Contenido

1. Estructuras de control

Si aún no dominas el manejo de los **operadores relacionales** y de los **operadores lógicos** te recomiendo que los repases a fondo porque los necesitarás a la hora de abordar este punto del tema.

1.1. Estructuras de selección

1.1.1. Condicional simple: if

La estructura condicional **if** en Java se utiliza para tomar decisiones en el flujo de un programa. Permite que ciertos bloques de código se ejecuten sólo si una condición específica se evalúa como verdadera (**true**).

La sintaxis básica del condicional **if** en Java es la siguiente:

```
if (condición) {
    // Código a ejecutar si la condición es verdadera
}
```

- **condición**: Es una expresión que se evalúa como un valor booleano (**true** o **false**).
- **{}**: Las llaves definen el ***bloque de código*** que se ejecutará si la condición es verdadera. Dicho bloque de código puede contener una o más instrucciones.

¿CÓMO FUNCIONA?

1. La **condición** dentro del paréntesis se evalúa primero.
2. Si la **condición** es **true**, el bloque de código dentro de las llaves **{}** se ejecuta.
3. Si la **condición** es **false**, el bloque de código se omite y el programa continúa con la siguiente instrucción después del bloque **if**.

EJEMPLO: ¿Qué pasaría si el valor de 'numero' fuese 3?

```
int numero = 10;

// Uso del condicional if
if (numero > 5) {
    System.out.println("El número es mayor que 5.");
}

// El programa continúa aquí
System.out.println("Fin del programa.");
```

1.1.2. Condicional doble: if-else

La estructura **if-else** en Java se utiliza para tomar decisiones en el flujo de un programa. Permite ejecutar un bloque de código si una condición es verdadera (**true**) y otro bloque de código si la condición es falsa (**false**).

La sintaxis básica de la estructura **if-else** en Java es la siguiente:

```
if (condición) {
    // Bloque de código que se ejecuta si la condición es verdadera
} else {
    // Bloque de código que se ejecuta si la condición es falsa
}
```

- **if**: Palabra clave que inicia la estructura condicional.
- **condición**: Expresión booleana que se evalúa como verdadera o falsa.
- **else**: Palabra clave que introduce el bloque de código que se ejecutará si la condición es falsa.

¿CÓMO FUNCIONA?

1. La **condición** dentro del paréntesis se evalúa primero.
2. Si la **condición** es **true**, el bloque de código dentro de las llaves **{}** que acompañan al **if** se ejecuta.
3. Si la **condición** es **false**, el bloque de código dentro de las llaves **{}** que acompañan al **else** se ejecuta.
4. Siempre se ejecutará un bloque de código u otro en función del valor de la condición. Nunca se ejecutarán los dos bloques de código.

EJEMPLO 1: ¿Qué pasaría si el valor de 'numero' fuese -3?

```
int numero = 10;  // Puedes cambiar este valor para probar diferentes casos

if (numero > 0) {
    System.out.println("El número es positivo.");
} else {
    System.out.println("El número no es positivo.");
}
```

EJEMPLO 2:

```java
int numero = 10;  // Cambia este valor para probar diferentes casos

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

1.1.3. Condicional múltiple: switch

La estructura condicional múltiple **switch** en Java se utiliza para ejecutar uno de los muchos bloques de código posibles. Es una alternativa más limpia y eficiente a una serie de sentencias **if-else if-else** cuando se necesita comparar una variable contra múltiples valores constantes.

Veámoslo con un **ejemplo**. Supongamos que tenemos una variable **calificacion** y queremos imprimir un mensaje en función de su valor. Con lo que hemos visto hasta, deberíamos hacerlo de la siguiente forma:

```java
int calificacion = 85;

if (calificacion >= 90) {
    System.out.println("Excelente");
} else if (calificacion >= 80) {
    System.out.println("Muy bien");
} else if (calificacion >= 70) {
    System.out.println("Bien");
} else if (calificacion >= 60) {
    System.out.println("Regular");
} else {
    System.out.println("Insuficiente");
}
```

Pero haciendo uso de la sentencia **switch** podemos conseguir el mismo comportamiento de una forma más limpia:

