CURSO 2023-2024
CICLO SUPERIOR DE DESARROLLO DE APLICACIONES WEB
IES AGUADULCE

Programación

Francisco Javier Sueza Rodríguez

2 de octubre de 2023

Índice general

Índice de figuras

Tema 1

Introducción a la Programación

En esta primera unidad vamos a estudiar los conceptos básicos de la programación de aplicaciones. Comenzaremos estudiando que es la programación, que técnicas podemos emplear, que herramientas podemos utilizar y cual es objetivo que pretendemos alcanzar. Analizaremos las diferentes paradigmas de programación existentes, identificaremos las fases del desarrollo de un programa.

Una vez realizada una introducción general, detallaremos las características relevantes de los principales lenguajes de programación, para a continuación centrarnos en el lenguaje que vamos a usar durante toda esta asignatura, **Java**, dando a conocer también que herramientas podemos usar para que nuestro desarrollo sea más sencillo con este lenguaje.

1.1 Programas: Buscando una Solución

La principal razón que mueve a una persona hacia el aprendizaje de la programación es utilizar el ordenador como una herramienta para resolver diferentes problemas. Al igual que en la vida real, las búsqueda y obtención de una solución requiere de una serie de **pasos fundamentales**.

En la vida real...	En Programación...
Observación de la situación o problema.	**Análisis del problema:** requiere que el problema sea definido y comprendido claramente para que pueda ser analizado con todo detalle.
Pensamos en una o varias posibles soluciones.	**Diseño o desarrollo de algoritmos:** procedimiento paso a paso para solucionar el problema dado.
Aplicamos la solución que estimamos más adecuada.	**Resolución del algoritmo elegido en la computadora:** consiste en convertir el algoritmo en programa, ejecutarlo y comprobar que soluciona verdaderamente el problema.

Figura 1.1.1: Pasos para la resolución de un problema

Para que una **solución** se considere **correcta** tiene que tener principalmente dos características:

- **Corrección y Eficacia**: si resuelve el problema de forma adecuada.
- **Eficiencia**: si lo realiza en un tiempo mínimo y con un uso óptimo de los recursos del sistema.

Para construir esta solución, hay que tener en cuenta algunos conceptos ligados a la programación, como son:

1. **Abstracción**: se trata de realizar una análisis del problema para descomponerlo en problemas más pequeños y de menos complejidad de manera precisa. **Divide y Vencerás**: es una filosofía general para resolver problemas y de aquí que no solo forme parte del vocabulario informático, sino que también se utiliza en otros muchos ámbitos.
2. **Encapsulación**: consiste en ocultar la información de un objeto o función de forma que se pueda implementar de diferentes maneras sin que esto afecte al resto de objetos.
3. **Modularidad**: estructuraremos cada parte en módulos independientes, cada uno con su función correspondiente.

Todo estos conceptos, deberemos tenerlos en cuenta a la hora de analizar el problema, para llegar a un solución lo más óptima posible.

1.1.1 Algoritmos y Programas

Una vez realizado el análisis del problema, tenemos que diseñar y desarrollar un **algoritmo** adecuado que pueda solucionarlo. Pero, ¿que es un algoritmo?

Un **algoritmo** es una secuencia ordenada de pasos, descrita sin ambigüedades, que conducen a la solución de un problema.

Los algoritmos deben ser **independientes** de los **lenguajes de programación** y de las **computadoras** donde se ejecutan, de forma que puedan implementarse sobre cualquier ordenador empleando cualquier lenguaje de programación. Esto facilita que una misma solución pueda emplearse para el mismo problema en diferentes dispositivos.

La **diferencia** entre un algoritmo y un **programa** es que en este último lo pasos deben escribirse en un **lenguaje de programación concreto** para que pueda ser ejecutado en el ordenador y así obtener la solución deseada.

Los **lenguajes de programación** son solo un medio para expresar el algoritmo y el ordenador un procesador para ejecutarlo. El diseño de algoritmos es una tare que requiere de la creatividad y conocimientos de las técnicas de programación del programador, así, diferentes programadores pueden desarrollar diferentes algoritmos para resolver un mismo problema.

Las principales **características** que debe cumplir un **algoritmo** son:

- Debe ser **preciso** e indicar el orden en el que se realiza cada paso.
- Debe estar **bien definido**, si se ejecuta dos o más veces, debemos obtener el mismo resultado.
- Debe ser **finito**, teniendo un número de pasos bien determinado.

Cuando los problemas complejos, debemos descomponer estos en subproblemas más simples, y estos a su vez en otros más pequeños. Es lo que se conoce como **diseño descendente** o **diseño modular** y se basa en el lema **Divide y Vencerás**.

Para **representar gráficamente** los algoritmos tenemos diferentes herramientas que nos ayudarán a describir su comportamiento de una forma precisa y genérica, que nos facilitará la implementación del algoritmo en diferentes lenguajes de programación. Las principales herramientas que tenemos son:

- **Diagramas de Flujo**: esta técnica utiliza símbolos gráficos para representar el flujo de ejecución del algoritmo y suelen ser empleados en la fase de análisis.

- **Pseudocódigo**: se basa en el uso de palabras clave en lenguaje natural, representando las constantes, variables y otras estructuras de programación de forma escrita. Es la técnica mas utilizada actualmente.
- **Tablas de Decisión**: es un tabla que representa las diferentes condiciones del problema con sus respectivas acciones. Suele ser una técnica de apoyo a pseudocódigo cuando existen circunstancias condicionales complejas.

1.2 Fases de la Programación

Sea cual sea el estilo que escojamos para resolver el problema, deberemos realizar el proceso aplicando un método a nuestro trabajo. Así, el **proceso de creación de software** se puede dividir en las siguientes **fases**:

- **Fase de resolución del problema**
- **Fase de implementación**
- **Fase de explotación y mantenimiento**

En los siguientes puntos, analizaremos cada una de estas fases.

1.2.1 Resolución del Problema

Esta es la primera fase del desarrollo del programa, para la cuál deberá estar bien definido cual es el problema que se quiere solucionar y tener un comprensión clara de éste para poder realizar su análisis y diseño. Esta fase se puede dividir en **dos etapas**:

1. **Etapa de Análisis**:

   En esta primera etapa se debe analizar el problema, lo que nos va a indicar las especificaciones y requisitos que la aplicación debe cumplir. Para llevar esto acabo, se deberán realizar diferentes entrevistas entre el programador y el cliente/usuario para precisar cuales son las características que debe tener la aplicación, especificando, entre otras cosas, los procesos y estructuras que deberá tener ésta. La creación de **prototipos** será muy útil en esta fase para saber con mayor exactitud los puntos a tratar.

   Esta etapa proporcionará una idea general de lo que se solicita, realizando sucesivos refinamientos posteriormente que servirá para determinar cual es la información que ofrecerá la resolución del problema y que datos son necesarios para resolver este.

2. **Etapa de Diseño**:

   En esta tapa se convierte la especificación de la etapa de análisis en un diseño más detallado que define el comportamiento o la secuencia lógica de instrucciones capaz de resolver el problema planteado. Estos pasos sucesivos, constituyen lo que ya hemos definido como algoritmo.

   Antes de pasar a la implementación del algoritmo, tenemos que tener claro que la solución que se propone es la adecuada. Para ello, toda solución necesitará de la **prueba** o **traza** del programa. Este procedimiento consistirá en el seguimiento paso a paso de cada instrucción del algoritmo utilizando los datos correctos. Si la solución aportada contiene errores, deberemos volver a la etapa de análisis, si no, podremos pasar a la fase de implementación.

1.2.2 Implementación

Esta fase consiste en plasmar en un código la solución a la que hemos llegado en la fase de resolución del problema y consta de las siguientes etapas:

1. **Etapa de Codificación**:

   Esta etapa consiste en transformar o traducir la solución a la que hemos llegado en un lenguaje de programación concreto. Para comprobar la estabilidad y calidad de la solución, deberemos realizar una serie de pruebas para comprobar que los módulos funcionan correctamente (**pruebas unitarias**), que dichos módulos funcionan bien entre ellos (**pruebas de interconexión**) y todo el conjunto funciona correctamente (**pruebas de integración**).

   Cuando realizamos la traducción deberemos tener en cuenta la reglas gramaticales y sintácticas del lenguaje de programación que estemos usando, lo que generará lo que se conoce como **código fuente**, es decir, el programa en sí mismo. Pero para que nuestro programa funcione correctamente, antes deberá ser traducido a **código máquina**, el único lenguaje que entiende el ordenador. Este proceso de traducción será llevado a cabo por un **compilador** o un **interprete**, dependiendo del lenguaje de programación que estemos empleando.

   Algunos de los términos que están asociados a esta fase son los siguientes:

   - **Compilación**: proceso mediante el cual el código fuente, es decir, el conjunto de instrucciones escritas en un determinado lenguaje de programación, es traducido a código máquina, un código binario que es el único lenguaje que entiende el computador.
   - **Compilador**: es la aplicación informática que se encarga de realizar la traducción. Esta recibe el código fuente, y tras realizar una análisis lexicográfico, semántico y sintáctico, generando en primer lugar un código intermedio sin optimizar, el cual se optimiza y genera el código objeto para la plataforma especifica.
   - **Interprete**: aplicación informática capaz de analizar y traducir programas escritos en un determinado lenguaje de programación. A diferencia de los compiladores, que realizan una traducción completa del programa a lenguaje máquina, los interpretas van realizando la traducción a medida que va siendo necesaria, típicamente, realizándola instrucción por instrucción. Además, no suelen guardar los resultados de dicha traducción.

2. **Prueba, Ejecución y Validación**:

   Una vez que el programa ha sido codificado, estará listo para ser ejecutado. Para ello, deberemos implantar la aplicación en el sistema donde va a ser ejecutado para comprobar su funcionamiento correcto. Utilizando diferentes datos, se comprobará si el programa cumple con los requisitos que se han especificado, si se detectan nuevos errores o si la interfaz es amigable. Se trata de poner a prueba nuestro programa para ver su respuesta en diferentes situaciones.

   Mientas que el programa tenga errores, no podremos pasar a la siguiente fase. Una vez que se corrijan, se habrán generado diferentes documentos sobre la aplicación que entrarán el alguna de las siguientes dos categorías:

   - **Documentación Interna**: encabezados, descripciones, declaraciones del problema y comentarios que se incluyen dentro de código fuente.
   - **Documentación Externa**: son los manuales que se crean para una mejor ejecución y utilización del programa, así como los diferentes diagramas generados durante el proceso de desarrollo que nos ayuden a comprender mejor su funcionamiento y arquitectura.

1.2.3 Explotación

En esta última fase, el software ya esta instalado en el sistema y esta siendo de utilidad para los usuarios, siendo esta la **etapa de explotación**.

Periódicamente será necesario realizar evaluaciones y, si es necesario, llevar a cabo modificaciones para que el programa se adapte o actualice a las nuevas necesidades de los usuarios, pudiendo también realizarse la corrección de errores no detectados anteriormente. Esta es la **etapa de mantenimiento**.

Así, se define el **mantenimiento de software** como el proceso de mejora y optimización del software después de su entrega al usuario final. Involucra la realización de cambios para corregir defectos y dependencias encontradas durante su uso o para la adición de funcionalidades para mejorar su usabilidad y aplicabilidad.

Sera indispensable que se haya generado la documentación adecuada para facilitar la labor del programador en la compresión, uso y modificación de la aplicación.

1.3 Ciclo de Vida del Software

Sean cuales sean las fases que las fases en las que realicemos el proceso de desarrollo de software, siempre habrá que aplicar un **modelo de ciclo vida**, siendo este una sucesión de estados o fases por las cuales pasa el software a lo largo de su vida.

Este proceso debe tener siempre la siguientes etapas mínimas:

- **Especificación** y **Análisis de Requisitos**
- **Diseño**
- **Codificación**
- **Pruebas**
- **Instalación** y **Producción**
- **Mantenimiento**

Existen diferentes modelos de desarrollo como el modelo en cascada, en espiral, incremental, evolutivo, etc.. En la siguiente entrada de Wikipedia podemos ver más información sobre el proceso de desarrollo de software y las diferentes metodologías.

1.4 Lenguajes de Programación

Como hemos visto en los puntos anteriores, una de las etapas del desarrollo es la codificación del programa. Para ello, emplearemos un lenguaje que exprese cada uno de los pasos que se deben ejecutar. Este lenguaje recibe el nombre de **lenguaje de programación**.

Se entiende por **lenguaje de programación** el conjunto de reglas sintácticas y semánticas, símbolos y palabras especiales establecidos para la construcción del programa. Todo lenguaje se compone de:

- **Gramática**: reglas aplicables al conjunto de símbolos y palabras empleados en el lenguaje de programación para la construcción de sentencias correctas.
- **Léxico**: es el conjunto de símbolos y palabras especiales, es decir, el vocabulario del lenguaje.
- **Sintaxis**: son las posibles combinaciones de símbolos y palabras especiales.

- **Semántica**: es el significado de cada construcción del lenguaje, es decir, la acción que se llevará a cabo.

