PGV-UT1

PROCESOS EN PARALELO

LingCheng Jiang
2ºDAM T | PGV

Contenido

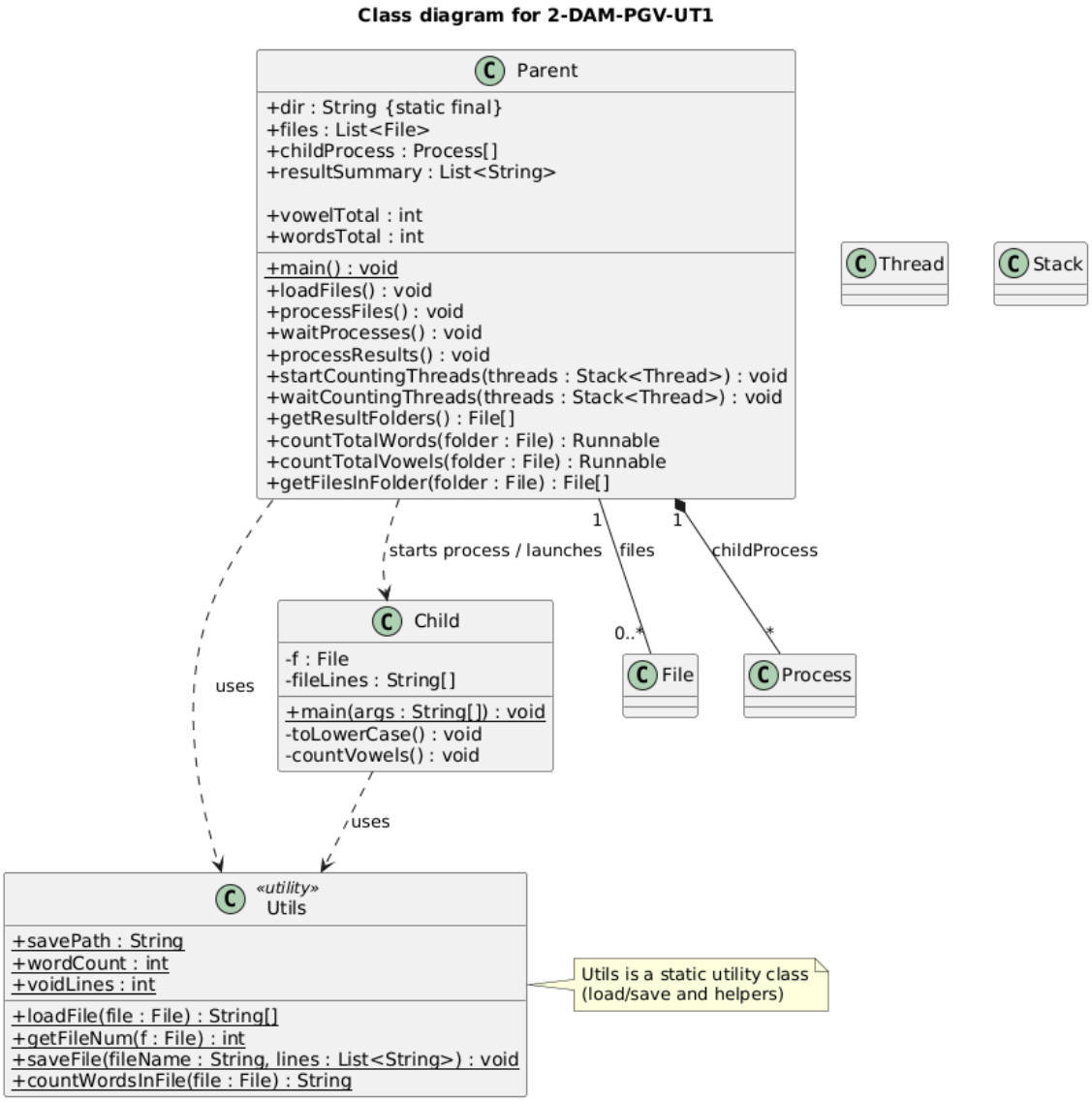
0. Introducción

Esta actividad tiene como objetivo leer datos desde varios archivos y con los datos obtenidos pasar las minúsculas y aparte contar las vocales de cada palabra, por cada archivo encontrado, crear un subproceso y que este lo gestione. Guardando los datos en archivos a parte para su posterior lectura desde el proceso/hilo principal.

Para el desarrollo de este proyecto, se usará la IDE IntelliJ Community, en consecuencia, Java.