```
int calificacion = 85;

switch (calificacion / 10) {
    case 9:
    case 10:
        System.out.println("Excelente");
        break;
    case 8:
        System.out.println("Muy bien");
        break;
    case 7:
        System.out.println("Bien");
        break;
    case 6:
        System.out.println("Regular");
        break;
    default:
        System.out.println("Insuficiente");
}
```

La sintaxis básica de la estructura **switch** en Java es la siguiente:

```
switch (expresion) {
    case valor1:
        // Bloque de código a ejecutar si expresion == valor1
        break;
    case valor2:
        // Bloque de código a ejecutar si expresion == valor2
        break;
    // ...
    default:
        // Bloque de código a ejecutar si la expresión no coincide con ningún valor
}
```

¿CÓMO FUNCIONA?

1. **Expresión de Comparación**: La expresión dentro del **switch** debe resultar en un valor que pueda ser comparado con los valores en los **case**. Puede ser un número entero, un carácter o una enumeración. No se permiten tipos de datos como **float**, **double**, ni objetos (excepto **String** desde Java 7 en adelante).
2. **Case Constantes**: Los valores en los **case** deben ser constantes y únicos.
3. **Break**: La sentencia **break** se utiliza para salir del bloque **case**. Si se omite, se ejecutarán todos los bloques **case** siguientes hasta encontrar un **break** o llegar al final del **switch**.
4. **Default**: Es opcional y se ejecutará si ninguno de los **case** coincide con la expresión.

EJEMPLO 1:

```
char operador = '+';
int num1 = 10, num2 = 20;

switch (operador) {
    case '+':
        System.out.println(num1 + num2);
        break;
    case '-':
        System.out.println(num1 - num2);
        break;
    case '*':
        System.out.println(num1 * num2);
        break;
    case '/':
        if (num2 != 0) {
            System.out.println((double) num1 / num2);
        } else {
            System.out.println("División por cero");
        }
        break;
    default:
        System.out.println("Operador no válido");
}
```

EJEMPLO 2: Supongamos que estamos desarrollando un sistema para una tienda y queremos manejar los estados de un pedido. Podemos definir una ***enumeración*** para los estados y luego usar un **switch** para manejar las acciones correspondientes.

```
// Definición de la enumeración para los estados de un pedido
enum EstadoPedido {
    PENDIENTE,
    EN_PROCESO,
    ENVIADO,
    ENTREGADO,
    CANCELADO
}
```

```
// Variable que contiene el estado actual del pedido
EstadoPedido estadoActual = EstadoPedido.EN_PROCESO;

// Uso de switch con enumeración
switch (estadoActual) {
    case PENDIENTE:
        System.out.println("El pedido está pendiente de procesamiento.");
        break;
    case EN_PROCESO:
        System.out.println("El pedido está siendo procesado.");
        break;
    case ENVIADO:
        System.out.println("El pedido ha sido enviado.");
        break;
    case ENTREGADO:
        System.out.println("El pedido ha sido entregado exitosamente.");
        break;
    case CANCELADO:
        System.out.println("El pedido ha sido cancelado.");
        break;
    default:
        System.out.println("Estado desconocido.");
}
```

En este ejemplo, la enumeración **EstadoPedido** define los posibles estados de un pedido (que internamente tomará el valor de números enteros). Luego utilizamos una sentencia **switch** para ejecutar un bloque de código específico según el valor del **estadoActual** de un pedido.

Este enfoque hace que el código sea más legible y mantenible, especialmente cuando se trabaja con un conjunto fijo de constantes, como es el caso de los estados de un pedido en este ejemplo.

1.2. Estructuras de repetición

1.2.1. Bucle controlado por condición: while

El bucle **while** es una estructura de control que permite ejecutar un bloque de código de manera repetitiva mientras se cumpla una condición determinada. Es especialmente útil cuando no se conoce de antemano el número de iteraciones que se deben realizar.

La sintaxis básica del bucle **while** en Java es la siguiente:

