

FACUNDO UFERER



JAVA PARA PRINCIPIANTES

Contenidos

Contenidos	1
Sobre éste Apunte	8
La Inteligencia Artificial delante de este libro	8
El Algoritmo	9
Las Instrucciones	10
Buenas Prácticas de programación:	11
1) Nombres descriptivos:	11
2) Comentarios claros y concisos:	11
3) Divide y vencerás:	11
4) Evita la duplicación de código:	11
5) Mantén la simplicidad:	12
6) Gestión de errores y excepciones:	12
7) Formato y estilo de código:	12
8) Prueba tu código:	12
9) Aprende de los demás:	12
10) Practica regularmente:	13
Empezando con Java	14
¿Qué es Java?	14
Las ventajas en aprender Java	15
El primer programa en Java	15
¿Qué es lo que acabamos de hacer?	16
La salida estándar	16
System.out.println()	17
System.out.print()	17
System.out.printf()	18
System.out.write()	18
Las variables	19
Tipos de datos	20
Tipos de datos primitivos	20
Enteros	21

byte	21
short	21
int	22
long	22
Flotantes	22
float	22
double	22
Caracteres	23
Booleanos	23
Resumiendo en una tabla	23
Operadores	24
Operadores Aritméticos:	24
Suma (+)	24
Resta (-)	24
Multiplicación (*)	24
División (/)	25
Módulo (%)	25
Incremento (++) y Decremento (--)	25
Operadores de Comparación:	26
Igualdad (==)	26
Desigualdad (!=)	26
Mayor que (>)	27
Menor que (<)	27
Mayor o igual que (>=)	27
Menor o igual que (<=)	28
Operadores Lógicos:	28
Operador AND (&&)	28
Operador OR ()	29
Operador NOT (!)	29
Operadores de Asignación:	30
Operador "="	30
Operador "+="	30
Operador "-="	30
Operador "*="	31

Operador "/="	31
Operador "%="	31
La estructura de control de flujo	32
Estructuras Condicionales:	32
if	32
if-else	32
if-else-if	33
Estructura switch-case o condicional múltiple	33
Estructuras de Repetición o bucles	34
while:	34
for	34
Arrays	35
Características principales de los Arrays	36
Formas de inicializar un array	36
Inicialización con valores explícitos	36
Inicialización con constructor	37
Inicialización con tamaño y asignación de valores posterior	37
Recorrer un Array	38
¿Cómo mezclar dos arrays?	39
Arrays Multidimensionales	40
Declaración e inicialización de un arreglo multidimensional:	40
Declaración e inicialización de un arreglo bidimensional	40
Declaración e inicialización de un arreglo tridimensional	41
Acceso a los elementos de un arreglo multidimensional:	41
Acceso a un elemento en un arreglo bidimensional	41
Acceso a un elemento en un arreglo tridimensional	41
Inicialización de un arreglo multidimensional con valores:	41
Inicialización de un arreglo bidimensional	42
Inicialización de un arreglo tridimensional	42
Iteración sobre un arreglo multidimensional:	42
Iteración sobre un arreglo bidimensional	42
Iteración sobre un arreglo tridimensional	42

String	43
Recorrer un String	43
Concatenación de String	44
Métodos y operaciones principales de los Strings	45
Concatenar Strings	45
Comparar Strings	46
Tamaño de un String	47
Pasar un String a mayúscula o minúscula	48
Indicar si contiene otro string	49
Verificar si comienza o termina con otro String	49
Carácter de escape	50
Funciones	51
Programación Orientado a Objetos (POO)	53
Programación estructurada vs. orientada a objetos	53
Encapsulamiento:	53
Herencia	55
Polimorfismo	55
Sobreescritura de métodos	55
Sobrecarga de métodos	56
Abstracción:	57
Clase Abstracta	57
La interfaz	58
Cómo implementar una interface	58
Modificadores de acceso	59
public	60
private	60
protected	61
default	62
Clases	63
Los elementos de instancia	65
Elementos de Clase o Estáticos	67
Variables de Clase	67
Métodos de Clase	67
Constructores	68
Objetos	70
Los Wrappers	71

Colecciones	72
Excepciones	72
Tipos de errores pueden ocurrir durante la ejecución de un programa	72
Errores de tiempo de compilación	73
Errores de tiempo de ejecución	73
Tipos de Excepciones	73
Verificadas	73
No verificadas	73
Bloque try-catch	74
Bloque finally	74
Interfaces funcionales	75
Definición de una interfaz funcional	75
Expresiones lambda	76
Métodos de referencia:	76
Uso de interfaces funcionales predefinidas	76
Expresiones lambda	77
Sintaxis de una expresión lambda:	77
Expresión lambda que no toma argumentos	78
Expresión lambda en una variable	78
Expresión lambda en una lista:	79
Aplicación de Lambdas en colecciones	79
forEach	79
anyMatch	80
allMatch	80
map	81
Aplanación de colecciones	81
Collections.min()	81
Collections.max()	82
IntStream.sum()	82
Collectors.joining()	82
Clasificación	84
1. Patrones de creación	84

2. Patrones de estructura:	85
3. Patrones de comportamiento:	85
Patrón State	85
Ejemplo del Patrón State	86
1. Crear las clases de estado	86
2. Crear la clase Contexto:	88
3. Utilizar el patrón State	89
Patrón Template Method	89
Estructura del Patrón Template Method	90
1. Clase Abstracta (Clase Base)	90
2. Clases Concretas (Subclases):	90
Ejemplo del Patrón Template Method	90
Paso 1: Clase abstracta que define el Template Method	90
Paso 2: Clases concretas (jugadores) que heredan de la clase abstracta	91
Paso 3: Programa principal que utiliza el patrón Template Method	92
Cómo escribir y leer archivos en Java:	93
Clases principales:	93
Proceso básico:	93
Leer un Archivo	94
Clases principales:	94
Proceso básico:	94
try-catch en archivos	96
Ventajas de utilizar Maven	98
El Archivo POM	98
Las partes principales de un archivo POM	100
Elemento de proyecto (project):	100
Elemento de dependencias (dependencies):	100
Elemento de plugins (plugins):	101
Elemento de perfiles (profiles):	101
Instalar Maven	102
Compilar	105
Ejecutar el programa	106
Testing	106
El testing busca encontrar fallas en el producto	107
Busca la eficiencia	107
Busca encontrar la mayor cantidad de fallas	108

Intenta no detectar fallas que en realidad no lo son	108
Pretende encontrar las fallas más importantes	108
Las condiciones de prueba	108
Casos de prueba	109
Tipos de Tests	110
Pruebas Unitarias	110
Pruebas de Integración	110
Pruebas de Aceptación de Usuario (UAT)	110
Pruebas de Stress	111
Pruebas de Volumen y Performance	111
Pruebas de Regresión	111
Pruebas Alfa y Beta	111
jUnit	112
Algunas de las principales anotaciones de JUnit son:	112

Sobre éste Apunte

La Inteligencia Artificial delante de este libro

La tapa de este libro es una obra de arte generada por una inteligencia artificial que permitió crear una imagen única basada en diseños de artistas humanos, pero la cosa no se detiene allí. El texto que estás leyendo también fue corregido y revisado por una inteligencia artificial, para garantizar una lectura fluida y sin errores gramaticales o de ortografía que yo no podría garantizar.

La inteligencia artificial se ha convertido en una herramienta poderosa que nos permite crear y mejorar productos de formas nunca antes imaginadas. Este libro es solo un ejemplo de cómo la inteligencia artificial puede mejorar la experiencia de lectura y abrir nuevas posibilidades.

Conceptos Básicos

de programación

El Algoritmo

Un algoritmo es un conjunto de pasos que nos ayudan a resolver un problema. Es como una receta de cocina, donde cada paso tiene que ser seguido en orden para obtener el resultado deseado.

Por ejemplo, si queremos hacer una torta, tenemos que seguir un algoritmo que nos indica los pasos a seguir: mezclar los ingredientes, poner la mezcla en un molde, hornear la torta, y luego decorarla. Si seguimos los pasos correctamente, tendremos una deliciosa torta para disfrutar.

Ahora, en Java, podemos escribir algoritmos en forma de código. Por ejemplo, si queremos escribir un algoritmo que sume dos números, podemos escribir el siguiente código:

```
int numero1 = 5;  
int numero2 = 3;  
int resultado = numero1 + numero2;  
System.out.println("El resultado es: " + resultado);
```

Aquí, estamos declarando dos variables, numero1 y numero2, y luego sumándolas y almacenando el resultado en una tercera variable, resultado. Luego, estamos imprimiendo el resultado en la consola dandonos como resultado lo siguiente:

```
El resultado es: 8
```

Las Instrucciones

Las instrucciones son el conjunto de órdenes que le damos a un programa de computadora para que realice una tarea específica. Cada instrucción representa una acción que el programa debe llevar a cabo y, por lo general, se escribe en un lenguaje de programación como Java.

Las instrucciones en Java pueden ser de diferentes tipos, dependiendo de lo que queramos que el programa haga. Algunos ejemplos de instrucciones en Java son:

Asignación de valores a variables: Esta instrucción se utiliza para asignar un valor a una variable en el programa. Por ejemplo, si queremos asignar el valor 5 a una variable llamada "numero", podemos escribir la siguiente instrucción:

```
int numero = 5;
```

Operaciones matemáticas: Las instrucciones de operaciones matemáticas se utilizan para realizar cálculos en el programa. Por ejemplo, si queremos sumar dos números y almacenar el resultado en una variable llamada "resultado", podemos escribir la siguiente instrucción:

```
int resultado = 5 + 3;
```

Condiciones: Las instrucciones de condiciones se utilizan para tomar decisiones en el programa. Por ejemplo, si queremos comprobar si un número es mayor que otro y realizar una acción en consecuencia, podemos escribir la siguiente instrucción:

```
if (numero1 > numero2) {  
    System.out.println("El número 1 es mayor que el número 2");  
}
```

Aquí, la instrucción "if" comprueba si la condición especificada (en este caso, si "numero1" es mayor que "numero2") es verdadera. Si es así, se ejecutará la instrucción dentro de las llaves (en este caso, imprimir un mensaje en la consola).

Buenas Prácticas de programación:

1) Nombres descriptivos:

- Utiliza nombres de variables, funciones y clases descriptivos que reflejen su propósito y función en el programa.
- Evita abreviaturas poco claras y nombres genéricos como a, x, temp, etc.

2) Comentarios claros y concisos:

- Escribe comentarios que expliquen el propósito y la funcionalidad de secciones de código importantes.
- Evita comentarios obvios que solo repitan lo que hace el código. En su lugar, enfócate en explicar por qué se hace algo o en detalles importantes que no son evidentes de inmediato.

3) Divide y vencerás:

- Divide tu código en funciones o métodos más pequeños y cohesivos que realicen tareas específicas.
- Esto hace que el código sea más fácil de entender, mantener y depurar.

4) Evita la duplicación de código:

- Identifica partes repetitivas del código y encapsúlalas en funciones o métodos reutilizables.
- Esto mejora la legibilidad y facilita los cambios futuros.

5) Mantén la simplicidad:

- Escribe código simple y directo. Evita soluciones excesivamente complicadas para problemas simples.
- La simplicidad facilita la comprensión y reduce la posibilidad de errores.

6) Gestión de errores y excepciones:

- Maneja los errores de manera apropiada utilizando mecanismos como try-catch-finally para evitar fallos inesperados en el programa.
- Propaga las excepciones solo cuando sea necesario y captura solo las excepciones que puedas manejar.

7) Formato y estilo de código:

- Mantén un formato de código consistente y sigue un estilo de codificación establecido (por ejemplo, convenciones de nombres, indentación, etc.).
- Utiliza herramientas de formateo automático para mantener un estilo uniforme.

8) Prueba tu código:

- Realiza pruebas exhaustivas en tu código para garantizar que funcione correctamente en diferentes escenarios.
- Prueba tanto el camino feliz como los casos de borde y las situaciones de error.

9) Aprende de los demás:

- Lee y estudia el código de otros programadores, especialmente de aquellos con más experiencia.
- Participa en comunidades de desarrollo, foros y proyectos de código abierto para obtener retroalimentación y aprender nuevas técnicas.

10) Practica regularmente:

- La práctica constante es clave para mejorar en la programación. Dedica tiempo a escribir código regularmente y a enfrentarte a nuevos desafíos.



Empezando con Java

Es genial que hayas decidido iniciar tu camino en este lenguaje de programación tan popular y útil.

Java es un lenguaje de programación de propósito general que se utiliza en una amplia gama de aplicaciones, desde aplicaciones móviles hasta aplicaciones empresariales y de escritorio. Con Java, podrás crear aplicaciones eficientes y de alta calidad que se ejecuten en diferentes sistemas operativos y plataformas.

A medida que avances en tu aprendizaje de Java, podrás descubrir la flexibilidad y portabilidad del lenguaje, así como su amplia demanda en el mercado laboral. Además, tendrás acceso a una gran cantidad de recursos y soporte de una comunidad activa de desarrolladores de Java.

Ya sea que estés buscando crear tu primera aplicación o deseas avanzar en tu carrera en el desarrollo de software, Java es un lenguaje de programación excelente para aprender.

Estoy emocionado de tenerte como parte de la comunidad de aprendizaje de Java y espero acompañarte en tu camino hacia el éxito en el mundo de la programación.

¡Vamos juntos a explorar todo lo que Java tiene para ofrecer!

¿Qué es Java?

JavaScript es un lenguaje de programación interpretado, orientado a objetos y de alto nivel, que se utiliza principalmente en el desarrollo de aplicaciones web del lado del cliente (en el navegador).

Fue creado originalmente por Brendan Eich en 1995 y ha evolucionado desde entonces para convertirse en uno de los lenguajes de programación más populares del mundo. JavaScript se utiliza para agregar interactividad, validación de datos, animaciones, efectos visuales y mucho más a las páginas web.

Las ventajas en aprender Java

Hay varias ventajas en aprender Java, algunas de las más importantes son:

Amplia demanda laboral: Java es uno de los lenguajes de programación más populares y ampliamente utilizados en el mundo, lo que significa que hay una gran demanda de desarrolladores de Java en la industria. Aprender Java puede abrirte muchas oportunidades laborales en diferentes sectores, como la banca, las finanzas, la salud, la educación y la tecnología.

Portabilidad: Java es un lenguaje de programación portátil, lo que significa que los programas escritos en Java se pueden ejecutar en diferentes plataformas y sistemas operativos sin necesidad de reescribir el código. Esto hace que Java sea una opción popular para desarrollar aplicaciones empresariales y para la web.

Flexibilidad: Java es un lenguaje de programación muy flexible que se puede utilizar para desarrollar una amplia gama de aplicaciones, desde aplicaciones móviles hasta aplicaciones empresariales y de escritorio. Además, Java tiene una gran cantidad de bibliotecas y frameworks que facilitan el desarrollo de aplicaciones.

Comunidad activa: Java tiene una gran comunidad de desarrolladores activa y dedicada que comparte conocimientos y recursos a través de foros en línea, blogs, grupos de discusión y redes sociales. Esto significa que siempre hay una gran cantidad de recursos y soporte disponibles para aquellos que están aprendiendo Java.

El primer programa en Java

```
public class MiPrograma {  
    public static void main(String[] args) {  
        System.out.println("¡Hola, mundo!");  
    }  
}
```

Este es un programa simple que imprime "¡Hola, mundo!" en la consola. La función "main" es la función principal que se ejecuta cuando ejecutas el programa.

1. Una vez que hayas terminado de escribir tu código, puedes ejecutar el programa haciendo clic derecho en el archivo Java en el lado izquierdo y seleccionando "Run" (Ejecutar).

¡Y listo! Ahora has creado y ejecutado tu primer programa en Java utilizando IntelliJ.

