```
#!/bin/sh

#
# Copyright © 2015-2021 the original authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

##############################################################################
#
#   Gradle start up script for POSIX generated by Gradle.
#
#   Important for running:
#
#   (1) You need a POSIX-compliant shell to run this script. If your /bin/sh is
#       noncompliant, but you have some other compliant shell such as ksh or
#       bash, then to run this script, type that shell name before the whole
#       command line, like:
#
#           ksh Gradle
#
#       Busybox and similar reduced shells will NOT work, because this script
#       requires all of these POSIX shell features:
#         * functions;
#         * expansions «$var», «${var}», «${var:-default}», «${var+SET}»,
#           «${var#prefix}», «${var%suffix}», and «$( cmd )»;
#         * compound commands having a testable exit status, especially «case»;
#         * various built-in commands including «command», «set», and «ulimit».
#
#   Important for patching:
#
#   (2) This script targets any POSIX shell, so it avoids extensions provided
#       by Bash, Ksh, etc; in particular arrays are avoided.
#
#       The "traditional" practice of packing multiple parameters into a
#       space-separated string is a well documented source of bugs and security
#       problems, so this is (mostly) avoided, by progressively accumulating
#       options in "$@", and eventually passing that to Java.
#
#       Where the inherited environment variables (DEFAULT_JVM_OPTS, JAVA_OPTS,
#       and GRADLE_OPTS) rely on word-splitting, this is performed explicitly;
#       see the in-line comments for details.
#
#       There are tweaks for specific operating systems such as AIX, CygWin,
#       Darwin, MinGW, and NonStop.
#
#   (3) This script is generated from the Groovy template
#       https://github.com/gradle/gradle/blob/master/subprojects/plugins/src/main/resources/org/gradle/a
#       within the Gradle project.
#
#       You can find Gradle at https://github.com/gradle/gradle/.
#
```

```
##############################################################################

# Attempt to set APP_HOME

# Resolve links: $0 may be a link
app_path=$0

# Need this for daisy-chained symlinks.
while
    APP_HOME=${app_path%"${app_path##*/}"}  # leaves a trailing /; empty if no leading path
    [ -h "$app_path" ]
do
    ls=$( ls -ld "$app_path" )
    link=${ls#*' -> '}
    case $link in             #(
      /*)   app_path=$link ;; #(
      *)    app_path=$APP_HOME$link ;;
    esac
done

APP_HOME=$( cd "${APP_HOME:-./}" && pwd -P ) || exit

APP_NAME="Gradle"
APP_BASE_NAME=${0##*/}

# Add default JVM options here. You can also use JAVA_OPTS and GRADLE_OPTS to pass JVM options to this s
DEFAULT_JVM_OPTS='"-Xmx64m" "-Xms64m"'

# Use the maximum available, or set MAX_FD != -1 to use that value.
MAX_FD=maximum

warn () {
    echo "$*"
} >&2

die () {
    echo
    echo "$*"
    echo
    exit 1
} >&2

# OS specific support (must be 'true' or 'false').
cygwin=false
msys=false
darwin=false
nonstop=false
case "$( uname )" in                #(
  CYGWIN* )         cygwin=true  ;; #(
  Darwin* )         darwin=true  ;; #(
  MSYS* | MINGW* )  msys=true    ;; #(
  NONSTOP* )        nonstop=true ;;
esac

CLASSPATH=$APP_HOME/gradle/wrapper/gradle-wrapper.jar


# Determine the Java command to use to start the JVM.
if [ -n "$JAVA_HOME" ] ; then
    if [ -x "$JAVA_HOME/jre/sh/java" ] ; then
        # IBM's JDK on AIX uses strange locations for the executables
        JAVACMD=$JAVA_HOME/jre/sh/java
```

```
    else
        JAVACMD=$JAVA_HOME/bin/java
    fi
    if [ ! -x "$JAVACMD" ] ; then
        die "ERROR: JAVA_HOME is set to an invalid directory: $JAVA_HOME

Please set the JAVA_HOME variable in your environment to match the
location of your Java installation."
    fi
else
    JAVACMD=java
    which java >/dev/null 2>&1 || die "ERROR: JAVA_HOME is not set and no 'java' command could be found

Please set the JAVA_HOME variable in your environment to match the
location of your Java installation."
fi

# Increase the maximum file descriptors if we can.
if ! "$cygwin" && ! "$darwin" && ! "$nonstop" ; then
    case $MAX_FD in #(
      max*)
        MAX_FD=$( ulimit -H -n ) ||
            warn "Could not query maximum file descriptor limit"
    esac
    case $MAX_FD in  #(
      '' | soft) :;; #(
      *)
        ulimit -n "$MAX_FD" ||
            warn "Could not set maximum file descriptor limit to $MAX_FD"
    esac
fi

# Collect all arguments for the java command, stacking in reverse order:
#   * args from the command line
#   * the main class name
#   * -classpath
#   * -D...appname settings
#   * --module-path (only if needed)
#   * DEFAULT_JVM_OPTS, JAVA_OPTS, and GRADLE_OPTS environment variables.

# For Cygwin or MSYS, switch paths to Windows format before running java
if "$cygwin" || "$msys" ; then
    APP_HOME=$( cygpath --path --mixed "$APP_HOME" )
    CLASSPATH=$( cygpath --path --mixed "$CLASSPATH" )

    JAVACMD=$( cygpath --unix "$JAVACMD" )

    # Now convert the arguments - kludge to limit ourselves to /bin/sh
    for arg do
        if
            case $arg in                                #(
              -*)   false ;;                            # don't mess with options #(
              /?*)  t=${arg#/} t=/${t%%/*}              # looks like a POSIX filepath
                    [ -e "$t" ] ;;                      #(
              *)    false ;;
            esac
        then
            arg=$( cygpath --path --ignore --mixed "$arg" )
        fi
        # Roll the args list around exactly as many times as the number of
        # args, so each arg winds up back in the position where it started, but
        # possibly modified.
```

```
        #
        # NB: a `for` loop captures its iteration list before it begins, so
        # changing the positional parameters here affects neither the number of
        # iterations, nor the values presented in `arg`.
        shift                   # remove old arg
        set -- "$@" "$arg"      # push replacement arg
    done
fi

# Collect all arguments for the java command;
#   * $DEFAULT_JVM_OPTS, $JAVA_OPTS, and $GRADLE_OPTS can contain fragments of
#     shell script including quotes and variable substitutions, so put them in
#     double quotes to make sure that they get re-expanded; and
#   * put everything else in single quotes, so that it's not re-expanded.

set -- \
        "-Dorg.gradle.appname=$APP_BASE_NAME" \
        -classpath "$CLASSPATH" \
        org.gradle.wrapper.GradleWrapperMain \
        "$@"

# Use "xargs" to parse quoted args.
#
# With -n1 it outputs one arg per line, with the quotes and backslashes removed.
#
# In Bash we could simply go:
#
#   readarray ARGS < <( xargs -n1 <<<"$var" ) &&
#   set -- "${ARGS[@]}" "$@"
#
# but POSIX shell has neither arrays nor command substitution, so instead we
# post-process each arg (as a line of input to sed) to backslash-escape any
# character that might be a shell metacharacter, then use eval to reverse
# that process (while maintaining the separation between arguments), and wrap
# the whole thing up as a single "set" statement.
#
# This will of course break if any of these variables contains a newline or
# an unmatched quote.
#

eval "set -- $(
        printf '%s\n' "$DEFAULT_JVM_OPTS $JAVA_OPTS $GRADLE_OPTS" |
        xargs -n1 |
        sed ' s~[^-[:alnum:]+,./:=@_]~\\&~g; ' |
        tr '\n' ' '
    )" '"$@"'

exec "$JAVACMD" "$@"
```

```
plugins {
    kotlin("jvm") version "1.6.10" // Use the appropriate Kotlin version
    id("com.github.johnrengelman.shadow") version "7.0.0" // Shadow plugin for creating a fat JAR
    id("org.jlleitschuh.gradle.ktlint") version "12.1.0"
    `maven-publish` // Required for publishing the library
}

group = "dev.onelenyk" // Replace with your group id
version = "0.1.1" // Your library version

repositories {
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib"))
    testImplementation(kotlin("test"))
    testImplementation("org.mockito.kotlin:mockito-kotlin:5.0.0")
    testImplementation("org.junit.jupiter:junit-jupiter:5.9.2")
}

// Configure publishing
publishing {
    publications {
        create("mavenJava") {
            from(components["java"])
        }
    }
}

// Include necessary information for JitPack
tasks.named("jar") {
    manifest {
        attributes["Implementation-Title"] = project.name
        attributes["Implementation-Version"] = project.version
    }
}

// Shadow plugin configuration to create a fat JAR (if needed)
tasks.shadowJar {
    archiveClassifier.set("")
    manifest {
        attributes["Main-Class"] = group + "MainKt" // Replace with your main class
    }
}

tasks.test {
    useJUnitPlatform()
}
```

```
rootProject.name = "gitignore-parser"
```

```
.gradle
build/
!gradle/wrapper/gradle-wrapper.jar
!**/src/main/**/build/
!**/src/test/**/build/

### IntelliJ IDEA ###
.idea/modules.xml
.idea/jarRepositories.xml
.idea/compiler.xml
.idea/libraries/
*.iws
*.iml
*.ipr
out/
!**/src/main/**/out/
!**/src/test/**/out/

### Eclipse ###
.apt_generated
.classpath
.factorypath
.project
.settings
.springBeans
.sts4-cache
bin/
!**/src/main/**/bin/
!**/src/test/**/bin/

### NetBeans ###
/nbproject/private/
/nbbuild/
/dist/
/nbdist/
/.nb-gradle/

### VS Code ###
.vscode/

### Mac OS ###
.DS_Store
/.idea/
```