```
while (condición) {
    // Código a ejecutar mientras la condición sea verdadera
}
```

¿CÓMO FUNCIONA?

1. **Evaluación de la Condición**: Antes de cada iteración, se evalúa la condición especificada entre paréntesis.
2. **Ejecución del Bloque de Código**: Si la condición es verdadera (**true**), se ejecuta el bloque de código dentro de las llaves **{}**.
3. **Reevaluación**: Tras ejecutar el bloque de código, se vuelve a evaluar la condición. Si sigue siendo verdadera, el bloque de código se ejecuta de nuevo.
4. **Terminación**: El bucle se detiene cuando la condición se evalúa como falsa (**false**).

EJEMPLO:

```
int contador = 1; // Inicializamos el contador

while (contador <= 10) { // Condición
    System.out.println("Número: " + contador); // Acción
    contador++; // Incremento del contador
}
```

En este ejemplo, el bucle **while** comienza con **contador** igual a 1. Mientras **contador** sea menor o igual a 10, se imprimirá el valor de **contador** y luego se incrementará en 1. Cuando **contador** llegue a 11, la condición **contador <= 10** será falsa, y el bucle se detendrá.

IMPORTANTE: Como programador, *debes asegurarte que la condición que controla el bucle va a dejar de ser cierta en algún momento para que así el bucle pueda terminar* y el programa siga su ejecución porque de no ser así el bucle se ejecutará infinitamente y el usuario verá que el programa "se ha colgado".

1.2.2. Bucle controlado por condición: do-while

El bucle **do-while** es una estructura de control en Java que permite ejecutar un bloque de código mientras una condición específica sea verdadera. A diferencia del bucle **while**, el bucle **do-while** garantiza que el bloque de código se ejecute al menos una vez, ya que la evaluación de la condición se realiza después de la primera ejecución del bloque.

La sintaxis básica del bucle **do-while** en Java es la siguiente:

```
do {
    // Código a ejecutar
} while (condición);
```

¿CÓMO FUNCIONA?

1. Primero, el bloque de código dentro del **do** se ejecuta una vez.
2. Luego, se evalúa la condición especificada en el **while**.
3. Si la condición es verdadera, el bloque de código se ejecuta de nuevo.
4. Este proceso se repite hasta que la condición se evalúe como falsa.

EJEMPLO:

```
int suma = 0;
int i = 1;

do {
    suma += i;  // suma = suma + i
    i++;        // i = i + 1
} while (i <= 10);

System.out.println("La suma de los números del 1 al 10 es: " + suma);
```

En este ejemplo, la variable **i** se inicializa con 1 y la variable **suma** con 0. El bucle **do-while** se ejecuta mientras **i** sea menor o igual a 10. Dentro del bucle, se suma el valor de **i** a **suma**, y luego se incrementa **i** en 1. Finalmente, se imprime la suma total, que debería ser 55.

IMPORTANTE: Como programador, *debes asegurarte que la condición que controla el bucle va a dejar de ser cierta en algún momento para que así el bucle pueda terminar* y el programa siga su ejecución porque de no ser así el bucle se ejecutará infinitamente y el usuario verá que el programa "se ha colgado".

1.2.3. Bucle controlado por repetición: for

El bucle **for** en Java se utiliza para ejecutar un bloque de código un número determinado de veces. Es especialmente útil cuando sabes de antemano cuántas veces se debe ejecutar un bloque de código.

La sintaxis básica del bucle **for** es la siguiente:

```
for (inicialización; condición; actualización) {
    // Código a ejecutar
}
```

¿CÓMO FUNCIONA?

- **Inicialización**: Se ejecuta una vez al comienzo del bucle.
- **Condición**: Se evalúa antes de cada iteración. Si la condición es **true**, el bloque de código dentro del bucle se ejecutará.
- **Actualización**: Se ejecuta después de cada iteración del bucle.

EJEMPLO:

```
// Imprimir números del 1 al 10
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

- **int i = 1;**: Inicializamos una variable **i** con el valor 1.
- **i <= 10;**: Esta es la condición. Mientras **i** sea menor o igual a 10, el bucle continuará.
- **i++**: Esto incrementa el valor de **i** en 1 después de cada iteración.

