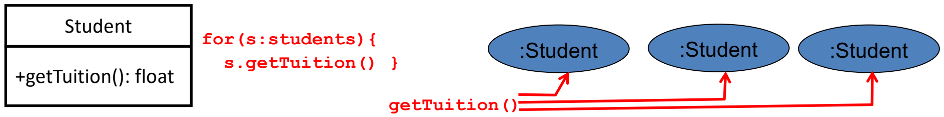
Visitor Design Pattern

Visitor Design Pattern

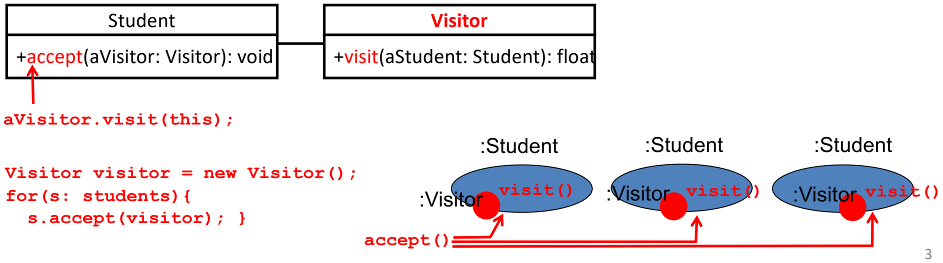
- Intent
  - Separate (or decouple) a set of objects and the operations to be performed on those objects.

- In a traditional (or normal) design, if an operation is performed on some objects, it is defined as a method of a class for those objects.



```
for(s:students){
  s.getTuition()  }
```

- With *Visitor*, the operation is defined as a method of Visitor.

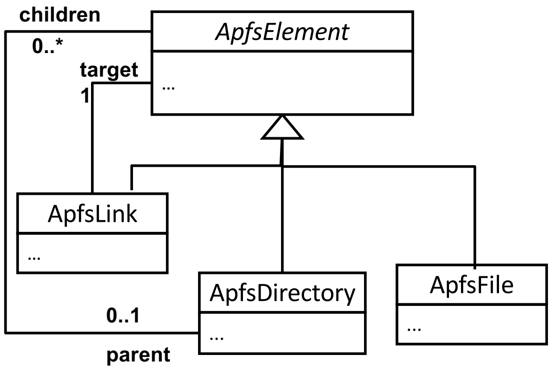


```
aVisitor.visit(this);
```

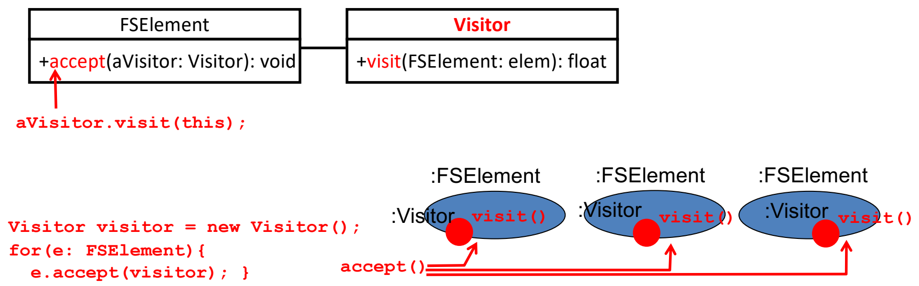
```
Visitor visitor = new Visitor();
for(s: students){
  s.accept(visitor); }
```

File System Examples (1)

- Count the number of directories, the number of files and the number links in a file system

