YO PROGRAMO

{ en JAVA }

DANIEL E. AGUIL MALLEA

[códigos utilizados]

Introducción a la programación

utilizando JAVA como herramienta

Daniel E. Aguil Mallea

Este libro fue escrito para enseñar a programar.

Adopta el enfoque orientado a objetos y explica los conceptos básicos

de la programación necesarios para resolver problemas.

Aguil Mallea, Daniel Eugenio

Yo programo : en java / Daniel Eugenio Aguil Mallea ; Contribuciones de Germán Carlos Tejero ; Nadia Patricia Ramos ; Simon Paradiso ; Coordinación general de Daniel Eugenio Aguil Mallea. - 1a edición para el alumno - Ushuaia : Daniel Eugenio Aguil Mallea, 2024.
Libro digital, PDF

296 p. : il. ; 25 x 20 cm.

ISBN 978-631-00-5204-5

1. Lenguajes de Programación. 2. Tecnología Informática. 3. Material Auxiliar para la Enseñanza. I. Tejero, Germán Carlos, colab. II. Ramos, Nadia Patricia, colab. III. Paradiso, Simon, colab. IV. Título.

CDD 005.4

Colaboradores:

German Carlos Tejero
Nadia Patricia Ramos
Antonio Retamar
Simon Paradiso

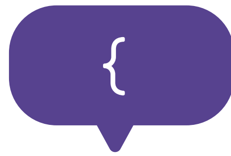
Créditos:

Ilustraciones de capítulos por Ana Longo
`https://www.behance.net/anitalongobucco`

Imagen de tapa y contratapa por Freeepik
`https://www.freepik.es`

Retoques de diseño y estilo en imágenes por Adrian Aguil Mallea `https://www.behance.net/adrianaguilmallea`

Maquetación por Paula Castillo
`http://paucast.com.ar`



Contenidos

Prólogo | 14

Objetivos | 17

Audiencia | 18

Agradecimientos | 19

Capítulo 1 : Análisis y diseño de algoritmos

Objetivo | 22

Definiciones de conceptos básicos | 22

Modelización de problemas del mundo real | 24

Etapas en la resolución de problemas con una computadora | 26

[-] análisis | 26

[-] diseño de la solución | 28

[-] especificación de algoritmos | 29

[-] escritura de programas | 32

[-] verificación | 35

Algoritmos, programas y lenguajes de programación | 36

[-] lenguaje de programación | 39

[-] herramientas | 41

[-] mi primer programa | 42

[-] sintaxis | 44

[-] semántica | 44

Ejercicios | 46

Capítulo 2 : Representación de datos

Objetivo | 50

Dato | 50

Variable | 52

[-] ámbito | 53
Tipo de dato | 53
[-] entero | 55
[-] decimal | 56
[-] lógico | 56
[-] caracter | 56
[-] texto | 57
Operaciones | 57
[-] operaciones de asignación | 58
[-] operaciones matemáticas | 58
[-] operaciones de comparación | 59
[-] operaciones lógicas | 60
[-] operaciones de concatenación | 61
Proposición | 61
[-] tablas de verdad | 64
Ejercicios | 68
Capítulo 3 : Entrada y salida de datos
Objetivo | 74
Variables, constantes y expresiones | 74
[-] constante | 75
[-] expresión | 76
La operación asignación | 77
Funciones internas | 78
Entrada y Salida | 82
[-] entrada | 84
[-] salida | 85
Ejercicios | 89
Capítulo 4 : Estructuras de control
Objetivo | 92
Flujo de ejecución | 92

Secuencia | 93
Selección | 93
Repetición | 98
[-] for | 98
[-] while y do-while | 99
Estructuras adicionales control | 103
[-] break | 103
[-] continue | 105
[-] switch | 106
[-] try-catch-finally | 108
[-] for-each | 109
Traza de ejecución | 110
Ejercicios | 113
Capítulo 5 : Estructuras de datos compuestas
Objetivo | 116
Datos compuestos | 116
Arreglos | 117
[-] vectores | 117
[-] matrices | 121
[-] arreglos n-dimensionales | 123
Ejercicios | 125
[-] vectores | 125
[-] matrices | 126
Capítulo 6 : Estructuras de programa
Objetivo | 132
Subproblemas | 132
Función | 133
[-] identificador | 134
[-] parámetros | 136
[-] variable local | 139

Librería | 141
Ejercicios | 143
Capítulo 7 : Recursión
Objetivo | 148
Recursión | 148
Ejercicios | 157
Capítulo 8 : Algoritmos fundamentales
Objetivo | 162
Algoritmos | 162
Búsqueda | 163
[-] lineal | 163
[-] binaria | 167
[-] métodos provisto por el lenguaje | 170
Ordenamiento | 171
[-] método de selección | 171
[-] método de burbuja | 174
[-] método de inserción | 176
[-] métodos previsto por el lenguaje | 179
Ejercicios | 180
Capítulo 9 : Paradigma orientado a objetos
Objetivo | 184
Paradigma | 184
Clase | 186
Objeto | 187
[-] constructor | 187
[-] el operador igualdad | 191
[-] el operador asignación | 193
[-] el valor nulo | 196
[-] contexto estático | 198

[-] acceso | 200

[-] mensajes | 202

[-] this | 205

[-] tipo de dato envoltorio | 207

Contenedores | 209

[-] lista | 211

Tipo abstracto de dato | 223

Ejercicios | 231

Capítulo 10 : Herencia y polimorfismo

Objetivo | 236

Herencia | 236

[-] subclases y superclases | 237

[-] sobrescritura de métodos | 245

Clase abstracta | 248

Interfaz | 253

[-] comparable | 257

[-] comparador | 263

Polimorfismo | 270

Genéricos | 279

Ejercicios | 283

Anexo | 287

Utilizar el entorno de desarrollo | 288

Bibliografía | 293

Índice de figuras

Fig.01 - modelo. | 25

Fig.02 - casos de usos. | 27

Fig.03 - diagrama de clases. | 29

Fig.04 - arquitectura de Von Neumann. | 37

Fig.05 - proceso de compilación e interpretación. | 40

Fig.06 - tabla de verdad - negación | 65

Fig.07 - tabla de verdad - conjunción | 65

Fig.08 - tabla de verdad - disyunción | 66

Fig.09 - entrada y salida. | 83

Fig. 10 - secuencia. | 93

Fig.12 - estructura de selección con alternativa. | 96

Fig.13 - estructura de repetición for. | 98

Fig.14 - estructura de repetición while. | 100

Fig.16 - for con break. | 104

Fig.17 - for con continue. | 105

Fig.18 - vector. | 117

Fig.19 - matriz. | 122

Fig.20 - partes de una función. | 134

Fig.21 - invocación de una función. | 137

Fig.22 - alcance. | 140

Fig.23 - factorial recursivo. | 151

Fig.24 - pila de llamadas. | 152

Fig.25 - búsqueda lineal o secuencial. | 164

Fig.26 - búsqueda lineal o secuencial ordenada. | 165

Fig.27 - búsqueda binaria. | 168

Fig.28 - búsqueda binaria vs secuencial. | 170
Fig.29 - ordenamiento por selección. | 172
Fig.30 - ordenamiento por burbuja. | 174
Fig.31 - ordenamiento por inserción. | 177
Fig.32 - clase y objetos. | 185
Fig.33 - memoria - creación de un objeto. | 188
Fig.34 - memoria - creación de un objeto con parámetros. | 190
Fig.35 - memoria - operador de igualdad ==. | 192
Fig.36 - memoria - asignación de otro objeto. | 195
Fig.38 - lista. | 212
Fig.39 - lista de objetos. | 215
Fig.40 - mapa. | 218
Fig.41 - mapa de objetos. | 222
Fig.42 - jerarquía de clases. | 238
Fig.43 - sobreescritura de métodos. | 245
Fig.44 - clase abstracta | 249
Fig.45 - interfaz. | 254
Fig.46 - comparable. | 259
Fig.47 - comparator. | 264
Fig. Anexo 01 - Pantalla inicial. | 288
Fig. Anexo 02 - Proyecto nuevo. | 289
Fig. Anexo 03 - Clase nueva | 289
Fig. Anexo 05 - Salida de la ejecución | 293

Prólogo

¿Por qué aprender a programar?

La tecnología está en todas partes. Desde el momento que suena el despertador, cuando miramos una serie, escuchamos un podcast, vemos una transmisión en vivo, usamos el celular y tantas cosas más que se nos pueden ocurrir. Todos los dispositivos que usamos a diario funcionan con software.

Programar es el arte de crear software. Es escribir las instrucciones que le dicen a una máquina qué hacer.

Aprender a programar nos permite:

- Entender cómo funcionan los dispositivos que usamos.
- Crear nuestras propias aplicaciones y páginas web.
- Ser más creativos e innovadores.
- Tener mejores oportunidades laborales.
- Desarrollar habilidades de pensamiento crítico y habilidades para la resolución de problemas.
- Trabajar de forma lógica y metódica.
- Mejorar nuestra comunicación y la manera en que colaboramos.
- Ser más independientes y autosuficientes.

Más allá de si queremos convertirnos en un programador profesional -un desarrollador- o simplemente para aprender algo nuevo, aprender a programar es una excelente inversión.

Programar está de moda. Cada vez hay más demanda de programadores en el mercado laboral. ¿Por qué? La respuesta es sencilla: la tecnología está en todas partes. Cómo dijimos al principio, desde que nos despertamos hasta que nos dormimos usamos dispositivos que funcionan con software.

La digitalización ha transformado nuestra vida. Miles de aplicaciones y plataformas ne-

cesitan ser programadas para funcionar y posibilitar cada vez más funciones. Además, la automatización está presente en sectores como la industria, la robótica y la inteligencia artificial, donde los sistemas necesitan realizar tareas de forma autónoma.

Por suerte, aprender a programar es más fácil que nunca. Hay muchas plataformas de educación en línea y recursos gratuitos disponibles. Esto ha hecho que las habilidades en programación y tecnología sean más accesibles, con un impacto significativo en el mercado laboral.

Si juntamos la alta demanda de programadores, los avances tecnológicos y la tendencia a la interdisciplinariedad, encontraremos en el mundo de la programación términos que aparecen en primera plana como Machine learning, Big data, Automatización, IoT, DevOps, BlockChain. Aunque lo cierto es, como sucede con todas las modas, que dos de ellos están dando mucho que hablar últimamente: “programador fullstack” e “inteligencia artificial”.

Un programador fullstack es una persona que es capaz de trabajar en todas las capas del desarrollo de una aplicación, en general web. Esto incluye desde la parte visual -la llama-

mos front-end- hasta la lógica del negocio y sus servicios -le decimos back-end.

Si bien existen cursos y especializaciones en fullstack, para dominar cualquiera de las dos áreas es fundamental tener conocimientos sólidos de programación. Es muy difícil, por no decir imposible, que un desarrollador sea productivo sin dominar los conceptos básicos de la programación. Ser productivo implica hacer las cosas de la mejor manera posible, utilizando los recursos de manera eficiente y en un tiempo razonable.

En general un programador que no logra ser productivo en un tiempo razonable, difícilmente será tenido en cuenta para futuras ocasiones. Por lo tanto se recomienda, como se suele decir, comenzar por el principio.

La inteligencia artificial (IA) es una rama de la informática que busca crear máquinas que puedan imitar la inteligencia humana. Esto incluye el aprendizaje, el razonamiento y la resolución de problemas.

En los últimos años, ha crecido la idea de que la IA reemplazará a los programadores. Lo cierto es que no es probable que suceda en un futuro cercano o incluso a largo plazo.

La IA es una herramienta que puede ayudar a los programadores a automatizar tareas repetitivas y acelerar el proceso de desarrollo. Sin embargo, los programadores siguen siendo necesarios para:

- Diseñar y desarrollar aplicaciones de calidad.
- Comprender las necesidades de los usuarios.
- Tomar decisiones críticas durante el proceso de desarrollo.

La IA no es un sustituto completo de la creatividad, el juicio o el criterio humano. Estas habilidades, fundamentales en los programadores, se desarrollan en gran parte a través de la experiencia y no son fácilmente replicables por una máquina.

La programación es un campo en constante evolución. Los programadores necesitan estar al día con las últimas tecnologías y tendencias para seguir creciendo.

La IA puede ayudar a los programadores a mantenerse actualizados y a ser más productivos. Sin embargo, no reemplaza la necesidad de tener una comprensión profunda de los principios fundamentales de la programación. La experiencia sigue siendo esencial en la creación de software de alta calidad.

Objetivos

El principal objetivo del libro es enseñar a programar, lo que implica también proporcionar habilidades y conocimientos necesarios para crear, desarrollar y utilizar software.

Se pretende:

Fomentar el pensamiento computacional. La programación enseña a las personas a pensar de manera lógica y a descomponer problemas complejos en problemas pequeños y manejables. El pensamiento computacional es una habilidad fundamental de la disciplina y además ayuda a resolver problemas en otras áreas.

Estimular la creatividad y la innovación. La programación permite a las personas crear nuevas soluciones, desarrollar aplicaciones y diseñar sistemas que resuelvan problemas y satisfagan necesidades específicas. Aprender a programar fomenta la creatividad.

Mejorar la resolución de problemas. La programación implica enfrentarse a desafíos y buscar soluciones eficientes. Al aprender a programar, las personas adquieren habilidades para abordar problemas de manera sistemática, analizar situaciones y encontrar soluciones lógicas y estructuradas.

Impulsar el pensamiento analítico. La programación requiere un enfoque analítico para descomponer un problema en componentes más pequeños y comprender cómo interactúan entre sí. Esto ayuda a desarrollar habilidades de pensamiento crítico y analítico que son valiosas en muchas áreas profesionales.

Prepararse para la era digital. Vivimos en una sociedad cada vez más digitalizada, donde la tecnología desempeña un papel importante en casi todos los aspectos de nuestras vidas. Aprender a programar nos prepara para comprender y participar en esta era digital, y nos brinda una herramienta adicional para adaptarnos a los avances tecnológicos, que por cierto son cada vez más “vertiginosos”.

Fomentar habilidades de colaboración y trabajo en equipo. Programar no es solo un esfuerzo individual, sino que por lo general implica colaborar con otras personas, sobre todo en proyectos de mediana o alta complejidad. Aprender a programar promueve la colaboración, la comunicación y el trabajo en equipo, habilidades esenciales en el entorno laboral actual.

Audiencia

Este libro ha sido escrito para una audiencia diversa. Servirá de guía para la introducción al mundo de la programación, sin importar la formación o experiencia previa.

A los estudiantes de una carrera afín a la disciplina, les servirá para afianzar los fundamentos de la programación y desarrollar el pensamiento computacional. Este libro proporciona una base sólida para el futuro profesional en el campo de la tecnología.

A los estudiantes autodidactas, les servirá para aprender a su ritmo, sin necesidad de contar con un docente o instructor guía. El estudiante autodidacta podrá saltar de capítulo en capítulo volviendo o yendo a los lugares que quiera explorar, aunque se recomienda el recorrido en orden, dado que la complejidad de los temas que se desarrollaron se encuentran ordenados de menor a mayor.

A los profesionales de la disciplina, les servirá para actualizar sus conocimientos sobre el paradigma y quizá repasar algún detalle olvidado de los objetos. Por supuesto siempre utilizando JAVA como lenguaje.

A los profesionales de otras áreas o simplemente a quienes quieran aprender a programar por curiosidad, o para mejorar sus habilidades digitales, este libro les ofrece una introducción completa a los fundamentos de la programación, con explicaciones claras y concisas de los conceptos clave, ejemplos prácticos y ejercicios para poner en práctica los conocimientos.

Este libro te ofrece:

- Una introducción completa a los fundamentos de la programación.
- Explicaciones claras y concisas de los conceptos clave.
- Ejemplos prácticos y ejercicios para poner en práctica los conocimientos.
- Una guía completa para el aprendizaje autodidacta.
- Recursos adicionales para profundizar el aprendizaje.

Agradecimientos

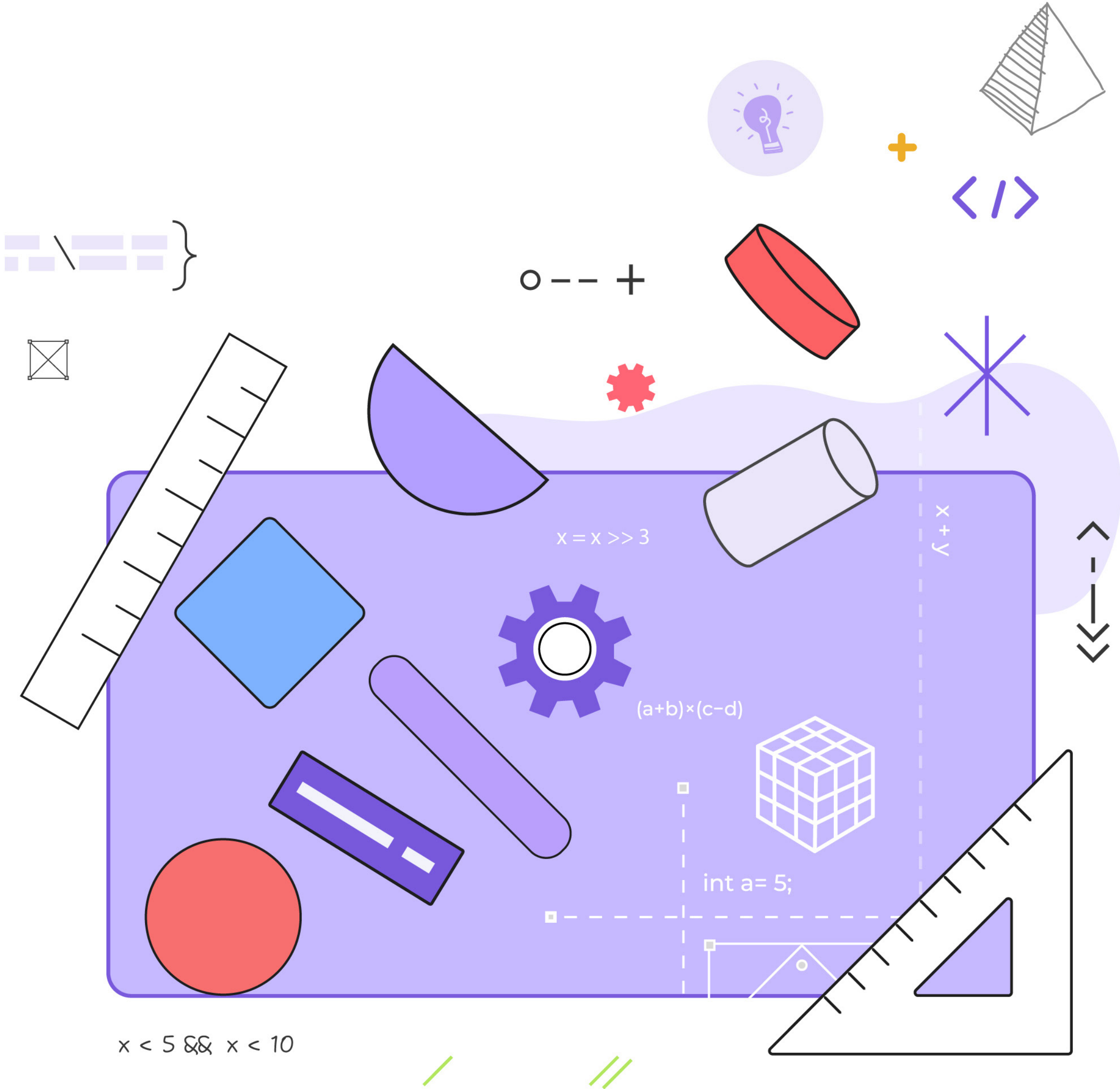
Cuando decidí continuar lo que había empezado hace un tiempo y darle forma de libro, no sabía realmente el tiempo que me llevaría, por eso quiero comenzar agradeciendo a mi familia, que se bancaron todo el proceso. A mis hijos, aceptando que el tiempo dedicado al proyecto es también para ellos, aunque sé que en el fondo no terminan de comprender del todo, esta es una inversión y es para ellos. A mi "pareja" -le encanta que la presente de ese modo-, que con todo su amor siempre está acompañando el plan, poniendo el pecho a todas las circunstancias que se presentan. A mis viejitos, por su amor desde siempre y creer en lo que hago. A mis "brokis" por estar ahí para escucharme y animarme.

Un agradecimiento muy especial a mi mentora, mi profesora, mi colega, Beatriz Depetris. Fue la primera persona que confió en que me convertiría algún día en un "buen" profesional. Sus consejos, experiencia , paciencia y cariño han sido fundamentales para mi crecimiento profesional y personal. Agradezco profundamente su confianza en mí y por haberme brindado la oportunidad de aprender.

A mis amigos, colegas, profesores y a todas las personas que han estado presentes en este proceso. Agradezco su apoyo y motivación. A cada uno, sea a través de palabras cálidas, palabras que muchas veces me han movilizado y ayudado, o inclusive palabras de esas que son más difíciles de digerir pero que me han servido también, las palabras de crítica.

Capítulo 1

Análisis y diseño de algoritmos



Capítulo 1
Análisis y diseño de algoritmos

Objetivo

El capítulo tiene como objetivo proporcionar las definiciones fundamentales y necesarias para adquirir una comprensión inicial de la disciplina, en particular sobre el análisis y diseño de algoritmos. Se abordarán aspectos relacionados con la resolución de problemas utilizando computadoras, explicando el por qué de la necesidad de utilizar una metodología y contar con pasos bien definidos. Además se explicarán en detalle las herramientas que se utilizarán para la creación de programas.

Al concluir este capítulo, estarás preparado para realizar y ejecutar tu primer programa, consolidando así los conocimientos adquiridos.

Definiciones de conceptos básicos

La informática es la ciencia que estudia el análisis y resolución de problemas utilizando computadoras.

Si bien todos saben lo que es una computadora, es interesante definirla con algo más de profundidad dado que ello nos permitirá explorar poco a poco cómo se relaciona con la programación. Entonces, podemos definir a una computadora como una máquina digital y sincrónica, con cierta capacidad de cálculo numérico y lógico, controlada por programas almacenados.

Una computadora es digital porque sólo utiliza dos estados -prendido (1) y apagado (0)- para representar y procesar la información, y sincrónica porque todas las operaciones se realizan en intervalos regulares -en una misma cantidad de tiempo- intervalos que se encuentran controlados por un reloj central.

Los programas almacenados, que controlan la máquina, no son ni más ni menos que un conjunto de instrucciones. Las instrucciones son acciones que serán ejecutadas por la computadora, que en definitiva son el conjunto de acciones que permitirán realizar una función específica.

La ciencia informática, también conocida como computación, es una ciencia que abarca

una amplia gama de temas relacionados con el tratamiento automático de la información. Su objetivo principal es desarrollar métodos y técnicas para almacenar, procesar y transmitir datos de forma eficiente.

La ciencia informática: Un mundo de posibilidades

Imaginemos por un instante a las computadoras como si fueran personas, pero con capacidades y funciones específicas. Cada punto que a continuación mencionamos sirve para reflexionar acerca de los temas que se pueden desarrollar, profundizar o investigar teniendo como eje principal la computación.

Al igual que el cuerpo humano, la computadora tiene una estructura física que la sostiene y le da forma, la llamamos hardware. Esta estructura está compuesta por distintos componentes, como la placa madre, el procesador, la memoria, el disco rígido y otros elementos que trabajan en conjunto para permitir el funcionamiento del sistema.

Así como los sentidos nos permiten percibir el mundo exterior, las computadoras tienen canales de entrada que les permiten recibir información del entorno. Estos canales incluyen el teclado, el mouse, la cámara, el micrófono, entre otros dispositivos que capturan datos y los convierten en señales que la computadora puede procesar.

El cerebro es el centro de control del cuerpo humano, y de manera similar, las computadoras tienen un procesador que se encarga de ejecutar las instrucciones y realizar los cálculos necesarios para llevar a cabo las tareas.

El sistema nervioso conecta las distintas partes del cuerpo humano y transmite señales que permiten coordinar los movimientos y las acciones. De forma análoga, las computadoras tienen un canal de datos que conecta los diferentes componentes y permite la transferencia de información entre ellos.

La memoria almacena recuerdos y experiencias, de forma similar la memoria de la computadora guarda datos e información. Así como nuestra memoria se divide en aquella que retenemos por largos períodos y otros que duran muy poquito, la memoria de la computadora también. La memoria RAM almacena los datos que la computadora está utilizando, mientras que el disco rígido y otros dispositivos de almacenamiento guardan los datos para ser recuperados en cualquier otro momento.

Los humanos nos comunicamos a través del lenguaje, y las computadoras también tienen sus propios lenguajes de programación. Estos lenguajes permiten dar instrucciones a la computadora y controlar su comportamiento. Existen muchos lenguajes de programación, cada uno con sus propias características y aplicaciones.

Las personas aprenden y adquieren nuevas habilidades a lo largo de toda su vida, de forma

similar las computadoras también pueden “aprender” mediante la inteligencia artificial. El aprendizaje automático y el aprendizaje profundo son técnicas que permiten a las computadoras analizar datos, identificar patrones y tomar decisiones de forma autónoma.

Más allá de estas analogías entre las computadoras y las personas, la ciencia informática abarca muchos más temas que sólo los relacionados con una computadora y sus “partes”. Veamos algunos de ellos como para destacar la gran variedad de aristas que la disciplina también permite desarrollar, profundizar o investigar:

- Redes de computadoras: cómo conectar dispositivos entre sí para compartir información y recursos.
- Seguridad informática: cómo proteger la información y los sistemas de accesos no autorizados.
- Interacción humana: cómo diseñar interfaces que sean fáciles de usar y que permitan una interacción natural entre el usuario y el dispositivo.
- Computación gráfica: cómo generar imágenes y videos realistas.
- Bioinformática: cómo utilizar la informática para analizar e interpretar datos biológicos y médicos.
- Robótica: cómo crear robots que puedan interactuar con el mundo físico de forma inteligente.

Modelización de problemas del mundo real

La resolución de problemas utilizando una computadora es una habilidad poderosa que nos permite automatizar tareas, analizar datos y tomar decisiones más inteligentes. Sin embargo, para que la computadora pueda comprender y resolver un problema, primero debemos traducirlo a un lenguaje que la máquina pueda entender.

Imaginemos que queremos usar una computadora para calcular la ruta más rápida entre dos ciudades. En el mundo real, hay muchos factores que pueden influir en la elección de la ruta, por ejemplo el tráfico, las condiciones climáticas del momento, si es de tierra o se encuentra pavimentada o inclusive las preferencias de quien maneja.

Para modelar este problema en la computadora, primero debemos abstraernos de aquellos aspecto o elementos que no son relevantes y quedarnos con los que sí lo son, con los aspectos esenciales del problema. Esto significa identificar los elementos que son importantes para la solución, como las ciudades de origen y destino, las conexiones posibles entre ellas, o sea las rutas, y las restricciones de tiempo.

Luego, debemos simplificar la representación de estos elementos. Por ejemplo, podemos repre-

sentar las ciudades como puntos en un mapa y las rutas como líneas que conectan estos puntos. También podemos pensar en representar el tiempo de viaje, la distancia y otros factores relevantes.

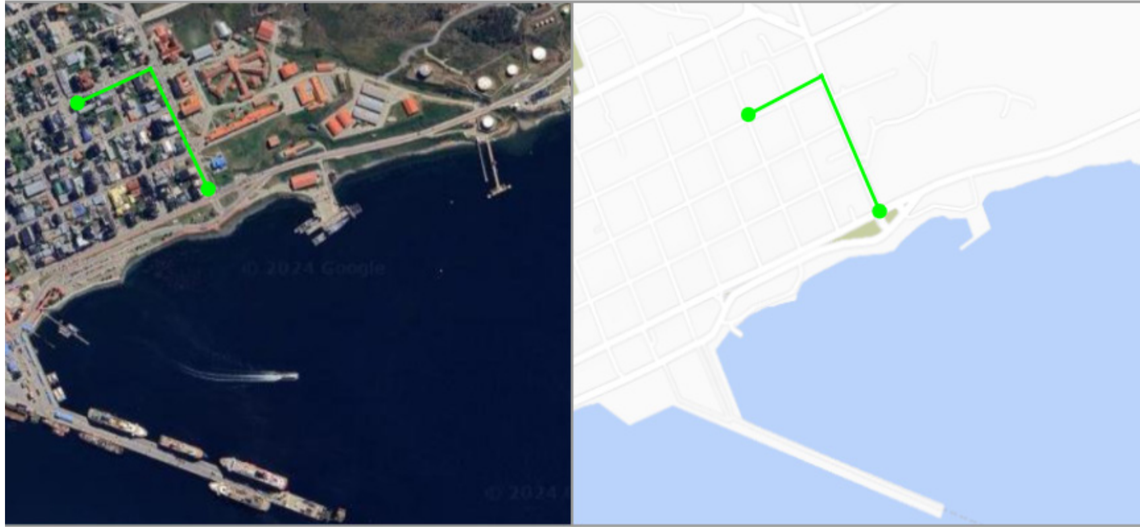


Fig.01 - modelo.

El resultado de este proceso es un modelo, que básicamente será la representación simplificada del problema del mundo real. El modelo puede ser una fórmula matemática, un diagrama, un programa de computadora o cualquier otra forma que facilite la comprensión y la solución del problema.

La modelización es un proceso creativo que requiere una buena comprensión del problema, del contexto en el que se presenta y de las capacidades de la computadora.

Es importante encontrar el equilibrio adecuado entre la simplicidad del modelo y su capacidad para representar los aspectos relevantes del problema.

Algunos beneficios de utilizar modelos:

- Comunicación clara: permite comunicar el problema de forma precisa y concisa.
- Análisis y solución: facilita el análisis del problema y la búsqueda de soluciones eficientes.
- Reutilización: los modelos pueden ser reutilizados para resolver problemas similares.
- Predicción: permite realizar predicciones sobre el comportamiento del sistema modelado.

Etapas en la resolución de problemas con una computadora

En esta actividad, la de programar, como en cualquier otra disciplina que implique la resolución de problemas, es fundamental contar con pasos bien definidos. Esto se debe a que un enfoque estructurado y organizado ofrece numerosas ventajas que luego ayudarán a realizar “mejores” programas.

Al seguir un conjunto de pasos predefinidos, evitamos perder tiempo y esfuerzo en tareas innecesarias o redundantes. Nos enfocamos en lo que realmente importa, optimizando el proceso y reduciendo el tiempo de desarrollo.

Un buen conjunto de pasos nos asegura que no olvidaremos ningún aspecto importante del problema. Abordamos todas las aristas, minimizando la posibilidad de errores o soluciones incompletas.

Dividir el problema en etapas más pequeñas y manejables nos permite visualizarlo mejor. Tendremos una comprensión más profunda del mismo y podremos identificar las relaciones entre las diferentes partes.

Si la solución no funciona como esperábamos, una estructura clara nos ayudará a rastrear el error de forma más rápida y eficiente. Podremos identificar la causa del problema y encontrar la solución más adecuada.

Los pasos que seguiremos en la solución de problemas utilizando una computadora serán, por ahora, el análisis, el diseño, la especificación, la escritura y finalmente la verificación de nuestra solución.

[-] análisis

La etapa de análisis es fundamental en la programación y, en general, en la resolución de problemas. Es aquí donde se define el problema a resolver y se establecen las bases para la creación del programa.

En esta etapa realizaremos las siguientes actividades:

- Comprensión del problema: se analiza el problema en su contexto, en el mundo real, identificando las necesidades del usuario, las limitaciones del sistema y los objetivos a alcanzar.
- Recopilación de requisitos: se obtienen los requisitos del usuario a través de entrevistas, encuestas, análisis de documentos u otros mecanismos. Los requisitos es lo que se espera que haga el programa, lo que debe hacer el sistema.
- Modelado del problema: se crea un modelo del problema, o también llamado modelo del ambiente, que representa el entorno, los datos, las operaciones y el objetivo final.

- Definición de la solución: se establecen las características y funcionalidades del programa que se desarrollará. Es el alcance que tendrá el sistema.

Un componente importante de este modelo son los datos a utilizar y las transformaciones que llevarán al objetivo.

Algunas herramientas que suelen utilizarse en el análisis, bajo el paradigma orientado a objetos, son los casos de uso, los diagramas de clase preliminares y diagrama de secuencias, entre otras.

Estas herramientas no las vamos a profundizar, sólo se mencionan para tener una idea de que cada etapa se puede desarrollar aún con mayor profundidad.

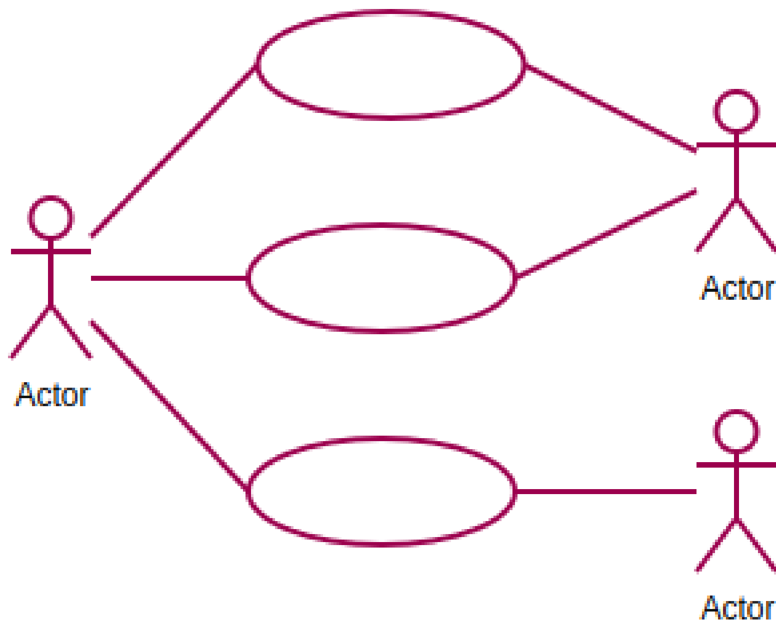


Fig.02 - casos de usos.

Podemos decir que la etapa de análisis es la etapa del "qué"

- ¿Qué problema se quiere resolver?
- ¿Qué necesita el usuario?
- ¿Qué funcionalidades debe tener el programa?
- ¿Qué datos se necesitan para el programa?
- ¿Qué hará en definitiva el programa?

Veamos algunos ejemplos para ver qué se podría realizar en cada actividad:

- Desarrollar una calculadora:
  Comprensión del problema → se analiza qué operaciones matemáticas se suelen realizar con una calculadora. Cuáles son las operaciones más frecuentes. ¿Qué tipo de calculadora quiere realizarse, científica o básica?.

  Recopilación de requisitos → se consulta al usuario qué tipo de cálculos se pretenden realizar, se determina si será de tipo básica, científica o ambas, y qué funciones adicionales le gustaría tener (porcentaje, memoria).

  Modelado del problema → se crea un modelo que represente los diferentes tipos de operaciones, los números y el resultado.

  Definición de la solución → se define cómo será la calculadora, qué operaciones matemáticas se podrán realizar, cuáles serán las funciones adicionales solicitadas por el usuario que ingresan en este desarrollo.

- Creación de una agenda personal: se analiza cómo el usuario gestiona actualmente sus eventos, reuniones, recordatorios y tareas. Se entrevista al usuario para conocer sus necesidades con respecto a la agenda. La visualización de eventos, ¿como la piensa?, ¿se debe sincronizar con otros dispositivos?. Se crea un modelo que representa los diferentes tipos de eventos, las tareas, el calendario y las notificaciones. Se define el alcance de la agenda, que permite al usuario organizar sus eventos, crear tareas, establecer recordatorios y sincronizar la información con otros dispositivos.

- Un juego de adivinanzas: se analiza cómo funciona un juego de adivinanzas tradicional. Se define el tipo de palabras que se utilizarán en el juego, por ejemplo animales, objetos y personajes. Se define el número de intentos que tendrá el jugador. Se crea un modelo que representa la palabra a adivinar, las letras disponibles y el estado del juego. Se define el alcance del juego de adivinanzas, por ejemplo qué se le permite al jugador para adivinar la palabra secreta y la cantidad de intentos que tendrá, entre otras reglas.

[-] diseño de la solución

La etapa de diseño es donde se empieza a concretar la solución al problema planteado, analizado en la etapa anterior. Aquí se definen cómo se piensan realizar las características y funcionalidades del programa, así como la estructura interna y su arquitectura general.

En esta etapa se realizan las siguientes actividades:

- Modularización del problema: se divide el problema en subproblemas más pequeños y manejables.
- Diseño de la arquitectura: se define la estructura general del programa, incluyendo los

diferentes módulos y componentes, con sus relaciones.

- Diseño de las interfaces: se define cómo los usuarios interactúan con el programa.
- Diseño del soporte de los datos: se define el soporte a los datos que deberá gestionar el programa. Por ejemplo, la estructura de la base de datos que se utilizará.

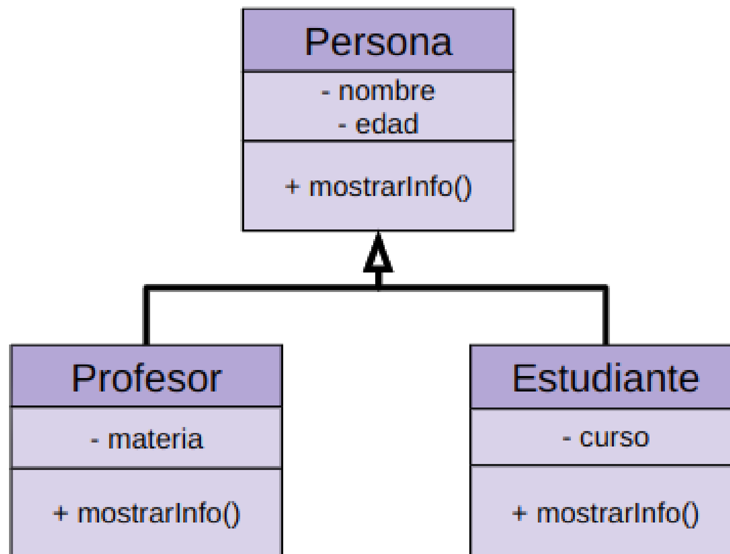


Fig.03 - diagrama de clases.

Podemos decir que la etapa de diseño es dónde se traza el plan para llegar a la solución, es la etapa del "cómo". Cómo se va a resolver el problema.

Algunas herramientas que suelen utilizarse en la etapa de diseño, bajo el paradigma orientado a objetos, son los diagramas de clase, diagramas de secuencias y modelo de datos, entre otras.

[-] especificación de algoritmos

Se detallan los pasos para la solución.

En la etapa de especificación de algoritmos se definen los pasos precisos que se seguirán para resolver cada subproblema del problema original.

Se realizan las siguientes actividades:

- Análisis del algoritmo: se evalúa la eficiencia del algoritmo y se comparan diferentes opciones.
- Selección del algoritmo: se elige el algoritmo más adecuado para cada subproblema, teniendo en cuenta la eficiencia, la complejidad y la disponibilidad de recursos.
- Validación del algoritmo: se prueba el algoritmo con casos de prueba para verificar que funciona correctamente.

La elección del algoritmo adecuado es fundamental para garantizar la eficiencia de la solución.

Un algoritmo es una secuencia ordenada y finita de pasos elementales, no ambiguos, que permiten resolver un problema.

Podemos pensar en los algoritmos como si fueran recetas, que del mismo modo que una receta nos brinda una serie de instrucciones precisas y con un orden específico, el algoritmo tendrá un cantidad arbitraria de instrucciones que se ejecutarán en un orden determinado y que en su conjunto lograrán que la máquina realice alguna actividad.

Veamos un ejemplo a través de una receta típica y pensemos los pasos que hay que realizar para hacer un bizcochuelo:

1. Precalentar el horno
2. Separar las yemas
3. Batir las claras
4. Agregar el azúcar
5. Agregar las yemas
6. Incorporar la harina
7. Verter la mezcla en un molde
8. Hornear durante 25-30 minutos
9. Desmoldar

Si tomamos esta receta de 9 pasos y la compartimos con alguien que ya tiene experiencia de como hacer un bizcochuelo, podría servir de guía tranquilamente. Ahora bien, siguiendo

con este mismo ejemplo, podemos ver a las máquinas como aquellas personas que no tienen experiencia alguna para hacer un bizcochuelo y tendríamos un problema.

Una máquina se asemeja a una persona sin experiencia, hará lo que se le dice que haga aunque al no contar con información detallada tendrá que tomar algunas decisiones que, justamente por su simpleza o falta de experiencia, puede tomar malas decisiones y hacer algo que no se ha contemplado.

Volvamos a leer la receta pero ahora pensando que no contamos con ninguna experiencia, ¿qué sucede?. ¡Exacto! Es muy difícil que alguien que no sepa cómo hacer un bizcochuelo lo logre, o al menos le salga "rico". Esto se debe a que los pasos cuentan con poco detalle, o dicho de otro modo, las instrucciones son ambiguas.

El algoritmo, por definición, debe contar con pasos bien definidos. Cuando leemos el 1er paso, "precalentar el horno", ¿qué entendemos? ¿Hay que calentarlo a cuantos grados? ¿el tiempo que lo calentamos, es importante?. Tenemos que lograr que los pasos carezcan de ambigüedad.

Intentemos nuevamente generar la receta con algo más de detalle, el algoritmo para la creación del bizcochuelo presenta los siguientes pasos:

1. Precalentar el horno
    - Encender el horno y esperar que esté a 180°C.
      Este paso es crucial para que el bizcochuelo se hornee de manera uniforme y se eleve correctamente.

2. Separar las yemas
    - Romper los huevos cuidadosamente y separar las yemas de las claras.
      Es importante que las claras no contengan restos de yema, ya que esto podría afectar su capacidad para batirse.

3. Batir las claras
    - En un recipiente limpio y seco, batir las claras a punto de nieve.
      El punto nieve significa que las claras batidas deben estar firmes.

4. Agregar el azúcar
    - Con una espátula, incorporamos el azúcar a las claras batidas en forma de lluvia, mezclando suavemente con movimientos envolventes.

      Es importante hacerlo con cuidado para evitar que las claras pierdan volumen.

5. Agregar las yemas
   - Añadir las yemas, una a una, a la mezcla de claras y azúcar, batiendo suavemente después de cada incorporación.
6. Incorporar la harina
   - Verter la harina junto con la sal sobre la mezcla de claras, yemas y azúcar.
7. Verter la mezcla en un molde
   - Engrasar y enharinar un molde para horno.
   - Verter la mezcla de bizcochuelo en el molde.
8. Hornear durante 25-30 minutos
   - Introducir el molde en el horno precalentado y hornear durante 25-30 minutos o hasta que un palito o cuchillo al ser insertado en el centro salga limpio.
9. Desmoldar
   - Retirar el bizcochuelo del horno y lo dejamos enfriar en el molde durante unos minutos.

Como se puede apreciar, ahora, la receta tiene mucho más detalle. Podemos decir que a los pasos se le ha quitado el componente de ambigüedad que tenían, o al menos, decir que hemos reducido la ambigüedad.

Por definición, un lenguaje de programación para ser considerado como tal, no deberá contener instrucciones ambiguas, por lo tanto nos restará indicar en detalle lo que queramos hacer.

Más adelante veremos los componentes importantes que nos están faltando para definir nuestro algoritmo de forma completa, la pre y post condición del algoritmo.

[-] escritura de programas

Traduciendo la lógica a código

Hasta aquí hemos logrado definir los pasos que se deben seguir y la información que se necesita. Ahora es el momento de dar vida al algoritmo, convirtiéndolo en un programa que una computadora pueda entender y ejecutar.

Este proceso se llama escritura de programas y consiste en traducir la lógica del algoritmo a código escrito en un lenguaje de programación específico.

Cada lenguaje de programación tiene su propio conjunto de reglas y símbolos, como si fuera un idioma diferente. Los programadores aprenden estos lenguajes para poder escribir código que la computadora pueda interpretar. Este código es el que se denomina “código fuente”.

La escritura de programas implica una serie de actividades:

- Selección del lenguaje de programación: si bien es una actividad que suele realizarse en etapas anteriores, en particular en la etapa de diseño, hay ocasiones en las que la elección se posibilita recién en esta etapa. Hay muchos lenguajes de programación disponibles, cada uno con sus propias características y ventajas. La elección del lenguaje más adecuado dependerá de varios factores, como la complejidad del programa, la disponibilidad de recursos y por supuesto muchas veces la elección depende del programador, de su experiencia.
- Escritura del código fuente: el código fuente es un conjunto de instrucciones escritas en un lenguaje de programación que la computadora puede entender. El código fuente se organiza en archivos y la mayoría de las veces esos archivos son acompañados en el nombre con una extensión específica, como .py para Python, .java para Java y .js para JavaScript. Cabe la aclaración que esta extensión es para que nosotros podamos rápidamente identificar de qué se trata.
- Pruebas y depuración: es importante probar el programa con diferentes casos de prueba para detectar y corregir errores. La depuración es el proceso de identificar y corregir los errores en el código fuente. Existen herramientas de depuración que ayudan a los programadores a encontrar y corregir errores de forma más eficiente.
- Documentación del programa: la documentación del programa es una descripción clara y concisa del código fuente. Facilita la comprensión y el mantenimiento del programa. La documentación debe incluir información sobre el propósito del programa, la estructura del código y los algoritmos implementados.

Además de JAVA, el lenguaje que vamos a utilizar durante el desarrollo de todos los temas que vienen por delante, existen otros también bastante populares:

- Python
- C# (se pronuncia C sharp)
- TypeScript (superset de javascript)
- C++ (se pronuncia C más más)

A modo de ejemplo se muestra como sería mostrar por pantalla la frase “yo programo” con cada uno de los lenguajes mencionados:

El archivo para el lenguaje C++ podría llamarse “ejemplo.cpp”

```
#include <iostream>

int main() {
  std::cout << "yo programo" << std::endl;
  return 0;
}
```

El archivo para el lenguaje C# podría llamarse “ejemplo.cs”

```
using System;

public class Program {
    public static void Main(string[] args) {
        Console.WriteLine("yo programo");
    }
}
```

El archivo para el lenguaje TypeScript podría llamarse “ejemplo.ts”

```
console.log("yo programo");
```

El archivo para el lenguaje python podría llamarse “ejemplo.py”

```
print("yo programo")
```

[-] verificación

¿Tu programa funciona como debería?

Imaginemos que hemos escrito un programa para calcular el área de un triángulo. Hemos seguido las instrucciones, elegido el lenguaje de programación adecuado y escrito el código fuente y el mismo no presenta errores aparentes. Pero, ¿cómo podemos estar seguros de que el programa funciona correctamente?

Aquí es donde entra en juego la verificación. La verificación consiste en probar el programa con diferentes datos para asegurarnos de que el programa produce el resultado que nos han especificado, que nos han requerido.

Para realizar la verificación se pueden realizar las siguientes actividades:

- Definir qué significa "correcto": ¿Qué resultados se espera que produzca el programa para cada tipo de entrada?. Definir casos de prueba que representen diferentes escenarios posibles.
- Seleccionar datos de prueba: seleccionar datos que sean representativos del problema real a resolver. Incluir casos de prueba que cubran diferentes situaciones, como valores límite, valores negativos y valores "raros" o no esperados que son lo que podrían generar problemas a los algoritmos.
- Ejecutar el programa con los datos de prueba: comparar los resultados del programa con los resultados esperados.
- Repetir el proceso con diferentes conjuntos de datos: es importante probar el programa con una variedad de datos para aumentar la confianza de que hemos realizado el programa correcto. Cuantos más casos de prueba se ejecuten, mayor será la seguridad de que el programa funciona correctamente.

Es importante destacar que la verificación no puede garantizar que el programa es correcto en su totalidad. Siempre existe la posibilidad de que haya errores que no se detecten con los casos de prueba.

Como ya lo dijo Bertrand Meyer, en su libro "Object-Oriented Software Construction", la ausencia de errores demostrables no implica la ausencia de errores.

La verificación es una herramienta fundamental para aumentar la calidad del software.

Algoritmos, programas y lenguajes de programación

Para entender cómo funcionan, en términos generales, los programas primero debemos entender la arquitectura de una computadora. Esto es importante dado que establece la base sobre la cual se ejecutan y operan.

La arquitectura de una computadora se divide en varias partes, siendo las principales de ellas las siguientes:

- Unidad central de proceso (CPU): es el "cerebro" de la computadora. Realiza las operaciones, los cálculos y el procesamiento de los datos que los programas requieren durante su ejecución.
- Unidad de control: es la parte de la computadora que se encarga de llevar adelante la ejecución de los distintos programas. Lee las instrucciones, las decodifica y las ejecuta.
- Memoria: es donde se almacenan los datos y las instrucciones que la CPU necesita para trabajar. Hay diferentes tipos de memoria, la memoria volátil que al dejar de recibir energía pierde su estado y la memoria persistente que por más que pierda la energía el estado se mantiene. La más conocida del primer tipo es la memoria de acceso aleatorio -RAM, random access memory- y la del segundo tipo es el disco rígido.
- Dispositivos de entrada: son los medios a través de los cuales se ingresan datos y comandos a la computadora. En general la entrada se produce a través de los dispositivos como el teclado y el mouse.
- Dispositivos de salida: son los medios a través de los cuales la computadora muestra o proporciona resultados al usuario. Es habitual ver los resultados a través de los dispositivos como el monitor, los parlantes o inclusive la impresora.
- Bus de datos: es la vía de comunicación interna que permite la transferencia de datos entre diferentes componentes de la computadora, como la CPU, la memoria y los dispositivos de entrada o salida.

Las partes mencionadas son los elementos básicos del modelo que definió Von Neumann en el año 1945. Al día de hoy, con algunas diferencias desde su creación, se continúa utilizando para el desarrollo de computadoras.