IMPORTANTE: Como programador, *debes asegurarte que la condición que controla el bucle va a dejar de ser cierta en algún momento para que así el bucle pueda terminar* y el programa siga su ejecución porque de no ser así el bucle se ejecutará infinitamente y el usuario verá que el programa "se ha colgado".

1.3. Estructuras de salto

En Java, existen tres estructuras de salto que permiten controlar el flujo de ejecución del programa saltando a diferentes partes del código. Estas estructuras son:

1.3.1. break

La instrucción **break** se utiliza para salir de un bucle (*for*, *while*, o *do-while*) o de un bloque *switch* antes de que se haya completado normalmente. Cuando se ejecuta la instrucción *break*, el control del programa sale del bucle o del bloque *switch* y continúa con la siguiente instrucción después del bucle o bloque. Es especialmente útil cuando se quiere terminar prematuramente un bucle basándose en una condición.

EJEMPLO:

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break; // Terminar el bucle cuando i sea igual a 5
    }
    System.out.println(i);
}
```

1.3.2. continue

La instrucción **continue** se utiliza para saltar a la siguiente iteración de un bucle sin ejecutar el resto del código que sigue después de ella dentro de la misma iteración. Es útil cuando se quiere omitir ciertas iteraciones del bucle basándose en una condición.

EJEMPLO:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue; // Omitir la iteración cuando i sea igual a 3
    }
    System.out.println(i);
}
```

1.3.3. return

La instrucción **return** se utiliza para salir de una función y devolver un valor si la función tiene un tipo de retorno diferente a *void*. Cuando se ejecuta la instrucción *return*, el control del programa regresa al lugar donde se llamó a la función y se proporciona el valor especificado como resultado de la función.

EJEMPLO:

```
public int sumar(int a, int b) {
    int resultado = a + b;
    return resultado; // Devolver el resultado de la suma
}
```

Estas estructuras de salto son útiles para controlar el flujo de ejecución del programa en diferentes situaciones y optimizar el comportamiento del código en función de las condiciones específicas.

2. Control de excepciones

2.1. Excepciones

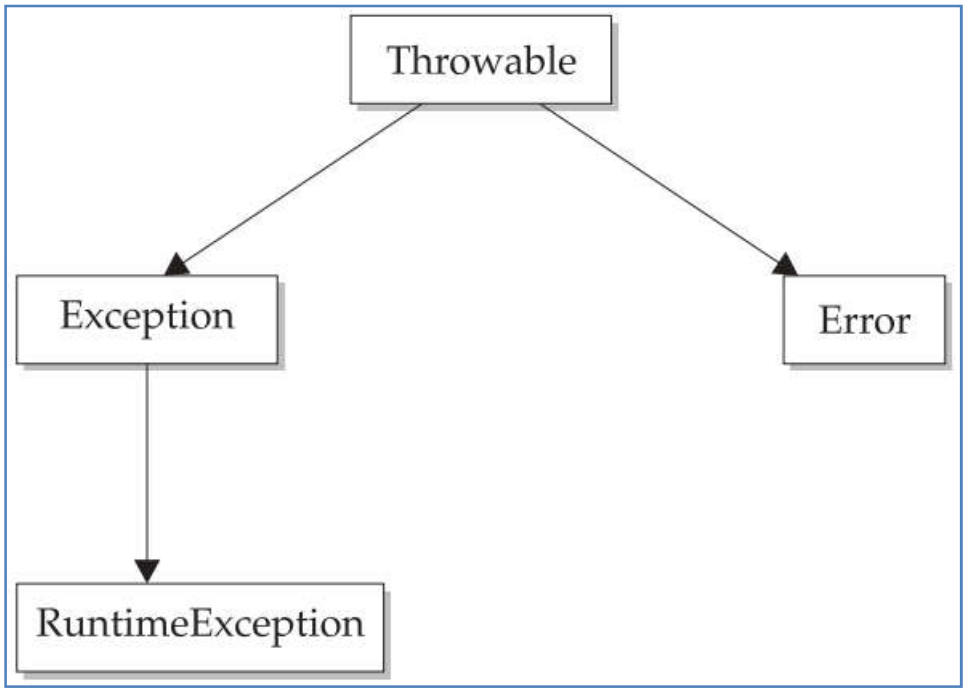
Las **excepciones** en Java son eventos que alteran el flujo normal de ejecución de un programa. Son objetos que representan errores o situaciones inesperadas. Java utiliza un modelo de manejo de excepciones robusto para capturar y manejar estos errores de manera eficiente.