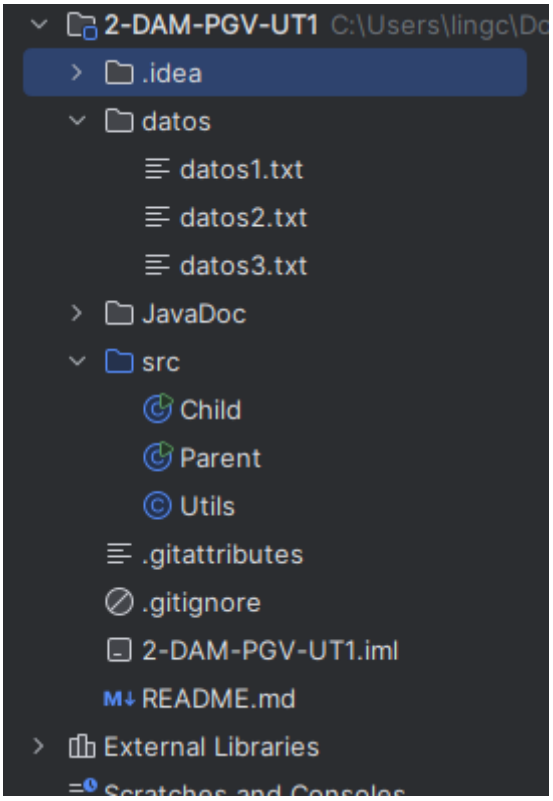
0.1. UML

Para una idea general de la actividad se crea este uml



1. Estructura del proyecto

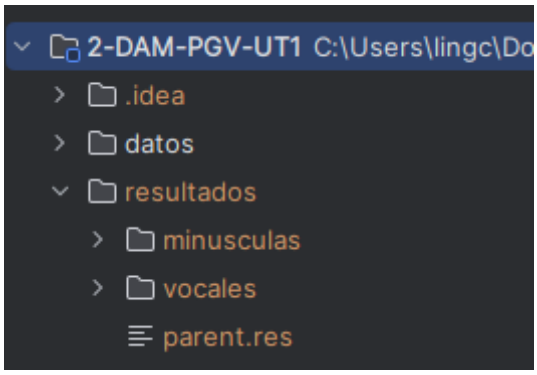
Este proyecto en esencia se estructura de la siguiente manera, con una carpeta en la raíz llamada datos, dentro de este estarán los distintos archivos llamados datosX.txt siendo X un número.



Aparte, estará la carpeta src, donde se encontrarán las clases creadas para esta actividad. Siendo Child, Parent y Utils.

Donde Parent será la clase/proceso principal, Child serán las instancias de los subprocesos, y Utils utilidades que se van creando para asistir a las otras 2 clases.

Tras la ejecución, si no ocurre ningún error, se debería generar una carpeta resultados, y dentro de esta, otras 2 carpetas, minúsculas y vocales, aparte de un archivo parent.res.



2. Clase Parent

Como se mencionó anteriormente, es la clase principal, donde se ejecuta todo el programa, al ser una actividad relativamente sencilla, no se ha creado una clase launcher, sino que se hace todo desde la función main dentro de parent. Por ello, hacemos que función será estática.

```
static void main() {  YarCrasy
    Parent p = new Parent();
    p.loadFiles();
    p.processFiles();
    p.waitProcesses();
    p.processResults();
}
```

2.1. Main()

Al ser una función estática, para que esta funcione, deberemos instanciar la clase parent dentro de él. Esto se hace más para simular el comportamiento de tener una clase launcher, pero se podría hacer sin static y también podría funcionar.

Dentro, de parent tenemos distintas funciones, que principalmente son las distintas "etapas" del programa, desde leer los archivos hasta procesar los archivos resultados de los subprocesos. Esto se hace para tener una lógica separada y ordenada.

2.2. loadFiles()

En loadFiles(), necesitaremos crear 2 variables de clases que se usarán en funciones futuras, siendo dir (la dirección de los archivos de lectura), y files (los archivos obtenidos del directorio).

```
final static String dir = "./datos/";  2 usages
public List<File> files = new ArrayList<>();  3 usages
```

Aquí simplemente intentamos leer los archivos, en caso de que no haya ninguno, imprime un error por pantalla.

```
void loadFiles()  1 usage  YarCrasy
{
    File folder = new File(dir);
    File[] listOfFiles = folder.listFiles();
    if (listOfFiles == null) {
        System.err.println("No files loaded from directory: " + dir);
        return;
    }
    Collections.addAll(files, listOfFiles);
}
```

2.3. processFiles()

Aquí es donde empezamos a procesar los archivos leidos, para ello, creamos otras variables, o mejor dicho, un array de procesos.

```
Process[] childProcess;  3 usages
```

Esto lo usaremos para crear instancias de la clase Child en estos, siendo cada proceso instancias independientes. Cada vez que se cree un nuevo subproceso, redirigimos el I/O al proceso principal, entones, intentamos iniciar el proceso, todo ello en un buble for para cada archivo.