Los lenguajes permiten que los programadores pueden expresar el código de forma que sea legible para ellos y otros programadores. Estos se clasifican según lo cercanos que estén al lenguaje máquina o al lenguaje natural, así como si son lenguajes interpretados o compilados, lo que vamos a ver en los siguientes puntos.

1.4.1 Lenguaje Máquina

Este lenguajes es él único que entiende el ordenador y se compone de un conjunto de instrucciones codificadas en binario que se encarga de ejecutar el procesador. Este lenguaje es directamente interpretable por un circuito microprogramable.

Se trata del primer lenguaje utilizado para la programación de computadoras. De hecho, cada máquina tenía su propio conjunto de instrucciones codificadas en ceros y uno. Esto, evidentemente, genera un conjunto de **inconvenientes**:

- Cada programa era válido **solo para un tipo de procesador** u ordenador.
- La **lectura** e **interpretación** de este tipo de programas es **extremadamente difícil**, por lo que realizar modificaciones resultaba extremadamente costoso.
- Los programadores de la época debían **memorizar largas cadenas de unos y ceros**, que equivalían a las instrucciones disponibles en los diferentes tipos de procesadores.
- Además, la introducción de estas cadenas suponía **largos tiempos de espera y posibles errores**.

En la siguiente tabla muestran algunos ejemplos de operaciones codificadas en binario.

Operación	Lenguaje máquina
SUMAR	00101101
RESTAR	00010011
MOVER	00111010

Figura 1.4.1: Operaciones en binario

Dada la dificultad de este lenguaje, con el tiempo fue sustituido por otros de más fácil compresión, aunque hay que tener en cuenta que en última instancia todos los lenguajes deben ser traducido a éste para que puedan ser interpretados y ejecutados por el ordenador.

1.4.2 Lenguaje Ensamblador

La evolución del lenguaje máquina fue el **lenguaje ensamblador**. En éste, las instrucciones ya no son secuencias binarias sino que son códigos de operación que describen operaciones básicas del procesador.

Estos códigos, conocidos como **mnemotécnicos**, son palabras especiales que sustituyen largas secuencias de unos y ceros, utilizadas para referirse a diferentes operaciones disponibles en el juego de instrucciones que soporta un procesador concreto.

En la siguiente tabla, podemos ver alguno ejemplos de estos mnemotécnicos con diferentes operaciones.

Operación	Lenguaje Ensamblador
MULTIPLICAR	MUL
DIVIDIR	DIV
MOVER	MOV

Figura 1.4.2: Operaciones en ensamblador

Aunque el ensamblador fue un intento de acercar el lenguaje máquina al lenguaje humano, aún presentaba diferentes **dificultades**:

- Los programas seguían **dependiendo** directamente del **hardware** que los soportaba.
- Los programadores debían **conocer** detalladamente **la máquina** sobre la que programaban, ya que debían hacer un uso adecuado de los recursos del sistema.
- La **lectura**, **interpretación** o **modificación** de los programas seguía presentando dificultades.

Todo programa escrito en ensamblador necesita ser traducido al lenguaje máquina para poder ser ejecutad. Esta función la lleva a cabo el **programa ensamblador**, el cual convierte el programa original escrito en lenguaje ensamblador (código fuente) en el programa traducido a lenguaje máquina (código objeto).

1.4.3 Lenguajes Compilados

Para paliar los defectos del lenguaje ensamblador y acercar los lenguajes de programación al lenguaje humano nacieron los **lenguajes compilados**.

Así surgieron lenguajes como **Pascal**, **FORTRAN**, **Algol**, **C**, **C++**, etc. Estos lenguajes, más cercanos al humano, también se denominan **lenguajes de alto nivel**. Son más fáciles de utilizar y comprender, ya que las instrucciones que emplean y los signos que usan son fácilmente reconocibles por el programador.

Entre las principales **ventajas** de estos lenguajes tenemos:

- Son mucho más **fáciles** de **aprender y utilizar** que sus predecesores.
- Se **reduce** el **tiempo** para desarrollar programas así como el **coste**.
- Son **independientes del hardware**, es decir, los programas pueden ejecutarse en diferentes tipos de máquina.
- La **lectura**, **modificación** e **interpretación** de los programas es **mucho más sencilla**.

Un programa que este escrito en un lenguaje de alto nivel también tiene que **traducirse** a código máquina para que pueda ser ejecutado por el ordenador. Este trabajo, lo lleva a cabo el **compilador**, mediante un proceso de traducción que se conoce como **compilación**.

En la siguiente imagen podemos ver de forma mas ilustrativa cual es el proceso de compilación.

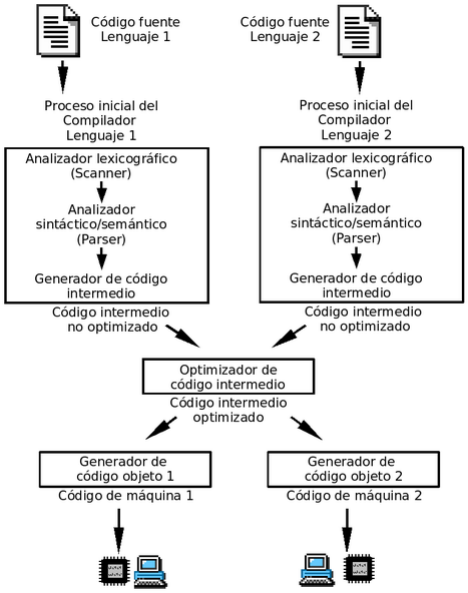

Figura 1.4.3: Proceso de compilación

El compilador realizará la traducción y ademas **informará** de posibles **errores**. Una vez subsanados se generará el programa traducido a código máquina, conocido como **código objeto**. Este programa aún no puede ser ejecutado hasta que no se añadan los módulos de enlace o bibliotecas, durante el proceso de enlazado. Una vez realizado, se obtiene el **código ejecutable**.

1.4.4 Lenguaje Interpretados

Los lenguajes interpretados están diseñados para que su ejecución se realice a través de un **interprete**. Cada instrucción de un lenguaje interpretado se analiza, traduce y ejecuta tras haber sido verificada.

Un **interprete** es un traductor de lenguaje de alto nivel donde el proceso de traducción y ejecución se llevan a cabo simultáneamente, es decir, la instrucción se pasa al lenguaje máquina y se ejecuta directamente. No se genera código objeto ni ejecutable, como en los lenguajes compilados.

Los lenguajes interpretados tienen el inconveniente que son un poco más lentos que los lenguajes compilados, además, se necesita tener el intérprete instalado en la máquina donde se quiere ejecutar el programa, algo que no es necesario con los lenguajes compilados.

Algunos ejemplos de lenguajes interpretados son **Perl**, **PHP**, **Python** o **Javascript**, entre otros.

A medio camino entre los lenguajes compilados y los interpretados existen los lenguajes que podríamos llamar **pseudo-compilados** o **pseudo-interpretados**, como es el caso de **Java**. Java puede verse como un lenguaje compilado, pero también como un lenguaje interpretado, ya que su código fuente se compila para obtener el código binario conocido como **bytecode**, que son estructuras parecidas a las instrucciones máquina, con la importante propiedad de no se dependiente de ningún tipo de máquina.

La **Maquina Virtual Java** se encargará de interpretar este código, y para su ejecución, lo traducirá al código máquina del procesador en particular donde se quiera ejecutar la aplicación.

1.5 El Lenguaje de Programación Java

Java es un lenguaje de programación con un sintaxis similar a C++, donde se han eliminado algunos de sus elementos complicados, como el tratamiento de punteros. Es un lenguaje **orientado a objetos** que nos permite la utilización de gran cantidad de bibliotecas fomentando la reutilización de código y evitando que tengamos que reescribir código ya existente.

Una de las principales virtudes de Java es su **independencia del hardware**, ya que el código que se genera es válido para cualquier plataforma. Este código será ejecutado en una maquina virtual denominada **Máquina Virtual Java** (JVM), que interpretará el código generado traduciéndolo al código máquina de la plataforma donde queramos ejecutarlo. De este modo, el programa se escribe una única vez y puede hacerse funcionar en cualquier lugar, llevando a la práctica el lema del lenguaje **"Write once, run everywhere"**.

Antes de la aparición de Java, el lenguaje de programación C era uno de los más versátiles y extendidos, pero cuando los programas en C aumentaban de volumen, su manejo se volvía muy complejo. Aunque mediante la programación modular y estructura se conseguía reducir esta complejidad, no era suficiente.

Fue entonces cuando entró en escena la **Programación Orientada a Objetos** (POO), aproximando más la construcción al pensamiento humano y haciendo más sencillo todo el proceso. Los problemas se dividen en objetos que tienen propiedades e interactúan con otros objetos, pudiendo el programador centrarse en cada objeto y los detalles las propiedades y funcionalidades que este posee. Fue entonces cuando surgió el lenguaje Java.

Las principales **características** del **lenguaje Java** son:

- El código generado por el compilador es independiente del hardware.
- Es totalmente orientado a objetos.
- Su sintaxis es similar a C y C++.
- Es distribuido, es decir, preparado para aplicaciones TCP/IP.
- Dispone de un amplio conjunto de bibliotecas.
- Es robusto, realizando comprobaciones tanto en tiempo de compilación como de ejecución.
- La seguridad está garantizada, ya que las aplicaciones Java no acceden directamente a zonas delicadas de la memoria o el sistema.

1.5.1 Breve Historia de Java

Java surgió en 1991 cuando un pequeño grupo de ingenieros de Sun Microsystems trataron de diseñar un lenguaje de programación destinado a programar pequeños dispositivos electrónicos. El problema con estos dispositivos es que cambian rápidamente de un modelo a otro y el software debe ser reescrito, por lo que necesitaban un lenguaje que se fuera **independiente del dispositivo**.

No fue hasta 1995 cuando el lenguaje adoptó el nombre de Java, dándose a conocer al mundo como lenguaje de programación de computadores. El hecho de que sea un lenguaje orientado a objetos, independiente de la plataforma y su facilidad para la creación de aplicaciones TCP/IP, han hecho que Java sea uno de los lenguajes más utilizados.

El factor determinante para su expansión fue la inclusión de un interprete en la versión 2.0 del navegador Netscape, lo que supuso un gran revuelo en internet. A principio de 1997 apareció **Java 1.1**, que proporcionó sustanciales mejoras en el lenguaje. A finales de 1998 salió **Java 1.2**, posteriormente rebautizado como **Java 2**.

Para el desarrollo de programas en Java es necesario el uso de un entorno de desarrollo denominado **JDK** (Java Development Kit), que provee de un compilador y un entorno de ejecución, conocido como **JRE** (Java Runtime Environment), para ejecutar los bytecode generados. Al igual que el lenguaje, JDK y JRE han sido mejorados con cada versión del lenguaje.

Java 2 es la tercera versión del lenguaje, pero no solo incluye el lenguaje, sino incluye:

- El lenguaje de programación: Java.
- Un conjunto de bibliotecas estándar que vienen incluidas en el lenguaje y son necesarias en todo el entorno Java. Es el **Java Core**.
- Un conjunto de herramientas para el desarrollo de programas como compilador de bytecode, el generador de documentación, un depurador, etc...
- Un entorno de ejecución que en definitiva es una máquina virtual para ejecutar los bytecodes generados.

En la siguiente imagen podemos ver un esquema de los elementos de Java 2.

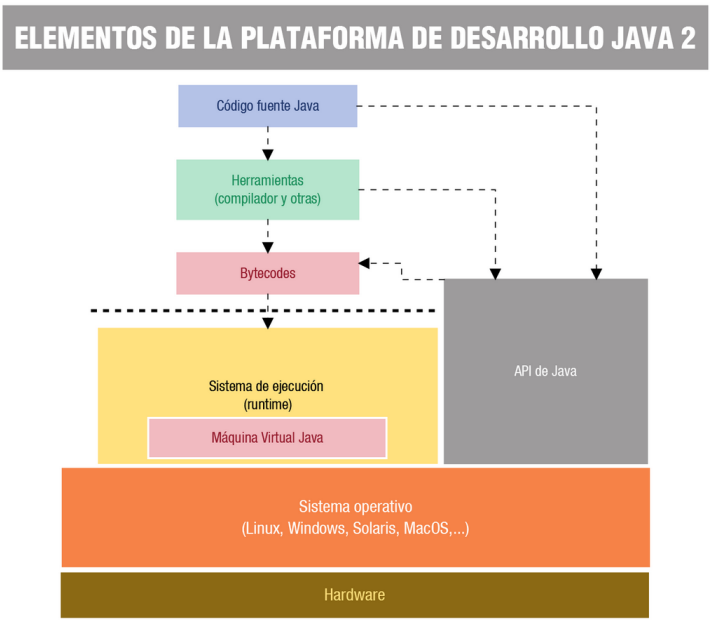

Figura 1.5.1: Elementos de Java 2

Actualmente hay varias ediciones de la plataforma Java y que de forma resumida se podrían clasificar en:

- **Java SE**: es la plataforma base para el desarrollo de aplicaciones con Java. Es usado para desarrollar aplicaciones de escritorio, applets y otros tipos de aplicaciones. Es fundamental, ya que es la basa en la que se cimienta el resto de plataformas Java.
- **Java EE**: es una plataforma de desarrollo de aplicaciones empresariales y del lado del servidor.