2.2. Jerarquía de excepciones

En Java, el concepto de **jerarquía de excepciones** se refiere a la organización estructurada de las clases de excepción dentro de la biblioteca estándar de Java. La jerarquía de excepciones *permite una gestión estructurada y flexible de los errores*.

Esta jerarquía es crucial para entender cómo se manejan los diferentes tipos de errores y excepciones en Java. En ella, todas las clases de excepción tienen una relación de herencia que comienza desde la clase **java.lang.Throwable**.

A continuación se detalla una descripción general simplificada de la jerarquía:



Fuente: Java – The Complete Reference Twelfth Edition – Herbert Schildt

1. **Throwable**: Esta es la superclase para todas las clases de errores y excepciones. Tiene dos subclases principales: **Error** y **Exception**.

    - **Error**: Representa problemas graves que una aplicación normalmente no debería intentar atrapar. Estos son problemas que normalmente están fuera del control del usuario o del programador, como **OutOfMemoryError** o **StackOverflowError**.

    - **Exception**: Representa condiciones que una aplicación razonablemente podría querer capturar. Las **Exception** se dividen en dos categorías principales:

        - **Excepciones Comprobadas (Checked Exceptions)**: Son aquellas que el compilador obliga a manejar. Estas excepciones generalmente se relacionan con condiciones externas a la aplicación, como problemas de I/O (Input/Output en inglés que sería E/S o Entrada/Salida en español), que un programa bien escrito debería anticipar y recuperarse. Un ejemplo es **IOException**.

        - **Excepciones No Comprobadas (Unchecked Exceptions)**: Incluyen **RuntimeException** y sus subclases. Estas excepciones no necesitan ser declaradas en una cláusula **throws**, y el compilador no requiere que se manejen explícitamente. Suelen indicar errores de programación, como intentar acceder a un objeto nulo (**NullPointerException**) o dividir por cero (**ArithmeticException**).

Esta jerarquía es fundamental en Java ya que permite manejar errores de manera estructurada y controlada, facilitando así la escritura de programas más robustos y fiables. Además, al entender esta jerarquía, **los desarrolladores pueden crear sus propias clases de excepción personalizadas** que se ajusten a esta estructura, permitiendo un manejo de errores más específico y detallado en sus aplicaciones.

Ahora vamos a profundizar en los métodos más relevantes que implementa la clase base de la jerarquía de excepciones y que, por tanto, estarán presentes en sus subclases.

La clase base **java.lang.Throwable** es la superclase de todas las clases de errores y excepciones en Java.

- **Métodos Importantes**:

    - **getMessage()**: Devuelve el mensaje detallado de la Throwable.

    - **getCause()**: Retorna la causa que produjo la excepción o **null** si la causa es inexistente o desconocida.

Ahora vamos a ver un resumen de las subclases de esta jerarquía:

- Excepciones Comprobadas: Subclases de **Exception**
    - **IOException**: Captura fallos de entrada/salida.
    - **SQLException**: Maneja errores relacionados con la base de datos.
    - **ClassNotFoundException**: Indica que no se encontró una clase.
- Excepciones No Comprobadas: Subclases de **RuntimeException**
    - **NullPointerException**: Indica que se está intentando utilizar una referencia nula.
    - **IndexOutOfBoundsException**: Señala que se ha accedido a un índice ilegal de algún tipo de colección.
    - **ArithmeticException**: Reporta errores en operaciones aritméticas, como la división por cero.
- Errores: Subclases de **Error**
    - **OutOfMemoryError**: Indica que la JVM no tiene más memoria disponible.
    - **StackOverflowError**: Se produce cuando el tamaño de la pila excede su límite.
    - **NoClassDefFoundError**: Error que indica que una clase no pudo ser encontrada o cargada.

