falsa, la negación lógica la convierte en verdadera.

```
public class App
{
    no usages
    public static void main( String[] args ) {
        boolean a = true, b = true, c = false, d = false;
        // AND
        System.out.println(a && b); // true
        System.out.println(a && c); // false
        System.out.println(c && d); // false
        System.out.println(a && b && c); // false
        System.out.println(a && b && !c); // true
        // OR
        System.out.println(a || b); // true
        System.out.println(a || c); // true
        System.out.println(c || d); // false
        System.out.println(a || b || c); // true
        System.out.println(a || b || !c); // true
        // Negación
        System.out.println(a); // true
        System.out.println(c); // false
        System.out.println(!a); // false
        System.out.println(!c); // true
    }
}
```

Imagen 16: Elaboración propia.

Operadores de incremento y decremento

Son los operadores que se utilizan para incrementar o decrementar el valor de una variable en una unidad. Estos operadores se pueden usar tanto en expresiones como en sentencias, y vienen en dos formas preincremento/predecremento (aumenta o disminuye el valor de la variable antes de que se evalúe la expresión) y postincremento/postdecremento (aumenta o disminuye el valor de la variable después de que se evalúe la expresión).

- Incremento (++)
 - Pre Incremento (++variable)
 - Post Incremento (variable++)
- Decremento (--)
 - Pre Decremento (--variable)
 - Post Decremento (variable --)

```
public class App
{
    no usages
    public static void main( String[] args ) {
        int a = 5;
        // Pre Incremento / Decremento
        int b = ++a; // b es 6, a es 6
        System.out.println(b); // 6
        System.out.println(a); // 6
        int c = --a; // c es 5, a es 5
        System.out.println(c); // 5
        System.out.println(a); // 5

        // Post Incremento / Decremento
        int d = a++; // b es 5, a es 6
        System.out.println(d); // 5
        System.out.println(a); // 6
        int e = a--; // c es 6, a es 5
        System.out.println(e); // 6
        System.out.println(a); // 5
    }
}
```

Imagen 17: Elaboración propia.

Es importante tener en cuenta que los operadores de incremento y decremento solo se pueden usar con variables numéricas. Además, no se deben abusar de estos operadores, ya que pueden dificultar la legibilidad y el mantenimiento del código.

Operadores ternarios

Son los operadores que se utilizan para tomar decisiones basadas en una expresión booleana. El operador ternario en Java se compone de tres partes: la expresión booleana, la expresión que se evalúa si la expresión booleana es verdadera y la expresión que se evalúa si la expresión booleana es falsa.

```
expresión_booleana ? expresión_si_verdadero : expresión_si_falso;
```

Imagen 18: Elaboración propia.

Este operador evalúa una expresión booleana y devuelve uno de los dos valores posibles, dependiendo de si la expresión es verdadera o falsa. Si la expresión es verdadera, se devuelve la expresión "si verdadero", y si es falsa, se devuelve la expresión "si falso".

```
public class App
{
    no usages
    public static void main( String[] args ) {
        int a = 5;
        int resultado = (a > 10) ? 1 : 0;
        System.out.println(resultado); // 0
        int resultado2 = (a < 10) ? 1 : 0;
        System.out.println(resultado2); // 1
    }
}
```

Imagen 19: Elaboración propia.

Estructuras de control

En programación, las estructuras de control **son bloques de código que permiten tomar decisiones y controlar el flujo de ejecución de un programa**. Estas estructuras se utilizan para definir la lógica y el comportamiento del programa y permiten que éste se adapte a diferentes situaciones y condiciones. Existen tres tipos de estructuras de control en Java:

- estructuras de selección
- estructuras de repetición
- estructuras de salto

Estructuras de Selección

Las estructuras de selección permiten ejecutar diferentes bloques de código en función de condiciones específicas, dependiendo del valor de una expresión booleana. En Java estas son la estructuras de selección que existen:

- if: ejecuta un bloque de código si la expresión booleana es verdadera.
- if-else: ejecuta un bloque de código si la expresión booleana es verdadera y otro bloque de código si la expresión booleana es falsa.
- switch: permite seleccionar un bloque de código para ejecutar dependiendo del valor de una expresión.

```
public class App
{
    no usages
    public static void main( String[] args ) {
        int a = 5, b = 10;
        if(a == 5) {
            // código a ejecutar si la condición es verdadera
            System.out.println("Ejecuta solo si a es igual a 5");
        }
        System.out.println("Siempre se ejecuta esta línea de código");
        if(a == b) {
            // código a ejecutar si la condición es verdadera
            System.out.println("Ejecuta solo si a es igual a b");
        } else {
            // código a ejecutar si la condición es falsa
            System.out.println("Ejecuta como alternativa si la condición de if no se cumple.");
        }

        switch (a) {
            case 1:
                // código a ejecutar si la expresión tiene el valor de este caso (a == 1)
                System.out.println("a es igual a 1");
                break;
            case 5:
                // código a ejecutar si la expresión tiene el valor de este caso (a == 5)
                System.out.println("a es igual a 5");
                break;
            // ... Mas casos pueden ser incluidos
            default:
                // código a ejecutar si la expresión no coincide con ninguno de los casos anteriores
                System.out.println("a no es ni 1 ni 5");
        }
    }
}
```

Imagen 20: Elaboración propia.

Switch

La estructura de control "switch" en Java permite evaluar una condición y seleccionar el bloque de código a ejecutar. La variable es evaluada y **se compara con cada caso. Si el valor coincide con uno de los casos, se ejecuta el bloque de código correspondiente** y se sale del "switch" con la instrucción "break". Si no se incluye la instrucción "break" en el caso, el "switch" seguirá evaluando las condiciones y ejecutará todas aquellas con las que coincida. Si no se encuentra una coincidencia, se ejecuta el bloque de código en el "default" (opcional) y se sale del "switch".

```
public class App
{
    no usages
    public static void main( String[] args ) {
        int diaDeLaSemana = 4;
        String diaString = "";

        switch (diaDeLaSemana) {
            case 1:
                diaString = "Lunes";
            case 2:
                diaString = "Martes";
            case 3:
                diaString = "Miércoles";
            case 4:
                diaString = "Jueves";
            case 5:
                diaString = "Viernes";
            case 6:
                diaString = "Sábado";
            case 7:
                diaString = "Domingo";
        }

        System.out.println("Hoy es " + diaString); // Hoy es Domingo
    }
}
```

Imagen 21: Elaboración propia.

En este ejemplo, la variable “**diaString**” debería tener el valor “Jueves”, ya que “**diaDeLaSemana**” es igual a 4. Sin embargo, como no hay una instrucción `break` al final de cada caso, **el código ejecutará todas las instrucciones desde el caso 4 hasta el final del switch**, asignando el valor “Viernes”, “Sábado” y luego “Domingo” a la variable “**diaString**”. Para evitar esto, es importante incluir la instrucción `break` al final de cada caso.

Es importante destacar que la expresión o variable que se evalúa en el “switch” debe ser de tipo numérico (byte, short, int, char) o un objeto que implemente la interfaz “Comparable”. En Java 7 se añadió la capacidad de utilizar cadenas de caracteres (“String”) como expresiones en el “switch”.

Estructuras de repetición

Las estructuras de repetición, también conocidas como bucles, son una parte fundamental de la programación que permiten ejecutar un bloque de código varias veces hasta que se cumpla una condición específica. En Java, existen tres tipos de estructuras de repetición:

- `while` (0-n): se utiliza para repetir un bloque de código mientras se cumpla una condición específica
- `do-while` (1-n): es similar a la estructura `while`, pero se ejecuta al menos una vez, incluso si la condición es falsa.
- `for` (n): se utiliza para repetir un bloque de código un número determinado de veces.

`while`

En este ciclo el bloque iterativo se va a ejecutar una cantidad de veces que es desconocida en tiempo de compilación. Antes de iniciar cada iteración se evalúa una condición, la cual indica si se debe ejecutar la vuelta. Si la condición es verdadera, se ejecuta, si es falsa se interrumpe el ciclo. Dado que la primera vez que se evalúa la condición la misma puede ser falsa, el bloque puede no ejecutarse nunca.

