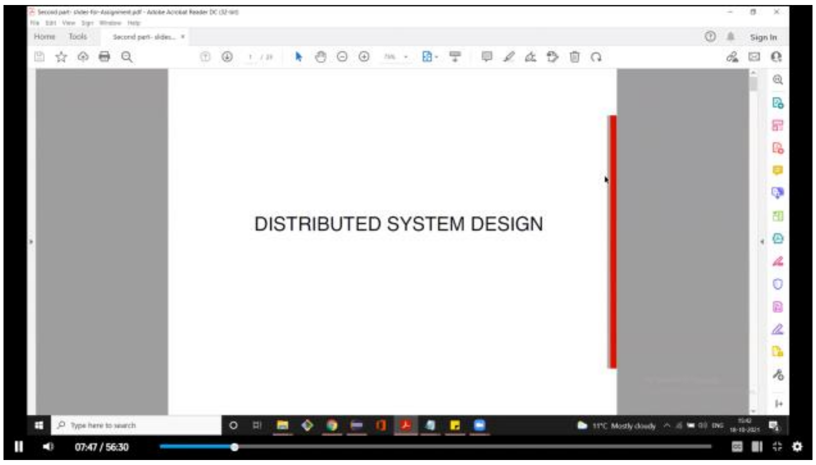
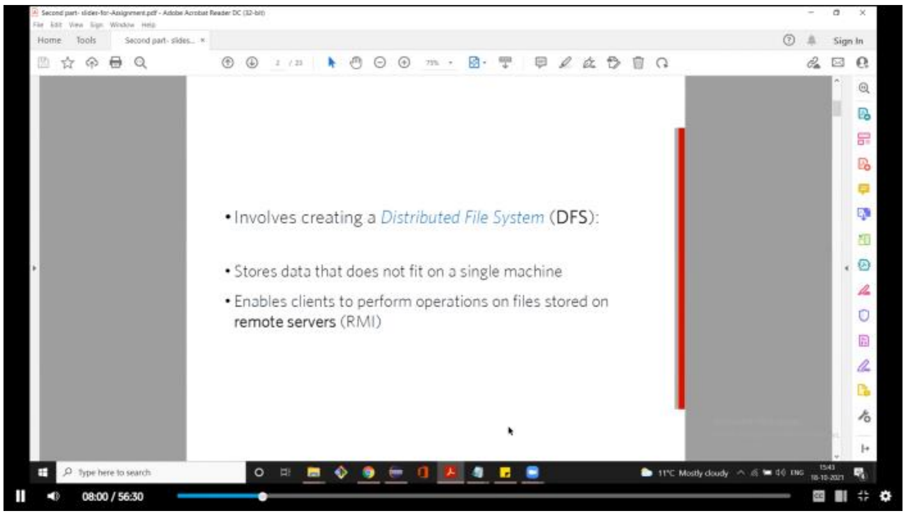
Lab 1

30 October 2021 09:41

- So we know what that we're working on a system that stores files in a distributed system, the data that does not fit on a single machine.
- So the clients want to do some actions (read or write) that will be done on the servers, this will be done using **RMI**

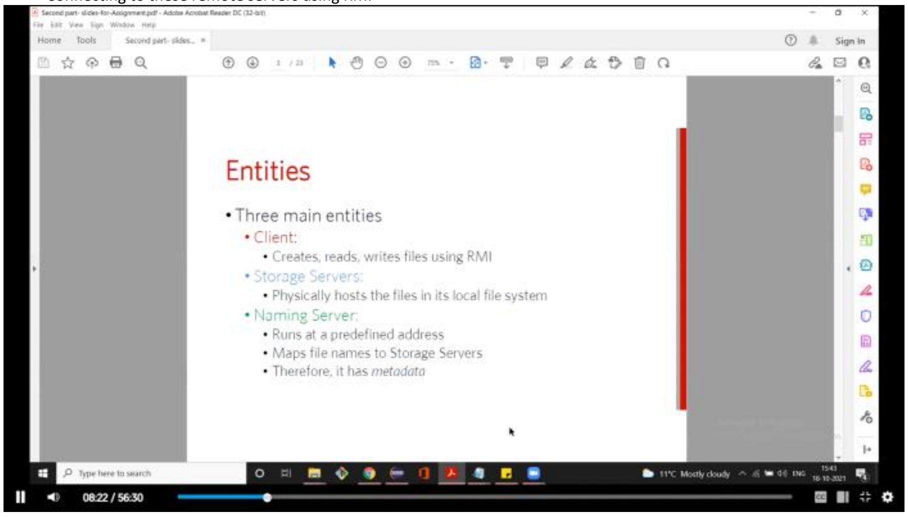


DISTRIBUTED SYSTEM DESIGN



- Involves creating a *Distributed File System* (DFS):
  - Stores data that does not fit on a single machine
  - Enables clients to perform operations on files stored on remote servers (RMI)