¿Qué es lo que acabamos de hacer?

```
public class MiPrograma {
```

Esta línea define una clase pública llamada "MiPrograma". En Java, un programa debe tener al menos una clase pública con un método "main" que se ejecutará cuando se inicie el programa.

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

Este es el método principal de la clase, el cual se llama "main". Este método es el punto de entrada para el programa y se ejecuta cuando se inicia. La línea también define que el método es público y estático. Además, recibe un parámetro llamado "args" que es un array de cadenas (String).

```
        System.out.println("¡Hola, mundo!");
```

Esta línea imprime "¡Hola, mundo!" en la consola. La función "println" es un método que pertenece a la clase "System" y muestra el texto que le pasamos como argumento en la pantalla seguido de un salto de línea.

```
    }
```

Estas líneas cierran el método "main" y la clase "MiPrograma".

La salida estándar

En Java, se utiliza la clase System y su método out para imprimir resultados por la salida estándar. El método out es estático, lo que significa que podemos usarlo directamente en cualquier parte de nuestro código sin tener que crear una instancia de la clase System.

System.out.println()

Para imprimir en la consola, utilizamos el método `println()`, que significa "imprimir línea". El método `println()` añade un salto de línea después de la salida, lo que significa que la próxima salida aparecerá en la línea siguiente.

```
System.out.println("Hola, mundo!");
```

En este ejemplo, estamos utilizando el método `println()` para imprimir la cadena "Hola, mundo!" en la consola. Al ejecutar este programa, se imprimirá "Hola, mundo!" en la consola.

System.out.print()

También podemos imprimir variables utilizando `println()`. Por ejemplo:

```
int edad = 25;
System.out.println("La edad es: " + edad);
```

En este ejemplo, estamos imprimiendo la variable `edad` en la consola. Utilizamos el operador de concatenación (+) para combinar la cadena "La edad es: " con el valor de la variable `edad`.

También podemos imprimir resultados de expresiones utilizando `println()`.

```
int a = 5;
int b = 3;
int resultado = a + b;
System.out.println("El resultado de la suma es: " + resultado);
```

En este ejemplo, estamos imprimiendo el resultado de una expresión (`a + b`) en la consola. Utilizamos el operador de concatenación (+) para combinar la cadena "El resultado de la suma es: " con el valor de la variable `resultado`.

Además del método `println()`, también podemos utilizar el método `print()`, que significa "imprimir". El método `print()` funciona de la misma manera que `println()`, pero no añade un salto de línea después de la salida.

```
System.out.print("Hola, ");
System.out.println("mundo!");
```

En este ejemplo, estamos imprimiendo dos cadenas separadas utilizando los métodos `print()` y `println()`. Al ejecutar este programa, se imprimirá "Hola, mundo!" en la misma línea de la consola.

System.out.printf()

Permite formatear la salida con formato específico, similar a `printf()` en C.

Utiliza marcadores de posición en la cadena de formato (%d para enteros, %f para flotantes, %s para cadenas, etc.).

Es útil para controlar la presentación de los datos.

```
public class SalidaBasica {
    public static void main(String[] args) {
        String nombre = "Juan";
        int edad = 25;
        double altura = 1.75;

        // Imprimir con formato
        System.out.printf("Nombre: %s, Edad: %d, Altura: %.2f\n",
nombre, edad, altura);
    }
}
```

System.out.write()

Escribe un solo carácter en la salida estándar.

No es tan comúnmente usado como las anteriores.

```
public class SalidaBasica {  
    public static void main(String[] args) {  
        // Imprimir un carácter  
        System.out.write('A');  
    }  
}
```

Las variables

Las variables son espacios de memoria en un programa de computadora que se utilizan para almacenar valores temporales o permanentes. Puedes pensar en una variable como un contenedor que puede almacenar diferentes tipos de información, como números, letras, palabras, entre otros.

En Java, las variables se declaran especificando el tipo de datos que contendrán, seguido del nombre que le daremos a la variable. Por ejemplo, si queremos almacenar un número entero en una variable llamada "edad", podemos escribir lo siguiente:

```
int edad = 25;
```

Aquí, hemos declarado una variable de tipo "int" (entero) llamada "edad" y le hemos asignado un valor de 25. La variable puede ser utilizada más adelante en el programa, ya sea para realizar cálculos o para imprimir el valor almacenado en la consola.

También es importante mencionar que en Java, una vez que se ha declarado una variable, podemos cambiar su valor en cualquier momento del programa. Por ejemplo, si queremos cambiar el valor de la variable "edad" a 30, podemos escribir lo siguiente:

```
edad = 30;
```

Tipos de datos

Los tipos de datos en Java son una forma de clasificar y definir el tipo de valor que una variable puede contener. En Java, existen varios tipos de datos predefinidos que se pueden usar para declarar variables, cada uno con un rango y un tamaño específico.

A continuación, un ejemplo de cómo se declaran variables utilizando diferentes tipos de datos en Java:

```
// Declaración de variables enteras
byte edad = 26;
short numero = 10000;
int entero = 300000000;
long grande = 90000000000L;

// Declaración de variables flotantes
float flotante = 3.14f;
double doble = 3.141592;

// Declaración de variables de caracteres
char letra = 'a';

// Declaración de variables booleanas
boolean verdadero = true;
boolean falso = false;
```

Tipos de datos primitivos

Los tipos de datos primitivos son los tipos de datos más básicos que se utilizan en la programación y que están predefinidos en el lenguaje de programación. En Java, existen ocho tipos de datos primitivos que se utilizan para definir variables y constantes.

Los tipos de datos primitivos son útiles porque son simples y eficientes en términos de uso de memoria y velocidad de procesamiento. Además, son fundamentales en la programación ya que se utilizan para crear variables y estructuras de datos más complejas.

En Java, los tipos de datos primitivos son aquellos tipos de datos básicos que están integrados en el lenguaje y no son objetos. Estos tipos de datos se utilizan para definir variables y almacenar valores simples. Los tipos de datos primitivos en Java se dividen en cuatro categorías: enteros, flotantes, caracteres y booleanos.

A continuación, se explican cada uno de los tipos de datos primitivos de Java, junto con sus características y ejemplos de uso:

Enteros

Los tipos de datos enteros en Java se utilizan para almacenar números enteros sin decimales. Hay cuatro tipos de datos enteros en Java, cada uno con un tamaño diferente:

byte

Este tipo de datos se utiliza para almacenar números enteros en el rango de -128 a 127.

```
byte edad = 25;
```

short

Este tipo de datos se utiliza para almacenar números enteros en el rango de -32,768 a 32,767.

```
short numero = 10000;
```

int

Este tipo de datos se utiliza para almacenar números enteros en el rango de -2,147,483,648 a 2,147,483,647. Es el tipo de datos más comúnmente utilizado para almacenar enteros.

```
int entero = 1000000;
```

long

Este tipo de datos se utiliza para almacenar números enteros en un rango más amplio que int, de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807. Para declarar un valor long, se debe agregar una "L" al final del número.

```
long grande = 123456789L;
```

Flotantes

Los tipos de datos flotantes en Java se utilizan para almacenar números con decimales. Hay dos tipos de datos flotantes en Java:

float

Este tipo de datos se utiliza para almacenar números de punto flotante en un rango de aproximadamente 3.4×10^{-38} a 3.4×10^{38} . Para declarar un valor float, se debe agregar una "f" al final del número.

```
float flotante = 3.14f;
```

double

Este tipo de datos se utiliza para almacenar números de punto flotante en un rango de aproximadamente 1.7×10^{-308} a 1.7×10^{308} . Es el tipo de datos más comúnmente utilizado para números con decimales.

```
double doble = 3.141592;
```

Caracteres

El tipo de datos `char` en Java se utiliza para almacenar un solo carácter Unicode. Este tipo de datos se utiliza para representar caracteres individuales, como letras y números.

```
char letra = 'a';
```

Booleanos

El tipo de datos `boolean` en Java se utiliza para almacenar valores verdadero o falso. Este tipo de datos es útil para declaraciones condicionales y lógicas.

```
boolean verdadero = true;  
boolean falso = false;
```

Resumiendo en una tabla

Tipo	Grupo	Tamaño	Rango
<code>boolean</code>	Lógico	1 bit	true o false
<code>byte</code>	Entero	8 bits	-128 a 127
<code>char</code>	Carácter	16 bits	'\u0000' a '\uffff' (0 a 65535)
<code>short</code>	Entero	16 bits	-32768 a 32767
<code>int</code>	Entero	32 bits	-2147483648 a 2147483647
<code>long</code>	Entero	64 bits	-9223372036854775808 a 9223372036854775807
<code>float</code>	Real	32 bits	-3.4028235E38 a -1.4E-45 y de +1.4E-45 a +3.4028235E38
<code>double</code>	Real	64 bits	-1.7976931348623157E308 a -4.9E-324 y de +4.9E-324 a +1.7976931348623157E308

Operadores

En programación, los operadores son elementos fundamentales para realizar cálculos y manipular datos. En Java, existen diversos tipos de operadores que se utilizan en combinación con los tipos de datos para crear expresiones que realicen una tarea específica. A continuación, se describen los operadores más comunes en Java y se presentan algunos ejemplos de cómo se pueden utilizar para crear expresiones.

Operadores Aritméticos:

Los operadores aritméticos son aquellos que se utilizan en Java para realizar operaciones matemáticas básicas como sumas, restas, multiplicaciones y divisiones entre valores numéricos. Los operadores aritméticos en Java son los siguientes:

Suma (+)

El operador suma se utiliza para sumar dos o más valores numéricos.

```
int resultado = 5 + 3;  
System.out.println(resultado); // imprime 8
```

Resta (-)

El operador resta se utiliza para restar dos valores numéricos.

```
int resultado = 7 - 4;  
System.out.println(resultado); // imprime 3
```

Multiplicación (*)

El operador multiplicación se utiliza para multiplicar dos o más valores numéricos.

```
int resultado = 3 * 4;  
System.out.println(resultado); // imprime 12
```

División (/)

El operador división se utiliza para dividir dos valores numéricos.

```
int resultado = 12 / 4;  
System.out.println(resultado); // imprime 3
```

Cabe destacar que, en la división, si uno de los operandos es un número decimal, el resultado será un número decimal. Además, si se intenta dividir por cero, se producirá una excepción de tipo `ArithmeticException`.

Módulo (%)

El operador módulo se utiliza para obtener el resto de la división entre dos valores numéricos.

```
int resultado = 13 % 4;  
System.out.println(resultado); // imprime 1
```

Este operador es muy útil para comprobar si un número es par o impar, ya que si un número es par, su resto al dividirlo por 2 será 0, mientras que si es impar, su resto será 1.

Incremento (++) y Decremento (--)

Los operadores de incremento y decremento se utilizan para aumentar o disminuir en una unidad el valor de una variable numérica.

```
int i = 5;  
i++; // incrementa i en una unidad  
System.out.println(i); // imprime 6  
  
int j = 8;  
j--; // decrementa j en una unidad  
System.out.println(j); // imprime 7
```

Es importante tener en cuenta que el operador de incremento (++) y el operador de decremento (--) pueden utilizarse tanto en la forma prefija como en la forma posfija, como se muestra a continuación:

```
int i = 5;  
++i; // incrementa i en una unidad (forma prefija)  
System.out.println(i); // imprime 6  
  
int j = 8;  
j--; // decrementa j en una unidad (forma posfija)  
System.out.println(j); // imprime 7
```

En general, los operadores aritméticos en Java se utilizan para realizar operaciones matemáticas básicas en los programas, por lo que son fundamentales en el desarrollo de aplicaciones en este lenguaje.

Operadores de Comparación:

Los operadores de comparación en Java se utilizan para comparar dos valores y obtener un resultado booleano (verdadero o falso) como respuesta. Aquí hay algunos de los operadores de comparación más comunes en Java:

Igualdad (==)

Este operador comprueba si dos valores son iguales o no. Si los valores son iguales, entonces devuelve verdadero, de lo contrario, devuelve falso.

```
int a = 5;  
int b = 7;  
boolean resultado = (a == b); // resultado es falso
```

Desigualdad (!=)

Este operador comprueba si dos valores son diferentes o no. Si los valores son diferentes, entonces devuelve verdadero, de lo contrario, devuelve falso.

```
int a = 5;
```

```
int b = 7;
boolean resultado = (a != b); // resultado es verdadero
```

Mayor que (>)

Este operador comprueba si el primer valor es mayor que el segundo valor. Si el primer valor es mayor que el segundo valor, entonces devuelve verdadero, de lo contrario, devuelve falso.

```
int a = 5;
int b = 7;
boolean resultado = (a < b); // resultado es verdadero
```

Menor que (<)

Este operador comprueba si el primer valor es menor que el segundo valor. Si el primer valor es menor que el segundo valor, entonces devuelve verdadero, de lo contrario, devuelve falso.

```
int a = 5;
int b = 7;
boolean resultado = (a < b); // resultado es verdadero
```

Mayor o igual que (>=)

Este operador comprueba si el primer valor es mayor o igual que el segundo valor. Si el primer valor es mayor o igual que el segundo valor, entonces devuelve verdadero, de lo contrario, devuelve falso.

```
int a = 5;
int b = 7;
boolean resultado = (a >= b); // resultado es falso
```

Menor o igual que (<=)

Este operador comprueba si el primer valor es menor o igual que el segundo valor. Si el primer valor es menor o igual que el segundo valor, entonces devuelve verdadero, de lo contrario, devuelve falso.

```
int a = 5;
int b = 7;
boolean resultado = (a <= b); // resultado es verdadero
```

Estos son solo algunos de los operadores de comparación más comunes en Java. Es importante recordar que el resultado de estos operadores siempre será un valor booleano (verdadero o falso) y se pueden usar en combinación con instrucciones condicionales para tomar decisiones en su programa.

Operadores Lógicos:

Los operadores lógicos son aquellos que se utilizan para comparar dos expresiones booleanas y devolver un resultado que también es un valor booleano (true o false). Estos operadores son muy útiles en la programación ya que nos permiten tomar decisiones basadas en múltiples condiciones.

A continuación se presentan los tres operadores lógicos principales en Java:

Operador AND (&&)

Este operador devuelve true si y sólo si ambas expresiones booleanas son verdaderas.

```
int x = 5;
int y = 10;
if (x > 0 && y > 0) {
    System.out.println("Ambos números son positivos");
}
```

En este ejemplo, la condición dentro del if sólo se cumplirá si ambas variables son mayores que cero.

Operador OR (||)

Este operador devuelve true si al menos una de las expresiones booleanas es verdadera.

```
int x = 5;
int y = -10;
if (x > 0 || y > 0) {
    System.out.println("Al menos uno de los números es positivo");
}
```

En este ejemplo, la condición dentro del if se cumplirá si al menos una de las variables es mayor que cero.

Operador NOT (!)

Este operador se utiliza para negar una expresión booleana, es decir, si la expresión es verdadera, el operador NOT la convierte en falsa y viceversa.

```
boolean x = true;
if (!x) {
    System.out.println("La variable x es falsa");
} else {
    System.out.println("La variable x es verdadera");
}
```

En este ejemplo, el operador NOT se utiliza para negar el valor de la variable booleana x. Si el valor de x es verdadero, la condición dentro del if no se cumplirá y se imprimirá "La variable x es verdadera".

Es importante recordar que los operadores lógicos tienen una jerarquía de precedencia, es decir, se evalúan en un orden específico. Por lo general, se evalúan primero los operadores NOT, seguidos de AND y finalmente OR. Siempre es recomendable utilizar paréntesis para asegurarse de que las condiciones se evalúan en el orden correcto.

Operadores de Asignación:

Los operadores de asignación en Java se utilizan para asignar un valor a una variable. En Java, los operadores de asignación tienen la siguiente sintaxis:

```
variable operador= expresión;
```

Donde "variable" es el nombre de la variable a la que se le va a asignar un valor, "operador" es el operador de asignación y "expresión" es la expresión que se evalúa para asignar un valor a la variable.