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-8.2-bin.zip
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
```

```
kotlin.code.style=official
```

```
@rem
@rem Copyright 2015 the original author or authors.
@rem
@rem Licensed under the Apache License, Version 2.0 (the "License");
@rem you may not use this file except in compliance with the License.
@rem You may obtain a copy of the License at
@rem
@rem      https://www.apache.org/licenses/LICENSE-2.0
@rem
@rem Unless required by applicable law or agreed to in writing, software
@rem distributed under the License is distributed on an "AS IS" BASIS,
@rem WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
@rem See the License for the specific language governing permissions and
@rem limitations under the License.
@rem

@if "%DEBUG%" == "" @echo off
@rem ##########################################################################
@rem
@rem  Gradle startup script for Windows
@rem
@rem ##########################################################################

@rem Set local scope for the variables with windows NT shell
if "%OS%"=="Windows_NT" setlocal

set DIRNAME=%~dp0
if "%DIRNAME%" == "" set DIRNAME=.
set APP_BASE_NAME=%~n0
set APP_HOME=%DIRNAME%

@rem Resolve any "." and ".." in APP_HOME to make it shorter.
for %%i in ("%APP_HOME%") do set APP_HOME=%%~fi

@rem Add default JVM options here. You can also use JAVA_OPTS and GRADLE_OPTS to pass JVM options to thi
set DEFAULT_JVM_OPTS="-Xmx64m" "-Xms64m"