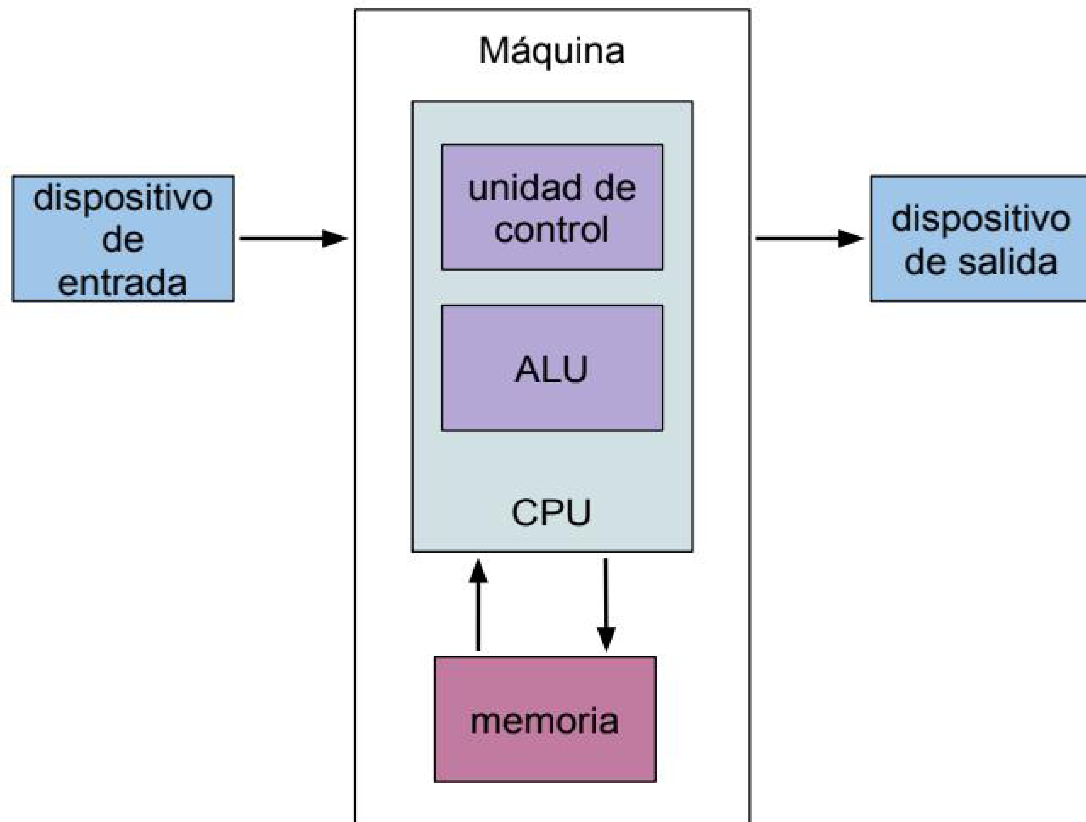


Fig.04 - arquitectura de Von Neumann.

La arquitectura define la forma en que los datos y las instrucciones son representados y manipulados internamente. En un nivel más bajo, la computadora utiliza el sistema binario para codificar y almacenar datos y programas.

El sistema binario utiliza sólo dos dígitos, el 0 y el 1, para representar la información. Cada dígito binario se llama "bit", y un conjunto de 8 bits se llama "byte".

El código binario es utilizado por la CPU para ejecutar instrucciones de programa y realizar operaciones aritméticas y lógicas.

Los programas escritos en lenguajes de programación de alto nivel, que escribiremos nosotros, son traducidos a código binario mediante un proceso llamado compilación o interpretación, o más genérico aún, traducción. Estos programas en código binario son luego cargados en la memoria y ejecutados por la CPU.

En la arquitectura de Von Neumann, la computadora almacena tanto los programas como los datos en la misma memoria. Los programas son la base del funcionamiento de las

computadoras. Se componen de algoritmos, como hemos definido anteriormente, secuencias ordenadas de pasos que se ejecutan para resolver un problema específico.

La arquitectura define cómo se ejecutan estos algoritmos. Los programas se almacenan en la memoria principal junto con los datos. La unidad central de procesamiento lee las instrucciones del programa desde la memoria y las ejecuta una por una.

Cada instrucción del programa le dice a la CPU qué operación realizar. Estas operaciones pueden ser:

- Aritméticas: sumar, restar, multiplicar y dividir.
- Lógicas: comparar valores, determinar si una condición es verdadera o falsa.
- Control de flujo: decidir qué instrucción ejecutar a continuación.

Las precondiciones y postcondiciones de un algoritmo son relevantes para comprender su funcionamiento.

Son dos conceptos fundamentales en el diseño de algoritmos. Ayudan a garantizar que los algoritmos se ejecuten correctamente y produzcan los resultados esperados.

La precondición define las condiciones que deben cumplirse antes de que se ejecute un algoritmo. Es importante verificar que la precondición se cumpla antes de ejecutar el algoritmo. Si la precondición no se cumple, el algoritmo puede no funcionar correctamente o incluso puede producir resultados erróneos.

Ejemplos:

- Un algoritmo para calcular la raíz cuadrada de un número requiere que el número sea positivo.
- El algoritmo para calcular el área del triángulo debe tener dos lados con longitud positiva.
- Volviendo a la receta del bizcochuelo, sería necesario contar con los ingredientes y más específicamente la cantidad de cada uno. Además de los materiales para su elaboración. Todo esto sería la precondición de la receta para poder comenzar a ejecutarla.

La postcondición define las condiciones que se cumplirán después de que se ejecute un algoritmo. La postcondición ayuda a verificar que el algoritmo haya producido los resultados esperados.

Ejemplos:

- Un algoritmo para calcular la raíz cuadrada de un número debe retornar un valor no negativo.

- Un algoritmo para calcular el área del triángulo debe retornar un valor no negativo.
- La receta del bizcochuelo, si hemos cumplido con la precondición y hemos ejecutado cada paso, entonces tendremos un bizcochuelo.

La precondición y la postcondición están relacionadas entre sí. La postcondición sólo puede cumplirse si la precondición se cumple antes de ejecutar el algoritmo.

[-] lenguaje de programación

Como hemos definido anteriormente, los lenguajes de programación son herramientas que nos permiten comunicarnos con las computadoras. Son el lenguaje que utilizamos para darles instrucciones precisas sobre qué queremos que hagan.

Las computadoras no entienden nuestro lenguaje natural, sólo comprenden un lenguaje binario, compuesto por ceros y unos. Los lenguajes de programación actúan como un traductor, convirtiendo nuestras ideas en instrucciones que la máquina puede entender y ejecutar.

Existen cientos de lenguajes de programación, cada uno con sus propias características y aplicaciones, aunque en este camino utilizaremos para desarrollar los ejercicios y avanzar conceptos claves de la disciplina, el lenguaje de programación Java.

Java es uno de los lenguajes de programación más populares y ampliamente utilizados en la industria del software, lo que lo convierte en una excelente opción para enseñar programación, inclusive a quien comienza.

Algunos de los puntos en los que el lenguaje se destaca sobre otros:

Sintaxis clara y fácil de entender: su sintaxis es clara, fuertemente tipada y estructurada, lo que facilita su aprendizaje. Los términos “fuertemente tipado” y “estructurado” los profundizaremos más adelante, por ahora diremos que fomentarán las buenas prácticas de programación, como claridad, legibilidad y modularidad.

- Es orientado a objetos: significa que está diseñado para representar objetos del mundo real en el código. Esto ayuda a los estudiantes a comprender mejor los conceptos de la programación orientada a objetos.
- Amplia comunidad: Java tiene una gran comunidad de desarrolladores, lo cual es útil al momento de lidiar con diversas situaciones y una amplia variedad de recursos de aprendizaje, como tutoriales en línea, libros y videos.
- Portabilidad: el código escrito en Java se puede ejecutar en diferentes plataformas, sistemas operativos, sin la necesidad de reescribir.
- Amplio contexto para desarrollo: desde aplicaciones de escritorio hasta aplicaciones

web y móviles. Esto permite a los estudiantes explorar diferentes áreas de la programación y ver cómo se aplica Java en diferentes contextos.

Java se diferencia de otros lenguajes por su proceso de ejecución en dos fases, esto es una fase de compilación y otra de interpretación. Para ejecutar los programas requiere de una máquina virtual denominada Java Virtual Machine (JVM).

El código fuente escrito en Java se compila a un lenguaje intermedio llamado bytecode.

Este código intermedio se almacena en archivos con extensión .class. Es el conjunto de instrucciones que la JVM puede interpretar.

La JVM está disponible en diferentes plataformas, lo que permite que los programas Java sean portables y se ejecuten en cualquier sistema operativo que tenga una JVM instalada.

La JVM también se encarga de la gestión de memoria, la seguridad y otras tareas importantes para el correcto funcionamiento del programa.

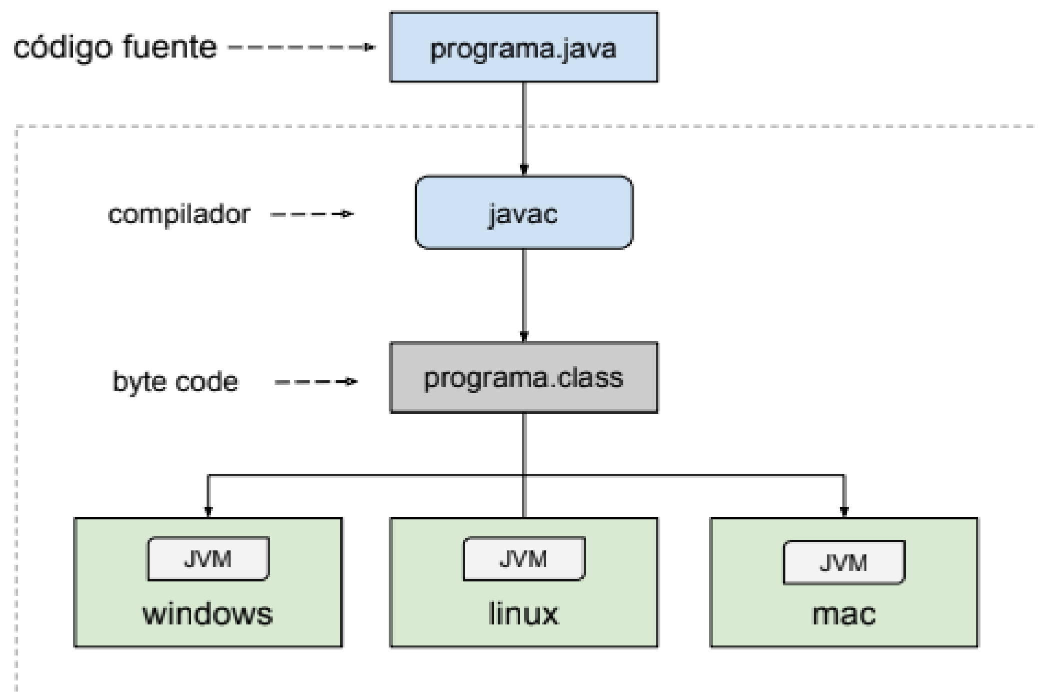


Fig.05 - proceso de compilación e interpretación.

[-] herramientas

Todos los ejemplos y ejercicios resueltos en este documento han sido desarrollados bajo el sistema operativo linux, en particular la distribución Ubuntu, aunque al utilizar java como lenguaje no habría inconveniente en que funcionen en otros sistemas operativos, por ejemplo bajo el sistema operativo windows.

INTÉRPRETE

Para hacer funcionar los programas deberemos contar con el kit de desarrollo de Java, si bien existen varias implementaciones sugerimos utilizar el kit de desarrollo abierto OpenJDK.

OpenJDK es una implementación gratuita y de código abierto de la plataforma Java SE (Java Standard Edition) desarrollada por Oracle Corporation. OpenJDK fue creado como un proyecto de colaboración de la comunidad de desarrolladores de Java y está disponible bajo la Licencia Pública General de GNU (GPL) versión 2.

OpenJDK incluye un conjunto completo de herramientas de desarrollo para la creación de aplicaciones Java, incluyendo el compilador de Java (javac), el intérprete de bytecodes de Java (java), la herramienta de creación de archivos JAR (jar) y muchos otros componentes y bibliotecas. Además, OpenJDK también incluye una implementación de la Máquina Virtual Java (JVM) que puede ejecutar programas Java en una variedad de sistemas operativos.

OpenJDK es utilizado por una gran variedad de organizaciones y desarrolladores en todo el mundo, y es una de las implementaciones Java más populares y ampliamente utilizadas en la actualidad.

A continuación se dejan las implementaciones más utilizadas del producto.

Se recomienda descargar la última versión y que además tenga soporte de largo periodo.

- Descargar desde la página de la comunidad (recomendado)
  https://openjdk.org/projects/jdk/
- Descargar desde la página de Oracle
  https://jdk.java.net/
- Descargar desde Microsoft
  https://learn.microsoft.com/es-es/java/openjdk/download

IDE

IDE significa Entorno de Desarrollo Integrado (Integrated Development Environment) y es una herramienta de software que ayuda a los programadores a desarrollar y depurar aplicaciones de manera más fácil.

Las funciones más comunes que incluyen son:

- Editor de código: un editor de texto avanzado con funciones como resaltado de sintaxis, autocompletado, indentación automática, etc.
- Depurador: una herramienta que permite a los programadores ejecutar su código paso a paso para detectar y corregir errores.
- Gestión de proyectos: una herramienta que permite a los programadores organizar sus archivos de código fuente y recursos en un proyecto.
- Integración de herramientas externas: integración con otras herramientas como control de versiones, herramientas de análisis de código, generación de documentación, etc.

Algunos ejemplos populares de IDEs son IntelliJ IDEA, Eclipse, NetBeans, Visual Studio Code y Xcode. También se ha popularizado para la enseñanza el IDE Geany.

Se puede descargar e instalar los siguientes IDE

- IntelliJ IDEA (recomendado)
  https://www.jetbrains.com/es-es/idea/download/
- NetBeans
  https://netbeans.apache.org/front/main/index.html
- Visual Studio Code
  https://code.visualstudio.com/

[-] mi primer programa

Se recomienda leer el anexo # Utilizar el entorno de desarrollo que indica paso a paso como realizar un proyecto y luego escribir el programa para posteriormente realizar su ejecución.

El siguiente código muestra el primer programa que correremos en nuestra máquina.

```
public class HolaMundo {

   //
   public static void main(String[] args) {
```

```
        //
        System.out.println("hola mundo…yo programo!");
    }
}
```

Existen varias partes del programa que aún no hemos mencionado.

Las palabras "**public**", "**class**", "**static**" y "**void**", que aparecen en nuestro primer programa, son palabras reservadas del lenguaje que tienen un significado específico.

A medida que avancemos iremos explicando cada una de ellas. Por ahora es conveniente mencionar que los lenguajes de programación tienen ciertas reglas y son muy importantes para que todo funcione.

Existen reglas externas, que son necesarias para que el compilador o intérprete entienda cuales son los programas que se quieren ejecutar y reglas internas, que se refiere a la estructura gramatical del lenguaje.

Algunas reglas:

- El programa, por ahora, está dentro de lo que llamamos "clase principal".
- El nombre del archivo debe tener el mismo nombre que la clase principal seguido por la extensión .java
- Los comentarios de varias líneas se colocan entre /* y */
- Los comentarios de una línea se indican mediante //
- Para mostrar información por pantalla se utiliza System.out.println( )
- Las mayúsculas y minúsculas son tratadas de forma diferente.
- Los espacios se pueden utilizar de forma flexible
- Las sentencias finalizan con **;**

Al igual que un idioma natural, un lenguaje de programación tiene dos pilares fundamentales: la sintaxis y la semántica. Juntos, permiten crear instrucciones que las computadoras puedan entender y ejecutar para realizar tareas específicas.

[-] sintaxis

La sintaxis define las reglas gramaticales que rigen la estructura del código fuente. Es como el esqueleto del programa, que determina cómo se deben organizar las palabras, los símbolos y los operadores para que sean válidos.

Los elementos de la sintaxis incluyen a las palabras clave, identificadores, operadores y signos de puntuación. Si se viola alguna regla sintáctica, el programa no se ejecutará e informará de esta situación para que podamos resolver el inconveniente.

Si miramos el programa anterior, podemos ver ciertas reglas gramaticales, podemos apreciar parte de la especificación sintáctica, o sea, su sintaxis:

```
System.out.println(“hola mundo…yo programo!”);
```

Por supuesto que toda esta sentencia la entenderemos de forma completa más adelante, por ahora saber que hay elementos sintácticos presentes que se han respetado, por ejemplo que la frase que quiero que salga por la pantalla se encuentra determina por el comienzo y fin de las dobles comillas, o incluso apreciar que la sentencia termina con el punto y coma.

Para terminar de entender a qué nos referimos con sintaxis, pensemos en la gramática de nuestro lenguaje natural, que define las reglas para formar oraciones correctas.

Es la forma de escribir.

Podemos ver a las palabras clave, los símbolos y los operadores de un lenguaje de programación como las palabras del lenguaje natural. Con estas palabras armamos oraciones y expresamos ideas, lo mismo que sucede con el lenguaje de programación, con estas palabras armamos instrucciones y sentencias.

[-] semántica

La semántica se encarga del significado de las instrucciones o sentencias. Define cómo se interpretan y ejecutan las instrucciones en tiempo de ejecución y cómo afectan al estado del programa.

Los elementos de la semántica incluyen a los tipos de datos, las estructuras de control, las funciones y las expresiones.

Si la semántica del código es incorrecta, el programa puede ejecutarse pero no dará el resultado deseado.

Si miramos nuevamente la sentencia anterior, podemos determinar que la semántica de la misma será dirigir el contenido que se encuentra entre los paréntesis hacia la salida estándar, que comúnmente será nuestra pantalla, por lo que veremos en la pantalla la frase:

```
System.out.println("hola mundo…yo programo!");
```

La sintaxis es la forma, la semántica el significado.

En definitiva, la semántica define el comportamiento que tendrán nuestros programas y los resultados que producen.

Volvamos al ejemplo de la receta del bizcochuelo. La sintaxis sería la forma en que está escrita la receta, junto a la especificación de los ingredientes. Si se encuentra bien escrita entonces la entenderemos y podremos ejecutarla.

La semántica hace referencia a su propósito y resultado. Dependiendo de lo que se haya escrito y los ingredientes especificados, podremos decir cuán saludable es el bizcochuelo. Del resultado podemos luego decir si el bizcochuelo salió “bien” a criterio del cocinero, o inclusive que tan “rico” salió a criterio de los comensales.

Ejercicios

1. Usted tiene 10 pilas de monedas, cada una con 10 monedas de 1 peso. Toda una pila es de monedas falsas, pero usted no sabe cuál. Conoce el peso de una moneda genuina y también le han dicho que cada moneda falsa pesa un gramo más de lo debido. Usted puede pesar las monedas en una balanza. ¿Cuál es el número menor de pesadas necesarias para determinar cuál es la pila falsa?

2. En la orilla de un río se encuentran un lobo, una cabra y un repollo. Sólo hay un barco para cruzar todo hasta el otro lado. Esto sería sencillo si no fuera porque existen algunos inconvenientes:

    a. el barco sólo puede transportar de a uno.

    b. el lobo se come la cabra

    c. la cabra se come el repollo

    ¿Cómo hacemos para transportar todos hasta el otro lado?

3. El problema de los cuatro cuatros es uno de los problemas enunciados en el libro El hombre que calculaba (de Malba Tahan).

    El problema consiste en encontrar la forma matemática para representar los números del 0 al 10 usando para ello solo cuatro cuatros (cuatro números cuatro), y operaciones básicas.

    NOTA: no hay una única forma de hacerlo.

    Ejemplo para formar el número 0:

    0 = 44 - 44

    0 = 4 - 4 + 4 - 4

4. Detrás de una duna el hombre perdido en el desierto se encuentra con cuatro botellas enterradas en la arena y un cadáver al lado de ellas. Cada botella tiene una etiqueta:

    a. Acá hay agua o soda

    b. Acá hay agua o soda

    c. Acá hay veneno o jugo

    d. Acá hay veneno o agua

Las cuatro botellas tienen líquidos con distinto aspecto, con lo cual nuestro hombre intuye, acertadamente, que contienen cuatro líquidos distintos.

¿Qué botella contiene veneno?

5. Va otro del libro "El hombre que calculaba".
   El calculador, Beremiz, y su compañero Hanak, se encontraban viajando por el desierto en un solo camello, cuando se encontraron con tres hombres que discutían acaloradamente sobre una herencia. El calculador, interesado en el problema, se detuvo a hablar con los hombres, quienes explicaron:

   «Somos hermanos —dijo el más viejo— y recibimos, como herencia, esos 35 camellos. Según la expresa voluntad de nuestro padre, debo yo recibir la mitad, mi hermano Hamed Namir una tercera parte, y Harim, el más joven, una novena parte. No sabemos sin embargo, como dividir de esa manera 35 camellos, y a cada división que uno propone protestan los otros dos, pues la mitad de 35 es 17 y medio. ¿Cómo hallar la tercera parte y la novena parte de 35, si tampoco son exactas las divisiones?».

   Rápidamente, Beremiz propuso una solución y explicó que podría hacer perfectamente la división para dejar a todos conformes. Anunció que todos, incluídos él mismo y su compañero Hanak, saldrían favorecidos.

   ¿Qué propuso?

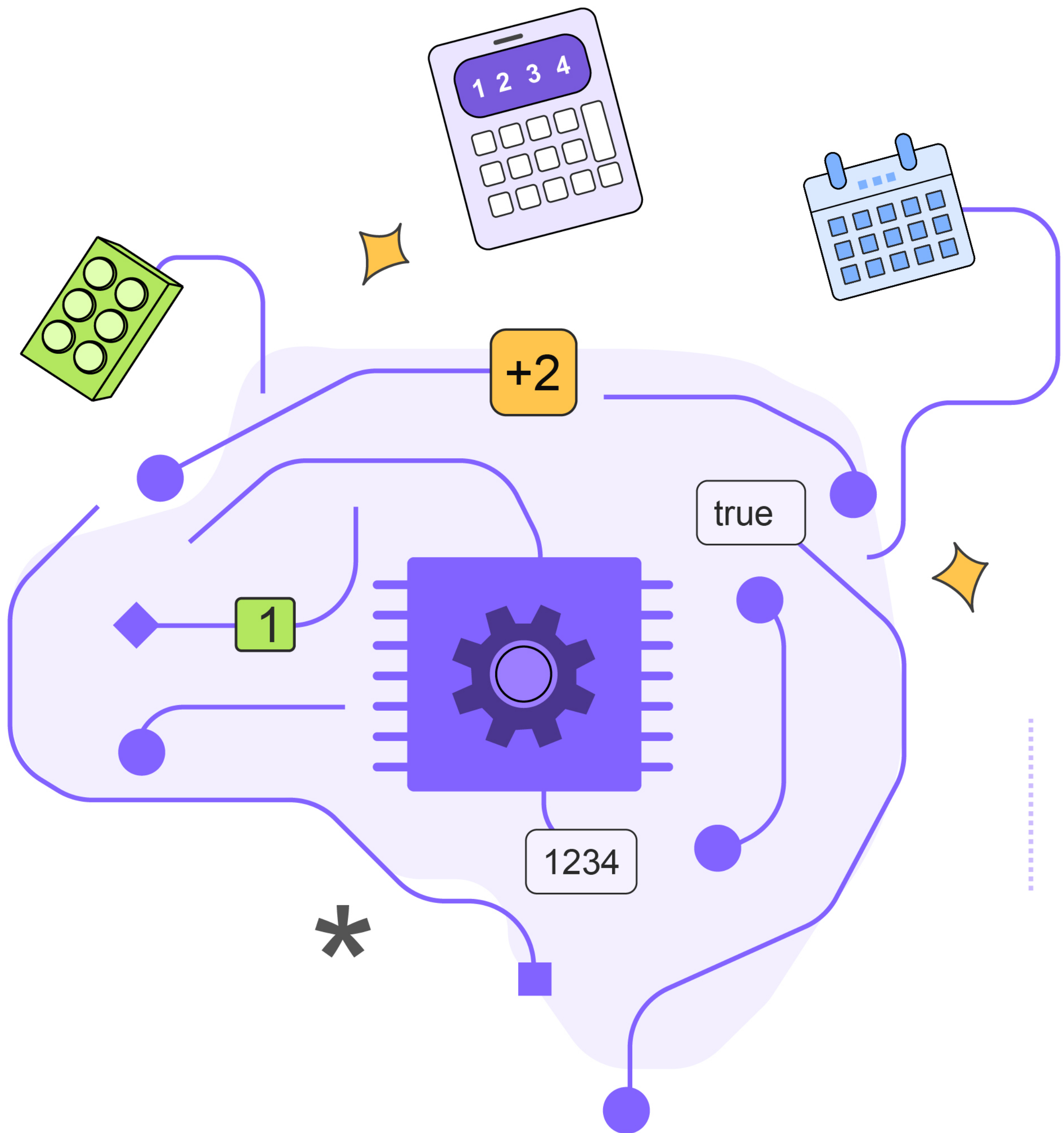
6. Si se colocara sobre un tablero de ajedrez (8x8), lo suficientemente grande, un grano de trigo en el primer casillero, dos en el segundo, cuatro en el tercero y así sucesivamente, doblando la cantidad de granos en cada casillero.

   ¿Cuántos granos de trigo habría en el tablero al final?

   Y si asumimos que 1 kg de trigo son 20.000 granos aproximadamente, ¿Cuántos kilos tendríamos en total?

Capítulo 2

Representación de datos



Capítulo 2

Representación de datos

Objetivo

El capítulo tiene como objetivo presentar conceptos relacionados con la especificación, representación y operaciones de los datos a través de un lenguaje de programación concreto. Además se definirá el concepto de variable, el cuál es fundamental para el desarrollo de soluciones algorítmicas bajo el enfoque que aborda el presente, orientado a objetos.

Al concluir este capítulo, habrás adquirido una comprensión de los diferentes tipos de datos disponibles, las operaciones que se pueden realizar con ellos y el concepto de variable. Además, estarás en condiciones de seleccionar el tipo de dato adecuado para diversas situaciones, utilizar las operaciones disponibles para manipular los datos y modelar la información del problema.

Dato

En informática, un dato es la unidad básica de información. Es una representación abstracta de un hecho, concepto o evento. No son las cosas en sí mismas, sino que representan las características de esas cosas de una manera que la computadora puede entender.

Para que una computadora pueda procesar datos, estos deben ser representados de forma simbólica, en un formato que la máquina comprenda. La abstracción juega un papel fundamental en este proceso, ya que permite convertir las características del mundo real en entidades manipulables por la máquina.

Existen diferentes formas de representar simbólicamente los datos:

- Números: se utilizan para representar cantidades, como la temperatura, el precio de un producto o la edad de una persona.
- Texto: se utiliza para representar palabras, frases y oraciones.
- Imágenes: se utilizan para representar objetos y escenas del mundo real.
- Sonidos: se utilizan para representar música, voces y otros efectos de audio.

La elección de la forma de representación simbólica de datos depende del tipo de información que se desea procesar. Utilizar la representación adecuada es fundamental para garantizar la precisión y eficiencia del procesamiento de la información.

Para que los datos sean útiles necesitan, en algún momento, ser procesados y convertidos en información.

Por ejemplo si hablamos de “clientes”, “capacidad de un balde” o inclusive “una fecha” como valores que resultan de interés para la solución que queremos desarrollar, entonces podemos decir que serán los datos de nuestro programa.

Otros ejemplos de datos:

- La temperatura de una habitación en grados Celsius: 22°C
- El peso de un objeto en kilogramos: 2,5 kg
- La edad de una persona en años: 35 años
- La altura de una montaña en metros: 4810 m
- El color de los ojos de una persona: marrón

No todos los datos son iguales. Algunos datos son entidades por sí mismas, mientras que otros dependen de otros datos para tener significado.

Un dato como entidad por sí misma es un dato único que tiene significado y utilidad sin necesidad de compararlo con otros datos.

Un ejemplo, de una dato como entidad, sería el número de celular dado que identifica una línea específica y se puede usar para realizar llamadas sin información adicional. El número es único y tiene un significado claro por sí mismo.

En cambio, un dato dependiente es un dato que no tiene significado por sí mismo y requiere de otros datos para ser interpretado.

La fecha de nacimiento, es un dato dependiente. No tiene significado por sí mismo, ya que no sabemos a quién pertenece. Para que la fecha de nacimiento tenga sentido necesitamos información adicional, como el nombre de la persona o su número de documento.

Variable

Una variable es un espacio en la memoria de la computadora que se utiliza para almacenar un valor. Este valor puede ser un número, un texto, un objeto o una referencia a un objeto.

Se identifica con un nombre único denominado identificador. Este identificador se emplea para acceder al valor almacenado en la variable y está vinculado a un tipo de dato particular que define el tipo de información que puede contener.

Las variables son elementos fundamentales en la programación y en particular en el paradigma que abordamos, ya que permiten almacenar y manipular datos.

Los datos que guardamos o modificamos, a través de las variables, lo hacemos durante la ejecución del programa. Esto significa que los valores almacenados en las variables pueden cambiar a lo largo del tiempo, lo que brinda flexibilidad y, como veremos más adelante, otorga capacidad de respuesta a la lógica de nuestros programas.

En definitiva las variables nos permiten almacenar datos durante la ejecución del programa y utilizar estos para realizar cálculos y tomar decisiones.

En el capítulo anterior definimos al lenguaje JAVA como tipado y estructurado. Dijimos que ser tipado implica que el tipo de dato asociado a una variable debe establecerse con anterioridad a su ejecución. Esta asociación en tiempo de escritura permite al lenguaje verificar la compatibilidad de tipos durante la compilación o ejecución del programa, lo que ayuda a prevenir errores y mejorar la confiabilidad del código producido.

A continuación se muestran tres declaraciones de variables que, como se puede observar, tendrán diferente propósito e intuitivamente podemos inferir el tipo de dato asociado:

```
int suma;

float promedio;

boolean puedeContinuar;
```

Una variable es un contenedor de datos que tiene un nombre único y puede almacenar valores que pueden cambiar durante la ejecución del programa.

[-] ámbito

El ámbito de una variable, también conocido como alcance o “scope” en inglés, se refiere a la o las partes del programa donde esa variable es accesible y puede ser utilizada.

El ámbito estará determinado por la ubicación donde se realiza la declaración de la variable.

El significado de “ámbito” lo ampliaremos cuando se desarrolle el tema #función, específicamente cuando se explique el concepto de variable local.

Tipo de dato

Los tipos de datos son una forma de clasificar y definir el dato o información que se puede almacenar y procesar en un programa. Cada lenguaje de programación tiene su propio conjunto de tipos de datos, aunque hay algunos que son básicos y la mayoría de los lenguajes los tienen.

Los tipos de datos sirven para ayudar al programador a entender qué tipo de información se espera o cómo se debe procesar. En nuestro caso los tipos de datos los veremos asociados principalmente a las variables y funciones.

Podemos ilustrar lo mencionado de la siguiente manera, consideremos el escenario en el que necesitamos determinar si una puerta está abierta o cerrada antes de realizar alguna acción.

En este caso, a primera vista, podríamos pensar en utilizar los valores “abierta” o “cerrada” para representar el estado de la puerta. Sin embargo, un programador experimentado reconocerá que el tipo de dato más apropiado en este caso es el tipo “lógico”, donde los valores posibles son simplemente “verdadero” o “falso”.

Es importante destacar que el proceso de abstracción que se requiere para realizar esta actividad se puede conseguir con el estudio de saberes propios de la disciplina y por supuesto, poniendo estos conceptos en práctica.

Los tipos de datos se encuentran caracterizados por:

- Rango de valores posibles: es el conjunto de valores que puede almacenar un dato de ese tipo.
- Operaciones: son las acciones que se pueden realizar con un dato de ese tipo.
- Representación interna: es la forma interna que tiene el dato. El dato interno también puede tener otra representación y así hasta llegar a cómo se guarda el dato en la memoria de la computadora.

Es importante destacar en este momento que los bits son la unidad básica de información en una computadora. Un bit puede ser un 0 o un 1. Como mencionamos anteriormente los

tipos de datos tienen una representación interna y en definitiva son un conjunto de bits que se interpretan de una manera específica.

Los valores numéricos se pueden representar utilizando bits. Esto se conoce como codificación binaria. Entonces, para representar valores numéricos, se utiliza una combinación de bits donde cada uno representa un valor específico.

La representación decimal, el sistema numérico que utilizamos diariamente, se basa en 10 símbolos -0,1,2,3,4,5,6,7,8 y 9- y con estos formamos todos los números. Por ejemplo el 45, está compuesto por el símbolo 4 y el símbolo 5, aunque ambos tienen un peso diferente que se lo dá su posición. Otra forma de representar este valor es a través de la expresión 40 + 5, y esta exŕesión la podemos armar en función a las potencias de diez, dado que es su base. Cada dígito decimal tiene un valor asociado, que se multiplica por la potencia de diez correspondiente a su posición.

Por ejemplo, en el número decimal 123, el dígito "1" representa el valor 1×10^2, el dígito "2" representa el valor 2×10^1 y el dígito "3" representa el valor 3×10^0. Esto da la expresión 100 + 20 + 3 que finalmente resulta en 123.

Ahora si pensamos en la representación binaria, tendríamos la misma forma de obtener el valor pero con una base de 2.

En base 2, que es lo mismo que decir en binario, el número 110 equivale a 6 en el sistema decimal. Siguiendo el mismo procedimiento que antes, ahora tenemos que el dígito "1" representa el valor 1 x 22 , el dígito "1" representa el valor 1×2^1 y el dígito "0" representa el valor $0 \times 2^0 \Rightarrow 4 + 2 + 0 = 6$.

Entonces, si tenemos 8 bits disponibles, podemos representar valores numéricos del 0 al 255. Cómo vimos, cada bit en la secuencia tiene un valor asociado según su posición, comenzando desde la derecha y aumentando en potencias de 2. La posición más a la derecha -menos significativa- representa el valor 1, la siguiente posición el valor 2, luego el valor 4, y así sucesivamente.

Por lo tanto, si todos los bits están en 0, el valor representado sería 0. Si todos los bits están en 1, el valor representado sería el máximo posible con 8 bits, es decir, 255. Para representar otros valores, se cambian los bits individuales de acuerdo con la cantidad necesaria.

Por ejemplo, si queremos representar el número 5 con 8 bits, podemos usar la representación binaria "00000101". Aquí, el bit más a la derecha representa el valor 1, el tercer bit representa el valor 4, y la suma de ambos da como resultado 5.

```
Ejemplo de un valor binario de 8 bits a su representación decimal:

En binario, con 8 bits = 00011011

En decimal, 1*2^0 + 1*2^1 + 0*2^2 + 1*2^3 + 1*2^4 + 0*2^5 + 0*2^6 + 0*2^7  =
              1   +  2  +  0   +  8  + 16  +   0  +   0 +  0   =  27
```

Por último, a modo de complementar el tema, hay dos maneras comunes de representar números en formato binario.

Representación con signo y magnitud: en esta representación, un bit se utiliza para indicar el signo del número (0 para positivo, 1 para negativo) y los bits restantes se utilizan para representar el valor absoluto del número. Esta representación puede ser simple de enten-der, pero puede presentar problemas al realizar operaciones aritméticas.

Complemento a 2: en esta representación, el valor negativo de un número se representa mediante el complemento a 2 del valor absoluto del número positivo correspondiente. Esta representación tiene la ventaja de que simplifica la aritmética y maneja de manera unifor-me tanto los números positivos como los negativos.

Ambos sistemas tienen sus ventajas y desventajas, y la elección entre ellos puede de-pender de factores como la eficiencia del hardware, la facilidad de implementación y las necesidades específicas del sistema en cuestión.

A continuación definiremos algunos de los tipos de datos simples que la mayoría de los lenguajes soportan.

[-] entero

Representa números enteros, positivos o negativos, sin decimales.

Ejemplos: 1, -5, 1000.

- int: es un entero de 32 bits con signo, puede representar valores desde -2147483648 hasta 2147483647.
- long: es un entero de 64 bits con signo, puede representar valores desde -9223372036854775808 hasta 9223372036854775807.

```
int valor = -123;

//
long grande = 9876543210;
```

[-] decimal

También conocido como flotante, representa números con decimales.

Ejemplos: 3.14, -0.5, 100.0.

- float: es un número de coma flotante de precisión simple de 32 bits que puede representar valores desde 1.4e-45 hasta 3.4028235e38.

  1.4e-45 = 1.4 x 10-45 =

  0.0000000000000000000000000000000000000000000014

  a

  3.4028235e38 = 3.4025235 x 1038 =

  340282350000000000000000000000000000000

- double: es un número de coma flotante de precisión doble de 64 bits que puede representar valores desde 4.9e-324 hasta 1.8e308.

[-] lógico

Representa el valor verdadero o falso.

- boolean: es un valor que puede ser true o false.

[-] caracter

Representa un carácter "Unicode" individual. Ejemplos: 'a', 'B', '5'.

Unicode es un estándar de codificación. Implica utilizar un código único para cada carácter, independientemente del idioma o plataforma.

- char: es un carácter Unicode de 16 bits que puede representar valores desde '\u0000' (o 0) hasta '\uffff' (o 65,535)

[-] texto

Representa una serie de caracteres.

Ejemplos: “yo programo”, “Hola, mundo!”, “12345”, “un texto largo”.

- En java no es un tipo de dato simple, es un caso especial. Podemos decir, por ahora, que podemos representar cualquier longitud o conjunto de caracteres de un texto.

Las ventajas que nos aportan la utilización de los tipos de datos son varias, siendo las más significativas las siguientes:

- Flexibilidad: en el caso de que se desee modificar la forma en que se representa la información sólo se debe modificar la declaración y no buscar en todos los lugares donde aparecen ciertos valores.
- Documentación: si se hace un buen uso del identificador se puede transmitir cómo se deben utilizar y los valores esperados, facilitando de esta manera el entendimiento y lectura del código.
- Seguridad: se reducen los errores de correspondencia entre el valor que se pretende asignar a una variable y el que soporta el tipo. También el compilador o intérprete puede hacer comprobaciones sobre el código aumentando su confiabilidad.

Operaciones

A continuación describimos las operaciones comunes que pueden llevarse a cabo con los tipos de datos mencionados.

Es importante mencionar que la mayoría de los operadores, que se describen a continuación, utilizan dos operandos para funcionar.

Dos operandos:

- `<operando> <operador> <operando>`
  o `2 + 3`

Un operando:

- `<operador> <operando>`
  `variable++`

[-] operaciones de asignación

Se pueden asignar valores a una variable utilizando el operador = (asignación).

También se pueden realizar operaciones de asignación abreviadas, como:

- += : asigna a la variable la suma de sí misma con la expresión de la derecha.
- -= : asigna a la variable la resta de sí misma con la expresión de la derecha.
- *= : asigna a la variable la multiplicación de sí misma con la expresión de la derecha.
- /= : asigna a la variable la división de sí misma con la expresión de la derecha.
- %= : asigna a la variable el módulo de sí misma con la expresión de la derecha.

Ejemplos:

```
//declaro e inicializo el valor de la variable
int suma = 0;

//asigno a la variable el contenido de suma + 5
suma = suma + 5;

//asigno a la variable el contenido de suma + 5
suma += 5;
```

[-] operaciones matemáticas

Se pueden realizar operaciones aritméticas básicas con los tipos de datos numéricos (byte, short, int, long, float y double), tales como:

- + : suma
- - : resta
- * : multiplica
- / : divide
- % : retorna el resto de la división, se llama módulo.

- ++ : incrementa el valor de la variable. Puede ir del lado izquierdo o derecho de la variable.
- -- : decrementa el valor de la variable. Puede ir del lado izquierdo o derecho de la variable.

Ejemplos:

```
//realizo el cálculo y asigno el resultado a la variable
promedio = suma / 2;

//
temp = temp * 73;

//
resto = numero % 10;

//incremento en 1 el valor que tiene contador y luego lo asigna a
cantidad.
cantidad = contador++;

//asigna el valor que tiene contador a cantidad y luego incremen-
ta en 1 el valor que tiene contador.
cantidad = ++contador;
```

[-] operaciones de comparación

Se pueden comparar dos valores utilizando los operadores de comparación:

- == : es igual
- != : es diferente
- > : es mayor que
- >= : es mayor o igual que
- < : es menor que
- <= : es menor o igual que

Todos los operadores de comparación retornan un valor lógico, que dependiendo la expresión será verdadero o falso.

Ejemplos:

```
//compara el valor de las variables, el resultado se asigna
//verdadero si la variable suma es 100 sino falso
varLogica = suma == 100;

//verdadero si la edad es mayor o igual a 18
esMayor = edad >= 18;
```

[-] operaciones lógicas

Se pueden realizar operaciones lógicas con los valores lógicos -booleanos- utilizando los operadores:

- && : si los dos operando lógicos son verdaderos retorna verdadero caso contrario falso.
- || : si al menos un operando es verdaderos retorna verdadero caso contrario falso.
- ! : retorna la negación del valor del operando lógico. Si es verdadero retorna falso y si es falso retorna verdadero.

Ejemplos:

```
//varLogica será verdadero si suma es 100 o resta es 0
varLogica = (suma == 100) || (resta == 0);

//continuar será verdadero si la cantidad tiene un valor
//menor a 10 y además la variable puedoSeguir es verdadera
continuar = (cantidad < 10) && ( puedoSeguir == true);
```

[-] operaciones de concatenación

Se pueden juntar cadenas de caracteres utilizando el operador +. Este operador tiene un comportamiento distinto dependiendo si se utiliza con números o si se utiliza con textos. Esto se llama sobrecarga de operadores.

Al igual que vimos con la asignación y suma o asignación y resta, entre otros, acá también se puede utilizar el operador += para concatenar y asignar en una sola operación.

Ejemplos:

```
//declaro e inicilizo el texto de la variable
String frase = "Hola";

//junto el valor de la variable con el texto " mundo!"
frase = frase + " mundo!";

//muestro por pantalla el resultado
System.out.println(frase);
```

Proposición

Una proposición es una oración que afirma o niega algo, es una declaración que puede ser verdadera o falsa.

En lógica y matemáticas, una proposición es una unidad básica de conocimiento que se utiliza para construir razonamientos y argumentos.

Las proposiciones son como las piezas de encastre de la programación. Son ideas básicas que se pueden combinar para construir cosas más complejas, como los programas:

- Permiten crear instrucciones precisas para la computadora. Las computadoras sólo entienden dos cosas: "si" o "no", o sea, 1 o 0. Las proposiciones se usan para traducir nuestras ideas en ese lenguaje binario.
- Ayudan a organizar el código. Al dividir el código en proposiciones pequeñas y manejables, es más fácil de entender, escribir y corregir.

- Permiten tomar decisiones. Los programas necesitan tomar decisiones todo el tiempo. Las proposiciones se utilizan para definir las condiciones que deben cumplirse para tomar esas decisiones.

Una proposición se distingue de una pregunta o una orden, ya que estas no tienen un valor de verdad definido.

Por ejemplo, “El sol es una estrella” o “hace frío” son proposiciones, ya que se pueden evaluar como verdaderas o falsas, mientras que “¿hace frío?” o "¿el sol es una estrella?” son preguntas y no tienen un valor de verdad definido, lo que persigue la pregunta es obtener información.

Las proposiciones se utilizan en la lógica formal para construir argumentos y demostrar teoremas. También se utilizan en el lenguaje cotidiano para expresar ideas y comunicarse de manera precisa.

Vamos a considerar a una proposición como simple cuando sea una declaración que no se puede descomponer en proporciones más pequeñas. Otro modo de ver esto es, una proposición será simple si no tiene conectores lógicos, por ejemplo “y”, “o”, “si...entonces”.

Una proposición simple es una unidad básica de conocimiento que se considera como un enunciado completo y que tiene un valor de verdad definido, es decir, puede ser verdadera o falsa.

> Las proposiciones simples suelen ser enunciados concretos que expresan una idea o un hecho específico.

Ejemplos de proposiciones simples:

- Hoy es lunes.
- El agua hierve a 100 grados celsius.
- La Tierra gira alrededor del sol.
- Mi perro se llama Marley.

Estas afirmaciones no requieren de ninguna otra proposición para ser evaluadas en cuanto a su valor de verdad, o sea, su veracidad o falsedad. Cada proposición simple puede ser analizada de manera independiente.

Las proposiciones simples son utilizadas como componentes básicos en la construcción de proposiciones compuestas, que se forman mediante la combinación de proposiciones más simples utilizando conectores lógicos. Estos conectores lógicos, también conocidos como

conectivos u operadores lógicos, permiten establecer relaciones lógicas entre las proposiciones y construir proposiciones más complejas.

Los conectores lógicos utilizados para formar proposiciones compuestas son:

Conjunción: se representa con el símbolo "∧" o mediante la letra "y". Indica que ambas proposiciones deben ser verdaderas para que la proposición compuesta sea verdadera. Ejemplo: "Llueve y hace frío".

Disyunción: se representa con el símbolo "∨" o mediante la letra "o". Indica que al menos una de las proposiciones debe ser verdadera para que la proposición compuesta sea verdadera.

Estudiaré informática o electrónica.

Negación: se representa con el símbolo "¬"o mediante la palabra "no". Niega el valor de verdad de una proposición.

No estoy estudiando.

Implicación: se representa con el símbolo "➔" o mediante la estructura "si...entonces". Indica que si la primera proposición - el antecedente- es verdadera, entonces la segunda proposición - el consecuente- también debe ser verdadera.

Si estudias entonces no tendrás dudas

Si no estudias entonces tendrás dudas

Si estudias, aprobarás.

Doble implicación: se representa con el símbolo "↔" o mediante la estructura "si y sólo si". Indica que las dos proposiciones tienen el mismo valor de verdad, es decir, ambas son verdaderas o ambas son falsas.

Un número es par si y sólo si es divisible por 2.

Ejemplos de proposiciones compuestas:

- Si estudias, aprobarás el examen. [implicación]
- El perro ladra y el gato maulla. [conjunción]

- El libro es caro o está agotado en la tienda. [disyunción]
- Si está soleado, iremos a la playa; de lo contrario, nos quedaremos en casa. [implicación y disyunción]

Por último, hay dos tipos adicionales de proposiciones compuestas que se utilizan como elementos dentro de las proposiciones compuestas, las categóricas y las existenciales.

Las proposiciones categóricas son aquellas que expresan una relación entre dos conceptos o clases. Suelen ser enunciados de la forma "Todos los/as..." o "Algunos/as...".

Todos los perros ladran.

Las proposiciones existenciales, por otro lado, son proposiciones que afirman la existencia o inexistencia de algo. Expresan si algo está presente o no.

Existe vida en otros planetas.

Las proposiciones son la base para construir programas que sean eficientes, fáciles de entender y que también, si se requiere, puedan realizar tareas complejas.

Terminemos con un ejemplo. Imaginemos que quisiéramos crear un programa que nos ayude a elegir qué comer según el clima:

- Si hace frío, tomar una sopa caliente.
- Si hace calor, comer una ensalada fresca.
- Si está lloviendo, comer tortafritas.

Tener en cuenta que este ejemplo muestra el uso de proposiciones pero no implica que esté armado para que se elija sólo una opción, ¿qué pasaría si hace calor y está lloviendo?

[-] tablas de verdad

Las tablas de verdad también son herramientas de la lógica y las matemáticas, sirven para determinar los valores de verdad de una proposición.

Al utilizar las tablas de verdad, es posible determinar si una proposición es verdadera o falsa en función de los valores de verdad de las proposiciones que la componen, siguiendo las reglas de la lógica y los conectores lógicos utilizados en la construcción de la proposición compuesta.

Como ya mencionamos, los operadores más utilizados son la negación, la disyunción y la conjunción.

Recordemos cuál es el funcionamiento de cada uno a través del uso sobre las proposiciones.

La negación invierte el valor de verdad de una proposición.

Por ejemplo si A fuese "está lloviendo", NOT(A) sería "no está lloviendo".

p	!p
v	f
f	v

Fig.06 - tabla de verdad - negación

La conjunción devuelve verdadero sólo si ambas proposiciones son verdaderas.

Por ejemplo si A fuese "tengo hambre" y B fuese "tengo plata", A AND B sería "tengo hambre y plata".

p	q	q & q
v	v	v
v	f	f
f	v	f
f	f	f

Fig.07 - tabla de verdad - conjunción

La disyunción devuelve verdadero si al menos una de las proposiciones es verdadera.

Por ejemplo, si A fuese "es de día" y B fuese "está nublado", A OR B sería "es de día o está nublado".

p	q	q | q
v	v	v
v	f	v
f	v	v
f	f	f

Fig.08 - tabla de verdad - disyunción

Veamos otros ejemplos de utilización que nos acerca, de a poco, al código:

1. Supongamos que tenemos:

   p - Juan estudia = V (verdadero)

   q - Pedro trabaja = F (falso)

   Entonces, de acuerdo a los valores de las tablas de verdad de los conectores lógicos, el resultado de las siguientes proposiciones sería:

   Proposición: Juan estudia y Pedro trabaja

   Representación simbólica: $p \wedge q$

   Resultado Falso

   Proposición: Juan estudia o Pedro trabaja

   Representación simbólica: $p \vee q$

Resultado Verdadero

2. Sigamos con la misma lógica pero ahora con variables:

```
int num1 = 5;

int num2 = 20;

boolean p = (num1 > num2) || (num1 > 10); //Es falso

boolean q = (num2 < 100) && (num1 <= 5); //Es verdadero

boolean r = ! (num2 < num1);  //Es verdadero
```

Entonces, de acuerdo a las tablas de verdad de cada conectivo lógico, el resultado de las siguientes proposiciones compuestas serían:

$p \land q$ (falso) $q \land r$ (verdadero)

$p \lor q$ (verdadero) $q \lor r$ (verdadero)

Para concluir, las proposiciones y los lenguajes de programación están estrechamente relacionados. Las proposiciones se utilizan en la lógica y el razonamiento, mientras que en los lenguajes de programación son herramientas utilizadas para escribir instrucciones y algoritmos que controlan el comportamiento de un programa.

En los lenguajes de programación, las proposiciones se representan mediante expresiones lógicas y condiciones que evalúan si una afirmación es verdadera o falsa. Este concepto lo retomaremos en el capítulo sobre #Estructuras de control.

Ejercicios

1. Escribir un programa que muestre tu nombre por pantalla.
2. Modificar el programa anterior para que además muestre tu dirección y tu número de teléfono. Asegurarse de que los datos se muestran en líneas separadas.
3. Escribir un programa que muestre por pantalla 10 palabras en inglés junto a su correspondiente traducción al español. Las palabras deben estar distribuidas en dos columnas y alineadas a la izquierda como se muestra a continuación:

   Ejemplo:

   ```
   computer    computadora
   student     alumno
   cat         gato
   penguin     pingüino
   machine     máquina
   nature      naturaleza
   light       luz
   green       verde
   book        libro
   pyramid     pirámide
   ```

4. Indique el tipo de dato de cada una de las siguientes expresiones.
   a. 613
   b. 613.0

c. “613”

d. “4.32”

e. 3.012 e-12

f. f. 3+7

g. 3.5+6.5

5. Escribir las asignaciones necesarias para intercambiar el valor de X con el valor de Y.

X ← 10

Y ← 20

6. En cuáles de los siguientes pares de asignaciones es importante el orden, es decir, si se modifica el orden de ellos cambia el resultado final.