A continuación se presentan algunos ejemplos de operadores de asignación:

Operador "="

Este es el operador de asignación básico que se utiliza para asignar un valor a una variable.

```
int x = 10;
```

En este caso, se está asignando el valor 10 a la variable "x" de tipo entero.

Operador "+="

Este operador se utiliza para sumar un valor a una variable y asignar el resultado a la misma variable.

```
int x = 5;
x += 3;
```

En este caso, se está sumando 3 a la variable "x" y asignando el resultado (8) a la misma variable.

Operador "-="

Este operador se utiliza para restar un valor a una variable y asignar el resultado a la misma variable.

```
int x = 10;  
x -= 4;
```

En este caso, se está restando 4 a la variable "x" y asignando el resultado (6) a la misma variable.

Operador "*="

Este operador se utiliza para multiplicar una variable por un valor y asignar el resultado a la misma variable.

```
int x = 2;  
x *= 5;
```

En este caso, se está multiplicando la variable "x" por 5 y asignando el resultado (10) a la misma variable.

Operador "/="

Este operador se utiliza para dividir una variable por un valor y asignar el resultado a la misma variable.

```
int x = 10;  
x /= 2;
```

En este caso, se está dividiendo la variable "x" entre 2 y asignando el resultado (5) a la misma variable.

Operador "%="

Este operador se utiliza para obtener el resto de la división de una variable por un valor y asignar el resultado a la misma variable.

```
int x = 11;  
x %= 3;
```

En este caso, se está obteniendo el resto de la división de la variable "x" entre 3 y asignando el resultado (2) a la misma variable.

La estructura de control de flujo

Estructuras Condicionales:

Las estructuras condicionales son aquellas que nos permiten controlar el flujo de ejecución del programa basándose en una o más condiciones. Las estructuras condicionales en Java son:

if

Es la estructura condicional más básica. Permite ejecutar un bloque de código si se cumple una condición determinada.

```
int num = 10;  
  
if(num > 0){  
    System.out.println("El número es positivo");  
}
```

if-else

Permite ejecutar un bloque de código si se cumple una condición y otro bloque de código si no se cumple la condición.

```
int num = -5;  
  
if(num > 0){  
    System.out.println("El número es positivo");  
}  
else{
```

```
        System.out.println("El número es negativo o cero");
    }
```

if-else-if

Permite evaluar varias condiciones consecutivas y ejecutar el bloque de código correspondiente a la primera condición que se cumpla.

```
int num = 10;

if(num < 0){
    System.out.println("El número es negativo");
}
else if(num == 0){
    System.out.println("El número es cero");
}
else{
    System.out.println("El número es positivo");
}
```

Estructura switch-case o condicional múltiple

La estructura switch-case nos permite evaluar una expresión y ejecutar un bloque de código correspondiente a cada posible valor de la expresión.

```
int num = 3;

switch(num){
    case 1:
        System.out.println("El número es uno");
        break;
    case 2:
```

```
        System.out.println("El número es dos");
        break;
    case 3:
        System.out.println("El número es tres");
        break;
    default:
        System.out.println("El número no es uno, dos ni tres");
}
```

Estructuras de Repetición o bucles

Las estructuras de repetición nos permiten ejecutar un bloque de código varias veces, mientras se cumpla una determinada condición. En Java, existen dos estructuras de repetición:

while:

Permite ejecutar un bloque de código mientras se cumpla una condición.

```
int num = 0;

while(num < 10){
    System.out.println("El valor de num es: " + num);
    num++;
}
```

for

Permite ejecutar un bloque de código un número determinado de veces.

```
for(int i = 0; i < 10; i++){
    System.out.println("El valor de i es: " + i);
}
```

Arrays

Los arrays son una forma de almacenar múltiples valores del mismo tipo en una sola variable. Es como si tuvieras una caja con muchos compartimentos, donde cada compartimento puede contener un valor diferente. Para acceder a un valor específico en el array, puedes hacerlo utilizando un índice numérico.

Por ejemplo, si quisieras almacenar los nombres de tus amigos en una sola variable, podrías usar un array. Aquí hay un ejemplo de cómo declarar un array de nombres en Java:

```
String[] nombres = {"Juan", "Ana", "Pedro", "Maria"};
```

En este ejemplo, `nombres` es el nombre del array y `String` es el tipo de datos que se almacenará en el array. Los valores que se almacenarán en el array están dentro de las llaves {} y están separados por comas.

Para acceder a un valor específico en el array, debes utilizar su índice numérico. El primer valor en el array tiene un índice de 0, el segundo valor tiene un índice de 1, y así sucesivamente.

Aquí hay un ejemplo de cómo imprimir el tercer nombre en el array:

```
System.out.println(nombres[2]);
```

Este código imprimirá Pedro en la consola, ya que Pedro es el tercer valor en el array y su índice es 2.

Los arrays también se pueden utilizar para almacenar valores numéricos, booleanos y otros tipos de datos en Java. Es una herramienta útil para almacenar y manipular datos de manera eficiente.

Características principales de los Arrays

- **Son estructuras de datos estáticas:** una vez que se define el tamaño del array, éste no puede ser modificado durante la ejecución del programa.
- **Contienen elementos del mismo tipo de datos:** todos los elementos del array deben ser del mismo tipo de datos, ya sea numérico, booleano, carácter o String.
- **Son objetos:** en Java, los Arrays son objetos y tienen un identificador de objeto que se puede utilizar para acceder a sus elementos.
- **Acceso a los elementos:** los elementos de un array se acceden mediante un índice numérico, empezando por el índice cero (0).
- **Capacidad limitada:** la capacidad de almacenamiento del array está limitada por la cantidad de memoria disponible en el sistema.
- **Los elementos del array están ordenados:** los elementos de un array están ordenados secuencialmente, lo que permite realizar operaciones de búsqueda y ordenación.
- **Se pueden inicializar en la declaración o mediante un bucle:** los elementos de un array se pueden inicializar al declarar el array o mediante un bucle.

Formas de inicializar un array

Inicialización con valores explícitos

```
// Inicialización de un array numérico
int[] numeros = {1, 2, 3, 4, 5};

// Inicialización de un array de strings
String[] nombres = {"Juan", "Pedro", "Ana", "Lucía"};
```

Inicialización con constructor

```
// Inicialización de un array numérico
int[] numeros = new int[]{1, 2, 3, 4, 5};

// Inicialización de un array de strings
String[] nombres = new String[]{"Juan", "Pedro", "Ana", "Lucía"};
```

Inicialización con tamaño y asignación de valores posterior

```
// Inicialización de un array numérico
int[] numeros = new int[5];
numeros[0] = 1;
numeros[1] = 2;
numeros[2] = 3;
numeros[3] = 4;
numeros[4] = 5;

// Inicialización de un array de strings
String[] nombres = new String[4];
nombres[0] = "Juan";
nombres[1] = "Pedro";
nombres[2] = "Ana";
nombres[3] = "Lucía";
```

Es importante mencionar que en la inicialización con valores explícitos y con constructor, el tamaño del array es determinado automáticamente por la cantidad de valores que se le asignen. Mientras que en la inicialización con tamaño y asignación de valores posterior, se debe especificar el tamaño del array y asignar los valores uno por uno posteriormente.

Recorrer un Array

Recorrer un array en Java significa acceder a cada uno de sus elementos para realizar alguna acción o realizar alguna operación sobre ellos. Para recorrer un array se utilizan bucles, en particular, el bucle for es el más comúnmente utilizado.

La sintaxis básica para recorrer un array utilizando un bucle for es la siguiente:

```
for (int i = 0; i < array.length; i++) {  
    // Operación a realizar con el elemento del array en la  
    // posición i  
}
```

En esta sintaxis, array es el nombre del array que se desea recorrer y array.length es la propiedad que devuelve la longitud del array. El bucle for comienza en la posición 0 y continúa hasta que la variable i sea menor que la longitud del array. En cada iteración del bucle, se realiza la operación con el elemento del array en la posición i.

A continuación, se presenta un ejemplo de cómo recorrer un array de enteros en Java utilizando un bucle for:

```
int[] numeros = {2, 4, 6, 8, 10};  
  
for (int i = 0; i < numeros.length; i++) {  
    System.out.println("El elemento en la posición " + i + " es: " + numeros[i]);  
}
```

En este ejemplo, se declara un array de enteros llamado numeros con 5 elementos. Luego, se utiliza un bucle for para recorrer el array y se imprime el elemento en cada posición del array junto con su índice en la salida estándar utilizando el método System.out.println().

¿Cómo mezclar dos arrays?

Para mezclar los dos arrays, podemos crear un tercer array y utilizar un bucle for para intercalar los elementos de cada array. Aquí te muestro un ejemplo de cómo hacerlo:

```
int[] numeros = {1, 2, 3, 4, 5};  
String[] nombres = {"Juan", "Pedro", "Ana", "Lucía", "Carla"};  
  
// Creamos un tercer array con tamaño igual a la suma de los dos  
// arrays originales  
  
Object[] mezcla = new Object[numeros.length + nombres.length];  
  
// Usamos un contador para saber en qué posición insertar los  
// elementos de cada array  
  
int contador = 0;  
  
// Recorremos los arrays y vamos intercalando los elementos  
  
for (int i = 0; i < numeros.length && i < nombres.length; i++) {  
    mezcla[contador++] = numeros[i];  
    mezcla[contador++] = nombres[i];  
}  
  
// Si uno de los arrays es más largo que el otro, añadimos los  
// elementos restantes al final del array de mezcla  
  
if (numeros.length > nombres.length) {  
    for (int i = nombres.length; i < numeros.length; i++) {  
        mezcla[contador++] = numeros[i];  
    }  
} else {
```

```
for (int i = numeros.length; i < nombres.length; i++) {  
    mezcla[contador++] = nombres[i];  
}  
  
// Imprimimos el resultado  
  
System.out.println(Arrays.toString(mezcla));  
// [1, Juan, 2, Pedro, 3, Ana, 4, Lucía, 5, Carla]
```

En este ejemplo, creamos un tercer array mezcla con tamaño igual a la suma de los tamaños de los dos arrays originales. Luego, utilizamos un contador para ir insertando los elementos de cada array en el array de mezcla. En el bucle for, intercalamos los elementos de cada array en cada posición par e impar del array de mezcla. Por último, si uno de los arrays es más largo que el otro, añadimos los elementos restantes al final del array de mezcla.

Arrays Multidimensionales

Son arreglos que almacenan elementos en dos o más dimensiones. Estos arreglos pueden visualizarse como matrices, donde cada dimensión adicional agrega una nueva "capa" de elementos.

Declaración e inicialización de un arreglo multidimensional:

En Java, puedes declarar e inicializar un arreglo multidimensional de la siguiente manera:

Declaración e inicialización de un arreglo bidimensional

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

Declaración e inicialización de un arreglo tridimensional

```
int[][][] cubo = new int[3][3][3];
```

En el ejemplo anterior, `matriz` es un arreglo bidimensional de enteros con dimensiones 3x3, mientras que `cubo` es un arreglo tridimensional de enteros con dimensiones 3x3x3.

Acceso a los elementos de un arreglo multidimensional:

Para acceder a un elemento específico en un arreglo multidimensional, necesitas especificar los índices correspondientes a cada dimensión del arreglo.

Acceso a un elemento en un arreglo bidimensional

```
int valor = matriz[1][2]; // Obtiene el elemento en la fila 1 y  
columna 2
```

Acceso a un elemento en un arreglo tridimensional

```
int valor = cubo[0][1][2]; // Obtiene el elemento en la capa 0,  
fila 1 y columna 2
```

Inicialización de un arreglo multidimensional con valores:

Puedes inicializar un arreglo multidimensional con valores directamente durante la declaración.

Inicialización de un arreglo bidimensional

```
int[][] matriz = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

Inicialización de un arreglo tridimensional

```
int[][][] cubo = {
    {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}},
    {{10, 11, 12}, {13, 14, 15}, {16, 17, 18}},
    {{19, 20, 21}, {22, 23, 24}, {25, 26, 27}}
};
```

Iteración sobre un arreglo multidimensional:

Puedes utilizar bucles anidados para iterar sobre todos los elementos de un arreglo multidimensional.

Iteración sobre un arreglo bidimensional

```
for (int i = 0; i < matriz.length; i++) {
    for (int j = 0; j < matriz[i].length; j++) {
        System.out.print(matriz[i][j] + " ");
    }
    System.out.println();
}
```

Iteración sobre un arreglo tridimensional

```
for (int i = 0; i < cubo.length; i++) {
    for (int j = 0; j < cubo[i].length; j++) {
        for (int k = 0; k < cubo[i][j].length; k++) {
            System.out.print(cubo[i][j][k] + " ");
        }
        System.out.println();
    }
    System.out.println();
}
```

String

Un String es una secuencia de caracteres que se utiliza para representar texto. Es una clase predefinida en el lenguaje y, a diferencia de los tipos de datos primitivos como int o double, es un objeto.

Para crear un String en Java, se utiliza el operador de asignación = y se encierra el texto entre comillas dobles ". Por ejemplo:

```
String saludo = "Hola mundo!";
```

En este ejemplo, se ha creado un String llamado saludo con el valor "Hola mundo!".

Los String en Java son inmutables, lo que significa que no se pueden cambiar una vez que se han creado. Por ejemplo, si se intenta cambiar el valor de un String, se crea un nuevo objeto String con el nuevo valor.

```
String saludo = "Hola";
saludo = saludo + " mundo!";
```

En este caso, se ha creado un nuevo String con el valor "Hola mundo!" y se ha asignado a la variable saludo. El objeto String original con el valor "Hola" ya no es accesible y será eventualmente eliminado por el recolector de basura.

Recorrer un String

Para recorrer un String en Java, se puede utilizar un bucle for que itere a través de cada carácter del String.

```
String saludo = "Hola mundo!";
for (int i = 0; i < saludo.length(); i++) {
    char c = saludo.charAt(i);
```

```
        System.out.println(c);
    }
```

En este ejemplo, se ha utilizado el método `length()` del `String` para obtener el número de caracteres en el `String`. Luego se utiliza un bucle `for` para iterar a través de cada índice del `String`, utilizando el método `charAt()` para obtener el carácter en ese índice. Por último, se utiliza el método `println()` para imprimir cada carácter en una línea separada.

Concatenación de String

La concatenación de cadenas de texto (strings) es una operación muy común en la programación en Java. En términos simples, la concatenación se refiere a la unión de dos o más cadenas de texto para formar una sola cadena de texto.

En Java, puedes concatenar cadenas de texto utilizando el operador `"+"`, que actúa como un operador de concatenación. El operador `"+"` se utiliza para unir dos o más cadenas de texto en una sola cadena.

Por ejemplo, si queremos concatenar las cadenas "Hola" y "Mundo" para formar la cadena "Hola Mundo", podemos hacerlo de la siguiente manera:

```
String cadena1 = "Hola";
String cadena2 = "Mundo";
String concatenada = cadena1 + " " + cadena2;
System.out.println(concatenada); // imprimirá "Hola Mundo"
```

En este ejemplo, hemos creado dos variables de tipo `String` llamadas `"cadena1"` y `"cadena2"` y las hemos inicializado con las cadenas "Hola" y "Mundo", respectivamente. Luego, hemos creado otra variable de tipo `String` llamada `"concatenada"` y hemos asignado el resultado de la concatenación de `"cadena1"`, un espacio en blanco y `"cadena2"` utilizando el operador `"+"`. Finalmente, hemos impreso la cadena concatenada utilizando el método `"println()"` de la clase `System`.

También puedes utilizar el método `"concat()"` de la clase `String` para concatenar cadenas de texto en Java. Este método toma una cadena de texto como argumento y la agrega al final de la cadena actual. Por ejemplo:

```
String cadena1 = "Hola";
String cadena2 = "Mundo";
String concatenada = cadena1.concat(" ").concat(cadena2);
System.out.println(concatenada); // imprimirá "Hola Mundo"
```

En este ejemplo, hemos utilizado el método "concat()" dos veces para concatenar las cadenas "cadena1" y "cadena2" y agregar un espacio en blanco entre ellas.

Es importante recordar que la concatenación de cadenas de texto en Java puede consumir una cantidad significativa de recursos de memoria y tiempo de ejecución, especialmente cuando se trabaja con grandes cantidades de datos. Por lo tanto, es importante optimizar el uso de la concatenación de cadenas de texto en tus programas.

Métodos y operaciones principales de los Strings

Concatenar Strings

Para concatenar dos Strings en Java, se puede usar el operador +. Aquí hay un ejemplo:

```
String s1 = "Hola";
String s2 = "mundo";
String s3 = s1 + " " + s2;
System.out.println(s3);
```

Este código imprimirá "Hola mundo". En el ejemplo, s1 y s2 son Strings y se concatenan usando el operador +. Luego, el resultado se asigna a s3.

Es importante tener en cuenta que cuando se concatena una cadena con otro tipo de datos, Java convierte implícitamente los otros tipos de datos en Strings.