Además, recopilamos la cantidad de palabras que hay en el archivo para que en el resumen final del padre pueda imprimirse.

```
final List <String> resultSummary = new ArrayList<>();  5 usages
```

Cabe a destacar, que para la creación de los procesos hubo que usar el System.*getProperty*("java.class.path"); porque sin pasar una classpath, devolvía error.

```
void processFiles()  1 usage  YarCrasy
{
    childProcess = new Process[files.size()];
    String cp = System.getProperty("java.class.path");
    int i = 0;
    for (File file : files) {
        resultSummary.add(Utils.countWordsInFile(file));
        ProcessBuilder pb = new ProcessBuilder( ...command: "java", "-cp", cp, "Child", file.getPath());
        pb.inheritIO();

        try {
            childProcess[i++] = pb.start();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

2.4. waitProcesses()

```
void waitProcesses()  1 usage  YarCrasy
{
    for (Process process : childProcess) {
        try {
            process.waitFor();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

una vez iniciado los procesos, en la función anterior, habrá que decir al programa que debe esperar a que acabe todos los subprocesos para poder seguir, ya que si no se generan los ficheros .res, cuando los intente leer darán unos resultados errores o directamente un error de ejecución. Para ello, usaremos la función waitFor() dentro de un bucle for, esto lo que hará es que, por cada proceso en el array, ejecute el waitFor() de este, esperando a que termine un proceso para empezar a esperar el siguiente. Haciendo que aparte de esperar el primer subproceso, los otros tengan también tiempo a acabar.

2.5. processResults()

Como parte final, en el padre se hará un resumen de los resultados de los hijos, en la actividad no se menciona si deberíamos usar paralelismo o no, pero para practicar se usaron hilos en este.

```java
void processResults()  1 usage
{
    Stack<Thread> threads = new Stack<>();
    startCountingThreads(threads);
    waitCountingThreads(threads);

    float aux = (wordsTotal == 0) ? 0 : (float)vowelTotal / wordsTotal;
    resultSummary.add("Promedio de vocales por palabra: " + aux);

    Utils.saveFile( fileName: "parent", resultSummary);
}
```

Para este resumen, se pide que cuente la cantidad de palabras procesadas por cada hijo, y hacer una suma de la cantidad de vocales encontradas. Y para terminar llamar hacer un promedio de cuantas vocales hay por palabras.

(Durante la realización del programa entendí de hacer un promedio de todos lo resultados y no de cada fichero resultado).

Para ello, separé aún más la lógica, creando subfunciones:

```java
void startCountingThreads(Stack<Thread> threads) {  1 usage   YarCrasy
    File[] resultFolders = getResultFolders();
    if (resultFolders.length == 0) {
        System.err.println("No result folders found.");
        return;
    }
    for (File folder : resultFolders) {
        String path = folder.getPath();
        if (path.equals(".\\resultados\\minusculas")) {
            threads.push(new Thread(countTotalWords(folder))).start();
        }
        else if (path.equals(".\\resultados\\vocales")) {
            threads.push(new Thread(countTotalVowels(folder))).start();
        }
    }
}
```

```java
void waitCountingThreads(Stack<Thread> threads) {  1 usage  YarCrasy
    while (!threads.isEmpty()) {
        try {
            threads.pop().join();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Estos tienen la misma lógica de los subprocesos Child, es decir, crearlos y esperar a que acaben para seguir, con la diferencia de que el código está en parent.

También como se podrá ver, se creó otras 3 funciones en startCountingThreads(), 2 de tipo Runnable, y un de tipo File[].

La de tipo File[] obtendrá todas las subcarpetas en el directorio resultado.

```java
File[] getResultFolders() {  1 usage  YarCrasy
    File resFolder = new File(Utils.savePath);
    File[] resultFiles = resFolder.listFiles();
    if (resultFiles == null) {
        System.err.println("No result folders found in directory getResultFolders(): " + Utils.savePath);
        return new File[0];
    }
    return resultFiles;
}
```

Para que en startCountingThreads() pueda leerlos y dependiendo del tipo de resultados, ejecutar un hilo u otro.

Mientras que los de tipo Runnable simplemente son para contar las palabras o vocales y escribirlos en el resumen, aparte de actualizar las nuevas variablesde clases creadas.