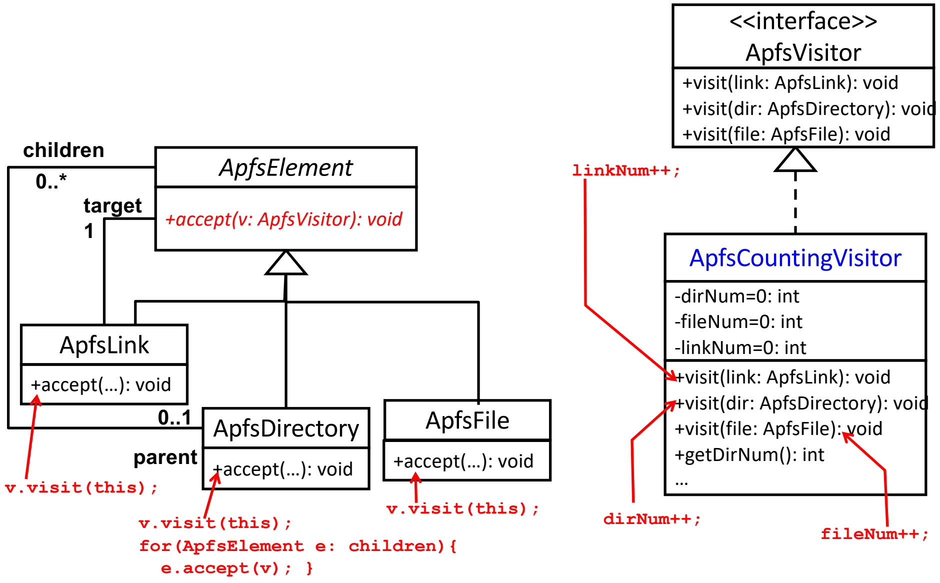


- With *Visitor*, an operation to count FS elements can be defined as a method of Visitor.



```
aVisitor.visit(this);
```

```
Visitor visitor = new Visitor();
for(e: FSElement){
  e.accept(visitor); }
```



```
v.visit(this);
```

```
v.visit(this);
for(ApfsElement e: children){
  e.accept(v); }
```

```
v.visit(this);
```

```
linkNum++;
```

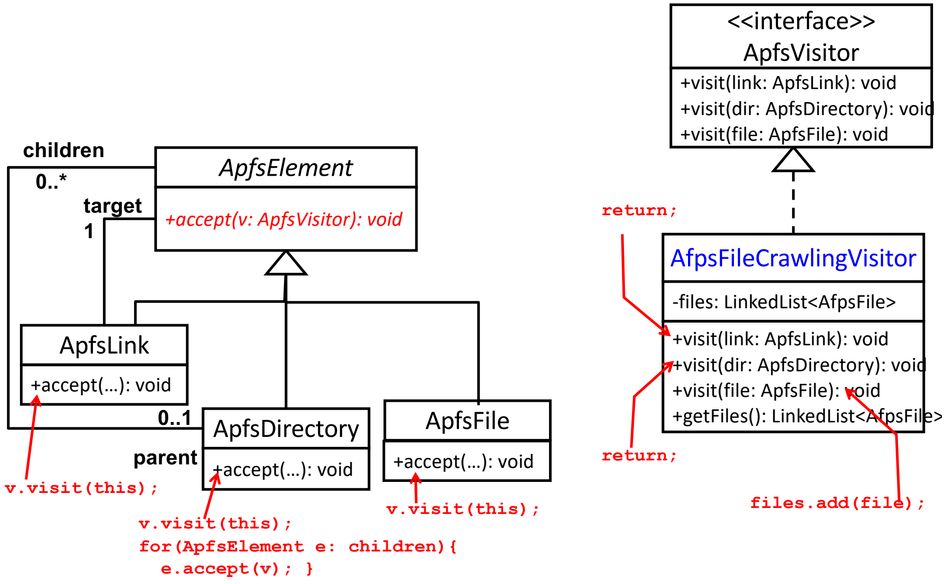
```
dirNum++;
```

```
fileNum++;
```

```
ApfsCountingVisitor visitor = new ApfsCountingVisitor();
rootDir.accept( visitor );
visitor.getDirNum(); visitor.getFileNum(); visitor.getLinkNum();
```

File System Examples (2)

- Index files in a file system
  - c.f. Operating system's indexing service
    - e.g., Windows indexing service and Mac/iOS's Spotlight
  - Key functionalities
    - Crawl a file system to identify files
    - Index those files for later file searches.
      - Extract and keep each file's metadata
        - » e.g., Path, name, size, creation time, owner's name, last-modified timestamp, checksum
- With *Visitor*, the file-crawling operation can be defined as a method of Visitor.