En Java esta es la forma de escribir este tipo de estructura:

```
while (condición) {  
    // Código a ejecutar mientras se cumpla la condición  
}
```

Imagen 22: Elaboración propia.


```
public class App
{
    no usages
    public static void main( String[] args ) {

        int a = 0;

        while (a > 5) {
            System.out.println("Este mensaje nunca se mostrará en la consola");
        }
    }
}
```

Imagen 23: Elaboración propia.

do-while

Este ciclo es similar al anterior pero la condición de corte es evaluada luego de ejecutar el bloque iterativo. A causa de esto, aunque en la primera vuelta la condición devuelva falso, el bloque iterativo ya se habrá ejecutado. A causa de esto, este ciclo garantiza que al menos una vuelta como minimo es ejecutada. En el lenguaje java se lo programa con la instrucción do while.

En Java esta es la forma de escribir este tipo de estructura:

```
do {
    // Código a ejecutar al menos una vez
} while (condición);
```

Imagen 24: Elaboración propia.

```
public class App
{
    no usages
    public static void main( String[] args ) {

        int a = 0;

        while (a > 5) {
            System.out.println("Este mensaje nunca se mostrará en la consola");
        }

        do {
            System.out.println("Este mensaje se mostrará una vez en la consola");
        } while (a > 5);
    }
}
```

Imagen 25: Elaboración propia.

for

El ciclo exacto se denomina de esta manera ya que la cantidad de vueltas que realiza es conocida con exactitud con antelación a la ejecución del mismo. Para lograr esto se requiere una variable que funcione como un contador de vueltas y una condición de corte que compare dicho contador con la cantidad de vueltas requeridas.

En Java esta es la forma de escribir este tipo de estructura:

```
for (inicialización; condición; actualización) {  
    // Código a ejecutar mientras se cumpla la condición  
}
```

Imagen 26: Elaboración propia.

```
public class App  
{  
    no usages  
    public static void main( String[] args ) {  
        int a = 5;  
        for (int i = 0; i < a; i++) {  
            System.out.println("El valor de i es " + i);  
        }  
    }  
}
```

Imagen 27: Elaboración propia.

```
El valor de i es 0  
El valor de i es 1  
El valor de i es 2  
El valor de i es 3  
El valor de i es 4  
  
Process finished with exit code 0
```

Imagen 28: Elaboración propia.

Entrada/Salida estándar

Salida por consola

En las aplicaciones de consola de texto, la presentación de información al usuario y el ingreso de datos por éste se realizan a través de los dispositivos de entrada y salida estándar, es decir, el teclado y la ventana de texto. Para poder realizar estas operaciones Java provee dos objetos que manipulan todos los tipos de datos primitivos sin necesidad de conversiones o interpretación de la entrada (parsing).

Para presentar información al usuario se dispone del objeto `System.out` el cual posee varios métodos de impresión. Los dos que más se utilizan son `print()` y `println()`. Ambos funcionan de la misma manera, reciben un parámetro con el dato que se desea imprimir. El dato puede ser cualquier expresión, es decir: un valor constante, una variable o una operación. En este último caso se calcula el resultado de la operación y se lo muestra, sin necesidad de almacenarlo en una variable temporal.

La diferencia entre los métodos `print` y `println` reside en que el último agrega automáticamente al final de la impresión un carácter de nueva línea ("`\n`") forzando de esta manera a que el cursor baje a la línea siguiente para que la próxima impresión se visualice en otra línea. Si se utiliza `print` el cursor no se mueve y la próxima impresión se visualizará a continuación en la misma línea.

A causa de esto último se debe que `println()` puede ser invocado sin parámetros para dibujar una línea en blanco. En cambio, carece de sentido que `print()` se invoque sin parámetros y por lo tanto el compilador rechaza una llamada de esa naturaleza.

Ingreso de datos

Así como existe el objeto `System.out`, Java provee un objeto similar llamado `System.in` que permite el ingreso de datos por parte del usuario. Sin embargo `System.in` sólo ofrece métodos para ingresar datos de a un carácter por vez, y cuando se requiere ingresar números o cadenas debe invocarse múltiples veces al método de lectura, y procesando los caracteres ingresados concatenándolos y luego convirtiéndolos para obtener el dato definitivo. Esta tarea es sumamente compleja para un objetivo que debería ser muy simple, como el de ingresar un número entero o una cadena de caracteres.