```
int edad = 20;
String mensaje = "Tengo " + edad + " años";
System.out.println(mensaje);
```

Este código imprimirá "Tengo 20 años". En el ejemplo, edad es un entero y se concatena con la cadena "Tengo " y la cadena " años" usando el operador +. Java convierte automáticamente la variable edad en una cadena antes de concatenarla.

Comparar Strings

Para comparar dos objetos de tipo String se utilizan los métodos equals() y equalsIgnoreCase(). El método equals() compara dos cadenas de caracteres y devuelve un valor booleano que indica si son iguales o no, teniendo en cuenta las mayúsculas y minúsculas. Por otro lado, el método equalsIgnoreCase() compara dos cadenas de caracteres y devuelve un valor booleano que indica si son iguales o no, ignorando las mayúsculas y minúsculas.

A continuación, se muestran algunos ejemplos de cómo comparar cadenas de caracteres en Java:

```
String cadena1 = "Hola";
String cadena2 = "hola";
String cadena3 = "Hola";

// Comparar dos cadenas con el método equals()
if (cadena1.equals(cadena2)) {
    System.out.println("cadena1 y cadena2 son iguales");
} else {
    System.out.println("cadena1 y cadena2 son diferentes");
}

if (cadena1.equals(cadena3)) {
    System.out.println("cadena1 y cadena3 son iguales");
} else {
```

```
System.out.println("cadena1 y cadena3 son diferentes");
}

// Comparar dos cadenas con el método equalsIgnoreCase()
if (cadena1.equalsIgnoreCase(cadena2)) {
    System.out.println("cadena1 y cadena2 son iguales ignorando
mayúsculas y minúsculas");
} else {
    System.out.println("cadena1 y cadena2 son diferentes incluso
ignorando mayúsculas y minúsculas");
}
```

En el primer ejemplo, la primera cadena es "Hola" y la segunda es "hola". Como son cadenas de caracteres diferentes debido a las mayúsculas y minúsculas, la salida del programa será "cadena1 y cadena2 son diferentes".

En el segundo ejemplo, la primera cadena es "Hola" y la tercera también es "Hola". Como son cadenas de caracteres idénticas, la salida del programa será "cadena1 y cadena3 son iguales".

En el tercer ejemplo, la primera cadena es "Hola" y la segunda es "hola". Como son cadenas de caracteres diferentes debido a las mayúsculas y minúsculas, pero se utiliza el método `equalsIgnoreCase()`, que no tiene en cuenta las mayúsculas y minúsculas, la salida del programa será "cadena1 y cadena2 son iguales ignorando mayúsculas y minúsculas".

Tamaño de un String

Para conocer el tamaño de un String en Java se puede utilizar el método `length()`. Este método devuelve la cantidad de caracteres que tiene el String.

Aquí hay un ejemplo de cómo utilizarlo:

```
String mensaje = "Hola, mundo!";
int tamaño = mensaje.length();
System.out.println("El tamaño del mensaje es: " + tamaño);
```

En este ejemplo, el método `length()` se utiliza para obtener el tamaño del String `mensaje`. Luego, se almacena el resultado en la variable `tamaño` y se muestra por consola el mensaje "El tamaño del mensaje es:" seguido del valor de `tamaño`. En este caso, el tamaño del mensaje es 12, ya que hay 12 caracteres en "Hola, mundo!".

Pasar un String a mayúscula o minúscula

En Java, para pasar un String a mayúscula o minúscula, podemos utilizar los métodos `toUpperCase()` y `toLowerCase()`, respectivamente.

```
String mensaje = "Hola Mundo!";
String mensajeMayusculas = mensaje.toUpperCase(); // HOLA MUNDO!
String mensajeMinusculas = mensaje.toLowerCase(); // hola mundo!
```

El método `toUpperCase()` convierte todas las letras del String a mayúsculas, mientras que el método `toLowerCase()` convierte todas las letras a minúsculas.

También podemos utilizar los métodos `toUpperCase(Locale)` y `toLowerCase(Locale)` para especificar una localización específica en la que se aplicará la conversión de mayúsculas y minúsculas.

```
String mensaje = "Hola Mundo!";
String mensajeMayusculas = mensaje.toUpperCase(Locale.ENGLISH);
// HOLA MUNDO!
String mensajeMinusculas = mensaje.toLowerCase(Locale.ENGLISH);
// hola mundo!
```

En este caso, se utiliza la localización `Locale.ENGLISH`, lo que significa que se aplicará la conversión de mayúsculas y minúsculas según las reglas de la lengua inglesa.

Indicar si contiene otro string

Puedes utilizar el método contains de la clase String para determinar si un String contiene otro String. Este método devuelve true si el String objetivo contiene el String especificado y false en caso contrario. Aquí hay un ejemplo:

```
String frase = "El perro marrón salió corriendo";
if (frase.contains("perro")) {
    System.out.println("La frase contiene la palabra 'perro'");
} else {
    System.out.println("La frase no contiene la palabra 'perro'");
}
```

En este ejemplo, la salida será: "La frase contiene la palabra 'perro'".

También puedes utilizar el método indexOf de la clase String para buscar la posición de un String dentro de otro String. Si el String objetivo no contiene el String especificado, el método devuelve -1. Aquí hay un ejemplo:

```
String frase = "El perro marrón salió corriendo";
int posicion = frase.indexOf("marrón");
if (posicion != -1) {
    System.out.println("La palabra 'marrón' comienza en la
    posición " + posicion);
} else {
    System.out.println("La palabra 'marrón' no está en la frase");
}
```

En este ejemplo, la salida será: "La palabra 'marrón' comienza en la posición 9".

Verificar si comienza o termina con otro String

Puedes usar los métodos startsWith() y endsWith() en un String para verificar si comienza o termina con otro String, respectivamente. Ambos métodos devuelven un

valor booleano (true o false) según si el String cumple o no con la condición especificada.

Aquí te dejo algunos ejemplos de cómo utilizarlos:

```
const miString = 'Hola mundo';

// Comprobar si el String comienza con 'Hola'
console.log(miString.startsWith('Hola')) // Devuelve true

// Comprobar si el String termina con 'mundo'
console.log(miString.endsWith('mundo')) // Devuelve true
```

Ten en cuenta que ambos métodos son sensibles a mayúsculas y minúsculas, es decir, si especificas una letra en mayúscula cuando en realidad está en minúscula, el método devolverá false. Para evitar esto, puedes convertir todo el String a minúsculas o mayúsculas antes de usar los métodos.

Carácter de escape

El carácter especial \ en un String se utiliza como carácter de escape. Esto significa que se utiliza para indicar que el siguiente carácter en el String debe ser tratado de una manera especial. Algunas de las cosas que se pueden hacer con el carácter \ en un String son:

Incluir comillas dobles dentro de un String que está delimitado por comillas dobles:

```
String ejemplo = "Este es un \"ejemplo\" de uso del caracter \\";
System.out.println(ejemplo);
// Salida: Este es un "ejemplo" de uso del caracter \
```

Incluir comillas simples dentro de un String que está delimitado por comillas simples:

```
String ejemplo = 'Este es un \'ejemplo\' de uso del caracter \\';
System.out.println(ejemplo);
// Salida: Este es un 'ejemplo' de uso del caracter \
```

Incluir el carácter \ dentro de un String:

```
String ejemplo = "Este es un ejemplo de uso del caracter \\";
System.out.println(ejemplo);
// Salida: Este es un ejemplo de uso del caracter \
```

Incluir caracteres especiales, como tabulaciones y saltos de línea, dentro de un String:

```
String ejemplo = "Este es un ejemplo\tde uso del caracter \\n";
System.out.println(ejemplo);
// Salida: Este es un ejemplo      de uso del caracter
//           (se imprime una tabulación y luego se salta de línea)
```

Incluir valores hexadecimales o Unicode dentro de un String utilizando la notación \uXXXX:

```
String ejemplo = "\u00BFQu\u00E9 tal est\u00E1s?";
System.out.println(ejemplo);
// Salida: ¿Qué tal estás?
```

Funciones

En Java, una función es un bloque de código que se puede llamar desde cualquier parte del programa. También se le conoce como método, y puede aceptar parámetros de entrada y devolver un valor de salida. Las funciones son útiles para reutilizar el código y evitar tener que repetir el mismo código varias veces en diferentes partes del programa.

```
tipoDeDatoDeRetorno nombreDeLaFuncion(tipoDeDato argumento1,
tipoDeDato argumento2, ...) {
    // Cuerpo de la función
    return valorDeRetorno;
```

```
}
```

Donde tipoDeDatoDeRetorno es el tipo de dato que devuelve la función y nombreDeLaFunción es el nombre que se le da a la función. Los argumentos son los valores que se pasan a la función para que la función pueda hacer algo con ellos. El cuerpo de la función es donde se escribe el código que hace la tarea que se desea realizar y se puede acceder a los argumentos dentro del cuerpo de la función.

```
public class FuncionEjemplo {  
    public static void main(String[] args) {  
        int a = 3, b = 5;  
        int resultado = suma(a, b);  
        System.out.println("La suma de " + a + " y " + b + " es: "  
+ resultado);  
    }  
  
    public static int suma(int num1, int num2) {  
        int suma = num1 + num2;  
        return suma;  
    }  
}
```

En este ejemplo, la función suma toma dos números enteros como argumentos y devuelve la suma de esos dos números. En el método main, se llama a la función suma y se le pasan dos valores enteros a y b. El resultado devuelto por la función suma se almacena en la variable resultado y se imprime por pantalla utilizando System.out.println(). El resultado será: "La suma de 3 y 5 es: 8".

Las funciones son útiles porque permiten la reutilización del código y también ayudan a simplificar el código, ya que las tareas complejas se pueden dividir en tareas más pequeñas y manejables.

Programación Orientado a Objetos (POO)

La programación orientada a objetos (POO) es un paradigma de programación que se basa en el concepto de objetos. En la POO, se modela un problema como una colección de objetos que interactúan entre sí para resolver el problema. Cada objeto tiene un estado, que se representa mediante sus atributos, y un comportamiento, que se representa mediante sus métodos.

La POO se basa en cuatro conceptos fundamentales: encapsulamiento, herencia, polimorfismo y abstracción.

Programación estructurada vs. orientada a objetos

Programación estructurada	Programación orientada a objetos
Se basa en dividir el programa en módulos o funciones que se ejecutan secuencialmente.	Se basa en crear objetos que contienen datos y métodos que interactúan entre sí.
Se enfoca en el proceso o la acción que se realiza.	Se enfoca en el dato o el estado del objeto.
No permite la abstracción, la encapsulación, el polimorfismo ni la herencia.	Permite la abstracción, la encapsulación, el polimorfismo y la herencia.
Es más fácil de entender y leer, pero más difícil de modificar y reutilizar.	Es más difícil de entender y leer, pero más fácil de modificar y reutilizar.

Encapsulamiento:

El encapsulamiento es el concepto que permite ocultar los detalles internos de un objeto y exponer sólo lo que es necesario para su uso externo. Para implementar el

encapsulamiento, se utiliza el modificador de acceso private en los atributos de un objeto, lo que significa que solo pueden ser accedidos directamente desde dentro de la clase. Para acceder a estos atributos desde fuera de la clase, se deben utilizar métodos públicos, que se conocen como getters y setters. Los getters se utilizan para obtener el valor de un atributo y los setters se utilizan para modificarlo.

Ejemplo:

```
public class Persona {  
    private String nombre;  
    private int edad;  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public int getEdad() {  
        return edad;  
    }  
  
    public void setEdad(int edad) {  
        this.edad = edad;  
    }  
}
```

En este ejemplo, la clase Persona tiene dos atributos privados, nombre y edad, y los métodos públicos getNombre, setNombre, getEdad y setEdad, que permiten acceder a estos atributos desde fuera de la clase.

Herencia

La herencia es el concepto que permite crear una nueva clase a partir de una clase existente, heredando sus atributos y métodos. La clase existente se conoce como clase padre o superclase y la nueva clase se conoce como clase hija o subclase. La subclase puede añadir nuevos atributos y métodos o modificar los existentes.

```
public class Empleado extends Persona {  
    private int salario;  
  
    public int getSalario() {  
        return salario;  
    }  
  
    public void setSalario(int salario) {  
        this.salario = salario;  
    }  
}
```

En este ejemplo, la clase Empleado hereda de la clase Persona y añade un nuevo atributo, salario.

Polimorfismo

El polimorfismo se refiere a la capacidad de los objetos de una clase para comportarse de diferentes maneras. En Java, el polimorfismo se implementa a través de la herencia y la sobrecarga de métodos.

Sobreescritura de métodos

La subclase puede sobrescribir o redefinir los métodos heredados de la superclase para adaptarse a sus necesidades.

```
public class Animal {
```

```

public void hacerSonido() {
    System.out.println("El animal hace un sonido");
}

public class Perro extends Animal {
    public void hacerSonido() {
        System.out.println("El perro ladra");
    }
}

public class Gato extends Animal {
    public void hacerSonido() {
        System.out.println("El gato maulla");
    }
}

```

En este ejemplo, la clase Animal es la superclase y las clases Perro y Gato son subclases. Ambas subclases sobrescriben el método hacerSonido() de la superclase para que se ajuste al sonido que hace el animal correspondiente.

Sobrecarga de métodos

La sobrecarga de métodos se refiere a la capacidad de una clase de tener múltiples métodos con el mismo nombre, pero con diferentes parámetros. En Java, el compilador determina qué método se debe llamar según los parámetros que se le pasen.

