Walking the File Tree

Do you need to create an application that will recursively visit all the files in a file tree? Perhaps you need to delete every `.class` file in a tree, or find every file that has not been accessed in the last year. You can do so with the `FileVisitor` interface.

The FileVisitor Interface

To walk a file tree, you first need to implement a `FileVisitor`. A `FileVisitor` specifies the required behavior at key point the traversal process: when a file is visited, before a directory is accessed, after a directory is accessed, or when a failure occurs. The interface has four methods that correspond to these situations:

- `preVisitDirectory()` – Invoked before a directory's entries are visited.
- `postVisitDirectory()` – Invoked after all the entries in a directory are visited. If any errors are encountered, the spec exception is passed to the method.
- `visitFile()` – Invoked on the file being visited. The file's `BasicFileAttributes` is passed to the method, or you can the file attributes package to read a specific set of attributes. For example, you can choose to read the file's `DosFileAttributeView` to determine if the file has the "hidden" bit set.
- `visitFileFailed()` – Invoked when the file cannot be accessed. The specific exception is passed to the method. You choose whether to throw the exception, print it to the console or a log file, and so on.

If you do not need to implement all four of the `FileVisitor` methods, instead of implementing the `FileVisitor` interface can extend the `SimpleFileVisitor` class. This class is an adapter, which implements the `FileVisitor` interface, visits al files in a tree and throws an `IOError` when an error is encountered. You can extend this class and override only the metho that you require.

Here is an example that extends `SimpleFileVisitor` to print all entries in a file tree. It prints the entry whether the entry is regular file, a symbolic link, a directory, or some other "unspecified" type of file. It also prints the size, in bytes, of each file. exception that is encountered is printed to the console.

The `FileVisitor` methods are shown in the following code:

```
1  import static java.nio.file.FileVisitResult.*;
2
3  public static class PrintFiles
4      extends SimpleFileVisitor<Path> {
5
6      // Print information about
7      // each type of file.
8      @Override // from FileVisitor
9      public FileVisitResult visitFile(Path file,
10                                      BasicFileAttributes attr) {
11         if (attr.isSymbolicLink()) {
```

```
            System.out.format("Symbolic link: %s ", file);
        } else if (attr.isRegularFile()) {
            System.out.format("Regular file: %s ", file);
        } else {
            System.out.format("Other: %s ", file);
        }
        System.out.println("(" + attr.size() + "bytes)");
        return CONTINUE;
    }

    // Print each directory visited.
    @Override // from FileVisitor
    public FileVisitResult postVisitDirectory(Path dir,
                                              IOException exc) {
        System.out.format("Directory: %s%n", dir);
        return CONTINUE;
    }

    // If there is some error accessing
    // the file, let the user know.
    // If you don't override this method
    // and an error occurs, an IOException
    // is thrown.
    @Override // from FileVisitor
    public FileVisitResult visitFileFailed(Path file,
                                           IOException exc) {
        System.err.println(exc);
        return CONTINUE;
    }
}
```

Kickstarting the Process

Once you have implemented your `FileVisitor`, how do you initiate the file walk? There are two `walkFileTree()` methods the Files class.

- `walkFileTree(Path, FileVisitor)`
- `walkFileTree(Path, Set, int, FileVisitor)`

The first method requires only a starting point and an instance of your `FileVisitor`. You can invoke the `PrintFiles` file visitor as follows:

```
Path startingDir = ...;
PrintFiles pf = new PrintFiles();
Files.walkFileTree(startingDir, pf);
```

The second `walkFileTree()` method enables you to additionally specify a limit on the number of levels visited and a set `FileVisitOption` enums. If you want to ensure that this method walks the entire file tree, you can specify `Integer.MAX_VALUE` for the maximum depth argument.

You can specify the `FileVisitOption` enum, `FOLLOW_LINKS`, which indicates that symbolic links should be followed.

This code snippet shows how the four-argument method can be invoked:

```
import static java.nio.file.FileVisitResult.*;

```

```
Path startingDir = ...;

EnumSet<FileVisitOption> opts = EnumSet.of(FOLLOW_LINKS);

Finder finder = new Finder(pattern);
Files.walkFileTree(startingDir, opts, Integer.MAX_VALUE, finder);
```

Considerations When Creating a FileVisitor

A file tree is walked depth first, but you cannot make any assumptions about the iteration order that subdirectories are vis

If your program will be changing the file system, you need to carefully consider how you implement your `FileVisitor`.