- **Java ME**: es una plataforma de desarrollo de aplicaciones para dispositivos móviles.

En este enlace puedes encontrar más información sobre las diferentes plataformas de Java y como trabajan entre ellas.

1.5.2 La POO y Java

En Java, los datos y el código (funciones o métodos) se combinan en entidades llamadas **objetos**. El objeto tendrá un **comportamiento** (su código interno) y un **estado** (los datos). Los objetos permiten la reutilización de código y pueden considerarse a sí mismos con piezas reutilizables en diferentes proyectos. Esta característica permite reducir el tiempo de desarrollo de aplicaciones.

Al incorporar los conceptos de Programación Orientada a Objetos (POO), Java incluye las tres características propias de este paradigma de programación:

- **Encapsulación**: se denomina así al ocultamiento de la información interna de un objeto. Este solo se puede modificar a través de sus operaciones definidas.
- **Herencia**: mecanismo que permite derivar una clase de otra, de manera que extienda su funcionalidad.
- **Polimorfismo**: capacidad para que varias clases derivada de otra utilicen un mismo método de forma diferentes.

Los patrones o tipos de objetos se denominan **clases** y los objetos que utilizan estos patrones o pertenecen a dichos tipos se denominan **instancias**.

En la siguiente figura podemos ver algunas de las ventajas de la POO de forma más esquemática.

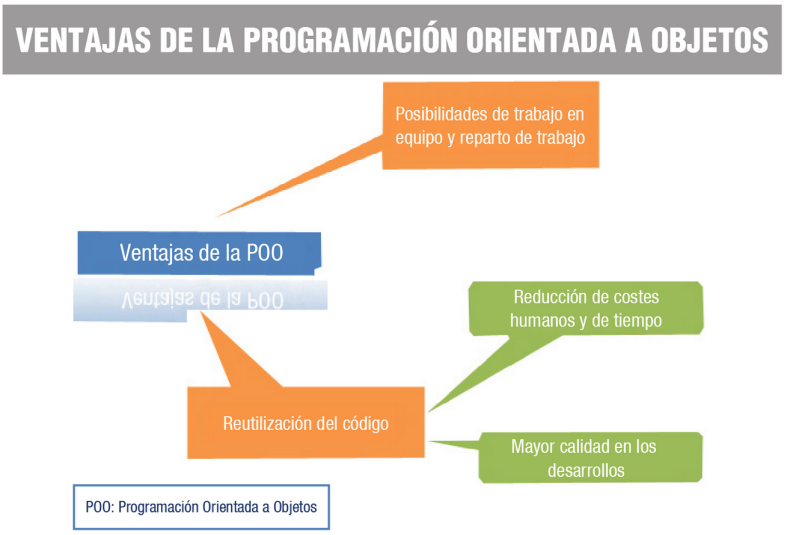

Figura 1.5.2: Ventajas de la POO

1.5.3 Independencia de la Plataforma

Hay dos características de distinguen a Java de otros lenguajes de programación, como son la **independencia de la plataforma** y la posibilidad de **crear aplicaciones** para **trabajar en red**.

Estas características consisten en:

- **Independencia**: los programas escritos en Java pueden ser ejecutados en cualquier tipo de hardware. El código fuente se compila generando el código conocido como Java Bytecode, el cual será interpretado y ejecutado por la **Máquina Virtual Java**, que es un programa escrito en código nativo de la plataforma destino. Con eso se evita tener que realizar un programada diferente para diferentes CPUs.
- **Trabajo en red**: esta capacidad del lenguaje ofrece múltiples posibilidades para la comunicación vía TCP/IP. Para poder hacer esto, existen librerías que permiten el acceso y la interacción con protocolos como **http**, **ftp**, etc.., facilitando las tareas del programador para el tratamiento de la información.

1.5.4 Seguridad y Simplicidad

Además de las características vistas en el punto anterior, cabe destacar dos virtudes de este lenguaje que hacen que este tan extendido: la **seguridad** y la **simplicidad**.

- **Seguridad**: en primer lugar, el acceso a zonas sensibles de memoria que podemos tener en otros lenguajes como C o C++ se han eliminado en Java.

  En segundo lugar, el código en Java se comprueba y verifica para evitar que determinadas secciones del código produzcan efectos adversos. Los test que se aplican garantizan que las operaciones, operandos, conversiones y uso de clases son seguras.

  En definitiva, podemos afirmar que Java es un lenguaje seguro.

- **Simplicidad**: aunque Java es tan potente como C o C++, es bastante más sencillo. Tiene una curva de aprendizaje muy rápida y para alguien que aprende a programar con este lenguaje, le resultará más fácil comenzar a escribir aplicaciones interesantes.

  Java elimina ciertos mecanismos complejos que si encontramos en C o C++ como pueden ser la aritmética de punteros, los registros, la definición de tipos, la gestión de memoria, etc.., reduciendo considerablemente la posibilidad de cometer errores comunes en otros lenguajes de programación.

  Un elemento que ayuda a la simplicidad de Java es el **Recolector de Basura** (Garbage Collector), que permite al programador liberarse de la gestión de memoria y hace que ciertos bloques de memoria puedan reaprovecharse, disminuyendo el número de huevos libres, lo que se conoce como **fragmentación de memoria**.

Como vemos, además de ser independiente de la plataforma, Java es un lenguaje más seguro y simple que otros parecidos, manteniendo la misma potencia que estos.

1.5.5 Java y los Bytecodes

Un programa en Java no es directamente ejecutable, es necesario que el código sea interpretado por la Máquina Virtual.

Una vez escrito el código fuente (con extensión .java), este es precompilado generándose los códigos de bytes o Bytecodes (con extensión .class), que serán interpretados directamente por la Máquina Virtual y traducidos a código nativo de la plataforma donde queramos ejecutarlo.

Un **Bytecode** es un conjunto de instrucciones en lenguaje máquina que no son específicos de ningún procesador o sistema de cómputo. Un intérprete de bytes para una plataforma concreta será el que los ejecute. A este interprete también se les conoce como Máquina Virtual Java.

En el proceso de compilación, existe un verificador de códigos de bytes que se asegurará que se cumplen las siguientes condiciones:

- El código satisface las especificaciones de la Máquina Virtual Java.
- No existe amenaza contra la integridad del sistema.
- No se produce desbordamiento de memoria.
- Los parámetros y sus tipos son adecuados.
- No existen conversiones de datos no permitidas.

Para que un bytecode puede ser ejecutado en cualquier plataforma es imprescindible que la plataforma cuente con el intérprete adecuado, es decir, la **máquina virtual** específica para **dicha plataforma**.

1.6 Programas en Java

Hasta ahora hemos descrito el lenguaje de programación Java y hablado un poco sobre su historia y características. En este punto, ya vamos a empezar ha hablar de los programas en Java, cuales son su elementos básicos, como debemos escribir el código y los tipos de aplicaciones que podremos crear con este lenguaje.

1.6.1 Estructura de un Programa

En la siguiente figura, se presenta una estructura general de un programa en Java, la cual vamos a explicar en esta sección punto por punto.

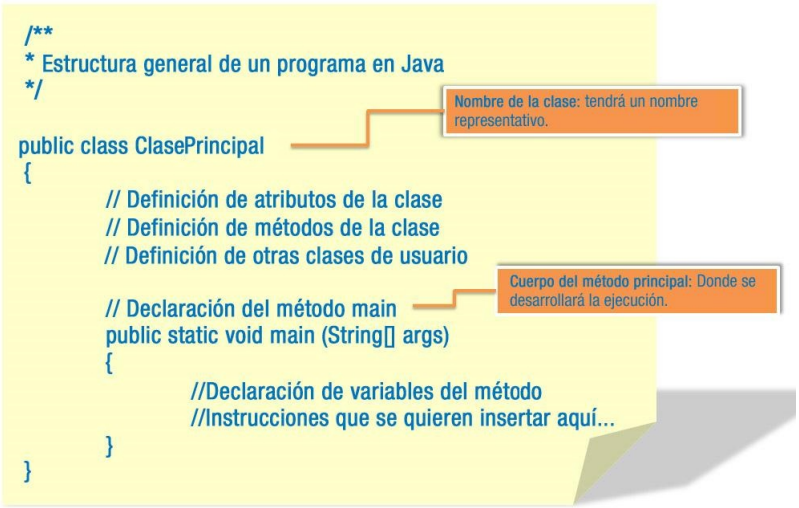

Figura 1.6.1: Estructura general de un programa en Java

Los diferentes elementos que podemos ver en esta figura son los siguientes:

- **public class ClasePrincipal**: todos los programas en Java deben incluir este elemento. Podrá llamarse ClasePrincipal, ProgramaPrincipal, ... o como queramos, pero debe tener un nombre. Se trata de una clase general donde se incluyen todos los demás elementos del programa. En unidades posteriores veremos que es una clase y cuales son sus componentes principales. Por ahora es suficiente que sepamos que nuestro programa debe comenzar con este elemento.

- **public static void main (String args[])**: dentro del elemento anterior podemos ver el método **main()** que contiene las líneas de código de nuestro programa. Más adelante también veremos que es un método. Basta saber por ahora que al igual que la línea anterior nuestro programa debe contener la sentencia **public static void main (String args[])**. Aquí dentro podremos incluir las instrucciones que veamos oportunas para ejecución del programa.
- **Comentarios**: los comentarios suelen introducirse en el programa para realizar aclaraciones, anotaciones o cualquier otra indicación que el programador estime oportuna. Estos comentarios pueden introducirse de dos formas:
  - Con **//** estaríamos estableciendo una línea de comentario, es decir, todo lo que hay detrás de este símbolo es un comentario hasta que se produzca un salto de línea.
  - Si queremos indicar que un comentario tiene varias líneas debemos usar **/*** para comenzar el comentario y ***/** para cerrarlo.
- **Bloques de Código** ({ }): con conjuntos de instrucciones que se marcan mediante la apertura y cierre de llaves **{ }**. En código así marcado se considera interno al bloque.
- **Punto y Coma** (**;**): aunque en el ejemplo de la imagen no tenemos ninguna línea de código acabada con un punto y coma, para no distraernos de momento con estos detalles, hay que hacer hincapié en que cada línea de código ha de terminar con un punto y coma. En caso de no hacerlo, tendremos errores sintácticos.

1.6.2 El Entorno Básico de Desarrollo de Java

La herramienta básica para comenzar a desarrollar aplicaciones en Java es el **JDK** (**Java Development Kit**), que incluye un compilador y un interprete para la línea de comandos. Estos dos programas son los empleados en la compilación y la interpretación del código.

Como veremos, existen diferentes entornos para el desarrollo de aplicaciones en Java que incluyen una gran variedad de herramientas, pero por ahora nos centraremos en el entorno básico, extendido y gratuito, el Java Development Kit. JDK es un entorno para construir aplicaciones, applets y componentes utilizando el lenguaje de programación Java, incluyendo todas las herramientas necesarias para su construcción y ejecución.

Junto con JDK se incluye una implementación del entorno de ejecución Java, conocido como **Java Runtime Environment** (**JRE**), para ser utilizado por JDK. El JRE incluye la **Java Virtual Machine** (**JVM**), bibliotecas de clases y otros ficheros que soportan la ejecución de programas escritos en Java.

Java fue creado por Sun Microsystems, pero posteriormente fue absorbida por Oracle, los que no han dejado de lanzar versiones de JDK. Con el lanzamiento de Java 11, Oracle hizo un cambio de licencia de modo que se convirtió en tecnología de pago en caso de usarlo en ciertas circunstancias. Podemos usar JDK u otras implementaciones abiertas. En nuestro caso, usaremos **OpenJDK**.

OpenJDK es la versión libre de la plataforma de desarrollo Java, en concreto de su versión **Java SE** (Standard Edition). Esta bajo la licencia **GPL 2.0** y desde la versión 7 de Java SE, OpenJDK es la versión de referencia. [1]

En este enlace podemos ver una guía de como instalar OpenJDK en las principales distribuciones.

1.6.3 La API de Java

Dentro del kit de desarrollo de Java que acabos de ver se incluyen gratuitamente todas las bibliotecas de Java, los que se conoce como la **API** (Advanced Programing Interface) de Java, lo que se conoce

como **Biblioteca de Clases Java**. Este conjunto de bibliotecas proporcionar al programador paquetes de clases útiles para la realización de diferentes tareas dentro de un programa. Las bibliotecas están organizadas en paquetes donde cada paquete tiene un conjunto de clases relacionadas semánticamente.

En décadas pasadas una biblioteca era un conjunto de programas que contenía cientos de rutinas. Estas rutinas manejaban las tareas de todos o casi todos los programas que necesitaban. El programador podía recurrir a estar bibliotecas para para desarrollar programas con rapidez.