- Connecting to these remote servers using RMI

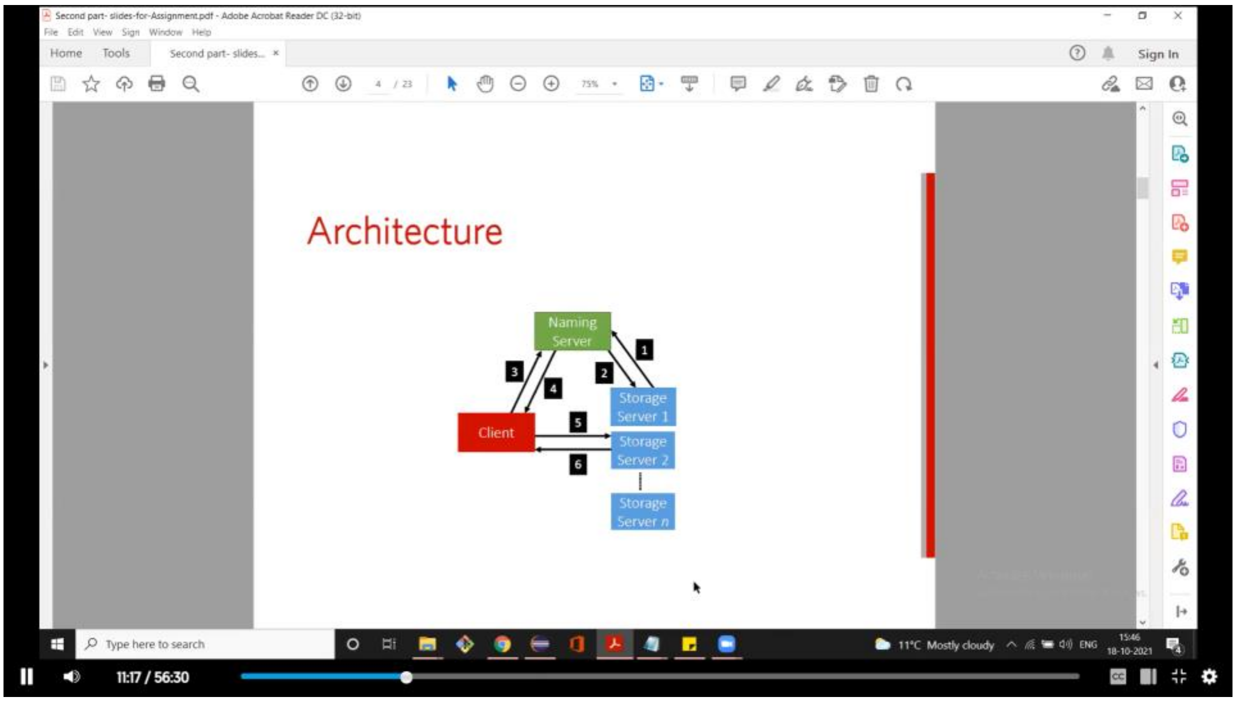


Entities

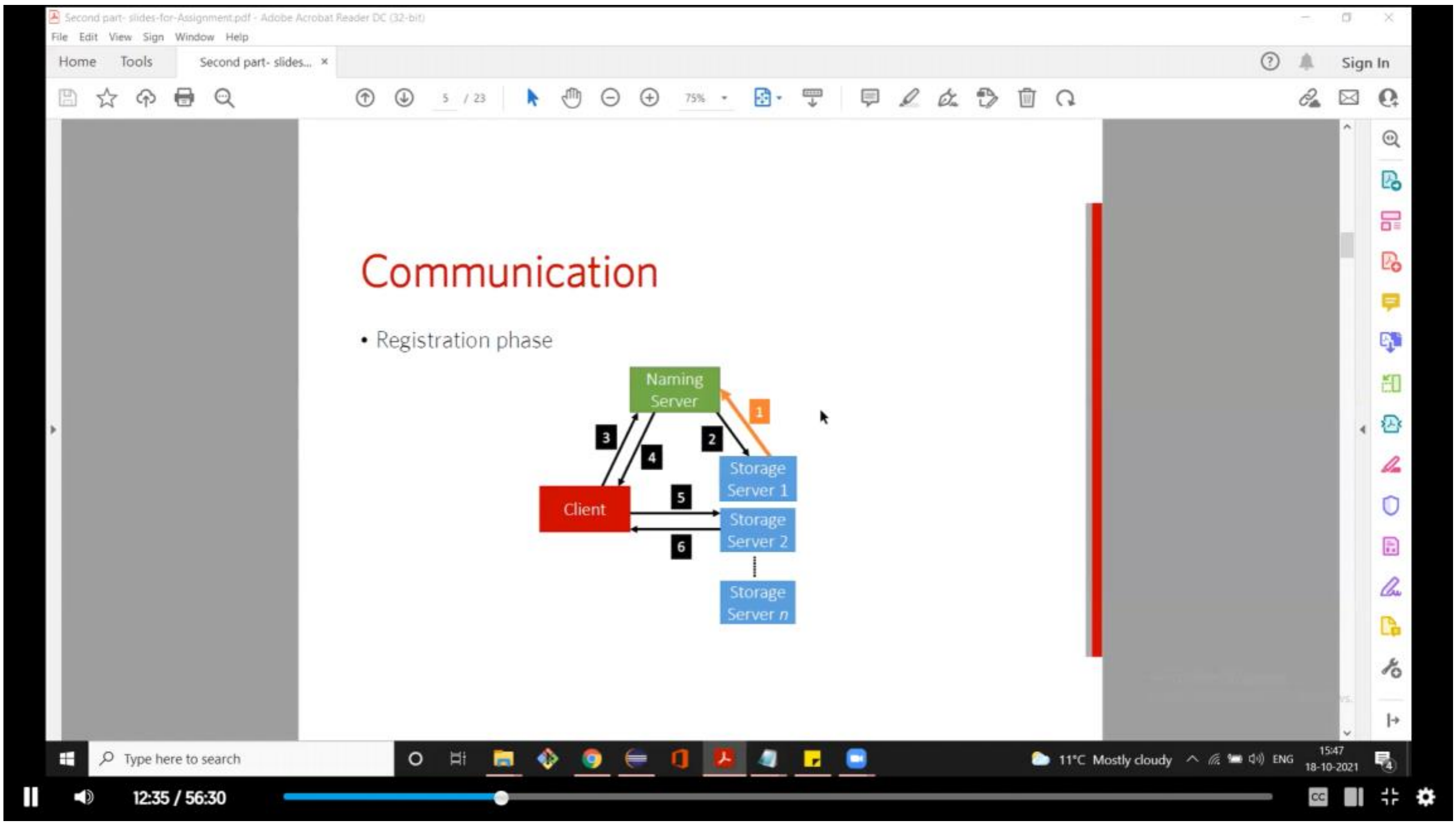
- Three main entities
  - Client:
    - Creates, reads, writes files using RMI
  - Storage Servers:
    - Physically hosts the files in its local file system
  - Naming Server:
    - Runs at a predefined address
    - Maps file names to Storage Servers
    - Therefore, it has *metadata*

Naming server:

- It helps the client look for a particular storage server
- It stores the metadata of the data present in the storage servers (metadata is temp data to explain the actual data)
- Client does not the location of the file, it only knows the naming of the files. N.S. map that names to the storage servers and help client finfding the req server where the file is stored



Architecture

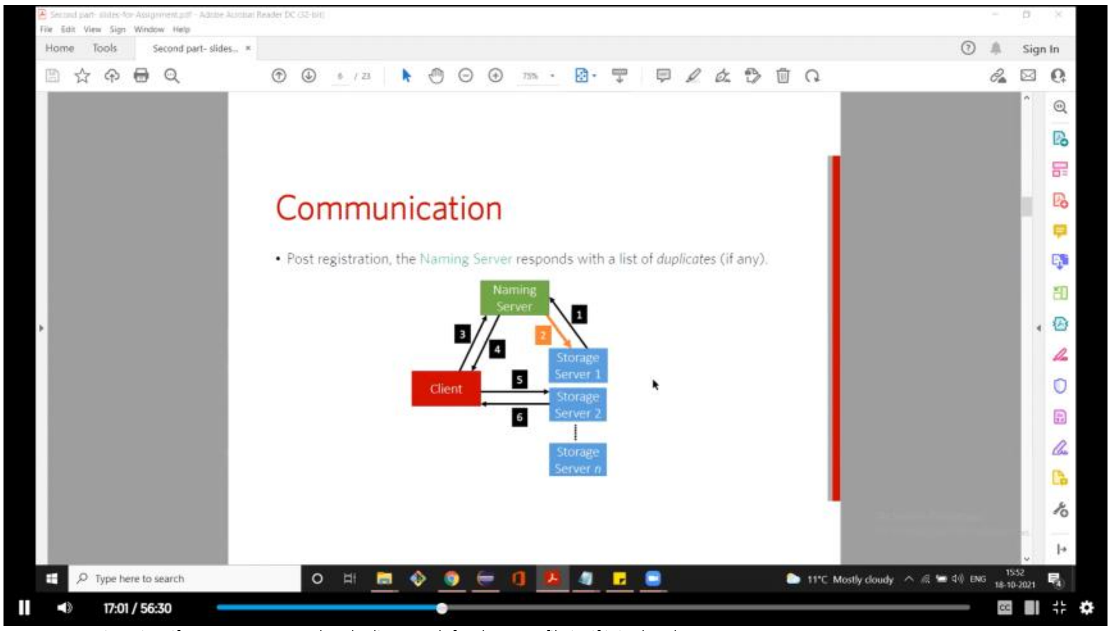


Communication

- Registration phase

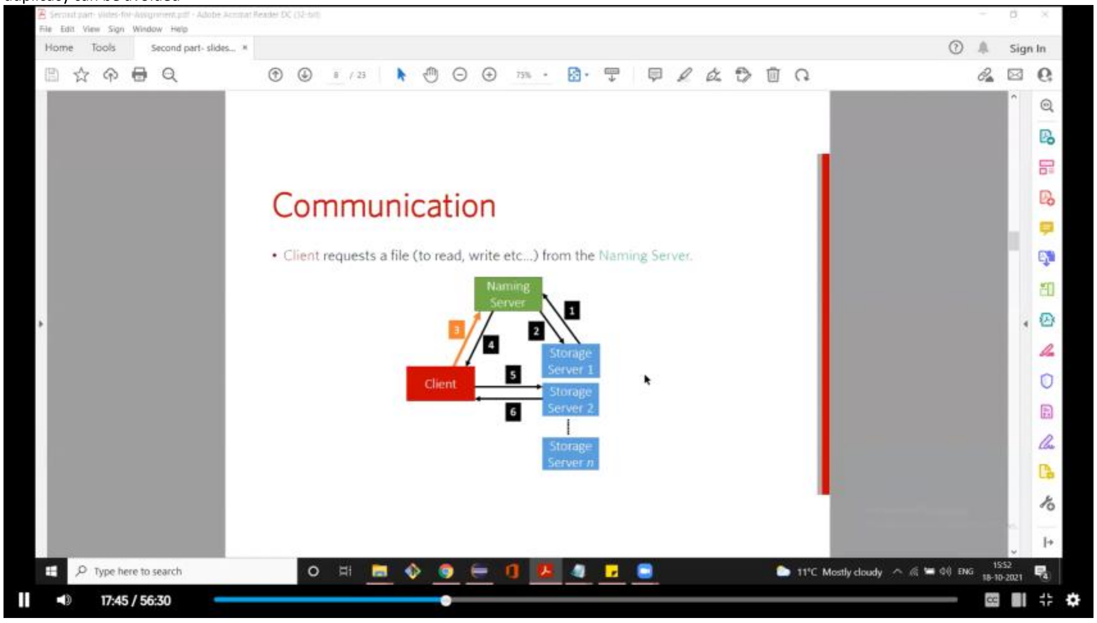
1. Storage should be registered with the naming server
2. Storage server sends a req with the req params to the naming server. NS stores data like Ipaddress, storagename or hostname or port and all the other files of the storage server
3. Client sends name to the NS -> NS returns an object of the Storage server using which client can perform actions on the storage server
4. Registration process is stored on the NS
5. Why are we using interfaces? ->
   Client does not have access to actual server. It has access to a proxy which "implements" the server object. Here the proxy is an instance

   But why is that? The reason is already mentioned in the text of the first example: The actual instance of RemotePrintServer resides in a different virtual machine (process) from the client process. Its actual implementation code is unknown to the client. The object returned by the registry to the client is just a Proxy instance which "implements" all interfaces of the original object which inherit from Remote. RemotePrintInterface is such an interface. If a method of t proxy object is called, it will send a message with the method name and parameters to the actual object on the server side. The server object will invoke the method and send back the results. Then the proxy's method can return as well. This way, the communication and "remoteness" of the procedure invocation is invisible (transparent) to the user of the client
6. SS, NS and Client a re on diff machine. There will be an interface which has all the required method which SS will use to register with the NS (first step)

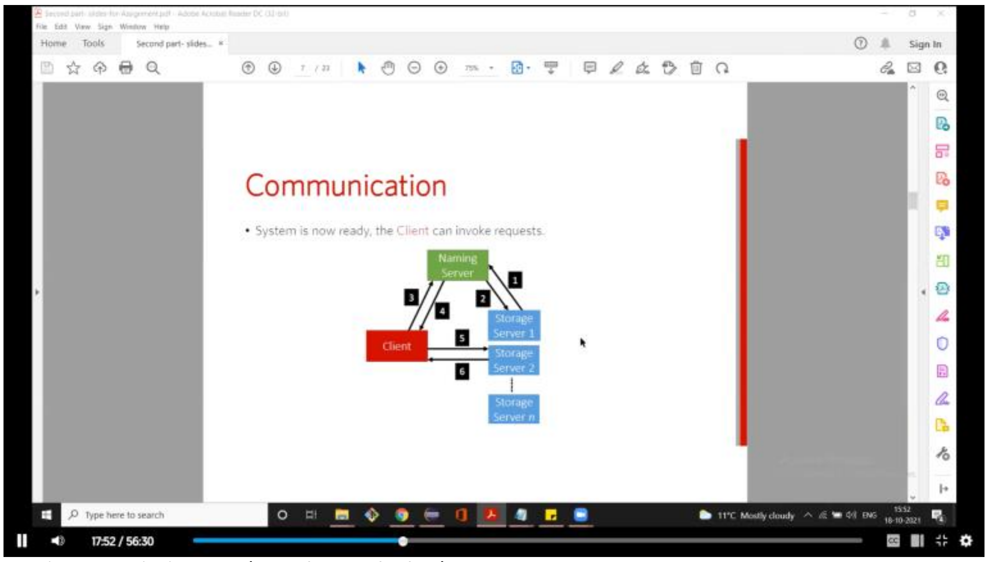

Communication

- Post registration, the Naming Server responds with a list of *duplicates* (if any).