Por eso Java agregó una clase que se encarga de tal tarea liberando al programador de dicha responsabilidad. La clase en cuestión se llama `Scanner` y se encuentra en el paquete `java.util`. Para poder utilizarla se necesita agregar ANTES del bloque `class` una instrucción "`import`" que permita acceder a la misma.

```
package ar.edu.utn.frc.tup.lciii;  
  
import java.util.Scanner;  
  
no usages  
public class App  
{  
    no usages  
    public static void main( String[] args ) {  
  
        Scanner scanner = new Scanner(System.in);  
  
        System.out.println("Ingrese un número: ");  
        int num1 = scanner.nextInt();  
  
        System.out.println("Ingrese otro número: ");  
        int num2 = scanner.nextInt();  
  
        int sum = num1 + num2;  
        System.out.println("La suma de " + num1 + " y " + num2 + " es: " + sum);  
    }  
}
```

Imagen 29: Elaboración propia.

```
Ingrese un número:  
10  
Ingrese otro número:  
5  
La suma de 10 y 5 es: 15  
  
Process finished with exit code 0
```

Imagen 30: Elaboración propia.

El nombre de la variable “scanner” puede ser modificado por cualquier otro, tal como consola, teclado o entrada (o cualquiera que el programador desee). El parámetro del constructor de Scanner se indica como System.in para poder leer datos desde el teclado, pero Scanner puede leer datos directamente desde un archivo de texto y lo único que se requiere para ello es reemplazar System.in por el archivo desde el que se desea leer los datos. (En realidad es un poco más difícil pero no resulta de interés analizar los detalles para esta materia).

Para leer un dato desde el teclado se invoca a uno de varios métodos llamados nextXXX(), reemplazando XXX por un tipo de datos. Existe un método por cada tipo de datos, cada uno de ellos convirtiendo y retornando un dato específico. De esta manera, para cargar un número entero simplemente se invoca a .nextInt() y el retorno del mismo se puede guardar en una variable de tipo int, sin necesidad de hacer una operación de parseo.

PROGRAMACIÓN ORIENTADA A OBJETOS

Introducción

La programación orientada a objetos es un paradigma de programación que se enfoca en modelar la realidad a través de objetos y sus interacciones. Java es uno de los lenguajes de programación más populares que utiliza la programación orientada a objetos como su enfoque principal. En Java, todo es un objeto, cada objeto tiene atributos y métodos, lo que significa que puede contener datos y realizar acciones “...las clases son moldes o modelos para construir objetos. En ellas se especifican los atributos y métodos generales a todos los objetos contruidos a partir de esa clase...” (Corso, 2012)

La encapsulación es una característica importante de la programación orientada a objetos en Java, que se refiere a la ocultación de los detalles internos del objeto y el acceso a ellos solo a través de métodos definidos. La herencia y el polimorfismo son otros conceptos clave de la programación orientada a objetos en Java, que permiten a los desarrolladores crear jerarquías de clases y definir comportamientos comunes para objetos relacionados.

Fundamentos de la programación orientada a objetos

La programación orientada a objetos (POO) es un paradigma de programación que se enfoca en el uso de objetos y sus interacciones para modelar el mundo real. A continuación, se presentan los fundamentos de la programación orientada a objetos:

1. Clases: una clase es una plantilla o modelo que define las propiedades y métodos comunes a un conjunto de objetos. En la POO, los objetos se crean a partir de clases y tienen las mismas propiedades y métodos definidos en la clase.
2. Objetos: un objeto es una instancia de una clase y tiene un estado (valores de sus atributos) y un comportamiento (métodos que se pueden invocar en el objeto).
3. Atributos: los atributos son las propiedades o variables que definen el estado de un objeto. Cada objeto de una clase tiene su propio conjunto de valores de atributos, aunque comparten la misma estructura de atributos definida en la clase.
4. Métodos: los métodos son las funciones o procedimientos que definen el comportamiento de un objeto. Los métodos pueden tener parámetros de entrada y pueden devolver un valor como resultado. Los métodos se pueden

utilizar para realizar operaciones sobre los atributos de un objeto y para interactuar con otros objetos.