<<interface>>
ApfsVisitor
+visit(link: ApfsLink): void
+visit(dir: ApfsDirectory): void
+visit(file: ApfsFile): void
children
0..*
target
1
ApfsElement
+accept(v: ApfsVisitor): void
ApfsLink
+accept(...): void
0..1
parent
ApfsDirectory
+accept(...): void
ApfsFile
+accept(...): void
AfpsFileCrawlingVisitor
-files: LinkedList<AfpsFile>
+visit(link: ApfsLink): void
+visit(dir: ApfsDirectory): void
+visit(file: ApfsFile): void
+getFiles(): LinkedList<AfpsFile>

```
v.visit(this);
```

```
v.visit(this);
for(ApfsElement e: children){
  e.accept(v); }
```

```
v.visit(this);
```

```
return;
```

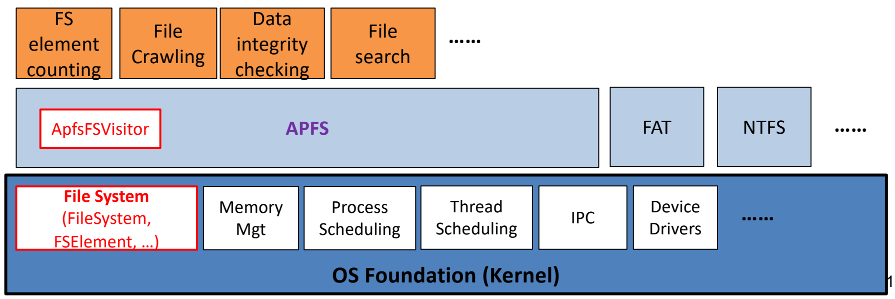
```
return;
```

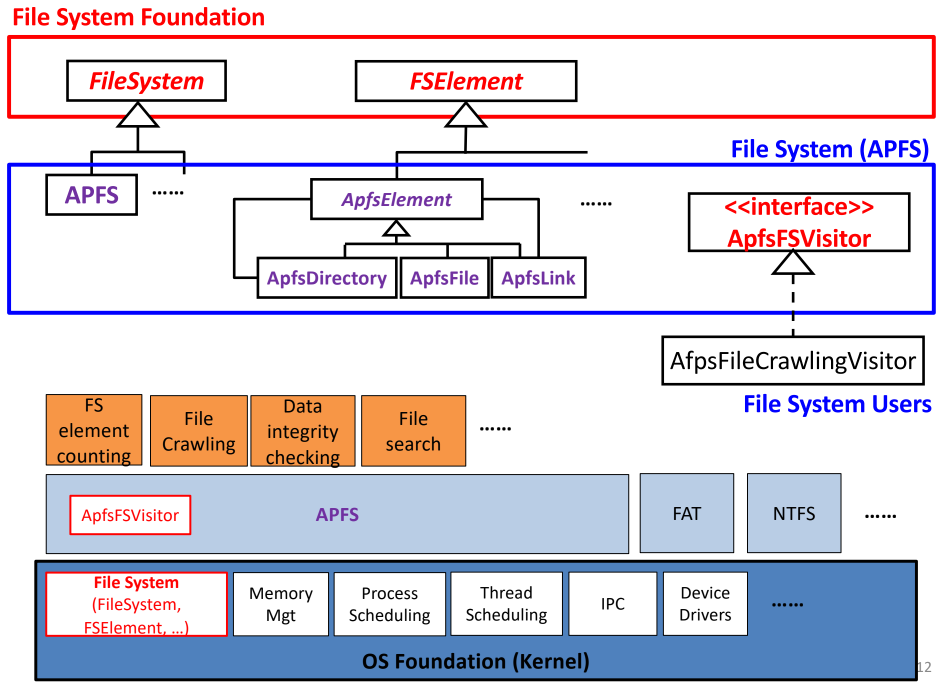
```
files.add(file);
```

```
FileCrawlingVisitor visitor = new FileCrawlingVisitor();
rootDir.accept( visitor );
visitor.getFiles();
```

What’s the Point?