Una biblioteca de clases es un conjunto de clases en programación orientada a objetos. Estas clases contienen métodos que son útiles para los programadores realizando diferentes tareas. En el caso de Java, cuando descargamos el JDK obtenemos la biblioteca de clases API. Utilizar las clases y métodos de la API de Java acelera el proceso de desarrollo de los programas. También, existen diferentes bibliotecas de clases desarrolladas por terceros que contienen componentes reutilizables de software y están disponibles a través de la web.

En la página oficial de Oracle podemos encontrar información más detallada sobre la API de Java.

1.6.4 Afinando la Configuración

Para que podamos compilar y ejecutar ficheros Java es necesario que realicemos unos pequeños ajustes en la configuración del sistema. Vamos a indicarle donde puede encontrar los ficheros necesarios para realizar la labores de compilación y ejecución, es esta caso los ficheros **javac.exe** y **java.exe**, así como las librerías incluidas en la API de Java y las clases de usuario. Esto lo haremos definiendo o editando las siguientes variables del sistema.

- **Variable PATH**: como aún no disponemos de un IDE (Integrated Development Environment), la única forma de ejecutar programas es a través de la línea de comandos. Pero sólo podremos ejecutar programas directamente si la ruta hacia ellos esta indicada en la variable de sistema PATH. Es necesario que incluyamos la ruta hacia estos programas en nuestra variable PATH, incluyendo la ruta donde se instaló JDK hasta su directorio **bin**.
- **Variable CLASSPATH**: esta variable de entorno establece donde encontrar la librerías y clases de la API de Java, así como las clases creadas por el usuario. Es decir, los ficheros **.class** que se obtienen una vez compilado el código fuente de un programa escrito en Java. Es posible que en dicha ruta existan ficheros comprimidos en los formatos **zip** o **jar** que pueden ser utilizados directamente por JDK, conteniendo en su interior los archivos **class**.

1.6.5 Codificación, Compilación y Ejecución de Aplicaciones

Una vez que la configuración del entorno de Java y tenemos el código fuente de nuestro programa escrito en un archivo con extensión **.java**, la compilación de aplicaciones se realiza mediante el programa **javac** incluido en JDK.

Para llevar a cabo esta compilación deberemos usar el comando ***javac archivo.java***, desde la línea de comandos, donde archivo.java es el archivo con nuestro código fuente.

El resultado de la compilación sera un nuevo archivo con el mismo nombre que el nuestro pero con una extensión **.class**. Este es el archivo con el código a bytecode, es decir, con el código precompilado. Si en el código fuente de nuestro programa figuraran más de una clase, veremos como al realizar la compilación se generarán tantos archivos .class como clases tengamos. Además, si esta clases tenía el método **main()** podremos ejecutarlas por separado para ver el funcionamiento de dichas clases.

Para que el programa pueda ser ejecutado, siempre y cuando este incluido en el interior el método **main()**, podemos usar el interprete incluido en JDK. Para ello, usaremos el comando ***java archivo***.

Donde **java** es el interprete y **archivo** es el archivo con el código precompilado a bytecode, es decir, el archivo con extensión **.class**. Hay que destacar que no hay que incluir la extensión del archivo .class al llamar al interprete, solo debemos poner el nombre del archivo.

1.6.5.1 Estandarización del Código

Cada vez que escribamos código es importante que sigamos unas normas o estándares que nos ayuden a realizar este proceso siempre de la misma manera. De este modo todos los programas tendrán una estructura similar y nos resultará mucho más sencillo localizar los componentes en cada uno.

En caso del lenguaje Java ya hemos visto la estructura básica de un programa. A partir de aquí, vamos a desarrollar una plantilla que nos ayudará a desarrollar nuestro programas sin tener que reescribir las partes comunes una y otra vez.

Nuestra plantilla podría tener la siguiente **estructura**:

1. **Declaración de la clase principal**: aquí podríamos poner el **nombre de nuestro programa**. Es decir, en lugar de llamar a la clase **ClasePrincipal**, podríamos llamar **Concurso**, **Juego**, **CalculoDeAreas**, etc. Esto es, algún nombre que nos de una pista sobre el funcionamiento de nuestro programa.
2. **Método main**: dentro del componente anterior y encerrado entre llaves. Este nombre no se puede cambiar. En su interior estarán las líneas de código de nuestro programa:
    a) **Declaración de constantes y variables**:
        1) Declaración de **constantes**
        2) Declaración de **variables de entrada**
        3) Declaración de **variables de salida**
        4) Declaración de **variables auxiliares**
    b) **Cuerpo del Programa**
        1) **Entrada de datos**
        2) **Procedimientos**
        3) **Salida de datos**

Si seguimos esta estructura, será muy fácil analizar nuestro programas pues estarán divididos en distintos elementos con significado propio cada uno de ellos. Ahora bien, tampoco hay que ser excesivamente rígidos. Dependiendo de la naturaleza y al complejidad del programa es posible que alguna vez estas tres partes se fundan en una, especialmente cuando el programa sea interactivo y se sigan introduciendo datos a la vez que se van procesando e incluso devolviendo resultados.

1.6.5.2 Problemas con Caracteres Acentuados

A veces cuando intentamos ejecutar un programa Java y este tiene caracteres acentuados podemos obtener símbolos extraños, en vez de dichos caracteres, especialmente si estamos trabajando con la consola de comandos.

Una solución simple que suele funcionar en la mayoría de los casos es compilar el archivo de código fuente con normalidad, indicando cuando se vaya a realizar su ejecución la opción ***-Dfile.encoding*** con la codificación que queremos emplear, es decir:

```
java -Dfile.encoding=cp850 PROG_programa1
```

En este ejemplo, la página de códigos que usamos es la 850, pero se puede forzar al interprete a usar cualquier codificación que queramos, siempre y cuando este la soporte. Para saber que codificación usa nuestro sistema, podemos usar el comando ***chcp*** desde la consola, que nos mostrará dicha codificación.

1.6.6 Tipos de Aplicaciones en Java

La versatilidad del lenguaje de programación Java permite al programador crear distintos tipos de aplicaciones, los cuales listamos a continuación:

- **Aplicaciones de Consola**
  - Son programas independientes al igual que los creados con otros lenguajes tradicionales.
  - Se componen como mínimo de un archivo **.class** que debe contener necesariamente el método **main()**.
  - No necesitan un navegador y se ejecutan cuando invocamos el comando **java**. Si no se encuentra el método **main()** la aplicación no podrá ejecutarse.
  - Estas aplicaciones leen y escriben hacia y desde la entrada y salida estándar, sin ninguna interfaz de usuario.
- **Aplicaciones Gráficas**
  - Aquellas que utilizan clases con capacidades gráficas, como **Swing**, que es la biblioteca para la interfaz gráfica de la API de Java.
  - Incluyen instrucciones de tipo **import**, que indican al compilador de Java que clases del paquete **javax.swing** se incluyan en la compilación.
- **Applets**
  - Son programas incrustados en otras aplicaciones, normalmente páginas web que se muestran en el navegador. Cuando el navegador carga una web que contienen applets, estos se descargan en el navegador y comienza su ejecución. Esto nos permite crear programas que cualquier usuario con un navegador web puede utilizar.
  - Los applets se descargan junto con una página HTML y se ejecutan en la máquina del cliente.
  - No tienen acceso a partes sensibles a menos que uno le de los permisos oportunos.
  - No tienen un método principal.
  - Son multiplataforma y pueden ejecutarse en cualquier navegador que soporte Java.

  Servlets
  - Son componente de la parte del servidor de Java EE, encargados de generar respuestas a las peticiones realizadas por los usuarios.
  - Al contrario que los applets, están pensados para trabajar en el lado del servidor y procesar las peticiones de los clientes.
- **Midlets**

- Son aplicaciones Java creadas para su ejecución en sistemas de propósito simple o móviles.
- Se usan en dispositivos embebidos, mas específicamente para la máquina virtual Java Micron Editio (Java ME).
- Generalmente son juegos y aplicaciones que se ejecutan en teléfonos móviles.

Como vemos, la variedad de aplicaciones que podemos crear con Java es enorme, ya dependerá de lo que estemos interesados en crear, usaremos unas u otras opciones y librerías.

1.7 Entornos Integrados de Desarrollo

En los comienzos de Java la utilización de la línea de comandos era algo habitual. El programador escribía el código utilizando un editor de textos básico y a continuación utilizaba el compilador para obtener el código compilado. En un paso posterior necesitaba usar una herramienta para ensamblar el programa. Por último, podía probar el programa desde la línea de comandos. El problema surgía cuando ocurría algún error y había que iniciar el proceso de nuevo.

Esto hacía que el proceso de desarrollo no estuviera optimizado. Con el paso del tiempo se fueron desarrollando aplicaciones que incluían todas las herramientas necesarias para que el proceso de desarrollo fuera más rápido, sencillo y fiable. Para cada lenguaje de programación existen múltiples entornos de desarrollo, cada uno con sus ventajas e inconvenientes. Dependiendo de las necesidades y gustos del programador, se elegirá uno u otro.

Para el lenguaje de programación Java existen diferentes alternativas, siendo los principales entornos de desarrollo **Netbeans**, desarrollado por Sun y **Eclipse**. Ambos gratuitos, con soporte de idiomas y multiplataforma (Windows, Linux y MacOS).

En nuestro caso, el entorno que vamos a usar durante nuestros desarrollos va a ser **Netbeans**, ya que es de código abierto y además ha sido desarrollado por la misma compañía que desarrollo el lenguaje Java.

1.7.1 ¿Que es un IDE?

Los **IDE** (Integrated Development Environment) son aplicaciones que permiten llevar todo el proceso de desarrollo de software a través de una misma aplicación. Podremos realizar labores de edición, compilación, depuración, detección de errores, corrección y ejecución de programas escritos en Java o en otros lenguajes de programación bajo un entorno gráfico. Junto a estas características, cada entorno añade otras que ayudan a realizar el proceso de programación, como por ejemplo: resaltado de sintaxis, plantillas de diferentes tipos de aplicaciones, creación de proyectos, etc...

Hay que tener en cuenta que un IDE es solo una fachada para el proceso de desarrollo, por lo que tendremos que tener instalados compiladores, interpretes y demás para su correcto funcionamiento.

1.7.2 IDE Actuales

Existen una gran variedad de IDEs en el mercado para el lenguaje de programación Java, orientados a principiantes, para profesionales, gratuitos, de pago, libres, propietarios, etc. A continuación damos una lista de los principales.

- IDEs Gratuitos y de Libre Distribución:
  - **Netbeans**
  - **Eclipse**

    - **BlueJ**
    - **jGRASP**
    - **JCreator LE**
- IDEs Propietarios:
    - **IntelliJ IDEA**
    - **JDeveloper**

Actualmente, los más utilizados por la comunidad son **Netbeans**, **Eclipse** y **IntelliJ IDEA**. En los siguiente epígrafes vamos a ver las características de **Netbeans**, aunque si quieres ver una comparativa más exhaustiva sobre los diferentes IDE puede consultar esta entrada en Wikipedia.

1.7.3 El Entorno Netbeans

Como hemos comentado anteriormente, nosotros vamos a usar **Netbeans** como entorno de desarrollo, así que lo primero que vamos a hacer es estudiar sus características y ver que puede aportar al proceso de desarrollo.

Se trata de un entorno de desarrollo **orientado** principalmente al **lenguaje de programación Java**, aunque también soporta otros lenguajes. Es un entorno libre y gratuito sin restricciones de uso, siendo un **proyecto de código abierto** con una gran comunidad de usuario en continuo crecimiento y apoyado por varias empresas.

Netbeans lleva un tiempo pugnando con Eclipse por convertirse en la plataforma mas importante de desarrollo de aplicaciones Java y hasta el nombre, Eclipse, es una declaración de intenciones por hacerle la competencia a Oracle, la empresa propietaria de Netbeans. Aunque Oracle adquirió Sun Microsistems en 2009, Netbeans sigue siendo de código abierto y ofrece las siguiente características, entre otras:

- Permite escribir código en **C++**, **C**, **Ruby**, **Groovy**, **Javascript**, **CSS** y **PHP**, además de por supuesto, **Java**.
- Permite crear aplicaciones **J2EE** gracias a que incorpora servidores de aplicaciones Java como **Glassfish** y **Tomcat**.
- Permite crear aplicaciones gráficas con **Swing** de forma sencilla al estilo del Visual Studio de Microsoft.
- Permite crear aplicaciones **JME** para dispositivos móviles.

La plataforma Netbeans permite el desarrollo de aplicaciones a partir de un conjunto de componente de software llamados módulos. Un **módulo** no es más que un archivo Java que contiene un conjunto de clases escritas para interactuar con la API de Netbeans y un archivo especial, **manifest file**, que lo identifica como módulo.

Las aplicaciones creadas a partir de módulos pueden ser extendidas añadiendo más módulos. Debido a que los módulos pueden ser desarrollados por cualquiera, las aplicaciones basadas en esta plataforma pueden ser extendidas por cualquier desarrollador.