@rem Find java.exe
if defined JAVA_HOME goto findJavaFromJavaHome

set JAVA_EXE=java.exe
%JAVA_EXE% -version >NUL 2>&1
if "%ERRORLEVEL%" == "0" goto execute

echo.
echo ERROR: JAVA_HOME is not set and no 'java' command could be found in your PATH.
echo.
echo Please set the JAVA_HOME variable in your environment to match the
echo location of your Java installation.

goto fail

:findJavaFromJavaHome
set JAVA_HOME=%JAVA_HOME:"=%
set JAVA_EXE=%JAVA_HOME%/bin/java.exe

if exist "%JAVA_EXE%" goto execute

echo.
echo ERROR: JAVA_HOME is set to an invalid directory: %JAVA_HOME%
echo.
echo Please set the JAVA_HOME variable in your environment to match the
```

```
echo location of your Java installation.

goto fail

:execute
@rem Setup the command line

set CLASSPATH=%APP_HOME%\gradle\wrapper\gradle-wrapper.jar


@rem Execute Gradle
"%JAVA_EXE%" %DEFAULT_JVM_OPTS% %JAVA_OPTS% %GRADLE_OPTS% "-Dorg.gradle.appname=%APP_BASE_NAME%" -classp

:end
@rem End local scope for the variables with windows NT shell
if "%ERRORLEVEL%"=="0" goto mainEnd

:fail
rem Set variable GRADLE_EXIT_CONSOLE if you need the _script_ return code instead of
rem the _cmd.exe /c_ return code!
if  not "" == "%GRADLE_EXIT_CONSOLE%" exit 1
exit /b 1

:mainEnd
if "%OS%"=="Windows_NT" endlocal

:omega
```

GitIgnoreParser
[](https://jitpack.io/#onelenyk/gitignore-parser)

GitIgnoreParser is an advanced Kotlin library designed to parse `.gitignore` files and determine file ex
on gitignore specifications with efficiency and precision. This tool is particularly useful for develope
programmatically apply `.gitignore` rules in their Kotlin-based projects.

Features

- **Kotlin-Friendly**: Designed with Kotlin's modern language features in mind for a seamless integratio
- **Efficient Parsing**: Converts `.gitignore` glob patterns to regex for precise and fast matching.
- **Nested `.gitignore` Support**: Handles `.gitignore` files in subdirectories, ensuring comprehensive
  application.
- **Directory Specific Rules**: Accurately applies rules specific to directories, adhering to `.gitignor
- **Optimized for Large Projects**: Ideal for use in large-scale projects with extensive file structures
  performance.
- **Advanced Logging**: Integrates sophisticated logging mechanisms for enhanced debugging and operation

Installation

Using JitPack

Add the JitPack repository to your `build.gradle.kts`:

```gradle
allprojects {
    repositories {
        ...
        maven(url = "https://jitpack.io")
    }
}
```

Add the dependency:

```gradle
dependencies {
    implementation ("com.github.onelenyk:gitignore-parser:v0.1.0")
}
```

Usage

The GitIgnoreParser library is seamlessly integrable into your Kotlin projects. Here’s how to use it:

1. **Initialize the Parser**: Create an instance of `GitIgnoreParser` by passing the path to your `.giti

    ```kotlin
    val gitIgnoreFilePath = "/path/to/your/.gitignore"
    val parser = GitIgnoreParser(gitIgnoreFilePath)
    ```

2. **Check File Exclusion**: To determine if a specific file is excluded by the `.gitignore` rules, use
   the `isExcludedByGitignore` method. Provide the relative path of the file to this method.

    ```kotlin
    val relativeFilePath = "src/main/Example.kt"
    val isExcluded = parser.isExcludedByGitignore(relativeFilePath)
    println("Is the file excluded: $isExcluded")
    ```

3. **Get Exclusion Pattern**: To find out which specific `.gitignore` pattern is causing a file to be ex

   the `isExcludedByGitignoreWithPattern` method. This returns the matching pattern, if any.

    ```kotlin
    val exclusionPattern = parser.isExcludedByGitignoreWithPattern(relativeFilePath)
    exclusionPattern?.let {
        println("Excluded by pattern: $it")
    } ?: println("File is not excluded.")
    ```

4. **Process Files in a Directory**: To process all files in a directory, utilize the `FileProcessor` cl
   traversing the directory, checking each file against the `.gitignore` rules, and compiling a list of

    ```kotlin
    val projectRoot = Paths.get("/path/to/your/project")
    val fileProcessor = FileProcessor(projectRoot, parser)
    val includedFiles = fileProcessor.processFiles()
    ```

5. **Print Summary**: You can print a summary of the operation after processing the files, including the
   processed, files included, and patterns used.

    ```kotlin
    fileProcessor.printSummary()
    ```

This Kotlin adaptation of the library offers a modern and concise approach, enhancing the user experienc
Kotlin-based projects.

Sample Console Output

To give you a better idea of how the GitIgnoreParser works in practice, here is a sample of the console
expect. This output demonstrates the library's process of evaluating files against `.gitignore` rules an
insights into its operation.

Sample Log:

```plaintext
i Initializing GitIgnoreParser
 Processed pattern: .gradle as (^|/.*/)\.gradle
...
 Processed pattern: .DS_Store as (^|/.*/)\.DS_Store
i Loaded and parsed .gitignore successfully.
 Processed pattern: .*\.idea(/|$) as .*\.idea(/|$)
...
 Processed pattern: .*\.jar$ as .*\.jar$
i Custom rules added.
i Initialization complete. Total patterns loaded: 28
 Starting file processing
 Included:  (DIRECTORY)
 Excluded: build/kotlin/compileKotlin (DIRECTORY) by pattern .*build/(|.*/.*)
 Excluded: .git (DIRECTORY) by pattern .*\.git(/|$)
...
 Included: /Users/lenyk/IdeaProjects/gitignore-parser/gradlew.bat (FILE)
 Excluded: .idea (DIRECTORY) by pattern .*\.idea(/|$)
 Excluded: src/build/tmp/shadowJar (DIRECTORY) by pattern .*build/(|.*/.*)
 Included: src/build/reports (DIRECTORY)
...
 Excluded: src/build/reports/ktlint (DIRECTORY) by pattern .*build/(|.*/.*)
 Included: /Users/lenyk/IdeaProjects/gitignore-parser/src/main/kotlin/FileProcessor.kt (FILE)
 Included: /Users/lenyk/IdeaProjects/gitignore-parser/src/main/kotlin/Main.kt (FILE)
 Included: /Users/lenyk/IdeaProjects/gitignore-parser/src/main/kotlin/GitIgnoreParser.kt (FILE)
 File processing completed
```

 Summary:
i Total Items Processed: 52
i Total Files: 14
i Total Directories: 38
i Files/Directories Skipped: 3
i Files selected: 11
 Patterns Used: ▓▓.*\.jar$▓▓, ▓▓(^|/.*/)\.gradle▓▓, ▓▓.*build/(|.*/.*)▓▓, ▓▓.*\.git(/|$)▓▓, ▓▓.*\.idea(
```

This log shows the detailed process of how the GitIgnoreParser assesses each file, including any `.gitig
that apply, and the final decision on whether each file is ignored or included.

Note: The actual output may vary based on your project's `.gitignore` file and the specific files being

## Contributions

Contributions are welcome! Please feel free to submit a pull request or open an issue for any improvemen

## Acknowledgements

- **onelenyk** - Initial work and maintenance.
- **ChatGPT by OpenAI** - Assisted in development by providing code insights, optimization strategies, a
  support.

## License

This project is licensed under the [MIT License](LICENSE) - see the LICENSE file for details.
```

```
import org.junit.jupiter.api.Assertions.assertNotNull
import org.junit.jupiter.api.Test
import kotlin.test.assertEquals
import kotlin.test.assertFalse
import kotlin.test.assertNull
import kotlin.test.assertTrue

class GitignoreRulesTest {
    @Test
    fun testRegexConversion() {
        val gitignoreRules =
            GitignoreRules(
                listOf("*.log", "*.tmp"),
                customRules = listOf(".*\\.idea(/|\$)", ".*\\.jar\$"),
                key = "/",
            )
        // Test a standard pattern
        assertTrue(gitignoreRules.excludingPattern("test.log") != null, "test.log should be excluded")
        // Test a custom regex pattern
        assertTrue(
            gitignoreRules.excludingPattern("project.idea") != null,
            "project.idea should be excluded by custom rule",
        )
        // Test a pattern that should not be excluded
        assertFalse(gitignoreRules.excludingPattern("test.txt") != null, "test.txt should not be exclude
    }

    @Test
    fun testEmptyRules() {
        val emptyRules = GitignoreRules(emptyList(), key = "/")
        assertNull(emptyRules.excludingPattern("file.txt"), "No pattern should match as rules are empty'
    }

    @Test
    fun testRuleIgnoring() {
        val rulesWithComments = listOf("# This is a comment", "\n", "*.tmp")
        val gitignoreRules = GitignoreRules(rulesWithComments, key = "/")
        assertNotNull(gitignoreRules.excludingPattern("test.tmp"), "test.tmp should be excluded")
    }

    @Test
    fun testNegationPatternIgnoring() {
        val rulesWithNegation = listOf("!*.tmp", "*.tmp")
        val gitignoreRules = GitignoreRules(rulesWithNegation, key = "/")
        assertNotNull(gitignoreRules.excludingPattern("test.tmp"), "test.tmp should be excluded despite
    }

    @Test
    fun testCustomRulesApplication() {
        val customRules = GitignoreRules(emptyList(), listOf(".*\\.custom\$"), key = "/")
        assertNotNull(customRules.excludingPattern("file.custom"), "file.custom should be excluded by cu
    }

    @Test
    fun testCachingEfficiency() {
        val rules = listOf("*.cache")
        val gitignoreRules = GitignoreRules(rules, key = "/")
        val firstCall = gitignoreRules.excludingPattern("lenyk/test.cache")
        val secondCall = gitignoreRules.excludingPattern("lenyk/test.cache")
        assertEquals(firstCall, secondCall, "Repeated calls should retrieve the same cached regex")
    }
```

```
@Test
fun testComplexPatterns() {
    val complexRules = listOf("**/*.complex")
    val gitignoreRules = GitignoreRules(complexRules, key = "/")
    assertNotNull(gitignoreRules.excludingPattern("dir/subdir/file.complex"), "Complex pattern shoul
}

@Test
fun testMultipleWildcards() {
    val rules = listOf("*/*.log")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    assertNotNull(gitignoreRules.excludingPattern("dir/file.log"), "Multiple wildcards pattern shoul
}

@Test
fun testNestedWildcards() {
    val rules = listOf("**/logs/*.log")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    assertNotNull(gitignoreRules.excludingPattern("dir/logs/file.log"), "Nested wildcards pattern sh
}

@Test
fun testMixedWildcardsAndCharacters() {
    val rules = listOf("*.log.*")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    assertNotNull(gitignoreRules.excludingPattern("file.log.txt"), "Mixed wildcards and characters p
}

@Test
fun testCharacterClasses() {
    val rules = listOf("*.[tj]s")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    assertNotNull(gitignoreRules.excludingPattern("file.ts"), "Character classes pattern should excl
}

@Test
fun testEscapedCharacters() {
    val rules = listOf("\\*.log")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    assertNull(gitignoreRules.excludingPattern("file.log"), "Escaped characters pattern should not e
}

@Test
fun testDirectories() {
    val rules = listOf("logs/")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    assertNotNull(gitignoreRules.excludingPattern("logs/file.txt"), "Directories pattern should excl
}