For example, if you are writing a recursive delete, you first delete the files in a directory before deleting the directory itself. this case, you delete the directory in `postVisitDirectory()`.

If you are writing a recursive copy, you create the new directory in `preVisitDirectory()` before attempting to copy the fil it (in `visitFiles()`). If you want to preserve the attributes of the source directory (similar to the UNIX `cp -p` command), y need to do that after the files have been copied, in `postVisitDirectory()`. The `Copy` example shows how to do this.

If you are writing a file search, you perform the comparison in the `visitFile()` method. This method finds all the files tha match your criteria, but it does not find the directories. If you want to find both files and directories, you must also perforn comparison in either the `preVisitDirectory()` or `postVisitDirectory()` method. The `Find` example shows how to do t

You need to decide whether you want symbolic links to be followed. If you are deleting files, for example, following symbc links might not be advisable. If you are copying a file tree, you might want to allow it. By default, `walkFileTree()` does no follow symbolic links.

The `visitFile()` method is invoked for files. If you have specified the `FOLLOW_LINKS` option and your file tree has a circul link to a parent directory, the looping directory is reported in the `visitFileFailed()` method with the `FileSystemLoopException`. The following code snippet shows how to catch a circular link and is from the `Copy` example: `visitFile()` method is invoked for files. If you have specified the `FOLLOW_LINKS` option and your file tree has a circular lir a parent directory, the looping directory is reported in the `visitFileFailed()` method with the `FileSystemLoopExceptio` The following code snippet shows how to catch a circular link and is from the `Copy` example:

```
@Override
public FileVisitResult
    visitFileFailed(Path file,
        IOException exc) {
    if (exc instanceof FileSystemLoopException) {
        System.err.println("cycle detected: " + file);
    } else {
        System.err.format("Unable to copy:" + " %s: %s%n", file, exc);
    }
    return CONTINUE;
}
```

This case can occur only when the program is following symbolic links.

Controlling the Flow

Perhaps you want to walk the file tree looking for a particular directory and, when found, you want the process to terminat
Perhaps you want to skip specific directories.

The `FileVisitor` methods return a `FileVisitResult` value. You can abort the file walking process or control whether a directory is visited by the values you return in the `FileVisitor` methods:

- `CONTINUE` – Indicates that the file walking should continue. If the `preVisitDirectory()` method returns `CONTINUE`, the directory is visited.
- `TERMINATE` – Immediately aborts the file walking. No further file walking methods are invoked after this value is return
- `SKIP_SUBTREE` – When `preVisitDirectory()` returns this value, the specified directory and its subdirectories are skip This branch is "pruned out" of the tree.
- `SKIP_SIBLINGS` – When `preVisitDirectory()` returns this value, the specified directory is not visited, `postVisitDirectory()` is not invoked, and no further unvisited siblings are visited. If returned from the `postVisitDirectory()` method, no further siblings are visited. Essentially, nothing further happens in the specified directory.

In this code snippet, any directory named SCCS is skipped:

```
public FileVisitResult
     preVisitDirectory(Path dir,
         BasicFileAttributes attrs) {
    (if (dir.getFileName().toString().equals("SCCS")) {
         return SKIP_SUBTREE;
    }
    return CONTINUE;
}
```

In this code snippet, as soon as a particular file is located, the file name is printed to standard output, and the file walking terminates:

```
import static java.nio.file.FileVisitResult.*;

// The file we are looking for.
Path lookingFor = ...;

public FileVisitResult
    visitFile(Path file,
        BasicFileAttributes attr) {
    if (file.getFileName().equals(lookingFor)) {
        System.out.println("Located file: " + file);
        return TERMINATE;
    }
    return CONTINUE;
}
```

Finding Files

If you have ever used a shell script, you have most likely used pattern matching to locate files. In fact, you have probably u
it extensively. If you have not used it, pattern matching uses special characters to create a pattern and then file names ca
compared against that pattern. For example, in most shell scripts, the asterisk, `*`, matches any number of characters. For
example, the following command lists all the files in the current directory that end in `.html`:

```
$ ls *.html
```

The java.nio.file package provides programmatic support for this useful feature. Each file system implementation provides a PathMatcher. You can retrieve a file system's PathMatcher by using the getPathMatcher(String) method in t FileSystem class. The following code snippet fetches the path matcher for the default file system:

```
String pattern = ...;
PathMatcher matcher =
    FileSystems.getDefault().getPathMatcher("glob:" + pattern);
```

The string argument passed to getPathMatcher(String) specifies the syntax flavor and the pattern to be matched. This example specifies glob syntax. If you are unfamiliar with glob syntax, see the section What is a Glob.