Cada módulo provee una función bien definida, tales como soporte para Java, edición o soporte para sistemas de control de versiones. Netbeans contiene todos los módulos necesarios para el desarrollo de aplicaciones Java en una sola descarga, permitiendo que la persona que va a trabajar con el comience a hacerlo inmediatamente.

En la página oficial de Netbeans puede encontrar toda la información que necesites sobre este IDE, así como aprender de forma más exhaustiva su uso diario.

1.7.4 Instalación y Configuración

En este curso, vamos utilizar la versión **Netbeans 19** del IDE, que es la última versión estable disponible. Para realizar su instalación debemos seguir los siguientes pasos:

1. Descargar la versión que deseemos de su página oficial, la cual podemos encontrar en este enlace. En nuestro caso, descargaremos la versión 19.
2. Ejecutar el fichero de instalación, en caso de que sea en Windows, o utilizar el gestor de paquetes APT para realizar la instalación del paquete, en caso de que usemos Linux.

A continuación, se proporcionan 2 enlace donde se explica la instalación y creación de un proyecto con Netbeans. Estos vídeos son sobre la versión 16 de Netbeans, pero el proceso es el mismo que la versión 19, que es la que usaremos.

- Instalación de Netbeans 16 - YouTube
- Creación del primer proyecto con Netbeans - YouTube

1.7.5 Aspecto del Entorno y Gestión de Proyectos

La pantalla inicial de nuestro entorno ofrece accesos directos a las operaciones más usuales: aprendizaje inicial, tutoriales, ejemplos, demos, los últimos programas realizados y las novedades de la versión.

Para describir el aspecto del entorno, es necesario crear un nuevo proyecto accediendo al menú "***File ->New Project***", donde indicaremos el tipo de aplicación que vamos a crear.

Una vez creado nuestro proyecto, la interfaz de Netbeans cambiará, y nos mostrará un conjunto de pestañas he información que podemos ver de forma resumida en la siguiente figura.

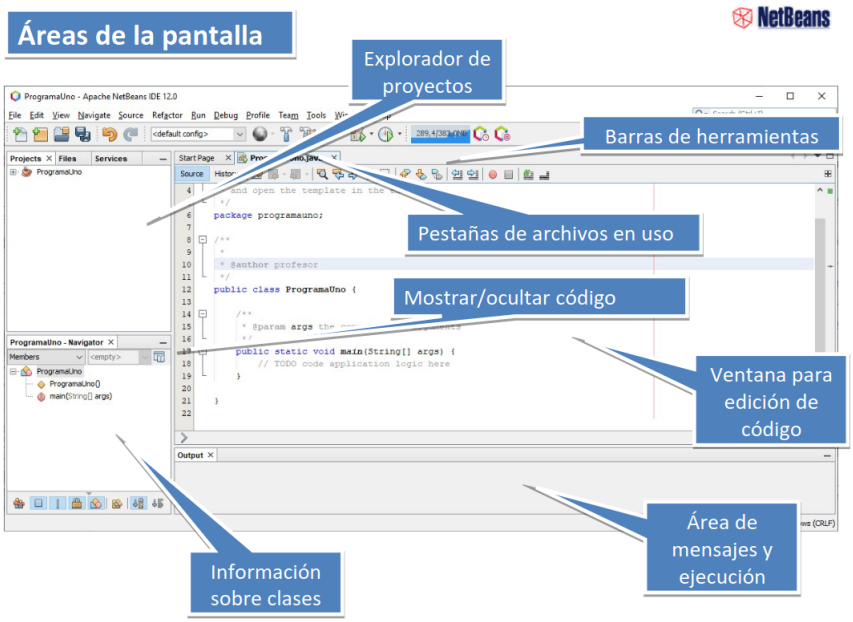

Figura 1.7.1: Ventana de proyecto de Netbeans

Cuando trabajemos con Netbeans nuestros proyectos harán usos de clases para poder desarrollar las operaciones de nuestros programas. Estas clases se agrupan en paquetes. En la siguiente figura se muestra un esquema de como se gestiona la creación de paquetes.

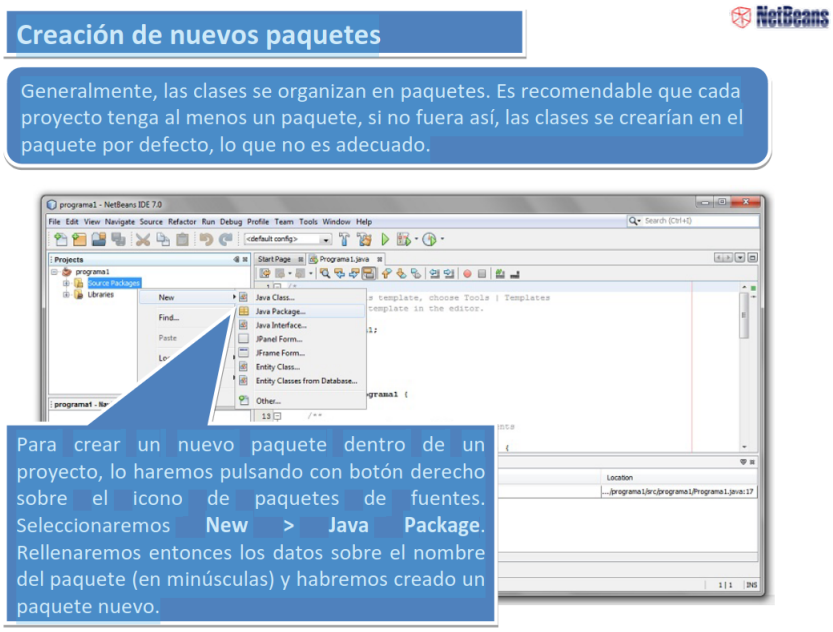

Figura 1.7.2: Creación de paquetes en Netbeans

Al igual que los paquetes, las clases se pueden crear de la misma forma, solo tenemos que seleccionar la opción "**Java Class**" en el menú que se muestra en al figura anterior, en vez de la opción "**Java Package**".

Una de las ventajas que ofrece este entorno es que podemos examinar nuestro proyectos a través de la vista **Archivos**. Esta vista nos enseña los archivos del proyecto, como la carpeta **build**, que contiene los archivos **.class**, la carpeta **src**, que es donde se encuentran los archivos de código fuente y el resto, son archivos creador por Netbeans para comprobar la configuración del proyecto o los archivos necesarios para interpretación del código en otros sistema. Para activar esta vista, seleccionar la opción "**Window –>Files**.

1.8 Variables e Identificadores

Un programa maneja datos para hacer cálculos, presentar resultados, solicitarlos al usuario, guardarlos en disco, etc. Para poder manejar estos datos, el programa los guarda en **variables**.

Una **variable** es una zona de memoria del computador que puede almacenar un valor de un determinado tipo para ser usado más tarde en el programa. Las variables vienen determinadas por:

- Un **nombre** que permite al programa acceder al valor que contiene en memoria. Debe ser un identificador válido, por ejemplo, podría llamarse *precioVenta*. A este nombre, se le conoce como **identificador**.
- Un **tipo de datos**, que especifica la clase de información que puede ser guardada por la variable en esta zona de memoria. Por ejemplo, puede ser de tipo entero, o de tipo de cadena de caracteres, o de tipo real, etc. La variable anterior, si pretendemos que almacene precios, podría definirse

como de tipo real, para que admita decimales. Cualquier lenguaje permitirá definir uno o varios tipos de reales. En Java, por ejemplo, **float** y **double** sería dos tipos de reales.

- Un **rango de valores** que puede admitir dicha variable. Establece el valor máximo y mínimo que se puede almacenar en la variable, así como la precisión. Normalmente va asociado al tipo de dato de la variable. Siguiendo con el ejemplo, en Java la diferencia entre definir una variable de tipo **float** o **double** supone que se podrán almacenar números reales más pequeños o más grandes y en la precisión que se puede usar. Así, para un precio, con **float** tendremos suficiente rango y precisión, ya que solo necesitamos 2 decimales y este tipo nos permite usar incluso más.

1.8.1 Identificadores

Un **identificador** en Java es una secuencia ilimitada, sin espacios, de letras y dígitos Unicode, de forma que el **primer símbolo** de la secuencia **debe ser una letra**, un **símbolo de subrayado** (_) o el **símbolo del dolar** ($). A continuación vemos unos ejemplos de identificadores válidos y no válidos.

- Identificadores **válidos**: *x5*, *NUM_MAX*, *numCuenta*, *_data* o *$PI*.
- Identificadores **no válidos**: *5p*, *-precio* o *%porcentaje*

En la definición anterior decimos que un identificador puede ser una secuencia ilimitada de caracteres Unicode, pero... ¿qué es Unicode? Unicode es un código de caracteres o o sistema de codificación, un alfabeto que recoge prácticamente todos los caracteres de los idiomas más importantes del mundo. Las lineas de código en los programas se escriben usando este conjunto de caracteres.

El estándar Unicode originalmente utilizaba 16 bits, pudiéndose representar hasta 65.536 caracteres distintos. Actualmente, Unicode puede utilizar más o menos bits dependiendo del formato que se utilice, así, **UTF-8**, **UTF-16** o **UTF-32** pueden usar 8, 16 y 32 bits respectivamente.

A cada carácter le corresponde un número entero unívoco perteneciente al intervalo de 0 a 2 elevado a n, siendo n el número de bits utilizado para representar los caracteres. Por ejemplo, la letra ñ es el entero 164. Además, el código Unicode es compatible con el **código ASCII**, ya que para los caracteres del código ASCII, Unicode asigna como código los mismos 8 bits, a los que se les añade a la izquierda otros 8 bits todos a 0. Así, la conversión de un carácter ASCII a Unicode es inmediata.

1.8.2 Convenios y Reglas

A la hora de escribir el nombre de las variables existen una **reglas de estilo** que, aunque no son obligatorias, se aceptan por convenio y se usan en la mayor parte del código escrito en Java, lo que ayuda a entender más rápidamente la semántica de cada identificador. Estas reglas para la nomenclatura de variables se pueden resumir en:

- Java **distingue mayúsculas de minúsculas**. Por ejemplo, **Alumno** y **alumno** son variables diferentes.
- **No** se suelen emplear identificadores que **comiencen por "$" o "_"**. Además, el símbolo del dolar, no se suele emplear nunca para identificadores que define el usuario.
- **No** se puede usar como nombre los valores booleanos **true** o **false**, así como tampoco el valor nulo, **null**.
- Los **identificadores** deben ser lo **más descriptivos posible**. Es mejor usar palabras completas en vez de abreviaciones crípticas. En muchos casos también hará que nuestro código se **autodocumente**. Por ejemplo, si tenemos que darle nombre a una variable que almacena los datos de un

cliente sería recomendable que la misma se llamara algo como **FicheroCliente** ó **Manejador-Cliente**, y no algo poco descriptivo como **Cl33**, por más que Java lo considere correcto.

Además de esta restricciones, en la siguiente tabla puedes ver otras convenciones, que no siendo obligatorias, sí son recomendable a la hora de crear identificadores en Java.

Convenciones sobre identificadores en Java

Identificador	Convención	Ejemplo
Nombre de variable.	Comienza por letra minúscula, y si está formado por varias palabras, se colocan juntas y todas las siguientes a partir de la segunda comenzarán por mayúsculas para ayudar a identificar visualmente dónde comienza cada nueva palabra a pesar de que no haya espacios.	`numAlumnosMatriculados`, `suma`
Nombre de constante.	Con todas sus letras en mayúsculas, separando las palabras con el guion bajo, y además por convenio el guion bajo no se utiliza en ningún otro sitio.	`TAM_MAX`, `PI`
Nombre de una clase.	Comienza por letra mayúscula.	`String`, `MiTipo`
Nombre de función o método.	Un nombre de una función o de un método comienza con letra minúscula. Sigue en realidad la misma nomenclatura que cualquier variable y se distingue que se trata de un método o función porque obligatoriamente debe ir seguido de paréntesis, que enmarcan la lista de parámetros que se le dan al método para trabajar. El paréntesis es obligado aunque no se le pase ningún parámetro.	`modificaValor()`, `obtieneValor()`

Figura 1.8.1: Convenciones de identificadores en Java

1.8.3 Palabras Reservadas

Las **palabras reservadas** son secuencias de caracteres ASCII cuyo uso se reserva para el lenguaje, y por tanto, **no pueden utilizarse** para crear identificadores. En la siguiente tabla se muestran todas las palabras reservadas en Java:

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

Figura 1.8.2: Palabras reservadas en Java

Hay palabras reservadas como **const** o **goto**, que apenas se usan en la implementación actual de Java. Por otro lado, puede haber otro tipo de palabras que aunque no sean reservadas tampoco se pueden usar

para la creación de identificadores. Es el caso de **true** y **false**, que son **literales booleanos**. Igualmente, **null** es considerada un literal, no una palabra reservada.

1.8.4 Tipos de Variables

Dentro de un programa podemos encontrar diferentes tipos de variables. La diferencia entre unas otras dependerá de diferentes factores como el tipo de dato que representan, si el valor puede cambiar o no, o cual es el papel que cumple en el programa.