Por último, te muestro una tabla con las excepciones más comunes en Java:

Excepción	Descripción
`` `ArithmeticException` ``	Errores en operaciones aritméticas, como división por cero.
`` `NullPointerException` ``	Referencia a un objeto `` `null` `` donde se requiere un objeto.
`` `ArrayIndexOutOfBoundsException` ``	Índice fuera de los límites de un array.
`` `ClassCastException` ``	Intento fallido de conversión de tipo de objeto.
`` `IOException` ``	Problemas de entrada/salida (I/O).
`` `FileNotFoundException` ``	Archivo no encontrado en el sistema de archivos.
`` `NumberFormatException` ``	Formato numérico incorrecto al convertir cadenas en números.
`` `IllegalArgumentException` ``	Argumento ilegal o inapropiado para un método.

2.3. Manejo de excepciones

En Java, el manejo de excepciones se realiza a través de cinco palabras: **try, catch, throw, throws** y **finally.** Vamos a verlas en detalle:

- **try** y **catch** permite al programador declarar un bloque de código que puede dar lugar a una excepción y, además, definir otro bloque de código que manejará dicha excepción en caso de que se produzca. Veamos un ejemplo:

```
try {
    // Intentamos una división por cero
    int division = 10 / 0;
} catch (ArithmeticException e) {
    // Capturamos la excepción y mostramos su mensaje
    System.out.println("Ha ocurrido un error: " + e.getMessage());
}
```

  Aquí hemos puesta en el bloque **try** el código que da lanza una excepción, mientras que en el bloque **catch** está el bloque que maneja dicha excepción (en este caso el manejo consiste en mostrar un mensaje por pantalla con el texto de la excepción)

- **throw:** Permite al programador lanzar explícitamente una excepción. Veamos un ejemplo:

```
public void verificarEdad(int edad) {
    if (edad < 18) {
        throw new ArithmeticException("Acceso denegado - Eres menor de edad.");
    }
}
```

- **throws:** Se utiliza en la firma de un método para indicar que puede lanzar una excepción. Delega la responsabilidad de manejar la excepción al método que lo invoca.

  Veamos un ejemplo de declaración de un método con esta estructura:

```
public void miMetodo() throws IOException {
    // código que puede lanzar IOException
}
```

Veamos ahora cómo deberíamos gestionar la llamada a dicho método para poder usarlo de forma adecuada teniendo en cuenta que pueden producirse excepciones:

```
try {
    // Intentamos ejecutar miMetodo
    miMetodo();
} catch (IOException e) {
    // Manejo de la IOException
    System.out.println("Se ha producido una IOException: " + e.getMessage());
    // Aquí podríamos incluir más lógica para manejar la excepción
}
```

- **finally:** Bloque que siempre se ejecuta, independientemente de si se lanzó una excepción o no. Útil para cerrar recursos, como flujos de archivos.

  Veamos un ejemplo donde, además, podemos observar que los bloques try-catch también se pueden anidar:

```
BufferedReader reader = null;
try {
    reader = new BufferedReader(new FileReader("miArchivo.txt"));
    String linea = reader.readLine();
    while (linea != null) {
        System.out.println(linea);
        linea = reader.readLine();
    }
} catch (IOException e) {
    System.out.println("Error al leer el archivo: " + e.getMessage());
} finally {
    // El bloque finally se ejecuta siempre, tanto si hay una excepción como si no.
    try {
        if (reader != null) {
            reader.close(); // Cerramos el archivo
        }
    } catch (IOException e) {
        System.out.println("Error al cerrar el archivo: " + e.getMessage());
    }
}
```

Por último, debe quedar claro que **si durante la ejecución del programa se lanza una excepción que no sea capturada y manejada por él, se abortará su ejecución** (o "petará" o "crasheará" o como lo que queráis llamar).

3. Aserciones

En Java, las **aserciones** son una característica que permite realizar comprobaciones en el código durante el tiempo de ejecución para asegurarse de que ciertas condiciones se cumplen. Las aserciones sirven para detectar errores de programación y condiciones inesperadas en fases de desarrollo y pruebas. No se recomienda usarlas para validaciones en el código de producción, ya que pueden ser deshabilitadas en tiempo de ejecución y ***no están diseñadas para manejar situaciones de error en un entorno de producción***.