1. Post registration, if storage server sends a duplicate path for the same file ie, if it is already present in the naming server, the NS send all the duplicate paths back to the SS so that this duplicacy can be avoided

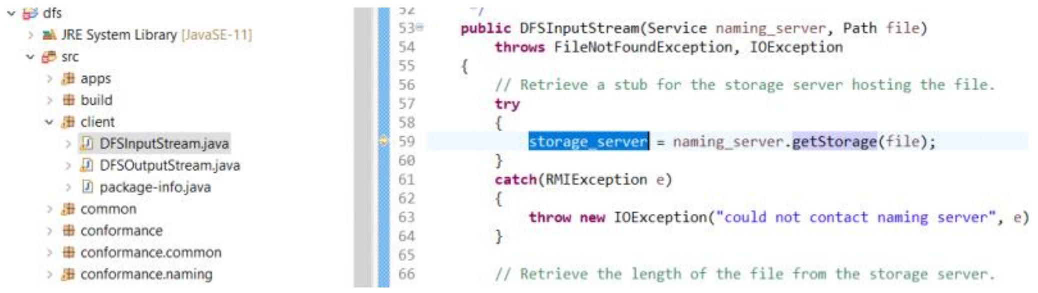

Communication

- Client requests a file (to read, write etc...) from the Naming Server.


Communication

- System is now ready, the Client can invoke requests.

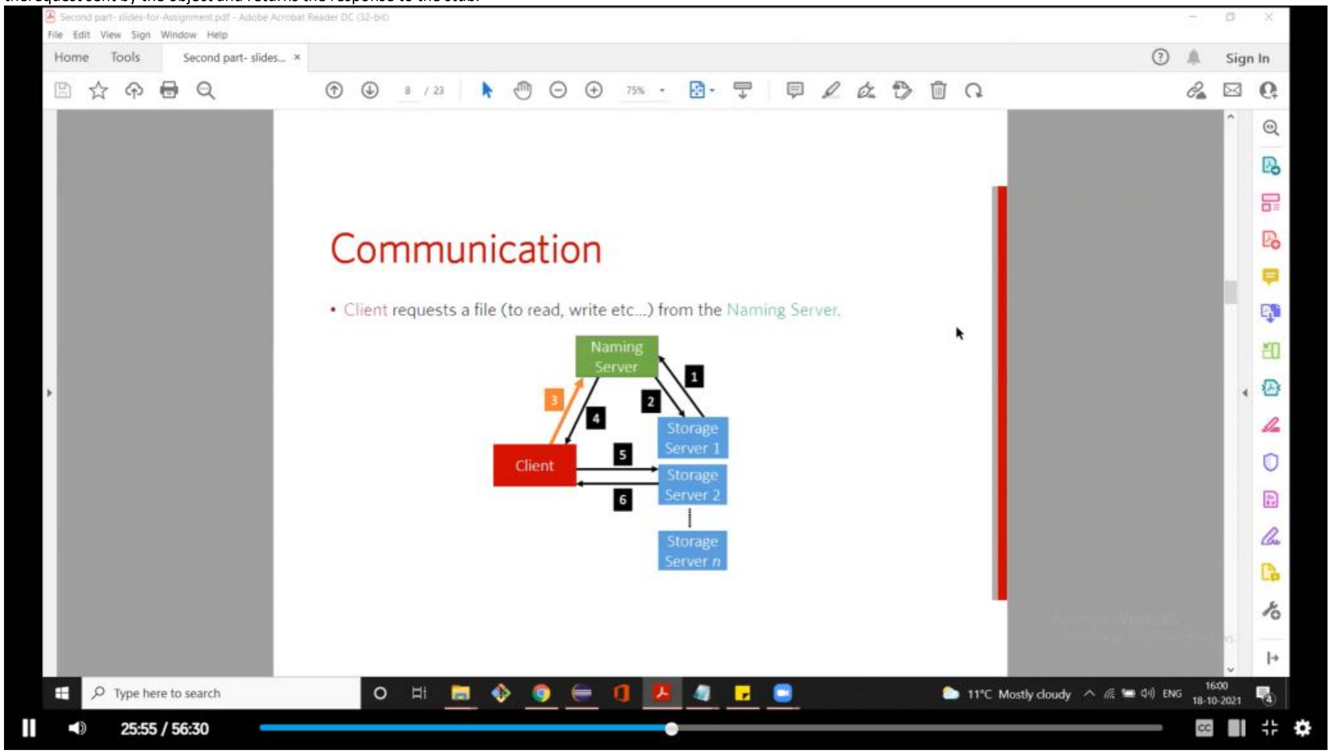
2. Now client can invoke the request **(no need to start the client)**
3. Client will communicate with NS. NS will return the info about the file to client. NS as list of paths and addresses for SS).
4. **Client only passes filename. NS already has mapping b/w name and directory or the paths.** Returns an object of that storage server back to the client. Client can do operations on the file using this object.



```java
    public DFSInputStream(Service naming_server, Path file)
        throws FileNotFoundException, IOException
    {
        // Retrieve a stub for the storage server hosting the file.
        try
        {
            storage_server = naming_server.getStorage(file);
        }
        catch(RMIException e)
        {
            throw new IOException("could not contact naming server", e)
        }

        // Retrieve the length of the file from the storage server.
```

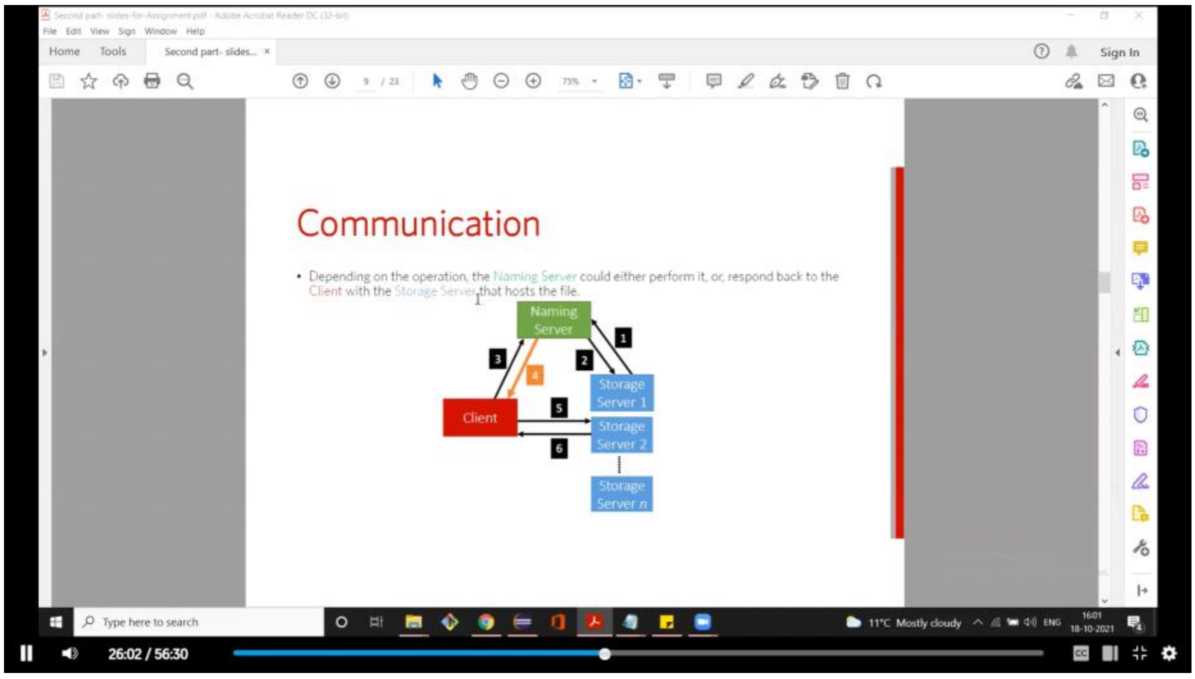
Refer getStorage
getStorage is implemented by namingServer

5. ServerStub is a proxy or an object of storage server to the client for a particular file when requested via naming server. Client can do read write create and delete operations using ServerStub
6. **A stub** is a remote object at the client side. It helps in unmarshalling and marshalling or serializing of data in the Client side. **It serves as a placeholder at the client side.** It communicates with the server side skeleton.
7. **Skeleton** dispatches a call to to an actual Remote object implementation. It delegates therequest sent by the object and returns the response to the stub.