En el **lenguaje de programación Java**, podemos distinguir los tipos de variables según los siguientes criterios:

- Según el **tipo de información** que contengan, podemos hablar de **variables de tipos primitivos** o **variables referencia**. En función de al grupo que pertenezca, podrá tomar unos valores u otros, y se podrán definir sobre ella unas operaciones u otras.
- Dependiendo de su **mutabilidad**, podemos hablar de variables **mutables** ó **inmutables**, es decir, según su valor pueda ser modificado o no durante la ejecución del programa.

En conclusión podemos decir sobre una variable que:

1. Es un **almacén temporal de información** que usan los programas para registrar datos y operar con ellos.
2. Puede contener un valor de **tipo primitivo** (entero, real, carácter,...) o bien **una referencia** (una dirección de memoria) a una zona de memoria que contendrá información mucho más compleja que un simple valor primitivo.
3. Se crea y se usa **dentro de un bloque de código**.
4. Deja de existir, es decir, **se destruye**, cuando la ejecución de ese bloque de código finaliza.

A continuación vamos a ver un ejemplo de definición de variables. Como vemos, en algunos casos se han indicado un valor inicial, mientras que en otros no.

```
int cantidadLargos = 10;
double longitudPiscina;
char letraDni;
String nombre;
final int MAXIMO_NUMERO_PLANTAS= 12;
final double PI= 3.1415926536;
```

Como vemos, son pequeños casilleros donde se almacenan cierta información básica (tipos **primitivos**)o bien una referencia (dirección de memoria) a información más compleja (tipos **referenciados**). En unos casos puede tratarse de información cambiante (**mutable**) y en otros constante (**inmutable**).

Por último, apuntar que en **Java**, para indicar que que una variables es **inmutable** o **constante**, se emplea el modificador ***final*** en su declaración.

En el siguiente ejemplo, vemos la creación de dos variables:

- **MAXIMO**: variable **inmutable** (**final**), con el valor **4.245**. Al haber sido declarada como constante no podrá modificarse a lo largo de la vida del programa.

- **y**: variable **mutable** que solo podrá ser accedida dentro del bloque de código donde se ha declarado, en este caso, el método **main()**, ya que fuera de él no existe. Podrá modificarse el valor que contiene tantas veces como se quiera a lo largo de la vida del programa.

En aparatados posteriores veremos como expandir más la funcionalidad de nuestro programas, por ahora, este ejemplo, solo muestra 3 líneas indicando que este es el primer programa y el valor de las variables.

```java
public class EjemploVariables {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int y ; // y es una variabe local
        final double MAXIMO = 4.245 ;

        // Asignamos un valor a la variable y
        y = 2 ;

        // aquí iría el código de nuestra aplicación
        System.out.println ("Hola, este es un primer programa") ;

        // Podemos escribir el valor de la variabel y así:
        System.out.println ("El valor de y es: " + y) ;

        System.out.println ("El valor máximo es: " + MAXIMO) ;

    }

}
```

La salida del programa por pantalla será la siguientes:

```
Hola, este es un primer programa
El valor de y es: 2
El valor máximo es: 4.245
```

1.9 Los Tipos de Datos

En los **lenguajes fuertemente tipados**, como es el caso de Java, a todo dato le corresponde un tipo que es conocido antes de que se ejecute el programa.

El tipo, limita el valor de la variable o expresión, las operaciones que se pueden realizar sobre esos valores y el significado de esas operaciones. Esto es así porque **un tipo**, no es más que la **especificación** de los valores que son **válidos** para esa variable y las operaciones que se pueden realizar sobre ella.

Durante el proceso de compilación, el compilador revisa todas las variables y sus tipos, sabiendo de antemano que rango de valores va a contener cada una. En el caso de que intentemos asignar a una variable un valor de otro tipo, el compilador mostrará un error. Aún así, en Java podemos realizar **conversiones entre ciertos tipos de datos**, lo que se conoce como **casting**.

Aunque ahora no vamos a entrar en detalle sobre la conversión de tipos, si es conveniente que sepas los tipos de datos que podemos encontrar en Java. Estos tipos de datos, pueden ser principalmente de dos categorías:

- **Tipos de datos primitivos**: representan valores simples que vienen predefinidos en el lenguaje, como por ejemplo, un carácter o un número.
- **Tipos de datos referencia**: se definen como un nombre o referencia que contiene la dirección de memoria de un valor o grupo de valores. Dentro de este tipo podemos encontrar, por ejemplo, los **vectores**, más usualmente conocidos como **arrays**, que son una serie de elementos del mismo tipo, o las **clases**, que son los modelos a partir de los cuales se crean objetos.

En los siguientes apartados veremos con más detalle estos dos tipos de datos.

1.9.1 Tipos de Datos Primitivos

Los **tipos primitivos** son aquellos datos sencillos que constituyen el tipo de datos más habituales: números, caracteres y valores lógicos. Al contrario que en otros lenguajes de programación, en Java, estos valores **no son objetos**.

Una de las mayores **ventajas** de tratar con datos primitivos es que el compilador puede optimizar mejor su uso. Además, cada uno de los tipos tiene **idéntico tamaño** en las **diferentes versiones de Java** y para **cualquier tipo de ordenador**. Esto quiere decir que no debemos de preocuparnos como se representa cada tipo de datos en las diferentes plataformas, asegurando la **portabilidad** de los programas, a diferencia de lo que ocurro con otros lenguajes.

Es importante que consultes este enlace, donde se explica con más detalle los diferentes tipos de datos primitivos que nos podemos encontrar en Java y el rango de valores que pueden tomar.

1.9.1.1 Tipo Entero y Carácter

Hay una peculiaridad con los datos primitivos en Java, y es que el tipo de dato ***char*** es considerado un número por el compilador, ya que los valores que guarda son el código Unicode correspondiente al carácter que representa, no en carácter en sí, por lo que **puede operarse con caracteres** como si fueran **números**.

A la hora de elegir un tipo deberemos **tener en cuenta** como es la información que va a almacenar, si es un número, una letra, un valor booleano,.. así como el tango de valores que va a poder tomar. En este sentido, hay veces que aunque no queramos representar un número con decimales, deberemos usar un número real.

Por ejemplo, si queremos representar la población mundial del planeta, no podremos usar un dato de tipo **int**, ya que el valor máximo que puede alcanzar es 2.147.483.647, siendo este el número máximo de combinaciones posibles con 32 bits, teniendo en cuenta que en Java la representación de los números enteros se realiza en complemento a 2. Si queremos representar a la población mundial deberemos usar al menos un tipo de dato **long**, o si tenemos problemas de espacio, de tipo ***float***. Sin embargo los **tipos reales** tiene otro problema, **la precisión**.

1.9.1.2 Tipo Real

El **tipo de dato real** permite representar números con decimales. Al igual que ocurre con los enteros, la mayoría de lenguajes define diferentes tipos de datos reales, dependiendo del número de bits que utilicen para realizar la representación. Cuanto mayor bits se usen:

- **Más grande** podrá ser el número real representado.
- **Mayor será** la **precisión** de la parte decimal.

Entre dos números reales cualesquiera, matemáticamente siempre tendremos infinitos números reales, pero un ordenador no puede representar infinitos números, porque no dispone de capacidad ilimitada, por lo que la mayoría de números reales **se representan** de **forma aproximada**.

Los números reales se representan en **coma flotante** o **notación científica**, que consiste en trasladar la coma a la primera cifra significativa del valor, con objeto de poder representar el máximo de números posibles.

Internamente, un número se representa en el interior de un computador con la siguiente formula:

$$Valor = mantisa * 2^{exponente}$$

Donde la **mantisa** son las cifras significativas del número. De este modo, para almacenar el número, solo se guardan la mantisa y el exponente al que va a ser elevada la base. Los bits empleados por la **mantisa** representan la **precisión** del número real, es decir, el número de cifras decimales significativas que puede tener el número, mientras que los bits del **exponente** representan la diferencia entre el mayor y menor número representable, es decir, el **intervalo de representación**.

En Java las variables de tipo ***float*** se emplean para representar los números de coma flotante de simple precisión de **32 bits**, de los cuales 24 bits son para la mantisa y 8 bits para el **exponente**. Por su parte, las variables de tipo ***double*** representan los números en coma flotante con doble precisión de **64 bits**, de los cuales 53 son para la mantisa y 11 para el exponente.

La mayoría de programadores usa ***double*** en Java cuando están tratando con datos de tipo real. Es una forma de asegurarse que los errores cometidos en la sucesivas aproximaciones sea menor. De hecho, Java considera los datos de tipo coma flotante como ***double*** por defecto. Así el literal **3.25** será considerado como tipo double. En caso de querer representarlo como float deberemos indicarlo en su definición con **3.25f** o **3.25F**.

Con objetivo de conseguir la máxima portabilidad de los programas, Java usa el estándar IEE 754 para la representación interna de los números en coma flotante, para asegurarse que el resultado de los cálculos sea el mismo en todas las plataformas.

1.9.2 Tipos Referenciados

A partir de los tipos de datos primitivos, se pueden construir otros tipos de datos. Estos tipos de datos se llaman **tipos referenciados** o **referencias**, porque se utilizan para almacenar la dirección de los datos en la memoria del ordenador.

```
int[] arrayDeEnteros;
Cuenta cuentaCliente;
```

En la primera línea declaramos una lista de elementos del mismo tipo, en este caso, enteros, lo que se suele conocer como un **array de enteros**. En la segunda línea declaramos un objeto, ***cuentaCliente***, como una referencia del tipo **Cuenta**.

Cualquier aplicación de hoy en día necesita tratar con un conjunto de datos. Cuando estos datos tiene características similares suelen agruparse en estructuras para facilitar el acceso a los mismos. Son los llamados **datos estructurados**. Algunos ejemplos de estos datos son los **arrays**, **listas**, **arboles**, etc. Pueden estar en la memoria del programa en ejecución, guardados en el disco como ficheros o almacenados en una base de datos.

Además de los 8 tipos de datos primitivos visto anteriormente, Java proporciona un tratamiento especial a los textos o cadenas de caracteres mediante el tipo de dato ***String***. Java crea automáticamente este tipo de dato cuando se encuentra un cadena de caracteres encerrada entre comillas dobles. En realidad se trata de **objetos** y por tanto son **tipos referenciados**, aunque el lenguaje nos permite usarlos de forma sencilla como si fueran tipos primitivos.

En esta unidad no veremos como definir ni tratar los datos referenciados, los cuales veremos en unidades posteriores de este curso con más detalle.

1.9.3 Tipos Enumerados

Los **tipos de datos enumerados** permiten declarar una variable con un conjunto restringido de valores. Por ejemplo: los dias de las semana, los meses de un año, etc..

Para declararlos, se usa la palabra reservada ***enum***, seguida del nombre de la variable y la lista de valores que puede tomar entre llaves. Los valores que puede tomar se consideran constantes, van separados por comas y deben ser valores únicos.

Este tipo de dato es considerado por Java como un tipo de clase, por lo que no solo podemos definir los valores del tipo enumerado, sino que también podemos definir operaciones a realizar con él y otro tipo de elementos, lo que hace que este tipo de dato sea más versátil y potente que en otros lenguajes de programación. Por el momento debemos quedarnos con el **tipo enum** nos **permite definir** un nuevo **tipo de dato** donde los valores que puede tomar son los que nosotros indicamos.

A continuación tienes un ejemplo de creación y uso del tipo de dato **enum** en Java. Declaramos un **enum Dias** que contiene valores que representan todos los posibles días de la semana. Para acceder a cada valor, se utiliza el nombre del **enum**, seguido de un punto y el valor de la lista.

```
public class TiposEnumerados {
    public enum Dias {LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO};

    public static void main(String[] args) {
        Dias diaActual = Dias.MARTES ;
        Dias diaSiguiente = Dias.MIERCOLES ;

        System.out.println("Hoy es: " + diaActual);
        System.out.println("Mañana es " + diaSiguiente);
    }
}
```

Este ejemplo ademas de declarar **enum Dias** con su posibles valores declara dos variables de ese nuevo tipo (**diaActual y diaSiguiente**) que acabamos de definir y les asigna valores de entre el conjunto de valores posibles.

1.9.4 Declaración e Inicialización

Como podrás imaginar, debemos de crear la variables antes de utilizarlas en nuestro programas, indicando que nombre van a tener y que tipo de información van a almacenar, es decir, debemos **declarar la variable**.

Las variables se pueden declarar en cualquier bloque de código, dentro de llaves. Y lo hacemos indicando su **identificador** y el **tipo de datos**, separadas por comas si vamos a declarar varias a la vez, por ejemplo:

```java
int numeroAlumnos = 15;
double radio = 3.14, importe = 102.95;
```

De esta forma, estamos declarando **numeroAlumnos** como una variable de tipo **int** y otras dos variables, **radio** e **importe**, de tipo **double**. En este caso se les asigna una valor en la declaración de las variables para inicializarlas, aunque esto no es obligatorio.