Glob syntax is easy to use and flexible but, if you prefer, you can also use regular expressions, or regex, syntax. For furthe information about regex, see the section Regular Expressions. Some file system implementations might support other syntaxes.

If you want to use some other form of string-based pattern matching, you can create your own PathMatcher class. The examples in this page use glob syntax.

Once you have created your PathMatcher instance, you are ready to match files against it. The PathMatcher interface has single method, matches(), that takes a Path argument and returns a boolean: It either matches the pattern, or it does not following code snippet looks for files that end in .java or .class and prints those files to standard output:

```
PathMatcher matcher =
    FileSystems.getDefault().getPathMatcher("glob:*.{java,class}");

Path filename = ...;
if (matcher.matches(filename)) {
    System.out.println(filename);
}
```

Recursive Pattern Matching

Searching for files that match a particular pattern goes hand-in-hand with walking a file tree. How many times do you kno file is somewhere on the file system, but where? Or perhaps you need to find all files in a file tree that have a particular file extension.

The Find example does precisely that. Find is similar to the UNIX find utility, but has pared down functionally. You can extend this example to include other functionality. For example, the find utility supports the -prune flag to exclude an ent subtree from the search. You could implement that functionality by returning SKIP_SUBTREE in the preVisitDirectory() method. To implement the -L option, which follows symbolic links, you could use the four-argument walkFileTree(Path, Set, int, FileVisitor) method and pass in the FOLLOW_LINKS enum (but make sure that you test for circular links in th visitFile() method).

To run the Find application, use the following format:

```
$ java Find <path> -name "<glob_pattern>"
```

The pattern is placed inside quotation marks so any wildcards are not interpreted by the shell. For example:

```
$ java Find . -name "*.html"
```

The Find Example

Here is the source code for the Find example:

```
/**
 * Sample code that finds files that match the specified glob pattern.
 * For more information on what constitutes a glob pattern, see
 * https://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob
 *
 * The file or directories that match the pattern are printed to
 * standard out.  The number of matches is also printed.
 *
 * When executing this application, you must put the glob pattern
 * in quotes, so the shell will not expand any wild cards:
 *              java Find . -name "*.java"
 */

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
import static java.nio.file.FileVisitResult.*;
import static java.nio.file.FileVisitOption.*;
import java.util.*;


public class Find {

    public static class Finder
        extends SimpleFileVisitor<Path> {

        private final PathMatcher matcher;
        private int numMatches = 0;

        Finder(String pattern) {
            matcher = FileSystems.getDefault()
                    .getPathMatcher("glob:" + pattern);
        }

        // Compares the glob pattern against
        // the file or directory name.
        void find(Path file) {
            Path name = file.getFileName();
            if (name != null && matcher.matches(name)) {
                numMatches++;
                System.out.println(file);
            }
        }

        // Prints the total number of
        // matches to standard out.
        void done() {
            System.out.println("Matched: "
                + numMatches);
        }

        // Invoke the pattern matching
        // method on each file.
        @Override
        public FileVisitResult visitFile(Path file,
                BasicFileAttributes attrs) {
            find(file);

```

```
            return CONTINUE;
        }

        // Invoke the pattern matching
        // method on each directory.
        @Override
        public FileVisitResult preVisitDirectory(Path dir,
                BasicFileAttributes attrs) {
            find(dir);
            return CONTINUE;
        }

        @Override
        public FileVisitResult visitFileFailed(Path file,
                IOException exc) {
            System.err.println(exc);
            return CONTINUE;
        }
    }

    static void usage() {
        System.err.println("java Find <path>" +
            " -name \"<glob_pattern>\"");
        System.exit(-1);
    }

    public static void main(String[] args)
        throws IOException {

        if (args.length < 3 || !args[1].equals("-name"))
            usage();

        Path startingDir = Paths.get(args[0]);
        String pattern = args[2];

        Finder finder = new Finder(pattern);
        Files.walkFileTree(startingDir, finder);
        finder.done();
    }
}
```

The Copy Example

```
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;
import java.nio.file.attribute.FileTime;
import java.util.EnumSet;
import java.util.stream.Stream;

import static java.nio.file.FileVisitResult.CONTINUE;
import static java.nio.file.FileVisitResult.SKIP_SUBTREE;
import static java.nio.file.StandardCopyOption.COPY_ATTRIBUTES;
import static java.nio.file.StandardCopyOption.REPLACE_EXISTING;

/**
 * Sample code that copies files recursively
 * from a source directory to a destination folder.