@Test
fun testNegatedCharacterClasses() {
    val rules = listOf("*.[^tj]s")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    assertNotNull(gitignoreRules.excludingPattern("file.cs"), "Negated character classes pattern sho
}

@Test
fun testComplexBraces() {
    val rules = listOf("{*.txt,*.log}")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    assertNotNull(gitignoreRules.excludingPattern("file.txt"), "Complex braces pattern should exclud
}
```

```
@Test
fun testLeadingDirectory() {
    val rules = listOf("/logs/*.log")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    assertNotNull(gitignoreRules.excludingPattern("/logs/file.log"), "Leading directory pattern shou
}

@Test
fun testLeadingDirectory2() {
    val rules = listOf("/logs/*.log")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    assertNotNull(gitignoreRules.excludingPattern("logs/file.log"), "Leading directory pattern shoul
}

@Test
fun testDirectoryAndFileCombination() {
    val rules = listOf("logs/*.log")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    assertNotNull(gitignoreRules.excludingPattern("logs/file.log"), "Directory and file combination
}

@Test
fun testBuildExcluding() {
    val rules = listOf("build/")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    val firstCall = gitignoreRules.excludingPattern("build/reports/tests/test/packages/default-packa

    assertNotNull(firstCall, "Repeated calls should retrieve the same cached regex")
}

@Test
fun testHiddenFilesAndDirectories() {
    val rules = listOf(".env", ".DS_Store")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    assertNotNull(gitignoreRules.excludingPattern(".env"), ".env should be excluded")
    assertNotNull(gitignoreRules.excludingPattern(".DS_Store"), ".DS_Store should be excluded")
}

@Test
fun testSpecificFileExclusion() {
    val rules = listOf("secrets.yaml")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    assertNotNull(gitignoreRules.excludingPattern("configs/secrets.yaml"), "secrets.yaml should be e
}

@Test
fun testComplexPathPatterns() {
    val rules = listOf("assets/images/**/*.png")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    assertNotNull(gitignoreRules.excludingPattern("assets/images/icons/home.png"), "PNG files in ass
}

// /

@Test
fun testRootLevelSpecificFile() {
    val rules = listOf("/todo.md")
    val gitignoreRules = GitignoreRules(rules, key = "/")
    assertNotNull(gitignoreRules.excludingPattern("todo.md"), "Root level todo.md should be excluded
}
```

```
    // 5. Directory Exclusion with Exception
//    @Test
//    fun testDirectoryExclusionWithException() {
//        val rules = listOf("logs/", "!logs/important.log")
//        val gitignoreRules = GitignoreRules(rules, key = "/")
//        assertNull(gitignoreRules.excludingPattern("logs/important.log"), "important.log in logs shoul
//    }

    // 6. Wildcard in Middle of Pattern
    @Test
    fun testWildcardInMiddleOfPattern() {
        val rules = listOf("doc/*.md")
        val gitignoreRules = GitignoreRules(rules, key = "/")
        assertNotNull(gitignoreRules.excludingPattern("doc/readme.md"), "Markdown files in doc should be
    }

    // 7. Case Sensitivity Test
    @Test
    fun testCaseSensitivity() {
        val rules = listOf("README.md")
        val gitignoreRules = GitignoreRules(rules, key = "/")
        assertNull(gitignoreRules.excludingPattern("readme.md"), "readme.md should not be excluded due t
    }

    // 8. Pattern with Spaces
    @Test
    fun testPatternWithSpaces() {
        val rules = listOf("notes/Meeting Notes.txt")
        val gitignoreRules = GitignoreRules(rules, key = "/")
        assertNotNull(gitignoreRules.excludingPattern("notes/Meeting Notes.txt"), "Meeting Notes.txt in
    }

    // 9. Deeply Nested Pattern
    @Test
    fun testDeeplyNestedPattern() {
        val rules = listOf("src/main/java/com/example/utils/")
        val gitignoreRules = GitignoreRules(rules, key = "/")
        assertNotNull(
            gitignoreRules.excludingPattern("src/main/java/com/example/utils/Logger.java"),
            "Files in com/example/utils should be excluded",
        )
    }
}
```

```
import kotlin.test.Test
import kotlin.test.assertNotNull
import kotlin.test.assertNull

class GitignoreExtendedRulesTest {
    // Test for root directory files
    @Test
    fun `Root directory files should be included when not specified`() {
        val gitignoreRules = GitignoreRules(listOf("*.log"), key = "/")
        assertNull(gitignoreRules.excludingPattern("rootFile.txt"), "Root directory file should be inclu
    }

    // Test for ignoring files in a specific folder
    @Test
    fun `Files in a specific folder should be excluded`() {
        val gitignoreRules = GitignoreRules(listOf("specificFolder/"), key = "/")
        assertNotNull(
            gitignoreRules.excludingPattern("specificFolder/test.txt"),
            "Files in specificFolder should be excluded",
        )
    }

    // Test for specific file types in any directory
    @Test
    fun `Specific file types in any directory should be excluded`() {
        val gitignoreRules = GitignoreRules(listOf("*.log"), key = "/")
        assertNotNull(
            gitignoreRules.excludingPattern("subdir/example.log"),
            "Log files should be excluded in any directory",
        )
    }

    // Test for excluding files with certain prefix
    @Test
    fun `Files with certain prefix should be excluded`() {
        val gitignoreRules = GitignoreRules(listOf("temp_*"), key = "/")
        assertNotNull(gitignoreRules.excludingPattern("temp_file.txt"), "Files with 'temp_' prefix shoul
    }

    // Test for excluding files with certain suffix
    @Test
    fun `Files with certain suffix should be excluded`() {
        val gitignoreRules = GitignoreRules(listOf("*_backup"), key = "/")
        assertNotNull(gitignoreRules.excludingPattern("data_backup"), "Files with '_backup' suffix shoul
    }