Communication

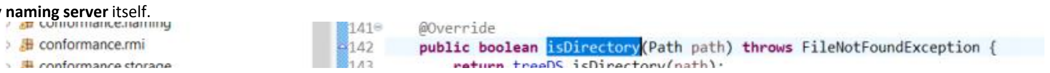
- Client requests a file (to read, write etc...) from the Naming Server.



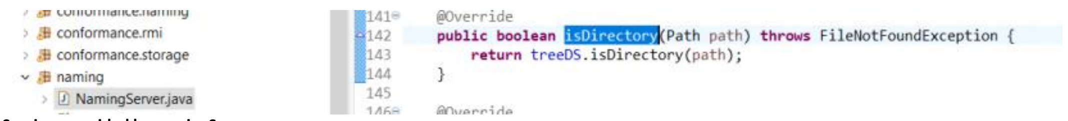
Communication

- Depending on the operation, the Naming Server could either perform it, or, respond back to the Client with the Storage Server that hosts the file.

8. Client can do read write create and delete operations using ServerStub. Some actions are done by **naming server** itself.



```java
    @Override
    public boolean isDirectory(Path path) throws FileNotFoundException {
        return treeDS.isDirectory(path);
```



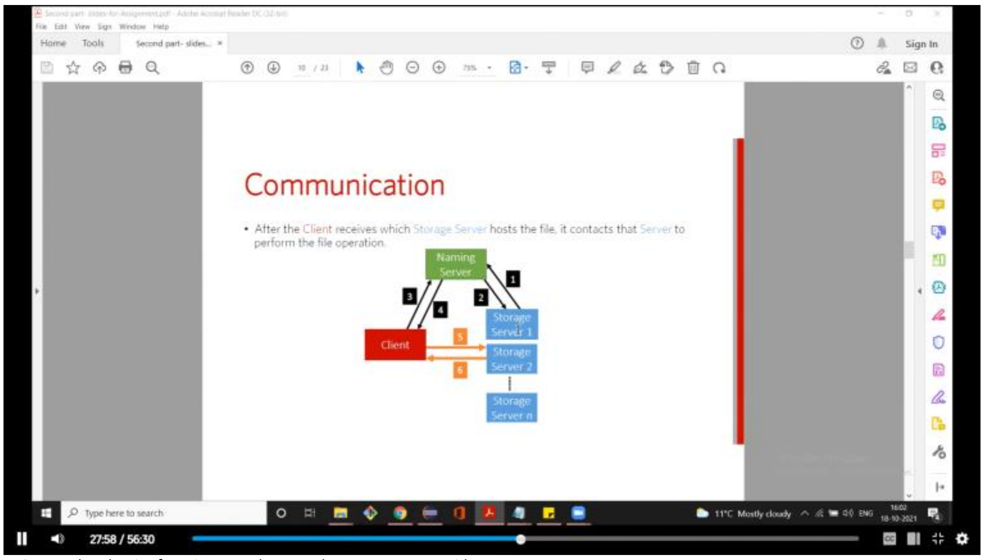
```
    @Override
    public boolean isDirectory(Path path) throws FileNotFoundException {
        return treeDS.isDirectory(path);
    }

    @Override
```

<u>Services provided by namingServer</u>:

Service.java ->
isDirectory,
List -> Lists the contents of a directory,
createFile,
createDirectory,
delete -> Path to the file or directory to be deleted, ***<u>Note this is done in naming server, not in the storage server</u>***
getStorage

Registration.java ->
register

For ex if a client wants to know if the file exist or not, the namingServer can do this action on its own as it already has mapping of name of the files with their corresponding locations.



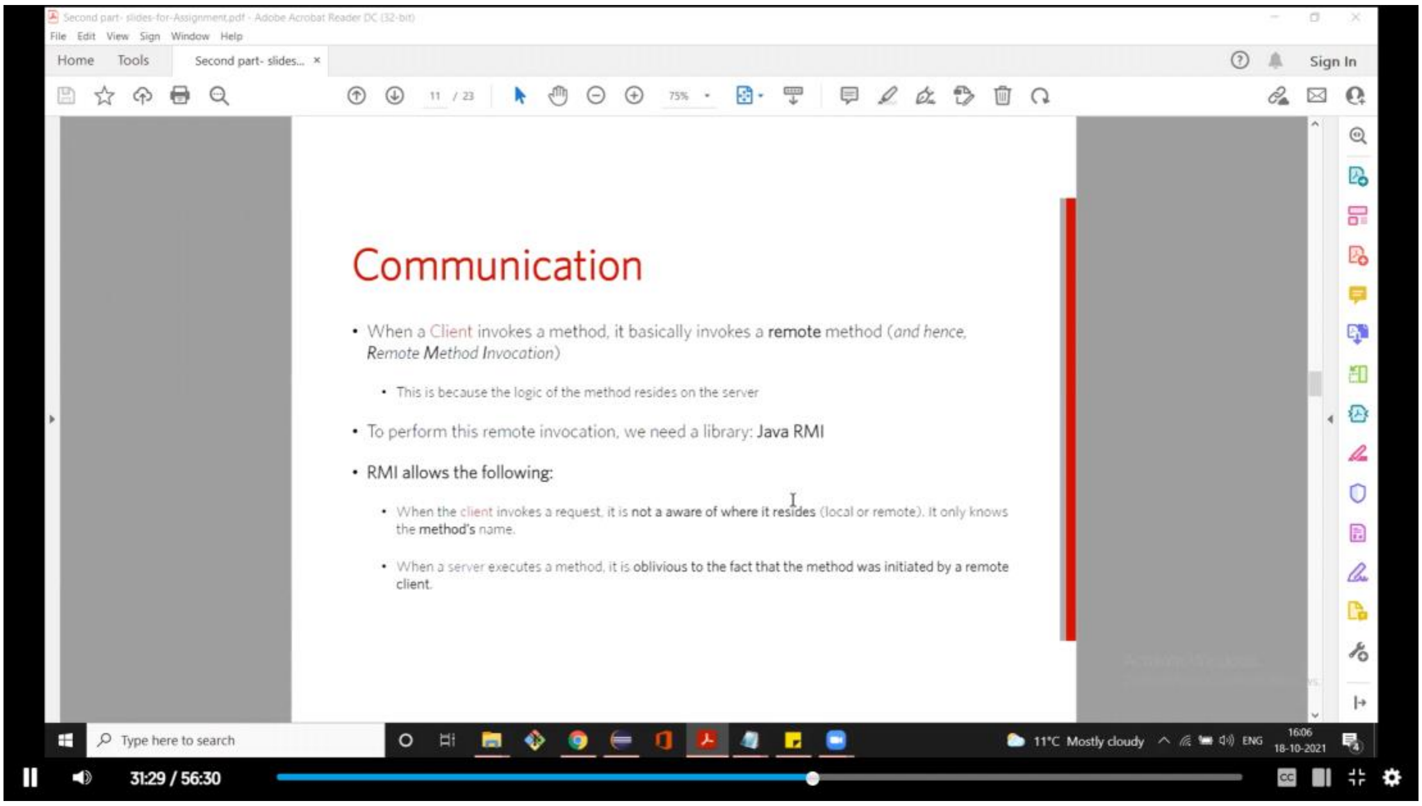
Communication

- After the Client receives which Storage Server hosts the file, it contacts that Server to perform the file operation.

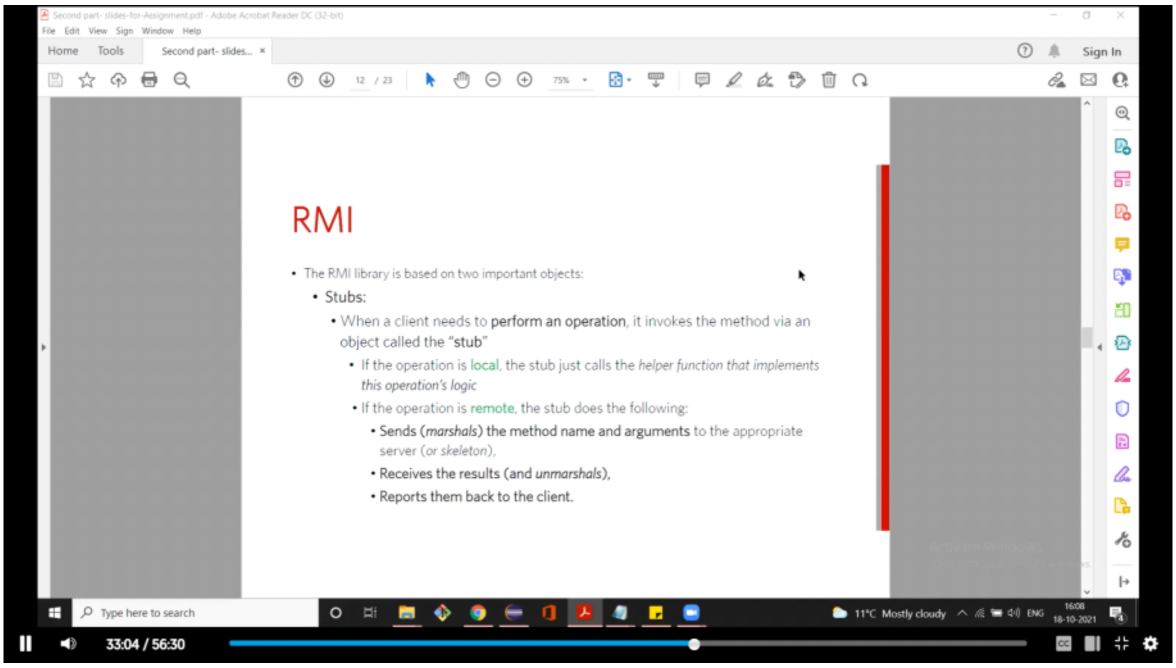
9. Read and write functions are done on the storage server side.
   <u>Services provided by StorageServer</u>:

   Storage.java ->
   size
   read
   write
   Command.java ->
   Create
   delete -> Deletes a file or directory on the storage server. ***<u>Note this is done in storage server, not in the naming server</u>***

<u>Proxy:</u>
1. Proxy is a structural design pattern, that provides the object that acts as a substitute for the real object
2. Stubs and servers are basically proxies, so to handle the connection beween the two, we will use **proxyHandler**
3. You need to implement **InvocationHandler**, which connects stub to the skeleton for your proxyHandler class (**ProxyHandler** implements **InvocationHandler** Interface which helps **connect stubs with the skeleton**)
4. Marshalling and unmarshalling are done by proxyHandler (And not stubs and skeleton)

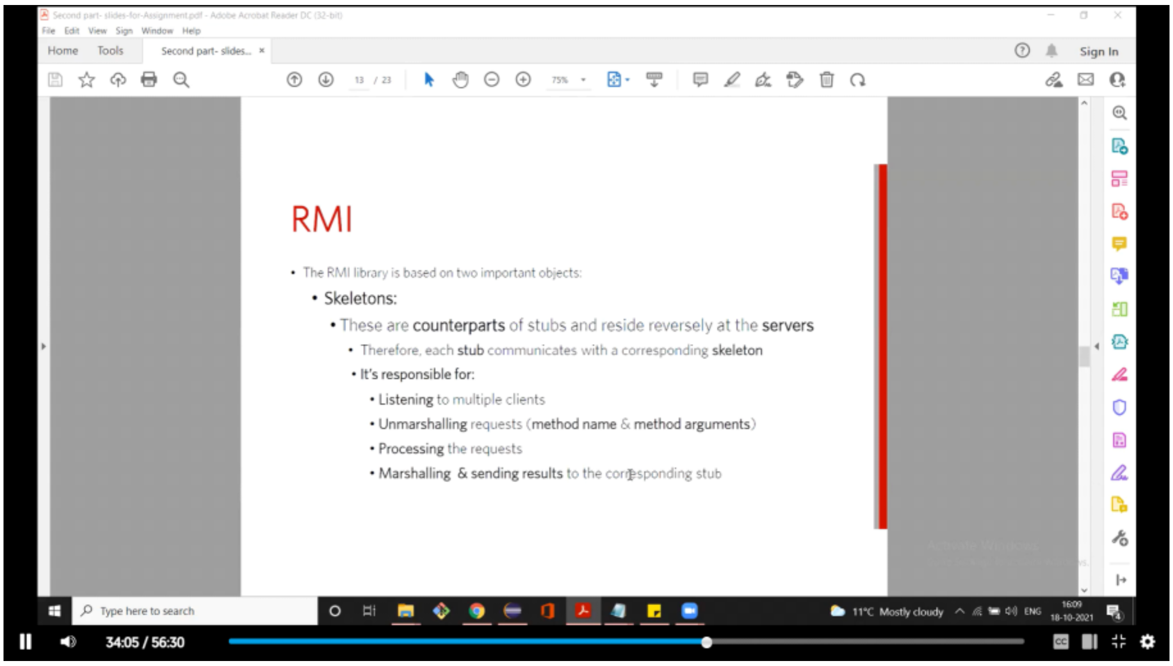

Communication

- When a Client invokes a method, it basically invokes a **remote** method (*and hence, Remote Method Invocation*)
  - This is because the logic of the method resides on the server
- To perform this remote invocation, we need a library: Java RMI
- RMI allows the following:
  - When the client invokes a request, it is not a aware of where it resides (local or remote). It only knows the method's name.
  - When a server executes a method, it is oblivious to the fact that the method was initiated by a remote client.


RMI

- The RMI library is based on two important objects:
  - Stubs:
    - When a client needs to **perform an operation**, it invokes the method via an object called the "stub"
      - If the operation is local, the stub just calls the *helper function that implements this operation's logic*
      - If the operation is remote, the stub does the following:
        - Sends (*marshals*) the method name and arguments to the appropriate server (or *skeleton*),
        - Receives the results (and *unmarshals*),
        - Reports them back to the client.

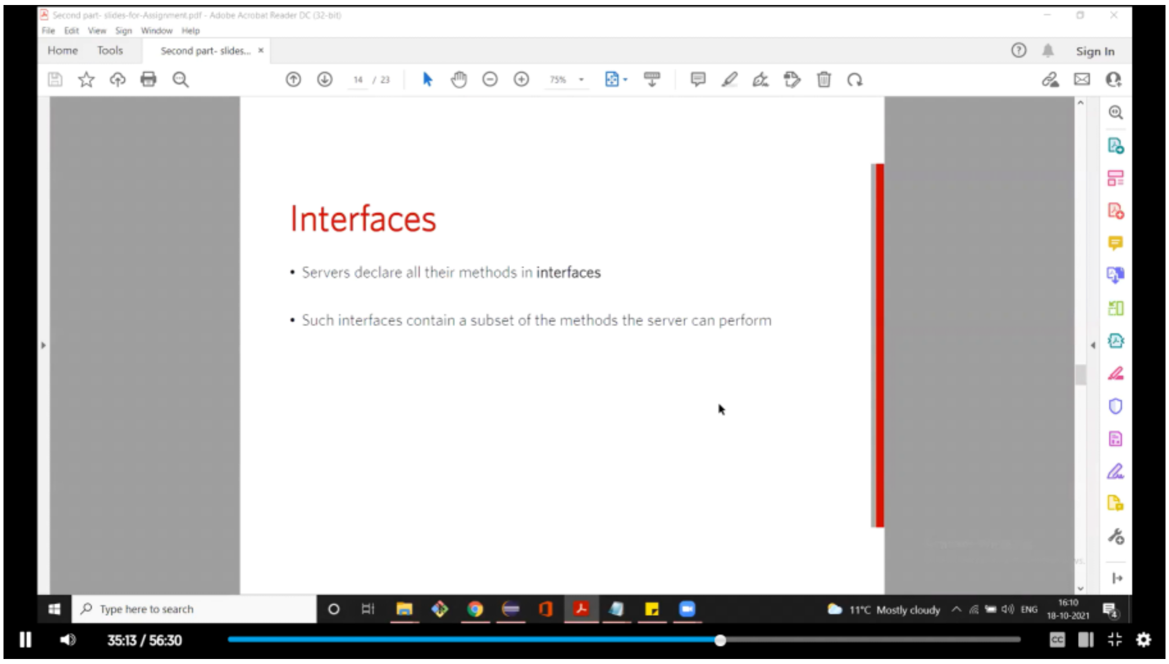
- **The stub do the marshalling and unmarshalling of the request. ProxyHandler handles the communication between the client and server**



RMI

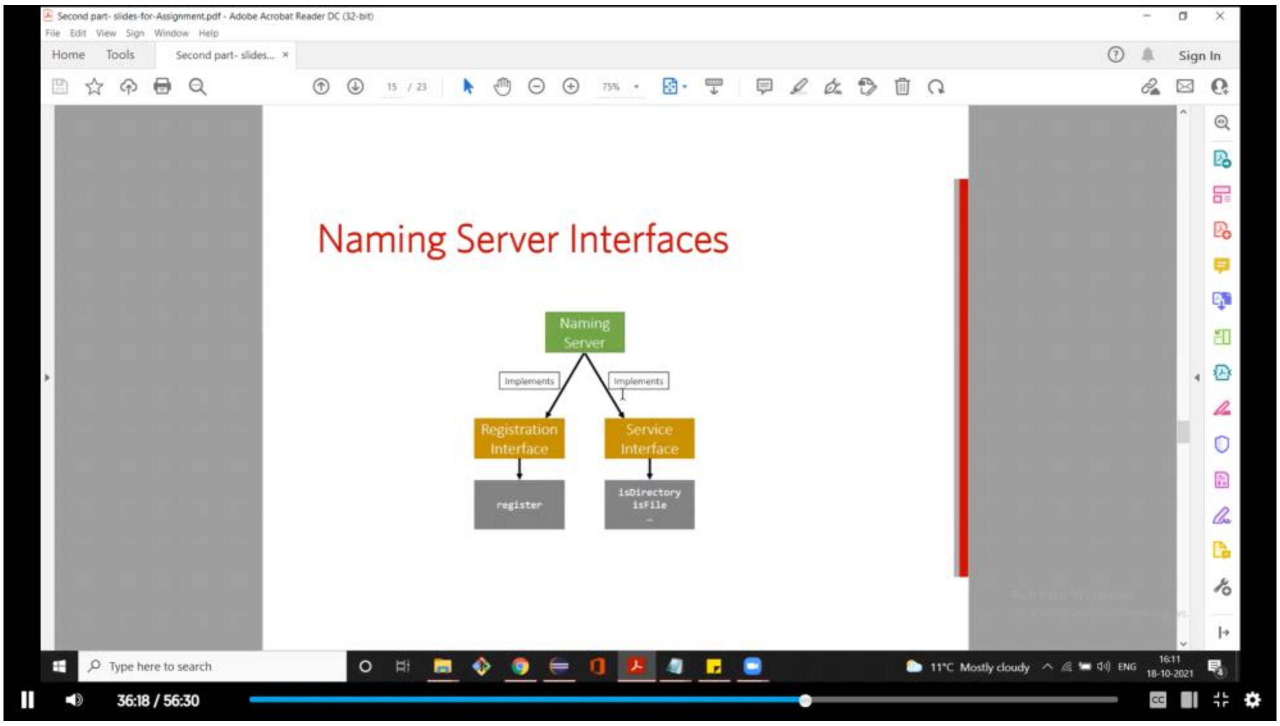
- The RMI library is based on two important objects:
  - Skeletons:
    - These are counterparts of stubs and reside reversely at the servers
      - Therefore, each stub communicates with a corresponding skeleton
      - It's responsible for:
        - Listening to multiple clients
        - Unmarshalling requests (method name & method arguments)
        - Processing the requests
        - Marshalling & sending results to the corresponding stub