5. Encapsulación: la encapsulación es un principio de la POO que consiste en ocultar el estado interno de los objetos y proporcionar una interfaz pública (métodos) para acceder y manipular ese estado. La encapsulación se utiliza para proteger los datos de la clase y para garantizar que los objetos interactúen solo a través de los métodos definidos en la clase.
6. Herencia: la herencia es un mecanismo en la POO que permite definir una clase nueva a partir de una clase existente (la clase base o superclase). La clase nueva (la subclase) hereda todas las propiedades y métodos de la clase base y puede agregar nuevos atributos y métodos. La herencia se utiliza para reutilizar el código y para definir jerarquías de clases.
7. Polimorfismo: el polimorfismo es la capacidad de objetos de diferentes clases de responder al mismo mensaje o método de una forma distinta. El polimorfismo permite diseñar soluciones más flexibles y escalables en la POO, ya que un objeto puede adoptar diferentes comportamientos dependiendo del contexto.

Clases

Una clase es una plantilla para la creación de objetos que comparten un conjunto común de atributos y comportamientos. **La clase describe los atributos y métodos que todos los objetos creados a partir de ella tendrán.** Para definir una clase se usa la palabra reservada “class”. La sintaxis de una clase es:

```
[public] [final | abstract] class NombreClase [extends ClaseBase]
[implements interface]
{
    // Atributos de la clase
    [modificadores] tipoDeDato nombreDeAtributo1;
    [modificadores] tipoDeDato nombreDeAtributo2;
    // getters & setters
    // ...

    // Métodos de la clase
    [modificadores] tipoDeRetorno nombreDelMetodo1 (parametros) {
        // Cuerpo del método
    }
    [modificadores] tipoDeRetorno nombreDelMetodo2 (parametros) {
        // Cuerpo del método
    }
}
```


Miembros

También según indica Corso, una clase contiene elementos, llamados miembros, que pueden ser datos, llamados atributos, y funciones que manipulan esos datos llamados métodos. En Java, los miembros de una clase son

- variables: Son espacios de memoria que se utilizan para almacenar valores.
- constantes: Son variables que se declaran con la palabra clave "final"
- métodos: Son bloques de código que realizan una acción específica
- clases anidadas: Son clases que se definen dentro de otra clase

Modificadores

En Java, existen cuatro modificadores de acceso que se pueden utilizar para controlar el nivel de acceso a los miembros de una clase:

- **public**: Se puede acceder al miembro desde cualquier lugar, incluso fuera de la clase.
- **private**: El miembro solo es accesible desde dentro de la misma clase. Esto significa que no se puede acceder a él desde otras clases ni siquiera a través de una instancia de la clase.
- **protected**: El miembro es accesible desde dentro de la misma clase y de sus subclases, pero no desde otras clases que no sean subclases.
- **default**: El miembro es accesible desde dentro del mismo paquete. Si no se especifica ningún modificador de acceso, por defecto se utiliza el modificador default.

Además de estos modificadores de acceso, también existen otros dos modificadores que se pueden utilizar para controlar el comportamiento de una clase:

- **final**: Indica que la clase no admite subclases.
- **abstract**: Indica que la clase es una clase abstracta. Las clases abstractas no se pueden instanciar directamente, pero se pueden utilizar como base para otras clases que las extiendan.

Atributos

En programación orientada a objetos, los atributos de una clase son las variables que se definen dentro de la clase y que representan las características o propiedades de los objetos que se crean a partir de la misma. También se les conoce como "campos" o "propiedades". Estos se definen como variables y se declaran dentro de la clase.

Todos los atributos se los declara con la siguiente sintaxis:

```
[modificadores] tipoDeDato nombreDeAtributo1;
```

Para acceder a los atributos de una clase, se utiliza el operador punto (.) seguido del nombre del atributo (siempre que sea publico) o seguido el metodo get de dicho atributo.

Propiedades (getters y setters)

El lenguaje Java no posee una construcción sintáctica especial para especificar las propiedades. A causa de esto se las simula mediante la creación de un par de métodos por cada atributo. Estos métodos son dos ya que se necesita uno para la operación de consulta de un atributo y otro para la asignación de un nuevo valor en el atributo.