    // Test for negated pattern
//        @Test
//        fun `Negated patterns should not exclude the file`() {
//            val gitignoreRules = GitignoreRules(listOf("!important.log"), key = "/")
//            assertNotNull(gitignoreRules.excludingPattern("important.log"), "Negated patterns should n
//        }

    // Test for nested directories exclusion
    @Test
    fun `Nested directories should be excluded`() {
        val gitignoreRules = GitignoreRules(listOf("outer/**/inner/"), key = "/")
        assertNotNull(
            gitignoreRules.excludingPattern("outer/middle/inner/file.txt"),
            "Nested directories should be excluded",
        )
    }
```

```
    // Test for file types in nested directories
    @Test
    fun `File types in nested directories should be excluded`() {
        val gitignoreRules = GitignoreRules(listOf("**/*.temp"), key = "/")
        assertNotNull(
            gitignoreRules.excludingPattern("outer/inner/file.temp"),
            "File types in nested directories should be excluded",
        )
    }

    // Test for excluding all except certain directories
//    @Test
//    fun `Exclude all except certain directories`() {
//        val gitignoreRules = GitignoreRules(listOf("/*", "!/include/"), key = "/")
//        assertNull(gitignoreRules.excludingPattern("exclude/file.txt"), "All directories except 'inclu
//        assertNotNull(gitignoreRules.excludingPattern("include/file.txt"), "Files in 'include' directo
//    }

    // Test for patterns with wildcards
    @Test
    fun `Patterns with wildcards should behave correctly`() {
        val gitignoreRules = GitignoreRules(listOf("*.log", "temp??.txt"), key = "/")
        assertNotNull(gitignoreRules.excludingPattern("error.log"), "Wildcard pattern '*.log' should exc
        assertNull(gitignoreRules.excludingPattern("temp12.txt"), "Wildcard pattern 'temp??.txt' should
    }
}
```

```
import org.junit.jupiter.api.AfterEach
import org.junit.jupiter.api.BeforeEach
import org.junit.jupiter.api.Test
import java.nio.file.Files
import java.nio.file.Path
import kotlin.test.assertEquals
import kotlin.test.assertNotNull

class GitIgnoreParserTest {
    private lateinit var gitignoreParser: GitIgnoreParser
    private lateinit var tempDir: Path
    private lateinit var tempGitignoreFile: Path
    private val expectedRules = listOf("*.log", "*.tmp")

    @BeforeEach
    fun setUp() {
        // Create a temporary directory
        tempDir = Files.createTempDirectory("gitignoreParserTest")

        gitignoreParser = GitIgnoreParser(rootDirectory = tempDir)

        // Create a .gitignore file in this directory
        tempGitignoreFile = tempDir.resolve(".gitignore")
        Files.write(tempGitignoreFile, expectedRules)
    }

    @Test
    fun testParseGitignore() {
        gitignoreParser.parseGitignore(tempGitignoreFile.parent)
        val rules = gitignoreParser.getRulesForDirectory(tempGitignoreFile.parent)
        assertNotNull(rules, "Rules should not be null")
        assertEquals(expectedRules, rules.rawRules, "Parsed rules should match the written content")
    }

    @AfterEach
    fun tearDown() {
        // Clean up: delete the temporary directory recursively
        tempDir.toFile().deleteRecursively()
    }
}
```

```
import util.Logs.log
import java.nio.file.Path

class GitIgnoreParser(
    val customRules: List = emptyList(),
    val rootDirectory: Path,
) {
    private val rulesMap = mutableMapOf()

    fun path(directory: Path): Path {
        val basePath = rootDirectory.parent
        val extractedPartSecond = basePath.relativize(directory)
        return extractedPartSecond
    }

    fun parseGitignore(directory: Path) {
        try {
            val gitignoreFile = directory.resolve(".gitignore").toFile()
            if (gitignoreFile.exists()) {
                val rules = gitignoreFile.readLines()
                val key = path(directory).toString()
                rulesMap[key] = GitignoreRules(rawRules = rules, customRules = customRules, key = key)
                log("Loaded and parsed .gitignore successfully.")
            } else {
                log("Error loading .gitignore file", isError = true)
            }
        } catch (e: Exception) {
            log("Error loading .gitignore file: ${e.message}", isError = true)
            throw e
        }
    }

    fun getRulesForDirectory(directory: Path): GitignoreRules? {
        var currentDir: Path? = directory
        while (currentDir != null) {
            val key = currentDir.toString()
            val rules = rulesMap[key]
            if (rules != null) {
                return rules
            }
            currentDir = currentDir.parent
        }
        return null // No rules found up to the root
    }
}
```

```
package util

import util.Logs.log
import java.nio.file.Path

class FileProcessorAnalytics {
    var totalFileCount: Int = 0
        private set
    var excludedFileCount: Int = 0
        private set
    var includedFileCount: Int = 0
        private set
    var excludedDirectoryCount: Int = 0
        private set
    var includedDirectoryCount: Int = 0
        private set
    var usedPatterns = mutableSetOf()
        private set

    fun onFileAppear() {
        totalFileCount++
    }

    fun onFileIncluded(directory: Path) {
        includedFileCount++
        log("Included: $directory (FILE)", prefix = "□")
    }

    fun onFileExcluded(
        directory: Path,
        excludingPattern: Regex,
    ) {
        excludedFileCount++
        onPatternUsed(excludingPattern)
        log("Excluded: $directory (FILE) by pattern $excludingPattern", prefix = "□")
    }

    fun onDirExcluded(
        directory: Path,
        excludingPattern: Regex,
    ) {
        excludedDirectoryCount++
        onPatternUsed(excludingPattern)
        log("Excluded: $directory (DIRECTORY) by pattern $excludingPattern", prefix = "□")
    }

    fun onDirIncluded(directory: Path) {
        includedDirectoryCount++
        log("Included: $directory (DIRECTORY)", prefix = "□")
    }

    private fun onPatternUsed(pattern: Regex) {
        usedPatterns.add(pattern)
    }

    fun report() {
        log("Summary:")
        log("Total Items Processed: $totalFileCount", isInfo = true)
        log("Directories included: $includedDirectoryCount", isInfo = true)
        log("Directories excluded: $excludedDirectoryCount", isInfo = true)
        log("Files included: $includedFileCount", isInfo = true)
        log("Files excluded: $excludedFileCount", isInfo = true)
```