X ← Y

Y ← Z

X ← Y

Z ← X

X ← Z

X ← Y

Z ← Y

X ← Y

7. Escribir un programa en el que se declaren las variables enteras x e y. Asignar los valores 300 y 900 respectivamente. Mostrar por pantalla el valor de:

a. la suma

b. la resta

c. la multiplicación

d. la división

e. el intercambio

8. Escribir un programa que pinte por pantalla una pirámide rellena a base de asteriscos. La base de la pirámide debe estar formada por 9 asteriscos.

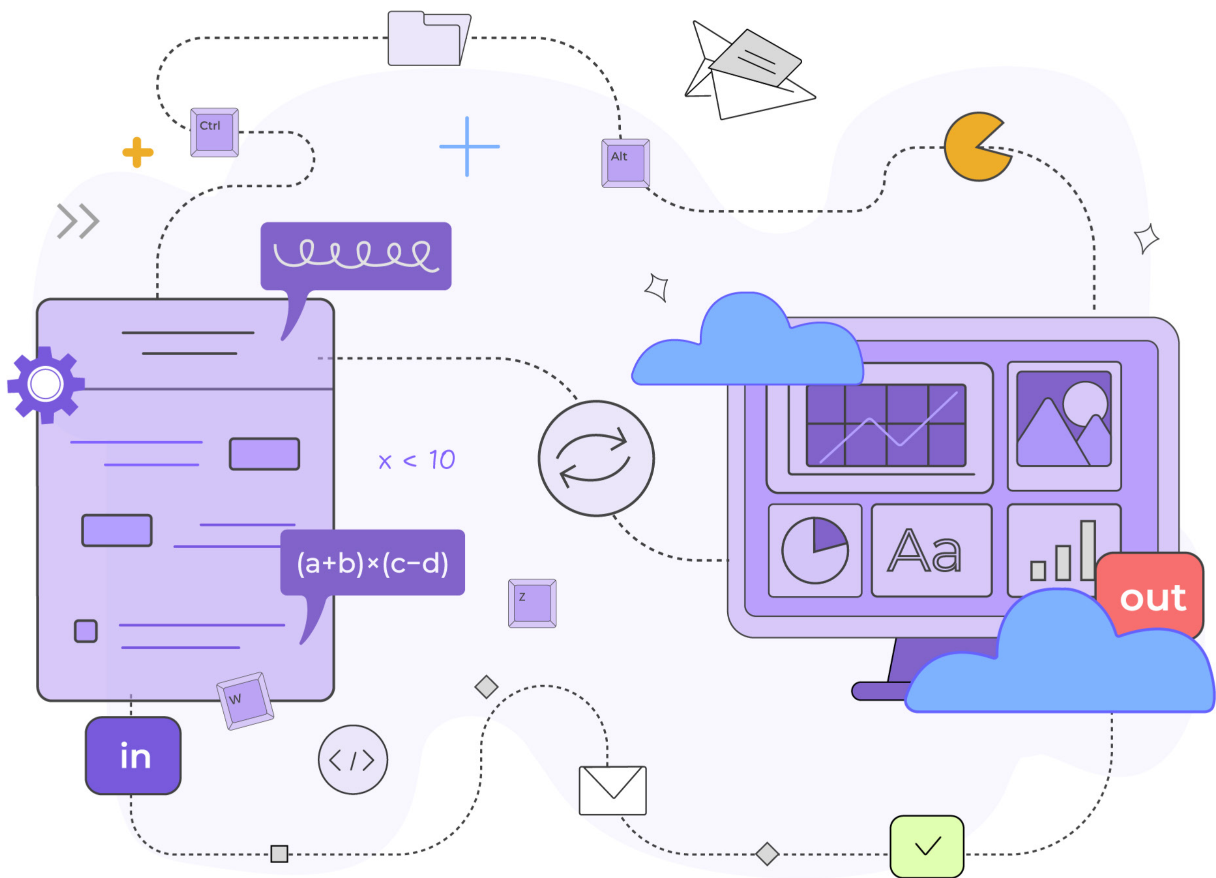
```
    *
   ***
  *****
 *******
*********
```

9. Igual que el programa anterior, pero esta vez la pirámide estará vacía (se debe ver únicamente el contorno hecho con asteriscos).

```
    *
   * *
  *   *
 *     *
* * * * *
```

Capítulo 3

Entrada y salida de datos



Capítulo 3

Entrada y salida de datos

Objetivo

El capítulo tiene como objetivo desarrollar la interacción entre un programa y el usuario. Se explicarán las herramientas que el lenguaje de programación nos ofrece para leer los datos del usuario, procesarlos y finalmente mostrar resultados de forma clara y atractiva.

Al finalizar este capítulo, podrás crear programas que interactúen con el usuario de forma efectiva. Habrás adquirido una comprensión profunda de las diferentes técnicas de entrada y salida de datos que provee el lenguaje, así como la importancia de la interacción con el usuario. Además, estarás en condiciones de diseñar preliminarmente interfaces intuitivas que brinden una experiencia satisfactoria.

Variables, constantes y expresiones

Como mencionamos anteriormente, una variable es un espacio en la memoria que un programa utiliza para almacenar un valor y que además puede cambiar durante su ejecución. Es un contenedor de datos al que se le puede asignar un valor y modificar durante la ejecución del programa.

La variable debe ser identificada con un nombre único dentro del bloque que la alcance. Esta última parte, que mencionamos sobre el alcance, lo veremos en profundidad más adelante. Por ahora pensemos el alcance como algo que cubre todo el programa.

A continuación declaramos una variable de tipo numérica, o sea, una variable que nos permitirá almacenar valores enteros. También aprovechamos la declaración para asignarle un valor inicial:

```
//
int enteros = 10;
```

Esta asignación inicial es opcional, puede omitirse.

[-] constante

Una constante es un espacio de memoria que contiene un valor, al igual que las variables. Este valor, una vez que ha sido establecido, no puede cambiar durante toda la ejecución del programa. El lugar donde se establece este valor es en su declaración.

Las constantes son útiles para:

- Mejorar la legibilidad del código: usar nombres descriptivos para valores que no cambian hace que el código sea más fácil de entender.
- Evitar errores: si un valor no cambia, no hay riesgo de que se modifique accidentalmente y cause un error.
- Compartir valores: las constantes se pueden usar para compartir valores entre diferentes partes del programa.

La forma de declarar una constante en Java es mediante el uso de la palabra reservada "**final**". Indica que el espacio de la memoria que contiene el valor no se puede modificar durante toda la ejecución del programa.

Por convención suelen utilizarse identificadores descriptivos y en mayúsculas.

Veamos cómo se declara una constante:

```
//una constante
final int DIAS_LABORALES = 5;

//una constante
final String MENSAJE_BIENVENIDA = "Buenos días...";
```

A diferencia de la variable, la posibilidad de asignar un valor en la declaración deja de ser una opción, es obligatorio.

[-] expresión

Una expresión es una combinación de valores, variables, operadores y funciones que se evalúan para obtener un resultado único. Es como una fórmula matemática que se puede calcular.

Las expresiones son útiles para:

- Realizar cálculos: se pueden usar para sumar, restar, multiplicar, dividir y realizar otras operaciones matemáticas y lógicas.
- Comparar valores: se pueden usar para comparar si dos valores son iguales, diferentes, mayores o menores.
- Obtener información: se pueden usar para obtener información de variables o funciones.

La evaluación de expresiones en Java se realiza siguiendo un orden de precedencia y asociatividad específica. Este orden determina cómo se interpretan los operadores y en qué orden se realizan las operaciones.

El orden de precedencia define qué operadores se evalúan primero. Los operadores con mayor precedencia se evalúan antes que los de menor precedencia.

El siguiente es el orden de precedencia de los operadores en Java, de mayor a menor:

- Operadores unarios: +, -, ~, !
- Operadores de multiplicación y división: *, /, %
- Operadores de suma y resta: +, -
- Operadores relacionales: <, <=, >, >=, ==, !=
- Operadores lógicos: &&, ||
- Operador de asignación: =

La asociatividad define en qué orden se evalúan los operadores que tienen la misma precedencia. En Java, la mayoría de los operadores son asociativos de izquierda a derecha, lo que significa que se evalúan de izquierda a derecha. Los operadores de asignación son asociativos de derecha a izquierda.

```
3 + 5 * 4   ← retorna 3 más la multiplicación de 5 por 4

x = y = 5  ← asigna 5 a la variable “y” y luego a x
```

Los paréntesis permiten modificar el orden de evaluación natural de una expresión, obligando a que las operaciones dentro de ellos se realicen primero.

```
( 3 + 5 ) * 4 ← retorna 3 más 5 y luego multiplica por 4
```

Las expresiones pueden ser más complejas, involucrando varias operaciones y funciones. Por ejemplo, la siguiente es una expresión que incluye el uso de una función, concepto que desarrollaremos más adelante:

```
(2 * x) + sin(y) - z
```

En esta expresión, se está multiplicando por 2 el valor de la variable "x", luego se realizará la suma del resultado de aplicar la función seno a la variable "y", finalmente se resta el valor de la variable "z".

La expresión retorna un valor único, que depende de los valores actuales de las variables x, y, y z.

> Las expresiones se utilizan en programación para calcular valores, tomar decisiones y controlar el flujo de ejecución de un programa.

La operación asignación

El operador de asignación se utiliza para asignar un valor a una variable. Se utiliza el símbolo "=".

Veamos dos asignaciones. Notemos que le daremos el valor a la variable x de la expresión que se encuentre a la derecha del operador:

```
//una asignación con el valor literal 10
x = 10;

//una asignación con el valor de la expresión
x = (y / z) - 10;
```

Después de ejecutar la primera línea, la variable x tendrá el valor de 10.

Después de ejecutar la segunda línea, la variable x tendrá el valor de haber resuelto la expresión “(y / z) - 10”.

Es importante tener en cuenta que el operador de asignación es parecido al de comparación “==”, aunque este último sólo sirve para comparar si dos valores son iguales.

Recordemos los tipos de asignación que nos ofrece el lenguaje:

- += → suma y asigna.
  variable += expresión;
- -= → resta y asigna.
  variable -= expresión;
- *= → multiplica y asigna.
  variable *= expresión;
- /= → divide y asigna el cociente.
  variable /= expresión;
- %= → divide y asigna el resto (módulo).
  variable %= expresión;
- &= → AND bitwise y asigna. ;
- |= → OR bitwise y asigna.
- ^= → XOR bitwise y asigna.
- <<= → desplazamiento de bits a la izquierda y asigna.
- >>= → desplazamiento de bits a la derecha y asigna.
- >>>=→ desplazamiento de bits sin signo a la derecha y asigna.

Funciones internas

El lenguaje nos provee una gran variedad de funciones y además provee un mecanismo para incorporar librerías que nos permite extender el conjunto de funciones utilizables.

Se listan las funciones más utilizadas para manipular tipos de datos simples.

```java
//imprimir en la pantalla un texto
System.out.println("un texto...");

//obtener un valor aleatorio
double aleatorio = Math.random();

//obtener a partir del texto un valor entero
int numero = Integer.parseInt("10");

//obtener a partir del texto un valor decimal (flotante)
float f = Float.parseFloat("10.5");

//obtener a partir del texto un valor decimal doble
double r = Double.parseDouble("10.5");
```

Las cadenas de caracteres (String) son una parte fundamental de la programación. Sirven para la comunicación entre el programa y el usuario, así como para el procesamiento y almacenamiento de información textual.

A continuación, se muestran algunas de las funciones más utilizadas.

- **`length():`** esta función se utiliza para obtener la longitud de un texto o, lo que equivalente, una cadena de caracteres. Devuelve un valor entero que representa la cantidad de caracteres en la cadena. El espacio, que separa las palabras en una frase, se considera un caracter también.

```java
//
String s = "Hola mundo";

//retorna la longitud del texto = 10
int longitud = s.length();
```

- **`charAt():`** esta función se utiliza para obtener el carácter en una posición específica de una cadena de caracteres. Toma un índice como argumento y devuelve el carácter correspondiente. El primer caracter se encuentra en la posición cero.

```
//
String s = "Hola mundo";

//retorna el caracter ‘m’
char c = s.charAt(5);
```

- **`substring():`** esta función se utiliza para obtener una subcadena de una cadena de caracteres. Toma dos índices como argumentos, que representan el inicio y el final de la subcadena, y devuelve la subcadena correspondiente.

```
//
String s = "Hola mundo";

//retorna el texto ‘Hola’
String sub = s.substring(0,4);
```

- **`indexOf():`** esta función se utiliza para obtener la posición de la primera aparición de un carácter o una subcadena en una cadena de caracteres. Toma el carácter o la subcadena como argumento y devuelve la posición correspondiente.

```
//
String s = "Hola mundo";

//retorna la posición 5
int pos = s.indexOf("m");
```

- **`toUpperCase():`** esta función se utiliza para convertir una cadena de caracteres a mayúsculas. Devuelve una nueva cadena de caracteres que es una versión en mayúsculas de la cadena original.

```
//
String s = "Hola mundo";

//retorna el texto 'HOLA MUNDO'
String up = s.toUpperCase();
```

- **`toLowerCase():`** esta función se utiliza para convertir una cadena de caracteres a minúsculas. Devuelve una nueva cadena de caracteres que es una versión en minúsculas de la cadena original.

```
//
String s = "Hola mundo";

//retorna el texto 'hola mundo'
String lo = s.toLowerCase();
```

- **`trim():`** esta función se utiliza para eliminar los espacios en blanco al principio y al final de una cadena de caracteres. Devuelve una nueva cadena de caracteres que es una versión sin espacios de blancos, iniciales y finales, de la cadena original.

```
//
String s = "  Hola mundo  ";

//retorna el texto 'hola mundo'
String t = s.trim();
```

- **replace():** esta función se utiliza para reemplazar un carácter o una subcadena en una cadena de caracteres. Toma dos argumentos, el carácter o la subcadena que se desea reemplazar y el nuevo carácter o subcadena que se desea usar en su lugar, y devuelve una nueva cadena de caracteres con el reemplazo realizado.

```
//
String s = "Hola mundo";

//retorna el texto ‘Hela munde’
String rep = s.replace('o','e');
```

Entrada y Salida

En programación, la entrada y salida -en inglés, Input and Output o simplemente I/O- es la comunicación entre un programa y el mundo exterior.

La entrada se refiere a los datos o cualquier tipo de información que un programa recibe, mientras que la salida se refiere a cualquier información que un programa produce.

La entrada puede llegar al programa a través de distintas formas, siendo las más frecuentes:

- Datos producidos por el teclado: el usuario ingresa información mediante el uso del teclado. Esta es una de las formas más utilizadas para interactuar con los programas.
- Datos producidos por el mouse: es un dispositivo de entrada que permite al usuario interactuar con diferentes partes del programa. En general este tipo de dispositivo desencadena eventos que son capturados por el programa.
- Datos de un archivo: el programa lee datos desde un archivo guardado en el disco rígido de la máquina.
- Datos de una red: el programa recibe datos a través de una conexión de red. Los datos provienen, en general, desde otra máquina.
- Datos producidos por sensores: el programa recibe información de sensores físicos.

La salida también puede tomar diferentes formas, algunas de las más frecuentas:

- La pantalla: el programa muestra información a través del dispositivo de salida más común, el monitor.
- Un archivo: el programa escribe datos en un archivo. La forma de escribir estos datos dependerá del tipo de información que se produzca.
- Datos en una red: el programa envía datos a través de una conexión de red. Los datos viajan por canales físicos o aéreos para llegar, generalmente, a otra máquina.
- Actuadores: el programa envía señales a actuadores físicos como motores, válvulas y relés entre otros.

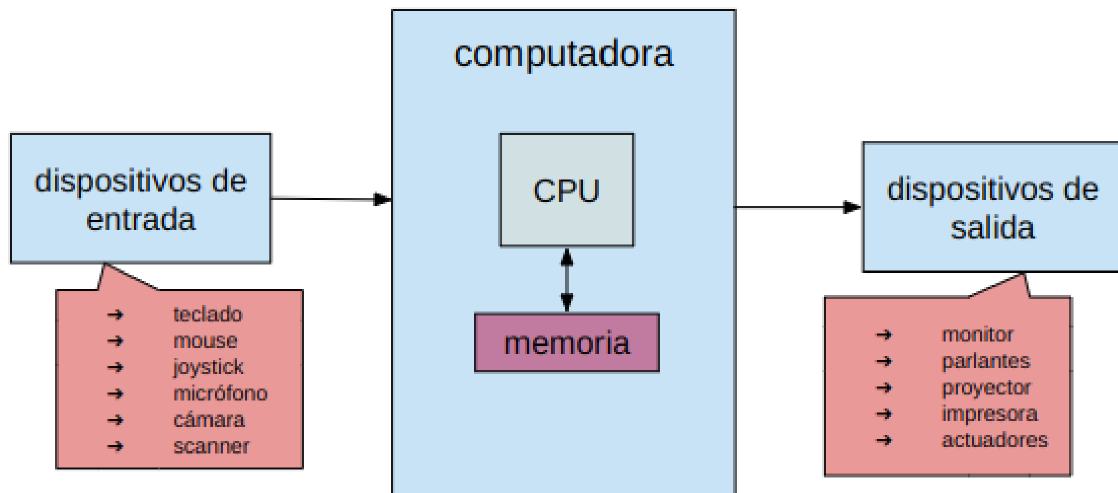


Fig.09 - entrada y salida.

La entrada y la salida son esenciales para que los programas interactúen con el mundo exterior, ya sea para leer datos y realizar cálculos con ellos, o para comunicarse con otros dispositivos o sistemas, enviando sus resultados. El manejo adecuado es fundamental para la creación de aplicaciones robustas y escalables.

Por el momento, nos enfocaremos en los dispositivos más comunes, el teclado y el monitor.

La entrada estándar y la salida estándar son canales de comunicación habituales en la programación, que nos permiten interactuar con el usuario o inclusive con otros programas.

En Java, estos canales se encuentran representados por dos objetos:

- **`System.in:`** este objeto representa la entrada estándar, generalmente asociada al teclado.
- **`System.out:`** este objeto representa la salida estándar, generalmente asociada a la consola o terminal.
- **`System.err:`** este objeto representa la salida de errores, también se la suele asociar a la consola.

[-] entrada

En java existen varias formas de obtener lo que el usuario ingresa a través del teclado.

Para obtener estos datos usaremos una de las maneras más sencillas que el lenguaje provee, obtener en una cadena de texto todos los símbolos ingresados a través del teclado.

> El teclado es el dispositivo de entrada más común al igual que el monitor lo es para la salida pero recordemos que no son los únicos.

Este método es sumamente simple dado que nos abstrae del proceso interno que se requiere para obtener el flujo de datos provenientes de la entrada estándar.

```
//
System.out.println("Ingrese su nombre: ");

//
Scanner lector = new Scanner(System.in);

//
String nombre = lector.nextLine();

//
System.out.println("Buenos días "+nombre);
```

Por el momento diremos que la segunda línea que contiene "`Scanner lector = new Scanner(System.in);`" es necesaria para acceder luego al texto que el usuario ingresa pero no profundizaremos más que eso.

La tercera línea, la que contiene **"`String nombre = scanner.nextLine();`"** es la que nos retornará en formato de texto los datos que el usuario ingresó hasta que dió un enter.

Finalmente se aporta el programa completo para mostrar la presencia de una línea especial en la cabecera del programa, se trata de la librería que nos permitirá que la función de lectura sea posible **"`import java.util.Scanner;`"**.

```
import java.util.Scanner;

public class HolaMundo {

   public static void main(String[] args){

        //
        System.out.println("Ingrese su nombre: ");

        //
        Scanner lector = new Scanner(System.in);

        //
        String nombre = lector.nextLine();

        //
        System.out.println("Buenos días "+nombre);

   }

}
```

[-] salida

Como hemos visto en el programa, la salida se produce con la sentencia "`System.out.println("Buenos días "+nombre);`"

Si el nombre fuese "Pepe", en la pantalla se vería la frase "Buenos días pepe".

Utilizamos el término "pantalla" para describir lo que se produce en la salida estándar del sistema. Generalmente esta salida, junto a la de errores, se asocia a una terminal.

Cuando se construye un programa o sistema que utiliza una interface gráfica o se utiliza en la web, la salida estándar y la de errores se suele redirigir a un archivo a modo de registro

de lo que sucede. Muchas veces este registro se convierte en el log del sistema.

Nuestros programas, al menos los que podremos realizar según el alcance del presente libro, utilizarán la salida estándar para mostrar y solicitar los datos del usuario.

Para mostrar en la salida estándar texto tendremos básicamente dos opciones:

Salida simple: utilizamos el método **`System.out.println().`** Imprime una línea de texto en la consola.

- `System.out.println("Hola mundo");`
- `System.out.println(variable);`
- `System.out.println("Hola mundo" + variable);`

Salida formateada: utiliza el método System.out.printf(). Permite mayor control sobre cómo queremos formatear la salida. Se utilizan marcas para especificar cómo debe ser tratado el dato que pasamos. Los más habituales son %d, %f y %s.

- `System.out.printf("un entero %d\n", varEntera);`
- `System.out.printf("un decimal %f\n", varDecimal);`
- `System.out.printf("un texto %s\n", varTexto);`

Notar que al final de cada argumento se utiliza "**`\n`**", esto es porque al método, a diferencia de "println", hay que especificar cuándo realizar un salto de línea.

A continuación mostramos algunas de las marcas o especificadores posibles.

Los siguiente se utilizan para reemplazar los valores según su tipo:

- %d: enteros
- %f: decimales
- %s: texto
- %c: caracteres
- %x: enteros en formato hexadecimal
- %o: enteros en formato octal
- %b: enteros en formato binario

Los que se utilizan para dar formato con más detalle:

- -: alinear a la izquierda
- +: agrega un signo positivo (+) delante del número
- 0: rellenar con ceros
- .: separar la parte entera de la parte decimal
- #: agregar un prefijo (0x para hexadecimal, 0 para octal)
- N: especificar el ancho mínimo del campo
- .M: especificar la precisión mínima de los decimales

Como hemos mencionado, la salida estándar se encuentra habitualmente visible a través de una terminal o consola.

A modo de "juego" para quien está comenzando y quisiera hacer programas distintos, dejamos algunas secuencias de caracteres llamada Secuencia ANSI o códigos de escape.

Las secuencias ANSI, también conocidas como códigos de escape ANSI, son una serie de caracteres especiales que se utilizan para controlar el aspecto del texto en la terminal.

Estas secuencias no imprimen ningún carácter visible en sí mismas, sino que envían instrucciones a la terminal para modificar la apariencia del texto que se quiere mostrar.

Las secuencias ANSI están formadas por el caracter de escape, representado por el código ASCII 27, en decimal, o \033 en cadenas de texto, seguido de un corchete de apertura [y una serie de parámetros que definen el aspecto que se quiere.

Por ejemplo, para cambiar el color del texto a rojo, podríamos utilizar la siguiente secuencia:

```
\033[31
```

El código lo insertamos delante del texto que queremos que salga en rojo:

```
System.out.println("\033[31mTexto en rojo");
```

Colores de texto:

- Negro: \033[30m
- Rojo: \033[31m
- Verde: \033[32m

- Amarillo: \033[33m
- Azul: \033[34m
- Cian: \033[36m
- Magenta: \033[35m
- Blanco: \033[37m

Colores de fondo:

- Negro: \033[40m
- Rojo: \033[41m
- Verde: \033[42m
- Amarillo: \033[43m
- Azul: \033[44m
- Cian: \033[46m
- Magenta: \033[45m
- Blanco: \033[47m

También se puede combinar, el color y el fondo:

- Texto rojo con fondo verde: \033[31m\033[42m

Siguiendo esta misma línea de “juego”, otro aspecto interesante es que se pueden enviar símbolos a la terminal del tipo “emoji” utilizando el código unicode específico:

```
System.out.println("\u26F1");
```

Sale por pantalla:

Todo lo anterior será posible si la terminal entiende texto en formato unicode e implementa la secuencia ANSI, caso contrario no se verán los emojis ni los colores.

Ejercicios

1. Crear la variable nombre y asignar tu nombre completo. Mostrar por pantalla como quedaría en MAYÚSCULAS.
2. Escribe un programa que declare 5 variables de tipo char. A continuación, crear otra variable como cadena de texto y prueba de asignar como valor a esta última la concatenación de las anteriores 5 variables de caracteres. Por último, mostrar la cadena de caracteres por pantalla.

   ¿Se presentó algún problema? si así sucedió, ¿ cómo lo solucionaste?

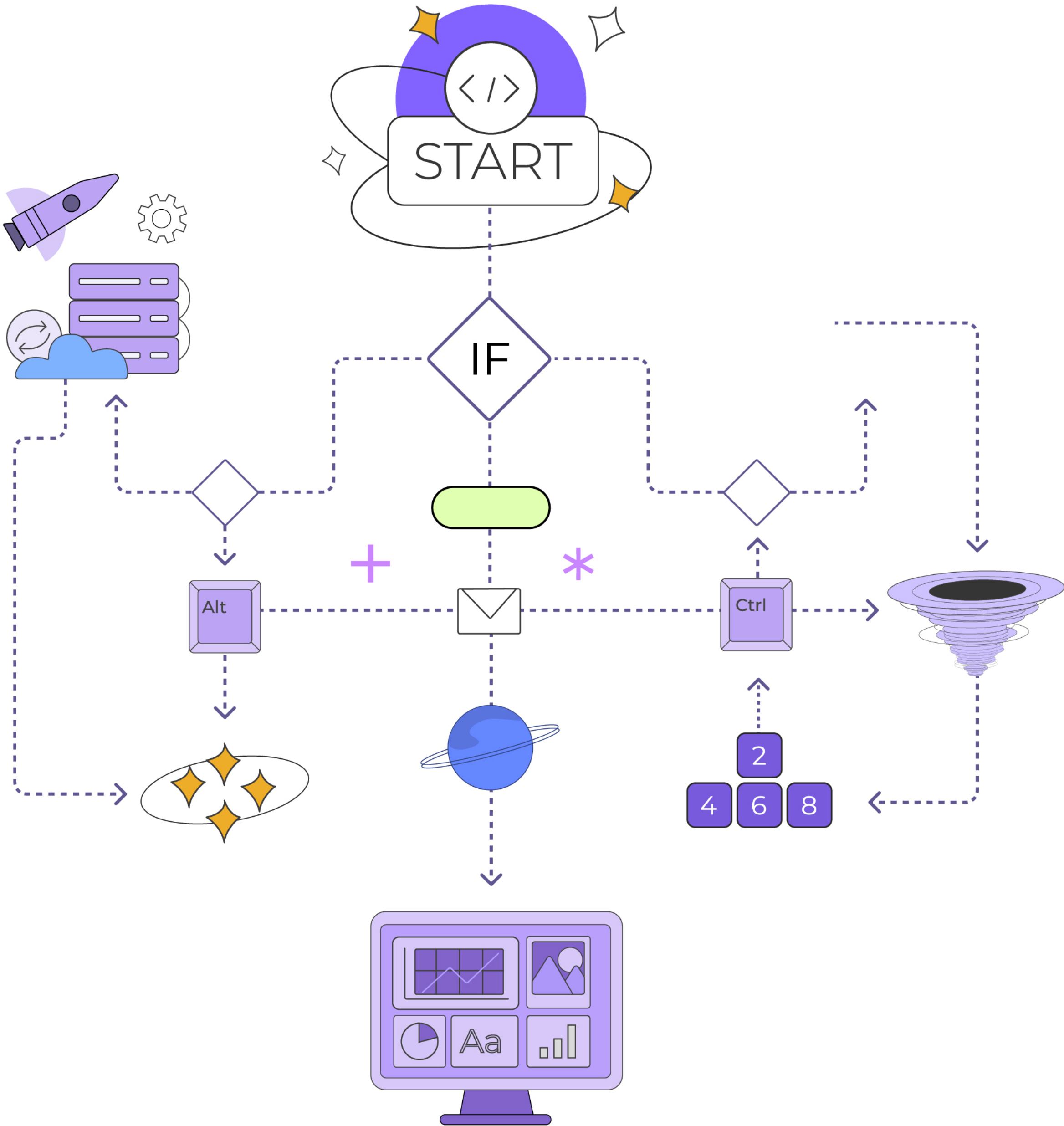
3. Pedir por pantalla el nombre y apellido de una persona. Mostrar por pantalla como serían sus iniciales. Asuma que la persona sólo tiene un nombre y un apellido.

   Ej: Alfonzo Zarate → AZ

4. Escribir un programa que pida al usuario un número de 4 dígitos y visualice por pantalla, en cada línea, cada dígito que lo compone.
5. Realizar un conversor de dólares a pesos. La cantidad de dólares debe ser introducida por teclado.
6. Escribir un programa que calcule el perímetro y área de un rectángulo. Los datos deben leerse desde el teclado.
7. Escribir un programa que calcule el perímetro y área de un triángulo. Los datos deben leerse desde el teclado.
8. Escribir un programa que calcule el sueldo mensual de un empleado en base a las horas trabajadas. Ambos datos se deben ingresar por teclado.
9. Realizar un conversor, que dado un valor expresado en Mb informe Gb, Kb y bytes.

Capítulo 4

Estructuras de control



Capítulo 4

Estructuras de control

Objetivo

El capítulo tiene como objetivo desarrollar uno de los temas centrales de la programación, las estructuras de control. Estas herramientas nos permiten tomar decisiones y ejecutar diferentes acciones en función de las condiciones que se presenten, otorgando a nuestros programas la flexibilidad y el dinamismo necesarios para resolver problemas complejos. Cómo tema adicional, se explicará la importancia de saber interpretar algoritmos que podrían no haber sido escritos por nosotros.

Al finalizar este capítulo, estarás capacitado para realizar programas donde la ejecución se encuentre totalmente controlada. Habrás adquirido una comprensión profunda de las diferentes estructuras disponibles, su funcionamiento y posibles aplicaciones. Además, estarás en condiciones de diseñar soluciones lógicas a problemas diversos, leer e interpretar algoritmos.

Flujo de ejecución

En programación, el flujo de ejecución se refiere al orden en que se ejecutan las instrucciones de un programa. Este orden puede ser lineal, es decir, una instrucción después de otra, o puede ser condicional o iterativo, dependiendo de las decisiones que se tomen durante la ejecución del programa.

Por lo tanto podemos definir a las estructuras de control como herramientas que permiten controlar el flujo de ejecución o control algorítmico. Estas herramientas se pueden clasificar en dos grupos, las que permiten tomar una decisión y las que permiten repetir un conjunto de instrucciones.

Una aclaración interesante que podemos mencionar en este momento es la diferencia que existe entre los términos "sentencia" e "instrucción", dado que a lo largo del libro se utilizan a menudo de forma indistinta para referirnos a las acciones que se ejecutarán.

Existe una sutil diferencia entre ambos términos:

- Instrucción: es la unidad básica de código que la computadora puede ejecutar. Es una orden simple y directa que no produce un valor por sí misma.

- Sentencia: es una unidad de código completa que puede producir un valor o realizar una acción. Una sentencia puede estar compuesta por una o más instrucciones.

Secuencia

Es una estructura de control especial.

Como mencionamos, las estructuras de control son herramientas que nos permiten controlar el flujo de ejecución de un programa, aunque existe una caso particular: la secuencia.

La secuencia la consideramos una estructura de control implícita. Es la base sobre la que se construyen las demás estructuras. Determina el orden en que se ejecutan las instrucciones, una después de otra, y no requiere palabras clave especiales para definirse porque se ejecuta de forma implícita en la mayoría de los lenguajes de programación.

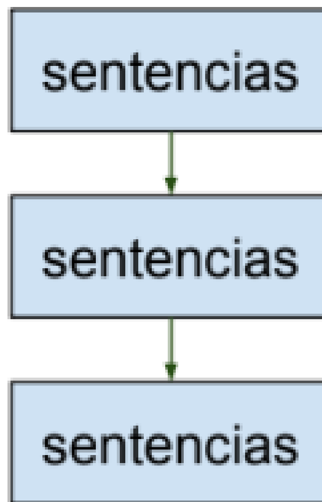


Fig. 10 - secuencia.

Selección

La estructura de control de selección permite que un algoritmo tome una decisión basada en una condición lógica y ejecute diferentes bloques de código según el resultado de la evaluación de la condición.

En diversas situaciones podríamos necesitar realizar un conjunto de acciones ante un determinado valor, ya sea el de una variable, una constante, una expresión o una función.

La sentencia de selección, justamente, nos aporta este comportamiento.

Se expresa mediante la palabra reservada “**if**” y se compone de la condición y el conjunto de acciones que se pretende realizar.

```
//se evalúa la expresión, si es verdadera ingresa al bloque
if ( expr ) {
   //sentencias a ejecutar
}
```

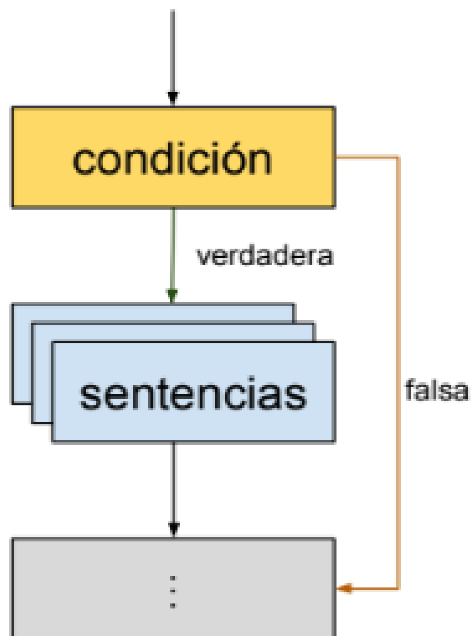


Fig.11 - estructura de selección.

Veamos un ejemplo de utilización:

```
//
System.out.println("Ingrese su edad: ");

//
Scanner lector = new Scanner(System.in);

//
int edad = lector.nextInt();

//
if (edad >= 18) {
   System.out.println("... mayor de edad");
}
```

Hay otras situaciones en las cuales existe un conjunto de acciones que se deben ejecutar cuando la condición ha sido evaluada como falsa. Dicho de otro modo, es lo que quiero realizar si no es verdadera la condición.

Java también nos brinda la posibilidad de expresar esta situación alternativa.

Se utiliza la palabra reservada “else” luego de terminar la parte de la sentencia que contiene el conjunto de acciones que se realizarán si la evaluación dió verdadero.

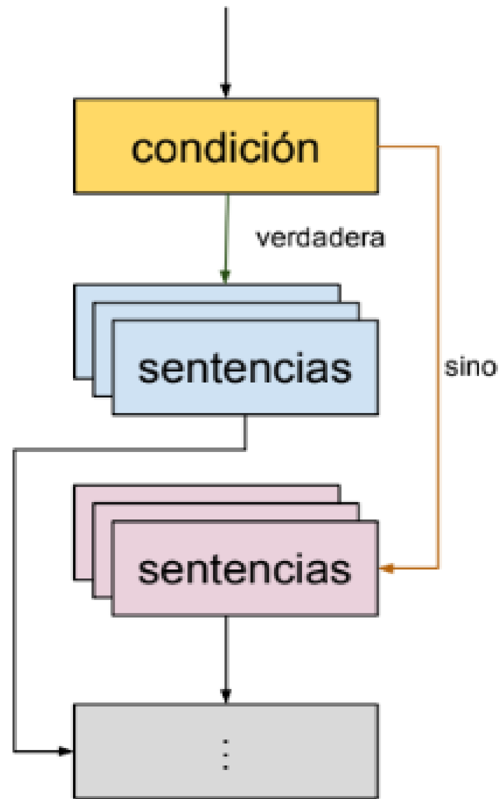


Fig.12 - estructura de selección con alternativa.

Veamos un ejemplo de utilización:

```
//
System.out.println("Ingrese su edad: ");

//
Scanner lector = new Scanner(System.in);

//
int edad = lector.nextInt();
```

```java
//
if (edad >= 18) {
    System.out.println("... mayor de edad");
} else {
    System.out.println("... menor de edad");
}
```

Por supuesto que la sentencia de selección se puede encadenar pudiendo establecer varios conjuntos de acciones en función de determinadas condiciones. Podemos anidar los caminos alternativos de ejecución asociados a cada uno.

Cada vez que una condición es evaluada y el resultado es falso se continúa con la siguiente parte de la sentencia. Si una condición es evaluada como verdadera se ejecutan sus acciones y luego termina la sentencia, o sea, no continua verificando el resto de las condiciones.

El encadenamiento se realiza mediante la construcción "**else if**".

Ejemplo de cómo podemos anidar o encadenar las sentencias:

```java
…

//
if (edad < 15) {

    System.out.println("...clase A");

} else if(edad >= 15 && edad <= 17) {

    System.out.println("...clase B");

} else if(edad >= 18 && edad <= 21) {

    System.out.println("...clase C");

} else {


    System.out.println("...clase D");

}
```

Repetición

La estructura de control de repetición permite a un algoritmo repetir un conjunto de instrucciones o sentencias.

Dependiendo de cuanta información tenga al momento de diseñar el algoritmo podré utilizar una estructura de repetición con un número específico de veces o una estructura de repetición que detendrá su ciclo en función de una condición lógica.

[-] for

Esta estructura se utiliza principalmente cuando el número de veces a repetir es conocido con anterioridad a la ejecución o puede determinarse a partir de un determinado conjunto de datos.

El conjunto de sentencias se ejecuta un número específico de veces.

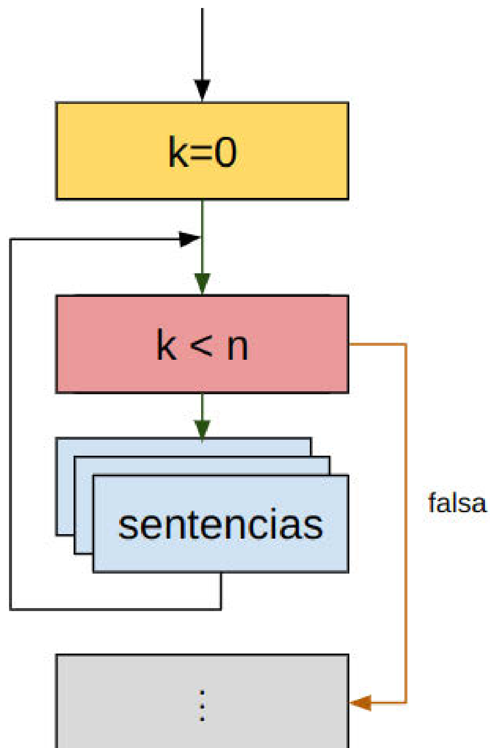


Fig.13 - estructura de repetición for.

Veamos la forma general de la estructura.

```
//
for (inicialización; condición; actualización) {
    //sentencias
}
```

Notar que el for tiene tres partes:

1. Inicialización: se ejecuta una única vez al inicio del bucle. Se utiliza para inicializar variables que se usarán dentro del bucle.
2. Condición: se evalúa antes de cada iteración del bucle. Si la condición es verdadera, se ejecuta el bloque de código del bucle. Si la condición es falsa, el bucle termina.
3. Actualización: se ejecuta al final de cada repetición del bucle. Se utiliza para actualizar las variables que se usan dentro del bucle.

¿Cómo se puede utilizar la estructura para mostrar en la pantalla 10 veces la frase "hola mundo..." ?

```
//
for (int i=1; i <= 10; i++){

   //
  System.out.println("Hola mundo…");
}
```

Existe un caso especial de uso de la estructura "for each" que se presenta cuando utilizamos la estructura para recorrer los elementos de un contenedor. Desarrollaremos este tema más adelante cuando veamos contenedores.

[-] while y do-while

Este tipo de estructura, tanto el while como el do-while, se utilizan cuando la cantidad de veces que se quiere repetir se encuentra determinada por algo que debe ocurrir, o sea, una

expresión lógica debe dar determinado resultado para que termine la repetición.

El conjunto de sentencias se ejecuta hasta que la condición sea falsa o dicho de otra manera, mientras la condición sea verdadera.

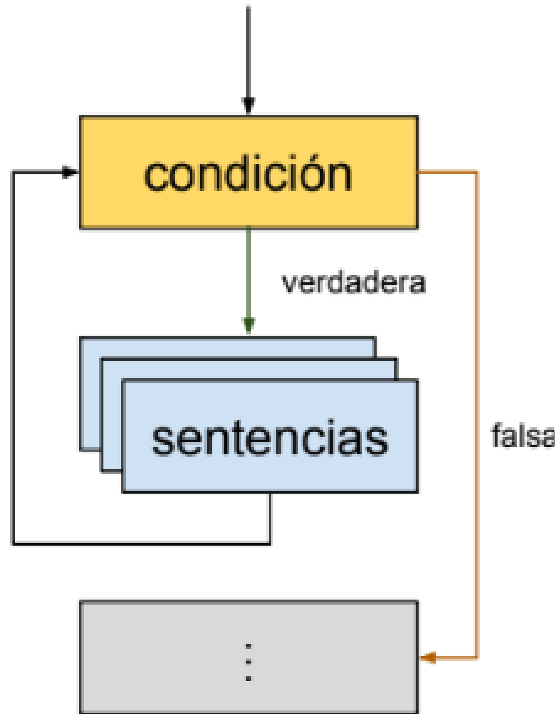


Fig.14 - estructura de repetición while.

Veamos la forma general de la estructura.

```
//
while( expr == true ){

    //sentencias

}
```

El while contiene una expresión que determinará si debe continuar ejecutando las sentencias o tiene que terminar y continuar el flujo normal de ejecución.

Es importante que en el cuerpo de la estructura exista al menos un elemento, una variables o función que modifique algo de la expresión que determina si debo continuar repitiendo o no.

Volvamos al ejemplo anterior, en el que mostramos por pantalla los números del 1 al 10. Realicemos el programa pero ahora con el while:

```
//
int numero = 1;

//
while(numero <= 10){

   System.out.println("El número es: "+numero);

   //
   numero++;
}
```

Como se puede apreciar, antes del ingreso al conjunto de sentencias que podrían repetirse se evalúa la condición de la estructura y en caso de que la misma sea falsa no se ejecuta ninguna de las sentencias contenidas.

Si quisiéramos ingresar al menos una vez tendríamos que utilizar algún artilugio algorítmico para lograrlo, que si bien es posible lo cierto es que nos degrada la legibilidad del algoritmo.

Para evitar tener que poner más código a la estructura, algunos lenguajes como Java nos brindan la posibilidad de ejecutar primero el conjunto de sentencias y al final evaluar la expresión de repetición. De esta manera averiguamos al final si tenemos que repetir o finalizar la estructura.

Entonces do-while lo utilizaremos cuando queremos ejecutar el conjunto de sentencias y luego determinar a través de una expresión lógica su repetición.

El conjunto de sentencias se ejecuta hasta que la condición sea falsa o, dicho de otra manera, mientras sea verdadera.

A continuación se muestra la forma general de la estructura. Notar que las sentencias si o si al menos una vez se ejecutan, luego se verifica si debe repetirse.

```
//
do {
    //sentencias
} while( expr == true )
```

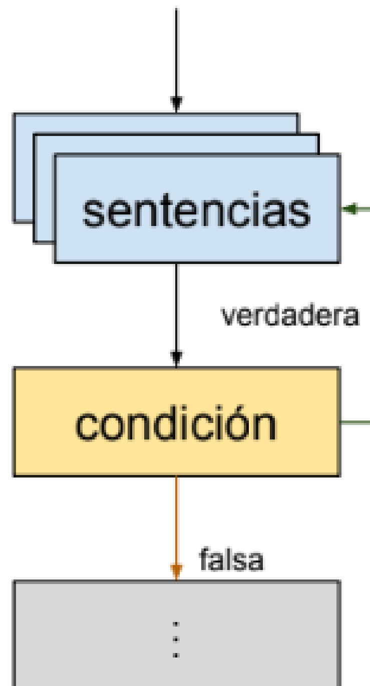


Fig.15 - estructura de repetición do-while.

El siguiente código muestra el cuadrado de los números ingresados por el usuario hasta que se ingresa un valor mayor a 10.

```
//
Scanner sc = new Scanner(System.in);
int num;
```

```
//
do{
    //
    System.out.print("Ingrese un número: ");
    num = sc.nextInt();
    //
    System.out.println(cuadrado: " + (num*num));
}while(num <= 10);
```

El conjunto de sentencias de la repetición se ejecutara 1 o más veces si utilizo do-while y 0 o más veces si utilizo while

La principal diferencia que existe entre ambas estructuras es la posibilidad de ingresar o no a la estructura. En el caso del while puede suceder que no se ingrese al bloque de sentencias por haber sido evaluada su condición y dar como resultado "falso". Por el contrario, con el do-while, siempre ingresará a ejecutar el bloque se sentencias al menos una vez, luego evaluará la condición y dependiendo del resultado volverá a ejecutar las sentencias o no.

Estructuras adicionales control

El lenguaje Java, como también sucede con la mayoría de los lenguajes de programación, aporta estructuras de control específicas que mejoran la legibilidad algorítmica y en algunos casos mejoran la eficiencia.

[-] break

Es una palabra reservada que tiene el lenguaje para indicar que se quiere romper el flujo de ejecución.

Se puede utilizar en el interior de un for, un while, un do-while o un switch.

En la ejecución, cuando el intérprete encuentra este símbolo, sale de la estructura y continúa la ejecución de las sentencias siguientes.

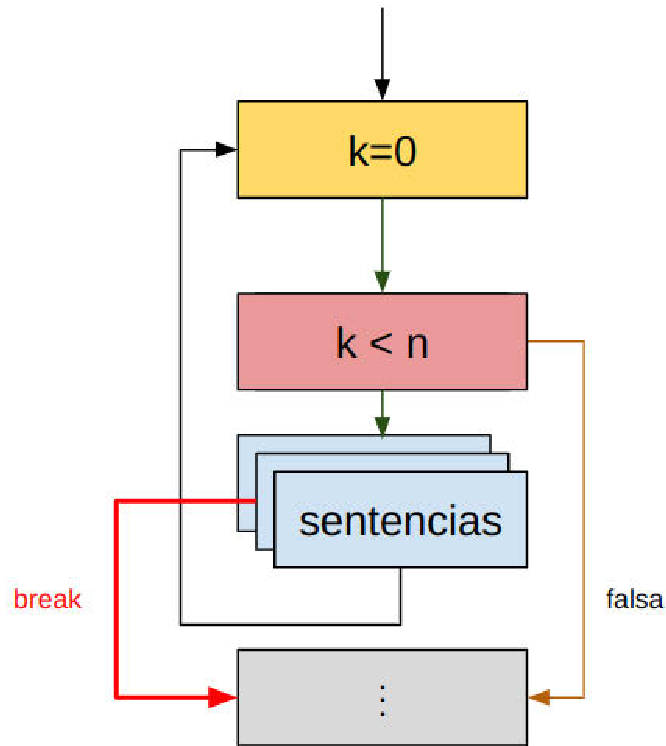


Fig.16 - for con break.

En el siguiente ejemplo se corta la repetición del for si la variable "num" toma el valor 5.

```
//
for (int num = 1; num <= 10; num++) {

    //
    System.out.println(num);

    //
    if (num == 5) {
        break;
    }

}
```

Saldrá por pantalla:

1

2

3

4

5

[-] continue

Es una palabra reservada que tiene el lenguaje para indicar que se quiere alterar el flujo de ejecución.

Se utiliza para saltar al próximo ciclo en una estructura de control de repetición.

Se puede utilizar con las estructuras de control for, while o do-while

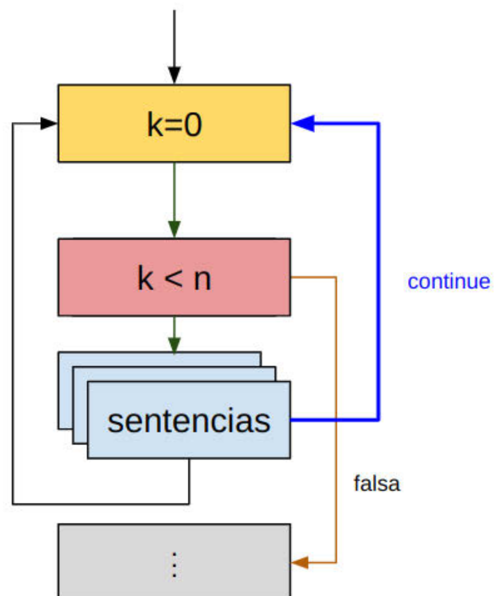


Fig.17 - for con continue.

En el siguiente ejemplo sólo se imprimen los números impares, cada vez que la variable sea par se pasa a la siguiente iteración del for.

```
for (int num = 1; num <= 10; num++) {

   if (num % 2 == 0) {

       continue;

   }

   System.out.println(num);

}
```

Saldrá por pantalla:

1

3

5

7

9

[-] switch

Se utiliza para evaluar una variable o expresión y ejecutar un bloque de código específico en función del valor resultante. Es una estructura de control que permite tomar decisiones, escribiendo las distintas posibilidades de forma más clara que con múltiples if-else anidados.

El switch en Java también tiene la posibilidad de especificar un bloque de sentencias por defecto si el valor no se encuentra especificado en ningún caso.

Se muestra a continuación el uso del switch para visualizar en pantalla qué día representa, en texto, el valor de la variable entera:

```java
//
int dia = 2;

//
switch (dia){
   case 1:
       System.out.println("lunes");
       break;
   case 2:
       System.out.println("martes");
       break;
   case 3:
       System.out.println("miércoles");
       break;
   case 4:
       System.out.println("jueves");
       break;
   case 5:
       System.out.println("viernes");
       break;
   case 6:
       System.out.println("sábado");
       break;
   case 7:
       System.out.println("domingo");
       break;
   //...
   default:
       System.out.println("día inválido");
       break;
}
```

[-] try-catch-finally

Se utiliza para manejar excepciones. Este tema no lo profundizaremos, sólo diremos por el momento que nos servirá para capturar situaciones que no son las habituales para nuestros algoritmos. En vez de que se produzca un error y este corte la ejecución del programa podremos capturarlo, tratarlo y eventualmente continuar con la ejecución.