```

public class Calculadora {
    public int sumar(int a, int b) {
        return a + b;
    }

    public double sumar(double a, double b) {
        return a + b;
    }
}

```

```
public class Main {  
    public static void main(String[] args) {  
        Calculadora calc = new Calculadora();  
  
        System.out.println(calc.sumar(1, 2)); // Output: 3  
        System.out.println(calc.sumar(1.5, 2.5)); // Output: 4.0  
    }  
}
```

En este ejemplo, la clase Calculadora tiene dos métodos sumar(), uno que toma dos parámetros int y otro que toma dos parámetros double. Cuando se llama al método sumar() con diferentes tipos de parámetros, el compilador determina qué método se debe llamar según los tipos de parámetros que se le pasen.

Abstracción:

La abstracción se refiere a la capacidad de representar conceptos y objetos del mundo real en un programa de computadora de manera simplificada y concreta. En Java, la abstracción se implementa a través de clases y interfaces.

La abstracción se logra en Java mediante la creación de clases abstractas e interfaces.

Clase Abstracta

Una clase abstracta es una clase que no se puede instanciar, pero puede ser utilizada como una superclase para otras clases que hereden de ella. Las clases abstractas pueden contener métodos abstractos, que son métodos que no tienen una implementación en la clase abstracta y deben ser implementados por las clases que hereden de ella.

La interfaz

Es una colección de métodos abstractos que define un conjunto de comportamientos que una clase debe implementar. Las interfaces en Java permiten definir un conjunto común de métodos que pueden ser utilizados por varias clases, independientemente de su jerarquía de herencia.

Aquí hay un ejemplo de una clase abstracta y una interfaz en Java:

```
public abstract class Animal {  
    private String nombre;  
    private int edad;  
  
    public Animal(String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
  
    public abstract void hacerSonido();  
}  
  
public interface Mascota {  
    public void jugar();  
}
```

En este ejemplo, la clase abstracta "Animal" define el estado (nombre y edad) y el comportamiento (método abstracto `hacerSonido()`) de un animal genérico. La clase "Perro" o "Gato" que herede de la clase "Animal" deberá implementar el método "hacerSonido".

Por otro lado, la interfaz "Mascota" define el comportamiento común que deben tener las mascotas, en este caso el método "jugar". Cualquier clase que implemente la interfaz "Mascota" deberá implementar el método "jugar".

Cómo implementar una interface

En Java, se puede implementar una interfaz mediante la palabra clave "implements". Cuando una clase implementa una interfaz, se compromete a implementar todos los métodos declarados en la interfaz.

Los pasos para implementar una interfaz son los siguientes:

Definir la interfaz que se desea implementar. Por ejemplo, la siguiente es una interfaz llamada "MiInterfaz" que declara un método llamado "miMetodo":

```
public interface MiInterfaz {  
    void miMetodo();  
}
```

Crear una clase que implemente la interfaz. La clase debe usar la palabra clave "implements" seguida del nombre de la interfaz.

```
public class MiClase implements MiInterfaz {  
    public void miMetodo() {  
        System.out.println("Este es mi método implementado");  
    }  
}
```

Implementar los métodos declarados en la interfaz. En el ejemplo anterior, se implementa el método "miMetodo" declarado en la interfaz "MiInterfaz".

Usar la clase que implementa la interfaz. Por ejemplo:

```
MiClase objeto = new MiClase();  
objeto.miMetodo();
```

En este ejemplo, se crea un objeto de la clase "MiClase" y se llama al método "miMetodo" que fue implementado en la clase.

Modificadores de acceso

Los modificadores de acceso son palabras clave que se utilizan en Java para controlar el acceso a las variables, métodos y clases. Java tiene cuatro modificadores de acceso: public, protected, private y default (también conocido como package-private).

public

Las variables, métodos o clases declaradas como public son accesibles desde cualquier parte del código. Esto significa que se pueden acceder desde cualquier clase, ya sea dentro del mismo paquete o en otro paquete diferente.

Ejemplo:

```
public class Persona {  
    public String nombre;  
    public int edad;  
  
    public void saludar() {  
        System.out.println("Hola, soy " + nombre + " y tengo " + edad  
+ " años.");  
    }  
}
```

private

Las variables, métodos o clases declaradas como private sólo son accesibles dentro de la misma clase. Esto significa que no se pueden acceder desde otra clase, ni siquiera si se encuentra en el mismo paquete.

Ejemplo:

```
public class Persona {  
    private String nombre;  
    private int edad;  
  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public void setEdad(int edad) {
```

```
        this.edad = edad;
    }

    public int getEdad() {
        return edad;
    }

    private void saludar() {
        System.out.println("Hola, soy " + nombre + " y tengo " + edad
+ " años.");
    }
}
```

protected

Las variables, métodos o clases declaradas como `protected` son accesibles desde cualquier clase dentro del mismo paquete, y desde cualquier subclase, incluso si se encuentra en un paquete diferente.

Ejemplo:

```
package paquete1;
public class Persona {
    protected String nombre;
    protected int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
}

package paquete2;

import paquete1.Persona;
```

```
public class Estudiante extends Persona {
    private String carrera;

    public Estudiante(String nombre, int edad, String carrera) {
        super(nombre, edad);
        this.carrera = carrera;
    }

    public void mostrarInfo() {
        System.out.println("Nombre: " + nombre);
        System.out.println("Edad: " + edad);
        System.out.println("Carrera: " + carrera);
    }
}
```

default

Las variables, métodos o clases que no tienen ningún modificador de acceso (es decir, no se especifica `public`, `private` o `protected`) se consideran con acceso de "paquete". Esto significa que sólo son accesibles dentro del mismo paquete.

```
package paquete1;

class Persona {
    String nombre;
    int edad;

    void saludar() {
        System.out.println("Hola, soy " + nombre + " y tengo " + edad
+ " años.");
    }
}
```

Es importante destacar que los modificadores de acceso se utilizan para mejorar la seguridad y la encapsulación en el código, y para facilitar el mantenimiento y la escalabilidad de las aplicaciones.

La tabla muestra si un elemento con un determinado modificador puede ser accedido desde la misma clase (C), desde otra clase del mismo paquete (P), desde una subclase del mismo paquete (SP), desde una subclase de otro paquete (OP) o desde cualquier otra clase (O).

Modificador	C	P	SP	OP	O
private	Sí	No	No	No	No
default	Sí	Sí	Sí	No	No
protected	Sí	Sí	Sí	Sí	No
public	Sí	Sí	Sí	Sí	Sí

Clases

Una clase es un modelo o plantilla que define las características y el comportamiento de un objeto. Las clases pueden considerarse como un plano o diagrama que describe los atributos y los métodos que tendrán los objetos creados a partir de ellas.

Por ejemplo, si queremos crear una clase "Perro", podemos definir los atributos que tendrá cada objeto de esa clase, como "nombre", "raza", "edad" y "color". También podemos definir métodos como "ladrar", "correr" y "comer". De esta manera, todos los objetos creados a partir de la clase "Perro" tendrán esas mismas características y comportamientos.

El sintaxis para definir una clase en Java es la siguiente:

```
public class NombreDeLaClase {  
    // Definición de atributos y métodos  
}
```

Por ejemplo, para crear la clase "Perro", podemos hacer lo siguiente:

```
public class Perro {  
    String nombre;  
    String raza;  
    int edad;  
    String color;  
  
    public void ladrar() {  
        System.out.println("¡Guau, guau!");  
    }  
  
    public void correr() {  
        System.out.println("El perro está corriendo");  
    }  
  
    public void comer() {  
        System.out.println("El perro está comiendo");  
    }  
}
```

En este ejemplo, la clase "Perro" tiene cuatro atributos (nombre, raza, edad y color) y tres métodos (ladrar, correr y comer). Cada uno de estos métodos tiene un comportamiento específico que puede ser utilizado por cualquier objeto creado a partir de la clase "Perro".

Una vez que hemos definido la clase "Perro", podemos crear objetos de esa clase utilizando el operador "new", como se muestra a continuación:

```
Perro miPerro = new Perro();
```

Este código crea un nuevo objeto "Perro" y lo asigna a la variable "miPerro". Podemos acceder a los atributos y métodos de este objeto utilizando la sintaxis de punto, como se muestra a continuación:

```
miPerro.nombre = "Firulais";  
miPerro.raza = "Labrador";  
miPerro.edad = 3;  
miPerro.color = "Café";
```

```
miPerro.ladrar();
miPerro.correr();
miPerro.comer();
```

En este ejemplo, hemos asignado valores a los atributos del objeto "miPerro" y hemos llamado a sus métodos "ladrar", "correr" y "comer". Cada uno de estos métodos tiene un comportamiento específico que se aplica a este objeto en particular.

Los elementos de instancia

Son variables que pertenecen a una instancia o objeto de una clase. Cada instancia de una clase tiene sus propios valores para los elementos de instancia, lo que significa que los valores de estos elementos pueden variar entre las diferentes instancias de la misma clase.

Los elementos de instancia también se conocen como variables de instancia o campos de instancia. Se definen dentro de una clase, pero fuera de cualquier método. Estos elementos pueden ser de cualquier tipo de dato primitivo de Java, como int, double, char, boolean, o de cualquier objeto que haya sido definido en una clase.

Para acceder a los elementos de instancia, primero se debe crear una instancia de la clase que los contiene. Luego, se puede acceder a los elementos de instancia a través del objeto de la clase utilizando el operador de punto ":".

Por ejemplo, consideremos la siguiente clase "Persona":

```
public class Persona {
    //elementos de instancia
    String nombre;
    int edad;

    //constructor
```

```
public Persona(String nombre, int edad) {  
    this.nombre = nombre;  
    this.edad = edad;  
}  
  
//métodos de instancia  
public void presentarse() {  
    System.out.println("Hola, mi nombre es " + nombre + " y  
tengo " + edad + " años.");  
}  
}
```

En este ejemplo, la clase "Persona" tiene dos elementos de instancia: "nombre" y "edad". Estos elementos se inicializan mediante un constructor. La clase también tiene un método de instancia llamado "presentarse" que muestra la información de la persona. Para acceder a los elementos de instancia desde fuera de la clase, primero se debe crear una instancia de la clase y luego acceder a los elementos de instancia a través del objeto de la clase, como se muestra a continuación:

```
public static void main(String[] args) {  
    //crear una instancia de la clase Persona  
    Persona persona1 = new Persona("Juan", 25);  
  
    //acceder a los elementos de instancia a través del objeto de  
    la clase  
    System.out.println(persona1.nombre);  
    System.out.println(persona1.edad);  
  
    //llamar al método de instancia  
    persona1.presentarse();  
}
```

En este ejemplo, se crea una instancia de la clase "Persona" llamada "persona1". Luego, se accede a los elementos de instancia "nombre" y "edad" utilizando el objeto "persona1". También se llama al método de instancia "presentarse" utilizando el objeto "persona1".

Elementos de Clase o Estáticos

Son aquellos que pertenecen a la clase en sí misma en lugar de pertenecer a una instancia específica de la clase. Esto significa que los elementos de Clase se comparten entre todas las instancias de la clase y se pueden acceder sin crear una instancia de la clase.

Hay dos tipos de elementos de Clase en Java: Variables de Clase y métodos de Clase.

Variables de Clase

Son variables que pertenecen a la clase en sí misma y no a una instancia específica de la clase. Se definen utilizando el modificador "static" y se pueden acceder utilizando el nombre de la clase.

Ejemplo:

```
public class Ejemplo {  
    public static int contador = 0;  
}
```

En este ejemplo, la variable "contador" es una variable de Clase que se puede acceder utilizando el nombre de la clase "Ejemplo".

Métodos de Clase

Son métodos que pertenecen a la clase en sí misma y no a una instancia específica de la clase. Se definen utilizando el modificador "static" y se pueden acceder utilizando el nombre de la clase.

```
public class Ejemplo {  
    public static void saludar() {  
        System.out.println("Hola desde el método de Clase.");  
    }  
}
```

```
}
```

En este ejemplo, el método "saludar" es un método de Clase que se puede acceder utilizando el nombre de la clase "Ejemplo".

Es importante tener en cuenta que los elementos de Clase se comparten entre todas las instancias de la clase, por lo que si se cambia el valor de una variable de Clase en una instancia de la clase, ese cambio se reflejará en todas las demás instancias de la clase.

Constructores

Un constructor es un método especial que se llama automáticamente cuando se crea una instancia de una clase. Su objetivo principal es inicializar los atributos de la clase y garantizar que el objeto creado esté en un estado válido.

Un constructor tiene el mismo nombre que la clase y no tiene ningún tipo de retorno. Puede tener parámetros o no. Si no se define ningún constructor, Java proporciona automáticamente uno sin parámetros (conocido como constructor predeterminado).

A continuación se muestra un ejemplo de una clase "Persona" con dos atributos (nombre y edad) y dos constructores:

```
public class Persona {  
    private String nombre;  
    private int edad;  
  
    // Constructor sin parámetros  
    public Persona() {  
        this.nombre = "Sin nombre";  
        this.edad = 0;  
    }  
  
    // Constructor con parámetros  
    public Persona(String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
}
```

```
// Métodos getter y setter
public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}
public int getEdad() {
    return edad;
}
public void setEdad(int edad) {
    this.edad = edad;
}
}
```

En este ejemplo, la clase "Persona" tiene dos constructores: uno sin parámetros y otro con dos parámetros (nombre y edad). El constructor sin parámetros inicializa los atributos con valores predeterminados, mientras que el constructor con parámetros toma valores específicos para inicializar los atributos.

Para crear una instancia de la clase "Persona" utilizando el constructor sin parámetros, se puede hacer lo siguiente:

```
Persona persona1 = new Persona();
```

Esto creará un objeto "Persona" con el nombre "Sin nombre" y la edad 0.

Para crear una instancia de la clase "Persona" utilizando el constructor con parámetros, se puede hacer lo siguiente:

```
Persona persona2 = new Persona("Juan", 30);
```

Esto creará un objeto "Persona" con el nombre "Juan" y la edad 30.

Objetos

Un objeto en Java es una entidad que representa información sobre una cosa dentro del código de un programa. Un objeto es una instancia de una clase, es decir, un ejemplar creado a partir de un modelo o plantilla.

En Java, la creación de un objeto implica la reserva de memoria para almacenar sus atributos y la asignación de valores iniciales a ellos. Los objetos se crean mediante la palabra clave "new", seguida del nombre de la clase y los parámetros que se necesiten para el constructor de la clase, si es que tiene uno.

Por ejemplo, si tenemos la siguiente clase "Persona":

```
public class Persona {  
    // Atributos  
    String nombre;  
    int edad;  
  
    // Constructor  
    public Persona(String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
  
    // Métodos  
    public void saludar() {  
        System.out.println("Hola, soy " + nombre + " y tengo " +  
edad + " años.");  
    }  
}
```

Podemos crear objetos de la clase Persona de la siguiente manera:

```
Persona p1 = new Persona("Juan", 30);  
Persona p2 = new Persona("María", 25);
```

Los Wrappers

Son clases que se utilizan para envolver o envolver tipos de datos primitivos en un objeto. Estos tipos de datos primitivos incluyen int, float, double, boolean, char, byte y short. Al envolver un tipo de datos primitivo en un objeto, se pueden utilizar métodos y características adicionales proporcionados por la clase Wrapper.

Por ejemplo, si deseas trabajar con un número entero, puedes utilizar la clase Wrapper Integer para envolver el valor int en un objeto. La clase Integer proporciona métodos útiles como parseInt() para convertir una cadena en un número entero, compareTo() para comparar dos valores enteros y valueOf() para obtener una instancia de la clase Integer a partir de un valor entero.

Aquí hay algunos ejemplos de cómo se utilizan los Wrappers en Java:

Envolver un tipo de datos primitivo en un objeto:

```
int num = 10;
Integer numObj = Integer.valueOf(num);
```

Convertir una cadena en un número entero:

```
String str = "123";
int num = Integer.parseInt(str);
```

Comparar dos valores enteros:

```
Integer num1 = 10;
Integer num2 = 20;
int result = num1.compareTo(num2);
```

Convertir un valor entero en una cadena:

```
Integer num = 123;
String str = num.toString();
```

Los Wrappers también son útiles cuando se trabaja con colecciones como Listas o Mapas que solo pueden contener objetos, no tipos de datos primitivos. Al envolver un tipo de datos primitivo en un objeto Wrapper, puedes agregarlo a una colección como un objeto.

Colecciones

Las colecciones son estructuras de datos que almacenan objetos en Java. Las colecciones permiten manipular grupos de objetos de forma eficiente y flexible.

Las colecciones en Java son un marco que proporciona una arquitectura para almacenar y manipular el grupo de objetos. Las colecciones de Java pueden realizar todas las operaciones que se hacen en un dato, como buscar, ordenar, insertar, manipular y eliminar. Las colecciones en Java se organizan en una jerarquía de interfaces y clases.

Excepciones

Las excepciones en Java son objetos que representan situaciones anómalas o errores que ocurren durante la ejecución de un programa. Java tiene muchos tipos de excepciones predefinidas para diferentes casos, como por ejemplo `ArithmaticException` para una división entre cero, `NullPointerException` para un objeto nulo que no puede serlo, `IOException` para un error de entrada/salida, etc.

Cuando se produce una excepción, el programa se detiene y muestra un mensaje de error, a menos que se maneje la excepción con un bloque `try-catch`.

Tipos de errores pueden ocurrir durante la ejecución de un programa

Durante la ejecución de un programa en Java pueden ocurrir dos tipos de errores: los de tiempo de compilación y los de tiempo de ejecución.

Errores de tiempo de compilación

Los errores de tiempo de compilación son aquellos que se detectan cuando se compila el código fuente, como por ejemplo errores de sintaxis, de tipos o de acceso. Estos errores deben ser corregidos antes de poder ejecutar el programa.

Errores de tiempo de ejecución

Los errores de tiempo de ejecución son aquellos que se producen cuando el programa está en funcionamiento, como por ejemplo divisiones por cero, acceso a posiciones inválidas en un arreglo o apertura fallida de un archivo. Estos errores pueden ser tratados mediante el uso de excepciones.

Tipos de Excepciones

Las excepciones se dividen en dos categorías: verificadas y no verificadas.

Verificadas

Las excepciones verificadas son aquellas que el compilador obliga a manejar mediante bloques try-catch o declarando que el método puede lanzarlas con la palabra clave throws. Estas excepciones suelen representar condiciones externas al programa que pueden recuperarse, como por ejemplo problemas con la entrada/salida o con la red.

No verificadas

Las excepciones no verificadas son aquellas que el compilador no obliga a manejar y que suelen representar errores lógicos del programador o condiciones irrecuperables, como por ejemplo argumentos inválidos, punteros nulos o violaciones aritméticas.

Bloque try-catch

Un bloque try-catch es una estructura que permite manejar las excepciones y evitar que el programa se termine abruptamente. El bloque try contiene el código que puede generar una excepción, y el bloque catch contiene el código que se ejecuta si se produce una excepción en el bloque try. Por ejemplo:

```
try {  
    int x = 10 / 0; // Esto genera una ArithmeticException  
} catch (ArithmetricException e) {  
    System.out.println("No se puede dividir por cero"); // Esto se  
    imprime si ocurre la excepción  
}
```

Puedes tener varios bloques catch para diferentes tipos de excepciones, o usar un bloque finally para ejecutar código después del try-catch, independientemente de si hubo o no una excepción.

Puedes usar un bloque try-catch para capturar dos tipos de excepciones en Java usando dos bloques catch con diferentes tipos de excepción. Por ejemplo:

```
try {  
    // Código que puede lanzar excepciones  
}  
catch (FileNotFoundException e) {  
    // Código para manejar la excepción de archivo no encontrado  
}  
catch (IOException e) {  
    // Código para manejar otras excepciones de entrada/salida  
}
```

Bloque finally

Un bloque finally es un bloque opcional que se asocia con un bloque try y se ejecuta siempre que el bloque try salga, ya sea por una excepción o no. El bloque finally es útil para liberar recursos o hacer limpieza después de un bloque try-catch. Por ejemplo:

```
try {  
    // Abrir un archivo  
} catch (IOException e) {  
    // Manejar el error de entrada/salida  
} finally {  
    // Cerrar el archivo  
}
```

Interfaces funcionales

Una interfaz funcional es una interfaz que contiene exactamente un método abstracto.

Estas interfaces se utilizan principalmente para aprovechar las [expresiones lambda](#) y nos permite la programación funcional en Java. Una interfaz funcional puede tener múltiples métodos predeterminados y estáticos, pero solo un método abstracto.

Facilita la escritura de código más conciso y legible cuando se trabaja con programación funcional en Java.

Definición de una interfaz funcional

Para definir una interfaz funcional, se utiliza la anotación `@FunctionalInterface` antes de la declaración de la interfaz.

Esto no es estrictamente necesario, pero ayuda a comunicar la intención de que esta interfaz debe ser utilizada como una interfaz funcional.

```
@FunctionalInterface  
interface MiInterfazFuncional {  
    void miMetodoAbstracto();  
}
```

Expresiones lambda

Una [expresión lambda](#) es una forma concisa de definir una instancia de una interfaz funcional. Se usa el operador `->` para definir una expresión lambda.

```
MiInterfazFuncional miLambda = () -> System.out.println("Hola  
desde la expresión lambda");  
miLambda.miMetodoAbstracto();
```

Métodos de referencia:

Los métodos de referencia son una forma aún más concisa de llamar a métodos existentes o constructores como expresiones lambda.

Ejemplo utilizando una referencia a un método estático:

```
// Método estático  
static void miMetodoEstatico() {  
    System.out.println("Hola desde el método estático");  
}  
MiInterfazFuncional referenciaMetodo =  
Ejemplo::miMetodoEstatico;  
referenciaMetodo.miMetodoAbstracto();
```

Uso de interfaces funcionales predefinidas

Java proporciona algunas interfaces funcionales predefinidas en el paquete **java.util.function** para realizar operaciones comunes en programación funcional, como **Predicate**, **Function**, **Consumer**, etc.

Estas interfaces funcionales se pueden utilizar para expresiones lambda y métodos de referencia.

```
import java.util.function.Predicate;  
  
Predicate<Integer> esPar = num -> num % 2 == 0;  
System.out.println(esPar.test(6)); // Devuelve true
```

Expresiones lambda

Permite definir funciones anónimas de manera más concisa. Estas funciones pueden tratarse como objetos y se utilizan principalmente en el contexto de programación funcional.

Las expresiones lambda son especialmente útiles cuando trabajas con interfaces funcionales.

Sintaxis de una expresión lambda:

La sintaxis básica de una expresión lambda consiste en los siguientes elementos:

(parámetros) -> expresión o cuerpo de la función

- Los **parámetros** son los valores de entrada que la función lambda toma.
- La flecha **->** separa los parámetros del cuerpo de la función.
- El **cuerpo de la función** contiene la lógica de la función lambda y puede ser una expresión o un bloque de código.

Supongamos que queremos definir una función lambda que toma dos números y devuelve su suma:

```
(int a, int b) -> a + b
```

En este caso, **(int a, int b)** son los parámetros y **a + b** es el cuerpo de la función lambda.

Expresión lambda que no toma argumentos

Si la función lambda no toma argumentos, la sintaxis sería la siguiente:

```
() -> "Hola, mundo"
```

Esta expresión lambda no tiene argumentos y devuelve la cadena "Hola, mundo".

Expresión lambda en una variable

Las expresiones lambda sin argumentos son útiles en situaciones donde no se requieren entradas específicas, pero se necesita realizar una tarea o devolver un valor.

Puedes asignar una expresión lambda a una variable y usar esa variable para llamar a la función lambda.

```
Runnable miRunnable = () -> {
    System.out.println("Hola desde la expresión lambda");
};  
miRunnable.run();
```

En este ejemplo, `Runnable` es una interfaz funcional que no toma argumentos y no devuelve ningún valor. La expresión lambda asignada a miRunnable se ejecuta cuando llamamos al método run().

Expresión lambda en una lista:

Las expresiones lambda son útiles para operaciones en colecciones, como filtrado, mapeo y reducción. Aquí hay un ejemplo que filtra una lista de números para encontrar los pares:

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
List<Integer> pares = numeros.stream()
    .filter(n -> n % 2 == 0)
    .collect(Collectors.toList());
```

En este caso, `n -> n % 2 == 0` es una expresión lambda que se utiliza como criterio de filtrado para mantener solo los números pares en la lista.

Aplicación de Lambdas en colecciones

Permite trabajar de manera más concisa y expresiva con colecciones de datos.

Supongamos que tenemos una lista de números enteros como esta:

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9,
10);
```

forEach

El método `forEach` se utiliza para aplicar una operación a cada elemento de la colección. Puedes usar una expresión lambda para definir la operación que se ejecutará en cada elemento.

En este ejemplo, la expresión lambda `numero -> System.out.println(numero)` se ejecutará para cada elemento de la lista, imprimiendo cada número en la consola.

```
numeros.forEach(numero -> System.out.println(numero));
```

anyMatch

El método `anyMatch` se utiliza para verificar si al menos un elemento de la colección cumple con cierta condición. Puedes usar una expresión lambda para definir la condición.

```
boolean hayNumerosPares = numeros.stream()
    .anyMatch(numero -> numero % 2 == 0);
System.out.println("¿Hay números pares? " + hayNumerosPares);
```

En este ejemplo, la expresión lambda `numero -> numero % 2 == 0` verifica si al menos un número en la lista es par. El resultado se almacena en la variable `hayNumerosPares`.

allMatch

El método **allMatch** se utiliza para verificar si todos los elementos de la colección cumplen con cierta condición. Puedes usar una expresión lambda para definir la condición.

```
boolean todosSonMayoresQueCero = numeros
    .stream()
    .allMatch(numero -> numero > 0);
System.out.println("¿Son todos los números mayores que cero? " +
    todosSonMayoresQueCero);
```

En este ejemplo, la expresión lambda `numero -> numero > 0` verifica si todos los números en la lista son mayores que cero. El resultado se almacena en la variable **todosSonMayoresQueCero**.

map

El método `map` se utiliza para transformar cada elemento de la colección en otro valor utilizando una expresión lambda. El resultado es una nueva colección con los valores transformados.

En este ejemplo, la expresión lambda `numero -> "Número: " + numero` se utiliza para transformar cada número en una cadena que incluye el texto "Número: ". El resultado es una nueva lista de cadenas.

```
List<String> numerosComoCadenas = numeros.stream()
    .map(numero -> "Número: " + numero)
    .collect(Collectors.toList());

numerosComoCadenas.forEach(System.out::println);
```

Aplanación de colecciones

Se refiere a la tarea de combinar y procesar los elementos de una colección para obtener un único resultado o valor.

Collections.min()

El método `Collections.min()` se utiliza para encontrar el elemento mínimo en una colección, como una lista.

```
List<Integer> numeros = new ArrayList<>();
numeros.add(5);
numeros.add(2);
numeros.add(8);

int min = Collections.min(numeros);
System.out.println("El número mínimo es: " + min);
```

En este ejemplo, `Collections.min()` encuentra el valor mínimo en la lista de números, que es 2.

Collections.max()

El método `Collections.max()` se utiliza para encontrar el elemento máximo en una colección.

```
List<Integer> numeros = new ArrayList<>();  
numeros.add(5);  
numeros.add(2);  
numeros.add(8);  
  
int max = Collections.max(numeros);  
System.out.println("El número máximo es: " + max);
```

En este ejemplo, **Collections.max()** encuentra el valor máximo en la lista de números, que es 8.

IntStream.sum()

El método **IntStream.sum()** se utiliza para calcular la suma de elementos en un flujo de enteros.

```
int[] numeros = {1, 2, 3, 4, 5};  
int suma = IntStream.of(numeros).sum();  
System.out.println("La suma de los números es: " + suma);
```

En este ejemplo, **IntStream.sum()** calcula la suma de los elementos en el flujo de enteros, que es 15.

Collectors.joining()

El método `Collectors.joining()` se utiliza para concatenar elementos de una colección en una sola cadena, separados por un delimitador opcional.

```
Stream<String> palabras = Stream.of("Hola", "mundo", "Java");
```

```
String resultado = palabras.collect(Collectors.joining(" "));  
System.out.println("Cadena concatenada: " + resultado);
```

En este ejemplo, **Collectors.joining(" ")** concatena las palabras del flujo en una sola cadena separada por espacios, resultando en "Hola mundo Java".

Patrones de Diseño

Los patrones de diseño son soluciones probadas y ampliamente aceptadas para problemas comunes que los desarrolladores pueden encontrar al diseñar y desarrollar software. Estos patrones proporcionan un enfoque estructurado y reutilizable para resolver problemas específicos en el diseño de software.

Los patrones de diseño en Java se basan en principios de la programación orientada a objetos (POO) y se utilizan para mejorar la modularidad, la reutilización del código, la facilidad de mantenimiento y la flexibilidad del software. Estos patrones se describen utilizando un lenguaje común que permite a los desarrolladores comunicarse de manera efectiva acerca de soluciones de diseño.

Cada patrón de diseño tiene un propósito específico y se utiliza en situaciones particulares. Los desarrolladores utilizan estos patrones para resolver problemas comunes de diseño de software de manera eficiente y elegante.

Los patrones de diseño **no son reglas rígidas**, sino más bien pautas que pueden adaptarse y personalizarse según las necesidades del proyecto. Al aplicar patrones de diseño en Java, se fomenta la escritura de código más limpio, mantenable y fácil de entender.

Clasificación

Existen varios tipos de patrones de diseño, pero generalmente se dividen en tres categorías principales:

1. Patrones de creación

Estos patrones se centran en la **creación de objetos**. Ayudan a abstraer y ocultar los detalles de la creación de objetos, lo que facilita la gestión de dependencias y la flexibilidad en la creación de instancias.

Ejemplos de patrones de creación incluyen Singleton, Factory Method y Abstract Factory.

2. Patrones de estructura:

Estos patrones se centran en **cómo se organizan y componen los objetos** para formar estructuras más grandes. Ayudan a definir la relación entre los objetos y cómo se ensamblan para formar sistemas más complejos.

Ejemplos de patrones de estructura incluyen Adapter, Decorator y Composite.

3. Patrones de comportamiento:

Estos patrones se centran en **cómo los objetos interactúan y se comunican entre sí**. Ayudan a definir el comportamiento de los objetos y cómo responden a eventos y cambios en el sistema.

Ejemplos de patrones de comportamiento incluyen Observer, Strategy y Command.

Patrón State

El patrón de diseño State es un patrón de comportamiento que permite que un objeto altere su comportamiento cuando su estado interno cambia.

Esto se logra representando cada estado como un objeto separado y permitiendo que el objeto principal cambie su estado delegando el comportamiento a estos objetos de estado.

El patrón State ayuda a que un objeto mantenga su coherencia y evite una gran cantidad de instrucciones condicionales en su código.

Ejemplo del Patrón State

Supongamos que estamos desarrollando un reproductor de música. El reproductor de música puede tener varios estados, como "Reproduciendo", "Pausado" y "Detenido". Utilizaremos el patrón State para implementar estos estados.

1. Crear las clases de estado

Creamos una interfaz `Estado` que define los métodos comunes para todos los estados:

```
interface Estado {  
    void reproducir();  
    void pausar();  
    void detener();  
}
```

Luego, implementamos tres clases que representan los estados concretos: `Reproduciendo`, `Pausado` y `Detenido`, que implementan la interfaz `Estado` y proporcionan implementaciones específicas para cada estado.

```
class Reproduciendo implements Estado {  
    @Override  
    public void reproducir() {  
        System.out.println("El reproductor ya está en estado  
de reproducción.");  
    }  
  
    @Override  
    public void pausar() {  
        System.out.println("Pausando la reproducción.");  
    }  
  
    @Override  
    public void detener() {  
        System.out.println("Deteniendo la reproducción.");  
    }  
}
```

```
}

class Pausado implements Estado {
    @Override
    public void reproducir() {
        System.out.println("Reanudando la reproducción.");
    }

    @Override
    public void pausar() {
        System.out.println("El reproductor ya está en estado
de pausa.");
    }

    @Override
    public void detener() {
        System.out.println("Deteniendo la reproducción.");
    }
}

class Detenido implements Estado {
    @Override
    public void reproducir() {
        System.out.println("Iniciando la reproducción.");
    }

    @Override
    public void pausar() {
        System.out.println("El reproductor está detenido. No
se puede pausar.");
    }

    @Override
    public void detener() {
```

```
        System.out.println("El reproductor ya está
detenido.");
    }
}
```

2. Crear la clase Contexto:

La clase **ReproductorMusica** es el contexto que utiliza el patrón State. Esta clase mantiene una referencia al estado actual y delega las operaciones a ese estado.