- Example, If client wants to connect with Naming server, it should an object of NS skeleton. Similarily for storage server
- If NS and SS wants to communicate, they have their own pair of stubs and skeletons



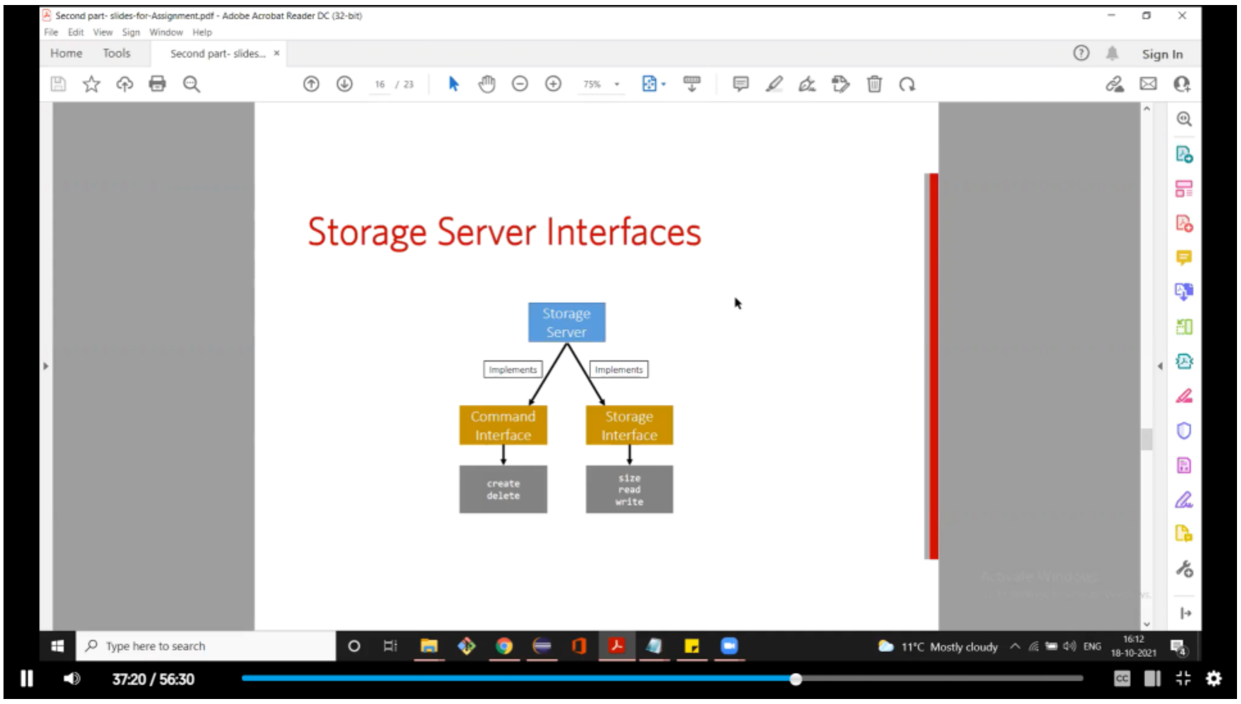
Interfaces

- Servers declare all their methods in interfaces
- Such interfaces contain a subset of the methods the server can perform



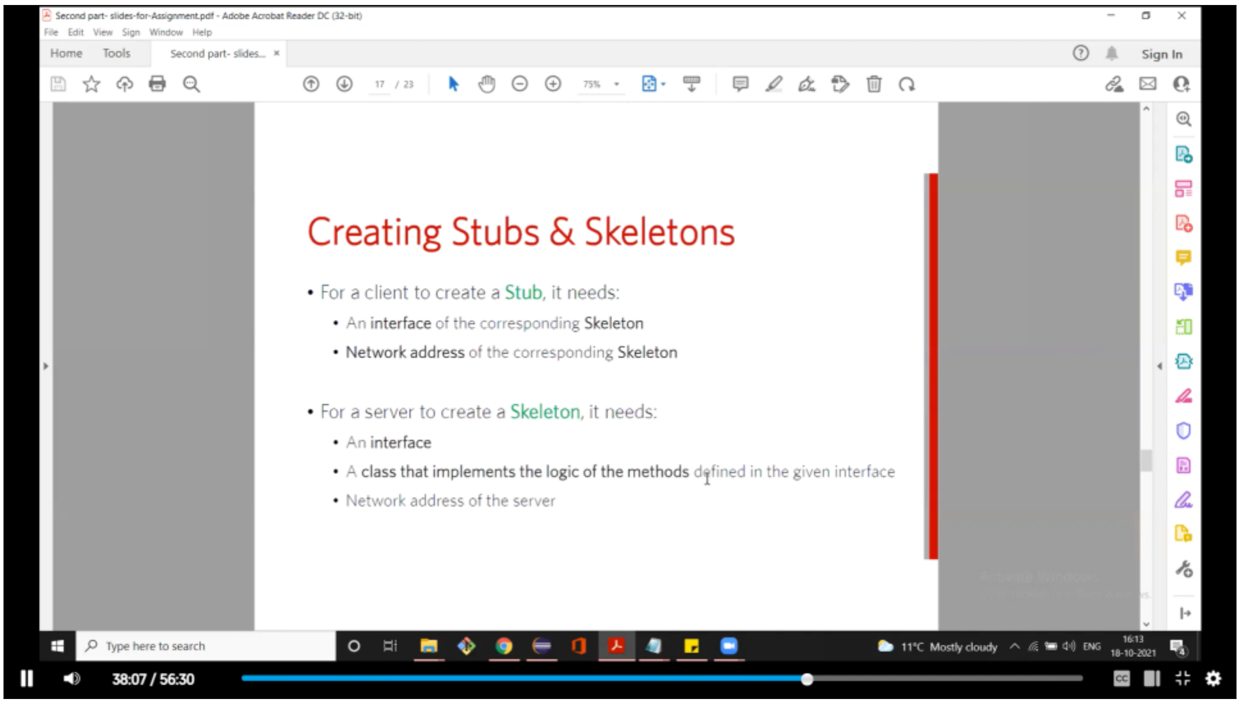
Naming Server Interfaces

- Registration Interface is for SS. Example registration and unregistration
- Service Interface is for client



Storage Server Interfaces

- Storage Interface contains info about the storage
- Command I. will have ops for file handling

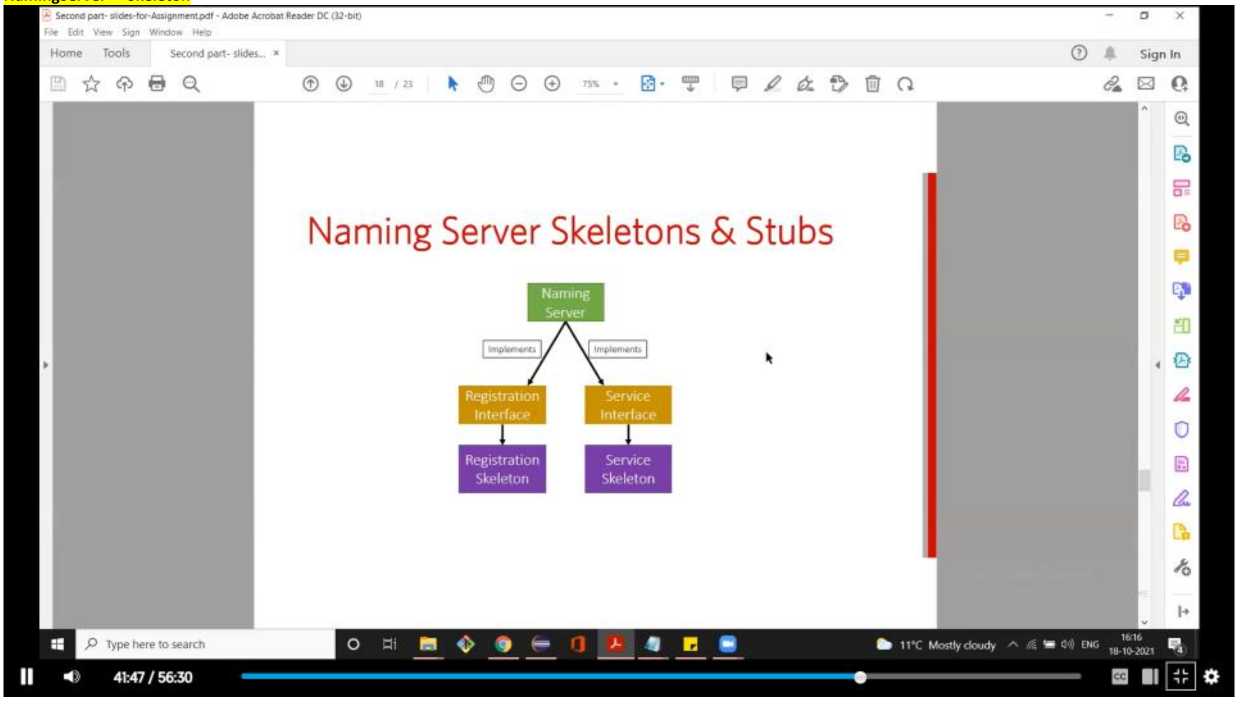


Creating Stubs & Skeletons

- For a client to create a Stub, it needs:
  - An interface of the corresponding Skeleton
  - Network address of the corresponding Skeleton
- For a server to create a Skeleton, it needs:
  - An interface
  - A class that implements the logic of the methods defined in the given interface
  - Network address of the server