```

```
 * The maximum number of directory levels to copy
 * is specified after -depth.
 * The number of files copied is printed
 * to standard out.
 * You can execute the application using:
 * @code java Copy . new -depth 4
 */

public class Copy {

    /**
     * A {@code FileVisitor} that finds
     * all files that match the
     * specified pattern.
     */
    public static class Replicator
            extends SimpleFileVisitor<Path> {

        Path source;
        Path destination;


        public Replicator(Path source, Path destination) {
            this.source = source;
            this.destination = destination;
        }

        // Prints the total number of
        // files copied to standard out.
        void done() throws IOException {
            try (Stream<Path> path = Files.list(Paths.get(destination.toUri()))) {
                System.out.println("Number of files copied: "
                        + path.filter(p -> p.toFile().isFile()).count());
            }

        }

        // Copy a file in destination
        @Override
        public FileVisitResult visitFile(Path file,
                                         BasicFileAttributes attrs) {
            System.out.println("Copy file: " + file);
            Path newFile = destination.resolve(source.relativize(file));
            try{
                Files.copy(file,newFile);
            }
            catch (IOException ioException){
                //log it and move
            }
            return CONTINUE;
        }

        // Invoke copy of a directory.
        @Override
        public FileVisitResult preVisitDirectory(Path dir,
                                                 BasicFileAttributes attrs) {
            System.out.println("Copy directory: " + dir);
            Path targetDir = destination.resolve(source.relativize(dir));
            try {
                Files.copy(dir, targetDir, REPLACE_EXISTING, COPY_ATTRIBUTES);
            } catch (IOException e) {
                System.err.println("Unable to create " + targetDir + " [" + e + "]");
                return SKIP_SUBTREE;
```

```
            }

            return CONTINUE;
        }

        @Override
        public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws IOException {
            if (exc == null) {
                Path destination = this.destination.resolve(source.relativize(dir));
                try {
                    FileTime time = Files.getLastModifiedTime(dir);
                    Files.setLastModifiedTime(destination, time);
                } catch (IOException e) {
                    System.err.println("Unable to copy all attributes to: " + destination + " [" + e + "]
                }
            } else {
                throw exc;
            }

            return CONTINUE;
        }

        @Override
        public FileVisitResult visitFileFailed(Path file,
                                               IOException exc) {
            if (exc instanceof FileSystemLoopException) {
                System.err.println("cycle detected: " + file);
            } else {
                System.err.format("Unable to copy:" + " %s: %s%n",
                        file, exc);
            }
            return CONTINUE;
        }
    }

    static void usage() {
        System.err.println("java Copy <source> <destination>" +
                " -depth \"<max_level_dir>\"");
        System.exit(-1);
    }

    public static void main(String[] args)
            throws IOException {

        if (args.length < 4 || !args[2].equals("-depth"))
            usage();

        Path source = Paths.get(args[0]);
        Path destination = Paths.get(args[1]);
        int depth = Integer.parseInt(args[3]);

        Replicator walk = new Replicator(source, destination);
        EnumSet<FileVisitOption> opts = EnumSet.of(FileVisitOption.FOLLOW_LINKS);
        Files.walkFileTree(source, opts, depth, walk);
        walk.done();
    }
}
```