Un ejemplo de utilización:

```
//
Scanner scanner = new Scanner(System.in);

//
System.out.print("Ingrese un número: ");

try {
   //
   int dia = scanner.nextInt();

   //
   System.out.println("El número ingresado es: "+dia);

}catch(InputMismatchException ex) {

   //
   System.out.println("el número no es válido");
}
```

Otro ejemplo:

```
try {

    int resultado = 10 / 0;

} catch(ArithmeticException e) {

    System.out.println("Error: " + e.getMessage());

} finally {

    System.out.println("Siempre se ejecuta");

}
```

[-] for-each

Se utiliza para recorrer una colección de elementos. Se desarrollará con mayor profundidad cuando se vean contenedores.

Ejemplo:

```
//
for (Integer elemento : coleccion ) {

   // sentencias
}
```

Traza de ejecución

La traza de ejecución es el seguimiento detallado de cómo un algoritmo se ejecuta paso a paso. Es especialmente útil para depurar y comprender algoritmos.

Al observar la traza, podremos ver cómo los datos se transforman a medida que el algoritmo avanza y cómo se modifican las variables en cada paso.

La traza es útil para descubrir Bugs.

> Se cree que esta palabra "bug" adquirió su significado en la década del 40 cuando Grace Hopper, una pionera en la programación, encontró un error en el Mark II, una computadora electromecánica.
>
> El equipo de Hopper descubrió que el mal funcionamiento era causado por un escarabajo que se había alojado en uno de los relés.

[-] realizar una traza

Realizar una traza implica seguir paso a paso la ejecución de un algoritmo, registrando los cambios en las variables de interés y, opcionalmente, registrando también el resultado de la expresiones obtenidas en cada paso.

Pasos a seguir:

- Identificar las variables, constantes y parámetros que se encuentren involucrados en el fragmento de código que se quiera seguir.
- Asignar los valores iniciales.
- Seguir en detalle los pasos del algoritmo, el flujo de ejecución.
- Registrar los resultados a medida que el algoritmo avanza.
- Repetir los pasos de la traza hasta terminar el algoritmo.
- Verificar la corrección del algoritmo o entender qué realiza el fragmento de código.

Para su realización se puede utilizar cualquier elemento que sirva de ayuda memoria, por ejemplo una hoja de papel, una pizarra o una planilla de cálculo. Por supuesto que los

IDE's modernos aportan alguna herramienta para hacer el seguimiento de los algoritmos a través del mismo entorno o inclusive permiten la integración de otras herramientas como GDB (GNU Debugger) o JDB (JAVA Debugger) para hacer la verificación del código -también llamada depuración-, ejecutando el algoritmo paso a paso y permitiéndonos ver el contenido de las variables o marcar líneas para que la ejecución se detenga.

Veamos un ejemplo para ver cómo se podría realizar el seguimiento de un algoritmo a través de una tabla.

línea	código
1	`Scanner lector = new Scanner(System.in);`
2	`int numero = lector.nextInt();`
3	`int resultado = 1;`
4	`for(int i = 1; i <= numero; i++){`
5	`    resultado = resultado * i;`
6	`}`
7	`System.out.println("El resultado es: " + resultado);`

Nos interesa observar las variables número, resultado e i. Lector no lo vamos a observar por el momento, sólo nos permite obtener la entrada de datos del usuario.

La 1er línea prepara el lector de datos.

La 2da asigna el valor introducido por el usuario a la variable número, aquí supondremos un valor que nos parezca razonable, por ejemplo 5.

Iremos registrando los valores en cada fila a medida que vayamos avanzando en el programa.

número	resultado	i	i <= numero	línea a ejecutar
5				`int numero = lector.nextInt();`
	1			`int resultado = 1;`
		1		`for(int i = 1; … ; … ){`
			true	`for( … i <= numero; …)`
	1 = (1*1)			`resultado = resultado * i;`
		2		`for( … ; … ; i++)`
			true	`for( … i <= numero; …)`

	2 = (1*2)			resultado = resultado * i;
		3		for(… ; … ; i++)
			true	for(… i <= numero; …)
	6 = (2*3)			resultado = resultado * i;
		4		for(… ; … ; i++)
			true	for(… i <= numero; …)
	24 = (6*4)			resultado = resultado * i;
		5		for(… ; … ; i++)
			true	for(… i <= numero; …)
	120 = (24* 5)			resultado = resultado * i;
		6		for(… ; … ; i++)
			false	for(… i <= numero; …)
				println("El resultado es: " + resultado);

El algoritmo imprime en la pantalla El resultado es: 120 y termina.

Luego de finalizado el seguimiento, se tendrá una idea más precisa de los valores que han surgido y cómo se han distribuido entre las distintas variables. Ahora es más sencillo determinar que realiza.

Pregunta: ¿Qué realiza el algoritmo?

Respuesta: El factorial de un número.

Puede pasar que al concluir la traza o seguimiento del algoritmo no pueda determinar con certeza qué realiza. Una estrategia es volver a realizar el seguimiento con distintos valo-res, en nuestro caso podríamos probar con 4 o 6. Se debe tener en cuenta la naturaleza del fragmento que estamos depurando. Por ejemplo, al ver que existen ciclos de repetición y estos dependen del valor de entrada que le asignemos, sería prudente establecer valores que no nos generen dificultades en el seguimiento.

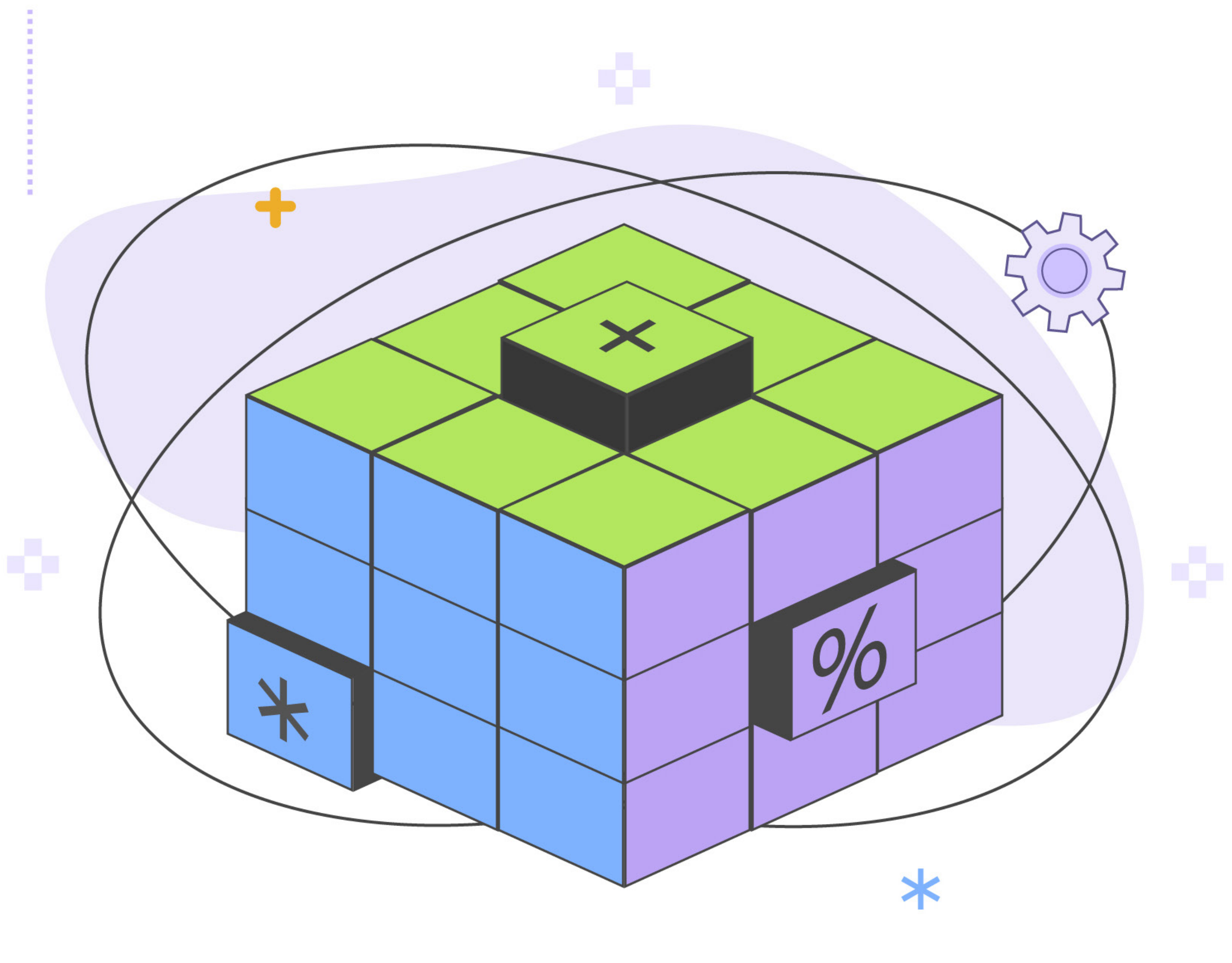
Realizar una traza de ejecución es una herramienta vital para la depuración de programas, la comprensión del comportamiento del código y la validación de la lógica de un programa.

Ejercicios

1. Escribir un programa en que dado un número del 1 al 7 escriba el correspondiente nombre del día de la semana.
2. Escribir un programa que pida una hora por teclado, en formato 24, y mostrar "buenos días", "buenas tardes" o "buenas noches" según los tramos de 6 a 12, de 13 a 20 y de 21 a 5. respectivamente.
3. Escribir un programa que calcule el sueldo mensual de un empleado en base a las horas trabajadas y al precio de la hora. Ambos datos se deben ingresar por teclado. Se debe tener en cuenta que las primeras 40 hs se pagan al precio de la hora y las excedentes al doble
4. Mostrar los números múltiples de 5 de 0 a 100 utilizando for.
   ej: 0, 5, 10, ..,100
5. Mostrar los números múltiples de 5 de 0 a 100 utilizando while.
6. Mostrar los números múltiples de 5 de 0 a 100 utilizando do-while.
7. Mostrar los números del 400 a 200, contando de 20 en 20 hacia atrás utilizando for.
8. Escribir un programa que calcule el factorial de un número entero leído por teclado.
   ej: factorial de 4 = 4x3x2x1 = 24
9. Realizar el control de acceso a una caja fuerte. Se deben pedir números hasta que coincida con el número de 4 dígitos establecido previamente. Ante cada intento desacertado deberá salir un mensaje con el texto “Lo siento, intente nuevamente...”.
10. Realizar el control de acceso a una caja fuerte. Se deben pedir números hasta que coincida con el número de 4 dígitos establecido previamente. Ante cada intento desacertado deberá salir un mensaje con el texto “Lo siento”.
11. En esta ocasión se tendrá sólo 4 intentos, en caso de fallar todos deberá salir un mensaje con el texto "Llamando a la policía..."

Capítulo 5

Estructuras de datos compuestas



Capítulo 5

Estructuras de datos compuestas

Objetivo

En este capítulo, exploramos el mundo de las estructuras de datos estáticas, como arreglos y matrices. Estas herramientas son fundamentales para almacenar y organizar datos de forma eficiente en nuestros programas. Aprenderemos a usarlas para resolver diversos problemas, desde el almacenamiento de información simple hasta la creación de estructuras complejas. Además, abordaremos las operaciones básicas como agregar, modificar, buscar y eliminar elementos.

Nos centraremos en estructuras de datos estáticas, que tienen un tamaño predefinido y que no puede cambiar durante la ejecución del programa.

Al finalizar este capítulo, estarás capacitado para utilizar las estructuras de datos estáticas y compuestas para modelar situaciones y resolver de manera eficiente muchos de los problemas que suelen surgir. Estarás en condiciones de seleccionar la estructura más adecuada para una gran diversidad de casos de uso.

Datos compuestos

Las estructuras de datos compuestas son estructuras que se construyen a partir de otras estructuras, por ejemplo de datos simples como los números, cadenas de texto, valores lógicos o caracteres.

Se utilizan para representar información más compleja, que se organiza y almacena en memoria de manera que se pueda acceder y manipular fácilmente.

En particular avanzaremos en los datos compuestos caracterizados por ser homogéneos e indexados.

Arreglos

El tipo de dato arreglo, es una colección ordenada e indexada de elementos, con las siguientes características:

- Todos los elementos son del mismo tipo. Esto le da la característica de ser un tipo de dato homogéneo.
- Los elementos pueden recuperarse en cualquier orden, se indica la posición que ocupan dentro de la estructura. Esto le da la característica de ser una estructura indexada.
- La memoria ocupada a lo largo de la ejecución del programa es fija. Esto le da la característica de ser una estructura estática.

Es importante destacar que se accede a cualquiera de los elementos de la colección a través del nombre del identificador de la variable y la posición que ocupa.

Los arreglos pueden ser de distintas dimensiones. La cantidad de dimensiones indica la cantidad de índices necesarios para acceder a un elemento.

En nuestra disciplina, el término "arreglo" es la traducción frecuentemente utilizada para referirse a un "array". Probablemente surgió como un intento de encontrar una palabra en español que capturara la idea de una colección ordenada de elementos, aunque "formación" podría ser una traducción más precisa desde un punto de vista etimológico (relacionado con "formar" una estructura).

[-] vectores

La forma más simple de un arreglo se denomina vector o arreglo unidimensional. Como su nombre lo indica, es una estructura que tendrá una única dimensión.

El siguiente gráfico muestra un vector de 6 elementos de números enteros.

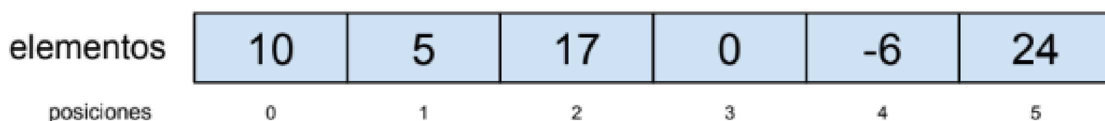


Fig.18 - vector.

Para utilizar este tipo de dato utilizaremos la sintaxis de la siguiente manera:

```
tipo [] vector = new tipo[capacidad];
```

El tipo de los elementos de un arreglo puede ser cualquiera de los que hemos visto hasta el momento, por ejemplo podríamos querer un vector de elementos enteros, para ello usaremos el tipo int. Para especificar la cantidad o "capacidad" de elementos usaremos una expresión del tipo ordinal, esto es, un tipo de dato en el cual sus valores se pueden contar y se les puede asignar un orden específico.

```
int [] vector = new int[10];
```

Tal como mencionamos con anterioridad, para acceder a los elementos de un vector debemos hacer referencia a su posición. Cada elemento del vector ocupa un lugar en la memoria de forma consecutiva, por lo tanto accederemos al lugar en la memoria a través del nombre de su variable y luego el desplazamiento indicado a través de su posición.

Por ejemplo si quisiéramos acceder al primer elemento del vector para guardar el valor en una variable, siempre del mismo tipo, lo podríamos hacer de la siguiente manera:

```
int variable = vector[0];
```

Se puede modificar el valor de cualquier elemento del vector respetando el tipo de dato asociado.

El siguiente ejemplo muestra la asignación de valores a cada unos de los elementos del vector:

```
//
int[] elementos = new int[6];

//
elementos[0] = 10;
elementos[1] =  5;
elementos[2] = 17;
elementos[3] =  0;
elementos[4] = -6;
elementos[5] = 24;
```

Hay que tener presente el tipo de dato asociado al vector, dado que el mismo será el que

restrinja qué operaciones pueden llevarse a cabo.

En el siguiente fragmento mostramos los valores de cada elemento en la pantalla:

```
//
System.out.print("Los valores del vector son: ");
System.out.print(elementos[0] +", " + elementos[1] +", ");
System.out.print(elementos[2] +", " + elementos[3] +", ");
System.out.print(elementos[4] +", " + elementos[5] );
```

El siguiente fragmento muestra cómo sumar el primer y último elemento del vector, para almacenarlo en una variable. Luego se muestra el valor de la suma por pantalla:

```
//
int suma = elementos[0] + elementos[5];

//
System.out.println("\nLa suma del 1ro + el 6to:" + suma);
```

Se puede notar, a través de los ejemplos, que los índices en JAVA comienzan a partir de 0 (cero).

Por el momento diremos que la palabra reservada "new" se utiliza para reclamar el espacio de memoria que se requiere para este vector.

Otra manera de acceder a un elemento es a través del uso de una expresión para, luego de su evaluación, indicar la posición que se quiere.

Veamos cómo podemos usar una expresión, en este caso el medio, para indicar la posición:

```
//
final int N = 6;


//
int[] elementos = new int[N];
```

```
//
int medio = N / 2;

//
System.out.println("El elemento ‘central’ es:" +
                    elementos[medio]);
```

Si quisiéramos mostrar por pantalla todos los elementos del vector podemos utilizar la estructura de control de repetición.

Recordar que si conocemos de antemano la cantidad de repeticiones que vamos a realizar podemos utilizar la estructura de repetición “for”.

El fragmento recorre todos los elementos del vector para mostrar por pantalla el valor de cada uno:

```
//
System.out.println("Los valores del vector son:");

//
for(int pos=0; pos<6; pos++){

   //
   System.out.println(elementos[pos]);
}
```

La inicialización estática es otro modo de crear un vector. Este modo puede resultar especialmente útil para resolver ciertas situaciones o probar código con valores arbitrarios.

Si bien los valores iniciales del vector serán los que se explicitan, luego se puede modificar cada uno sin ningún impedimento, por ejemplo por medio de una asignación directa.

```java
//inicialización estática
int[] elementos = {10, 5, 17, 0, -6, 24};

//modificamos el último elemento del vector
elementos[5] = 1;
```

Algo muy importante que tendremos que tener en cuenta es lo que sucede si intentamos acceder más allá de los límites definidos del vector. Indicar una posición por fuera de los límites desencadena un error en tiempo de ejecución. El intérprete, que en nuestro caso es la máquina virtual de java, lanza una excepción del tipo ArrayIndexOutOfBoundsException.

Si tuviéramos el arreglo previamente definido, y quisiéramos acceder a una posición más baja que el primer elemento o superior al último se lanzaría una excepción:

```java
//
System.out.println(elementos[-1]); ← error

//
System.out.println(elementos[6]);  ← error
```

[-] matrices

Las matrices representan otro caso particular de los arreglos, son aquellos que tienen dos dimensiones o bidimensionales. También se pueden pensar como un vector de vectores.

Una dimensión se utilizará para definir la cantidad de filas y la otra para especificar la cantidad de columnas. En definitiva una matriz tendrá una cantidad de elementos igual a las filas por columnas.

El siguiente gráfico muestra una matriz de 3 filas por 4 columnas. Se puede observar que la capacidad total de la colección es de 12 elementos dado que es la cantidad de espacios que me dá si multiplicamos las 3 filas por las 4 columnas.

		columnas			
		0	1	2	3
	0	10	5	17	0
filas	1	1	3	11	7
	2	-4	2	22	-12

Fig.19 - matriz.

Para definir este tipo de arreglos bidimensionales utilizaremos la siguiente forma sintáctica:

```
int[][] matriz = new int[filas][columnas];
```

Por un lado indicaremos que tendremos una matriz de enteros (`int[][] matriz`) y por otro que iniciamos la matriz con una cantidad de "**filas**" y "**columnas**" determinada (`new int[filas][columnas]`).

Es importante darnos cuenta que la cantidad se puede expresar mediante una expresión del tipo ordinal, no solamente a través de una constante o literal.

Como mencionamos con los vectores, los elementos de una matriz también se encuentran en la memoria de forma consecutiva, por lo que ahora tendremos que utilizar dos valores de índices para acceder a un elemento, uno representará la fila y el otro la columna.

El siguiente ejemplo declara una matriz de 3 filas por 4 columnas, de elementos enteros, y luego inicializa cada uno de ellos con un valor del tipo entero. Finalmente los imprimimos en la pantalla:

```
//
int[][] matriz = new int[3][4];

matriz[0][0] =  10;

matriz[0][1] =   5;
```

```
matriz[0][2] =  17;
matriz[0][3] =   0;

matriz[1][0] =   1;
matriz[1][1] =   3;
matriz[1][2] =  11;
matriz[1][3] =   7;

matriz[2][0] =  -4;
matriz[2][1] =   2;
matriz[2][2] =  22;
matriz[2][3] = -12;

System.out.println("Los valores de la matriz son:");

System.out.print(  matriz[0][0]+","+ matriz[0][1]+",");
System.out.println(matriz[0][2]+","+ matriz[0][3]);

System.out.print(  matriz[1][0]+","+ matriz[1][1]+",");
System.out.println(matriz[1][2]+","+ matriz[1][3]);

System.out.print(  matriz[2][0]+","+ matriz[2][1]+",");
System.out.println(matriz[2][2]+","+ matriz[2][3]);
```

[-] arreglos n-dimensionales

La cantidad de dimensiones que podrá tener un arreglo estará acotada por el lenguaje de programación que estemos utilizando, aunque lo cierto es que por lo general utilizaremos 2 o 3 dimensiones.

En Java, un arreglo puede tener hasta un máximo de 32,767 dimensiones. Sin embargo, es importante tener en cuenta que en la práctica, es poco común utilizar arreglos con más de tres o cuatro dimensiones debido a la complejidad y dificultad para gestionarlos.

A continuación dejamos un fragmento para que se vea de forma concreta cúal sería una

situación en la que podríamos necesitar modelar a través de un arreglo 3 dimensiones. Podemos pensar en la representación de una imagen del tipo RGB (Red - Green - Blue) de 100 píxeles de altura por 100 de ancho:

```
//
final int ALTURA  = 100;
final int ANCHO   = 100;
final int CANALES = 3;

//declaramos
int[][][] imagen = new int[ALTURA][ANCHO][CANALES];


// asignamos a los pixels el rojo
for(int i=0; i < ALTURA; i++) {
   for (int j = 0; j < ANCHO; j++) {

                // valores de píxeles para cada canal
                imagen[i][j][0] = 255; // Valor de rojo
                imagen[i][j][1] = 0;   // Valor de verde
                imagen[i][j][2] = 0;   // Valor de azul
            }
        }

//seguimos trabajando la imagen…
```

Ejercicios

[-] vectores

1. Definir un vector de 10 elementos. Asignar valores aleatorios a los elementos pares. Mostrar por pantalla. ¿Qué pasó con los valores no asignados?
2. Generar un vector con la siguiente indicación:
   a. V[j] = j para 1 <= j <= 10

   b. V[j] = ∑i(de i=1 a j) para 1 <= j <= 10
3. Escribir un programa que lea los datos de un vector de N componentes y visualice:
   a. La suma de las componentes del vector.

   b. El producto de las componentes del vector.
4. Escribir un programa que lea los datos de un vector de N componentes y visualice:
   a. El producto de los elementos menores que 10.

   b. La cantidad de elementos menores que 10.

   c. El promedio de los impares.
5. Escribir un programa que lea dos vectores de 30 componentes cada uno. Visualice los resultados de las opciones que siguen, pero sin generar un tercer arreglo:
   a. La suma.

   b. La diferencia.

   c. El producto escalar.
6. Se lee información que indica la cantidad de hectáreas sembradas cada día del mes de enero. Informe los días en que se sembró menos que el promedio del mes.
7. Leer un vector que representa la temperatura de cada día del mes de junio. Visualizar por pantalla la temperatura más baja, la más alta, y los días en que se produjeron respectivamente.

8. Dado un valor en Pesos:
    a. Escribir un programa que visualice por pantalla la cantidad mínima de billetes de 2000, 1000, 500, 100, 50, 20 y 10 que se necesitan para generar ese valor.

    Ej. $9670

    - 4 billetes de $2000
    - 1 billete de $1000
    - 1 billete de $500
    - 1 billete de $100
    - 1 billete de $50
    - 1 billete de $20

    b. Agregar la posibilidad de asignar distintas denominaciones de billetes. Suponga que se ingresan en orden decreciente (de mayor a menor).

9. Dado un vector de N componentes enteros, escriba un programa que visualice: La componente máxima y la posición de la misma. En caso de haber varios máximos mostrar la posición de cada uno de ellos.

10. Escribir un programa que lea los elementos de dos vectores de N y M componentes enteros. Cada uno representa los elementos de un conjunto. Suponga que los elementos que se ingresan no se encuentran repetidos. Hallar e visualizar por pantalla:
    a. La unión.
    b. La intersección.

[-] matrices

11. Dada una matriz cuadrada de N x N elementos enteros, con N dato y N <= 10, escribir un programa que calcule y visualice por pantalla:
    a. La cantidad de elementos nulos que existen en la diagonal principal
    b. La cantidad de elementos nulos que existen en la diagonal secundaria.
    c. La cantidad de elementos nulos que existen en el triángulo inferior.
    d. La cantidad de elementos nulos que existen en el triángulo superior.
    e. La matriz y su traspuesta
    f. Si es o no una matriz simétrica.

g. Si es o no la matriz identidad.

12. Realice un programa que permita calcular la suma, diferencia y producto de dos matrices, con valores numéricos.

13. Dada una matriz de M filas por N columnas, generar un vector columna que tenga la suma de cada fila.

14. Dada una matriz de M filas por N columnas, generar un vector columna que tenga el máximo de cada fila.

15. Una concesionaria de autos tiene 10 sucursales, numeradas de 1 a 10, y vende 6 tipos de vehículos, numerados de 1 a 6.
Al final de cada mes, desea sacar una estadística de ventas, y para ello procesa la información de todas las facturas de ese mes en la siguiente forma:

Nro de sucursal	Tipo de vehículo	Cant. unidades vendidas
4	2	5
1	1	1
3	5	2
4	3	1
7	4	1

Se desea:

a. Visualizar la información agrupada por número de sucursal, detallando por cada una la cantidad de unidades vendidas.

b. Cuál fue la sucursal que más vehículos vendió

c. Cuál fue el vehículo más vendido.

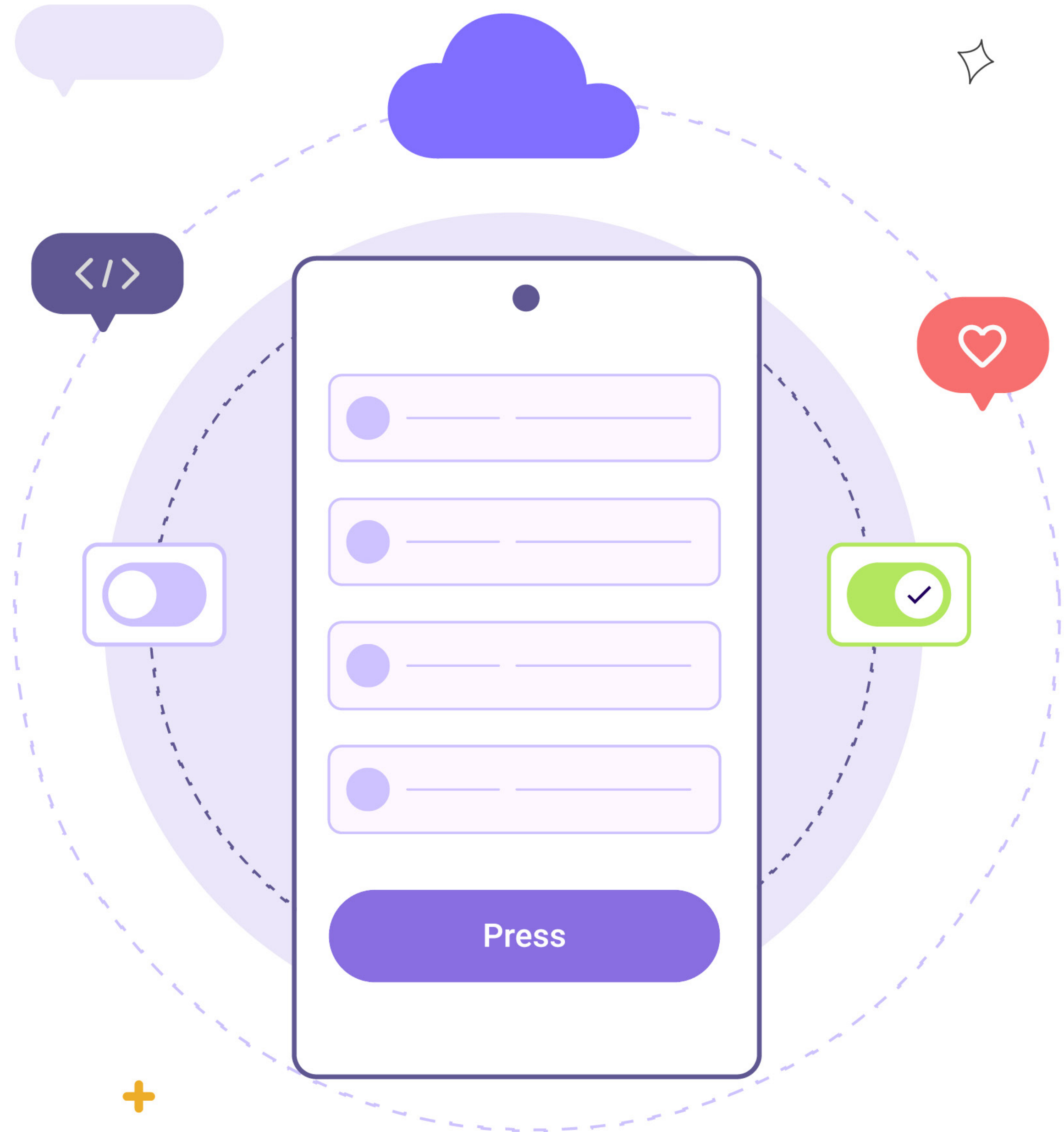
16. En un pueblo del interior, se realizaron las elecciones para intendente. Se presentaron cuatro candidatos A, B, C y D. Los datos se ingresarán por mesa, en el momento que los integrantes de esta última finalicen el escrutinio.

Mesa	A	B	C	D
5	10	100	31	0
7	20	200	50	3
1	17	50	20	2

17. El fin de carga, de los datos, lo da la mesa igual a 0 (cero)
18. Se pide calcular y visualizar por pantalla:
    a. ¿Cúal fue el candidato más votado?
    b. ¿Qué porcentaje de votos recibió?

Capítulo 6

Estructuras de programa



Capítulo 6

Estructuras de programa

Objetivo

El objetivo de este capítulo es enseñar a organizar el código de forma efectiva mediante el uso de funciones y librerías. Aprenderemos a descomponer problemas complejos en subproblemas más pequeños y manejables, y a implementar soluciones utilizando funciones que encapsulan la lógica de cada subproblema. Además, exploraremos el uso de librerías para aprovechar código predefinido y ampliar las capacidades de nuestros programas.

Al finalizar este capítulo, estarás capacitado para estructurar tus programas. Habrás adquirido una comprensión profunda de los beneficios de las diferentes estructuras de programa disponibles y cómo utilizarlas para escribir código organizado, más comprensible y reutilizable.

Subproblemas

A medida que los programas comiencen a ganar complejidad, necesitaremos mayor cantidad de líneas de código, mayor cantidad de algoritmos y seguramente vamos a requerir utilizar más de una vez estos algoritmos.

Los subproblemas son en definitiva unidades más pequeñas e independientes que forman parte de un problema mayor. Al dividir un problema en subproblemas, podemos abordarlo de forma más sencilla y organizada.

La descomposición del problema original en subproblemas nos traerá varios beneficios, entre ellos destacamos:

- Simplifican la complejidad: dividen un problema grande en partes más manejables, facilitando su comprensión y resolución.
- Reutilización de código: permiten usar las mismas soluciones para diferentes problemas, ahorrando tiempo y esfuerzo.
- Mantenimiento: facilitan la detección y corrección de errores, al aislar el problema en un subcomponente específico.
- Legibilidad: hacen que el código sea más claro y fácil de leer, al agrupar la lógica en unidades independientes.

Los diferentes paradigmas de lenguajes proporcionan mecanismos para permitir estas abstracciones y facilitarnos el crecimiento de la complejidad en la búsqueda de soluciones algorítmicas.

El paradigma “Divide y vencerás” será nuestro aliado.

Este libro se encuentra dirigido a la enseñanza de la programación bajo el paradigma orientado a objetos y si bien existe un mecanismo específico para lograr esta abstracción por el momento utilizaremos el término función para referirnos a esta característica algorítmica.

Función

Llamaremos función a la encapsulación de bloques de código, la cual irá acompañada de un nombre, ejecutará un conjunto de acciones y retornará un determinado valor.

La utilización de funciones, en un lenguaje de programación nos dará varias ventajas, en particular haremos hincapié en la “reutilización”.

La reutilización de código será una herramienta esencial que tendremos como programadores, dado que nos permite analizar, diseñar e implementar ciertos algoritmos una única vez y luego podemos utilizarla las veces que se requiera.

Las funciones se encuentran caracterizadas por los siguientes elementos:

- Identificador: tiene un nombre único que las identifica y las diferencia de otras en un determinado contexto.
- Cuerpo: es el bloque de código que define su tarea.
- Retorno: pueden devolver o no un valor como resultado de su tarea.
- Parámetros: es la forma en la que podemos comunicarnos con la función. Son valores que la tarea necesita para su realización.

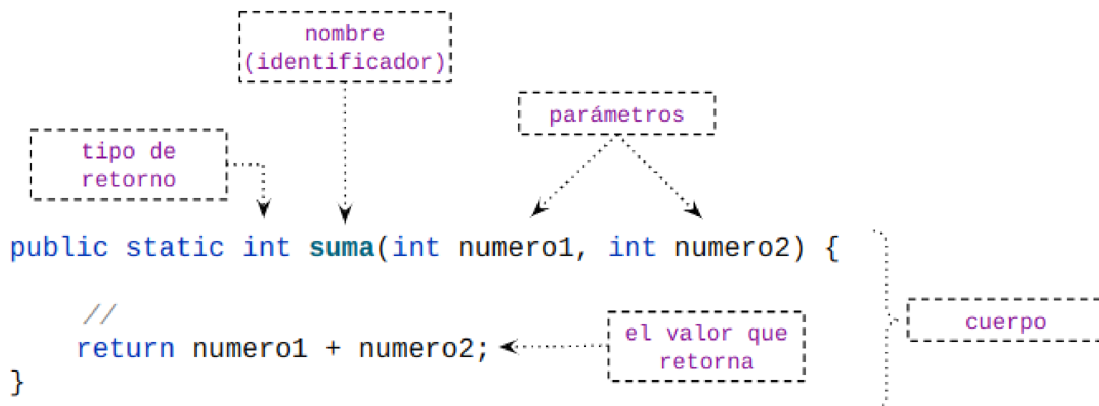


Fig.20 - partes de una función.

[-] identificador

Al igual que las variables, las funciones tendrán asociado un nombre único dentro del ámbito que se encuentren, concepto que profundizaremos más adelante.

Pueden existir funciones con el mismo nombre, siempre y cuando se encuentren en distintos ámbitos o sus parámetros sean diferentes en tipo o cantidad.

```
public static int suma(int a, int b){

    //
    return a + b;

}
```

[-] cuerpo

Es el bloque de código asociado a la función, entrará en ejecución cuando la función sea invocada.

El cuerpo de la función tendrá la dimensión que se necesite, dado que este dependerá de la tareas que se necesite realizar. Estas tareas pueden contar con variables locales, lo cual implica que su ámbito será dentro de la misma función.

Función que suma a y b:

```
public static int suma(int a, int b) {

    //
    int c = 10;

    //
    return a + b + c;

}
```

[-] retorno

El retorno de una función siempre debe especificarse.

Si la función no debe retornar nada, se utiliza el tipo especial “void”, que indica justamente esto. En el caso que se requiera retornar un valor, se debe especificar su tipo, por ejemplo si vamos a retornar un valor entero, especificamos int.

Función que retorna un valor entero:

```
public static int suma(int a, int b){

      //
      int c = 10;

      //
      return a + b + c;

}
```

Como mencionamos con anterioridad, una función siempre debe especificar un tipo de dato de retorno. Entonces, si quisiéramos establecer que una función no retorna ningún valor tendríamos que definir el tipo de dato de la función a “void”:

```java
public static void mostrarSuma(int a, int b){

      //
      System.out.println(a + b);
}
```

[-] parámetros

Los parámetros formales de una función son el medio que tendremos para comunicarnos al realizar una invocación desde algún punto del programa, inclusive desde otra función.

En java existen dos formas de utilizar los parámetros y estas dependen del modo de comunicación que se requiera. El modo de comunicación por valor y el modo de comunicación por referencia.

En java el uso de cada modo es automático, no podremos especificar el modo de comunicación como en otros lenguajes.

Los tipos de datos simples utilizan siempre el modo de comunicación por valor y los arreglos, y otros que veremos más adelante, utilizan el modo de comunicación por referencia.

El modo de comunicación por valor indica al intérprete que deberá realizar una copia del argumento que se utilice al momento de invocar la función. Esto significa que si modifico el valor del parámetro, en el cuerpo de la función no se modifica el valor del argumento con el que se ha invocado.

El modo de comunicación por referencia, en cambio, comparte el espacio de memoria que tiene el argumento y el parámetro, por lo tanto si el contenido se modifica el argumento también será modificado.

```java
//
public static int suma(int a, int b){

    //
    int c = 10;

    //
    return a + b + c;
}
```

La función “suma” podrá ser utilizada en cualquier lugar de nuestro programa, sólo tendremos que tener presente que la misma retorna un valor entero “int” y requiere dos valores enteros para su funcionamiento “a” y “b”.

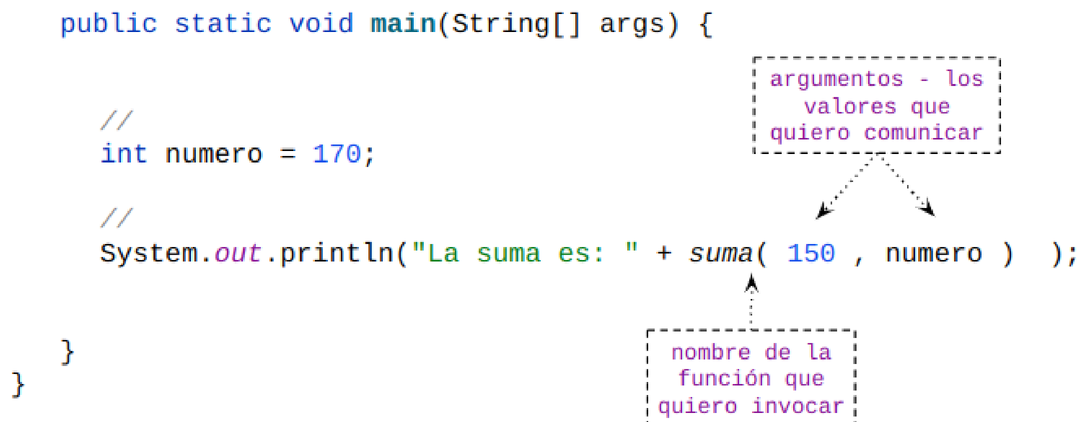


Fig.21 - invocación de una función.

A continuación mostramos algunos ejemplos en los cuales podremos ver como se llama a la función. Este proceso se conoce como “invocar a la función” y si la misma lo requiere habrá que suministrarle “argumentos” para que funcione.

En nuestro caso invocaremos a la función suma y usaremos los argumentos enteros 3 y 5 para los parámetro “a” y “b” respectivamente.

```
//
int resultado;

//
resultado = suma(3, 5);
```

Como esta función retorna siempre un valor entero “int”, podemos también utilizar la función usando el valor de retorno en alguna expresión. En el siguiente fragmento se vé como forma parte de la concatenación de una expresión de tipo cadena de texto:

```
//
System.out.println("la suma da: "+ suma(3,5));
```

Este último ejemplo utiliza un proceso de conversión implícito llamado "casting". Convierte el valor entero que retorna la función en un valor de tipo cadena de texto.

Tal como definimos previamente, existe el caso especial que la función no debe retornar ningún valor, para esto se utilizará la palabra reservada "void" en el tipo de dato de retorno.

```
//
public static void saludar(){

   System.out.println("Hola...");

}
```

Esta función no podrá ser parte de una expresión, dado que el tipo de dato de retorno es especial -void- y sólo se utiliza para que el intérprete reconozca que la función no va a retornar ningún dato.

Veamos un programa completo que utiliza el concepto anteriormente introducido para determinar la media aritmética, o sea, el promedio, a partir de un vector de 5 elementos.

```
public class Funcion {

   //
   public static float promedio(int[] vector){

      //
      float suma = 0;

      //
      for (int i=0; i< vector.length; i++){
         suma = suma + vector[i];
      }
```

```
        //
        return  suma / vector.length;
    }

    //
    public static void main(String[] args) {

        //
        int[] elementos = {1,5,3,7,8};

        //
        float prom = promedio(elementos);

        //
        System.out.println("El promedio es: "+ prom);
    }
}
```

[-] variable local

Cuando hablamos sobre el ámbito de una variable, nos referimos a la parte del programa donde una variable o función es accesible y puede ser utilizada.

El ámbito está determinado por la ubicación en el código donde se declara la variable o función. Si se declara una variable en el cuerpo de una función, su alcance será el de esa función y no podrá ser accedida desde otro lugar. Lo mismo sucede con el alcance de una función dentro del programa.

```java
public class Alcance {

    //
    final static int CONSTANTE = 10;

    //
    public static void alcance(){

        //
        int cantidad = 5;

        //
        for (int i=0; i< cantidad; i++){

            int parcial = i*2;

            System.out.println(parcial * CONSTANTE);
        }

    }

  …
}
```


Fig.22 - alcance.

Un breve ejemplo que muestra el alcance de la variables. La variable “cantidad” tendrá un alcance de todo el cuerpo de la función.

Las variables “parcial” e “i” tendrán un alcance al cuerpo de la estructura de control “for”:

```java
//
public static void alcance(){

    //
    int cantidad = 5;

    //
    for (int i=0; i< cantidad; i++){

        //
        int parcial = i*2*cantidad;
```

```
        //
        System.out.println(parcial);
    }
}
```

Librería

Una librería es un conjunto de funciones que están disponibles para ser utilizadas por un programa. Estas funciones se han prescrito para utilizarlas sin tener que escribirlas nuevamente.

Las librerías, también conocidas como bibliotecas, son esenciales para el desarrollo de software, ya que permiten a los programadores reutilizar código existente además de ahorrar tiempo y esfuerzo al no tener que escribir todo desde cero.

En Java, las librerías pueden ser clases individuales o paquetes de clases. Se utiliza la palabra clave "import" seguida de la ruta del paquete o clase que se quiera importar.

El siguiente ejemplo muestra cómo importar un paquete completo. Todas las clases que se encuentren en el paquete se importaran y podrán utilizarse. Se utiliza el símbolo * (asterisco) para indicar esto.

Ojo con el uso del ***** porque probablemente importe más de lo necesario. Importamos todas las clases "utilitarias" del lenguaje:

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

Cabe aclarar que los subpaquetes que podrían existir en el mencionado paquete "util" no se importarán, habrá que hacer explícita esta necesidad.

Una vez declarada esta importación podríamos utilizar todas las clases, inclusive "java.util.Scanner", dado que se encuentra en el paquete importado.

Este mecanismo también nos permitirá definir nuestras propias librerías y utilizarlas cuando haga falta.

El siguiente ejemplo muestra cómo importar las clases que necesitamos para realizar la lectura de un número y obtener otro de forma aleatoria. Las clases que se requieren im-

portar son Random y Scanner.

Otra forma de incorporar las clases es mediante la importación de todas las que se encuentren en el paquete "**import java.util.***". Nuevamente, tengamos presente que esta forma incorpora al código final todas las clases del paquete por más que no las utilicemos.