Las aserciones se utilizan con la palabra clave **assert**, seguida de una condición booleana y, opcionalmente, un mensaje de error:

```
assert [condición booleana] : "Mensaje de error";
```

Si la condición es **true**, el programa continúa su ejecución normalmente. Si la condición es **false**, se lanza una excepción del tipo **AssertionError** con el mensaje de error proporcionado.

EJEMPLO: Supongamos que tienes una función que calcula la raíz cuadrada de un número y quieres asegurarte de que el número proporcionado no sea negativo.

```
public class Test {
    public static void main(String[] args) {
        double result = sqrt(25);
        System.out.println("La raíz cuadrada es: " + result);

        result = sqrt(-1); // Esto provocará un AssertionError
    }

    static double sqrt(double value) {
        assert value >= 0 : "El valor debe ser no negativo";
        return Math.sqrt(value);
    }
}
```

En este ejemplo, si intentas calcular la raíz cuadrada de un número negativo, se lanzará un **AssertionError** con el mensaje "El valor debe ser no negativo". Las aserciones son muy útiles durante el desarrollo y las pruebas para validar suposiciones y detectar errores lógicos temprano en el proceso de desarrollo.

4. Prueba, depuración y documentación de una aplicación

4.1. Pruebas

En el contexto de desarrollo de software, una **prueba** es un proceso que se realiza para asegurar que un fragmento de código o una aplicación completa funcione correctamente. El **objetivo de las pruebas** es identificar errores, deficiencias o requisitos no cumplidos en comparación con los requisitos iniciales.

4.1.1. Tipos de Pruebas en Desarrollo de Software

1. **Pruebas Unitarias:** Verifican la correcta funcionalidad de unidades individuales de código (como funciones, métodos o clases). Son las más básicas y se enfocan en la menor parte de la aplicación. Herramientas como JUnit son muy utilizadas en Java.

2. **Pruebas de Integración:** Comprueban cómo diferentes módulos o servicios trabajan juntos. Son importantes para asegurar que las combinaciones de unidades funcionen correctamente en conjunto.

3. **Pruebas de Sistema:** Evalúan el sistema completo para asegurarse de que cumple con los requisitos especificados. Estas pruebas consideran aspectos como el rendimiento, la seguridad y el comportamiento bajo condiciones específicas.

4. **Pruebas de Aceptación:** Realizadas por usuarios finales para confirmar que el sistema hace lo que ellos esperan y necesitan. Se enfocan en la experiencia del usuario y la conformidad con los requisitos del negocio.

4.1.2. Importancia de las Pruebas

- **Calidad del Software:** Aseguran que el software funcione correctamente y cumpla con los requisitos establecidos.

- **Prevención de Errores:** Ayudan a identificar y corregir errores antes de que el software se despliegue o se entregue al usuario final.

- **Confianza en el Desarrollo:** Permiten a los desarrolladores hacer cambios y agregar características nuevas con la seguridad de que no se introducirán errores en partes ya probadas del software.

- **Documentación:** Las pruebas, especialmente las pruebas unitarias, pueden servir como una forma de documentación del código, mostrando cómo se espera que funcione.

Las pruebas son una parte esencial del desarrollo de software moderno, especialmente en metodologías ágiles y en la integración y despliegue continuos, donde la capacidad de probar rápidamente y con fiabilidad es crucial para el ciclo de vida del desarrollo de software.

4.2. Depuración

La **depuración** es el proceso de identificar, aislar y corregir errores ("bugs") en un programa de software. Este proceso es esencial en el desarrollo de software y es una habilidad crítica para cualquier programador. La depuración no sólo implica corregir errores evidentes, sino también resolver problemas de lógica y comportamientos inesperados en un programa.

4.2.1. Aspectos Clave de la Depuración

1. **Identificación del Problema:** Se trata de entender el error que se está presentando, lo cual puede incluir leer mensajes de error, revisar logs o replicar condiciones bajo las cuales el error ocurre.

2. **Aislamiento del Problema:** Una vez identificado el error, el siguiente paso es aislar la parte específica del código donde ocurre. Esto puede implicar revisar el código fuente, usar herramientas de depuración para ejecutar el programa paso a paso (stepping), y revisar el estado de las variables.