```
class ReproductorMusica {
    private Estado estado;

    public ReproductorMusica() {
        this.estado = new Detenido(); // Inicialmente, el
reproductor está detenido.
    }

    public void setEstado(Estado estado) {
        this.estado = estado;
    }

    public void reproducir() {
        estado.reproducir();
    }

    public void pausar() {
        estado.pausar();
    }

    public void detener() {
        estado.detener();
    }
}
```

3. Utilizar el patrón State

Ahora podemos usar el reproductor de música y cambiar su estado de manera dinámica.

```
public class Main {  
    public static void main(String[] args) {  
        ReproductorMusica reproductor = new  
ReproductorMusica();  
  
        reproductor.reproducir(); // Iniciando La reproducción  
        reproductor.pausar(); // El reproductor ya está en  
estado de reproducción.  
        reproductor.detener(); // Deteniendo La reproducción  
  
        reproductor.setEstado(new Pausado());  
        reproductor.reproducir(); // Reanudando La  
reproducción  
    }  
}
```

En este ejemplo, el patrón State permite que el reproductor de música cambie su comportamiento sin requerir instrucciones condicionales complejas en el código principal. Cada estado tiene su propia lógica de funcionamiento, y el contexto (**ReproductorMusica**) delega las llamadas a los métodos en el estado actual. Esto hace que el código sea más mantenable y extensible, ya que es fácil agregar nuevos estados sin afectar el código existente.

Patrón Template Method

El patrón de diseño Template Method es un patrón de comportamiento que se utiliza en la programación orientada a objetos para definir una estructura de algoritmo en una superclase, pero permitir que las subclases proporcionen implementaciones específicas de pasos individuales del algoritmo sin cambiar su estructura global.

Esto promueve la reutilización del código y permite a las subclases personalizar partes del algoritmo según sus necesidades.

Estructura del Patrón Template Method

El patrón Template Method se compone de los siguientes elementos:

1. Clase Abstracta (Clase Base)

Define un método llamado "**templateMethod**" que establece la estructura del algoritmo. Este método utiliza otros métodos (llamados "hooks" o "pasos") para realizar tareas específicas. Los pasos pueden ser abstractos o tener una implementación predeterminada.

2. Clases Concretas (Subclases):

Subclases que heredan de la clase abstracta y proporcionan implementaciones concretas para los pasos individuales del algoritmo.

Ejemplo del Patrón Template Method

Supongamos que estamos desarrollando un juego de mesa con dos jugadores y queremos implementar un algoritmo común para el turno de juego, pero permitir que cada jugador personalice su comportamiento de turno.

Paso 1: Clase abstracta que define el Template Method

```
abstract class JuegoMesa {  
    public void jugar() {  
        iniciarJuego();  
        realizarJugada();  
        finalizarJuego();  
    }  
    abstract void iniciarJuego(); // Paso abstracto  
    abstract void realizarJugada(); // Paso abstracto  
    abstract void finalizarJuego(); // Paso abstracto  
}
```

Paso 2: Clases concretas (jugadores) que heredan de la clase abstracta

```
class Jugador1 extends JuegoMesa {  
    @Override  
    void iniciarJuego() {  
        System.out.println("Jugador 1 comienza el juego.");  
    }  
  
    @Override  
    void realizarJugada() {  
        System.out.println("Jugador 1 realiza su jugada.");  
    }  
  
    @Override  
    void finalizarJuego() {  
        System.out.println("Jugador 1 gana el juego.");  
    }  
}  
  
class Jugador2 extends JuegoMesa {  
    @Override  
    void iniciarJuego() {  
        System.out.println("Jugador 2 comienza el juego.");  
    }  
  
    @Override  
    void realizarJugada() {  
        System.out.println("Jugador 2 realiza su jugada.");  
    }  
  
    @Override  
    void finalizarJuego() {  
        System.out.println("Jugador 2 gana el juego.");  
    }  
}
```

```
    }  
}
```

Paso 3: Programa principal que utiliza el patrón Template Method

```
public class Main {  
    public static void main(String[] args) {  
        JuegoMesa juego = new Jugador1();  
        juego.jugar();  
  
        System.out.println();  
  
        juego = new Jugador2();  
        juego.jugar();  
    }  
}
```

En este ejemplo, hemos creado un juego de mesa con dos jugadores. La clase **JuegoMesa** define el algoritmo común utilizando el método **jugar()**, que consta de tres pasos: **iniciarJuego()**, **realizarJugada()** y **finalizarJuego()**. Cada jugador (las clases **Jugador1** y **Jugador2**) hereda de la clase **JuegoMesa** y proporciona sus propias implementaciones para estos pasos.

Cuando ejecutas el programa, verás que cada jugador personaliza su comportamiento de turno, pero el algoritmo general del juego se mantiene sin cambios. Este es un ejemplo típico de cómo el patrón Template Method permite definir una estructura de algoritmo en una superclase y permitir que las subclases personalicen partes específicas de ese algoritmo.

Archivos

Escribir y leer archivos en Java es una operación común en el desarrollo de software. Java proporciona varias clases en el paquete `java.io` y en el paquete `java.nio.file` para manejar estas operaciones.

Cómo escribir y leer archivos en Java:

Clases principales:

- **FileWriter**: Permite escribir caracteres en un archivo.
- **BufferedWriter**: Proporciona un búfer para mejorar la eficiencia de escritura.

Proceso básico:

- Crear un objeto **FileWriter** y un objeto **BufferedWriter** asociado.
- Usar el método **write** para escribir datos en el archivo.
- Cerrar el **BufferedWriter** para asegurar que los datos se escriban correctamente y liberar los recursos.

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

public class EscribirArchivoJava {

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

```
String nombreArchivo = "miarchivo.txt";

try (FileWriter fileWriter = new
FileWriter(nombreArchivo);
        BufferedWriter bufferedWriter = new
BufferedWriter(fileWriter)) {

    bufferedWriter.write("Hola, este es un ejemplo de
escritura en Java.");
    bufferedWriter.newLine();
    bufferedWriter.write("Puedes agregar más líneas según
sea necesario.");

    System.out.println("Se ha escrito en el archivo
correctamente.");
}

} catch (IOException e) {
    System.err.println("Error al escribir en el archivo: "
+ e.getMessage());
}
}
```

Leer un Archivo

Clases principales:

- **FileReader:** Permite leer caracteres de un archivo.
- **BufferedReader:** Proporciona un búfer para mejorar la eficiencia de lectura.

Proceso básico:

- Crear un objeto **FileReader** y un objeto **BufferedReader** asociado.
- Usar el método **read** o **readLine** para leer datos del archivo.
- Cerrar el **BufferedReader** para liberar recursos.

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

public class LeerArchivoJava {

    public static void main(String[] args) {
        String nombreArchivo = "miarchivo.txt";

        try (FileReader fileReader = new FileReader(nombreArchivo);
        BufferedReader bufferedReader = new BufferedReader(fileReader)) {
            String linea;
            while ((linea = bufferedReader.readLine()) != null) {
                System.out.println(linea);
            }
        } catch (IOException e) {
            System.err.println("Error al leer el archivo: " +
e.getMessage());
        }
    }
}
```

- Es importante cerrar los recursos (usando el bloque **try-with-resources** o manualmente en el bloque **finally**) para evitar pérdida de datos y asegurar la liberación de recursos.
- Asegúrate de manejar las excepciones (`IOException`) que puedan ocurrir durante las operaciones de escritura y lectura.
- Puedes utilizar las clases File y Paths para trabajar con rutas de archivos de manera más flexible.

try-catch en archivos

Colocar operaciones relacionadas con archivos dentro de un bloque `try-catch` en Java es una buena práctica para manejar posibles excepciones que puedan ocurrir durante la lectura o escritura de archivos. Las operaciones de entrada/salida (I/O) en archivos pueden generar excepciones debido a diversas razones, como la falta de permisos para acceder al archivo, la pérdida de conexión durante la operación, o el intento de acceder a un archivo que no existe.

```
try (FileWriter fileWriter = new FileWriter("miarchivo.txt");
      BufferedWriter bufferedWriter = new
      BufferedWriter(fileWriter)) {

    // Código para escribir en el archivo

} catch (IOException e) {
    // Manejar la excepción de escritura
    System.err.println("Error al escribir en el archivo: " +
e.getMessage());
}

try (FileReader fileReader = new FileReader("miarchivo.txt");
      BufferedReader bufferedReader = new
      BufferedReader(fileReader)) {

    // Código para leer del archivo

} catch (IOException e) {
    // Manejar la excepción de Lectura
    System.err.println("Error al leer el archivo: " +
e.getMessage());
}
```




Maven es una herramienta de gestión de proyectos de software utilizada en la construcción, documentación y gestión de dependencias de proyectos Java. Fue desarrollada por la Apache Software Foundation y está escrita en Java. Maven es una herramienta de automatización de construcción de software que ayuda a los desarrolladores a crear aplicaciones de software de manera más eficiente.

La herramienta se basa en la noción de un "proyecto" y su "modelo de objeto de proyecto" (POM, por sus siglas en inglés). El archivo POM es un archivo XML que describe el proyecto, incluyendo información sobre las dependencias del proyecto, los recursos, los plugins y los objetivos. Con esta información, Maven es capaz de generar la estructura de directorios y los archivos de configuración necesarios para construir y ejecutar el proyecto.

Ventajas de utilizar Maven

- Gestión de dependencias:** Maven descarga automáticamente las dependencias necesarias para el proyecto desde un repositorio central de Maven, lo que hace que la gestión de dependencias sea más fácil y eficiente.
- Automatización de la construcción:** Maven utiliza scripts XML para la construcción del proyecto, lo que facilita la creación de proyectos complejos.
- Estandarización:** La estructura del proyecto se define de acuerdo con un estándar predefinido, lo que ayuda a mantener la coherencia entre los proyectos y facilita la integración de los proyectos entre sí.
- Integración con IDE:** Maven es compatible con la mayoría de las herramientas de desarrollo de Java.

El Archivo POM

El archivo pom.xml es el archivo de configuración central en un proyecto Maven. "POM" significa "Project Object Model". Este archivo define la información del proyecto, las dependencias, los plugins y las configuraciones de construcción que se utilizan para construir el proyecto.

El archivo pom.xml es utilizado por Maven para realizar una serie de tareas, incluyendo la descarga y la resolución de dependencias, la compilación del código fuente, la ejecución de pruebas, la generación de informes de pruebas y la creación de artefactos para el proyecto. También proporciona información importante sobre el proyecto para otros desarrolladores y herramientas, como su versión, descripción, licencia y desarrolladores.

El archivo pom.xml se crea en la raíz del proyecto y se utiliza para describir la estructura del proyecto, sus dependencias y otros metadatos.

Algunos de los elementos más comunes que se definen en el archivo pom.xml son:

- **groupId**: es el identificador único del grupo de proyectos al que pertenece el proyecto. Se utiliza para organizar los proyectos en grupos y se utiliza para generar el identificador de paquete Java.
- **artifactId**: es el identificador único del proyecto en el grupo de proyectos. Se utiliza para generar el nombre del archivo JAR, WAR o EAR que se crea para el proyecto.
- **version**: es la versión del proyecto. Se utiliza para identificar de manera única una versión específica del proyecto y para determinar qué dependencias se deben descargar.
- **dependencies**: es una lista de dependencias que se requieren para construir el proyecto. Estas dependencias se descargan automáticamente por Maven y se agregan al proyecto durante el proceso de compilación.
- **build**: contiene información sobre cómo construir el proyecto, incluyendo los plugins de construcción que se deben utilizar, los directorios de salida y los recursos adicionales que se deben incluir.

El archivo pom.xml es esencial para los proyectos de Maven, ya que proporciona toda la información necesaria para construir y gestionar el proyecto de forma coherente. Es importante que el archivo se mantenga actualizado y que se realice una buena gestión de dependencias para asegurar que el proyecto se construya correctamente.

Las partes principales de un archivo POM

Elemento de proyecto (project):

Es el elemento principal del archivo POM. Contiene toda la información sobre el proyecto, como su identificación, versión, descripción, etc. El elemento de proyecto tiene dos subelementos obligatorios: groupId, artifactId y version.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>my-project</artifactId>
    <version>1.0.0</version>

    <name>My Project</name>
    <description>A simple Maven project</description>
</project>
```

Elemento de dependencias (dependencies):

Contiene información sobre las dependencias del proyecto, como su groupId, artifactId y versión. Cada dependencia se especifica dentro de un elemento de dependencia.

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
    <dependency>
```

```
<groupId>com.example</groupId>
<artifactId>my-library</artifactId>
<version>1.0.0</version>
</dependency>
</dependencies>
```

Elemento de plugins (plugins):

Contiene información sobre los plugins utilizados en el proyecto. Los plugins se especifican dentro de un elemento de plugin.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Elemento de perfiles (profiles):

Contiene información sobre los perfiles del proyecto. Los perfiles se utilizan para personalizar la construcción del proyecto según diferentes entornos, como el desarrollo o la producción.

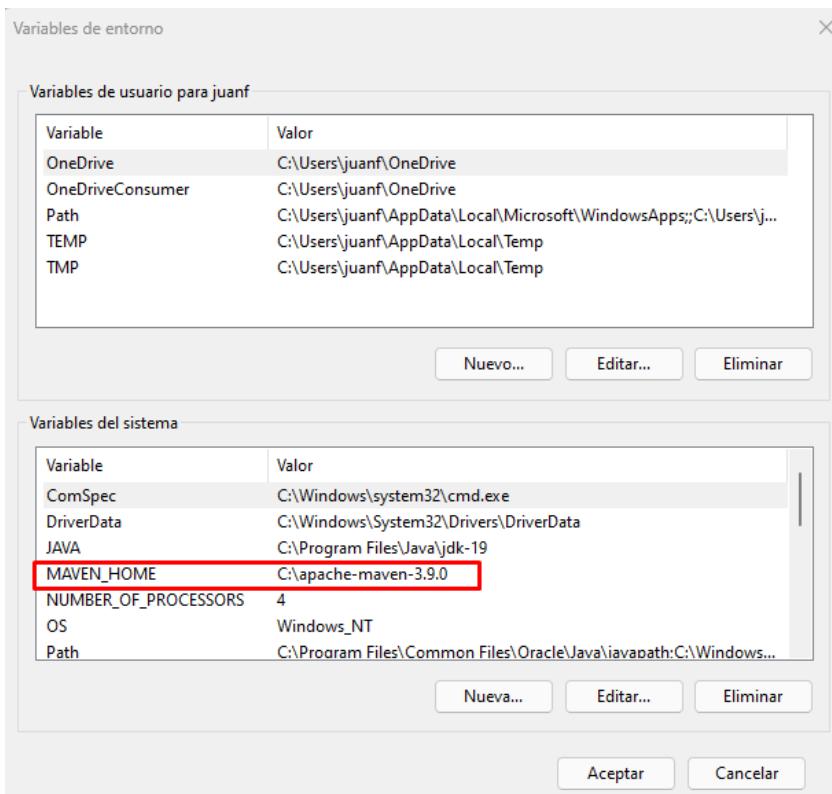
```
<profiles>
  <profile>
```

```
<id>dev</id>
<activation>
    <activeByDefault>true</activeByDefault>
</activation>
<properties>
    <db.url>jdbc:mysql://localhost/dev_db</db.url>
</properties>
</profile>
<profile>
    <id>prod</id>
    <properties>
        <db.url>jdbc:mysql://prod-server/prod_db</db.url>
    </properties>
</profile>
</profiles>
```

