|  |  |
| --- | --- |
| Présentation de la Javadoc - OpenClassrooms  JAVADOC EN JUNIT  Repositorio: git@github.com:mikelbarbe/AD\_2\_JavaDoc.git | Autores   * Mikel Barberena GitHub: mikelbarbe * Almudena   GitHub: AlmudenaGB   * Beatriz Waliño   GitHub: Beatriz Wali   * Adolfo   GitHub: AdolMF   * Juan Carlos   GitHub: Jcrev95 |

**Descripción**

En esta actividad en grupo, trabajaremos sobre un repositorio de GitHub compartido entre los integrantes de este. Cada integrante trabajara sobre su propia rama de desarrollo creando diferentes clases de operaciones matemáticas en el IDE IntelliJ.

Cada clase deberá de documentarse con etiquetas JavaDoc, para una vez terminadas todas las clases, generaremos la documentación en formato HTML a partir del código de Java.

También explicaremos como se crea la documentación en diferentes casos especiales. Y para finalizar realizaremos distintas pruebas unitarias del proyecto, es decir cada compañero, probara la clase creada por otro y verificara que funciona correctamente.

**Metodología**

1. Creación y configuración del proyecto Calculadora

Creamos un proyecto en IntelliJ y un nuevo repositorio en GitHub para gestionar el proyecto. Creamos un paquete “javabean” para las clases y el paquete “testing” para las pruebas:

* *IntelliJ:* IDE utilizado para el desarrollo del código Java.
* *GitBash:* interfaz de línea de comandos para usar Git en Windows.

1. Creación del Repositorio Remoto en GitHub

Abrimos un repositorio remoto en GitHub que después cada uno de los integrantes del proyecto clonara en su ordenador creándose su repositorio local, mediante el comando:

* git clone “dirección-del-repositorio-remoto”

1. Implementación de las Clases

Implementación de diferentes clases que correspondan a cálculos matemáticos. Cada clase tiene métodos que se encargan de una operación específica, como suma, resta, multiplicación, módulo y cociente. Para realizar las clases utilizamos la herramienta **IntelliJ** para escribir y compilar el código en Java.

1. Pruebas Unitarias

Se han creado pruebas unitarias usando JUnit para validar la correcta ejecución de los métodos de las clases de los compañeros. Los cambios que se han ido haciendo se han registrado en el repositorio local, para después subirlas al remoto en GitHub.

1. Uso de Gitbash:

Los comandos utilizados son:

* git add -A para agregar todos los cambios.
* git commit -m "mensaje" para realizar el commit con los cambios.
* git push origin main para subir los cambios al repositorio remoto en GitHub.
* git push origin “tu-rama” para subir los cambios a tu rama personal de desarrollo.

1. Colaboración en Equipo

Cada compañero se clona el repositorio remoto en su repositorio local. Crea o modifica los campos que vea oportunos o necesarios y los sube a su rama de desarrollo en GitHub. Una vez hecho esto se realiza un merge con la rama principal para que el resto de los compañeros puedan descargárselo en su repositorio local. Comandos principales:

* git pull origin main para traer los últimos cambios realizados por los compañeros.
* git push origin main (si los cambios realizados son en la rama main) para subir cambios al repositorio.
* git push origin tu-rama para subir los cambios realizados desde tu rama.

**Problemas y soluciones**

1. Sincronización con la clase Main

El principal problema que hemos tenido ha sido a la hora de sincronizar la rama Main. A la hora de subir nuestras clases o cambios desde nuestras ramas, a veces se nos olvidaba realizar el merge con la clase main. Esto provocaba que el siguiente compañero al hacer el “pull”, no tuviera las ultimas actualizaciones y diese error en el “push”. Esto nos hacia perder el tiempo intentando arreglarlo desde la consola y buscando en el propio repositorio cual podría ser el fallo.

Para solucionar esto, hemos tenido que estar muy pendientes de cada vez que se realizaba un push, ir al repositorio y crear el merge correspondiente.

**Conclusiones**

Alguna de las conclusiones a las que llegamos realizando este proyecto son:

* Trabajar con un equipo en un proyecto requiere coordinación constante, especialmente cuando se usa Git. Es clave organizarse bien para evitar conflictos en el código.
* Hemos aprendido a manejar ramas, fusionar cambios y evitar conflictos mediante el uso de ramas separadas.
* Realizar una buena documentación con JavaDoc es fundamental para facilitar su comprensión y mantenimiento en el futuro.