The Chmod Example

```
import java.nio.file.*;
import java.nio.file.attribute.*;
import static java.nio.file.attribute.PosixFilePermission.*;
import static java.nio.file.FileVisitResult.*;
import java.io.IOException;
import java.util.*;

/**
 * Sample code that changes the permissions of files in a similar manner to the
 * chmod(1) program.
 */

public class Chmod {

    /**
     * Compiles a list of one or more <em>symbolic mode expressions</em> that
     * may be used to change a set of file permissions. This method is
     * intended for use where file permissions are required to be changed in
     * a manner similar to the UNIX <i>chmod</i> program.
     *
     * <p> The {@code exprs} parameter is a comma separated list of expressions
     * where each takes the form:
     * <blockquote>
     * <i>who operator</i> [<i>permissions</i>]
     * </blockquote>
     * where <i>who</i> is one or more of the characters {@code 'u'}, {@code 'g'},
     * {@code 'o'}, or {@code 'a'} meaning the owner (user), group, others, or
     * all (owner, group, and others) respectively.
     *
     * <p> <i>operator</i> is the character {@code '+'}, {@code '-'}, or {@code
     * '='} signifying how permissions are to be changed. {@code '+'} means the
     * permissions are added, {@code '-'} means the permissions are removed, and
     * {@code '='} means the permissions are assigned absolutely.
     *
     * <p> <i>permissions</i> is a sequence of zero or more of the following:
     * {@code 'r'} for read permission, {@code 'w'} for write permission, and
     * {@code 'x'} for execute permission. If <i>permissions</i> is omitted
     * when assigned absolutely, then the permissions are cleared for
     * the owner, group, or others as identified by <i>who</i>. When omitted
     * when adding or removing then the expression is ignored.
     *
     * <p> The following examples demonstrate possible values for the {@code
     * exprs} parameter:
     *
     * <table border="0">
     * <tr>
     *   <td> {@code u=rw} </td>
     *   <td> Sets the owner permissions to be read and write. </td>
     * </tr>
     * <tr>
     *   <td> {@code ug+w} </td>
     *   <td> Sets the owner write and group write permissions. </td>
     * </tr>
     * <tr>
     *   <td> {@code u+w,o-rwx} </td>
     *   <td> Sets the owner write, and removes the others read, others write
     *     and others execute permissions. </td>
     * </tr>
     * <tr>
     *   <td> {@code o=} </td>
     *   <td> Sets the others permission to none (others read, others write and
     *     others execute permissions are removed if set) </td>
```

```
     * </tr>
     * </table>
     *
     * @param   exprs
     *          List of one or more <em>symbolic mode expressions</em>
     *
     * @return  A {@code Changer} that may be used to changer a set of
     *          file permissions
     *
     * @throws  IllegalArgumentException
     *          If the value of the {@code exprs} parameter is invalid
     */
    public static Changer compile(String exprs) {
        // minimum is who and operator (u= for example)
        if (exprs.length() < 2)
            throw new IllegalArgumentException("Invalid mode");

        // permissions that the changer will add or remove
        final Set<PosixFilePermission> toAdd = new HashSet<PosixFilePermission>();
        final Set<PosixFilePermission> toRemove = new HashSet<PosixFilePermission>();

        // iterate over each of expression modes
        for (String expr: exprs.split(",")) {
            // minimum of who and operator
            if (expr.length() < 2)
                throw new IllegalArgumentException("Invalid mode");

            int pos = 0;

            // who
            boolean u = false;
            boolean g = false;
            boolean o = false;
            boolean done = false;
            for (;;) {
                switch (expr.charAt(pos)) {
                    case 'u' : u = true; break;
                    case 'g' : g = true; break;
                    case 'o' : o = true; break;
                    case 'a' : u = true; g = true; o = true; break;
                    default : done = true;
                }
                if (done)
                    break;
                pos++;
            }
            if (!u && !g && !o)
                throw new IllegalArgumentException("Invalid mode");

            // get operator and permissions
            char op = expr.charAt(pos++);
            String mask = (expr.length() == pos) ? "" : expr.substring(pos);

            // operator
            boolean add = (op == '+');
            boolean remove = (op == '-');
            boolean assign = (op == '=');
            if (!add && !remove && !assign)
                throw new IllegalArgumentException("Invalid mode");

            // who= means remove all
            if (assign && mask.length() == 0) {
                assign = false;
```

```
                remove = true;
                mask = "rwx";
            }

            // permissions
            boolean r = false;
            boolean w = false;
            boolean x = false;
            for (int i=0; i<mask.length(); i++) {
                switch (mask.charAt(i)) {
                    case 'r' : r = true; break;
                    case 'w' : w = true; break;
                    case 'x' : x = true; break;
                    default:
                        throw new IllegalArgumentException("Invalid mode");
                }
            }