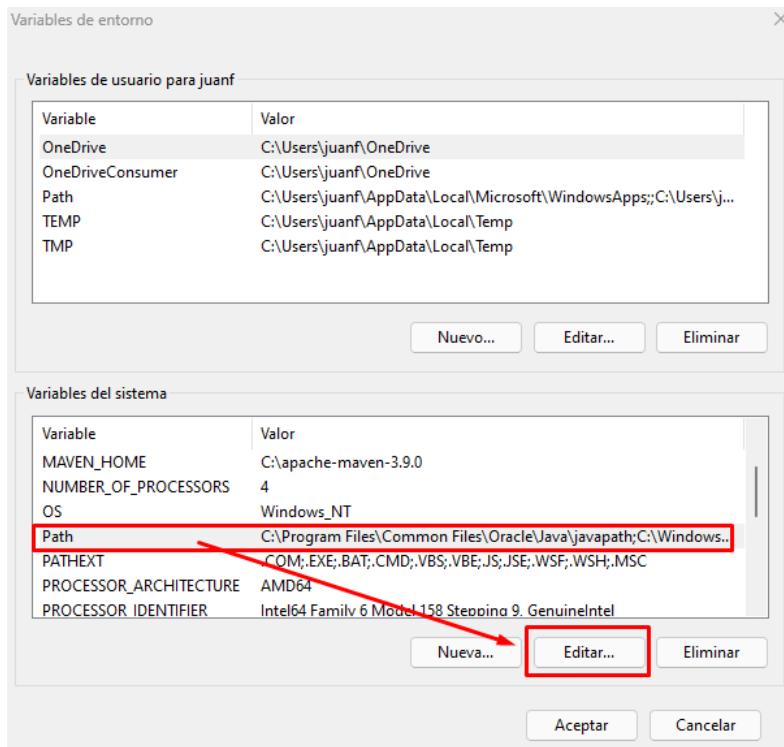
Instalar Maven

Antes de instalar Maven, debemos [instalar y configurar Java](#) en nuestro equipo. Para instalar y configurar Maven en Windows, sigue estos pasos:

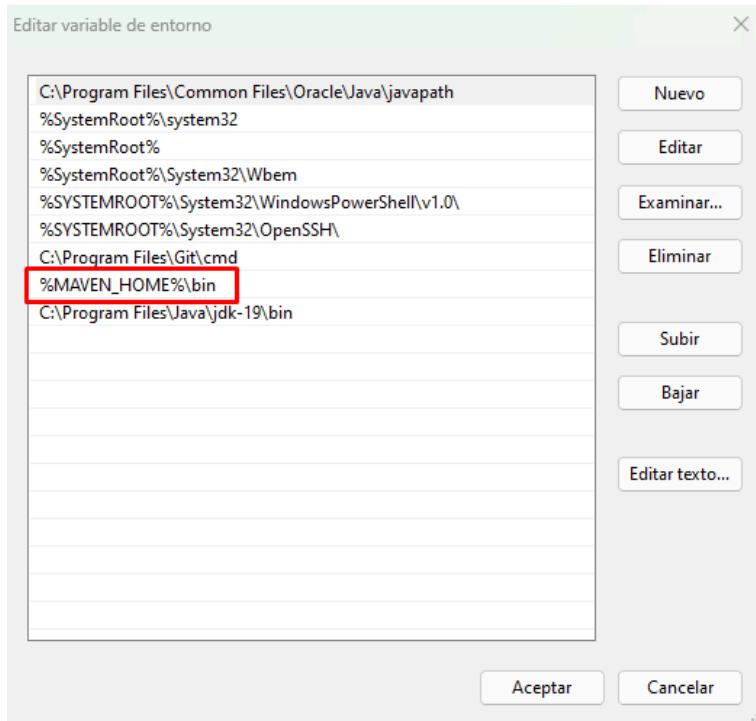
- 1) Descarga el archivo binario de Maven desde el sitio oficial de Apache Maven: <https://maven.apache.org/download.cgi> Selecciona la última versión estable y descarga el archivo ZIP correspondiente a la versión de Windows.
- 2) Extrae el archivo ZIP en una ubicación conveniente de tu disco duro, por ejemplo, C:\Program Files.
- 3) Crea una variable de entorno llamada MAVEN_HOME que apunte a la ubicación donde has descomprimido Maven.



- 4) Agrega la ruta de la carpeta bin de Maven al PATH del sistema llenando a "Variables del sistema" de la ventana de "Variables de entorno", selecciona la variable "Path" y haz clic en "Editar".



- 5) Haz clic en "Nuevo" y escribe la ruta completa de la carpeta bin de Maven (por ejemplo, C:\Program Files\apache-maven-3.8.3\bin).



Abre una nueva ventana de línea de comandos y ejecuta el comando mvn -v para verificar que Maven está instalado correctamente. Deberías ver una salida similar a la siguiente:

```
PS C:\Users\juanf\desarrollo\java> mvn -v
Apache Maven 3.9.0 (9b58d2bad23a66be161c4664ef21ce219c2c8584)
Maven home: C:\apache-maven-3.9.0
Java version: 19.0.2, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-19
Default locale: es_ES, platform encoding: UTF-8
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"
```

Compilar

Para compilar una aplicación en Maven, se pueden seguir los siguientes pasos:

Asegurarse de tener instalado Maven en el equipo. Puedes verificarlo ingresando el comando **mvn -version** en la terminal o símbolo del sistema.

Abrir la terminal o símbolo del sistema y navegar hasta la carpeta raíz del proyecto, donde se encuentra el archivo "pom.xml".

Ejecutar el siguiente comando para compilar la aplicación:

```
mvn compile
```

Si la compilación es exitosa, los archivos compilados se guardarán en el directorio "target/classes".

Además de compilar, es posible que se desee empaquetar la aplicación en un archivo JAR o WAR. Para hacerlo, se puede ejecutar el siguiente comando:

Para empaquetar en formato JAR:

```
mvn package
```

Esto generará un archivo "nombre_del_proyecto.jar" o "nombre_del_proyecto.war" en el directorio "target" del proyecto.

Ejecutar el programa

Una vez compilado en un archivo .jar puede ejecutar el archivo lleno a la carpeta target y ejecutar el programa

```
java -jar [nombre del archivo].jar
```

Testing

Imaginemos que tenemos una fábrica de chocolates y queremos asegurarnos de que cada lote de chocolates que salga de la línea de producción cumpla con los estándares de calidad que esperamos. Para lograr esto, realizamos pruebas en diferentes etapas del proceso de producción, desde la selección de los ingredientes hasta el empaque final del producto.

De manera similar, en el desarrollo de software, el testing es un proceso que nos permite asegurarnos de que el software que estamos creando cumpla con los requisitos y estándares de calidad esperados. Al igual que en la fábrica de chocolates, se realizan pruebas en diferentes etapas del proceso de desarrollo, desde la definición de los requisitos hasta la implementación final del software.

Existen diferentes tipos de pruebas que se pueden realizar durante el proceso de testing de software, como pruebas unitarias, de integración, funcionales, de rendimiento, entre otras. Cada una de estas pruebas tiene como objetivo detectar posibles problemas o errores en el software y garantizar que este funcione de manera adecuada y cumpla con los requisitos esperados.

Al igual que en la fábrica de chocolates, el testing de software es una parte fundamental del proceso de producción y nos permite asegurarnos de que el producto final cumpla con los estándares de calidad y satisfaga las necesidades de los usuarios.

El testing busca encontrar fallas en el producto

El objetivo principal del testing es encontrar fallas o errores en el producto de software para asegurarse de que cumple con los requisitos y especificaciones del cliente. Las fallas pueden ser errores en el código, problemas de rendimiento, problemas de seguridad, problemas de usabilidad, entre otros.

Busca la eficiencia

El testing también tiene como objetivo ser lo más eficiente posible en términos de tiempo y costo. Es importante realizar el testing de manera oportuna y no dejarlo para el final del proceso de desarrollo de software, ya que esto puede generar un retraso en la entrega del producto y aumentar los costos. Por lo tanto, se busca encontrar un equilibrio entre la eficiencia y la efectividad del testing.

Busca encontrar la mayor cantidad de fallas

El objetivo del testing es encontrar la mayor cantidad de fallas posible. Esto se logra mediante la utilización de diferentes técnicas y herramientas de testing, que permiten identificar problemas en diferentes áreas del software.

Intenta no detectar fallas que en realidad no lo son

El testing también busca evitar detectar falsos positivos, es decir, errores que no son en realidad errores. Esto se logra mediante la utilización de diferentes técnicas y herramientas de testing que permiten filtrar y priorizar los errores reales.

Pretende encontrar las fallas más importantes

El testing tiene como objetivo encontrar las fallas más importantes o críticas, es decir, aquellas que tienen un mayor impacto en el funcionamiento del software o en la experiencia del usuario. Estas fallas se priorizan para que se puedan resolver de manera oportuna y asegurarse de que el producto de software sea lo más confiable posible.

Las condiciones de prueba

Las condiciones de prueba son un componente clave en el proceso de pruebas de software. Son descripciones de situaciones específicas que se deben probar en el software, para determinar si el sistema responde como se espera. Las condiciones de prueba se crean para verificar que el software cumpla con los requisitos del usuario y que no tenga errores, defectos o problemas de rendimiento.

Por ejemplo, supongamos que se está desarrollando una aplicación de comercio electrónico y se debe verificar si el proceso de pago en línea funciona correctamente. Una condición de prueba para esta situación podría ser: "Al hacer clic en el botón de pago, el sistema debe solicitar información de pago del usuario y validar que la información sea correcta antes de continuar con la transacción". En este caso, la condición de prueba es la descripción de la situación que se debe verificar en el sistema para asegurarse de que el proceso de pago funcione correctamente.

Es importante tener en cuenta que cada condición de prueba debe ser específica, clara y medible, para que sea posible determinar si se cumplió o no. Además, se deben crear varias condiciones de prueba para cubrir diferentes situaciones y casos de uso del software, y así poder verificar que todo el sistema funcione correctamente en diferentes situaciones.

Casos de prueba

Los casos de prueba son una descripción detallada de las entradas, condiciones y valores esperados que se utilizarán en una prueba específica. En otras palabras, son conjuntos de datos y parámetros que se utilizan para probar una determinada función o característica de un software.

Cada caso de prueba se compone de un conjunto de valores de entrada, que se utilizan para probar una determinada funcionalidad del software, y un conjunto de valores de salida esperados. Estos valores de entrada y salida pueden ser numéricos, lógicos, textuales o cualquier otro tipo de datos que se utilicen en el programa.

Por ejemplo, si se está probando una función de una aplicación de gestión de inventario que permite agregar nuevos productos a una base de datos, un caso de prueba podría ser agregar un producto con un nombre, una descripción, una cantidad y un precio determinados, y luego verificar que los datos ingresados se han guardado correctamente en la base de datos.

Es importante señalar que los casos de prueba deben ser lo suficientemente amplios y variados como para probar todas las posibles combinaciones de entradas y condiciones de la función o característica que se está probando. Esto asegura que cualquier error o falla en el software se detecte y se corrija antes de que se publique o se implemente en producción.

Tipos de Tests

Pruebas Unitarias

Los tests unitarios son una técnica de pruebas de software en la que se evalúa cada unidad de código de forma aislada e independiente de otras partes del sistema. En general, una unidad de código es la mínima cantidad de código que puede ser probada de manera aislada, tal como una función o un método.

En la práctica, los tests unitarios se implementan mediante el uso de frameworks de pruebas unitarias, como JUnit en Java. Estos frameworks proveen una serie de herramientas para definir y ejecutar pruebas unitarias, como aserciones (assertions) para verificar el comportamiento esperado de una unidad de código.

Los tests unitarios pueden ser de caja negra o de caja blanca. Los tests de caja negra evalúan el comportamiento de una unidad de código sin conocer su implementación interna, mientras que los tests de caja blanca conocen los detalles de la implementación y pueden evaluar el comportamiento de cada camino de ejecución dentro de la unidad de código.

Los tests unitarios son importantes porque permiten detectar y corregir errores en las unidades de código de forma temprana, lo que puede ahorrar tiempo y recursos en etapas posteriores del ciclo de vida del software. Además, los tests unitarios facilitan la refactorización del código, ya que permiten verificar que los cambios realizados no introducen errores en el comportamiento de las unidades de código.

Pruebas de Integración

Son pruebas que se realizan después de las pruebas unitarias, y se enfocan en evaluar el correcto funcionamiento de la interacción entre diferentes componentes del sistema. El objetivo de estas pruebas es verificar que los distintos módulos o componentes del sistema funcionan de manera correcta y coordinada cuando se integran.

Pruebas de Aceptación de Usuario (UAT)

Son pruebas realizadas por los usuarios finales o representantes del cliente, con el objetivo de verificar que el sistema cumpla con las especificaciones y necesidades

del usuario. Estas pruebas se realizan una vez que se han completado las pruebas unitarias e integración.

Pruebas de Stress

Son pruebas que se realizan para evaluar el comportamiento del sistema bajo situaciones de alta carga o demanda, con el objetivo de identificar los límites y capacidades del sistema. Estas pruebas se realizan para asegurar que el sistema puede manejar la carga prevista y que no se produzcan fallos críticos.

Pruebas de Volumen y Performance

Son pruebas que se realizan para evaluar la capacidad del sistema para manejar grandes cantidades de datos y procesarlos de manera eficiente. Estas pruebas se realizan para verificar que el sistema puede funcionar correctamente en situaciones de alto volumen de información y que no se produzcan retrasos o fallos.

Pruebas de Regresión

Son pruebas que se realizan para verificar que las modificaciones realizadas en el código no afecten negativamente a funcionalidades ya existentes. Estas pruebas se realizan después de la implementación de cambios en el código.

Pruebas Alfa y Beta

Son pruebas que se realizan en las fases finales del proceso de desarrollo y antes del lanzamiento del software. Las pruebas alfa se realizan en un entorno controlado y supervisado, y se enfocan en verificar el correcto funcionamiento de todas las funcionalidades del software. Las pruebas beta se realizan en un entorno más abierto y con un grupo limitado de usuarios, con el objetivo de verificar el correcto funcionamiento del software en un ambiente real y detectar posibles errores o fallos que no hayan sido detectados durante las pruebas alfa.

jUnit

JUnit es un framework de pruebas unitarias para el lenguaje de programación Java. Es una herramienta ampliamente utilizada por los desarrolladores para realizar pruebas de unidad en aplicaciones Java. JUnit es una librería que proporciona clases y métodos para realizar pruebas unitarias en aplicaciones Java.

JUnit permite a los desarrolladores escribir casos de prueba para comprobar la funcionalidad de los métodos y clases individuales en el código Java. Estas pruebas se escriben en una clase separada que contiene los métodos de prueba, y se ejecutan automáticamente con el fin de verificar que el comportamiento del código sea el esperado.

Las pruebas se basan en aserciones (assertions), que son declaraciones booleanas que comprueban si una condición es verdadera o falsa. Por ejemplo, se puede afirmar que el resultado de una operación matemática es el esperado o que una condición específica es verdadera. Si la aserción falla, se considera que la prueba falló.

JUnit proporciona una gran variedad de anotaciones (annotations) que se utilizan para escribir pruebas unitarias. Estas anotaciones permiten especificar la configuración de las pruebas, cómo se deben ejecutar y cómo se deben manejar los errores.

Algunas de las principales anotaciones de JUnit son:

@Test: indica que un método es un caso de prueba.

@Before: indica que un método debe ser ejecutado antes de cada caso de prueba.

@After: indica que un método debe ser ejecutado después de cada caso de prueba.

@BeforeClass: indica que un método debe ser ejecutado antes de todos los casos de prueba.

@AfterClass: indica que un método debe ser ejecutado después de todos los casos de prueba.