```
        log("Patterns Used: ${usedPatterns.joinToString { "▓▓$it▓▓" }}", prefix = "\uD83D\uDCBC")
    }

    // Additional metrics and methods can be added as needed
}
```

```
package util

object Logs {
    fun log(
        message: String,
        isError: Boolean = false,
        isFine: Boolean = false,
    ) {
        val emoji =
            when {
                isError -> "□"
                isFine -> "□"
                else -> "i□"
            }
        println("$emoji $message")
    }

    fun log(
        message: String,
        isError: Boolean = false,
        isInfo: Boolean = false,
        prefix: String? = null,
    ) {
        val emoji =
            when {
                isError -> "□"
                isInfo -> "i□"
                prefix != null -> prefix
                else -> "□"
            }
        println("$emoji $message")
    }
}
```

```
import util.Logs.log

class GitignoreRules(
    val rawRules: List,
    val customRules: List = emptyList(),
    val key: String = "",
) {
    private val regexCache = mutableMapOf()
    private val regexRules: List =
        rawRules.mapNotNull { processLine(it) }
            .map {
                val regex = getOrCompileRegex(it)
                regex
            }
            .toMutableList()
            .apply { addAll(addCustomRules()) }

    private fun addCustomRules(): List {
        return customRules.map { rule ->
            val regexPattern = Regex(rule)
            log("Processed custom pattern: $rule as ${regexPattern.pattern}", isFine = true)
            return@map regexPattern
        }.apply {
            if (this.isNotEmpty()) {
                log("Custom rules added.")
            }
        }
    }

    private fun processLine(line: String): String? {
        val trimmedLine = line.trim()

        return when {
            trimmedLine.isBlank() || trimmedLine.isEmpty() -> {
                //   log("Processed pattern: $trimmedLine - ignored because of its empty", isFine = true
                null
            }

            trimmedLine.startsWith("#") -> {
                //   log("Processed pattern: $trimmedLine - ignored because of its comments", isFine = t
                null
            }

            trimmedLine.startsWith("!") -> {
                //   log("Processed pattern: $trimmedLine - ignored because of its negotiation pattern",
                null
            }

            else -> {
                trimmedLine
            }
        }
    }

    private fun convertGlobToRegex(pattern: String): Regex {
        // Immediately return regex for simple filename patterns
        when {
            isSimpleFilenamePattern(pattern) -> return Regex(handleSimpleFilenamePatterns(pattern))
            isRootedPattern(pattern) -> return Regex(handleRootedPatterns(pattern)) // drop the leading
            // other specific cases...
        }
```

```
    var adjustedPattern = pattern

    // Modularized handling steps for other patterns
    adjustedPattern = handleDoubleAsterisk(adjustedPattern)
    adjustedPattern = handleNegatedCharacterClasses(adjustedPattern)
    adjustedPattern = escapeSpecialRegexCharacters(adjustedPattern)
    adjustedPattern = unescapeBraces(adjustedPattern)
    adjustedPattern = handleBraces(adjustedPattern)
    adjustedPattern = replaceGlobPatterns(adjustedPattern)
    adjustedPattern = handleDirectorySpecificPatterns(adjustedPattern)

    // Construct the final regex pattern
    return Regex(adjustedPattern)
}

private fun handleSimpleFilenamePatterns(pattern: String): String = ".*$pattern$"

private fun isSimpleFilenamePattern(pattern: String): Boolean {
    // Check if the pattern is a filename without path or wildcards
    return !pattern.contains("/") && !pattern.contains("*") && !pattern.startsWith(".")
}

private fun handleDirectorySpecificPatterns(pattern: String): String {
    // Correctly handle directory-specific patterns
    return if (pattern.endsWith("/")) "^${pattern.dropLast(1)}/.*" else pattern
}

private fun handleDoubleAsterisk(pattern: String): String = if ("**" in pattern) pattern.replace("**

private fun handleNegatedCharacterClasses(pattern: String): String = if ("[^" in pattern) pattern.re

private fun escapeSpecialRegexCharacters(pattern: String): String = pattern.replace(Regex("[.^$+()|]

private fun unescapeBraces(pattern: String): String = pattern.replace(Regex("\\\\([{}])")) { it.grou

private fun replaceGlobPatterns(pattern: String): String =
    pattern.replace("*", "[^/]*")
        .replace("?", "[^/]")
        .replace("§§", ".*")
        .replace("UN1O", "[^")

private fun handleBraces(pattern: String): String {
    // This regex matches the content inside {...} and splits it by ','
    val braceRegex = Regex("\\{([^}]+)}")
    return if (braceRegex.containsMatchIn(pattern)) {
        pattern.replace(braceRegex) { matchResult ->
            val options = matchResult.groupValues[1].split(',')
            options.joinToString("|") { option ->
                option.replace("*", "[^/]*") // Replace * for each option
            }.let { "($it)" }
        }
    } else {
        pattern
    }
}

private fun handleRootedPatterns(pattern: String): String {
    // Remove the leading '/' and then replace glob patterns
    var trimmedPattern = pattern.drop(1)