Ejemplo:

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

public class Libreria {

   //
   public static void main(String[] args) {

       Scanner lector = new Scanner(System.in);
       Random aleatorio = new Random();

       int num1 = lector.nextInt();
       int num2 = aleatorio.nextInt(10);

       System.out.println("adivinamos ? " +
                                   (num1 == num2));
   }
}
```

Ejercicios

1. Escribir una función que retorne la suma de 2 números.

   Ej:

   ```
   a = 123
   b = 210
   resultado = suma(a,b)
   ```

   El resultado será 333.

2. Escribir una función que calcule el factorial de un número N.

   Ej:

   ```
   n = 5
   resultado = factorial(5)
   ```

   El resultado será 120 dado que 1*2*3*4*5 = 120.

3. Escribir una función que calcule el resto de la división.

   Ej:

   ```
   resto = mod(10,3);
   ```

   El resto será 1 dado que 10 divido 3 da como resto 1.

4. Escribir una función que determine si un número es capicúa.
5. Escribir una función que determine la potencia a partir de una base y su exponente.

   Ej:

   ```
   base = 2

   exponente = 3
   ```

```
resultado = potencia(2,3)
```

El resultado será 8 dado que 2*2*2 = 8.

6. Escribir una función que retorna la cantidad de dígito que compone un número.

   Ej:

   ```
   número = 223344
   resultado = digitos(numero)
   ```

   El resultado será 6 dado que el número se encuentra formado por 6 dígitos.

7. Escribir una función que retorne el día de la semana a partir del número de día.

   Ej:

   ```
   n = 2
   dia = diaDeSemana(n)
   ```

   El día será "martes" dado que el 1 corresponde al lunes, el 2 al martes y así sucesivamente hasta el 7 que corresponde al domingo.

8. Escribir un programa que pida al usuario en el formato DDMM y escriba en la pantalla si la fecha es correcta o no. Analice:

   a. Que el mes es correcto (Es decir, entre 1 y 12).

   b. Que el día es correcto, según el mes

9. Extienda el ejercicio anterior agregando la verificación del año. Ahora el formato será DDMMYYYY.

10. Escribir una función que asigne a un arreglo de N componentes valores aleatorios.

    Ej:

    ```
    int[] arreglo = new int[3];
    asignarValores(arreglo);
    ```

    Al retornar el arreglo tendrá sus elementos con valores aleatorios.

11. Escribir una función que pida al usuario valores para un arreglo de N componentes.

    Ej:

    ```
    int[] arreglo = new int[3];
    leerValores(arreglo);
    ```

    Al retornar el arreglo tendrá sus elementos con los valores del usuario.

12. Escribir una función que a partir de un arreglo retorne la posición del mayor elemento.

    Ej:

    ```
    int[] arreglo = {3, 0, 2, 5, 1, 2};
    int pos = mayor(arreglo);
    ```

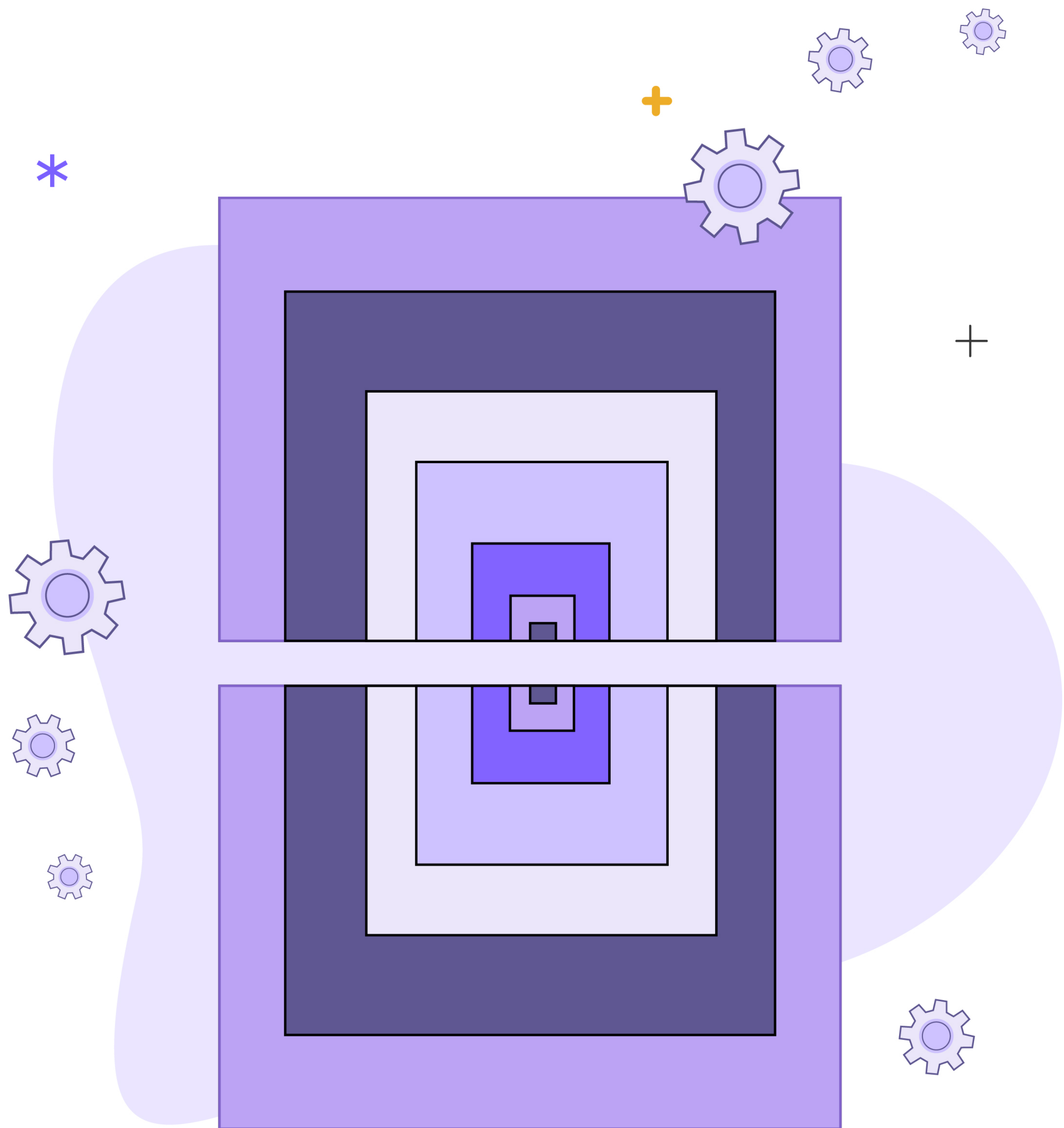
    El resultado de pos será 3 dado que el mayor elemento se encuentra en la posición 3;

13. Escribir un programa que asigne valores a un arreglo de N componentes y luego determine cuál es el mayor de ellos, imprimiendo por pantalla la posición y el valor.

    Nota: puede utilizar valores aleatorios o solicitar el ingreso al usuario.

Capítulo 7

Recursión



Capítulo 7

Recursión

Objetivo

El capítulo tiene como objetivo enseñar la recursión, una técnica que nos permite resolver problemas de forma elegante y sencilla ante determinados escenarios. Aprenderemos a formular soluciones recursivas y comprender los beneficios y las limitaciones de este enfoque.

Al finalizar este capítulo, estarás capacitado para utilizar la recursividad de forma efectiva en tus programas. Habrás adquirido una comprensión profunda de los beneficios y las limi-taciones de la recursividad, así como la capacidad de identificar problemas que se pueden resolver de forma recursiva y diseñar algoritmos para su solución.

Recursión

La recursión es una técnica que se utiliza para resolver problemas del tipo evolutivo, digamos que la búsqueda de la solución es posible a través de la división del problema original en problemas cada vez más pequeños y similares al del comienzo.

Esta idea de resolución, pensada a través de la utilización de algoritmos, se materializa por medio de subproblemas, o más concretamente, en funciones que se llaman a sí misma.

Para lograr que la función resuelva el problema y además se llame a sí misma y no caiga-mos en un bucle que no podrá detenerse, se debe llamar a sí misma con los argumentos, aunque sea ligeramente, diferentes.

Veamos un ejemplo práctico de utilización. El factorial de un número natural n, denotado como n!, se define como el producto de todos los números naturales positivos menores o iguales a n.

Formalmente:

```
0! = 1 - por convención

1! = 1

n! = 1 * 2 * 3 * ... * (n - 1) * n
```

Por ejemplo el cálculo de 5! es igual a 1 * 2 * 3 * 4 * 5, lo que da como resultado 120.

De lo anterior podemos ver que:

5! = 5 * 4!

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1 ⇒

Esta situación nos lleva a observar que estamos frente a la definición de un problema recursivo, aquel problema original que puede resolverse con la división en subproblemas similares.

Entonces, siguiendo la definición de cómo se debe calcular el factorial podemos realizar el algoritmo.

Tenemos que considerar 2 escenarios posibles:

- si el valor del parámetro n es igual a 0 o es igual a 1, el resultado del factorial es 1.
- si el valor del parámetro es otro, dentro de los valores posibles, el resultado es n por el factorial de n-1.

La recursión es un proceso, o conjunto de reglas, definido en términos de sí mismo.

A continuación vemos esta especificación en forma de código:

```
//
public static int factorial(int n) {

   //
   if (n == 0 || n == 1) {
       //
       return 1;
```

```
    } else {

        //
        return n * factorial(n-1);
    }
}
```

Como vemos, hicimos la traducción de la especificación a código, manteniendo las reglas, si el número “n” es igual a cero o uno, la función retorna 1, de lo contrario es n por el resultado de llamarse a sí mismo con el argumento “n-1”. El proceso continúa hasta que “n” llega a uno, momento en el cual la función retorna 1, y se completa la recursión.

Completarse la recursión significa que se comienza con el proceso inverso a las invocaciones, retornando los valores de cada una.

Como podemos ver, la recursión consta de dos partes:

- Caso base: es el caso más simple y directo que se puede resolver sin necesidad de una llamada recursiva, o sea, una llamada a si mismo. Es esencial para evitar que la función recursiva se llame infinitamente y cause un bucle infinito. En la recursión, el caso base proporciona el punto de terminación para las llamadas recursivas.
- Caso recursivo: es la parte de la función que se llama a sí misma, resolviendo una instancia más pequeña del mismo problema. En cada llamada recursiva, el problema se divide en subproblemas más pequeños y se resuelve recursivamente hasta que se alcanza el caso base.

La recursión es una buena herramienta y elegante para resolver problemas, pero es importante destacar que no es eficiente en términos de utilización de tiempo de cpu y cantidad de memoria.

Volviendo al ejemplo del cálculo del factorial, pero ahora gráficamente, podemos apreciar visualmente cómo se producen las llamadas cuando se invoca a la función para determinar el factorial de 4. También se puede observar cómo es el retorno de las llamadas una vez que se llega al caso base y se terminan las llamadas recursivas, o sea, los casos donde se llama a sí misma.

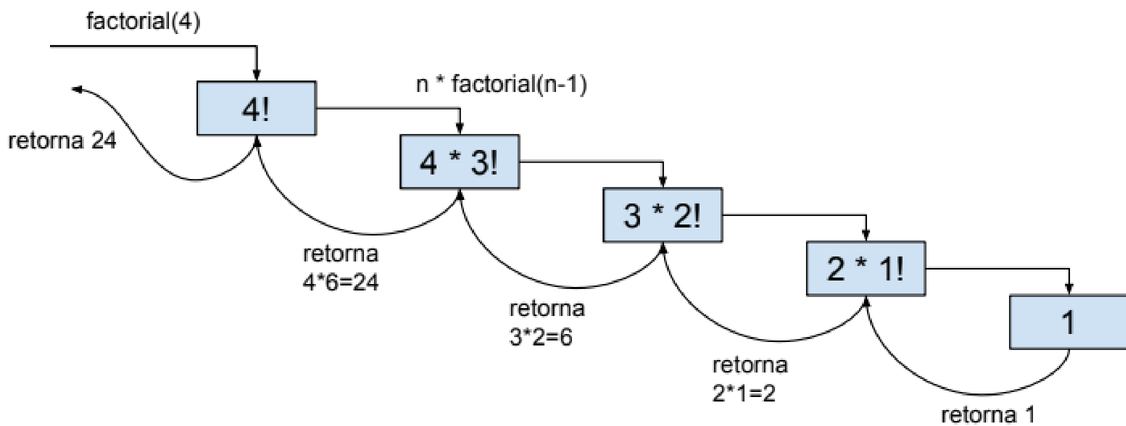


Fig.23 - factorial recursivo.

Llamar a la función factorial(4) produce el retorno del valor 24

Como mencionamos al principio del capítulo, la recursión puede ser una herramienta super interesante para resolver problemas con características evolutivas o recursivas, dada su elegancia para escribir el algoritmo.

Entonces, ¿por qué es preferible utilizar algoritmos iterativos?, es preferible porque los algoritmos recursivos usan mayor tiempo de CPU y mayor cantidad de memoria.

Utilizan mayor tiempo de CPU porque cada llamada recursiva a la función crea un nuevo contexto en la pila de llamadas y lógicamente hacer esto requiere tiempo de uso sobre la CPU debido a que tiene que reservar el espacio en la memoria va a utilizar, apilar el contex-to, ajustar el puntero en la pila y recién ahí comenzar con la ejecución de la función.

La pila de llamadas es un área de memoria que se utiliza para almacenar información sobre las funciones que se están ejecutando actualmente.

Cada contexto en la pila de llamadas contiene información sobre la función que se está ejecutando, como los valores de las variables locales y la dirección de retorno a la función que la invocó. Cuando una función recursiva se llama a sí misma, también se crea un nuevo contexto en la pila de llamadas.

Si la recursividad es profunda, es decir, si la función se llama a sí misma muchas veces, la pila de llamadas puede crecer mucho y en consecuencia consumir una cantidad significativa de tiempo de uso sobre la CPU.

Lo mencionado se puede apreciar en la siguiente figura, el uso de memoria será mayor.

Está situación si no es controlada puede consumir una cantidad significativa de memoria e inclusive puede provocar un agotamiento de la memoria, en particular la sección denominada pila (stack).

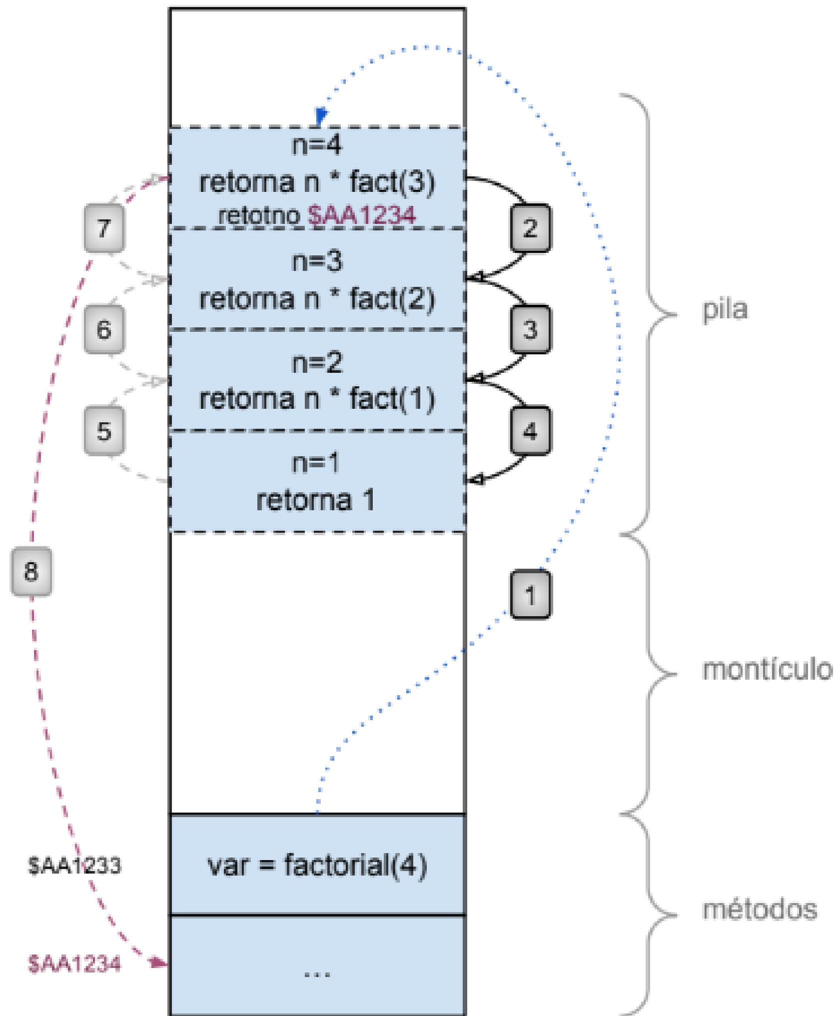


Fig.24 - pila de llamadas.

Si bien existen técnicas que ayudan a combatir estas situaciones, como la técnica de recursión de cola o la memoización, es importante saber que si existe un algoritmo recursivo, entonces existe su análogo iterativo.

Si bien podemos crear algoritmos recursivos que se adaptan a ciertos escenarios, que son más legibles y rápidos de escribir, si contamos con el tiempo suficiente, deberíamos crear algoritmos iterativos.

Se muestra nuevamente la función factorial pero ahora de forma iterativa:

```
//
public static int factorial(int n) {

   //
   int resultado = 1;

   //
   for (int i = 1; i <= n; i++) {
       resultado = resultado * i;
   }

   //
   return resultado;
}
```

Este algoritmo iterativo tiene un consumo de memoria constante, y el tiempo de uso de la cpu es considerablemente menor que su análogo recursivo.

Veamos otro ejemplo. Realizar la suma de los componentes de un vector.

A diferencia de lo que hicimos con el factorial, en esta ocasión vamos a partir del algoritmo iterativo, que sumará los elementos del vector, para luego convertir la función en recursiva.

Una estrategia para transformar un algoritmo iterativo en uno recursivo es reemplazar la estructura de control de repetición principal por llamadas a sí misma de la función recursiva.

Suma de los elementos del vector de forma iterativa:

```java
public class SumaArregloIterativo {

   public static int sumaArreglo(int[] arreglo) {
       //
       int suma=0;

       //
       for(int indice=0; indice<arreglo.length; indice++) {
           suma = suma + arreglo[indice];
       }

       return suma;
   }

   public static void main(String[] args) {

       //
       int[] arreglo = {1, 2, 3, 4, 5};

       //
       int suma = sumaArreglo(arreglo);

       //
       System.out.println("La suma es:  " + suma);
   }
}
```

El código tiene un arreglo de 5 elementos, los cuales inicializa de forma estática y luego invoca a la función “sumaArreglo” que retorna la suma de todos los valores del arreglo.

Dado que se conoce la cantidad de elementos, la función utiliza una estructura de control de repetición del tipo for para iterar sobre la estructura e ir sumando los valores del vector en la variable “suma”.

Veamos como sería el mismo programa pero con la función implementada con la estrategia recursiva. Notemos que el principal cambio es que ya no tiene la estructura de repetición, la cual ha sido reemplazada por las llamadas a sí mismo.

Suma de los elementos del vector de forma recursiva:

```
public class SumaArregloRecursivo {

  public static int sumaArreglo(int[] arreglo, int indice) {
       //
       if (indice == 0) {

           return arreglo[0];

       } else {

           return arreglo[indice]
                  + sumaArreglo(arreglo, indice - 1);
       }
   }

   public static void main(String[] args) {
       //
       int[] arreglo = {1, 2, 3, 4, 5};

       //
       int suma = sumaArreglo(arreglo, arreglo.length - 1);

       //
       System.out.println("La suma es: " + suma);
   }
}
```

La función utiliza la recursión para sumar los elementos. Establece el caso base en función del valor del índice, por lo tanto se invoca por primera vez con el valor de la cantidad de

elementos - 1. Cuando la función determina que el valor del índice es 0 termina la recursión, se retorna el valor del primer elemento del vector. Esto desencadena que comiencen a retornar las sumas parciales, o sea, la suma de los casos recurrentes.

Se recomienda utilizar la recursividad sólo cuando se adapte naturalmente a la estructura del problema, a la estructura de los datos o cuando la claridad y la legibilidad del código sean prioridades. Si la eficiencia es un factores crítico, entonces es necesario considerar la conversión del algoritmo a su equivalente iterativo.

Ejercicios

1. Escribir una función recursiva que calcule el producto de dos enteros a partir de la suma de sus argumentos.

2. Escribir una función recursiva que calle el enésimo término de la sucesión de Fibonacci.

3. Realizar la traza del siguiente programa e indicar que realiza:

```
public static int XX(int n, int m) {
   if (m == 0 || m == n) {
       return 1;
   } else {
       return XX(n - 1, m - 1) + XX(n - 1, m);
   }
}

public static void main(String[] args) {
   System.out.println(XX(3, 1));
}
```

4. Realizar la traza del siguiente programa, indicando que realiza y si además presenta algún inconveniente:

```
public static int ZZ(int N, int M) {
   if (N == M) {
       return 1;
   } else {
       return N * ZZ(N + 1, M);
   }
}

public static void main(String[] args) {
   System.out.println(ZZ(3, 6));
}
```

5. Escribir una función recursiva y una iterativa para calcular el N-ésimo coeficiente del polinomio de HERMITE que resulta de la siguiente definición:

   H0(X) = 1

   H1(X) = 2*X

   Hi(X) = 2*X*Hi-1(X) - 2*(i-1)*Hi-2(X)

6. 6.La función de Akerman se define en forma recursiva para enteros NO negativos:

   ```
   A(M,N) = N+1 si M=0
   A(M,N) = A(M-1 , 1) si M<>0 y N=0
   A(M,N) = A(M-1 , A(M,N-1)) si M<>0 y N<>0
   ```

   a. Demuestre que A(2,2) = 7

   b. Implemente la función.

7. Realizar una función recursiva que a partir de una cadena de texto imprima todos los caracteres leídos en orden inverso.
8. Realizar una función recursiva que lea una serie de caracteres hasta encontrar un caracter igual a “F” y luego imprima todos los caracteres leídos en orden inverso.
9. Realizar un programa que le pida al usuario un valor entero positivo e imprima sus dígitos en orden inverso, usando una función recursiva.

   Ejemplo: 341 imprimir 1 4 3

10. Realice la función de Euclides de forma recursiva.

    El algoritmo de Euclides es un método para calcular el máximo común divisor (MCD) de dos números enteros. El MCD de dos números enteros a y b es igual al MCD de a y b - a, siempre que b > a.

    Algoritmo:

    - Si a y b son iguales, el MCD es a.
    - Si a es mayor que b, se intercambia a y b.
    - Luego, se resta b de a tantas veces como sea posible.
    - El último resto no nulo es el MCD de a y b.

Ejemplo de MCD con a = 12 y b = 18

- a es menor que b, no se intercambian.
- 18 - 12 = 6.
- 12 - 6 = 6.
- 6 - 6 = 0.
- El último resto no nulo es 6.

Capítulo 8

Algoritmos fundamentales

Capítulo 8

Algoritmos fundamentales

Objetivo

Este capítulo tiene como objetivo brindar una comprensión profunda de los algoritmos fundamentales que son la base de la mayoría de las soluciones computacionales. Se explorará en detalle el funcionamiento de estos algoritmos, sus ventajas y desventajas, y su aplicación en diferentes escenarios. Se analizarán los diferentes tipos de algoritmos básicos de búsqueda y ordenamiento. A través de la explicación de diferentes situaciones se brindará una introducción a la complejidad algorítmica subyacente en cada uno.

Al finalizar este capítulo, estarás capacitado para utilizar los algoritmos fundamentales de forma efectiva en tus programas. Habrás adquirido una comprensión profunda de los dife-rentes tipos de algoritmos, su funcionamiento y una introducción práctica a la complejidad computacional. Además, estarás en condiciones de elegir el algoritmo adecuado para cada caso y diseñar soluciones eficientes a problemas diversos.

Algoritmos

En nuestra disciplina, específicamente en el área de programación, llamamos algoritmos "fundamentales" a aquellos algoritmos que son esenciales y se utilizan con frecuencia para resolver una gran variedad de problemas.

Estos algoritmos también son importantes porque su eficiencia y complejidad son un factor crítico en el rendimiento de los programas y su capacidad para manejar grandes cantidades de datos.

En este contexto diremos que la eficiencia se refiere a la capacidad de un algoritmo para realizar una tarea utilizando la menor cantidad de recursos posible, como el tiempo de ejecución y la cantidad de memoria. Un algoritmo eficiente es aquel que puede completar una tarea de manera rápida y utilizando además la mínima cantidad de recursos.

Cuando mencionemos la complejidad algorítmica, por ahora, diremos que se refiere a la cantidad de recursos que un algoritmo necesita para completar una tarea en función del tamaño de la entrada.

La complejidad se utiliza para medir la eficiencia de un algoritmo y predecir su comportamiento a medida que aumenta el tamaño de la entrada.

Entonces podemos advertir que la eficiencia de un algoritmo está directamente relacionada con su complejidad. Un algoritmo con baja complejidad generalmente será más eficiente que un algoritmo con alta complejidad.

La complejidad temporal mide la cantidad de tiempo que un algoritmo necesita para completar una tarea y se expresa en términos de unidades de tiempo, como por ejemplo segundos o inclusive en términos de operaciones.

La complejidad espacial mide la cantidad de memoria que un algoritmo necesita para completar una tarea y se expresa en términos de unidades de memoria, como bytes o kilobytes.

De acá hasta el final cuando hablemos de complejidad lo haremos refiriéndonos a la temporal, cuánto tiempo le cuesta, en función de la entrada, al algoritmo terminar.

En definitiva, parte de los algoritmos fundamentales que veremos son los que se relacionan con la búsqueda y el ordenamiento. Conocer estos algoritmos y entender cómo funcionan es fundamental para cualquier programador que desee escribir programas eficaces y sobre todo programas eficientes.

> La eficiencia es la capacidad de un algoritmo para utilizar de la mejor manera los recursos, como el tiempo de ejecución y la cantidad de memoria, para lograr el resultado. Teniendo en cuenta además que el algoritmo no pierda legibilidad y fundamentalmente pueda escalar.

Búsqueda

Los algoritmos de búsqueda son aquellos que permiten encontrar un elemento particular en una colección de elementos. Si bien existen varios tipos de búsqueda, dependiendo el problema y datos involucrados, nos centraremos en las más elementales: la búsqueda lineal y la búsqueda binaria.

[-] lineal

Para comprender cómo funciona la búsqueda lineal podemos imaginar que estamos buscando un libro en una estantería llena de libros.

Iniciamos desde el principio de la estantería, verificamos el primer libro leyendo el título y lo comparamos con el que buscamos. Si es el libro que buscábamos entonces finalizamos

la búsqueda, ahora si no lo es continuamos con el siguiente y volvemos a repetir el procedimiento anterior. La búsqueda la continuamos hasta el final o hasta que encontremos en el camino el libro.

Entonces, dado que ya entendemos la lógica de funcionamiento podemos decir que la búsqueda lineal comienza en el primer elemento de la colección y lo compara con el elemento que buscamos. Si no son iguales, se pasa al siguiente elemento y se repite la comparación. Este proceso continúa hasta que se encuentra el elemento o hasta que llegamos al final de la lista.

Podemos ver esto de forma gráfica. Buscamos en un vector de 6 elementos aquel cuyo valor sea 0.

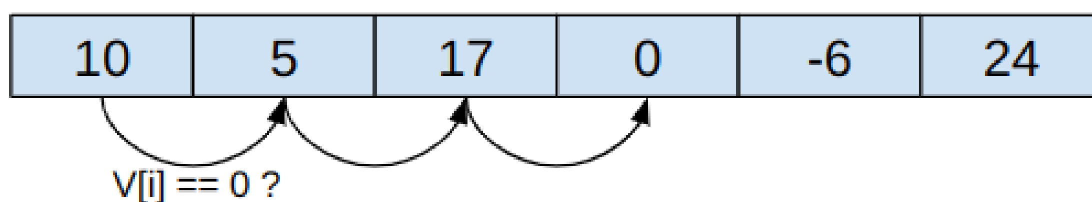


Fig.25 - búsqueda lineal o secuencial.

La búsqueda lineal es útil cuando la lista de elementos es relativamente pequeña, dado que si es grande podemos tardar mucho tiempo en determinar su existencia, o peor aún, su inexistencia.

Veamos como podría ser la implementación de una función que busca linealmente y retorna la posición del elemento si lo encuentra.

```
//
public static int busqueda(int[] vector, int e) {

   //
   for (int i=0; i < vector.length; i++){

       if(vector[i] == e){
           return i;
       }
   }
```

```
//
    return -1;
}
```

Como podemos apreciar en el código, si la función retorna -1 entonces significa que se llegó al final del vector y el elemento buscado no se encontró caso contrario retorna la posición del elemento en el vector.

En el ejemplo anterior, debido a la disposición de los elementos del vector, en el caso que no se encuentre el elemento buscado no hay otra alternativa que recorrer todos los elementos y sólo al final podemos asegurar que no existe.

¿Qué pasaría si contáramos con los elementos ordenados?, por ejemplo de forma ascendente -de más chico a más grande-, ¿podríamos mejorar el algoritmo para que aprovechara esta situación?.

La respuesta es si, la mejora consiste en determinar si vale la pena seguir buscando, o sea, si los elementos están ordenados de forma ascendente y en la búsqueda encuentro un elemento que no es el que buscado pero se dá que justamente el elemento del vector que estoy examinando es más grande, entonces puedo asegurar que no voy a encontrar el elemento y retorno la posición que indica su inexistencia, -1.

En el gráfico mostramos cuando se detendría la búsqueda si el elemento a buscar fuera el 8. Cuando examinemos el 4to elemento podríamos determinar que no existe en el vector el que buscamos, por lo tanto podemos retornar -1.

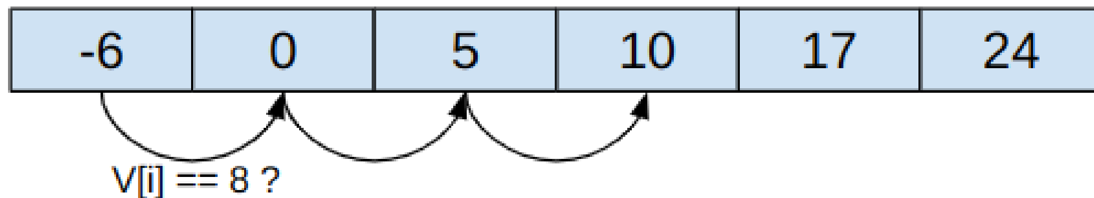


Fig.26 - búsqueda lineal o secuencial ordenada.

```
//
public static int buscar(int[] vector, int e) {

   //
   for (int i=0; i < vector.length; i++){
       if(vector[i] == e){
           return i;
       } else if (vector[i] > e) {
           return -1;
       }
   }

   //
   return -1;
}
```

Antes de pasar a otro tipo de búsqueda pensemos un segundo en la complejidad asociada para definir la eficiencia del método de búsqueda lineal o secuencial.

En el peor de los casos, el algoritmo tendrá que recorrer todo el vector para encontrar el elemento, lo que significa que el tiempo de ejecución será proporcional al tamaño del vector.

Usamos el peor de los casos porque nos interesa observar el comportamiento justamente en una situación límite y no satisfactoria, o sea este caso sería no encontrar el elemento.

En notación matemática, la complejidad temporal de la búsqueda lineal se expresa como O(n), donde n es el tamaño de la lista. A esta notación se la conoce como "o grande" (big O). No profundizaremos en este libro el tema, pero sería interesante que puedan hacerlo y de esta manera enriquecer su conocimiento sobre la disciplina.

Finalmente podríamos concluir que el método de la búsqueda lineal o secuencial será eficiente sólo con colecciones pequeñas de elementos, no más de 100 o 1000 elementos, dependiendo el contexto.

[-] binaria

Buscar un elemento en una colección será una tarea habitual, es una actividad que se realizará con mucha frecuencia. Si la colección es grande o peor aún, si la colección continúa creciendo, el tiempo para determinar que un elemento existe o no será cada vez mayor, entonces naturalmente surge la pregunta ¿cómo podríamos hacer para buscar más rápido? y además, ¿cómo podríamos hacer para que este tiempo sea constante o al menos no crezca con la misma velocidad con la que crece el tamaño?

Este tipo de preguntas aparecerán con cierta frecuencia ante el comportamiento de determinados algoritmos, y una estrategia que podemos tomar para expandir las posibilidades es atacar el problema antes que surja la necesidad de resolverlo.

Dicho de otra manera y de forma más concreta, si los elementos se encuentran en la colección y debo buscar ya poco podré hacer, ahora si me enfoco en la distribución de los elementos, por ejemplo que tengan cierto orden, entonces podrían plantear otras posibilidades al momentos de buscar.

La búsqueda binaria es un algoritmo de búsqueda que se utiliza para encontrar un elemento específico dentro de una colección ordenada. Este algoritmo es mucho más eficiente que la búsqueda secuencial, especialmente cuando la colección es grande.

El algoritmo de búsqueda binaria se basa en dos ideas específicas, por un lado los elementos deben estar ordenados y por el otro usar la estrategia de divide y vencerás.

El funcionamiento es básicamente:

- dividir la colección en dos mitades.
- comparar el elemento que se encuentra en la mitad de la colección con el elemento que se busca
- comprobar si es el elemento, en el caso que fuera retornamos la posición, ahora si no lo es debemos determinar qué mitad descartamos. Si el elemento que se busca es menor entonces descartamos la mitad superior o si es mayor, descartamos la mitad inferior.
- repetir los pasos anteriores hasta encontrar el elemento o hasta que no haya más elementos en la colección para seguir buscando.

A continuación se muestra gráficamente cómo funciona el algoritmo para buscar en un vector de 6 elementos aquel que tenga el valor igual a 10. Un dato importante, que puede apreciarse, es que los elementos de la colección se encuentran ordenados de menor a mayor.

El orden que tienen los elementos determinará cuál es la mitad que debo descartar en caso de seguir buscando.

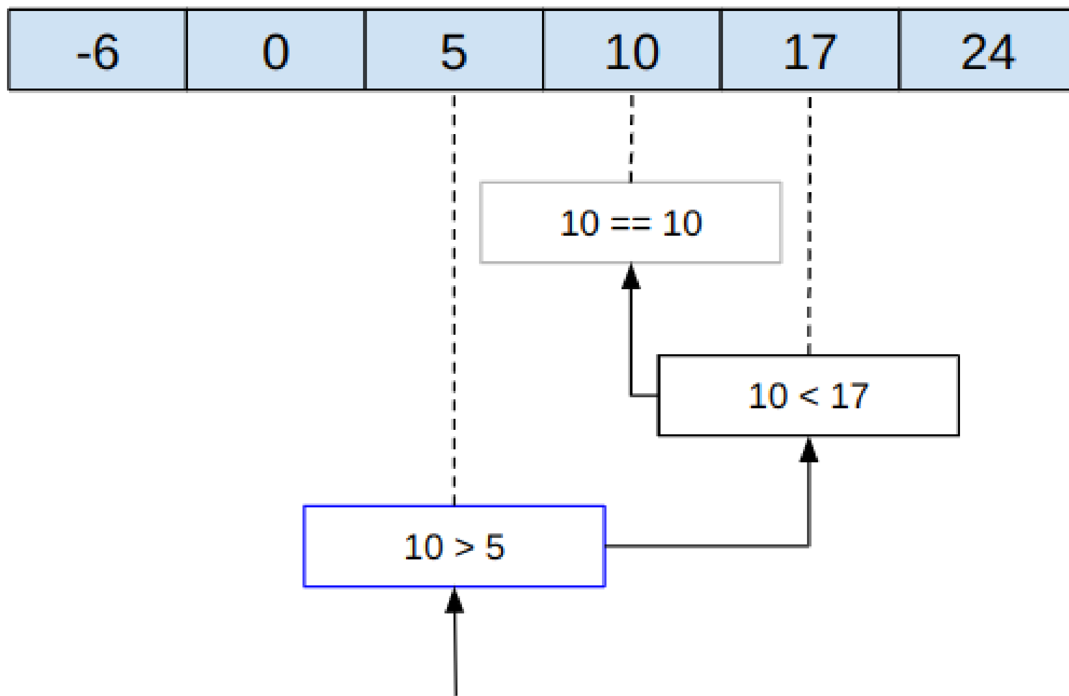


Fig.27 - búsqueda binaria.

Podemos apreciar gráficamente, en este caso, que para buscar el valor 10 dentro de la colección ordenada de 6 elementos se requieren realizar 3 verificaciones o consultas.

La cantidad de consultas para determinar que un elemento no se encuentra en la colección será de 4 y lógicamente esta situación es mejor que las 6 consultas que necesitaríamos para determinar esto mismo con la búsqueda secuencial.

De forma matemática, a medida que la cantidad de elementos crece la cantidad de consultas necesarias para determinar la inexistencia de un elemento crece pero lo hace mucho más despacio.

La búsqueda binaria tiene formalmente un orden O(log2 n) porque el número de consultas que se necesitan para encontrar un elemento en una lista ordenada es proporcional al logaritmo del tamaño de la colección.

En cada iteración de la búsqueda, se divide la colección por la mitad y se descarta la mitad que no puede contener el elemento que se busca. Esto significa que el número de elementos que se comparan en cada iteración se reduce a la mitad.

El logaritmo de un número es la potencia a la que hay que elevar a 2 para obtener ese número. Por ejemplo, el logaritmo en base 2 de 8 es 3 porque 2 elevado a la 3 es 8.

En la búsqueda binaria, el número de consultas que se necesitan para encontrar un elemento en una lista de tamaño n es igual al logaritmo en base 2 de n.

Por ejemplo, si quisiera buscar un elemento en una colección de 32 elementos, necesitaría como máximo hacer 5 consultas para encontrar el elemento o determinar su inexistencia.

Recordemos que nos enfocamos en la inexistencia de un elemento para comprender su comportamiento ,debido a que nos interesa cómo el algoritmo funciona en su peor caso.

```
//
public static int binaria(int[] vector, int e) {

   int inicio = 0;
   int fin = vector.length - 1;

   //
   while (inicio <= fin) {
       int medio = inicio + (fin - inicio) / 2;

       if (vector[medio] == e)
           return medio;

       if (vector[medio] < e)
           inicio = medio + 1;
       else
           fin = medio - 1;
   }

   //
   return -1;
}
```

Si bien este algoritmo es mucho más eficiente que la búsqueda secuencial se requiere que los elementos se encuentren ordenados y además sí o sí conocer a priori el criterio con el que se han ordenado los elementos, si ha sido de forma ascendente o descendente.

También es posible que alguien en este momento se pregunte, el tiempo para buscar un

elemento es bajo, pero ¿qué hay del tiempo que se requiere para mantener el orden en la colección?

Por el momento contestaremos que sí con poco esfuerzo puedo mantener la colección ordenada a lo largo de la vida del sistema, entonces la primera vez quizá no sea buena opción pero todas las posteriores sin duda será la mejor opción ante la búsqueda lineal o secuencial.

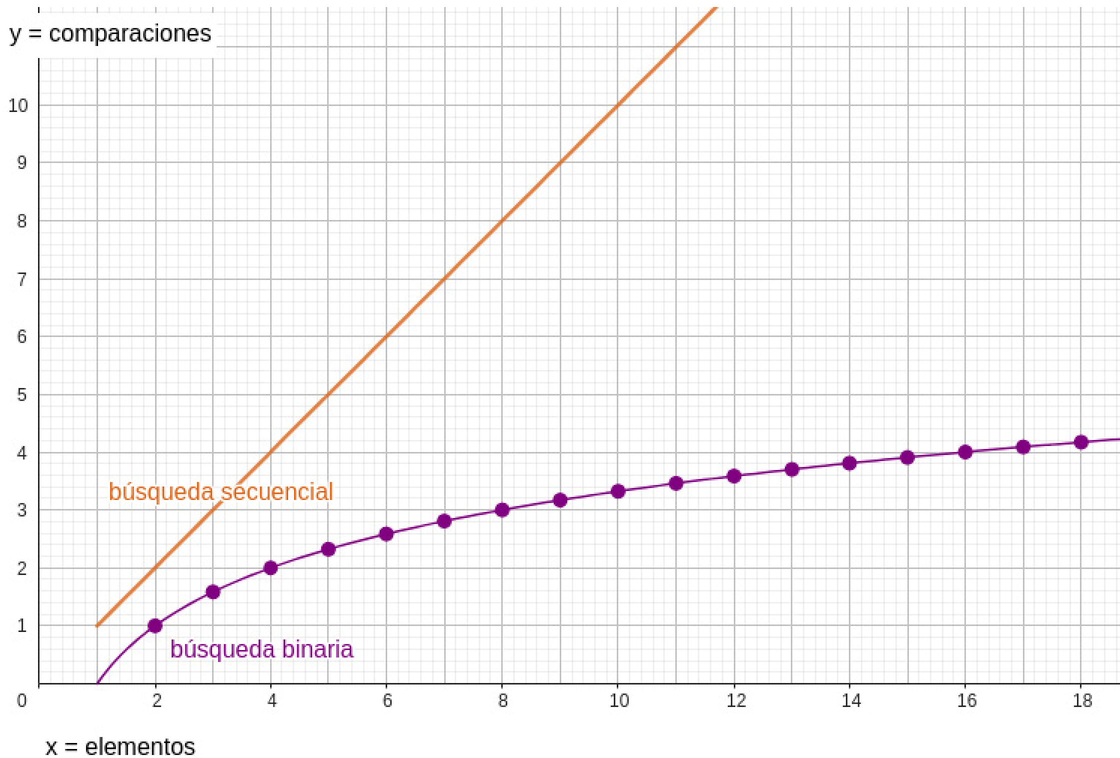


Fig.28 - búsqueda binaria vs secuencial.

[-] métodos provisto por el lenguaje

Java provee clases utilitarias para buscar elementos en una colección. Sólo se hará mención de algunos de ellos dado que para su completa comprensión aún se requieren profundizar conceptos como las interfaces, las estructuras de datos avanzadas y el polimorfismo:

- La clase **`Collections`** proporciona métodos estáticos para manipular y buscar elementos en colecciones. Algunos métodos de búsqueda incluyen **`binarySearch()`**, **`max(), min(), y frequency()`** entre otros.

- La clase **Arrays** ofrece métodos estáticos para manipular y buscar elementos en arreglos. Algunos métodos de búsqueda incluyen **binarySearch(), equals(), fill() y sort()** entre otros.
- **String**: La clase String en Java también proporciona métodos para buscar elementos en cadenas de caracteres, como **indexOf(), lastIndexOf(), contains(), startsWith(), endsWith()** y **matches().**
- La API de flujos (Stream) proporciona operaciones de búsqueda y filtrado, como **filter(), findAny(), findFirst(), anyMatch(), allMatch(), noneMatch(),** entre otros.
- Las interfaces **Map** y **Set** proporcionan métodos de búsqueda específicos para buscar claves o valores en mapas y elementos en conjuntos.

Ordenamiento

El ordenamiento es el proceso de organizar una colección de elementos en un orden específico, generalmente ascendente o descendente, según un criterio determinado. Esta organización no solo mejora la presentación y la legibilidad de la información, sino que también tiene un impacto significativo en la eficiencia de las operaciones que se realizan sobre ella.

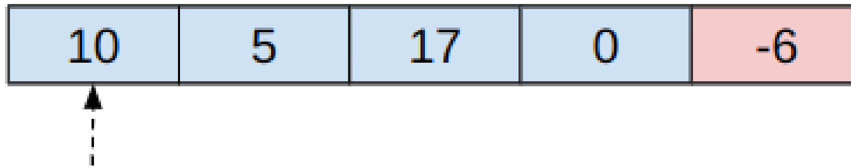
La búsqueda binaria, como vimos anteriormente, explota la organización de los datos para encontrar un elemento específico de forma rápida y eficiente. Este algoritmo sólo funciona si los elementos están ordenados, ya que se basa en la comparación de elementos específicos para dividir el espacio de búsqueda a la mitad en cada iteración.

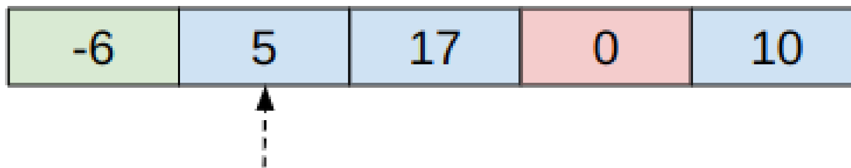
Alternativamente, tener los datos ordenados, nos permitirá presentarlos en pantalla, u otro dispositivo, en función de las necesidades de los usuarios.

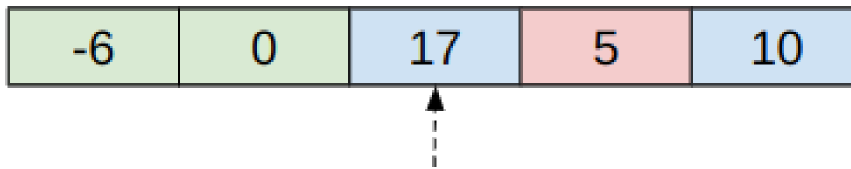
Si bien existen muchos métodos de ordenamientos nos centraremos por ahora en la comprensión de los tres más simples.

[-] método de selección

El método de selección es un método de ordenamiento que se basa en encontrar el elemento más pequeño de la colección e intercambiarlo con el que ocupa la posición más baja -la primera-, luego se busca el siguiente más pequeño y se intercambia con el que ocupa la siguiente posición -la segunda-. Este proceso se repite hasta que toda la colección queda ordenada.







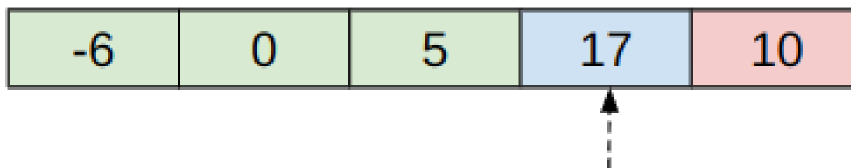


Fig.29 - ordenamiento por selección.

A continuación mostramos una implementación del método de ordenamiento por selección:

```
//
public static void seleccion(int[] vector) {

    //
    for (int i = 0; i < vector.length - 1; i++) {

        //
        int posMenor = i;

        //
        for (int j = i + 1; j < vector.length; j++) {

            //
            if (vector[j] < vector[posMenor]) {
                posMenor = j;
            }
        }

        //
        int menor = vector[posMenor];
        vector[posMenor] = vector[i];
        vector[i] = menor;
    }
}
```

Cabe aclarar que si quisiéramos ordenar los elementos de forma descendente entonces tendríamos que buscar el mayor de los elementos e intercambiarlo con el que se encuentra en la posición más baja del vector y de esta manera continuar repitiendo el proceso hasta acomodar el resto de los elementos.

[-] método de burbuja

El método de burbuja funciona recorriendo la colección, verificando en cada pasada como se encuentran los elementos adyacentes, o sea, si se encuentran en orden o no. En el caso de que no estén en orden se intercambian.

El procedimiento asegura llevar -o burbujear- el elemento más grande hacia el final de la colección.

Si el criterio es ordenar de forma ascendente los valores de la colección, entonces puedo asegurar que en la primera pasada lleva al final de la colección el elemento que tiene el valor más grande. Lógicamente si quisiera ordenar de forma descendente, el elemento que arrastro hacia el final será el más chico.

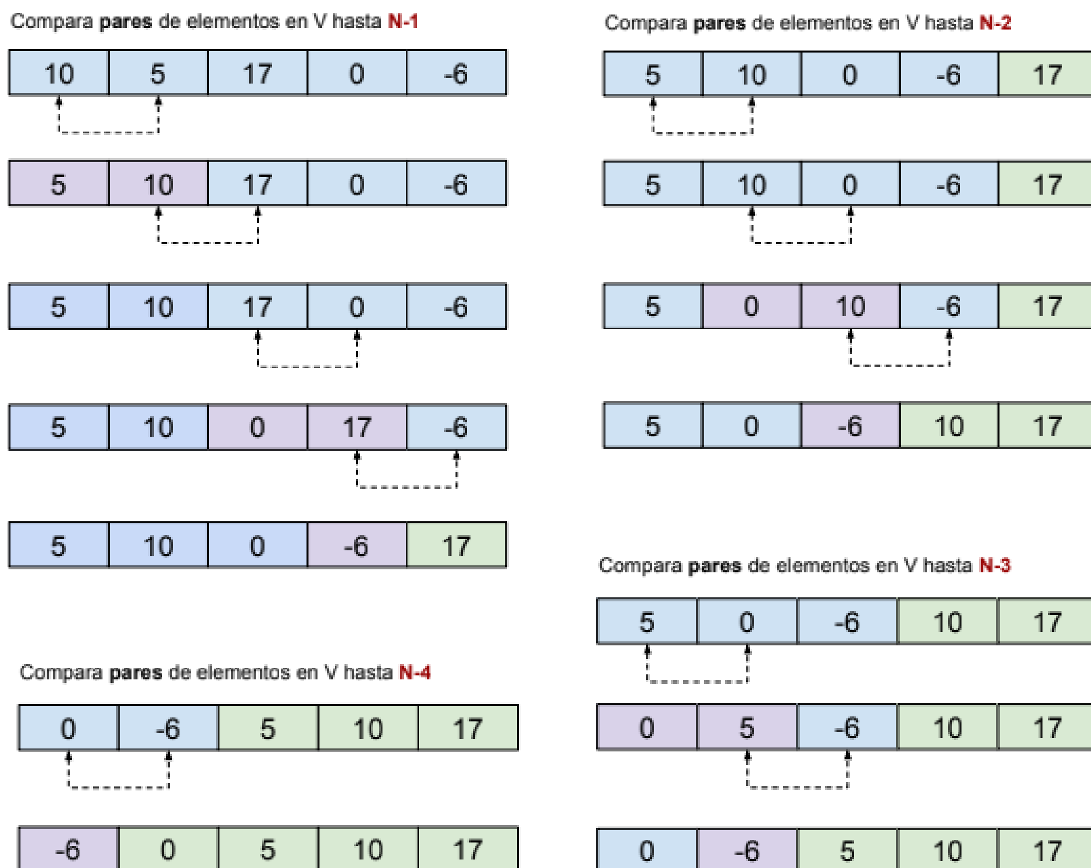


Fig.30 - ordenamiento por burbuja.

A continuación mostramos una implementación del método de ordenamiento por burbuja:

```
//
public static void burbuja(int[] vector) {

   //
   for (int i = 1; i < vector.length ; i++) {

       //
       for (int j = 0; j < vector.length - i ; j++) {

           //
           if (vector[j] > vector[j + 1]) {

               //
               int temp = vector[j];
               vector[j] = vector[j + 1];
               vector[j + 1] = temp;
           }
       }
   }
}
```

El algoritmo en cada pasada -el for que usa la variable j- detecta aquel elemento que no se encuentre en su posición y lo lleva hasta su lugar. Si luego de la pasada detectamos que no se producen intercambios podemos concluir que la colección se encuentra ordenada y terminar la ejecución. Al método con esta mejora se lo conoce como "burbuja mejorada".

Veamos cómo podemos implementar esta mejora:

```
//
public static void ordenar(int[] vector) {

  //
```

```
    for (int i = 1; i < vector.length ; i++) {

        //
        boolean ordenado = true;

        //
        for (int j = 0; j < vector.length - i ; j++) {

            //
            if (vector[j] < vector[j + 1]) {

                //
                int temp = vector[j];
                vector[j] = vector[j + 1];
                vector[j + 1] = temp;

                //
                ordenado = false;
            }
        }
        if(ordenado){
            break;
        }
    }
}
```

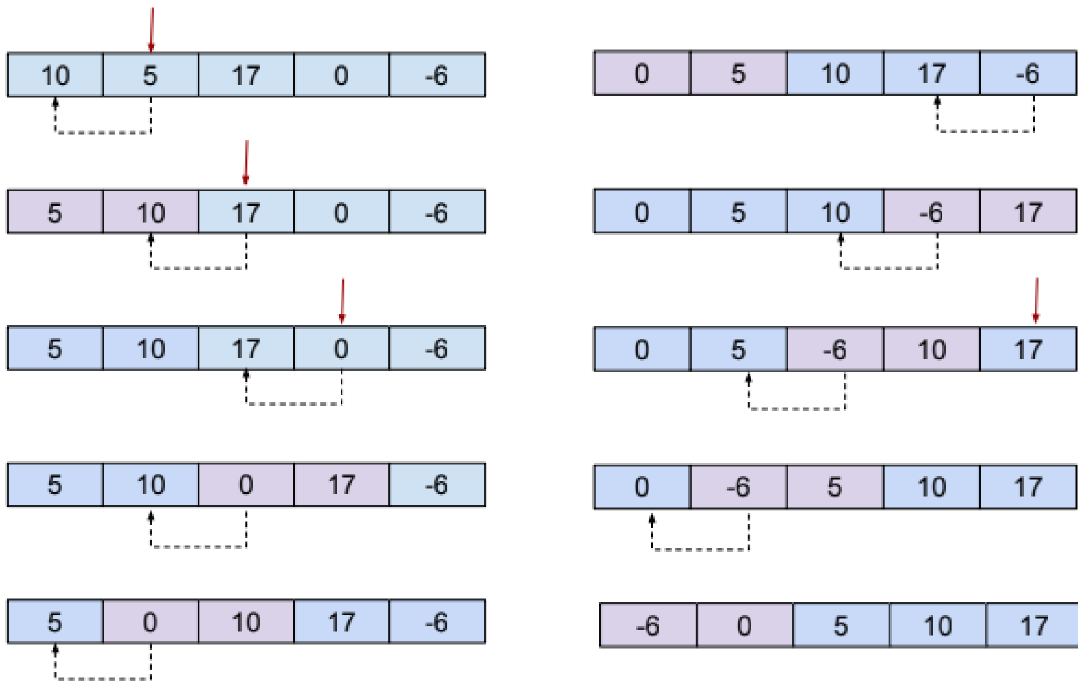
[-] método de inserción

El método de inserción funciona de un modo similar al de burbuja en el sentido que se recorre el vector desde la 2da posición hasta el final. En su camino llevará el elemento actual -el de la pasada- hasta su posición. Una vez ubicado retoma su posición y continúan repitiendo este proceso hasta llegar al final.

Para clarificar un poco el comportamiento de cómo funciona el método para ordenar una colección de forma ascendente diremos que los pasos que debe realizar son los siguientes:

1. Realizaremos una pasada completa, desde el 2do elemento de la colección hasta el último.
   a. Guardamos el valor del elemento actual -en el primer caso será el 2do.
   b. Desde el elemento actual iremos hacia atrás hasta determinar que se encuentra en su posición.
      i. Si el elemento actual es más grande o igual que el anterior significa que se encuentra en su posición y terminamos esta repetición.
      ii. Si el elemento actual es más chico entonces hay que buscar su posición, para esto reemplazamos el valor del elemento actual con el anterior y volvemos al punto (b) pero ahora con el anterior al actual.
   c. Hemos determinado la posición del elemento y si fue necesario hemos desplazado los que no estaban en su lugar, ahora establecemos el valor del elemento guardado en su lugar.
   d. Volvemos al punto (1) incrementando la posición al siguiente.

A continuación se mostrará el algoritmo de una forma gráfica. Las flechas rojas indican el primer bucle -el que va de la 2da posición hasta el final-, las líneas punteadas indican la



comparación que se realiza con el elemento anterior.

Fig.31 - ordenamiento por inserción.

Veamos una implementación del método:

```
//
public static void insercion(int[] vector) {

    //
    for (int i = 1; i < vector.length; ++i) {

        //
        int elemento = vector[i];
        int j = i - 1;

        //
        while (j >= 0 && vector[j] > elemento) {
            vector[j + 1] = vector[j];
            j = j - 1;
        }