**Desglose de tareas**

Mikel Barberena: Creación de clase Suma, MikelPruebaBea y el repositorio remoto. En el documento: Portada, Descripción y documentación de casos especiales.

Beatriz Wali: Creación de clase Resta y BeaPruebaAlmudena. En el documento: Metodología.

Almudena: Creación de clase Potencia y AlmuPrueba . En el documento: Conclusiones.

Adolfo: Creación clase Cociente y AdolPrueba . En el documento: Problemas y soluciones.

Juan Carlos: Creación de la clase Modulo y JuanPrueba . En el documento: Desglose de tareas.

**Documentación de Casos Especiales**

1. ¿Qué ocurre cuando alguno de los parámetros de entrada es cero?
   * **Suma y Resta**: En el caso de los métodos de suma y resta, si alguno de los parámetros de entrada es cero, la operación se realizaría sin problemas. La suma o resta no se ve afectada por el cero.
   * **Multiplicación**: Si alguno de los parámetros de entrada es cero, el resultado será siempre cero, ya que cualquier número multiplicado por cero es cero (n \* 0 = 0).
   * **División**: Si uno de los parámetros de entrada de la división es cero, el comportamiento varía dependiendo del contexto:
     + División de un número entre cero: Esto genera una excepción matemática (división por cero) y debe ser manejado adecuadamente en el código para evitar errores en tiempo de ejecución (ArithmeticException).
     + División de cero entre cualquier número: El resultado será cero (0 / a = 0), siempre que ‘a’ no sea cero.
2. Supongamos que la calculadora no puede utilizar números negativos ¿Cómo lo documentaríamos?

Si la calculadora no puede aceptar números negativos, se debe documentar en cada uno de los métodos. En caso de que se intente pasar un número negativo como parámetro, el método debería arrojar una excepción o devolver un mensaje de error indicando que los números negativos no son permitidos.

/\*\*

\* Suma dos números enteros.

\* @param a Primer número entero. Debe ser mayor o igual a cero.

\* @param b Segundo número entero. Debe ser mayor o igual a cero.

\* @return Resultado de la suma.

\* @throws IllegalArgumentException Si alguno de los números es negativo.

\*/

public int sumaEnteros(int a, int b) {

if (a < 0 || b < 0) {

throw new IllegalArgumentException("Los números no pueden ser negativos.");

}

return a + b;

}

1. ¿Qué ocurriría si la división no da un número exacto?

En el caso de la división que no resulta en un número exacto, el comportamiento dependerá de cómo se haya implementado la operación:

* + Para divisiones de números reales (tipo double), el resultado será el valor más preciso posible dentro de los límites del tipo de dato double.
  + Para divisiones de números enteros, el resultado será un número entero truncado, ya que Java no devuelve decimales en una división de enteros. Por ejemplo, 7 / 2 devolverá 3, truncando el valor decimal.

/\*\*

\* Realiza la división de dos números enteros.

\* @param a Dividendo.

\* @param b Divisor.

\* @return Resultado de la división truncado a número entero.

\* @throws ArithmeticException Si se intenta dividir por cero.

\*/

public int dividirEnteros(int a, int b) {

if (b == 0) {

throw new ArithmeticException("No se puede dividir entre cero.");

}

return a / b; // Truncará el valor decimal.

}

1. **¿**Qué ocurriría con la potencia si los números pasados son muy grandes?

En operaciones de potencia, cuando la base o el exponente son demasiado grandes, se podría producir un desbordamiento de memoria.

Si el resultado excede el rango del tipo double, se podría devolver Infinity (positivo o negativo), o si excede el rango de long o int, se podría producir un error de desbordamiento.

/\*\*

\* Calcula la potencia de un número con un exponente.

\* @param base La base de la potencia.

\* @param exponente El exponente al que se elevará la base.

\* @return Resultado de la potencia. Si el valor es demasiado grande, se puede devolver 'Infinity'.

\* @throws ArithmeticException Si ocurre un desbordamiento o error en el cálculo.

\*/

public double potencia(double base, int exponente) {

double resultado = Math.pow(base, exponente);

if (Double.isInfinite(resultado)) {

throw new ArithmeticException("El resultado de la potencia es demasiado grande.");

}

return resultado;

}