```java
int vowelTotal = 0;  3 usages
int wordsTotal = 0;  4 usages
```

```java
Runnable countTotalWords(File folder) {  1 usage  YarCrasy
    return () -> {
        File[] resultFiles = getFilesInFolder(folder);
        for (File f : resultFiles) {
            String[] lines = Utils.loadFile(f);
            assert lines != null;
            for (String line : lines) {
                wordsTotal += line.split( regex: "\\s+").length;
            }
        }
        resultSummary.add("Total words: " + wordsTotal);
    };
}
```

```java
Runnable countTotalVowels(File folder) {  1 usage  YarCrasy
    return () -> {
        File[] resultFiles = getFilesInFolder(folder);
        for (File f : resultFiles) {
            String[] lines = Utils.loadFile(f);
            assert lines != null;
            for (String line : lines) {
                vowelTotal += Integer.parseInt(line);
            }
        }
        resultSummary.add("Total vowels: " + vowelTotal);
    };
}
```

Tras esperar que acaben los hilos, se hace la media de las vocales por palabras, y termina la ejecución del programa, creando un parent.res como reporte final.

3. Clase Child

Esta clase como se mencionó anteriormente, será instanciada por subprocesos y se les pasará los archivos como args[] en el main. Si no se les pasa nada termina el proceso.

En este usaremos funciones de la clase Utils, para guardar los resultados de cada función y leer el archivo que se le pasó en un principio.

```java
public class Child {  YarCrasy *

    private File f;  4 usages
    String[] fileLines;  3 usages

    static void main(String[] args) {  YarCrasy
        Child c = new Child();
        if (args.length == 0) {
            System.err.println(c.getClass().getSimpleName() + ": No arguments provided");
            return;
        }
        c.f = new File(args[0]);
        c.fileLines = Utils.loadFile(c.f);
        c.toLowerCase();
        c.countVowels();
    }

    void toLowerCase() {  1 usage  YarCrasy
        List<String> aux = new ArrayList<>();
        for (String fileLine : fileLines) {
            aux.add(fileLine.toLowerCase());
        }
        Utils.saveFile( fileName: "minusculas/minusculas-"+Utils.getFileNum(f), aux);
    }

    void countVowels() {  1 usage  YarCrasy
        List<String> aux = new ArrayList<>();
        for (String fileLine : fileLines) {
            int count = 0;
            for (char c : fileLine.toCharArray()) {
                if ("aeiouAEIOUáéíóúÜü".indexOf(c) != -1) count++;
            }
            aux.add(count + "");
        }
        Utils.saveFile( fileName: "vocales/vocales-"+Utils.getFileNum(f), aux);
    }

}
```

4. Clase Utils

Esta clase principalmente tiene 4 funciones, pero anteriormente tenia un extra, un static void log(), que simplemente se usaba para hacer prints de prueba en el desarrollo, que se borró por ser mas tedioso de escribir que sout.

4.1. loadFile(File f)

carga todas las lines de un archivo que se le pasa y los devuelve en formato String[].

```
static String[] loadFile(File file) {  4 usages   YarCrasy
    try {
        return Files.readAllLines(file.toPath()).toArray(new String[0]);
    } catch (Exception e) {
        System.err.println("Error reading file: " + file.getName());
        return null;
    }
}
```

4.2. getFileNum(File f)

Devuelve el número del nombre del archivo que se le pasa

```
static int getFileNum(File f) {  2 usages   YarCrasy
    String name = f.getName();
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < name.length(); i++) {
        char c = name.charAt(i);
        if (Character.isDigit(c)) {
            sb.append(c);
        } else if (!sb.isEmpty()) {
            break;
        }
    }
    if (sb.isEmpty()) return 0;
    try {
        return Integer.parseInt(sb.toString());
    } catch (NumberFormatException e) {
        return 0;
    }
}
```

4.3. saveFile(String, List<String>)

guarda un archivo con extensión .res, si el archivo o directorio que se le pasa no existe lo crea.

```java
static void saveFile(String fileName, List<String> lines) {  3 usages  YarCrasy
    File file = new File( pathname: savePath + fileName + ".res");

    try {
        if (!file.exists()) {
            Files.createDirectories(file.toPath().getParent());
            Files.createFile(file.toPath());
        }
        Files.write(file.toPath(), lines);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

4.4. countWordsInFile(File)

Cuenta la cantidad de palabras y líneas vacías hay en una archivo, y lo guarda en una variable estática de clase.

```java
final static String savePath = "./resultados/";  3 usages
static int wordCount, voidLines;  3 usages
```

```java
static String countWordsInFile(File file) {  1 usage  YarCrasy
    String[] fileLines = loadFile(file);
    wordCount = voidLines = 0;
    assert fileLines != null;
    for (String line : fileLines) {
        if (line.trim().isEmpty()) {
            voidLines++;
            continue;
        }
        String[] words = line.trim().split( regex: "\\s+");
        wordCount += words.length;
    }
    return("Processing file: " + file.getName() + "\n" + wordCount + " words\n" + voidLines + " void lines\n");
}
```