        //
        vector[j + 1] = elemento;
    }
}
```

Los tres métodos de ordenamiento desarrollados comparten la característica de ser poco eficientes en comparación con otros algoritmos más avanzados que no profundizaremos en este libro, como el MergeSort, el QuickSort y el HeapSort, entre otros. La razón principal de su baja eficiencia radica en su complejidad temporal, que ya hemos definido como el tiempo que requiere el algoritmo o el número de operaciones para terminar. Entonces, si nos quedamos con las operaciones de comparación podemos determinar que los tres métodos realizan una cantidad de operaciones equivalente al tamaño de la colección que se quiere ordenar.

Los tres métodos tienen una complejidad temporal de O(n2), lo que significa que el tiempo de ejecución aumenta exponencialmente a medida que aumenta el tamaño del vector (n). En otras palabras, si tengo un vector de 10 elementos, necesitaré aproximadamente 102 operaciones para ordenar la colección.

Esta complejidad temporal se establece debido a que exploramos el peor caso, pero si hacemos una pausa y también analizamos el mejor caso o inclusive el caso promedio, de cada uno de los métodos, podremos determinar que el método de la burbuja -mejorada- y el método de la inserción pueden terminar antes si los elementos ya se encuentran en su posición.

El mejor caso será cuando los elementos estén ordenados o "casi" ordenados.

Por lo tanto el método de la burbuja -mejorada- y el de inserción serán de O(n) en el mejor de los casos y esto los convierte en candidatos.

[-] métodos previsto por el lenguaje

Java provee clases utilitarias para ordenar elementos de una colección. Sólo se hará mención de algunas de ellas dado que para su completa comprensión aún se requieren profundizar conceptos como las interfaces, las estructuras de datos avanzadas y el polimorfismo.

- Clase Collections.sort(): La clase "Collections" proporciona un método llamado sort() que puede ser utilizado para ordenar. Este método utiliza el algoritmo de ordenación "merge sort" para ordenar los elementos en orden natural o mediante un comparador personalizado.
- Clase Arrays.sort(): Similar a Collections.sort(), pero se aplica a arreglos. La clase Arrays proporciona un método sort() que utiliza "dual-pivot quicksort" para ordenar los elementos en un arreglo.
- Estructuras de datos ordenadas: son estructuras de datos específicas que mantienen automáticamente sus elementos en orden, como TreeSet y TreeMap. Utilizan estructuras de árbol para mantener los elementos organizados.

Ejercicios

1. Realizar una función que ordene los elementos de un vector de forma ascendente.

   Ej: void ordenar(int[ ] vector)

2. Realizar una función que ordene utilizando el método “sort” provisto por la clase utilitaria Arrays.
3. Realizar un programa que pida al usuario 10 valores y muestre por pantalla los elementos ordenados de menor a mayor (ascendentes).
4. Idem al anterior pero ahora los elementos deberán estar ordenados de manera descendente.
5. Escribir una función, cuyos parámetros serán un arreglo (desordenado) y un valor, y devuelva la posición donde se encuentra el valor en el arreglo. En caso de que no se encuentre deberá retornar -1.

   Ej: int buscar(int[ ] vector, int x)

6. Realizar un programa que a partir de un vector de 10 elementos, generado de forma aleatoria entre 1 y 99, determine el porcentaje de acierto que tuvo el usuario para acertar a un valor que contiene el arreglo.

   Ej: arreglo = {1,6,2,8,0}

   valores del usuario:

   3 :: no se encuentra, intente nuevamente

   5 :: no se encuentra, intente nuevamente

   7 :: no se encuentra, intente nuevamente

   0 :: ACIERTO!!!

   El porcentaje de acierto fue del 25%

7. Realizar un análisis de las precipitaciones durante el año. El objetivo es mostrar en orden descendente los meses acompañados por la cantidad de lluvia promedio ocurrida en cada uno.

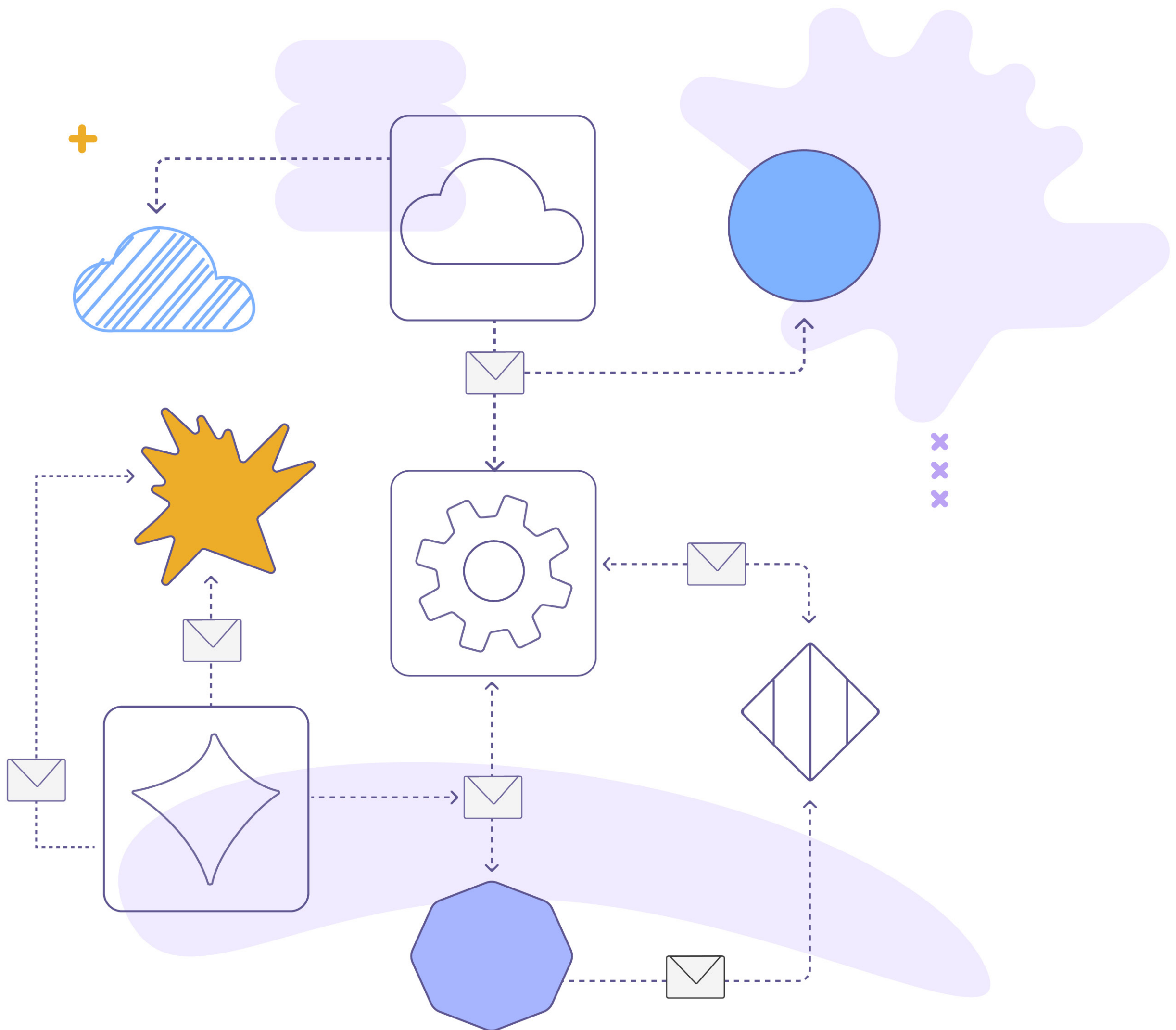
Ej:

- Enero: 60mm
- Febrero: 55 mm
- Marzo: 70 mm
- Abril: 80 mm
- Mayo: 75 mm
- Junio: 65 mm
- Julio: 60 mm
- Agosto: 55 mm
- Septiembre: 70 mm
- Octubre: 75 mm
- Noviembre: 80 mm
- Diciembre: 85 mm

8. A partir del ejercicio anterior, mostrar que mes superó el promedio de lluvia caída anualmente.

Capítulo 9

Análisis y diseño de algoritmos



Capítulo 9

Paradigma orientado a objetos

Objetivo

En este capítulo aprenderemos los fundamentos del paradigma orientado a objetos como metodología fundamental para la programación actual. Se brindarán las herramientas y el conocimiento para crear soluciones eficaces, flexibles y reutilizables. Se desarrollarán conceptos elementales como las clases, los objetos y los métodos, entre otros.

Al finalizar este capítulo, estarás capacitado para comprender y aplicar los fundamentos de la programación orientada a objetos. Habrás adquirido una comprensión profunda de los beneficios que aporta.

Paradigma

En nuestro mundo, el de la programación, un paradigma es una forma de pensar. Es el marco que define cómo se diseñan, estructuran y escriben los programas.

En este sentido, el paradigma orientado a objetos, es un modelo de programación que se basa en la idea de que un programa puede entenderse como un conjunto de objetos que interactúan entre sí para realizar una tarea.

El paradigma orientado a objetos se caracteriza por la encapsulación -capacidad de ocultar los detalles internos-, la herencia -capacidad de crear clases a partir de otras- y el polimorfismo -capacidad de que un objeto se comporte diferente dependiendo el contexto-, conceptos que luego ampliaremos dado que son claves para comprender su dominio.

Partamos de un ejemplo concreto como para tener una idea cercana a qué nos referimos cuando hablamos de objetos o clases.

En la vida real existen diferentes tipos de vehículos, autos, camiones o colectivos. Más allá de sus diferencias todos comparten ciertas características en común, todos tienen ruedas, un motor, una patente o un color determinado. Entonces podemos decir que “Vehículo” representa la clase de objetos que puede agrupar a todos estos. La clase especifica cuáles serán las propiedades comunes que podrían tener.

Por el otro lado, cada vehículo individual será un objeto de esta clase Vehículo. En nuestro

sistema, los vehículos individuales serán las instancias con sus características específicas. Por ejemplo un auto Fiat, modelo Punto, color rojo con la patente AA-002-AA.

Si bien podemos llegar a tener objetos de una clase con las mismas características, siempre habrá algo que debe identificarlos, por ejemplo la patente. A esto llamaremos identificación y será la o las características que definen de manera unívoca a un objeto.

A continuación podemos ver de manera gráfica la representación de tres objetos distintos que pertenecen a la misma clase Vehículo.

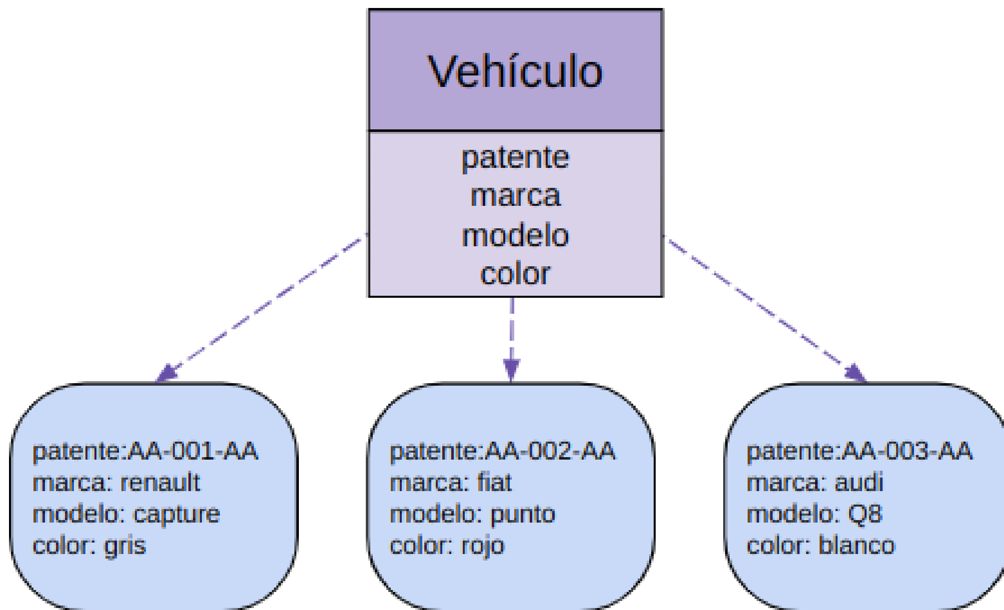


Fig.32 - clase y objetos.

Hasta este punto hemos abordado conceptualmente las ideas de clase y objeto. Sin embargo, es importante destacar que sólo hemos enfocado nuestra discusión en las características de estos objetos, sin profundizar en la importancia del comportamiento. La realidad es que la abstracción de objetos de interés no solo se limita a sus atributos, sino que también implica comprender y modelar su comportamiento. Por lo tanto, explorar cómo los objetos interactúan y se comportan en un sistema es un aspecto fundamental que desarrollaremos con mayor profundidad.

Clase

Una clase es una plantilla o modelo que describe las propiedades y comportamientos de un conjunto de objetos relacionados.

Es un tipo de dato que define un conjunto de atributos y métodos que caracterizan a los objetos creados a partir de ella.

Por el momento cuando hablemos de métodos pensemos en funciones.

En nuestro ejemplo la clase "Vehículo" describe sus atributos y será el medio por el cual crearemos los objetos necesarios.

Veamos algo de código para sustentar estas ideas. La clase Vehículo tendrá la siguiente definición:

```
public class Vehiculo {

   //
   private String patente;
   //
   private String marca;
   //
   private String modelo;
   //
   private String color;

   // Constructor
   public Vehiculo(){
    //…
   }

}
```

La clase “Vehiculo” tiene definidos los atributos patente, marca, modelo y color.

Luego hablaremos de la palabra reservada “private” y de otras relacionadas, por ahora pensemos que sólo los objetos de la clase “Vehiculo” podrán tener acceso a estos atributos.

Objeto

Un objeto es una instancia o ejemplar de una clase, es decir, es una entidad concreta que se crea a partir de un modelo o plantilla especificada por una clase.

Cada objeto tiene un estado, definido por los valores de sus atributos, y un comportamiento definido por los métodos de su clase, las funciones.

Es importante destacar que a diferencia de una clase que se crea al inicio de la ejecución y es eliminada al final, un objeto puede ser creado, modificado -los valores de sus atributos- y eliminado durante la ejecución de un programa las veces que sea necesario.

El siguiente código muestra una clase “Programa” que se utiliza para crear una instancia, o un objeto, de la clase “Vehiculo”.

```
public class Programa {

   public static void main(String[] args) {

       //
       Vehiculo miAuto = new Vehiculo();

   }
}
```

Del lado izquierdo de la asignación tenemos “Vehiculo miAuto”. Esto significa que el objeto se llama miAuto y se corresponde a la clase “Vehiculo”, o sea, es del tipo “Vehiculo”. Del lado derecho de la asignación se aprecia su construcción “new Vehiculo()”. Esta última sentencia indica que debe crearse el objeto a partir de la clase “Vehiculo”.

[-] constructor

Un constructor es un mecanismo especial que se utiliza para crear e inicializar objetos a partir de una clase. El mecanismo de construcción se invoca cuando se crea un objeto. El operador que se utiliza para especificar esta situación es “**new**”.

En otras palabras, el constructor le da vida al objeto y establece su estado inicial.

En Java, el constructor tiene el mismo nombre que la clase y puede tener parámetros para permitir la inicialización de los atributos del objeto con estos valores específicos.

En el código del programa se puede ver la invocación del constructor de la clase “Vehiculo” a través del uso del operador “**new**”.

```
//
Vehiculo miAuto = new Vehiculo();
```

El operador “**new**” reserva, por intermedio de la máquina virtual y ésta a través del sistema operativo, un espacio en la memoria para alojar al objeto creado. Esta acción es conocida como *alocar*

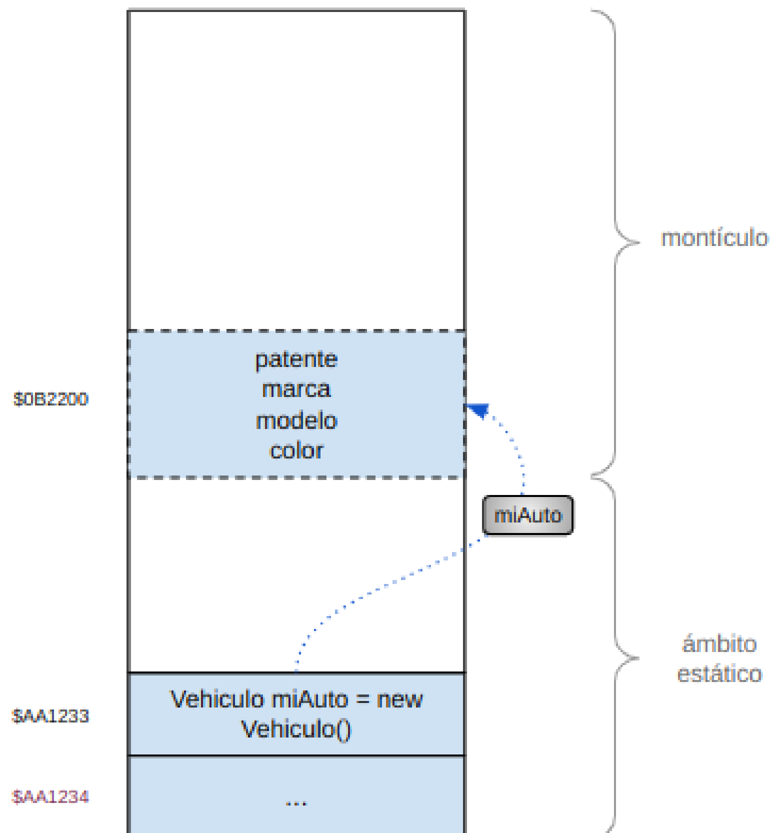


Fig.33 - memoria - creación de un objeto.

El constructor, además de crear, sirve para establecer los valores iniciales que tendrá el objeto. Para que este mecanismo sea posible se deberá declarar un constructor que exponga este requerimiento. Cada clase puede tener la cantidad de constructores que requiera.

Podemos agregar a la clase vehículo el constructor para establecer todos los atributos al momentos de instanciar el objeto:

```
// Constructor
public Vehiculo(String patente, String marca,
                String modelo, String color) {

   this.patente = patente;
   this.marca = marca;
   this.modelo = modelo;
   this.color = color;

}
```

Ahora en nuestro programa podríamos realizar la creación del objeto de dos maneras distintas, o utilizamos el constructor por defecto "`new Vehiculo();`" o utilizamos el constructor que establece el estado inicial a través de los argumentos con el cual lo invocamos:

```
//
Vehiculo vehiculo;

//
vehiculo = new Vehiculo("AA-001-AA",
                        "renault",
                        "capture",
                        "gris");
```

En la definición del constructor, utilizamos la palabra clave "this", la cual se utiliza para referirse al objeto actual (a este objeto - this object) en el que se está trabajando en un momento determinado. Se puede usar para referirse a los atributos o métodos de la clase actual.

En el ejemplo del constructor con parámetros podemos ver el uso de esta palabra clave, lo que indica es que al **atributo** del **objeto** se le asigne el valor del **parámetro**:

```
// Constructor
public Vehiculo(String patente, String marca.....) {

   this.patente = patente;
...
```

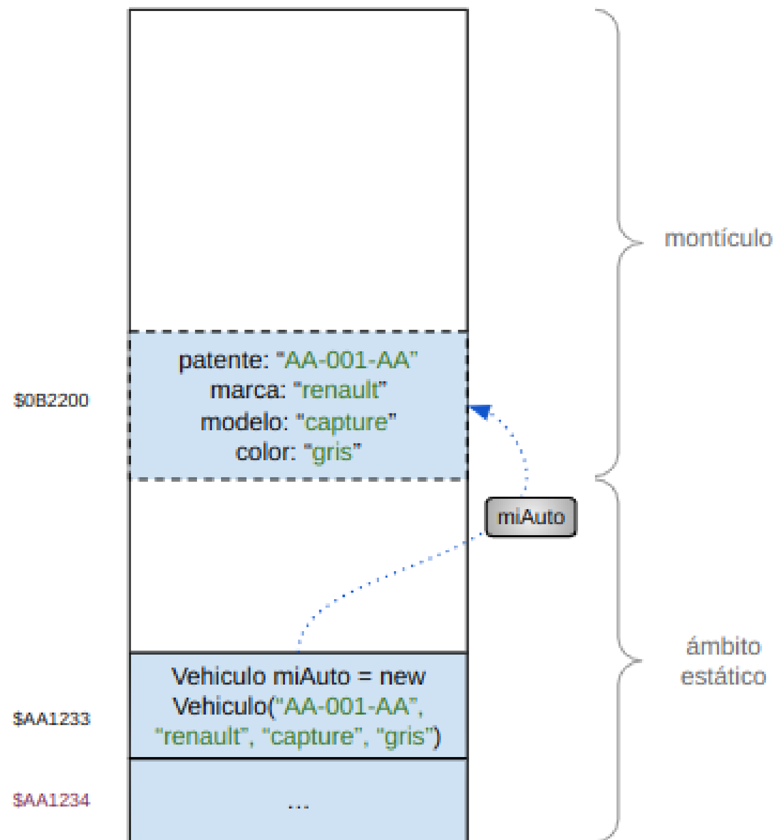


Fig.34 - memoria - creación de un objeto con parámetros.

[-] el operador igualdad ==

Los objetos residen en el área de la memoria llamada montículo, y el operador de comparación "==" lo que verifica es si el contenido en el espacio de memoria es igual del lado izquierdo que del lado derecho, por lo tanto si se utiliza este comparador entre dos objetos, aún de idénticos atributos, dará falso si sus referencia en la memoria son distintas, o sea, si residen en dos espacios de memorias diferentes.

El siguiente código crea objetos y luego imprime por pantalla si estos son iguales:

```
//creo un objeto
Vehiculo vehiculoA = new Vehiculo();

//creo un objeto
Vehiculo vehiculoB = new Vehiculo();

//comparación de las áreas donde residen
System.out.println("Los objetos son iguales = " +
(vehiculoA == vehiculoB));
```

Por supuesto que dará "false", debido a que se han comparado sus referencias en la memoria.

El gráfico a continuación muestra cómo se disponen en la memoria la creación de los objetos "vehiculoA" y "vehiculoB". Cuando se trata de objetos, el comparador de igualdad "==" verifica las direcciones de memoria y no los valores de sus atributos. Las direcciones de ambos objetos son diferentes, por lo tanto la comparación dará como resultado falso.

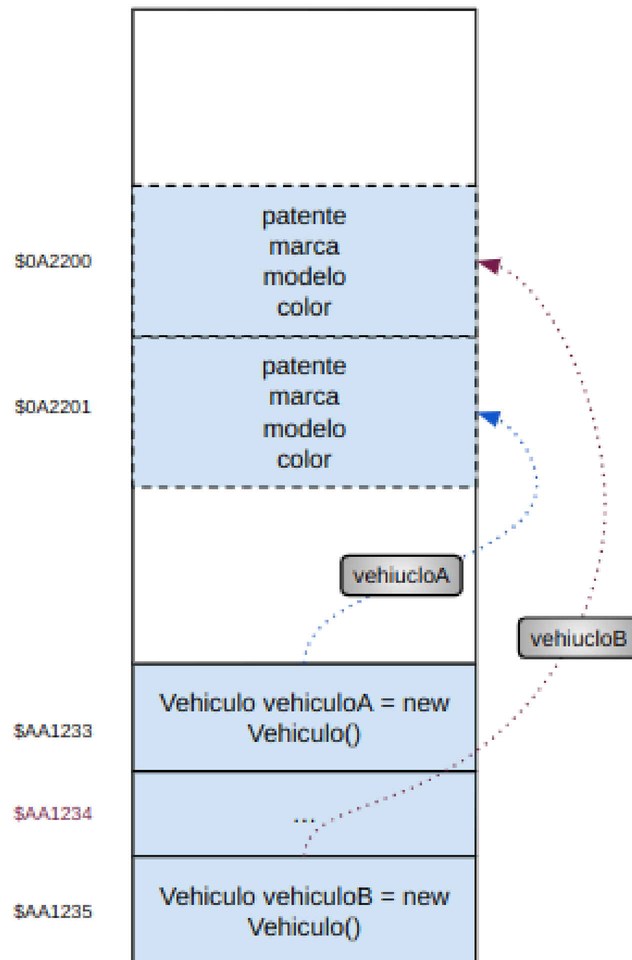


Fig.35 - memoria - operador de igualdad ==.

Los datos primitivos, como hemos visto en el caso de int o long, son valores simples que contienen como valor algún número. Cuando se comparan dos datos primitivos, el operador "==" verifica si tienen exactamente el mismo valor.

Para comparar el contenido de dos objetos, se debe usar el método equals() que compara la identidad entre los objetos.

```java
//creo un objeto
Vehiculo vehiculoA = new Vehiculo();

//creo un objeto
Vehiculo vehiculoB = new Vehiculo();

//comparación de los objetos
System.out.println("Los objetos son iguales = " +
(vehiculoA.equals(vehiculoB)) );
```

El método equals() está definido en la clase Object y, por ahora diremos, que todas las clases son de este tipo también, o sea son de tipo Object. Por lo tanto, puedo usar el método como si lo tuviera presente en mi clase.

Si utilizo el comparador de igualdad “==” entre objetos, estaré comparando sus direcciones de memoria.

Para comparar el contenido de dos objetos, primero se debe establecer qué los identifica, luego se debe sobreescribir el método equals(). Más adelante se explicará cómo especificar esto en la clase para que la comparación sea como pretendemos.

[-] el operador asignación =

El comportamiento del operador varía según si se trabaja con tipos primitivos o con objetos.

Como hemos visto, el operador de asignación con los tipos de datos primitivos -int, long, boolean y otros- funciona asignando el valor a una variable, se crea una copia del valor en la memoria. La variable original y la variable asignada tienen valores independientes.

Ahora, cuando se trata de objetos, el operador asigna la referencia del objeto a la variable. La variable original -el objeto- y la variable asignada comparten la misma referencia de la memoria al mismo objeto.

El siguiente código muestra la creación de dos objetos y luego uno de ellos se asigna a la variable del otro:

```java
//creo un objeto
Vehiculo vehiculoA = new Vehiculo();

//creo un objeto
Vehiculo vehiculoB = new Vehiculo();

//asigno la dirección donde reside vehiculoA
vehiculoB = vehiculoA;

//comparación
System.out.println("Los objetos son iguales = " +
(vehiculoA == vehiculoB) );
```

En este caso la comparación será verdadera, porque como hemos mencionado, el operador compara referencias en la memoria y justamente estas variables tienen las mismas direcciones. Veamos a través de un gráfico como sucede la asignación.

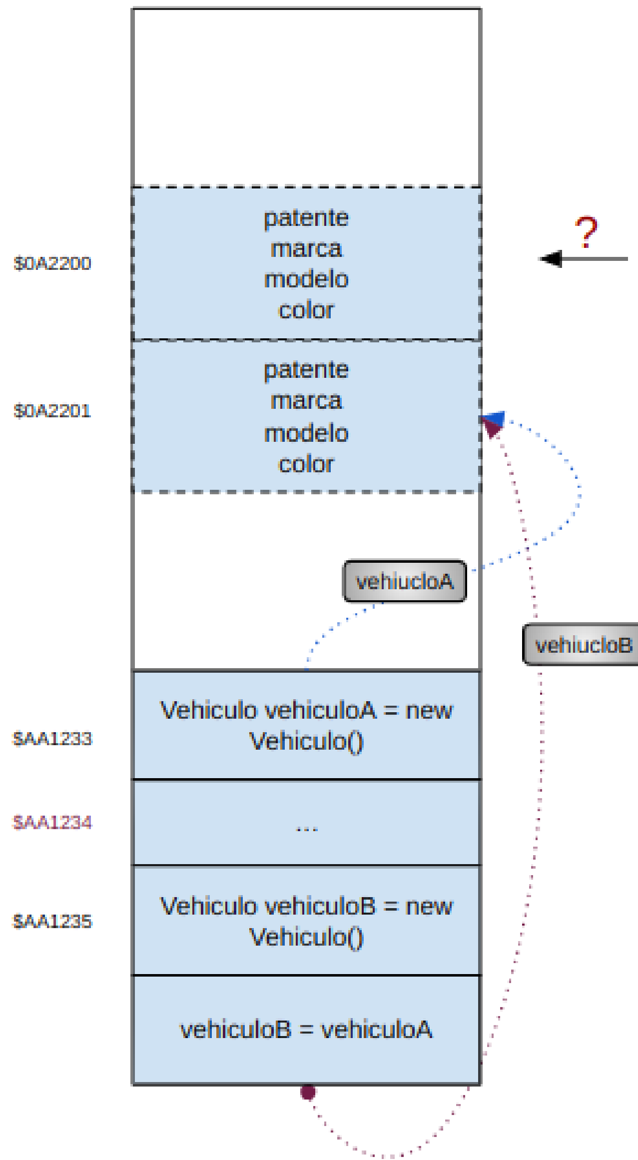


Fig.36 - memoria - asignación de otro objeto.

Analicemos el gráfico. Podemos ver que es lo que sucede luego de asignar a la variable "vehiculoB" la referencia en la memoria de "vehiculoA". Ha quedado un espacio en la memoria utilizado que no es posible llegar a través de ninguna variable de nuestro programa.

¿Cómo liberamos este espacio?

En Java, a diferencia de otros lenguajes, el mecanismo para liberar las zonas que contienen datos en la memoria y estas no son utilizadas por nadie, se realiza automáticamente a través de un componente que se llama "recolector de basura" (Garbage Collector). Cada cierto intervalo de tiempo, la máquina virtual ejecuta el recolector de basura, limpiando aquellas áreas que no son referenciadas por nadie, marcandolas como libres.

Si bien este mecanismo es muy sencillo de usar, tenemos que estar atentos de que no queden referencias a los objetos que ya no necesitamos porque si quedan enganchados no se va a liberar ese espacio y este problema en el tiempo ocasiona lo que se denomina comunmente como laguna de memoria (memory leak).

[-] el valor nulo

El valor nulo es un valor especial que pueden tener las variables. Se especifica a través de la palabra reservada "null". Se utiliza para indicar que una variable no apunta o referencia a ningún objeto válido en la memoria.

El valor null sólo puede utilizarse en variables que no sean primitivas, sólo aplica a los objetos. Cuando se asigna null a una variable diremos que esa variable no está haciendo referencia a ningún espacio en la memoria, o sea, no apunta a ningún objeto existente en ese momento.

Es importante diferenciar a una variable que contiene el valor null a una variable que tiene un objeto vacío. Un objeto vacío significa que se ha reservado espacio en la memoria pero aún no se ha asignado ningún valor a sus atributos. En cambio, null indica directamente que no hay un objeto asociado.

Los siguiente ejemplos muestran la inicialización de las variables como "nulas", no tienen un objeto referenciado en la memoria:

```
//
String texto = null;

//
Integer numero = null;

//
Vehiculo auto = null;
```

Hay que tener especial cuidado de no utilizar operaciones o enviar mensajes a variables que se encuentran nulas.

El siguiente ejemplo desencadena un error en tiempo de ejecución al intentar averiguar la longitud del texto. El error se produce porque el objeto "texto", de tipo String, no se ha creado, tiene la referencia de la memoria a null. Enviar un mensaje a un objeto en esta situación siempre dará un error:

```
//
String texto = null;

//
System.out.println("Longitud del texto: " +texto.length());
```

Este tipo de situaciones no suele surgir con tanta frecuencia, pero cuando tengamos alguna duda al respecto se recomienda verificar si la variable es nula, o mejor dicho, comprobar que no sea nula para poder actuar.

El siguiente ejemplo verifica que el objeto texto no sea nulo para poder enviar el mensaje al objeto y esperar que nos retorne su longitud:

```
//
String texto = null;

...

//
if (texto != null) {

   //
   System.out.println("Longitud: " +texto.length());

}else{

   //
   System.out.println("La variable es nula.");
}
```

[-] contexto estático

La palabra reservada "static" en Java se utiliza para declarar métodos, variables o bloques de inicialización que pertenecen a la clase en sí, en lugar de pertenecer a las instancias individuales, o sea, a los objetos de esa clase.

- Variables estáticas: las variables estáticas son variables de la clase, lo que implica que serán compartidas por todas las instancias de esa clase. Por ejemplo, podría utilizarse para llevar la cuenta de la cantidad de objetos que se crearon.
- Métodos estáticos: son las funciones que se asocian a la clase, no a los objetos creados. Se puede llamar a un método estático de la clase sin necesidad de crear una instancia. Por ejemplo, siguiendo la lógica anterior, podríamos tener un método que retorne la cantidad de instancias que se crearon. Otro ejemplo de método estático es el método para calcular la raíz cuadrada de un número de la clase Math. Math.sqrt(int) se utiliza sin crear un objeto, directamente a través de la clase puedo obtener la raiz cuadrada del argumento.

En general tendremos acceso a los métodos de clase y los atributos de clase quedarán en el ámbito privado de la misma.

El siguiente gráfico muestra el ejemplo que mencionamos acerca de llevar la cuenta de la cantidad de veces que se crearon objetos.

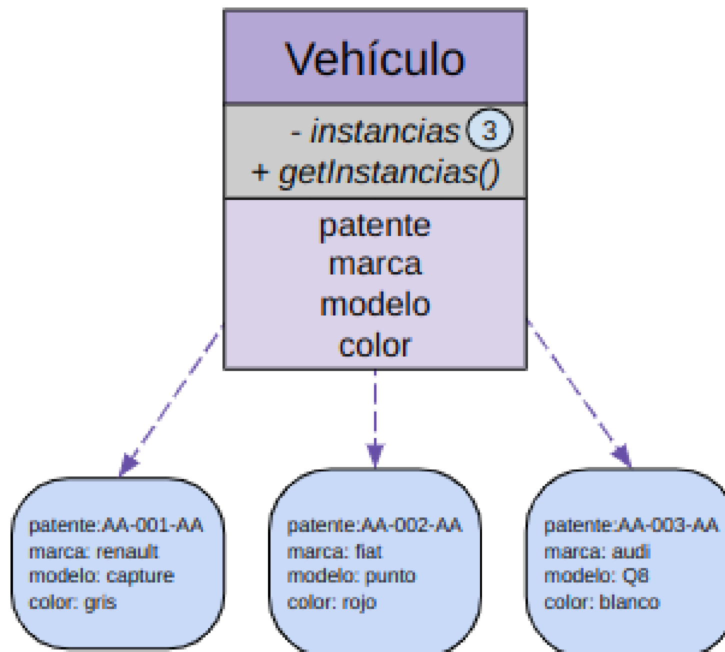


Fig.37 - contexto estático.

Continuemos con el ejemplo. Si queremos llevar la cuenta de la cantidad de veces que se crearon objetos de esta clase, debemos tener una variable de clase para guardar e incrementar su valor cada vez que se cree un objeto.

El lugar para incrementar el valor de la variable será lógicamente en nuestro constructor de clase. Cada vez que se invoca al constructor nosotros incrementaremos el valor de la variable “instancias”.

Veamos el código de este ejemplo:

```
public class Clase {

   private static int instancias = 0;

   public Clase(){

       //
       instancias++;
   }

   public static int getInstancias(){
       return instancias;
   }

   public static void main(String[] args) {

       Clase objeto1 = new Clase();
       Clase objeto2 = new Clase();
       Clase objeto3 = new Clase();

       System.out.println(Clase.getInstancias());
   }
}
```

Si nos fijamos en la última sentencia veremos que podemos obtener la cantidad de objetos creados a partir de la llamada al método de clase. Notar que a diferencia de los métodos de instancia este método, al ser de clase, se llama a través del nombre de la clase y a continuación el nombre del método:

```
Clase.getInstancias();
```

[-] acceso

El acceso a los atributos, constructores y métodos se encuentra relacionado con uno de los conceptos principales del paradigma, la encapsulación, la que se refiere a la idea de que los datos y operaciones relacionados con un objeto deben estar contenidos dentro de ese objeto, para que puedan ser protegidos de accesos no autorizados y para que el objeto pueda interactuar con otros objetos de manera segura.

Dicho de otro modo, haremos explícita la posibilidad de acceder a un atributo o a una operación a través del uso de las palabras reservadas “private”, “public” y “protected”.

En el ejemplo de la clase “Vehiculo”, los atributos están señalados como de uso privado “private”, o sea, sólo se puede acceder a los atributos desde la instancia del objeto y no desde otro lugar. En cambio el constructor está señalado como público a través de la palabra “public” lo cual permite que desde cualquier lugar se pueda realizar la operación, concretamente crear la instancia de la clase.

```
//
private String color;

...

// Constructor
public Vehiculo(...
```

Private indica que el miembro -atributo o método- sólo es accesible dentro de la clase en la que se declara. Esto significa que no se puede acceder a la variable o método marcado como "private" desde código que se encuentre fuera de la clase, incluso si se crea una instancia de la clase y se quiere acceder a un método, por ejemplo objeto.miMetodoPrivado(), no podrá accederse. Solo desde la clase y sus operaciones se puede acceder a los miembros marcados como "private".

Public indica que el miembro -atributo o método- es accesible desde cualquier parte del programa. Si una clase tiene por ejemplo un método definido como “public”, ya sea de instancia o de clase, entonces implica que se puede acceder a él desde otro lugar que no sea el propio código de la clase.

Protected indica que los miembros son visibles dentro de la clase en la que se declaran, en sus subclases -independientemente del paquete en el que se encuentren- y en las clases del mismo paquete. Este tipo de acceso tendrá sentido cuando desarrollemos el concepto de herencia.

Existe un modificador de acceso adicional, se llama Default o package-private. Este modificador se utiliza cuando no se hace explícito el uso de ninguno de los tres anteriores. Se emplea cuando no se utiliza ningún modificador. Indica que los atributos y métodos que lo usen pueden ser accedidos desde cualquier lugar del paquete donde se encuentre contenido.

```
public class Libro {

    //atributo                          ←se puede acceder desde
    String autor;                       cualquier lugar del mismo
                                        paquete

    //atributo privado
    private String titulo;              ←sólo se puede acceder
                                        desde el interior de la
                                        clase.

    //atributo público
    public String ISBN;                 ← se puede acceder desde
                                        cualquier lugar. NO SE
                                        RECOMIENDA

    //Constructor privado.
    private Libro(){ }                  ← sólo se puede invocar
                                        desde el interior de la
                                        clase.

    //Constructor público.
    public Libro(String titulo) {       ← se puede invocar desde
        this.titulo = titulo;           cualquier lugar
    }
}
```

[-] mensajes

Formalmente, el mensaje es la forma en que los objetos interactúan entre sí. Un objeto puede enviar un mensaje a otro objeto para solicitar información o para pedirle que realice alguna acción.

Una función, en el paradigma orientado a objetos, es un método. Como ya hemos visto, un método será de instancia si la operación la realiza bajo el contexto del objeto o será un método de clase si para la realización el contexto es la propia clase.

A partir del ejemplo del vehículo, si quisieran dar la posibilidad de que nuestro objeto en el tiempo pueda cambiar el color -por ejemplo alguien ha decidido pintarlo- tendríamos que hacer disponible la operación.

La clase "Vehiculo" debería contar con un método de instancia que permita establecer el nuevo color. Por convención usaremos la palabra set seguido del nombre del atributo y su parámetro será el nombre del atributo a establecer.

```
//
public void setColor(String color){
   this.color = color;
}
```

Si quisiéramos obtener el color de nuestro objeto, también tendríamos que contar con la operación obtener color. Por convención usaremos la palabra get seguido del nombre del atributo.

```
//
public String getColor(){
   return this.color;
}
```

A continuación mostramos un "Programa" que utilizará la clase "Vehiculo". El programa debe permitir crear un vehículo con valores preestablecidos, luego cambiarle el color y finalmente debe imprimir el color que tiene:

```
public class Vehiculo {

    //
    private String patente;
    private String marca;
    private String modelo;
    private String color;

    // Constructor
    public Vehiculo(){
    }

    // Constructor
    public Vehiculo(String patente, String marca,
                    String modelo, String color) {
        this.patente = patente;
        this.marca = marca;
        this.modelo = modelo;
        this.color = color;
    }

    //
    public void setColor(String color){
        this.color = color;
    }

    //
    public String getColor(){
        return this.color;
    }
}
```

```java
public class Programa {

   public static void main(String[] args) {

       //
       Vehiculo vehiculo;

       //
       vehiculo = new Vehiculo("AA-001-AA",
                               "renault",
                               "capture",
                               "gris");


       //
       vehiculo.setColor("negro");

       //
       System.out.println("El color del vehículo es:"
                           + vehiculo.getColor());
   }
}
```

Existe una convención para nombrar o identificar los mensajes que los objetos pueden recibir. Esta convención de nombres (java naming convention) establece una serie de reglas para nombrar los métodos, con el objetivo de que el código sea fácil de leer y entender para otros programadores.

Algunas de las reglas más importantes son:

- Usar set o get seguido del atributo si lo único que debe establecerse u obtener es el valor del atributo. Por ejemplo setColor o getColor donde color es el nombre del atributo.
- El nombre de un método debe empezar con una letra minúscula, y si es compuesto, la primera letra de cada palabra debe ser mayúscula. Por ejemplo, "calcularTotal".
- Los nombres de los métodos deben ser descriptivos y representar la acción que realiza el método.
- Los nombres de los métodos no deben ser demasiado largos, pero sí lo suficientemente descriptivos para que se entienda su función.

- Si un método devuelve un valor, su nombre debe reflejar ese valor. Por ejemplo, un método que devuelve la suma de dos números podría llamarse "suma".
- Si un método no devuelve ningún valor, se suele utilizar un verbo en infinitivo para indicar la acción que realiza el método. Por ejemplo, un método que imprime un mensaje por pantalla podría llamarse "mostrarMensaje".
- Los nombres de los métodos no deben empezar con números ni caracteres especiales.

[-] this

Como hemos visto en los códigos anteriores, aparece en varios métodos la palabra especial this. Esta palabra reservada “this” es una referencia especial que se utiliza dentro de una clase para hacer referencia al objeto actual.

Se utiliza para referenciar a la instancia que ha recibido cierto mensaje.

El siguiente ejemplo muestra cómo utilizar “this” para referirse a los miembros del objeto actual:

```
public class Vehiculo {

   //
   private String color;

   // Constructor
   public Vehiculo(){
    this.color = "no-establecido";
   }

   //
   public void setColor(String color){
      this.color = color;
   }

}
```

Como puede observarse, el método setColor y el constructor utilizan la palabra especial “this” para hacer referencia al objeto que ha invocado el método, o dicho de otro modo, al

objeto que se le ha enviado el mensaje "setColor" o, en el caso del constructor, han requerido una nueva instancia.

En el caso del método setColor:

this.color ← hace referencia al atributo del objeto actual.

color ← hace referencia al parámetro del método.

Otro ejemplo de utilización sería querer retornar el objeto a través de un método.

Por ejemplo podríamos querer retornar la referencia del objeto una vez se haya cambiado el color o su patente:

```
//
public Vehiculo setColor(String color){

   //
   this.color = color;

   //
   return this;
}

//
public Vehiculo setPatente(String patente){

   //
   this.patente = patente;

   //
   return this;
}
```

Notar que en esta ocasión ambos métodos retornan un objeto del tipo "Vehiculo". Esto implica que luego de realizar su operación los métodos retornan la referencia del objeto con el que se esta trabajando, el actual, retornan "this".

Con este nuevo diseño, los objetos de esta clase, podrían anidar las llamadas a sus métodos:

```
//
Vehiculo auto = new Vehiculo();

//
auto.setPatente("AA-123-AA").setColor("negro");
```

El método setColor se puede encadenar en el código porque al establecer la patente con el método setPatente se retornó la referencia del objeto, en este caso el objeto auto.

[-] tipo de dato envoltorio

Los tipos de datos "envoltorio" (wrappers) son clases que encapsulan tipos de datos primitivos en un objeto. En java existen ocho tipos de datos primitivos: boolean, byte, char, short, int, long, float y double.

Los tipos de datos "wrappers" proporcionan una forma de representar los tipos de datos primitivos como objetos, lo que permite utilizarlos en contextos que se requieren objetos, más adelante veremos dónde reside su utilidad.

Tipos de datos "wrappers" aportados por el lenguaje:

- Boolean: encapsula un valor booleano (true o false).
- Byte: encapsula un valor entero de 8 bits.
- Character: encapsula un carácter (Unicode).
- Short: encapsula un valor entero de 16 bits.
- Integer: encapsula un valor entero de 32 bits.
- Long: encapsula un valor entero de 64 bits.
- Float: encapsula un valor de punto flotante de 32 bits.
- Double: encapsula un valor de punto flotante de 64 bits.

Cada tipo de datos "wrapper" proporciona métodos para convertir los valores primitivos en objetos y viceversa, así como para realizar operaciones aritméticas y comparaciones entre los valores encapsulados.

El lenguaje aporta un mecanismo automático llamado boxing y unboxing que se explicará más adelante, por ahora basta con decir que el lenguaje se encarga automáticamente de

realizar la conversión de un tipo de dato primitivo con su envoltorio y viceversa.

Veamos algunos ejemplos:

```
//
Integer numA = 123; // asigno un primitivo a un envoltorio

//
Integer numB = Integer.valueOf(123);

//
int primitivo = 123;

//
Integer numC = Integer.valueOf(primitivo);

//
Integer numD = Integer.parseInt("123");

//
System.out.println(numA == 123); // true
System.out.println(numB == 123); // true
System.out.println(numC == 123); // true
System.out.println(numD == 123); // true
System.out.println(primitivo == 123); // true

//
System.out.println(numA.equals(123)); // true

//
System.out.println(numA == numB); // true
System.out.println(numB == numC); // true
System.out.println(numC == numD); // true
//
System.out.println(numA == primitivo); // true
```

Contenedores

Los contenedores son objetos que nos permiten almacenar otros objetos.

Partamos de una situación para comprender por qué los contenedores serán una herramienta esencial en nuestras soluciones.

Supongamos que nos piden que analicemos el registro de las temperaturas máximas y mí-nimas de una semana para determinar qué días tuvimos la máxima temperatura y la mínima. Hasta ahora, y con las herramientas que hemos desarrollado, podríamos resolver esto sin inconveniente.

Si pensamos un poco la solución podríamos tener un clase "Temperatura", que almacene el registro de la temperatura mínima y la máxima:

```
public class Temperatura {

   //
   private int minima;
   private int maxima;

   public Temperatura(int minima, int maxima) {
       this.minima = minima;
       this.maxima = maxima;
   }

   public int getMinima() {
       return minima;
   }

   public int getMaxima() {
       return maxima;
   }
}
```

El programa que utiliza esta clase y un vector para resolver la situación planteada podría ser el siguiente:

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

   //
   Temperatura[] registro = new Temperatura[7];

   for (int i = 0; i < 7; i++) {
       int min = (new Random()).nextInt(-10,5);
       int max = (new Random()).nextInt(min,min + 15);
       //agregamos una nueva temperatura
       registro[i] = new Temperatura(min, max);
   }

   //
   int maxima = -100;
   int minima =  100;

   //
   for (int i = 0; i < registro.length; i++) {
       if (registro[i].getMinima() < minima){
           minima = registro[i].getMinima();
       }
   }

   //Este fragmento es equivalente al anterior aunque
   //itera de otro modo el vector
   for (Temperatura dia: registro){
       if (dia.getMaxima() > maxima){
           maxima = dia.getMaxima();
       }
   }
}
```

Ahora, ¿qué pasaría si la cantidad de días a registrar fuera un dato con el que no contamos? En la misma línea que el desarrollo anterior podríamos pensar en utilizar un arreglo de 365 elementos por si llegan a necesitar cargar todos los días del año. ¿y si fueran más días? la frecuencia podría ser mayor a una mínima y máxima por día.

Si la cantidad de datos que vamos a registrar fuera mayor a la cantidad posible tendríamos que volver a modificar el programa, compilarlo y volver a ejecutarlo.

Si la cantidad de datos fuera significativamente menor, entonces estaríamos desperdiciando espacio reservado en la memoria que se podría utilizar para otros fines.

En este punto podemos decir que nuestra caja de herramientas nos queda chica para ajustar el modelo que resuelve el problema del mundo real. Entonces, ¿cómo seguimos?

La respuesta es sencilla, utilizando contenedores.

Los contenedores son objetos que nos permiten almacenar una cantidad arbitraria de objetos, en general del mismo tipo, y nos proporcionan operaciones para acceder, agregar, eliminar y modificar estos objetos.

En Java, existen varias clases de contenedores, nosotros desarrollaremos las que tienen que ver con listas y diccionarios.

[-] lista

Una lista es una colección ordenada de objetos que se puede acceder por su índice, en otras palabras, es un contenedor que permite almacenar un conjunto de objetos en un orden específico y nos permite acceder a estos por la posición que ocupan dentro de la lista.

Si bien esta definición es parecida a la que dimos cuando hablamos de arreglos, la principal diferencia es que el tamaño será dinámico a diferencia del arreglo que es estático, o sea, fijo. Si al realizar la declaración de un arreglo especificamos que tendremos "n" elementos, así será durante toda la ejecución, se usen o no se usen siempre tendremos n espacios reservados en la memoria.

La lista es una estructura de datos fundamental en programación que permite almacenar una colección ordenada de elementos. Los elementos de una lista pueden ser de cualquier tipo de dato, como números, cadenas, objetos o incluso otras listas.

Se utilizan para almacenar y organizar datos de forma eficiente. Algunas de las operaciones que se pueden realizar con una lista son:

- Agregar: se pueden agregar nuevos elementos al final o en cualquier posición de la lista.
- Eliminar: se pueden eliminar elementos de cualquier posición de la lista.

- Obtener: se puede acceder a cualquier elemento de la lista por su posición.
- Buscar: se puede buscar un elemento específico en la lista.
- Ordenar: se puede ordenar la lista de acuerdo a un criterio específico.

Veamos gráficamente cómo sería una lista a la que se le han agregado seis elementos:

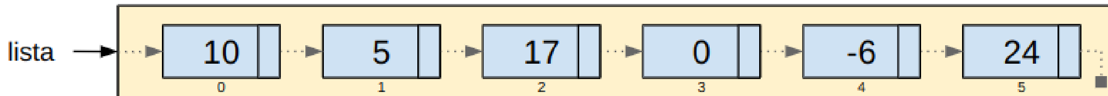


Fig.38 - lista.

Si bien existen varios tipos de lista como ArrayList, LinkedList, Vector y otras más, nosotros utilizaremos por el momento la del tipo ArrayList que además se ajusta a la definición que dimos anteriormente.

Veamos cómo sería la creación de una lista del tipo ArrayList:

```
//
List<String> lista = new ArrayList();
```

Como se puede ver, del lado izquierdo de la asignación encontramos "**`List<String> lista`**". Esto nos dice que queremos utilizar una lista que almacenará cadenas de texto. Del lado derecho "**`new ArrayList()`**". Nos indica que se ha decidido utilizar una lista del tipo ArrayList.

Más adelante diremos por qué es posible que del lado izquierdo sea del tipo "List" y del derecho del tipo "ArrayList" aunque intuitivamente podemos ver que lo que se quiere establecer es "Necesitamos una lista del tipo arraylist".

El objeto "lista", en este caso, tendrá disponible varios métodos para su manipulación, entre ellos destacamos:

- add: para agregar un elemento a la lista.
- get: para acceder a un elemento en una posición determinada de la lista.
- remove: para eliminar un elemento de la lista.

- size: para obtener el número de elementos en la lista.
- indexOf: para obtener la posición de un elemento en la lista.
- clear: para eliminar todos los elementos de la lista.
- isEmpty: para verificar si la lista está vacía.
- contains: para verificar si un elemento está presente en la lista.

A continuación veremos cómo definir una lista para almacenar el nombre de distintos colores. Finalmente recorreremos los elementos para mostrarlos en pantalla:

```
//
List<String> colores = new ArrayList<String>();

//
colores.add("rojo");
colores.add("verde");
colores.add("azul");
colores.add("blanco");
colores.add("amarillo");

//
System.out.println("Contenido de la lista: ");

//
for(int i=0; i < colores.size(); i++){
   String color = colores.get(i);
   System.out.println(color);
}
```

La forma que acabamos de emplear es similar a la que usábamos con los arreglos.