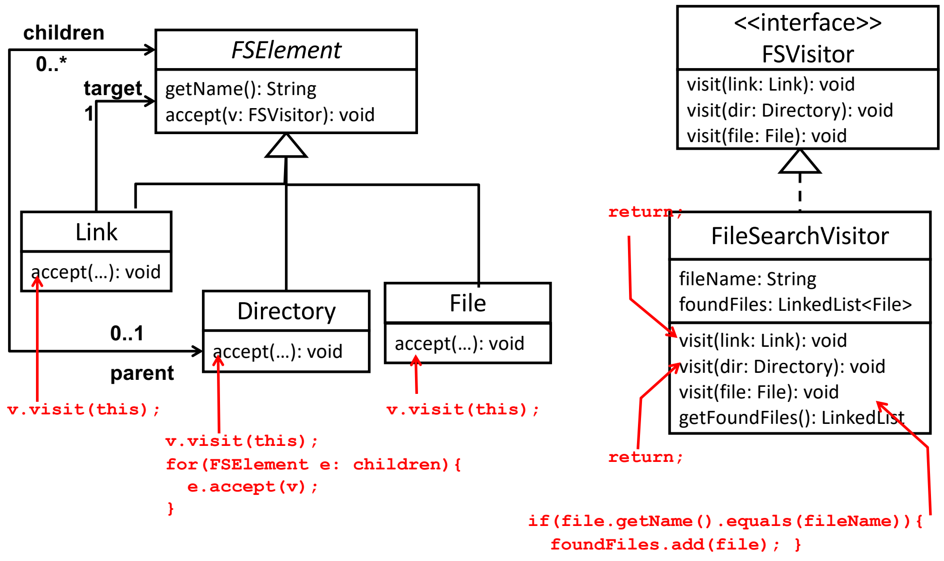
- *Visitor* can separate (decouple) FS data structures and the operations to be performed on those data structures.
  - Makes it easy to add, modify and remove those operations without changing FS data structures.
  - Allows those operations to be pluggable.





HW 9

- Define `ApfsVisitor` in the APFS package
- Implement it with 3 visitor class in the AFPS package
  - `AfpsCountingVisitor`
  - `AfspFileCrawlingVisitor`
  - `AfpsFileSearchVisitor`
    - Find a file with its name
- Use the 3 visitors on an example FS structure that you have used in previous HWs.
- Due: April 25 (Thu) midnight



```
v.visit(this);
```

```
v.visit(this);
for(FSElement e: children){
  e.accept(v);
}
```

```
v.visit(this);
```

```
return;
```

```
return;
```

```
if(file.getName().equals(fileName)){
  foundFiles.add(file); }
```

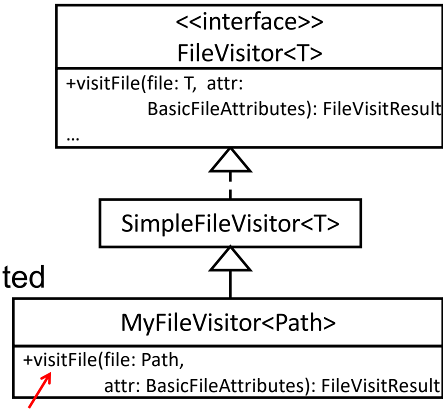
```
FileSearchVisitor visitor = new FileSearchVisitor(“a”);
rootDir.accept( visitor );
visitor.getFoundFiles().size();
```

Applicability of *Visitor*

- *Visitor* can be applied to any collection of objects, not limited to *Composite*-based tree structures.
  - – Set, list, graph, etc.

Visitor in Java API

- **`FileVisitor<T>`** and **`SimpleFileVisitor<T>`** in Java NIO (New I/O) package (**`java.nio`**)
  - – A visitor for files.
    - In **`java.nio.file`**
  - – **`visitFile(file, attr)`**
    - Invoked when a **`file`** is visited
    - **`attr`**: a set of attributes (metadata) of the **`file`**
    - **`Path`**: Represents a path. See Appendix.