Estos métodos pueden tener cualquier nombre, pero la convención que se ha establecido indica que todos los métodos de asignación deben llamarse setXXX reemplazando las XXX por el nombre del atributo, iniciado con mayúscula. Por su parte, los métodos de consulta deben llamarse getXXX.

```
1  package ar.edu.utn.frc.tup.lciii;
2
3  public class Persona {
4
5      private String nombre;
6      private int edad;
7      private char genero;
8
9      public String getNombre() { return nombre; }
12     public void setNombre(String nombre) { this.nombre = nombre; }
15     public int getEdad() { return edad; }
18     public void setEdad(int edad) { this.edad = edad; }
21     public char getGenero() { return genero; }
24     public void setGenero(char genero) { this.genero = genero; }
27
28     public Persona() {
29     }
30 }
31
```

Imagen 31: Elaboración propia.

Métodos

En Java, los métodos de una clase son bloques de código que definen las operaciones que pueden realizar los objetos de esa clase. Un método puede ser considerado como una función o subrutina que se ejecuta cuando se llama desde otro lugar del programa.

```
[modificadorAcceso][abstract] tipoRetorno nombreMetodo (tipoParametro1 parametro1, tipoParametro2 parametro2, ...) {  
    // Código del método  
    return valorRetorno;  
}
```

Parámetros

Cada método posee un conjunto de líneas de código que van a ser ejecutadas cuando el mismo es invocado. Normalmente los métodos realizan alguna tarea con los atributos del objeto que recibe la llamada, pero ocasionalmente un método puede necesitar otros datos que no están almacenados dentro de los atributos del objeto. Para poder obtener esos datos, el método puede solicitar argumentos o parámetros, que van a funcionar como variables locales, pero en las que el dato que almacenen sea especificado durante la invocación del método.

La lista de parámetros es la lista de nombres de variables separados por comas, con sus tipos asociados, que reciben los valores de los argumentos cuando se llama al método.

Un método puede recibir como parámetros valores prácticamente de cualquier tipo. Estos parámetros serán usados por el método para realizar operaciones que lleven a la acción que se espera generar.

Los parámetros que va a recibir un determinado método deben ser indicados en la cabecera del mismo, luego del identificador y entre paréntesis. La declaración propiamente dicha es similar a la que se realiza cuando se declara una variable, es decir, se indica el tipo y el nombre de cada parámetro, usando una coma para separar cada uno de ellos.

Los parámetros tienen el ámbito y duración del método. Esto significa que los parámetros se podrán ver y utilizar sólo dentro del método y al finalizar la ejecución del mismo, son borrados automáticamente.

Modificador de acceso	Clase actual	Mismo paquete	Subclases	Todas las clases
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
default	Sí	Sí	No	No
private	Sí	No	No	No

Tabla 3: Elaboración propia.

Constructores

Un constructor es un tipo especial de método que se llama automáticamente cuando se crea un objeto de una clase. El propósito principal de un constructor es inicializar los atributos de un objeto con valores predeterminados o valores proporcionados por el usuario. Los constructores también pueden realizar cualquier otra tarea necesaria para configurar un objeto para su uso.

En Java, el nombre del constructor debe ser el mismo que el nombre de la clase, y no debe tener un tipo de retorno. La sintaxis para definir un constructor en Java es la siguiente:

```
public class Persona {

    private String nombre;
    private int edad;
    private char genero;

    // Constructor sin parametros
    public Persona() {
    }

    // Constructor con parametros
    public Persona(String nombre, int edad, char genero) {
        this.nombre = nombre;
        this.edad = edad;
        this.genero = genero;
    }
}
```

Imagen 32: Elaboración propia.

BIBLIOGRAFÍA

- Ceballos Sierra, F. (2010). *Java 2. Curso de Programación*. 4ta Edición. Madrid, España. RA-MA Editorial.
- Corso, C; Colaccioppo, N. (2012). “*Apunte teórico-práctico de Laboratorio de Computación III*”. Córdoba, Argentina. Edición digital UTN-FRC.



Atribución-No Comercial-Sin Derivadas

Se permite descargar esta obra y compartirla, siempre y cuando no sea modificado y/o alterado su contenido, ni se comercialice. Universidad Tecnológica Nacional Facultad Regional Córdoba (S/D). Material para la Tecnicatura Universitaria en Programación, modalidad virtual, Córdoba, Argentina.