Podemos abstraernos de que estamos en frente de una cantidad de n elementos -**`colores.size()`**- y usar algo más cercano al mundo de los objetos. Podemos utilizar un objeto llamado “Iterator” para iterar sobre la colección. Este objeto nos lo provee la misma lista y además podemos consultar si hay más elementos para iterar o directamente solicitar el siguiente:

```java
//
Iterator<String> iterador = colores.iterator();

//
while(iterador.hasNext()){
   String color = iterador.next();
   System.out.println(color);
}
```

Finalmente, una forma sencilla y más legible aún, es utilizar el estilo de estructura que se llama forEach. Se utiliza básicamente para iterar sobre elementos de una colección de forma simplificada.

El for se repetirá desde el primer elemento hasta el final y luego saldrá de la estructura:

```java
//
for(String color: colores) {
   System.out.println(color);
}
```

Veamos varias operaciones que podrían realizarse con una lista de elementos del tipo String. Vale aclarar que esto se puede hacer con cualquier clase de tipo de dato.

```java
//
List<String> lista = new ArrayList();

//Agregamos objetos de tipo String
lista.add("Hola");
lista.add("Mundo");

//obtenemos el segundo elemento
String texto = lista.get(1);
```

```java
//obtenemos y removemos el segundo elemento
String elemento = lista.remove(1);

//obtenemos la cantidad de elementos
int tamaño = lista.size();

//obtenemos la posición del elemento "Hola"
int posicion = lista.indexOf("Hola");

//limpiamos los elementos de la lista
lista.clear();

//averiguamos si la lista está vacía
boolean vacia = lista.isEmpty();

//averiguamos si la lista contiene el elemento "algo"
boolean contiene = lista.contains("algo");
```

Los objetos del tipo lista pueden contener otros objetos, por ejemplo podríamos pensar en tener una lista de vehículos.

La lista de vehículos, de forma gráfica se vería:

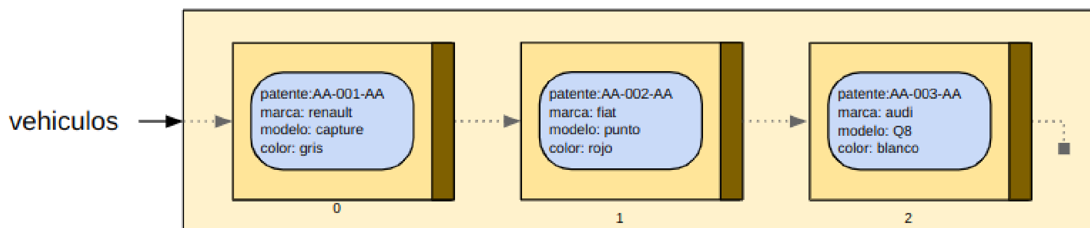


Fig.39 - lista de objetos.

El código que podríamos tener para armar esta lista de vehículos sería:

```
//
List<Vehiculo> vehiculos = new ArrayList();

//
vehiculos.add(
    new Vehiculo("AA-001-AA","renault","capture","gris"));

vehiculos.add(
    new Vehiculo("AA-002-AA","fiat","punto","rojo"));

vehiculos.add(
    new Vehiculo("AA-003-AA","audi","q8","blanco"));

//
mostrarEnPantalla(vehiculos);
```

El método mostrarEnPantalla será un método de clase, la que usamos para solucionar el problema, y estará definido de la siguiente manera:

```
//
public static void mostrarEnPantalla(List<Vehiculo>
 vehiculos){

  //
  for (Vehiculo vehiculo : vehiculos) {

     //
     vehiculo.mostrarDatos();
  }
}
```

Finalmente notemos que el fragmento utiliza el método de instancia “mostrarDatos” definido en la clase “Vehiculo” como se muestra a continuación:

```
//
public void mostrarDatos(){
   System.out.printf("Patente: %s - marca: %s
                     - modelo: %s - color: %s \n",
                     patente, marca, modelo, color);
}
```

Es muy importante darse cuenta que la lista contendrá referencias de los objetos, lo cual implica que si llegara a tener una referencia de un objeto que se encuentra en la lista y la misma realiza el envío de un mensaje para cambiar su estado interno, entonces la lista que lo contiene tendrá esa modificación porque, como indicamos al comienzo del párrafo, la lista tiene la referencia del objeto.

Realicemos un breve ejemplo de esta situación para clarificar. Creamos un objeto en la variable “v”, la agregamos al contenedor y luego modificamos el estado del objeto referenciado por “v”. Al mostrar nuevamente los elementos del contenedor veremos como se encuentra el vehículo con el cambio de estado producido:

```
//
List<Vehiculo> vehiculos = new ArrayList();

//
vehiculos.add( new Vehiculo("AA-001-AA",
                            "renault",
                            "capture",
                            "gris"));


//
Vehiculo v = new Vehiculo("AA-002-AA",
                          "fiat",
                          "punto",
                          "rojo");
```

```
//
vehiculos.add(v);

//
vehiculos.add(new Vehiculo("AA-003-AA",
                           "audi",
                           "q8",
                           "blanco"));


//
mostrarEnPantalla(vehiculos);

//
v.setColor("negro");

//
mostrarEnPantalla(vehiculos);
```

[-] mapa

Los mapas o diccionarios son estructuras de datos que almacenan pares o tuplas clave-valor, donde cada clave estará asociada a un valor.

Cada clave en un mapa es única, lo que significa que no puede haber duplicados de claves en la misma estructura.

Son útiles en una amplia variedad de situaciones debido a su capacidad para representar relaciones entre datos de manera eficiente.

Se utilizan principalmente para realizar búsquedas, almacenar configuraciones, contar elementos y representar relaciones.

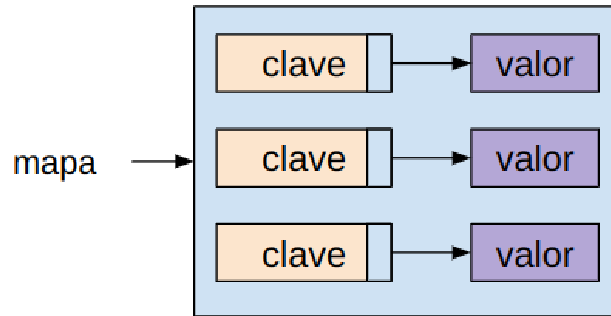


Fig.40 - mapa.

Java nos ofrece varios tipos de mapa al igual que sucede con las listas, nos ofrece HashMap, LinkedHashMap, TreeMap y Hashtable, entre otros. Nosotros utilizaremos por el momento el del tipo HashMap.

Un HashMap utiliza una tabla de hash internamente, lo cual nos permite almacenar pares de clave-valor, asegurando que al agregar esta tupla, se realiza una transformación a la clave que da una posición "única" para almacenar ambos. Esto permite recuperar los datos de una forma muy eficiente, el tiempo asociado a la búsqueda del contenido es O(1), es un tiempo constante.

A continuación mostramos cómo sería la creación de un mapa del tipo HashMap, En este caso nos interesa guardar claves del tipo texto -String- y el valor asociado de tipo entero -Integer-:

```
HashMap<String, Integer> agenda = new HashMap();
```

Por ejemplo, podríamos pensar en tener un mapa para almacenar tuplas del estilo (nombre,número de celular), o sea, un HashMap<String, Integer>:

- "pedro" - 2901401234
- "julieta" - 2901424789
- "ana" - 2213508595

Al igual que las listas, los mapas nos proveen de métodos para su manipulación.

Los principales son:

- put: agrega una entrada al mapa con la clave y el valor correspondiente.
- get: recupera el valor asociado a una clave específica.
- remove: elimina la entrada asociada a una clave específica y devuelve el valor asociado a esa clave antes de eliminarla.
- containsKey: retorna verdadero si el mapa contiene una clave específica.
- containsValue: retorna verdadero si el mapa contiene un valor específico.
- isEmpty: retorna verdadero si el mapa está vacío, es decir, si no contiene ninguna entrada.
- size: retorna el número de entradas (cuantos pares clave-valor) en el mapa.
- clear: elimina todas las entradas del mapa. Lo deja vacío.

Veremos algunas particularidades del uso del diccionario mediante fragmentos de código:

```
//Agregamos pares clave-valor
diccionario.put("primero",1230);
diccionario.put("segundo",340);
diccionario.put("tercero",10);
diccionario.put("cuarto",0);

//obtenemos el segundo elemento
Integer puntaje = diccionario.get("primero");

//obtenemos y removemos una relación
Integer valor = diccionario.remove("cuarto");

//obtenemos la cantidad de relaciones
int tamaño = diccionario.size();

//verificamos si existe la clave "quinto"
boolean existe = diccionario.containsKey("quinto");
```

```java
//limpiamos el diccionario
diccionario.clear();

//averiguamos si el diccionario está vacía
boolean vacia = diccionario.isEmpty();
```

A continuación mostramos un ejemplo que asocia a cada nombre una edad. Luego se recuperan, muestran e imprimen todas.

```java
// Inicializa claves de tipo String y valores de tipo Integer
HashMap<String, Integer> diccionario = new HashMap<>();

// Agrega elementos al diccionario
diccionario.put("Juan", 25);
diccionario.put("María", 30);
diccionario.put("Carlos", 28);

// Recupera el valor asociado a una clave
int edadMaria = diccionario.get("María");
System.out.println("La edad de María es: " + edadMaria);

// Comprueba si una clave está presente
boolean contieneJuan = diccionario.containsKey("Juan");
System.out.println("¿Contiene a Juan? " + contieneJuan);

// Elimina un elemento
diccionario.remove("Carlos");
```

```java
// Itera sobre las claves y valores
for (String nombre : diccionario.keySet()) {
    int edad = diccionario.get(nombre);
    System.out.println(nombre + " tiene " + edad + " años.");
}
```

Como mencionamos anteriormente, una colección puede tener una cantidad arbitraria de objetos, por lo tanto el mapa también puede tener asociado a cada clave un objeto.

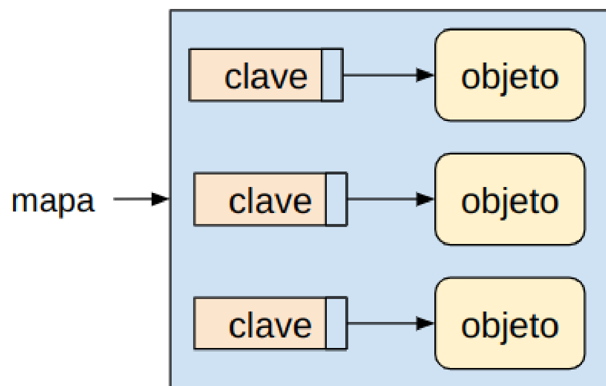


Fig.41 - mapa de objetos.

Veamos un ejemplo algo más completo.

Necesitamos almacenar alumnos, y luego permitir búsqueda, a través del legajo, para obtener los datos de un alumno en particular:

```java
Map<Integer, Alumno> diccionarioAlumnos = new HashMap();

// Cargar alumnos
diccionarioAlumnos.put(101, new Alumno("Juan", 101, 3.5));
diccionarioAlumnos.put(102, new Alumno("María", 102, 4.2));
diccionarioAlumnos.put(103, new Alumno("Pedro", 103, 3.9));
diccionarioAlumnos.put(104, new Alumno("Ana", 104, 4.8));
```

```java
diccionarioAlumnos.put(105, new Alumno("Carlos", 105, 2.7));

Scanner scanner = new Scanner(System.in);

System.out.print("Ingresa el legajo para buscar (-1 sale): ");
int legajo = Integer.parseInt(scanner.nextLine());

while (legajo != -1) {


   Alumno alumno = diccionarioAlumnos.get(legajo);

   if (alumno != null) {
       System.out.println("Información del alumno:");
       System.out.println("Nombre: " + alumno.getNombre());
       System.out.println("Legajo: " + alumno.getLegajo());
       System.out.println("Nota: " + alumno.getNotaCursada());
   } else {
       System.out.println("Alumno no encontrado.");
   }

   System.out.print("Ingresa el legajo para buscar(-1 sale): ");
   legajo = Integer.parseInt(scanner.nextLine());

}
```

Tipo abstracto de dato

El Tipo de Dato Abstracto (TDA) es un concepto fundamental en la programación, inclusive en el paradigma que estamos desarrollando, en el orientado a objetos. Permite en esencia crear estructuras de datos con características y funcionalidades específicas, encapsulando la implementación interna y exponiendo solo una interfaz pública para que podamos interactuar.

En nuestro contexto, los TDA se implementan utilizando clases, con sus atributos, métodos y relaciones.

Tenemos que pensar a los TDA como un tipo de dato específico, cuya representación interna no es accesible, y exportan o exponen un conjunto de operaciones para su manipulación.

Los TDAs y los objetos están estrechamente relacionados. Se utilizan para definir la estructura y el comportamiento de los objetos, y los objetos se crean a partir de clases que pueden incluir también a los TDA como atributos.

Veamos un ejemplo para ver estos conceptos de forma más clara.

Supongamos que tenemos que emular la forma en que se resuelven los retornos de cada llamada a los métodos, una especie de pila de llamadas.

Cada vez que se invoca a un método, se guarda la dirección de memoria a donde debería volver una vez finalizado para continuar la ejecución.

Entonces, nos interesa poder agregar direcciones de memoria, supongamos que tenemos tres llamadas:

Agregar la dirección 100000

Agregar la dirección 105000

Agregar la dirección 200100

Luego al ir terminando la ejecución de cada método deberíamos poder volver a la dirección que tenemos guardada. Como podemos ver, las direcciones deben obtenerse justamente en el orden inverso en el que han sido agregadas:

Volvemos a: 200100

Volvemos a: 105000

Volvemos a: 100000

Este problema lo podemos resolver con arreglos o listas, como mostramos a continuación:

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

    //
    int[] direcciones = new int[100];
    int pos = 0;
```

```java
    // Agregar las direcciones de memoria
    direcciones[pos] = 100000;
    pos++;
    direcciones[pos] = 105000;
    pos++;
    direcciones[pos] = 200100;
    pos++;


    //
    while( pos > 0 ){
        int direccion = direcciones[pos-1];
        System.out.println("Volvemos a: "+direccion);
        pos--;
    }
}
```

El problema está resuelto pero presenta algunas situaciones que terminan siendo desfavorables, por ejemplo, ¿qué pasa si tuviéramos más de 100 direcciones?, cómo puedo hacer para que el desarrollador no tenga que estar pensando en el manejo del arreglo, por ejemplo incrementar y decrementar adecuadamente el índice, y que pueda concentrarse en el problema en sí, este tipo de problema ¿será la última vez que que aparezca? ¿cómo puedo volver a utilizar la idea con la que soluciono el problema?

Si pudiéramos disponer de un tipo de dato que tenga este comportamiento, podríamos pensar en simplemente utilizarlo cada vez que se nos presente este tipo de problemas, otorgando la posibilidad al desarrollador de que se concentre en la solución sin tanto nivel de detalle y además reutilizando cada vez que sea necesario.

Lo que acabamos de describir es justamente lo que proponen los TDA, desarrollamos por única vez o utilizamos uno ya implementado como soporte a nuestras soluciones.

Al usar un TDA ya no tengo que preocuparme por el detalle, en este caso del manejo del arreglo, solo pensar en cómo llegar a la solución utilizando el TDA seleccionado.

Un tipo de dato abstracto que se adapta perfectamente a esta situación es el denominado “pila”.

Una pila es una estructura de datos lineal que sigue el principio de "último en entrar, primero en salir" (LIFO - Last In, First Out). En una pila, los elementos se agregan y eliminan

sólo desde un extremo, conocido como la cima de la pila. Esto significa que el último elemento que se agrega a la pila será el primero en ser extraído.

La pila, como tipo de dato abstracto, provee operaciones estándares:

- push: agrega un elemento a la cima de la pila.
- pop: elimina y retorna el último elemento agregado a la pila.
- peek: obtiene el último elemento agregado a la pila, pero no lo elimina de la estructura.
- isEmpty: retorna si la pila está vacía.
- Size: retorna el número de elementos que contiene la estructura.

Veamos entonces cómo podemos resolver ahora el problema a través del uso de este TDA.

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

    //
    Pila direcciones = new Pila();

    // Agregar las direcciones de memoria
    direcciones.push(100000);
    direcciones.push(105000);
    direcciones.push(200100);

    //
    while(!direcciones.isEmpty()){
        int direccion = direcciones.pop();
        System.out.println("Volvemos a: "+direccion);
    }
}
```

Como podemos apreciar, la forma de resolver este problema con una "pila" es mucho más legible y elegante, aunque también es cierto que tendremos que saber el comportamiento intrínseco de la pila sino jamás se nos hubiese ocurrido este tipo de solución. Por ahora no nos concentremos tanto en esto, sino más bien en la utilidad del TDA.

Como el código lo expone, no necesitamos saber cómo se guardan los elementos, ni cómo es posible que se retorne el último que ingresó. Sólo necesitamos saber el comportamiento y las operaciones que nos aporta.

Para terminar de comprender a los TDA y poder mencionar una ventaja significativa que tendremos, partamos de una posible implementación de nuestro TDAPila:

En este caso la implementación de la pila será estática, esto implica que usaremos un arreglo para guardar los elementos. Al ser estática tendremos que determinar una cantidad máxima de elementos, por ejemplo 100.

```
public class Pila {

   private final int MAX=100;
   private int[] elementos;
   private int pos;

   public Pila() {
       elementos = new int[MAX];
       pos = 0;
   }

   public void push(int elemento) {
       elementos[pos] = elemento;
       pos++;
   }

   public int pop() {
       pos--;
       return elementos[pos];
   }

   public int peek() {
       return elementos[pos - 1];
   }
```

```
  public boolean isEmpty() {
        return pos == 0;
    }


}
```

Después de un tiempo nos damos cuenta que los 100 elementos nos quedan cortos y como en el camino hemos incorporado más herramientas decidimos cambiar la implementación de la pila por una estructura dinámica que no tenga esta limitación. En vez de usar un arreglo usaremos una lista:

```
public class Pila {

    private List<Integer> elementos;

    public Pila() {
        elementos = new ArrayList<>();
    }

    public void push(int elemento) {
        elementos.add(elemento);
    }

    public int pop() {
        return elementos.remove(elementos.size() - 1);
    }

    public int peek() {
        return elementos.get(elementos.size() - 1);
    }
```

```java
    public boolean isEmpty() {
        return elementos.isEmpty();
    }
}
```

De esta manera, podemos comenzar a entender uno de los mayores beneficios que nos aporta el uso de los TDA.

Podemos arreglar algún problema que presente nuestro TDA sin la necesidad de tocar nada en nuestras soluciones, inclusive como acabamos de ver, podemos cambiar la implementación de la pila entera sin alterar nada de nuestra solución. Esto nos dará una gran flexibilidad a la hora de desarrollar.

Es importante darnos cuenta que los TDA también pueden utilizar otros TDA, en este caso la pila utiliza una lista, que en esencia es un TDA. La lista que usamos cuenta con una implementación del tipo ArrayList pero podríamos eventualmente utilizar otra como una LinkedList.

Formalicemos las características que los TDA nos aportan:

- Encapsulación: la implementación interna de las operaciones y datos del TDA se mantiene oculta. Sólo se expone una interfaz pública con la cual interactuar.
- Abstracción: quienes usan el TDA se focalizan en las características esenciales y funcionalidades que ofrece, descartando los detalles de implementación.
- Modularidad: se pueden crear TDA que encapsulan funcionalidades específicas y se pueden utilizar en diferentes partes del programa sin duplicar código.
- Reutilización: los TDA fomentan la reutilización de código, ya que se pueden crear componentes independientes que se pueden utilizar en diferentes partes del programa.
- Robustez: si se modifica la implementación interna del TDA y no se afecta la interfaz pública, se garantiza la compatibilidad con el código existente.
- Seguridad: el control de acceso a través de la encapsulación permite proteger los datos sensibles y evitar accesos no autorizados.

Finalmente, una situación que surge al desarrollar los TDA es sobre cómo gestionar la validación de los datos y sus operaciones. Por ejemplo, en el TDA pila, ¿quién se encarga de validar que se puede completar la operación “pop” ?, operación que en una situación

normal no parecería que se requiere de alguna validación pero si el contenedor no tuviera elementos y se ejecuta la operación “pop” ¿qué pasaría? ¿qué deberías hacer?

Esto no tiene una respuesta única, de hecho hay distintos autores que dicen que la res-ponsabilidad de validar estas situaciones es del TDA, o sea que la misma operación “pop” debería validar que es posible llevar adelante la operación de extraer el elemento y ante un caso de imposibilidad se debe desencadenar un error.

Por otro lado, también están los autores que aseguran que la tarea de validar si es posible realizar la operación es responsabilidad de quien usa el TDA a través de sus operaciones, y que esta forma brinda flexibilidad.

Entonces, se deben considerar ambas posibilidades y elegir la que a uno le parezca la mejor opción.

Ejercicios

1. Crear una clase "Persona" con sus atributos nombre, edad y género. Definir un constructor que permita inicializar estos atributos al crear un objeto. Implementar un método para imprimir la información de la persona.

2. Crear una clase "Rectángulo" con atributos privados para la base y la altura. Utiliza métodos de acceso (getters y setters) para acceder y modificar estos atributos de manera controlada. Definir métodos para calcular el área y el perímetro del rectángulo.

3. Crear una clase "Estudiante" con atributos como nombre y 5 calificaciones. Definir un constructor para inicializar el nombre y opcionalmente las 5 calificaciones. Implementar métodos para calcular el promedio de calificaciones, establecer las calificaciones y mostrar la información del estudiante.

4. Crear una clase "Libro" con atributos como título, autor y año de publicación. Implementar métodos para mostrar la información del libro de forma legible . Probar la clase creando 2 libros c/u con sus valores respectivos y finalmente mostrar por pantalla si son iguales.

5. Crear una clase "Cuenta Bancaria" con atributos para el nombre del titular, el número de cuenta y el saldo. Implementar métodos para depositar, retirar y consultar saldo.

6. Crea una clase "Fecha" con atributos para el día, el mes y el año.

   Definir un método para imprimir la fecha en formato DD/MM/AAAA y un método para incrementar los días

   Ej

   - fecha = 10/02/2023

     - incrementar(10)

     - imprimir

       → 20/02/2023

7. Agregar al ejercicio anterior la validación de las fechas al momento de establecer, en caso que no sea válida mostrar un mensaje que no se puede realizar la operación (puede usar valores por defecto para fechas no válidas).

8. Crear una clase "Mascota" con atributos nombre, especie y edad. Luego, crea una clase

"Dueño" que tenga la posibilidad de tener una mascota. Se debe permitir agregar y eliminar una mascota del dueño y mostrar la información completa del dueño.

[-] listas

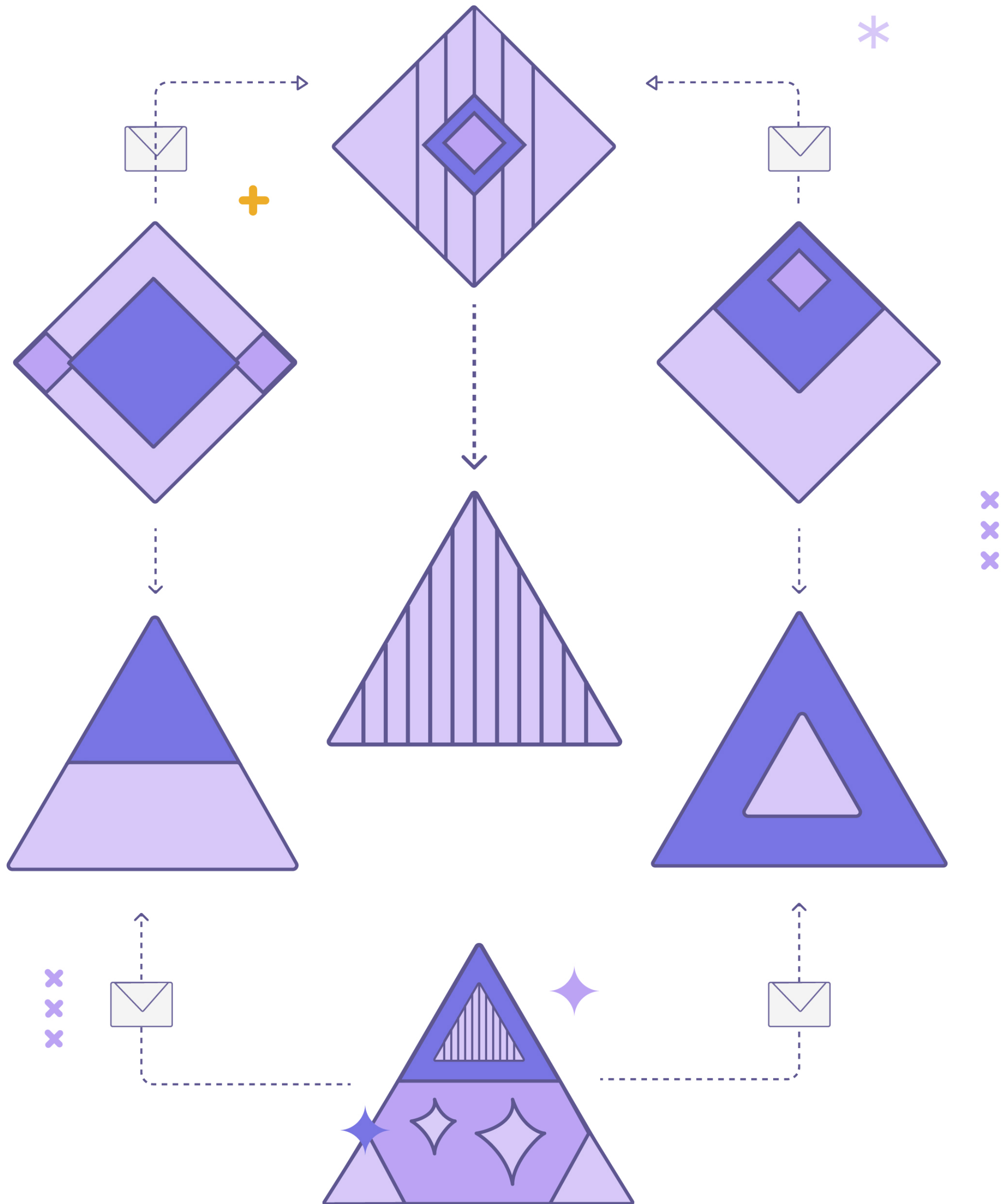
1. Calcular la suma de todos los elementos de una lista de números enteros.
2. Encontrar el número más grande en una lista de números enteros.
3. Contar cuántos números pares hay en una lista de números enteros.
4. Crear una lista de palabras y mostrar la longitud de cada palabra.
5. Buscar un elemento específico en una lista de palabras y mostrar su posición si se encuentra.
6. Invertir el orden de una lista.
7. Eliminar elementos duplicados de una lista.
8. Ordenar una lista de números enteros de menor a mayor.
9. Crear una lista de personas y mostrar por pantalla todas aquellas que tengan más de 30 años de edad.
10. Crear una lista de vehículos y mostrar por pantalla los vehículos ordenados. ¿Por qué atributo se han ordenado?.

[-] mapas

1. Crear un diccionario que traduzca palabras del español al inglés.
2. Contar cuántas veces aparece cada elemento en una lista y almacenar los resultados en un mapa.
3. Eliminar un elemento específico de un mapa.
4. Unir Mapas: Unir dos mapas en uno solo.
5. sin valores repetidos
6. con valores repetidos

Capítulo 10

Herencia y polimorfismo



Capítulo 10

Herencia y polimorfismo

Objetivo

En este capítulo exploramos en profundidad dos pilares fundamentales de la programación orientada a objetos, la herencia y el polimorfismo. Aprenderemos a construir jerarquías de clases, aprovechar la reutilización de código y aplicar conceptos avanzados como el polimorfismo para crear sistemas flexibles, mantenibles y escalables.

Al finalizar este capítulo, estarás capacitado para utilizar la herencia y el polimorfismo de forma eficaz en tus soluciones. Habrás adquirido una comprensión profunda de lo que implica la herencia, como así también sobre la reutilización de código. Además, estarás en condiciones de diseñar e implementar jerarquías de clases bien definidas, aprovechar el polimorfismo para escribir código más genérico y flexible, y crear sistemas mantenibles y escalables a través del uso de interfaces.

Herencia

Es un concepto fundamental de la Programación Orientada a Objetos que permite crear nuevas clases a partir de otras existentes, tomando como base sus características y comportamiento.

La herencia es la capacidad de una clase de heredar propiedades y métodos de una clase padre, lo que permite reutilizar código y hacer que las clases sean más fáciles de entender y mantener.

Interpretemos este concepto a través de la analogía que puede darse en la vida entre un padre y un hijo.

Imaginemos a Eugenio, un hombre de 35 años con pelo castaño y ojos marrones, y a su hijo Julián, un niño de 10 años con características físicas muy similares, pelo castaño y ojos marrones.

Julián ha heredado de Eugenio muchas características, tanto físicas como de personalidad y habilidades. Físicamente, ambos comparten el color de pelo y el color de sus ojos, además de tener una altura similar. En cuanto a la personalidad, Julián ha heredado la inteligencia, el sentido del humor y la amabilidad de su padre. Incluso, ha desarrollado

algunas de las habilidades de Eugenio, como la habilidad para tocar la guitarra y el gusto por la lectura.

Sin embargo, Julián también tiene características únicas que lo diferencian de su padre. Su color de piel, por ejemplo, es más claro que el de su padre. Además, tiene intereses distintos, como el gusto por jugar al fútbol, mientras que Eugenio prefiere jugar al squash. Finalmente, Julián ha desarrollado una habilidad que Eugenio no posee: la habilidad para patinar.

Esta analogía, que ilustra cómo funciona la herencia en la vida real, también aplica al mecanismo que el paradigma orientado a objetos nos provee. Una clase hija, como Julián, hereda las características de una clase padre, como Eugenio. De esta manera, la clase hija puede reutilizar el código de la clase padre, a la vez que tiene sus propias características únicas.

La herencia puede ser simple o múltiple, esto significa poder heredar de una clase o más de una. Java sólo permite a las clases la herencia simple, o sea, heredar de una sóla clase. La herencia múltiple en java queda sólo para las interfaces, un concepto que luego desarrollaremos.

[-] subclases y superclases

Las clases se organizan en una jerarquía, donde las del nivel superior se denominan "clases base" o "superclases", y las del nivel inferior se denominan "clases derivadas", "subclases" o incluso “clases hijas”.

La herencia permite a una subclase heredar los atributos y métodos de su superclase, permitiendo además añadir nuevos atributos y métodos que por supuesto serán propios.

El siguiente gráfico muestra la clase base -o superclase- Animal y sus clases derivadas -o subclases- Perro y Gato.

Como puede apreciarse en la figura, la clase Animal tiene atributos propios como el “nombre” y la “edad”. Las clases derivadas -clases hijas- tendrán estos mismos atributos aunque no se hayan especificado.

Lo mismo ocurre con los métodos, tanto la clase Perro como la clase Gato tendrán de forma implícita los métodos “comer()” y “dormir()” definidos en la superclase Animal.

Que las clases tengan los métodos implícitos significa que los objetos de estas clases derivadas podrán enviar estos mensajes. Del mismo modo, los métodos propios de las clases derivadas podrían utilizar los atributos de la clase padre.

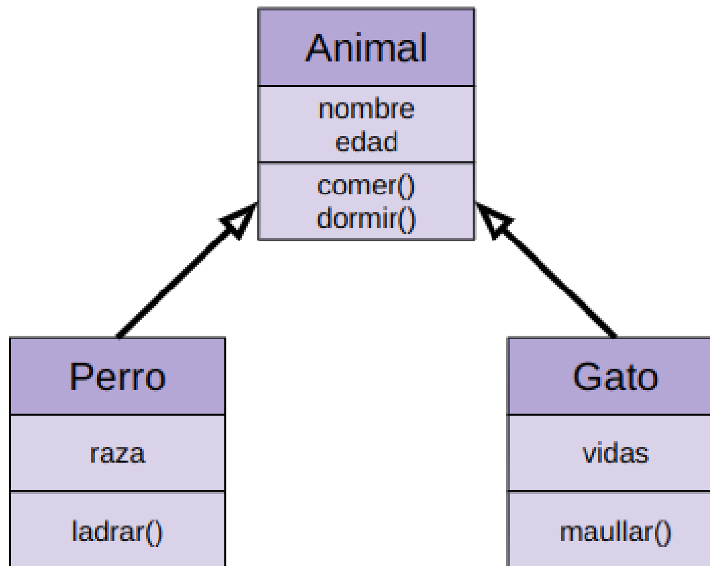


Fig.42 - jerarquía de clases.

Se puede llamar superclase a una clase siempre y cuando exista al menos una clase que herede de ella. Por lo tanto, la clase Animal es la superclase de Perro o de Gato, y estas no son superclase de nadie.

> La relación "es un" se utiliza para indicar una relación de tipo entre una clase hija y una clase padre.

Veamos como se especifican estas características en el código.

La clase Animal tendrá la siguiente implementación:

```
public class Animal {

    //
    private String nombre;
```

```
//
private int edad;

//
public Animal(String nombre, int edad) {
    this.nombre = nombre;
    this.edad = edad;
}

//
public int getEdad() {
    return edad;
}

//
public void setEdad(int edad) {
    this.edad = edad;
}

//
public String getNombre() {
    return nombre;
}

//
public void setNombre(String nombre) {
    this.nombre = nombre;
}

//
public void comer(){
    System.out.println("comiendo...");
}
```

```java
    //
    public void dormir(){
        System.out.println("durmiendo...");
    }
}
```

Ahora veremos cómo extender la clase Animal para crear la clase hija derivada “Perro”. Se utilizará la palabra clave **extends** y a continuación la clase de la cual extenderemos o heredaremos:

```java
public class Perro extends Animal{

    //
    private String raza;

    //
    public Perro(String nombre, int edad, String raza) {
        super(nombre, edad);
        this.raza = raza;
    }

  //
    public String getRaza() {
        return raza;
    }

    //
    public void setRaza(String raza) {
        this.raza = raza;
    }
```

```
    //
    public void ladrar(){
        System.out.println("gua gau...");
    }
}
```

Del mismo modo crearemos la clase “Gato”.

Esta clase también hereda todos los atributos y métodos de la clase “Animal”:

```
public class Gato extends Animal{

    //
    private int vidas;

    //
    public Gato(String nombre, int edad, int vidas) {
        //
        super(nombre,edad);
        //
        this.vidas = this.vidas;
    }

    //
   public int getVidas() {
        return vidas;
    }

    //
    public void setVidas(int vidas) {
        this.vidas = vidas;
    }
```

```
    //
    public void maullar(){
        System.out.println("miauuuuuuu");
    }
}
```

Y finalmente un clase “Programa” que nos permitirá crear y probar los objetos de las clases creadas, de “Perro” y “Gato”:

```
public class Programa {

    //
    public static void main(String[] args) {

    Perro miPerro = new Perro("Charo", 10, "Labrador");
    Gato miGato = new Gato("Viktor", 3, 7);

    //
    miPerro.comer();
    miPerro.ladrar();

    //
    miGato.dormir();
    miGato.maullar();
    }
}
```

Como se puede observar, los objetos pueden enviar el mensaje “comer()” y “dormir()” aun cuando ellos no se encuentren definidos en su clase directamente.

Si nos fijamos en el código del “Programa” podemos ver que hemos utilizado la clase “Perro” en ambos lugares de la asignación:

“*Perro miPerro = new Perro("Charo", 10, "Labrador");*”

Esto significa básicamente que el objeto miPerro será un Perro. Pero el lenguaje nos provee la posibilidad de ir un poquito más allá y definir, en la creación, que un objeto perro es un Animal:

```
Animal miPerro = new Perro("Charo", 10, "Labrador");
```

Este tipo de declaraciones nos ofrece varias ventajas, aunque hay que tener cuidado porque también tendremos alguna que otra desventaja.

Por ejemplo, el objeto miPerro, el que declaramos de tipo “Animal” no podrá enviar mensajes directamente a sus métodos ladrar u obtenerRaza, porque estamos diciendo que es un Animal.

Esta situación se puede corregir aunque lo conveniente es no seguir por esta senda, cuando se vean los conceptos de clases abstractas, interfaces y polimorfismo podremos solucionar de manera elegante esta situación.

De cualquier manera mostramos el código que podríamos utilizar si llegamos a estar en esta solución:

```
public class Programa {

//
public static void main(String[] args) {

   Animal miPerro = new Perro("Charo", 10, "Labrador");
   Animal miGato = new Gato("Viktor", 3, 7);

   //
   miPerro.comer();
   ((Perro)miPerro).ladrar();

   //
   miGato.dormir();
   ((Gato)miGato).maullar();
}
```

Este operación se llama "Cast", se encarga de convertir un dato en un tipo de dato distinto, en este caso al objeto del tipo Animal lo convierte en el tipo Perro o Gato según se explicite.

En la disciplina a esto se lo suele llamar "bad smell".

El término "huele mal" (bad smell) habitualmente lo utilizamos para referimos a ciertas prácticas de programación que, si bien no generan errores, son generadoras de un código potencialmente problemático o difícil de mantener

Otra forma de ver la utilidad que los objetos sean Animales sería la siguiente. Consideremos la posibilidad almacenar animales en un contenedor, ya sean perros o gatos, para luego enviar mensajes del tipo "comer" y "dormir" a todos los objetos contenidos:

```
Animal miPerro = new Perro("Charo", 10, "Labrador");
Animal miGato = new Gato("Viktor", 3, 7);

//
List<Animal> misMascotas = new ArrayList();

//
misMascotas.add(miPerro);
misMascotas.add(miGato);

//
for (Animal animal : misMascotas) {
   //
   animal.comer();
   animal.dormir();
}
```

De esta forma hemos creados dos objetos, aunque podrían ser una cantidad distinta, y los hemos almacenado en un contenedor del mismo tipo. Luego los recorremos, tratandolos de la misma manera, y enviamos los mensajes "comer" y "dormir".

Si no hubiésemos utilizado este tipo de construcción, tendríamos que tener una lista para los perros y otra lista para los gatos, y finalmente el recorrido lo tendríamos que hacer sobre ambas.

[-] sobrescritura de métodos

La sobrescritura de métodos, conocida como "method overriding", permite a una subclase proporcionar su propia implementación, cambiando el comportamiento heredado de la clase padre.

Partiendo del ejemplo anterior, la clase Animal tiene definido el método “dormir()”. La clase Perro, que hereda de Animal, podría redefinir el método para cambiar su comportamiento, o dicho de otro modo, para especializar su comportamiento.

Para sobrescribir un método, la subclase debe definir un método que tenga el mismo nombre, los mismos parámetros -cantidad y tipo- y el mismo tipo de retorno que el método heredado.

La firma del método (nombre, parámetros y tipo de retorno) debe ser idéntica para que se considere una sobrescritura.

Es importante destacar que la sobrescritura de métodos sólo se aplica a los métodos definidos en una clase padre que sean accesibles y modificables.

Los métodos privados o los métodos públicos marcados como “final” no pueden ser sobreescritos. Además, la sobrescritura de métodos sólo afecta al comportamiento de los objetos de la subclase y no al de los objetos de la clase padre o de otras subclases.

Veamos un ejemplo que cambia el comportamiento del método “mostrarInfo” definido en la clase padre “Persona”. Haremos uso de la sobreescritura de métodos en las clases derivadas.

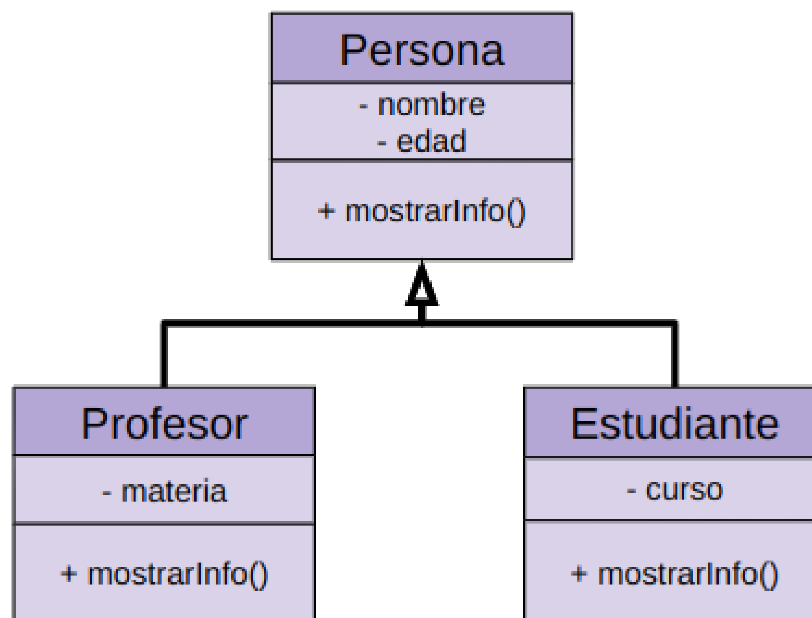


Fig.43 - sobreescritura de métodos.

La clase Persona, además de los atributos nombre y edad, tendrá el método "mostrarInfo()" que sirve para imprimir por pantalla el nombre y la edad:

```
public class Persona {

   private String nombre;
   private int edad;

   public Persona(String nombre, int edad) {
       this.nombre = nombre;
       this.edad = edad;
   }

   public void mostrarInfo() {
       System.out.println("Nombre: " + nombre);
       System.out.println("Edad: " + edad);
   }
}
```

La clase Profesor, hereda de Persona, por lo tanto tendrá todos los atributos y métodos accesibles de su superclase y, como puede observarse, podrá redefinir el comportamiento del método "mostrarInfo()". El nuevo comportamiento será imprimir la info del objeto "nombre y edad" y luego mostrar la materia.

```
public class Profesor extends Persona{
   private String materia;

   public Profesor(String nombre, int edad, String mat) {
       super(nombre, edad);
       this.materia = mat;
   }

   @Override
   public void mostrarInfo() {
```

```java
        super.mostrarInfo();
        System.out.println("Materia: " + materia);
    }
}
```

En el método se utiliza la palabra reservada **super** para referirse al método de la superclase. Esta palabra especial también puede usarse para invocar al constructor o atributos de la superclase.

La clase Estudiante hará lo propio con el comportamiento, en este caso además del nombre y la edad mostrará el curso:

```java
public class Estudiante extends Persona{

    private String curso;

    public Estudiante(String nom, int edad, String curso) {
        super(nom, edad);
        this.curso = curso;
    }

    @Override
    public void mostrarInfo() {
        super.mostrarInfo();
        System.out.println("Curso: " + curso);
    }
}
```

La sobreescritura de métodos facilita la especialización y la extensión de la funcionalidad, permitiendo que las clases derivadas -las hijas- se adapten a situaciones específicas.

Clase abstracta

Como dijimos, las clases se utilizan para modelar entidades del mundo real, pero no todas las entidades son concretas.

Este fue justamente el impulso principal para la creación de las clases abstractas. La necesidad de representar conceptos abstractos en el código.

En nuestro ejemplo, la clase "Animal" puede representar el concepto general de un animal, sin especificar las características o comportamientos específicos de ningún animal en particular. A partir de este concepto abstracto podemos especializar el comportamiento de las entidades concretas, o sea, de un perro o un gato.

En general, no tiene demasiado sentido tener un objeto que sea un animal, porque en realidad es sólo una clasificación. Sí tiene sentido tener un objeto que sea un perro o un gato, que comparten conceptualmente lo que son, Animales.

Para evitar crear instancias de la clase Animal hay que especificar que es una clase pero Abstracta. Las clases Perro y Gato serán concretas, con un comportamiento específico, y heredan los atributos que tienen los Animales, los atributos definidos en la clase abstracta.

Una clase abstracta es una clase que no puede ser instanciada directamente, sino que sirve como modelo o plantilla para que otras clases puedan heredar de ella.

La clase abstracta puede definir atributos y métodos. Algunos o todos de los métodos pueden no tener una implementación concreta, lo que implica que deberán ser implementados por las clases que heredan de ella. En otras palabras, podremos tener métodos abstractos y eventualmente métodos concretos en una clase abstracta.

El propósito de este tipo de clases es proporcionar una base común de características obligando a quien herede, que especifique o implemente su o sus características específicas.

Sintetizando lo anterior, algunas de las características y propósitos de una clase abstracta son:

- No se pueden crear instancias de una clase abstracta, sino que solo pueden ser utilizadas como superclases para otras clases (puede pensarse como una plantilla).
- La clase derivada de una clase abstracta, debe implementar todos los métodos abstractos que se definen en la clase abstracta, caso contrario deberá ser también abstracta.
- Una clase abstracta puede contener métodos no abstractos y estos métodos deben tener implementaciones concretas, que eventualmente también pueden ser sobreescritos (salvo que se hayan definido como final).

- Las clases abstractas son útiles cuando se quiere crear una jerarquía de clases y garantizar que cada clase derivada tenga la funcionalidad básica necesaria, pero también permitir que tengan su propia implementación específica.
- Las clases abstractas también pueden proporcionar una capa de abstracción adicional para que los programadores no tengan que preocuparse por los detalles internos de una clase concreta.

Haremos la clase Animal abstracta y definiremos el método dormir como abstracto para que las clases derivadas tengan que realizar su propia implementación:

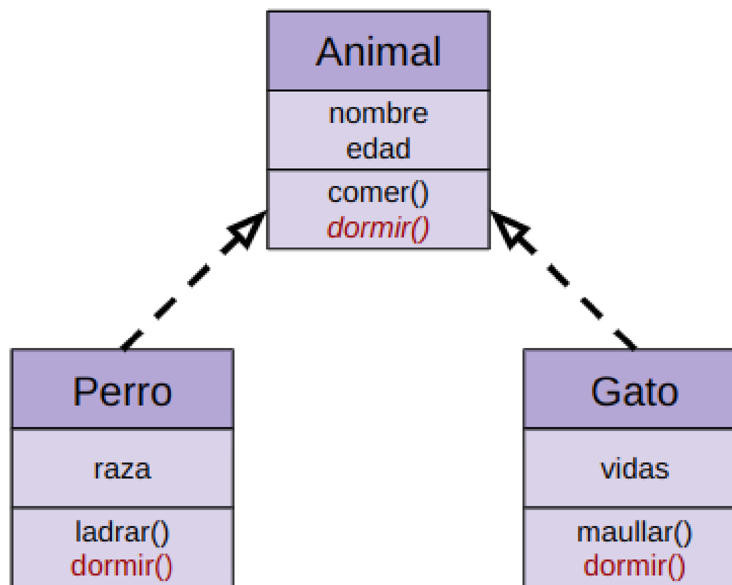


Fig.44 - clase abstracta

```
public abstract class Animal {

    //
    private String nombre;

    //
    private int edad;
```

```
    //
    public Animal(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    //
    … todos los métodos

    //
    public abstract void dormir();
}
```

Observar que el método dormir de la clase Animal no tiene cuerpo, no tiene una implementación concreta, solamente se encuentra declarada y serán las clases derivadas las responsables de aportar el comportamiento.

```
public class Perro extends Animal {

    //
    private String raza;

    //
    public Perro(String nom, int edad, String raza) {
        super(nom, edad);
        this.raza = raza;
    }

    //
    public void ladrar(){
        System.out.println("gua gau...");
    }
```

```
    //
    @Override
    public void dormir() {
        System.out.println("Durmiendo en alerta!”);
    }
}
```

La clase Gato, al heredar de Animal, también tendrá que implementar el método:

```
public class Gato extends Animal {

    //
    private int vidas;

    //
    public Gato(String nombre, int edad, int vidas) {
        //
        super(nombre,edad);
        //
        this.vidas = this.vidas;
    }

    //
    public void maullar(){
        System.out.println("miauuuuuuu");
    }

    @Override
    public void dormir() {
        System.out.println("Durmiendo... mucho....");
    }
}
```

Veamos un ejemplo de cómo podemos utilizar ahora las clases:

```
public class Programa {

    //
    public static void main(String[] args) {

        Perro miPerro = new Perro("Charo", 10, "Ladra..");
        Gato miGato   = new Gato("Viktor", 3, 7);

        //
        miPerro.ladrar();
        miPerro.dormir();

        //
        miGato.maullar();
        miGato.dormir();
    }
}
```

Hasta acá no ha cambiado demasiado en cuanto a cómo se invoca el método dormir()". Pero si volvemos al ejemplo que tiene ambos animales en un contenedor podremos visualizar algo más:

```
Animal miPerro = new Perro("Charo", 10, "Labrador");
Animal miGato = new Gato("Viktor", 3, 7);

//
List<Animal> misMascotas = new ArrayList();

//
misMascotas.add(miPerro);
misMascotas.add(miGato);
```

```java
//
for (Animal animal : misMascotas) {
   //
   animal.dormir();
}
```

Ahora, se envía el mensaje “dormir()” a cada objeto y cada uno responderá como tiene especificado su método. Se puede comprobar, al ejecutar el código, que saldŕa por pantalla:

```
Durmiendo, pero en alerta..

Durmiendo... mucho....
```

Por más que se utilizara el mismo método, de cada objeto del tipo animal, se ejecutó el método adecuado, el del perro y el del gato. Este es el principio de funcionamiento de algo que denominaremos polimorfismo y desarrollaremos en breve.

Interfaz

A partir de lo que se puede hacer con una clase abstracta surge la idea de proporcionar un mecanismo más fuerte que permita establecer una especie de contrato y otorgar responsabilidad a quien quiera considerarse parte de una familia de clases.

Una interfaz es una colección de métodos abstractos, como vimos anteriormente sin implementación y adicionalmente constantes. Se utilizan para definir en conjunto el comportamiento que una clase debe implementar. Entonces, una interfaz es un contrato que una clase debe cumplir si quiere ser considerada como parte de la familia.

La principal diferencia entre una interfaz y una clase abstracta es que en una interfaz todos los métodos deben ser declarados como abstractos, lo que significa que no se proporciona una implementación predeterminada en ningún caso.

En el siguiente ejemplo mostraremos gráficamente la interfaz llamada FiguraGeometrica. Cada clase que quiera ser considerada una figura geométrica deberá aceptar el contrato e implementar el comportamiento requerido, esto es, proporcionar su propio código a todos los métodos abstractos que define la interfaz.