<<interface>>
FileVisitor<T>
+visitFile(file: T, attr: BasicFileAttributes): FileVisitResult
…
SimpleFileVisitor<T>
MyFileVisitor<Path>
+visitFile(file: Path, attr: BasicFileAttributes): FileVisitResult

```
System.out.println(file);
System.out.println(attr.size());
System.out.println(attr.creationTime());
...
```

- **`java.nio.file.Files`**
  - – A utility class (i.e., a set of static methods) to process a file/directory.
  - – c.f. Appendix
- **`Files.walkFileTree()`**
  - – Visits each file in a file tree and calls **`visitFile()`** on a visitor.
  - – `static Path walkFileTree(Path start, FileVisitor<? super Path> visitor)`
    - ”? super T” means any super type (incl. super class) of T
    - “? extends T” means any sub type (incl. subclass) of T.

```
Path aDir = ...;
Files.walkFileTree( aDir, new MyFileVisitor<Path>() );
```

Appendix:
NIO-based File/Path Handling and Try-with-resources Statement

(1) Dealing with File/Directory Paths in NIO

- **`java.nio.Paths`**
  - A utility class (i.e., a set of static methods) to create a path in the file system.
    - Path: A sequence of directory names
      - Optionally with a file name in the end.
  - A path can be *absolute* or *relative*.
    - `Path absolute = Paths.get("/Users/jxs/temp/test.txt");`
    - `Path relative = Paths.get(”temp/test.txt");`
- **`java.nio.Path`**
  - Represents a path in the file system.
  - Given a path, *resolve* (or determine) another path.

    -
      ```
      Path absolute = Paths.get("/Users/jxs/”);
      Path another = absolute.resolve(“temp/test.txt”);
      ```

    -
      ```
      Path relative = Paths.get(“src”);
      Path another = relative.resolveSibling(“bin”);
      ```

Just in Case: Passing a Variable # of Parameters to a Method

- `Paths.get()` can receive a variable number of parameter values (1 to many values)
  - c.f. Java API documentation
  - `Paths.get(String first, String... more)`
    - `Paths.get(“temp/test.txt”);` // relative path
    - `Paths.get(“temp”, “test.txt”);` // relative path
    - `Paths.get(“/”, “Users”, “jxs”);` // absolute path
  - `String... More` → Can receive zero to many String values.
  - Introduced in Java 5 (JDK 1.5)

- Parameter values are handled with an array.

  -
    ```
    class Foo{
         public void varParamMethod(String... strings){
              for(int i = 0; i < strings.length; i++){
                   System.out.println(strings[i]); } } }
    ```

  -
    ```
    Foo foo = new Foo();
    foo.varParamMethod(“U”, “M”, “B”);
    ```
- `String... Strings` is a syntactic sugar for `String[] strings`.
  - Your Java compiler transforms the above code to:

    -
      ```
      class Foo{
           public void varParamMethod(String[] Strings){
                for(int i = 0; i < strings.length; i++){
                     System.out.println(strings[i]); } } }
      ```

    -
      ```
      Foo foo = new Foo();
      String[] strs = {“U”, “M”, “B”};
      foo.varParamMethod(strs);
      ```

Reading and Writing into a File w/ NIO

- **`java.nio.file.Files`**
  - A utility class (i.e., a set of static methods) to process a file/directory.
  - Reading a byte sequence and a char sequence from a file

    -
      ```
      Path path = Paths.get("/Users/jxs/temp/test.txt");
      byte[] bytes = Files.readAllBytes(path);
      String content = new String(bytes);
      ```

    -
      ```
      List<String> lines = Files.readAllLines(path);
      for(String line: lines){
        System.out.println(line); }
      ```
  - Writing into a file
    - `Files.write(path, bytes);`
    - `Files.write(path, content.getBytes());`
    - `Files.write(path, bytes, StandardOpenOption.CREATE);`
    - `Files.write(path, lines);`
    - `Files.write(path, lines, StandardOpenOption.WRITE);`
    - `StandardOpenOption: CREATE, WRITE, APPEND, DELETE_ON_CLOSE, etc.`

NIO (java.nio) v.s. Traditional I/O (java.io)

- NIO provides simpler or easier-to-use APIs.
  - Client code can be more concise and easier to understand.
- NIO:

```
– Path path = Paths.get("/Users/jxs/temp/test.txt");
  byte[] bytes = Files.readAllBytes(path);
  String content = new String(bytes);
```