3. **Corrección del Error:** Después de encontrar y aislar el error, se realiza la corrección. Esto puede ser tan simple como arreglar un tipo o tan complejo como reescribir grandes secciones de código.

4. **Pruebas Posterior a la Corrección:** Una vez corregido el error, se deben realizar pruebas para asegurarse de que el problema se ha resuelto y que no se han introducido nuevos errores.

4.2.2. Herramientas de Depuración

- **Depuradores Integrados en IDEs:** Herramientas como Eclipse o IntelliJ IDEA para Java tienen depuradores integrados que permiten ejecutar el código paso a paso, establecer puntos de interrupción (breakpoints), y observar el estado de las variables.

- **Logging:** La escritura de mensajes de log en el código puede ayudar a entender qué está pasando durante la ejecución del programa.

- **Herramientas de Análisis Estático:** Estas herramientas revisan el código fuente en busca de patrones comunes de errores sin ejecutar el programa.

4.2.3. Importancia de la Depuración

La depuración es crucial para el desarrollo de software de alta calidad. No solo se trata de arreglar errores, sino también de comprender por qué ocurren. Esto puede llevar a un mejor diseño del software y a prácticas de codificación más robustas. Además, la depuración efectiva a menudo requiere y fomenta habilidades importantes como el pensamiento crítico, la atención al detalle y la paciencia.

4.3. Documentación

La **documentación** en el contexto del desarrollo de software se refiere al conjunto de documentos escritos que acompañan al código fuente y explican cómo funciona el software, cómo se usa, cómo se mantiene y cómo se desarrolló. Es una parte esencial del proceso de desarrollo de software, y su importancia es crítica tanto para los desarrolladores como para los usuarios finales.

4.3.1. Tipos de Documentación en Desarrollo de Software

1. **Documentación del Código Fuente:**
    - **Comentarios en el Código:** Explicaciones directamente en el código fuente que ayudan a entender la funcionalidad de ciertas secciones del código.
    - **Documentación API:** Descripciones de las interfaces de programación, usualmente generadas automáticamente por herramientas como JavaDoc en el caso de Java.
2. **Documentación Técnica:**
    - **Manuales de Desarrollo:** Guías y especificaciones para desarrolladores que explican cómo contribuir al proyecto o cómo extender el software.
    - **Documentación de Arquitectura:** Descripciones detalladas de la arquitectura del software, incluyendo diagramas y explicaciones de cómo los diferentes componentes interactúan.
3. **Documentación para Usuarios:**
    - **Manuales de Usuario:** Guías que explican cómo usar el software, dirigidas a usuarios finales.
    - **FAQs y Ayuda en Línea:** Preguntas frecuentes y recursos de ayuda para resolver dudas comunes de los usuarios.

4.3.2. Importancia de la Documentación

- **Facilita la Comprensión:** Ayuda a los nuevos desarrolladores a entender rápidamente el código y la arquitectura del software.
- **Mejora la Mantenibilidad:** Una buena documentación facilita el mantenimiento y la actualización del software.
- **Fomenta la Colaboración:** Permite que los equipos de desarrollo colaboren de manera más efectiva.
- **Asistencia a Usuarios:** Proporciona a los usuarios la información necesaria para utilizar el software de manera eficiente.

4.3.3. Buenas Prácticas en Documentación

- **Mantenerla Actualizada:** La documentación debe actualizarse junto con el software para garantizar su relevancia.
- **Clara y Concisa:** Debe ser fácil de entender y directa, evitando información superflua.
- **Accesible:** Debería estar fácilmente disponible para aquellos que la necesiten.
- **Consistente:** Usar un estilo y formato consistentes en toda la documentación.

Se debe destacar la importancia de la documentación no solo como una práctica profesional, sino también como una herramienta para mejorar vuestro propio proceso de aprendizaje y comprensión del desarrollo de software.

5. Despedida

Ya tienes el poder para programar en lenguajes que sigan el paradigma de la programación estructurada. ¡Vamos a por la programación orientada a objetos!