Si la variable va a permanecer inalterable a lo largo del programa, la declararemos **constante**, utilizando la palabra reservada **final** de la siguiente forma:

```java
final double PI = 3.1415926536;
```

En algunas ocasiones declararemos variables sin asignarles un valor. Hay que tener en cuenta que las **variables no se inicializan de forma automática**, por lo que deberemos ser nosotros los que les asignemos un valor antes de ser usadas, ya que si el compilador detecta que intentamos usar una variable antes de que sea inicializada nos devolverá un error.

Por último, vamos a ver un ejemplo de declaración y uso de todos los tipos primitivos que hemos visto (lo referenciados se verán mas adelante), aunque también se incluye el tipo ***String***, que es de tipo referenciado pero Java permite trabajar con el como si fuera un tipo primitivo.

```java
public class EjemploTipos {

    public static void main(String[] args) {
        int i = 10;
        double d = 3.14;
        char c1 = 'a';
        char c2 = 65;
        boolean encontrado = true;
        String mensaje = "Bienvenido a Java";

        System.out.println("La variable i es de tipo entero y su valor es: " + i);
        System.out.println("La variable d es de tipo double y su valor es: " + d);
        System.out.println("La variable c1 es de tipo carácter y su valor es: " + c1);
        System.out.println("La variable c2 es de tipo carácter y su valor es: " + c2);
        System.out.println("La variable encontrado es de tipo booleano y su valor es: " + encontrado);
        System.out.println("La variable mensaje es de tipo String y su valor es: " + mensaje);
    }
}
```

El resultado que se obtendrá por pantalla al ejecutar este programa tendrá el siguiente aspecto:

```
La variable i es de tipo entero y su valor es: 10
La variable d es de tipo double y su valor es: 3.14
La variable c1 es de tipo carácter y su valor es: a
La variable c2 es de tipo carácter y su valor es: A
La variable encontrado es de tipo booleano y su valor es: true
La variable mensaje es de tipo String y su valor es: Bienvenido a Java
```

1.10 Literales de los Tipos Primitivos

Un **literal** es un valor concreto para para los tipos de datos primitivos del lenguaje, el tipo **string** o el valor **null**.

Los **literales booleanos** son dos únicos valores, los que puede aceptar el tipo: **true** y **false**. Por ejemplo, con las instrucción ***boolean encontrar = true;*** estamos declarando una variable de tipo booleana a la cual hemos asignado el valor literal **true**.

Los **literales enteros** se pueden representar con tres notaciones diferentes:

- **Decimal**: es la forma más común. Por ejemplo, **20** es un literal entero decimal.
- **Octal**: un número representado en base octal, que siempre comenzará por cero e irá seguido de dígitos octales (0 a 7). Por ejemplo, **024**.
- **Hexadecimal**: un número en base hexadecimal que siempre empezará por **0x** e irá seguido de dígitos hexadecimales (de 0 a 9 y de 'a' a la 'f' o de 'A' a la 'F'). Por ejemplo: **0x14**.

A los **literales de long** se les debe añadir detrás de una ele mayúscula o minúscula (**l** ó **L**), por ejemplo, **837L**, de lo contrario, se considera por defecto un literal de tipo **int**. Se suele usar **L** para evitar la confusión entre la ele minúscula y **1**.

Los **literales reales** se expresan con como decimal o en notación científica, osea, seguidos de un exponente **e**. El valor puede finalizarse con una **f** ó **F** para indicar que se trata de un literal de tipo **float** o con una **d** ó **D** para indicar que el formato es **double**, por defecto, si no se pone nada, se entiende que es de tipo **double**. Por ejemplo, podríamos representar un mismo literal de las siguientes formas: **13.2**, **13.2D**, **1.32e1** ó **1.32E1**.

Desde Java SE 7 y posteriores, puede aparecer el carácter "_" entre dígitos de un literal numérico. La idea subyacente es mejorar la legibilidad del código. Así, por ejemplo, sería válido escribir:

```
long numeroTarjeta = 1234_5678_9021_3456L
```

Un **literal carácter** puede escribir con un carácter entre comillas simples, como **'a'**, **'ñ'**,**'Z'**, etc., o por su código de la tabla Unicode, anteponiendo la secuencia de escape **'´** si el valor lo pones en octal, **'\u'** si el valor lo ponemos en hexadecimal. Por ejemplo, si sabemos tanto que tanto en código ASCII como en Unicode, la letra A (mayúscula) es el símbolo número 65, y que 65 en octal es 101 y 41 en hexadecimal, podemos representar esta letra comom **\101** en octal o **\u0041**. Existen, ademas, unos caracteres especiales que se presentan utilizando secuencias de escape.

Secuencia de escape	Significado	Secuencia de escape	Significado
\b	Retroceso	\r	Retorno de carro
\t	Tabulador	\''	Carácter comillas dobles
\n	Salto de línea	\'	Carácter comillas simples
\f	Salto de página	\\	Barra diagonal

Figura 1.10.1: Secuencias de escape en Java

Normalmente lo objetos en Java deben crearse empleando el operador ***new***. Sin embargo, los literales ***String*** no lo necesitan ya que son objetos que se crean implícitamente por Java, pudiendo prescindir del uso del operador ***new***.

Los **literales de cadenas de caracteres** se indican entre comillas dobles. En el ejemplo anterior "**El primer mensaje**" es un literal de tipo cadena de caracteres. Al construir una cadena de caracteres se puede incluir cualquier carácter Unicode excepto un carácter de retorno de carro. Por ejemplo, en la siguiente instrucción utilizamos la secuencia de escale "\" para escribir dobles comillas dentro del mensaje:

```
String texto = "Juan dijo: \"Hoy hace un dia fantastico\"";
```

En el ejemplo del apartado anterior ya estábamos usando secuencias de escape para introducir un salto de línea en una cadena de caracteres, utilizando el carácter especial '**\n**'.

1.11 Operadores y Expresiones

Los **operadores** llevan a cabo operaciones sobre un conjunto de datos u operandos, representados por literales o identificadores. Los operadores pueden ser **unarios**, **binarios** o **terciarios**, según el número de operandos que utilicen sean uno, dos o tres, respectivamente. Los operadores actúan sobre los tipos de datos primitivos y también devuelven tipos de datos primitivos.

Los operandos se combinan con los literales y/o identificadores para formar **expresiones**. Una **expresión** es una combinación de operadores y operandos que se evalúa produciendo un único resultado de un tipo determinado.

El resultado de una expresión puede ser usado en otra expresión o en una sentencia. Las expresiones, combinadas con algunas palabras reservadas o por sí solas forman las **sentencias** o **instrucciones**. Por ejemplo:

```
i + 1
```

Con esta expresión estamos usando operador aritmético para sumarle una cantidad a la variable ***i***, pero es necesario indicar al programa que hacer con el resultado. Así, podrías escribir:

```
suma = i + 1
```

De esta forma, le indicaríamos al programa que almacena el valor de esa expresión en la variable ***suma***, formando lo que se conoce como una **sentencia** o **instrucción**.

Más ejemplos de sentencias lo tenemos en la declaración de variables, que vimos en el apartado anterior o en las estructuras de control del lenguaje, como bucles, que veremos en puntos posteriores.

Como curiosidad comentar que las **expresiones de asignación**, al poder ser usadas como parte de otras asignaciones u operaciones, son consideradas tanto expresiones en sí mismas como sentencias.

1.11.1 Operadores Aritméticos

Los operadores aritméticos son aquellos operadores que combinados con los operandos forman expresiones matemáticas o aritméticas. En la siguiente tabla, podemos ver los operadores aritméticos básicos.

Operador	Operación Java	Expresión Java	Resultado
-	Operador unario de cambio de signo	`-10`	-10
+	Adición	`1.2 + 9.3`	10.5
-	Sustracción	`312.5 - 12.3`	300.2
*	Multiplicación	`1.7 * 1.2`	2.04
/	División (entera o real)	`0.5 / 0.2`	2.5
%	Resto de la división entera	`25 % 3`	1

Figura 1.11.1: Operadores aritméticos básicos

Dependiendo del tipo de operandos que se empleen en con estos operadores, el resultado de la operación tendrá un tipo u otro. Así, en la siguiente tabla podemos ver el tipo resultante dependiendo del tipo de los operandos.

Tipo de los operandos	Resultado
Un operando de tipo `long` y ninguno real (ni `float` ni `double`)	`long`
Ningún operando de tipo `long` ni real (`float o double`)	`int`
Al menos un operando de tipo `double`	`double`
Al menos un operando de tipo `float` y ninguno `double`	`float`

Figura 1.11.2: Tipo del resultado según operandos

Además de estos operadores aritméticos, otro tipo de **operadores unarios** son los de **incremento y decremento**. Producen un resultado del mismo tipo que el operando y podemos usarlos con **notación prefija**, si el operador aparece antes que el operando, o **notación sufija** si el operador aparece después que el operando. En la siguiente tabla se ve un ejemplo del uso de estos operadores.

Tipo operador	Expresión Java	
	Prefija:	Postfija:

++ (incremental)	`x=3;`
	`y=++x;`
	`// x vale 4 e y vale 4`
		`x=3;`
		`y=x++;`
		`// x vale 4 e y vale 3`

--(decremental)	`5--; // el resultado es 4`	

Figura 1.11.3: Operadores de incremento y decremento

En la siguiente tabla, podemos ver un ejemplo de uso de los **operadores aritméticos** en un programa básico donde se muestra además por pantalla el resultado de cada operación.

Ejemplo de código	Salida por pantalla

```java
public class OperadoresAritmeticos {
    public static void main(String[] args) {
        // Declaración de variables
        short x = 7;
        int y = 5;
        float f1 = 13.5f;
        float f2 = 8f;

        // Ejemplos de operaciones
        System.out.println("EJEMPLOS DE USO DE OPERADORES ARITMÉTICOS");
        System.out.println("----------------------------------------");
        System.out.println("El valor de x es " + x + ", y es " + y);
        System.out.println("El resultado de x + y es " + (x + y));
        System.out.println("El resultado de x - y es " + (x - y));
        System.out.println("División entera: x / y = " + (x/y));
        System.out.println("Resto de la división entera: x % y = " + (x % y));
        System.out.println("El valor de f1 es " + f1 + ", f2 es " + f2);
        System.out.println("El resultado de f1 / f2 es " + (f1 / f2));
    }
}
```

```
EJEMPLOS DE USO DE OPERADORES ARITMÉTICOS
----------------------------------------
El valor de x es 7, y es 5
El resultado de x + y es 12
El resultado de x - y es 2
División entera:
x / y = 1
Resto de la división entera: x % y = 2
El valor de f1 es 13.5, f2 es 8.0
El resultado de f1 / f2 es 1.6875
```

Figura 1.11.4: Ejemplo de uso de operadores aritméticos

1.11.2 Operadores de Asignación

El principal operador de esta categoría es el **operador de asignación** "**=**", que permite al programa asignar un valor a una variable y que ya hemos usado varias veces en esta unidad. Además de este operador, Java proporcionar otros operadores de asignación combinados con operadores aritméticos que permiten abreviar operaciones.

Por ejemplo, el operador "**+=**", suma el valor de la expresión a la derecha del operando con el valor de la variable que hay a la izquierda del operador y almacena el resultado en dicha variable. En la siguiente tabla, se muestran todos los operadores de asignación que podemos encontrarnos en Java.

Operador	Ejemplo en Java	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`

Figura 1.11.5: Operadores de asignación

1.11.3 Operadores de Relación

Los **operadores relacionales** se utilizan para **comparar datos primitivos** (número, carácter y booleano). El resultado de la comparación se usará en otras expresiones o sentencias, que permitan comprobar en una condición dicho resultado, y ejecutar una acción u otra en función de si se cumple ésta o no.

Estas expresiones en Java dan como resultado un valor booleano, **true** o **false**. En la siguiente tabla vemos todos los operadores relacionales que podemos encontrarnos en Java.