- java.io:

```
– File file = ...;
  FileInputStream fis = new FileInputStream(file);
  int len = (int)file.length();
  byte[] bytes = new byte[len];
  fis.read(bytes);
  fis.close();
  String content = new String(bytes);
```

NIO (java.nio) v.s. Traditional I/O (java.io)

- NIO:

```
– Path path = Paths.get("/Users/jxs/temp/test.txt");
  List<String> lines = Files.readAllLines(path);
```

- java.io:

```
– int ch=-1, i=0;
  ArrayList<String> contents = new ArrayList<String>();
  StringBuffer strBuff = new StringBuffer();
  File file = ...;
  InputStreamReader reader = new InputStreamReader(
                                 new FileInputStream(file));
  while( (ch=reader.read()) != -1 ){
      if( (char)ch == '\n' ){        //**line break detection
          contents.add(i, strBuff.toString());
          strBuff.delete(0, strBuff.length());
          i++;
          continue;
      }
       strBuff.append((char)ch);
  }
  reader.close();
```

** The perfect (platform independent) detection of a line break should be more complex.
Unix: '\n', Mac: '\r', Windows: '\r\n'   c.f. BufferedReader.read()

NIO (java.nio) v.s. Traditional I/O (java.io)

- NIO:

```
– Path path = Paths.get("/Users/jxs/temp/test.txt");
  List<String> lines = Files.readAllLines(path);
```

- java.io (a bit simplified version):

```
– int ch=-1, i=0;
  ArrayList<String> contents = new ArrayList<String>();
  StringBuffer strBuff = new StringBuffer();
  File file = ...;
  FileReader reader = new FileReader(file); //***
  while( (ch=reader.read()) != -1 ){
      if( (char)ch == '\n' ){  //** Line break detection
          contents.add(i, strBuff.toString());
          strBuff.delete(0, strBuff.length());
          i++;
          continue;
      }
      strBuff.append((char)ch);
  }
  reader.close();
```

*** FileReader: A convenience class for reading character files.

Files in Java NIO

- readAllBytes(), readAllLines()
  - Read the whole data from a file without buffering.
- write()
  - Write a set of data to a file without buffering.
- When using a large file, it makes sense to use `BufferedReader` and `BufferedWriter` with `Files`.

```
– Path path = Paths.get("/Users/jxs/temp/test.txt");
  BufferedReader reader = Files.newBufferedReader(path);
  while( (line=reader.readLine()) != null ){
      // do something
  }
  reader.close();
```

```
– BufferedWriter writer = Files.newBufferedWriter(path);
  writer.write(...);
  writer.close();
```

Just in case: Buffering

- At the lowest level, read/write operations deal with data *byte by byte*, or *char by char*.
  - File access occurs *byte by byte*, or *char by char*.
- Inefficient if you read/write a lot of data.
- Buffering allows read/write operations to deal with data in a coarse-grained manner.
  - Chunk by chunk, not byte by byte or char by char
  - Chunk = a set of bytes or a set of chars
    - The size of a chunk: 512 bytes by default, but configurable

Getting Input/Output Streams from `Files`

- Input and output streams can be obtained from `Files`.

  -
    ```
    Path path = Paths.get("/Users/jxs/temp/test.txt");
    InputStream is = Files.newInputStream(path);
    ```
    - `is` contains an instance of ChannelInputStream, which is a subclass of InputStream.
    - Make sure to call `is.close()` in the end.
- Can decorate the input/output stream with filters.

  -
    ```
    ZipInputStream zis = new ZipInputStream(
                                  Files.newInputStream(path) );
    ```
    - Make sure to call `zis.close()` in the end.

Never Forget to Call close()

- Need to call `close()` on each input/output stream (or its filer) in the end.
  - Must-do: Follow the *Before/After* design pattern.
    - In Java, use a *try-catch-finally* or *try-finally* statement.