To understand **Stub**, follow
Launcher -> StorageServerApp -> StorageServer -> Stub

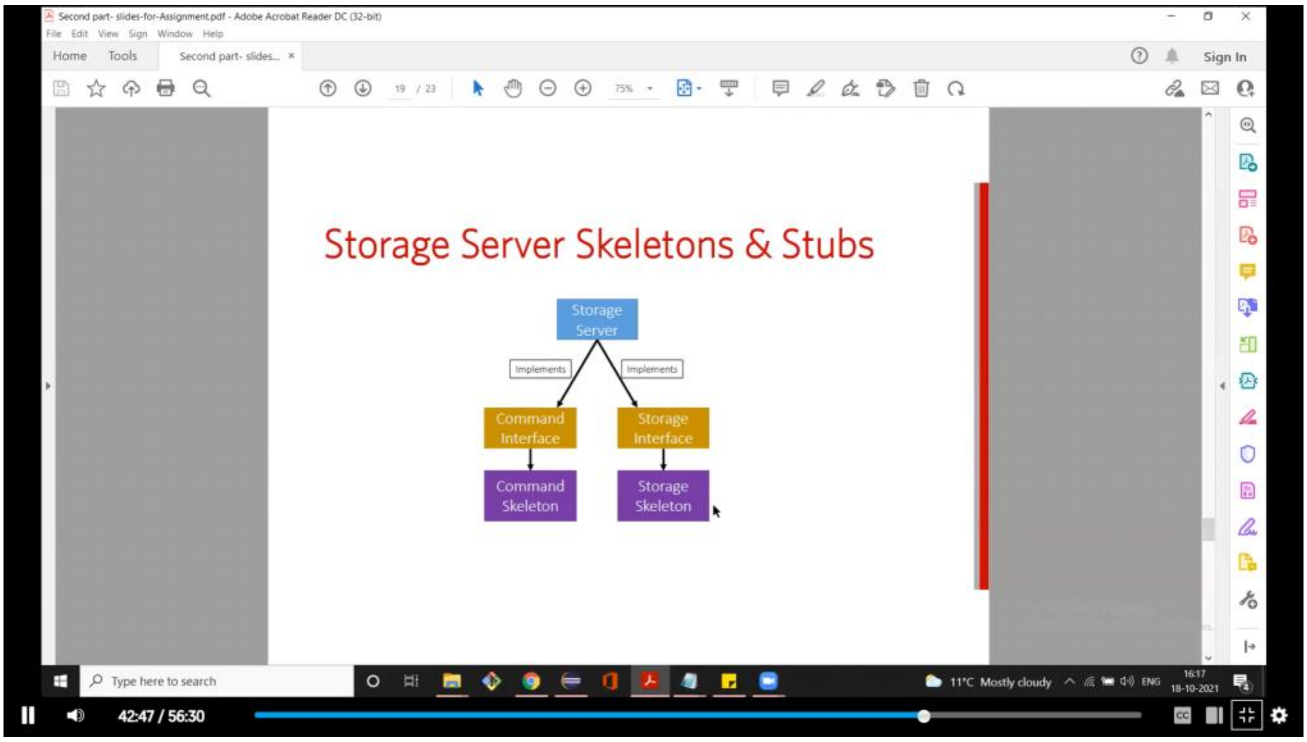
To understand **Skeleton**, follow
NamingServer -> Skeleton



Naming Server Skeletons & Stubs

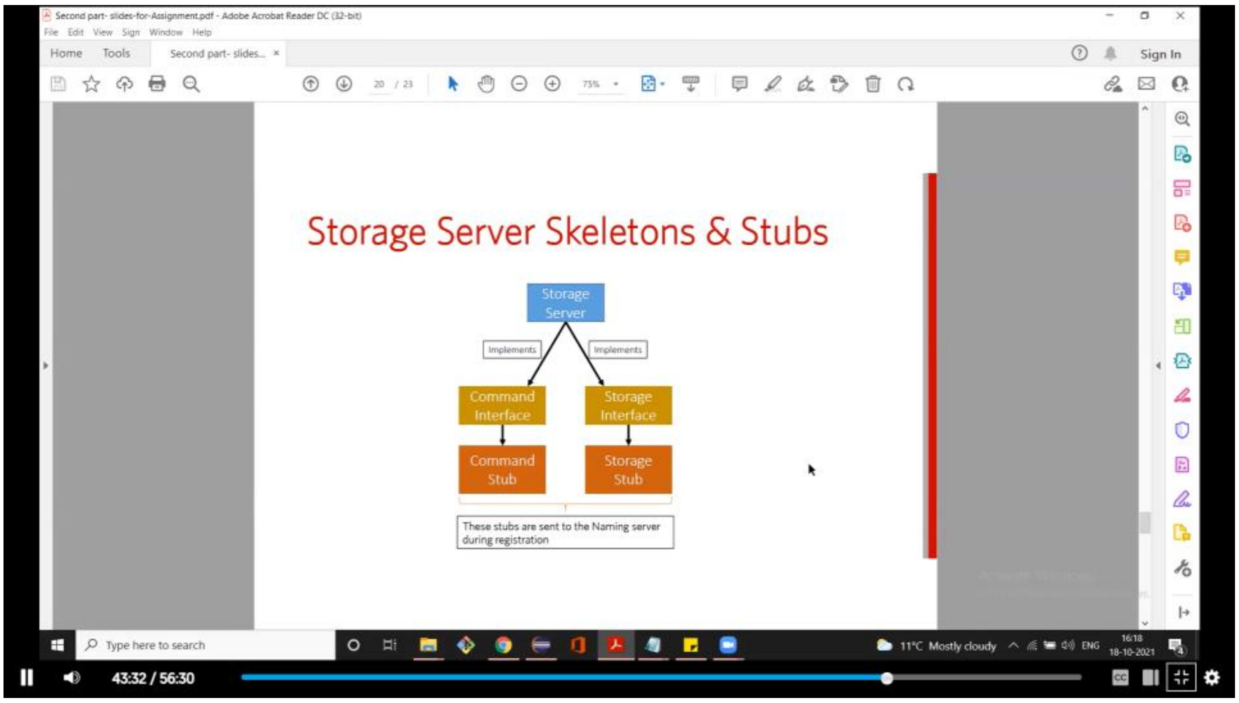
$$\begin{tikzcd}
 & \text{Naming Server} \arrow[dl, "\text{Implements}"'] \arrow[dr, "\text{Implements}"] & \\
\text{Registration Interface} \arrow[d] & & \text{Service Interface} \arrow[d] \\
\text{Registration Skeleton} & & \text{Service Skeleton}
\end{tikzcd}$$

- Registration Interface is for Storage Server. Example registration and unregistration
- Service Interface is for client



Storage Server Skeletons & Stubs

- Storage Interface contains info about the storage
- Command I. will have ops for file handling