Operador	Ejemplo en Java	Significado
==	op1 == op2	op1 igual a op2
!=	op1 != op2	op1 distinto de op2
>	op1 > op2	op1 mayor que op2
<	op1 < op2	op1 menor que op2
>=	op1 >= op2	op1 mayor o igual que op2
<=	op1 <= op2	op1 menor o igual que op2

Figura 1.11.6: Operadores relacionales

Hasta ahora hemos visto ejemplos que creaban variables y se inicializaban a alguna valor. Pero si queremos que el usuario introduzca un valor, en vez de asignarlo, deberemos añadirle interactividad, añadiendo por ejemplo la función **Scanner**. Aunque no hemos visto aún que son las clases y los objetos, de momento vamos a pensar que la clase **Scanner** nos va a permitir leer datos que se escriben por el teclado y que para usarla es necesario importar el paquete que la contiene.

```
import java.util.Scanner; //importamos el paquete necesario para poder usar la clase Scanner

public class EjemploRelacionales {

   public static void main( String args[] ){

      Scanner teclado = new Scanner( System.in );
      int x, y;
      String cadena;
      boolean resultado;

      System.out.print( "Introducir primer número: " );
      x = teclado.nextInt(); // pedimos el primer número al usuario
      System.out.print( "Introducir segundo número: " );
      y = teclado.nextInt(); // pedimos el segundo número al usuario

      // realizamos las comparaciones y las mostramos por pantalla
      cadena=(x==y)?"iguales":"distintos";
      System.out.printf("Los números %d y %d son %s\n",x,y,cadena);
      resultado=(x!=y);
      System.out.println("x != y  // es " + resultado);
      resultado=(x < y );
      System.out.println("x < y  // es " + resultado);
      resultado=(x > y );
      System.out.println("x > y  // es " + resultado);
      resultado=(x <= y );
      System.out.println("x <= y  // es " + resultado);
      resultado=(x >= y );
      System.out.println("x >= y  // es " + resultado);
    }
}
```

En este código de ejemplo, el programa se quedará esperando a que el usuario escriba algo por teclado y pulse la tecla intro. En ese momento se convierte el valor leído aun valor de tipo **int** y lo guarda en la variable indicada. Además de los operadores relacionales, en este ejemplo ejemplo usamos también el operador condiciona, que compara si los números son iguales o no. si lo son, devuelve la cadena iguales, y si no, la cadena distintos.

1.11.4 Operadores Lógicos

Los **operadores lógicos** realizan operaciones sobre valores booleanos o resultados de expresiones relacionales, dando como resultado un valor booleano.

Existen ciertos casos en los que el **segundo operando** de una expresión **no se evalúa**, para ahorrar tiempo de ejecución, porque con la evaluación del primero ya es suficiente para saber cual será el resultado de la expresión.

Por ejemplo, en la expresión ***a && b***, sí ***a*** es falso, no se sigue comprobando, ya que se sabe que la condición de que ambos sean verdaderos no se va a cumplir. En estos casos, es recomendable colocar el operando más propenso a ser falso en en lado de la izquierda. Igualmente ocurre con el operador ||, donde se deberá colocar a la izquierda el operando con más posibilidades de ser verdadero.

A continuación se muestra una tabla con los principales operadores booleanos que podemos encontrarnos en Java.

Operador	Ejemplo en Java	Significado
`!`	!op	Devuelve `true` si el operando es `false` y viceversa.
`&`	op1 & op2	Devuelve true si `op1` y `op2` son `true`
`|`	op1 | op2	Devuelve `true` si `op1` u `op2` son `true`
`^`	op1 ^ op2	Devuelve `true` si sólo uno de los operandos es `true`
`&&`	op1 && op2	Igual que `&`, pero si `op1` es `false` ya no se evalúa `op2`
`||`	op1 || op2	Igual que `|`, pero si `op1` es `true` ya no se evalúa `op2`

Figura 1.11.7: Operadores lógicos

En el siguiente código podemos ver algunos ejemplos del uso de los operadores lógicos que hemos visto en este apartado.

```
System.out.println(" false & false es : " + (false & false));
System.out.println(" true  & false es : " + (true & false));
System.out.println(" true  | false es : " + (true | false));
System.out.println(" false ^ true  es : " + (false ^ true));
System.out.println(" false && true  es : " + (false && true));
System.out.println(" true  && true  es : " + (true && true));
System.out.println(" true  || false es : " + (true || false));
```

1.11.5 Operador Condicional

El **operador condicional** "***? :***", sirve para **evaluar una condición** y **devolver un resultado** u **otro** en función de si esta es verdadera o falsa. Es el único **operador ternario** de Java, y como tal, necesita tres operandos.

- El **primer operando** se sitúa a la **izquierda del símbolo de interrogación** (***?***)y siempre será una expresión booleana, también llamada condición.
- El **segundo operador** se sitúa a la **derecha del signo de interrogación** (***?***) y **antes de los dos puntos** (***:***), y es el **valor que se devolverá** si la expresión del primer operando **es verdadera**.
- El **tercer y último operando**, que se sitúa **después de los dos puntos** (***:***), es la expresión cuyo **valor de devolverá** si la condición evaluada en el operando primero **es falsa**.

En la siguiente tabla, se muestra con más detalle la sintaxis de este operador para que quede más clara la idea.

Operador	Expresión en Java
`? :`	`condición ? exp1 : exp2`

Figura 1.11.8: Operador condicional

Por ejemplo, en la siguiente expresión:

```
(x>y) ? x : y ;
```

Se evalúa la condición de si **x es mayor que y**, en caso de que ésta **sea verdadera**, se **devolverá el valor de x**, en caso contrario, se **devolverá el valor de y**.

1.11.6 Operadores de Bits

Los **operadores a nivel de bits** se caracterizan porque realizan operaciones sobre los números enteros (o char) en su representación binaria, es decir, sobre cada dígito binario.

Aunque estos operadores no son de uso frecuente, sino más bien para aplicaciones especificas, no esta de más que al menos sepas cuales son. En la siguiente tabla, tienes todos los operadores binarios que podemos encontrar en Java.

Operador	Ejemplo en Java	Significado
`~`	**~op**	Realiza el complemento binario de `op` (invierte el valor de cada bit)
`&`	**op1 & op2**	Realiza la operación `AND` binaria sobre `op1` y `op2`
`|`	**op1 | op2**	Realiza la operación `OR` binaria sobre `op1` y `op2`
`^`	**op1 ^ op2**	Realiza la operación `OR-exclusivo` (`XOR`) binaria sobre `op1` y `op2`
`<<`	**op1 << op2**	Desplaza `op2` veces hacia la izquierda los bits de `op1`
`>>`	**op1 >> op2**	Desplaza `op2` veces hacia la derecha los bits de `op1`
`>>>`	**op1 >>> op2**	Desplaza `op2` (en positivo) veces hacia la derecha los bits de `op1`

Figura 1.11.9: Operadores binarios

1.11.7 Trabajo con Cadenas

Ya hemos visto en la sección sobre literales que el objeto ***String*** se corresponde con una secuencia de caracteres entrecomillados, por ejemplo, ***"Hola"***. Este tipo de literal se puede emplear en Java como si de un tipo primitivo se tratase, y como caso especial que es, no es necesario el uso del operador ***new***.

En ese punto vamos a utilizar determinadas operaciones sobre el objeto ***String***, y lo verás más claro con ejemplos descriptivos.

Para aplicar una operación a una variable de tipo de ***String***, escribiremos su nombre seguido de la operación, separados por un punto. En el siguiente ejemplo, vemos todas las operaciones que se pueden realizar sobre cadenas, las cuales pasaremos a explicar a continuación:

```java
public class EjemploCadenas {

    public static void main(String[] args) {

        // Cadenas de ejemplo con las que trabajar
        String cadena1 = "CICLO DAM-DAW";
        String cadena2 = "ciclo dam-daw";

        System.out.println ("EJEMPLOS DE OPERACIONES CON CADENAS");
        System.out.println ("-----------------------------------");

        // Mostramos las cadenas originales
        System.out.println ("La cadena cadena1 es " + cadena1);
        System.out.println ("La cadena cadena2 es " + cadena2);
        System.out.println ();

        System.out.println ("Longitud de cadena1: " + cadena1.length());
        System.out.println ("Longitud de cadena2: " + cadena2.length());

        // Concatenación de cadenas (concat o bien operador +)
        System.out.println ("Concatenación cadena1 y cadena2: " + cadena1.concat(cadena2));
        System.out.println ("Concatenación cadena2 y cadena1: " + cadena2 + cadena1);

        // Comparación de cadenas
        System.out.println ("cadena1.equals(cadena2) es: " + cadena1.equals(cadena2));
        System.out.println ("cadena1.equalsIgnoreCase(cadena2) es: " + cadena1.equalsIgnoreCase(cadena2));

        // Obtención de subcadenas
        System.out.println ("cadena1.substring(0,5) es: " + cadena1.substring(0, 5));

        // Pasar a minúsculas
        System.out.println ("cadena1.toLowerCase() es: " + cadena1.toLowerCase());
        System.out.println();
    }
}
```

Entre las principales operaciones que podemos ver en el código anterior tenemos las siguientes:

- **Creación**: como hemos visto en la sección de literales, podemos crear una variable de tipo ***String*** simplemente asignándole una cadena de caracteres entre comillas dobles.
- **Obtención del carácter**: podemos obtener el carácter que se encuentra en una posición determinada. Para ello se utiliza el método **charAt**.
- **Obtención de longitud**: si necesitamos obtener la longitud de un ***String***, podemos usar el método ***length***
- **Concatenación**: se utiliza el operador **+** entre dos cadenas de caracteres o el método ***concat()*** para concatenar cadenas de caracteres.

- **Comparación**: el método ***equals*** nos devuelve un valor booleano que nos indica si las dos cadenas son iguales o no. El método ***equalsIgnoreCase*** hace lo mismo, pero ignorando las mayúsculas de las cadenas que se comparan. La comparación entre objetos ***String***, **nunca** debe realizarse utilizando el operador **==**.
- **Obtención de subcadenas**: podemos obtener cadenas derivadas de una cadena original mediante el método ***substring()***, en el cual debemos indicar la posición de inicio y de fin de la subcadena a obtener.
- **Cambio de mayúsculas/minúsculas**: los métodos ***toUpperCase*** y ***toLowerCase*** devuelven una nueva cadena con todos los caracteres en mayúsculas o minúsculas respectivamente.
- **Conversiones**: utilizaremos el método ***valueOf*** para convertir un tipo de dato primitivo a una variable de tipo ***String***.

Sabiendo ya para que se usa cada operación, el resultado del ejemplo anterior que obtendríamos en la consola sería el siguiente:

```
EJEMPLOS DE OPERACIONES CON CADENAS
-----------------------------------
La cadena cadena1 es CICLO DAM-DAW
La cadena cadena2 es ciclo dam-daw

Longitud de cadena1: 13
Longitud de cadena2: 13
Concatenación cadena1 y cadena2: CICLO DAM-DAWciclo dam-daw
Concatenación cadena2 y cadena1: ciclo dam-dawCICLO DAM-DAW
cadena1.equals(cadena2) es: false
cadena1.equalsIgnoreCase(cadena2) es: true
cadena1.substring(0,5) es: CICLO
cadena1.toLowerCase() es: ciclo dam-daw
```

1.11.8 Precedencia de Operadores

El **orden de precedencia de los operadores** determina la secuencia en la que deben realizarse las operaciones cuando en una expresión intervienen diferentes operadores. Las reglas de operadores que utiliza Java **coinciden** con las **reglas de álgebra convencional**. Por ejemplo:

- La **multiplicación**, **división** y **resto** de una expresión se **evalúan en primer lugar**. Si dentro de la expresión hay varias operaciones de este tipo, se evaluarán de izquierda a derecha.
- La **suma** y la **resta** se evalúan después de los operadores anteriores. De igual manera, si se tiene varias operaciones de suma o resta se evaluarán de izquierda a derecha.

A la hora de evaluar una expresión, hay que tener también en cuenta las **asociatividad** de los operadores. La asociatividad nos indica que operador se evalúa antes, en condiciones de igualdad de precedencia.

Los **operadores de asignación**, el **operador condicional**, los **operadores incrementales** y el **casting**, son asociativos por la derecha. En cambio, el **resto de operadores** son asociativos por la izquierda, es decir, se evalúan en el orden en el que están escritos. Por ejemplo, en la siguiente expresión:

```
10 / 2 * 5
```

Realmente la operación que se realiza es **(10/2) * 5**, porque ambos operadores, división y multiplicación, tiene igual precedencia y por tanto se evalúa primero el que antes nos encontramos por la

izquierda, que es la división. El resultado de la operación es **25**. Si fueran asociados por la derecha el resultado sería **1**, ya que primero se multiplicaría **2 * 5** y después se realizaría la división entre **10**. En cambio, en la siguiente sentencia:

```
x = y = z = 1
```

Realmente la operación que se realiza es **x = (y = (z = 1))**. Primero se le asigna el valor a la variable**z**, luego a la variable **y**, para terminar asignando el resultado a la variable **x**.

En la siguiente tabla se detalla el orden de preferencia y la asociatividad de todos los operadores que hemos visto en este apartado. Los operadores se muestran de mayor a menor preferencia.

Operador	Tipo	Asociatividad
`++ --`	Unario, notación postfija	Derecha
`++ -- + - (cast) ! ~`	Unario, notación prefija	Derecha
`* / %`	Aritméticos	Izquierda
`+ -`	Aritméticos	Izquierda
`<< >> >>>`	Bits	Izquierda
`< <= > >=`	Relacionales	Izquierda
`== !=`	Relacionales	Izquierda
`&`	Lógico, Bits	Izquierda
`^`	Lógico, Bits	Izquierda
`|`	Lógico, Bits	Izquierda
`&&`	Lógico	Izquierda
`||`	Lógico	Izquierda
`?:`	Operador condicional	Derecha
`= += -= *= /= %=`	Asignación	Derecha

Figura 1.11.10: Preferencia de operadores

Bibliografía

[1] OpenJDK - Wikipedia. https://en.wikipedia.org/wiki/OpenJDK.