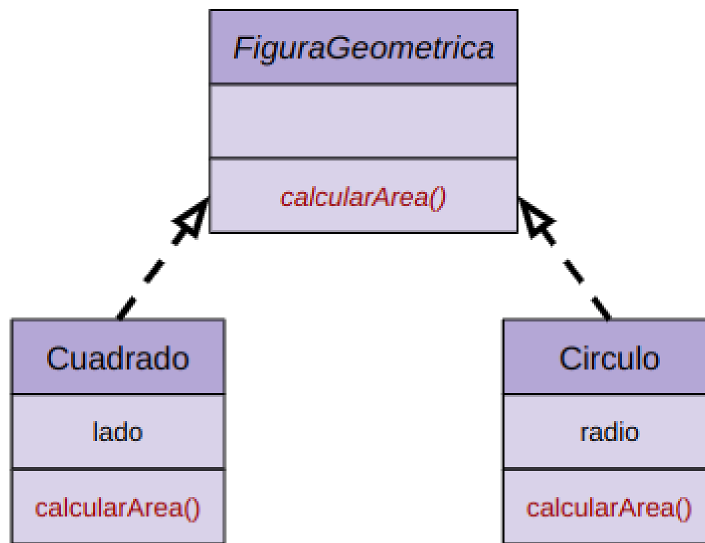


Fig.45 - interfaz.

El código para establecer la interfaz es el siguiente. Darse cuenta que no lleva la palabra reservada "class" y además no hace falta definir a los métodos como "abstract" dado que obligatoriamente serán abstractos.

```
public interface FiguraGeometrica {

    //
    double calcularArea();

}
```

Los métodos de la interfaz no llevan modificadores de acceso, son todos del tipo "public".

Cuando otra clase pretende considerarse, en este caso una figura geométrica, deberá aceptar el contrato que establece la interfaz y proporcionar el código al o los métodos definidos.

En este caso tenemos que implementar el método "calcularArea()". Este método será re-lativo a la clase de figura geométrica que estemos creando y deberá retornar el área de la figura. Por supuesto que podríamos establecer todos los métodos que se necesiten para

que una clase sea considerada como una figura geométrica.

Si la intención es crear un cuadrado entonces necesitaremos agregar los atributos necesarios para poder calcular el área.

Veamos cómo resuelve esto la clase Cuadrado:

```
public class Cuadrado implements FiguraGeometrica{

    //
    private double lado;

    //
    public Cuadrado(double lado) {
        this.lado = lado;
    }

    //
    @Override
    public double calcularArea() {
        return lado * lado;
    }
}
```

La clase Cuadrado establece un atributo llamado "lado", el cual es necesario para poder aportar el comportamiento que la interfaz requiere. Para calcular el área, en este caso, se quiere multiplicar "lado x lado".

El método calcularArea está marcado como sobreescrito "Override" dado que es exactamente lo que está sucediendo, se está especializando el comportamiento de cómo se calcula el área de esta figura geométrica.

> Las interfaces establecen un contrato entre piezas de código, mientras que las clases abstractas sólo proporcionan una especie de plantilla.

Veamos otro ejemplo de una figura que quiera considerarse una figura geométrica. En este caso un círculo:

```
public class Circulo implements FiguraGeometrica{

   //
   private double radio;

   //
   public Circulo(double radio) {
       this.radio = radio;
   }

   //
   @Override
   public double calcularArea() {
       return Math.PI * radio * radio;
   }
}
```

La clase Circulo necesita saber cual es el radio para calcular su área.

Veamos ambas figuras en funcionamiento, se crean las figuras geométricas, cada uno a partir de los datos que se establecieron y luego se envía el mensaje "calcularArea()" para mostrarlo en pantalla:

```
public class Programa {

   //
   public static void main(String[] args) {

       //
       FiguraGeometrica cuadrado = new Cuadrado(5);
```

```
        //
        FiguraGeometrica circulo = new Circulo(2.5);

        System.out.println("Área del cuadrado: " +
          cuadrado.calcularArea());
        System.out.println("Área del círculo: " +
 circulo.calcularArea());
    }
}
```

Nuevamente se nos presenta esta situación en la que los objetos envían el mismo mensaje y cada uno responde de manera diferente, aun cuando sean ambos del tipo FiguraGeometrica. Este comportamiento, como ya hemos mencionado, es posible debido al polimorfismo.

Ahora que comprendemos el concepto de interfaz podemos ir un poco más allá y ver cómo crear el comportamiento específico para ordenar la lista a través de la especificación de cómo se deben comparar los objetos.

Veremos el orden natural y el orden según diferentes criterios. Para esto es necesario profundizar las interfaces que provee el lenguaje, Comparable y Comparator.

[-] comparable

Supongamos que por un lado tenemos la clase Persona con sus atributos nombre y edad, y por otro tenemos una lista de objetos de esa clase. Queremos ordenar la lista a partir del nombre de cada objeto, la pregunta que surge es ¿cómo hacemos?. Hasta ahora hemos visto cómo ordenar arreglos, con las listas qué hacemos?

La interfaz Comparable, es introducida por el lenguaje para facilitar justamente esta situa-ción, la de ordenar objetos de forma natural.

Cuando decimos “ordenar de forma natural” nos referimos a lo que pensamos que puede ser el orden “natural”. Por ejemplo, una lista de personas, que tienen nombre y edad, podría considerarse natural el orden por el nombre y que además sea ascendente, de menor a mayor.

El contrato que establece este comportamiento, es la interfaz Comparable. Si necesitamos este comportamiento entonces podemos implementar esta interfaz, y nuestra clase sólo deberá sobreescribir un método "compareTo".

El método "compareTo" compara el objeto actual y otro que pasarán por parámetro, retornando un valor numérico que indica su orden relativo.

Avancemos con el ejemplo, veamos primero la clase Persona:

```
public class Persona {
   private String nombre;
   private int edad;

   public Persona(String nombre, int edad) {
       this.nombre = nombre;
       this.edad = edad;
   }

   public String getNombre() {
       return nombre;
   }

   public int getEdad() {
       return edad;
   }
}
```

Más allá de los métodos que mostramos, la clase Persona puede tener todos los atributos y métodos que se requieran. Con los métodos que especificamos por ahora es suficiente.

Cómo recién mencionamos, la interfaz Comparable, sólo especifica el método compareTo. Este método lo va a utilizar el método de ordenamiento para comparar los objetos de tipo Persona.

La interfaz adicionalmente utiliza una especificación que indica que se debe especificar el tipo al momento de implementar y es justamente ese tipo el que debe usarse en el método para considerar que la Clase lo implementa. Como veremos en breve el tipo **T** será luego **Persona**:

```java
public interface Comparable<T> {
    int compareTo(T o);
}
```

La intención es poder comparar objetos de la clase Persona, tendremos que implementar la interfaz Comparable y sobreescribir el método “compareTo()”.

Recordemos que tenemos que especificar el tipo.

```java
public class Persona implements Comparable<Persona>{
    …

    @Override
    public int compareTo(Persona o) {
          …//este método retorna -1, 0 o +1
    }
}
```

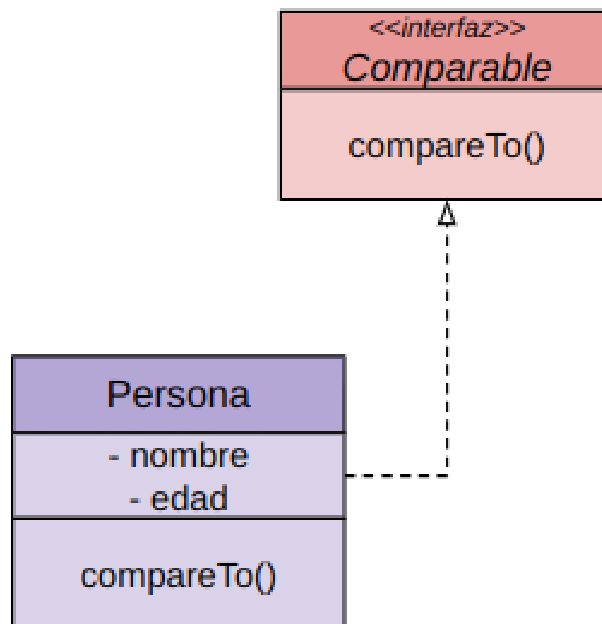


Fig.46 - comparable.

Ahora si, vemos como queda nuestra clase Persona con la implementación de la interfaz Comparable:

```
public class Persona implements Comparable<Persona>{
   private String nombre;
   private int edad;

   public Persona(String nombre, int edad) {
       this.nombre = nombre;
       this.edad = edad;
   }

   public String getNombre() {
       return nombre;
   }

   public int getEdad() {
       return edad;
   }

   @Override
   public int compareTo(Persona o) {
       return this.nombre.compareTo(o.nombre);
   }
}
```

El método compareTo recibe como parámetro un objeto de tipo Persona y lo que hace es retornar la comparación del nombre del objeto actual (`this.nombre`) con el nombre del objeto que me pasaron por parámetro (`o.nombre`).

En definitiva lo que estamos comparando son dos cadenas de texto, estamos comparando los nombres de ambos objetos. Los nombres son de la clase String y está clase también implementa comparable, por lo tanto delegamos la responsabilidad de comparar las cadenas a la clase String.

Si los nombres fueran “Ana” y “María”, la comparación de ambas cadenas retornaría -1,

dado que “Ana” debe posicionarse antes que “María”.

Si los nombres fueran “Juan” y “Ana”, la comparación de ambas cadenas retornaría +1, dado que “Juan” debe posicionarse después que “María”.

El método compareTo retorna

- -1 si el nombre actual es menor.
- 0 si son iguales
- +1 si el nombre actual es mayor.

Finalmente veamos un programa que arma una lista de personas, los ordena y luego muestra el nombre de cada uno por pantalla:

```
public class ListaPersonas {

   public static void main(String[] args) {

       //
       List<Persona> lista = new ArrayList();

       //
       Persona p1 = new Persona("Juan", 20);
       Persona p2 = new Persona("María", 19);
       Persona p3 = new Persona("Ana", 25);

       //
       lista.add(p1);
       lista.add(p2);
       lista.add(p3);

       //
       Collections.sort(lista);
```

```java
        //
        for (Persona p: lista) {
            System.out.println(p.getNombre() + " - " +
                    p.getEdad());
        }
    }
}
```

Saldrá por pantalla:

Ana - 25

Juan - 20

María - 19

Hemos logrado ordenar por nombre la lista de personas. Si quisiéramos ordenar por nombre pero de forma descendente, podemos utilizar

```java
Collections.reverse(lista);
```

Otra forma es cambiando lo que retorna el método compareTo, podemos multiplicar * -1 y así invertir el orden.

Si bien esta forma que propone el lenguaje es útil y simple para ordenar de forma natural cualquier tipo de objeto, nos surge una pregunta, ¿qué pasaría si quisiéramos ordenar la lista por edad después?

Este mecanismo está preparado para permitir el orden natural de las Personas, utilizando su nombre. No podemos tener otro método que se encargue de brindar el orden relativo pero ahora por edad.

El lenguaje, entendiendo esta necesidad, incorpora otra interfaz llamada comparator.

[-] comparador

Comparator es el nombre de la interfaz provista por el lenguaje para ordenar por diferentes criterios. A diferencia de lo que hemos visto con la interfaz Comparable, esta se encuentra pensada para que el comportamiento o el criterio de cómo se debe ordenar esté definido de forma externa al objeto en cuestión.

Veamos ahora que la clase persona queda sin métodos adicionales, como pasaba antes:

```
public class Persona {
   private String nombre;
   private int edad;

   public Persona(String nombre, int edad) {
       this.nombre = nombre;
       this.edad = edad;
   }

   public String getNombre() {
       return nombre;
   }

   public int getEdad() {
       return edad;
   }
}
```

La interfaz Comparator, sólo especifica el método compare. Este método lo va a utilizar el método de ordenamiento para comparar los objetos de tipo Persona.

Al igual que la interfaz anterior, esta utiliza una especificación que indica que se debe especificar el tipo al momento de implementar y es justamente ese tipo el que debe usarse en el método para considerar que la Clase lo implementa. Como veremos en breve el tipo **T** será luego **Persona**.

A diferencia de lo que hemos visto, esta interfaz define un método "compare" que tiene dos parámetros. Ambos parámetros son del mismo tipo y la comparación se realizará entre ellos:

```java
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Entonces necesitamos una nueva clase, por ejemplo OrdenXEdad que deberá implementar la interfaz Comparator y proporcionar el comportamiento del método “compare”.

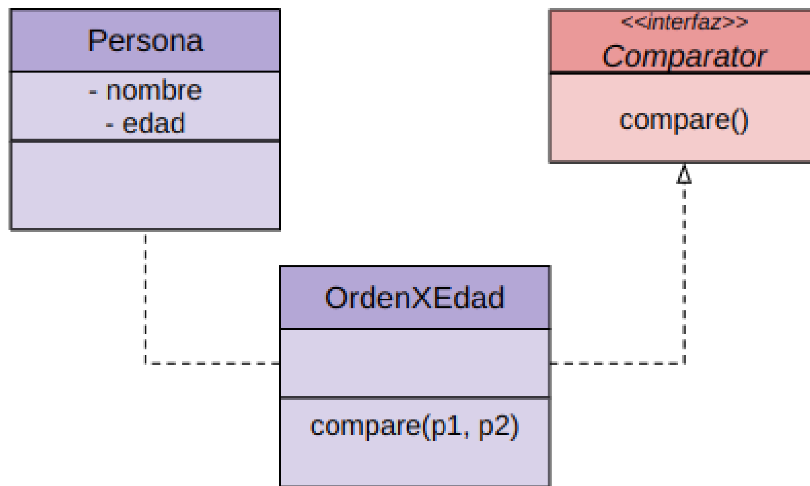


Fig.47 - comparator.

El código de esta nueva clase, teniendo en cuenta que queremos ordenar por edad, podría ser como el siguiente:

```java
public class OrdenXEdad implements Comparator<Persona> {

    @Override
    public int compare(Persona p1, Persona p2) {
        return p1.getEdad() - p2.getEdad();
    }
}
```

```
}
```

Siguiendo con el ejemplo de ordenar una lista de personas, pero ahora el criterio lo hemos cambiado por edad, tendremos el siguiente código. Si prestamos atención es similar al anterior aunque ahora hemos cambiado la forma de ordenar, utilizamos nuevamente el método sort pero ahora le pasamos una instancia de nuestra clase, la que sabe ordenar por edad:

```
Collections.sort(lista, new OrdenXEdad());
```

```
public class ListaPersonas {

   public static void main(String[] args) {

       //
       List<Persona> lista = new ArrayList();

       //
       Persona p1 = new Persona("Juan", 20);
       Persona p2 = new Persona("María", 19);
       Persona p3 = new Persona("Ana", 25);

       //
       lista.add(p1);
       lista.add(p2);
       lista.add(p3);

       //
       Collections.sort(lista, new OrdenXEdad());

       //
       for (Persona p: lista) {
           System.out.println(p.getNombre() + " - " +
                   p.getEdad());
       }
   }
}
```

Como podemos intuir, si quisiéramos tener más criterios de ordenamiento simplemente tendríamos que crear otras clases que implementen Comparator.

Las interfaces Comparable y Comparator son esenciales para ordenar colecciones de objetos en este lenguaje.

Interfaz Comparable:

- La implementamos en la clase del objeto a ordenar.
- Establece como ese objeto se compara con otros del mismo tipo.
- Sobreescribe el método compareTo(Tipo t).
- Se utiliza esencialmente para ordenar de forma natural.

Interfaz Comparator:

- La implementamos en otra clase.
- Establece como dos objetos, del mismo tipo, se comparan.
- Sobreescribe el método compare(Tipo t1, Tipo t2).
- Se utiliza para establecer criterios diferentes al natural.

La herencia múltiple es un mecanismo que proveen ciertos lenguajes para permitir a una clase heredar más de una clase. Java no permite este mecanismo directamente, dado que se presentan algunas desventajas considerables.

La herencia múltiple aumenta la complejidad del código, impactando en la legibilidad y el mantenimiento posterior. Además existe un principio, la substitución de Liskov -dice que cualquier subtipo debe ser sustituible por su padre en cualquier contexto- que no se podría respetar.

Si bien en Java no se permite la herencia múltiple, el lenguaje provee de un mecanismo de comportamiento múltiple a través de las interfaces.

Una clase puede implementar más de una interfaz y de esta manera podemos acercarnos al comportamiento que provee la herencia múltiple.

Vemos la idea a través de un ejemplo que utiliza más de una interfaz.

Volvamos con el ejemplo de las figuras geométricas. Primero armemos dos interfaces que nos explicitan el comportamiento que deberemos tener.

La interfaz Figura:

```java
public interface Figura {

    //
    double calcularArea();

    //
    double calcularPerimetro();
}
```

La interfaz Polígono:

```java
public interface Poligono {

    //
    int obtenerNumeroLados();

}
```

Las clases concretas podrían implementar ambas interfaces.

Veamos la clase Cuadrado como queda:

```java
public class Cuadrado implements Figura, Poligono {

    //
    private final double lado;

    //
    public Cuadrado(double lado) {
        this.lado = lado;
    }

```

```
    //
    @Override
    public double calcularArea() {
        return lado * lado;
    }

    @Override
    public double calcularPerimetro() {
        return 4 * lado;
    }

    @Override
    public int obtenerNumeroLados() {
        return 4;
    }
}
```

La clase Triángulo:

```
public class Triangulo implements Figura, Poligono {

    private double base;
    private double altura;

    public Triangulo(double base, double altura) {
        this.base = base;
        this.altura = altura;
    }

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

```
    @Override
    public double calcularPerimetro() {
        return base + 2 * Math.sqrt((base / 2) * (base / 2) + altu-
ra * altura);
    }

    @Override
    public int obtenerNumeroLados() {
        return 3;
    }
}
```

A partir de cómo hemos diseñado estas clases, mediante la implementación de las interfaces, los objetos del tipo "Triangulo" y "Cuadrado" podrán ser considerados como figuras y polígonos.

El programa que utilice estos objetos podría tener la siguiente forma:

```
public class Programa {

    //
    public static void main(String[] args) {

        //
        Cuadrado cuadrado = new Cuadrado(5);

        //
        Triangulo triangulo = new Triangulo(6,4);

        //
        List<Figura> figuras = new ArrayList();

        //
        figuras.add(cuadrado);
        figuras.add(triangulo);
```

```
        //
        for (Figura figura: figuras) {
            System.out.println("Perímetro: " +
                    figura.calcularPerimetro());
        }

        //
        List<Poligono> poligonos = new ArrayList(figuras);

        //
        for (Poligono poligono: poligonos) {
            System.out.println("Lados: " +
               poligono.obtenerNumeroLados());
        }
    }
}
```

En este caso, hemos armado dos listas, la primera agrega los objetos y los trata como si fueran figuras, la segunda parte de la lista de figuras y las trata como si fueran polígonos.

Luego se recorren las listas y se envían los mensajes “calcularPerimetro” y “obtenerNumeroLados” a los objetos, los cuales responden cada uno con su implementación interna.

Nuevamente se nos presenta esta situación en la que los objetos envían el mismo mensaje y cada uno responde de manera diferente. Este comportamiento es posible debido al polimorfismo.

Polimorfismo

Lo que hemos visto en los ejemplos anteriores, la capacidad de los objetos pertenecientes a una misma clase para responder de manera diferente a la llamada de una misma operación, es lo que denominamos polimorfismo.

El polimorfismo es un concepto fundamental en la programación orientada a objetos, se basa en dos mecanismos claves, en la herencia y en el enlace dinámico.

Como hemos desarrollado, la herencia establece una relación de jerarquía entre clases,

permitiendo que las clases derivadas hereden atributos y métodos de la clase padre.

En la memoria, esto se refleja en la disposición de los objetos. Los objetos de clases derivadas ocupan más espacio de memoria que los objetos de su clase padre, ya que contienen los atributos y métodos heredados, además de sus propios atributos y métodos específicos.

En tiempo de ejecución, la herencia permite que los objetos de clases derivadas respondan a mensajes enviados como lo harían los de su clase padre, porque en definitiva tienen los atributos y métodos en su espacio de memoria.

La otra clave detrás del polimorfismo es el enlace dinámico. Es el proceso de determinar qué implementación de un método se debe ejecutar cuando se invoca el método sobre un objeto.

En memoria, el enlazado dinámico se implementa mediante tablas de métodos asociadas a cada clase. Estas tablas contienen referencias a las implementaciones de los métodos definidos en la clase y sus clases padre.

En tiempo de ejecución, cuando se invoca un método sobre un objeto, el intérprete primero busca la implementación del método en la tabla de métodos de la clase del objeto. Si no se encuentra la implementación, el intérprete continúa y busca en las tablas de métodos de la clase padre, siguiendo la jerarquía de herencia, hasta encontrar una implementación del método.

Este proceso de búsqueda en tiempo de ejecución es lo que permite que el mismo mensaje se invoque sobre objetos de diferentes clases y se ejecute la implementación correspondiente a cada tipo de objeto.

Fijarse que ambos objetos, de la clase "FiguraGeometrica" invocan al mismo método "calcularArea()" y ambos responderán de diferente manera.

Veamos un ejemplo que demuestra nuevamente la potencia y flexibilidad que presenta el uso de interfaces y clases abstractas combinadas. En este caso, gracias al polimorfismo, podemos ver la simpleza del código final en el “Programa”.

La idea general es realizar un programa que nos permita manejar trabajadores. A cada trabajador, dependiendo el tipo de relación que tenga, se le abonará un determinado sueldo. Al trabajador por proyecto se le paga en función de la cantidad de proyectos que esté realizando y un valor fijo por cada uno. Al Trabajador por hora se le paga en función de la cantidad de horas y el valor de la misma, y finalmente al trabajador de planta se le paga en función del valor de la hora por una cantidad fija de horas mensuales y un porcentaje adicional en función de su antigüedad.

Podemos diseñar este problema utilizando una interfaz para representar el comportamiento de cada trabajador. De cada uno necesitamos que se pueda calcular el sueldo y mostrar cierta información del trabajador y que se tuvo en cuenta para el cálculo de su sueldo. También podemos pensar en tener una clase abstracta que nos proporcione los atributos necesarios que tiene un empleado, su nombre, DNI, domicilio y celular. La clase será abstracta porque no queremos que se instancie un empleado sin

decir de qué tipo es. Finalmente las tres clases concretas que representan el tipo de empleado. Comencemos con la interfaz Trabajador:

```
public interface Trabajador {

    //
    double calcularSueldo();

    //
    String mostrarInformacion();

}
```

En la interfaz definimos el comportamiento que debe tener una clase para considerarse, en este ejemplo, un trabajador.

Ahora pasemos a la especie de plantilla que necesitamos para crear un empleado, la clase abstracta Empleado nos dará esta herramienta:

```
public abstract class Empleado {

    private String nombre;
    private long dni;
    private String domicilio;
    private String celular;

    public Empleado(String nombre, long dni,
                    String domicilio, String celular) {
        this.nombre = nombre;
        this.dni = dni;
        this.domicilio = domicilio;
        this.celular = celular;
    }
```

```
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public long getDni() {
        return dni;
    }

    public String getDomicilio() {
        return domicilio;
    }

    public void setDomicilio(String domicilio) {
        this.domicilio = domicilio;
    }

    public String getCelular() {
        return celular;
    }

    public void setCelular(String celular) {
        this.celular = celular;
    }

    public abstract String mostrarDatosAdicionales();

    public String mostrarInformacion() {
        return "Persona{" +
                "nombre='" + nombre + '\'' +
                ", dni=" + dni +
```

```
                    ", domicilio='" + domicilio + '\'' +
                    ", celular='" + celular + '\'' +
                    "}, {"+ mostrarDatosAdicionales()+"}";
    }
}
```

Como se puede ver en la definición de la clase abstracta, cualquier tipo de empleado, además de heredar todos los atributos y métodos definidos, deberá proveer su método para obtener los datos adicionales a la información que se piensa retornar. Cuando se llame al método “mostrarInformacion()” se hará uso de la información adicional que se quiere presentar. La idea es que cada tipo de empleado proporcione la suya.

Veamos las clases que representan a los tres tipos de trabajadores:

```
public class TrabajadorXProyecto extends Empleado
                                 implements Trabajador {

    private static final int VALOR_PROYECTO = 1000;
    private int cantProyectos;

    public TrabajadorXProyecto(String nom, long dni,
                               String dom, String cel,
                               int cantProyectos) {
        super(nom, dni, dom, cel);
        this.cantProyectos = cantProyectos;
    }

    @Override
    public double calcularSueldo() {
        return cantProyectos * VALOR_PROYECTO;
    }

    public int getCantProyectos() {
        return cantProyectos;
    }
```

```java
    public void setCantProyectos(int cantProyectos) {
        this.cantProyectos = cantProyectos;
    }

    @Override
    public String mostrarDatosAdicionales() {
        return "cantProyectos=" + cantProyectos+
                 ", valorXProyecto=" + VALOR_PROYECTO;
    }
}
```

```java
public class TrabajadorXHora extends Empleado
                             implements Trabajador {

    private double valorHora;
    private int cantHoras;

    public TrabajadorXHora(String nom, long dni,
                           String dom, String cel,
                           double valorHora, int cantHoras) {
        super(nom, dni, dom, cel);
        this.valorHora = valorHora;
        this.cantHoras = cantHoras;
    }

    @Override
    public double calcularSueldo() {
        return valorHora * cantHoras;
    }
```

```
    public double getValorHora() {
        return valorHora;
    }

    public void setValorHora(double valorHora) {
        this.valorHora = valorHora;
    }

    public int getCantHoras() {
        return cantHoras;
    }

    public void setCantHoras(int cantHoras) {
        this.cantHoras = cantHoras;
    }

    @Override
    public String mostrarDatosAdicionales() {
        return "valorHora=" + valorHora +
               ", cantHoras=" + cantHoras;
    }
}
```

```
public class TrabajadorDePlanta extends Empleado
                                implements Trabajador {

    private int valorHora;
    private static final int HORA_MES = 140;
    private int antiguedad;
```

```java
    public TrabajadorDePlanta(String nom, long dni,
                              String dom, String cel,
                              int valHora, int ant) {
        super(nom, dni, dom, cel);
        this.valorHora = valHora;
        this.antiguedad = ant;
    }

    @Override
    public double calcularSueldo() {
        return (valorHora * HORA_MES)
             +((valorHora * HORA_MES)*antiguedad/100);
    }

    public int getAntiguedad() {
        return antiguedad;
    }

    @Override
    public String mostrarDatosAdicionales() {
        return  "valorHora=" + valorHora +
                ", horasXMes=" + HORA_MES +
                ", antiguedad=" + antiguedad;
    }
}
```

Cada clase concreta implementa el método que definió la interfaz, pero si se observa detenidamente, el método “mostrarInformacion” no se encuentra presente. Esto se debe a que el mismo se encuentra definido en la clase abstracta y como se hereda de la misma se cumple el contrato.

Finalmente veamos el programa que inicializa a tres tipos de trabajadores y muestra el sueldo de cada uno.

```
public class Programa {
   public static void main(String[] args) {

       //
       List<Trabajador> empleados = new ArrayList<>();

       //
       empleados.add(new TrabajadorXProyecto("Juan Pérez",
               12345678,
               "Calle 123",
               "1234567890",
               5));
       empleados.add(new TrabajadorXHora("María González",
               87654321,
               "Calle 456",
               "9876543210",
               20.00,
               120));
       empleados.add(new TrabajadorDePlanta("Pedro López",
               78901234,
               "Calle 789",
               "0987654321",
               20,
               10));

       //
       for (Trabajador empleado : empleados) {

           System.out.printf("Empleado: %s\n",
                empleado.mostrarInformacion());

           System.out.printf("Sueldo: %f$ \n",
                empleado.calcularSueldo());
```

```
            System.out.println("--------------------");
        }
    }
}
```

Genéricos

Los genéricos (generics) en Java son una característica del lenguaje que permite crear clases, interfaces y métodos que pueden trabajar con diferentes tipos de datos sin necesidad de especificarlos explícitamente en la declaración. Esto permite escribir código más flexible, reutilizable y seguro.

Es un concepto que permite crear clases, interfaces y métodos que operan con tipos específicos pero que son capaces de trabajar con distintos tipos de datos.

Surgen a partir de la necesidad de escribir código que fuera independiente del tipo de datos con el que se quiera trabajar.

Hemos visto códigos en los cuales se han especificado los tipos y esto ha resultado natural, por ejemplo la clase lista permite almacenar nombres si declaro que almacenará el tipo de dato String, o puede almacenar valores numéricos si declaro que almacenará Integer :

```
//
List<String> nombres = new ArrayList();

//
List<Integer> numeros = new ArrayList();
```

O un diccionario de ocurrencias, a partir de una palabra quiero contar la cantidad de veces que apareció. De la misma forma podría tener un diccionario que a partir de un valor, por ejemplo el dni, pueda obtener una persona:

```java
//
Map<String, Integer> ocurrencias = new HashMap();

//
Map<Integer, Persona> personas = new HashMap();
```

Antes de los genéricos, si se quería escribir un método que trabajara con diferentes tipos de datos, se necesitaba utilizar el tipo Object, lo que implicaba perder la información del tipo específico y además se requerían conversiones explícitas -el uso de cast- a lo que hemos mencionado como una estrategia que no es deseable.

El uso de genéricos permite especificar tipos como parámetros en la definición de clases, las interfaces y los métodos. Se utiliza la sintaxis <Tipo> para indicar el tipo genérico. Por convención se utiliza una letra en mayúscula.

La letra que se suele utilizar depende de lo que representa:

- T : un tipo de dato genérico (Type).
- E : un elemento en una colección (Element).
- K : una clave en un mapa (Key).
- V : un valor en un mapa (Value).
- S : un tipo de dato de origen (Source).
- U : un tipo de dato de destino (Target).

Cuando desarrollamos el concepto de Interfaz, en particular, el uso de Comparable y Comparator para ordenar, las mismas utilizan también la forma genérica de declaración.

```java
public interface Comparable<T> {

   int compareTo(T o);

}
```

Al momento de implementar la clase se debe especificar el tipo de dato que quiere utilizar. El tipo de dato puede ser de cualquier clase.

A partir de estas definiciones, creemos nuestra propia clase “Variable” que utilice el concepto para definir de forma genérica el tipo de dato que que se puede crear, almacenar y obtener:

```
public class Variable<T> {

   //
   private T valor;

   public Variable(T inicial) {

       this.valor = inicial;
   }

   public T getValor() {

       return valor;
   }

   public void setValor(T valor) {

       this.valor = valor;
   }
}
```

Finalmente veamos como se puede usar esta clase de objetos para utilizar variables que almacene un valor entero o uno de tipo texto:

```
public class Programa {

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

```
        //
        Variable<Integer> variable = new Variable(10);
        System.out.println(variable.getValor());

        //
        variable.setValor(20);
        System.out.println(variable.getValor());

        //
        Variable<String> otra = new Variable("Hola");
        System.out.println(otra.getValor());

        //
        otra.setValor("Hola Mundo, yo programo!");
        System.out.println(otra.getValor());
    }
}
```

El constructor, en ambos casos, acepta el argumento porque se infiere que el tipo que estoy pasando a T, es Integer y String respectivamente. Luego los métodos setValor y getValor esperan trabajar con este tipo de dato, ya sea para establecer el valor a través del argumento o cuando se retorne el valor contenido por el objeto variable.

La utilización de genéricos nos brinda varios beneficios. Podemos reutilizar clases e interfaces en diferentes contextos sin la necesidad de duplicar código. Dotamos a nuestras soluciones de mayor seguridad en términos de verificación de tipos. En tiempo de ejecución no vamos a encontrarnos con situaciones inesperadas, como las que se pueden producir cuando utilizamos operaciones de conversión de tipos. El código termina siendo más legible y fundamentalmente es más fácil de mantener.

Ejercicios

1. Definir una clase abstracta llamada Figura con un método abstracto calcularArea(). Luego, crear clases concretas como Círculo, Triángulo y Rectángulo que hereden de Figura e implementen calcularArea() y calcularPerimetro().
2. Crear una clase abstracta Empleado con atributos como nombre, DNI y horas trabajadas y un método para calcular el sueldo. Luego, las clases Gerente y Técnico. Calcular el sueldo de los empleados sabiendo que los gerentes tienen el valor de la hora a 10.000 y los técnicos a 6000.
3. Definir una interfaz llamada Conducible con métodos como acelerar() y frenar(). Luego, crear clases como Automóvil y Motocicleta que implementen esta interfaz.
4. Hacer un programa que pida palabras al usuario y luego las imprima.
   a. en orden alfabético.
   b. en orden inverso al alfabético.
   c. en orden ascendente según su longitud.
5. Crear una clase abstracta Animal con métodos como comer() y hacerSonido(). Luego, crea clases concretas como Perro y Gato que hereden de Animal e implementen los métodos.
6. Almacenar en una lista, una cantidad de N estudiantes. De cada estudiante interesa saber su nombre, apellido, año de ingreso y promedio general.
   Se pide mostrar la información de cada estudiante:
   a. ordenados apellido y nombre
   b. ordenados por promedio descendente.
7. Diseñar una jerarquía de clases que representen cuentas bancarias. La clase CuentaBancaria cuenta con los métodos depositar, retirar y saldo. Crear las clases CuentaCorriente y CuentaAhorros y probar su funcionamiento sabiendo que la cuenta corriente aplica una comisión del 1% en cada movimiento y la caja de ahorros puede aplicar un interés sobre el saldo de 1% cada vez que lo necesiten.
8. Crea una interfaz Transporte con métodos como mover, detener y un método especial que

emula el mal funcionamiento -puede utilizar una función aleatorio para su emulación-, luego implementar esta interfaz en clases como Automóvil y Avión. Probar su funcionamiento creando varios objetos del tipo transporte, almacenarlos, ponerlos en movimiento y si llegan a tener un mal funcionamiento detenerlos. Generar un informe al finalizar de cuántos se han detenido por mal funcionamiento y cuántos quedan operativos.

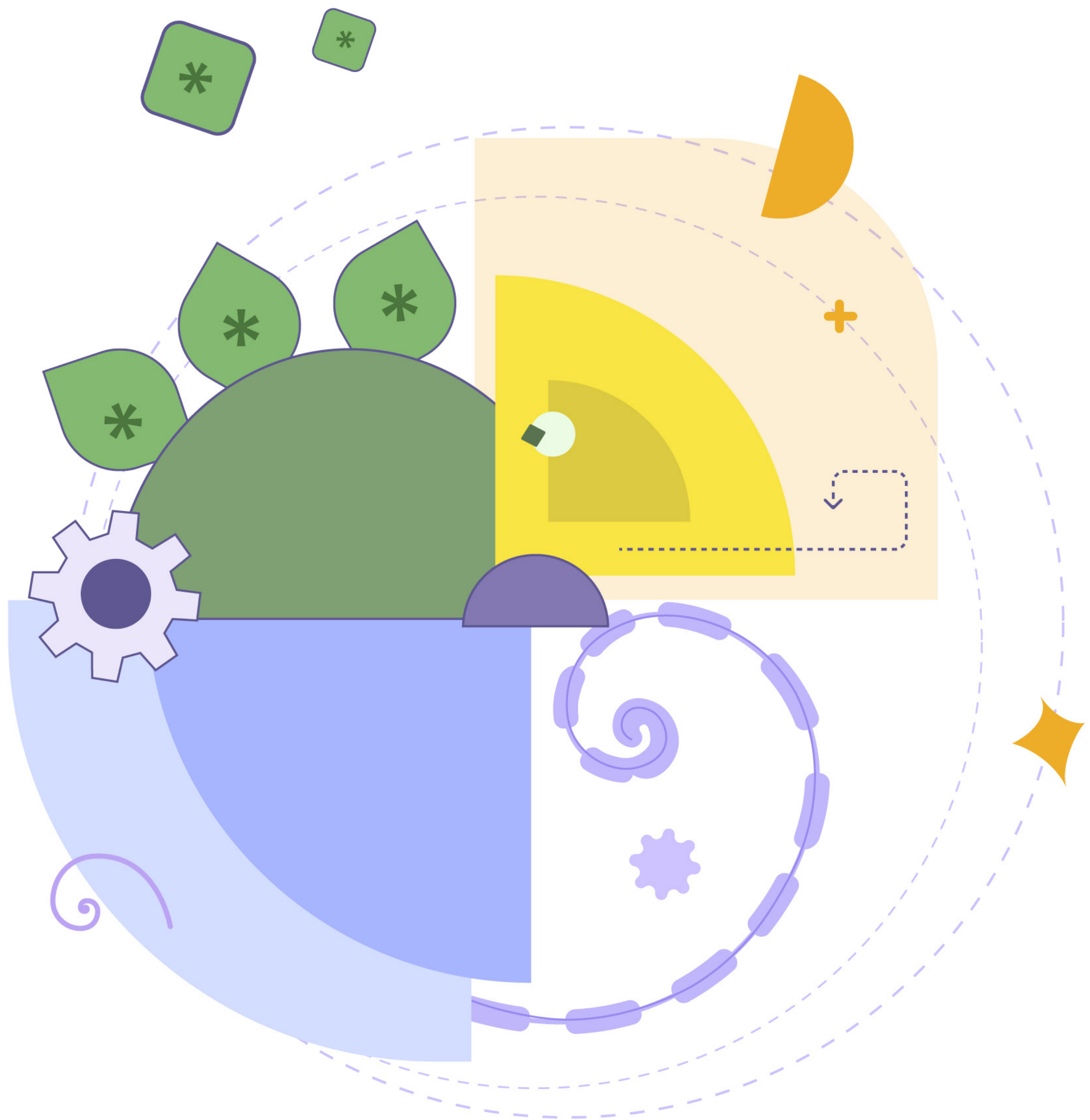
9. Extender el ejercicio de las figuras geométricas para incluir figuras tridimensionales como Esfera y Cubo.

10. Un zoológico tiene varios tipos de animales, como leones, tigres y osos. Diseñar una manera de alimentar a estos animales -pensar en el método comer- sin especificar cómo se alimenta a cada tipo. Organizar las clases de tal modo que para que puedas agregar nuevos tipos de animales no haya que cambiar la lógica de alimentación existente.

11. Simular un videojuego donde los personajes pueden recolectar diferentes tipos de recursos, como oro, madera y piedra. Diseñar las clases necesarias para manejar estos recursos sin especificar cómo se almacenan o utilizan en cada caso. Garantizar que se puedan agregar más tipos de recursos fácilmente.

12. Simular un sistema de comercio electrónico para manejar diferentes métodos de pago, como tarjetas de crédito, transferencias desde billeteras electrónicas y transferencias bancarias. Se deben diseñar una forma de procesar los pagos sin especificar el método de procesamiento en cada caso. Se debe estructurar la parte de métodos de pago para que sea flexible y extensible.

Anexo



Utilizar el entorno de desarrollo

→ Abrimos el IDE

→ Creamos un nuevo proyecto [+]

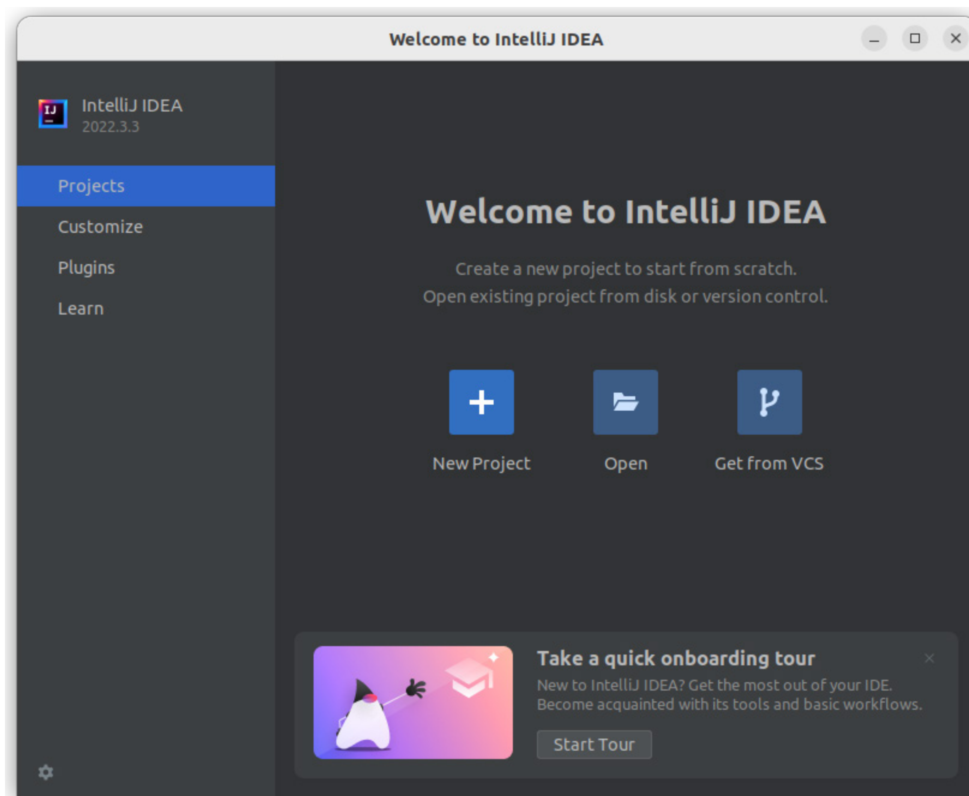


Fig. Anexo 01 - Pantalla inicial.

→ Establecemos los valores del proyecto y pulsamos crear

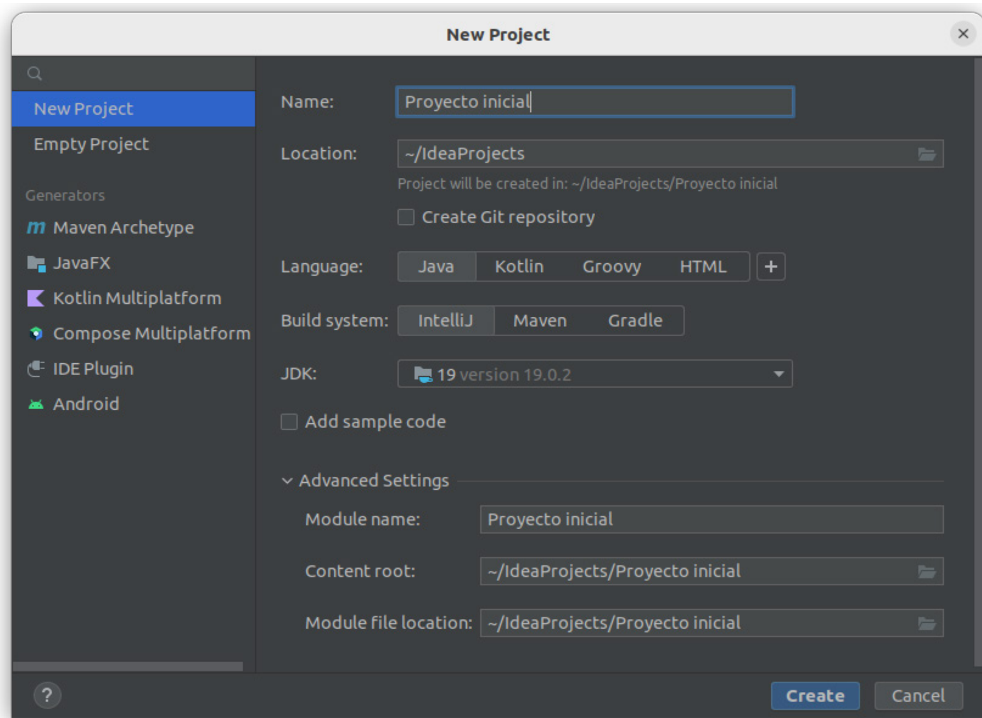


Fig. Anexo 02 - Proyecto nuevo.

→ pulsamos botón derecho sobre la carpeta src → seleccionamos new → java class

→ establecemos el nombre como “HolaMundo”, verificamos que esté seleccionado Class y pulsamos enter

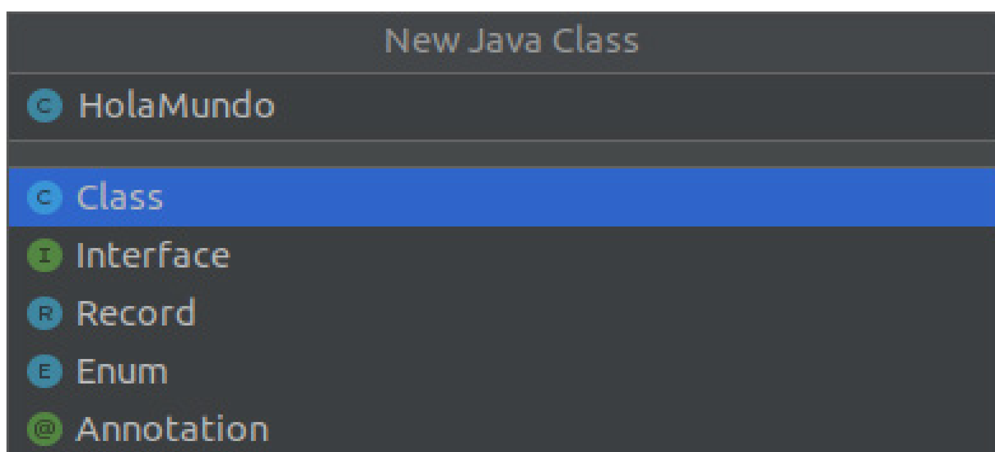


Fig. Anexo 03 - Clase nueva

→ Agregar el siguiente fragmento de código a nuestro primer programa

```
public static void main(String[] args){
    //
    System.out.println("Hola mundo!!!");
}
```

Deber verse como la siguiente imagen

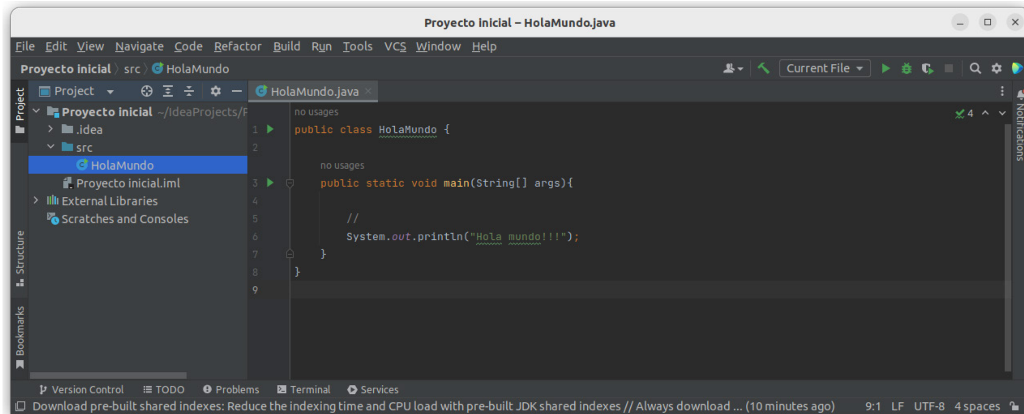


Fig. Anexo 04 - IDE IntelliJ

→ Ejecutamos el programa haciendo click en el play verde (➤)

→ veremos la "salida" de nuestra ejecución en la ventana inferior

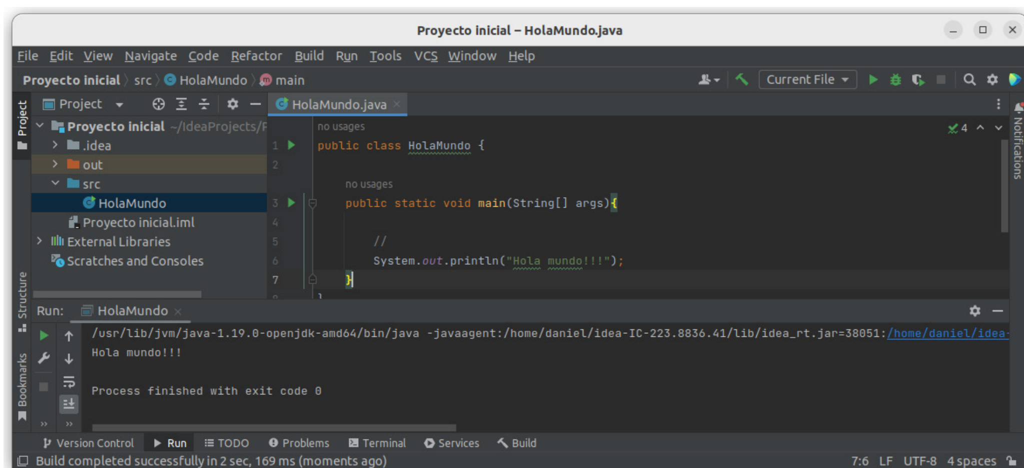
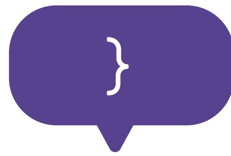


Fig. Anexo 05 - Salida de la ejecución

Bibliografía

- De Giusti A., Madoz M., Bertone R., Naiouf M., Lanzarini L., Gorga G., Russo C., Champredonde R.: Algoritmos, datos y programas. Pearson. 2001.
- Dale N., Joyce D., Weems C. Object-Oriented Data Structures Using Java, Jones & Bartlett Learning. 2016.
- Booch G., Maksimchuk R., Engle M., Young B., Conallen J., Houston K. Object-Oriented Analysis and Design with Applications. Addison-Wesley. 2007.
- Joshua Bloch: Effective Java (3ra ed). Addison-Wesley. 2018.
- Sierra K., Bates B., Gee T. Head First Java: A Brain-Friendly Guide. O'Reilly. 2022.
- Hillar G. Learning Object-Oriented Programming. Packt Publishing. 2015.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1994.
- Martin Fowler: Patterns of Enterprise Application Architecture. Addison-Wesley. 2002.



YO PROGRAMO

{ en JAVA }

El principal objetivo del libro es enseñar a programar, lo que implica también proporcionar habilidades y conocimientos necesarios para crear, desarrollar y utilizar software.

Prepararse para la era digital. Vivimos en una sociedad cada vez más digitalizada, donde la tecnología desempeña un papel importante en casi todos los aspectos de nuestras vidas. Aprender a programar nos prepara para comprender y participar en esta era digital, y nos brinda una herramienta adicional para adaptarnos a los avances tecnológicos.

DANIEL E. AGUIL MALLEA

ISBN 978-631-00-5204-5

9 786310 052045