            // update permissions set
            if (add) {
                if (u) {
                    if (r) toAdd.add(OWNER_READ);
                    if (w) toAdd.add(OWNER_WRITE);
                    if (x) toAdd.add(OWNER_EXECUTE);
                }
                if (g) {
                    if (r) toAdd.add(GROUP_READ);
                    if (w) toAdd.add(GROUP_WRITE);
                    if (x) toAdd.add(GROUP_EXECUTE);
                }
                if (o) {
                    if (r) toAdd.add(OTHERS_READ);
                    if (w) toAdd.add(OTHERS_WRITE);
                    if (x) toAdd.add(OTHERS_EXECUTE);
                }
            }
            if (remove) {
                if (u) {
                    if (r) toRemove.add(OWNER_READ);
                    if (w) toRemove.add(OWNER_WRITE);
                    if (x) toRemove.add(OWNER_EXECUTE);
                }
                if (g) {
                    if (r) toRemove.add(GROUP_READ);
                    if (w) toRemove.add(GROUP_WRITE);
                    if (x) toRemove.add(GROUP_EXECUTE);
                }
                if (o) {
                    if (r) toRemove.add(OTHERS_READ);
                    if (w) toRemove.add(OTHERS_WRITE);
                    if (x) toRemove.add(OTHERS_EXECUTE);
                }
            }
            if (assign) {
                if (u) {
                    if (r) toAdd.add(OWNER_READ);
                      else toRemove.add(OWNER_READ);
                    if (w) toAdd.add(OWNER_WRITE);
                      else toRemove.add(OWNER_WRITE);
                    if (x) toAdd.add(OWNER_EXECUTE);
                      else toRemove.add(OWNER_EXECUTE);
                }
                if (g) {
```

```
                    if (r) toAdd.add(GROUP_READ);
                      else toRemove.add(GROUP_READ);
                    if (w) toAdd.add(GROUP_WRITE);
                      else toRemove.add(GROUP_WRITE);
                    if (x) toAdd.add(GROUP_EXECUTE);
                      else toRemove.add(GROUP_EXECUTE);
                }
                if (o) {
                    if (r) toAdd.add(OTHERS_READ);
                      else toRemove.add(OTHERS_READ);
                    if (w) toAdd.add(OTHERS_WRITE);
                      else toRemove.add(OTHERS_WRITE);
                    if (x) toAdd.add(OTHERS_EXECUTE);
                      else toRemove.add(OTHERS_EXECUTE);
                }
            }
        }

        // return changer
        return new Changer() {
            @Override
            public Set<PosixFilePermission> change(Set<PosixFilePermission> perms) {
                perms.addAll(toAdd);
                perms.removeAll(toRemove);
                return perms;
            }
        };
    }

    /**
     * A task that <i>changes</i> a set of {@link PosixFilePermission} elements.
     */
    public interface Changer {
        /**
         * Applies the changes to the given set of permissions.
         *
         * @param   perms
         *          The set of permissions to change
         *
         * @return  The {@code perms} parameter
         */
        Set<PosixFilePermission> change(Set<PosixFilePermission> perms);
    }

    /**
     * Changes the permissions of the file using the given Changer.
     */
    static void chmod(Path file, Changer changer) {
        try {
            Set<PosixFilePermission> perms = Files.getPosixFilePermissions(file);
            Files.setPosixFilePermissions(file, changer.change(perms));
        } catch (IOException x) {
            System.err.println(x);
        }
    }

    /**
     * Changes the permission of each file and directory visited
     */
    static class TreeVisitor implements FileVisitor<Path> {
        private final Changer changer;

        TreeVisitor(Changer changer) {
```

```
            this.changer = changer;
        }

        @Override
        public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) {
            chmod(dir, changer);
            return CONTINUE;
        }

        @Override
        public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {
            chmod(file, changer);
            return CONTINUE;
        }

        @Override
        public FileVisitResult postVisitDirectory(Path dir, IOException exc) {
            if (exc != null)
                System.err.println("WARNING: " + exc);
            return CONTINUE;
        }

        @Override
        public FileVisitResult visitFileFailed(Path file, IOException exc) {
            System.err.println("WARNING: " + exc);
            return CONTINUE;
        }
    }

    static void usage() {
        System.err.println("java Chmod [-R] symbolic-mode-list file...");
        System.exit(-1);
    }

    public static void main(String[] args) throws IOException {
        if (args.length < 2)
            usage();
        int argi = 0;
        int maxDepth = 0;
        if (args[argi].equals("-R")) {
            if (args.length < 3)
                usage();
            argi++;
            maxDepth = Integer.MAX_VALUE;
        }

        // compile the symbolic mode expressions
        Changer changer = compile(args[argi++]);
        TreeVisitor visitor = new TreeVisitor(changer);

        Set<FileVisitOption> opts = Collections.emptySet();
        while (argi < args.length) {
            Path file = Paths.get(args[argi]);
            Files.walkFileTree(file, opts, maxDepth, visitor);
            argi++;
        }
    }
}
```

Last update: *January 4, 2024*