    // Modularized handling steps for other patterns
    trimmedPattern = handleDoubleAsterisk(trimmedPattern)
```

```
        trimmedPattern = handleNegatedCharacterClasses(trimmedPattern)
        trimmedPattern = escapeSpecialRegexCharacters(trimmedPattern)
        trimmedPattern = replaceGlobPatterns(trimmedPattern)

        return "^/?$trimmedPattern$"
    }

    private fun isRootedPattern(pattern: String): Boolean {
        return pattern.startsWith("/")
    }

    private fun getOrCompileRegex(pattern: String): Regex {
        return regexCache.getOrPut(pattern) {
            val regex = convertGlobToRegex(pattern)
            log("Processed pattern: $pattern as ${regex.pattern}", isFine = true)
            regex
        }
    }

    fun excludingPattern(filePath: String): Regex? {
        return regexRules.firstOrNull { regex -> regex.matches(filePath) }
    }
}
```

```
import util.FileProcessorAnalytics
import util.Logs.log
import java.io.IOException
import java.nio.file.FileVisitResult
import java.nio.file.Files
import java.nio.file.Path
import java.nio.file.SimpleFileVisitor
import java.nio.file.attribute.BasicFileAttributes

class FileProcessor(
    private val rootDirectory: Path,
    val customRules: List = emptyList(),
) {
    private val analytics = FileProcessorAnalytics()
    private val gitignoreParser = GitIgnoreParser(customRules = customRules, rootDirectory)

    // Customizable file processing function
    private var fileProcessorFunction: (Path) -> Unit = { path ->
    }

    fun process() {
        val includedPaths = mutableListOf()
        log("Starting file processing")

        Files.walkFileTree(
            rootDirectory,
            object : SimpleFileVisitor() {
                override fun preVisitDirectory(
                    dir: Path,
                    attrs: BasicFileAttributes,
                ): FileVisitResult {
                    try {
                        if (Files.exists(dir.resolve(".gitignore"))) {
                            log("Gitignore file detected in: ${dir.toAbsolutePath()} ", isInfo = true)
                            // Parse .gitignore if present in this directory
                            gitignoreParser.parseGitignore(dir)
                        }

                        // Check if the directory should be excluded based on parent rules
                        if (shouldExcludeDirectory(dir)) {
                            return FileVisitResult.SKIP_SUBTREE
                        } else {
                            val nicePath = gitignoreParser.path(dir)
                            analytics.onDirIncluded(nicePath)
                        }

                        return FileVisitResult.CONTINUE
                    } catch (e: IOException) {
                        log("Failed to visit directory: $dir (ERROR)", isError = true)

                        return FileVisitResult.SKIP_SUBTREE
                    }
                }

                override fun visitFile(
                    file: Path,
                    attrs: BasicFileAttributes,
                ): FileVisitResult {
                    try {
                        analytics.onFileAppear()
                        val nicePath = gitignoreParser.path(file.parent)
                        val parentRules = gitignoreParser.getRulesForDirectory(nicePath)
```

```
                    val niceFile = gitignoreParser.path(file)

                    if (parentRules != null) {
                        val pattern = Path.of(parentRules.key).relativize(niceFile)
                        val key = pattern.ifEmpty(nicePath.fileName)

                        val excludingPattern = parentRules.excludingPattern(key.toString())

                        if (excludingPattern == null) {
                            fileProcessorFunction(file)
                            analytics.onFileIncluded(niceFile)
                            includedPaths.add(niceFile)
                        } else {
                            analytics.onFileExcluded(niceFile, excludingPattern)
                        }
                    } else {
                        fileProcessorFunction(file)
                        analytics.onFileIncluded(niceFile)
                        includedPaths.add(niceFile)
                    }

                    return FileVisitResult.CONTINUE
                } catch (e: IOException) {
                    log("Failed to visit: ${file.toAbsolutePath()} (ERROR)", isError = true)
                    return FileVisitResult.CONTINUE
                }
            }

            override fun visitFileFailed(
                file: Path,
                exc: IOException,
            ): FileVisitResult {
                log("Failed to visit: ${file.toAbsolutePath()} (ERROR)", isError = true)
                return FileVisitResult.CONTINUE
            }

            private fun shouldExcludeDirectory(dir: Path): Boolean {
                val nicePath = gitignoreParser.path(dir)
                val parentRules = gitignoreParser.getRulesForDirectory(nicePath)

                return if (parentRules != null) {
                    val pattern = Path.of(parentRules.key).relativize(nicePath)
                    val key = pattern.ifEmpty(nicePath.fileName)

                    val excludingPattern = parentRules.excludingPattern(key.toString())
                    excludingPattern?.let {
                        analytics.onDirExcluded(nicePath, excludingPattern)
                    }
                    excludingPattern != null
                } else {
                    false
                }
            }
        },
    )
}

private fun Path.ifEmpty(default: Path): Path {
    return if (this.isEmpty()) default else this
}

private fun Path.isEmpty(): Boolean {
```

```
        return this.toString().isEmpty()
    }

    fun report() = analytics.report()

    fun setFileProcessorFunction(function: (Path) -> Unit) {
        fileProcessorFunction = function
    }
}
```

```
import java.nio.file.Path

class Library {
    companion object {
        @JvmStatic
        fun main(args: Array) {
            val rootPath = Path.of("/Users/lenyk/projects/android-driver").toAbsolutePath()

            //   val rootPath = Path.of("").toAbsolutePath()
            val fileProcessor =
                FileProcessor(
                    rootPath,
                    customRules =
                        listOf(
                            ".*\\.idea(/|\$)",
                            ".*\\.git(/|\$)",
                            ".*\\.jar\$",
                        ),
                )
            fileProcessor.process()

            println(fileProcessor.report())
        }
    }
}
```