      -
        ```
        Open a file here.
        try{
            Do something with the file here.
            Throw an exception if an error occurs.
        }catch(...){
            Error-handling code here.
        }finally{
            Close the file here.
        }
        ```
  - Note: No need to call `close()` when using `readAllBytes()`, `readAllLines()` and `write()` of `Files`.

-
  ```
  Path path = Paths.get("/Users/jxs/temp/test.txt");
  BufferedReader reader = Files.newBufferedReader(path);
  try{
    while( (line=reader.readLine()) != null ){
        // do something
    }
  }catch(IOException ex){
    ... // Error handling
  }finally{
    reader.close();
  }
  ```

(2) Try-with-resources Statement

- Allows you to skip calling close() explicitly in the finally block.
  - *Try-catch-finally*

```
– Open a file here.
  try{
      Do something with the file here.
  }catch(...){
      Handle errors here.
  }finally{
      Close the file here.
  }
```

  - *Try-with-resources*

```
• try ( Open a file here ){
      Do something with the file here.
  }
```

- `close()` is automatically called on a resource used for reading or writing to a file, when exiting a try block.

```
• try( BufferedReader reader =
           Files.newBufferedReader( Paths.get("test.txt")) ){
      while( (line=reader.readLine()) != null ){
          // do something }
  }
```

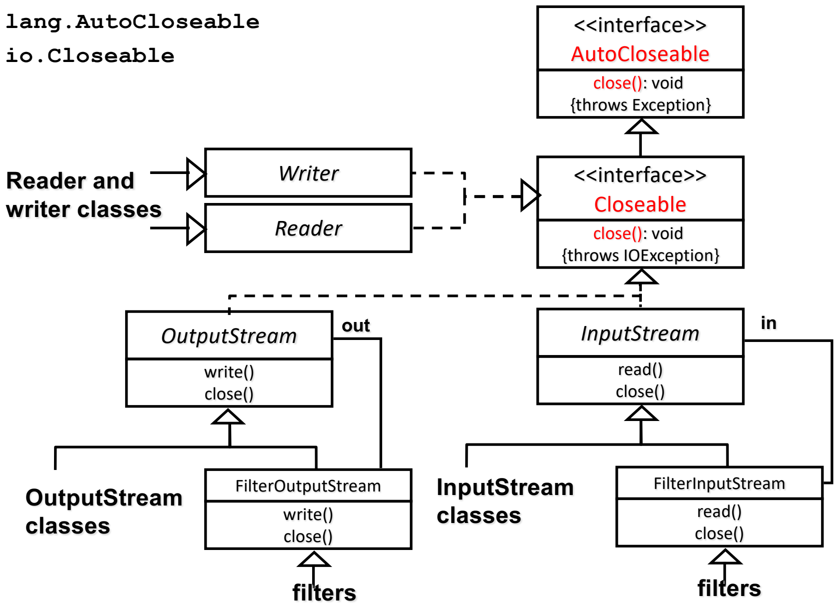
  - No explicit call of close() on `reader` in the finally block. `reader` is expected to implement the `AutoCloseable` interface.

```
• try( BufferedReader reader = Files.newBufferedReader(...);
       PrintWriter writer = new PrintWriter(...) ){
      while( (line=reader.readLine()) != null ){
          // do something
          writer.println(...); }
  }
```

  - Can specify multiple resources in a try block. close() is automatically called on all of them. They all need to implement `AutoCloseable`.

AutoCloseable Interface

- `java.lang.AutoCloseable`
- `java.io.Closeable`



<<interface>>
AutoCloseable
close(): void
{throws Exception}

<<interface>>
Closeable
close(): void
{throws IOException}

Reader and writer classes

Writer

Reader

OutputStream
write()
close()

out

InputStream
read()
close()

in

OutputStream classes

FilterOutputStream
write()
close()

filters

InputStream classes

FilterInputStream
read()
close()

filters

- Recap: No need to call `close()` when using `readAllBytes()`, `readAllLines()` and `write()` of `Files`.
  - Those methods internally use the try-with-resources statement to read and write to a file.

Try-with-resources-Catch-Finally

- Catch and finally blocks can be attached to a try-with-resources statement.

  -
    ```
    try( BufferedReader reader =
            Files.newBufferedReader( Paths.get(“test.txt")) ){
        while( (line=reader.readLine()) != null ){
            // do something. This part may throw an exception.}
    }catch(...){
        //This block runs if the try block throws an exception.
    }finally{
        ...
        //No need to do reader.close() here.
    }
    ```
    - The catch and finally blocks run (if necessary) AFTER close() is called on `reader`.