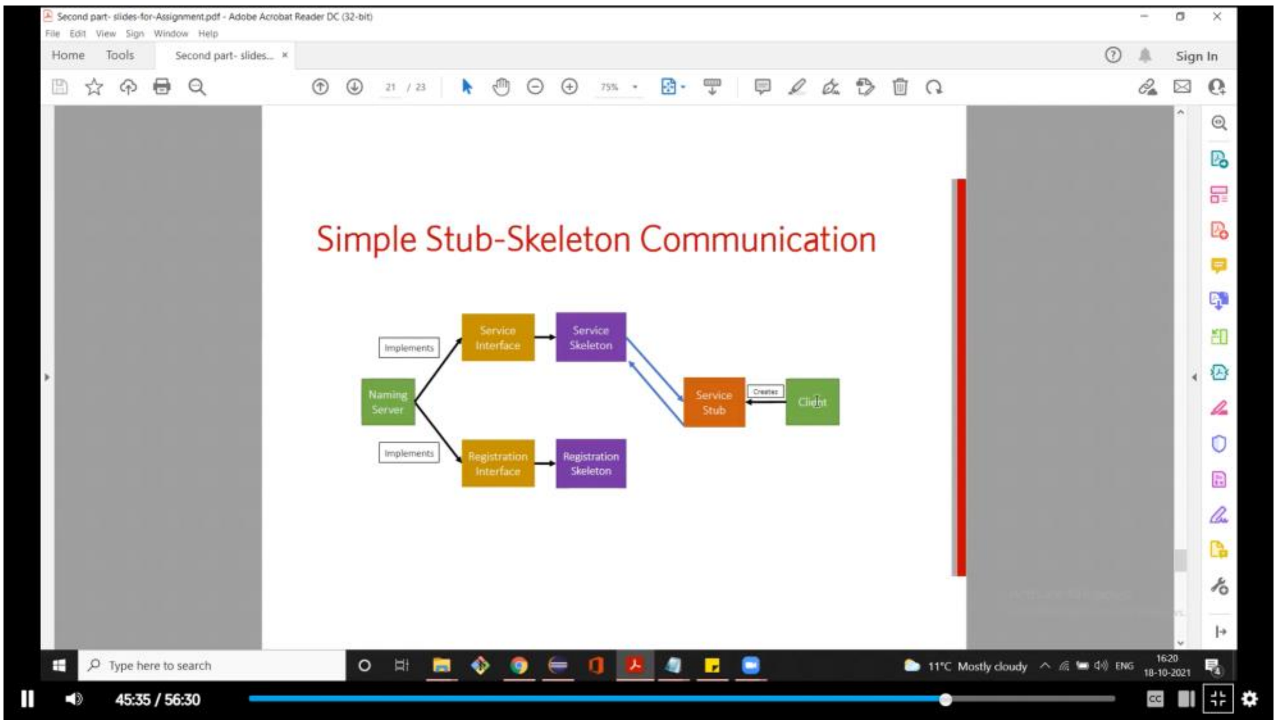


Storage Server Skeletons & Stubs

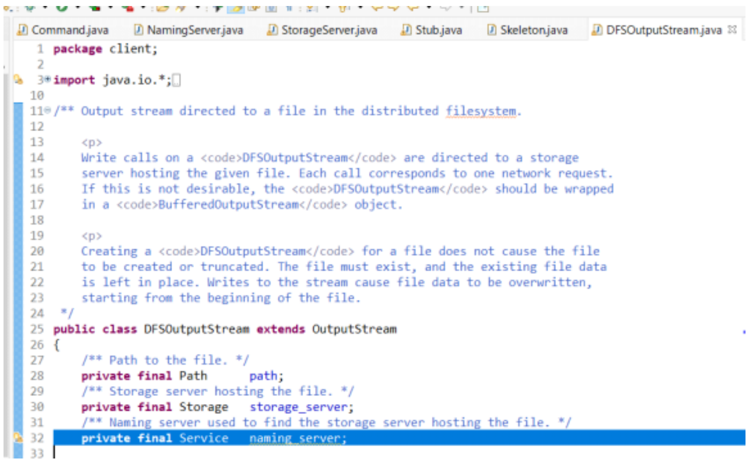
These stubs are sent to the Naming server during registration

Summary slides for Lab 1

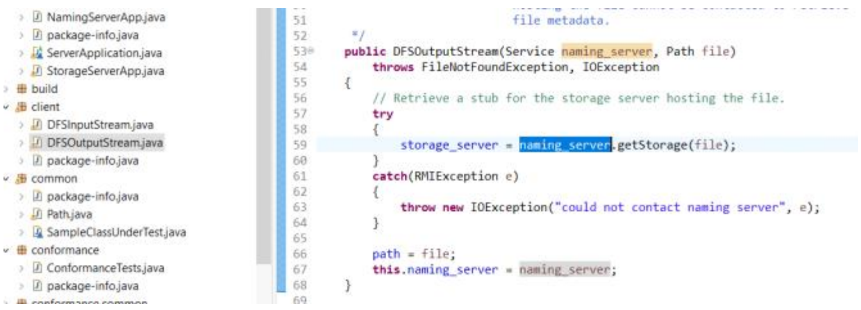
30 October 2021 21:52



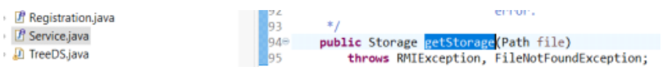
Client to naming server:



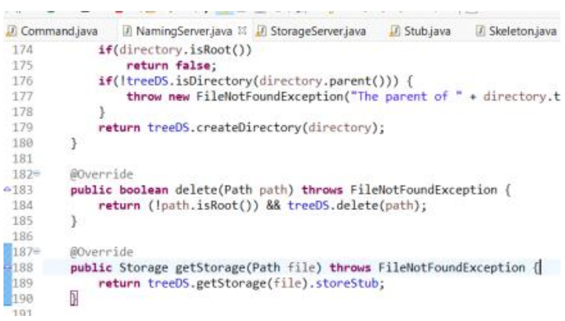
```
package client;

import java.io.*;

/** Output stream directed to a file in the distributed filesystem.

    <p>
    Write calls on a <code>DFSOutputStream</code> are directed to a storage
    server hosting the given file. Each call corresponds to one network request.
    If this is not desirable, the <code>DFSOutputStream</code> should be wrapped
    in a <code>BufferedOutputStream</code> object.

    <p>
    Creating a <code>DFSOutputStream</code> for a file does not cause the file
    to be created or truncated. The file must exist, and the existing file data
    is left in place. Writes to the stream cause file data to be overwritten,
    starting from the beginning of the file.
 */
public class DFSOutputStream extends OutputStream
{
    /** Path to the file. */
    private final Path      path;
    /** Storage server hosting the file. */
    private final Storage   storage_server;
    /** Naming server used to find the storage server hosting the file. */
    private final Service   naming_server;
```



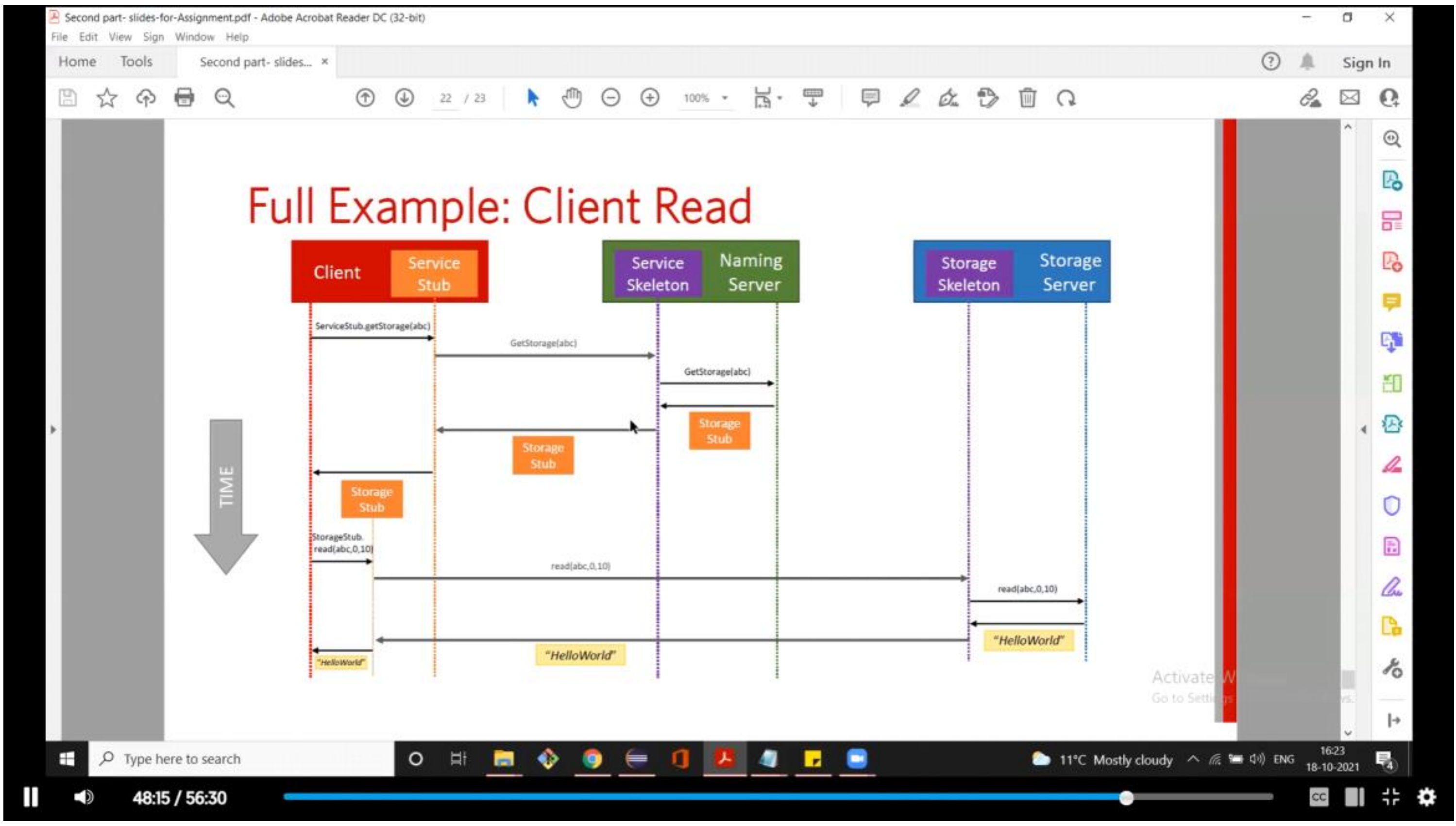
```
            file metadata.
     */
    public DFSOutputStream(Service naming_server, Path file)
        throws FileNotFoundException, IOException
    {
        // Retrieve a stub for the storage server hosting the file.
        try
        {
            storage_server = naming_server.getStorage(file);
        }
        catch(RMIException e)
        {
            throw new IOException("could not contact naming server", e);
        }

        path = file;
        this.naming_server = naming_server;
    }
```



```
     */
    public Storage getStorage(Path file)
        throws RMIException, FileNotFoundException;
```



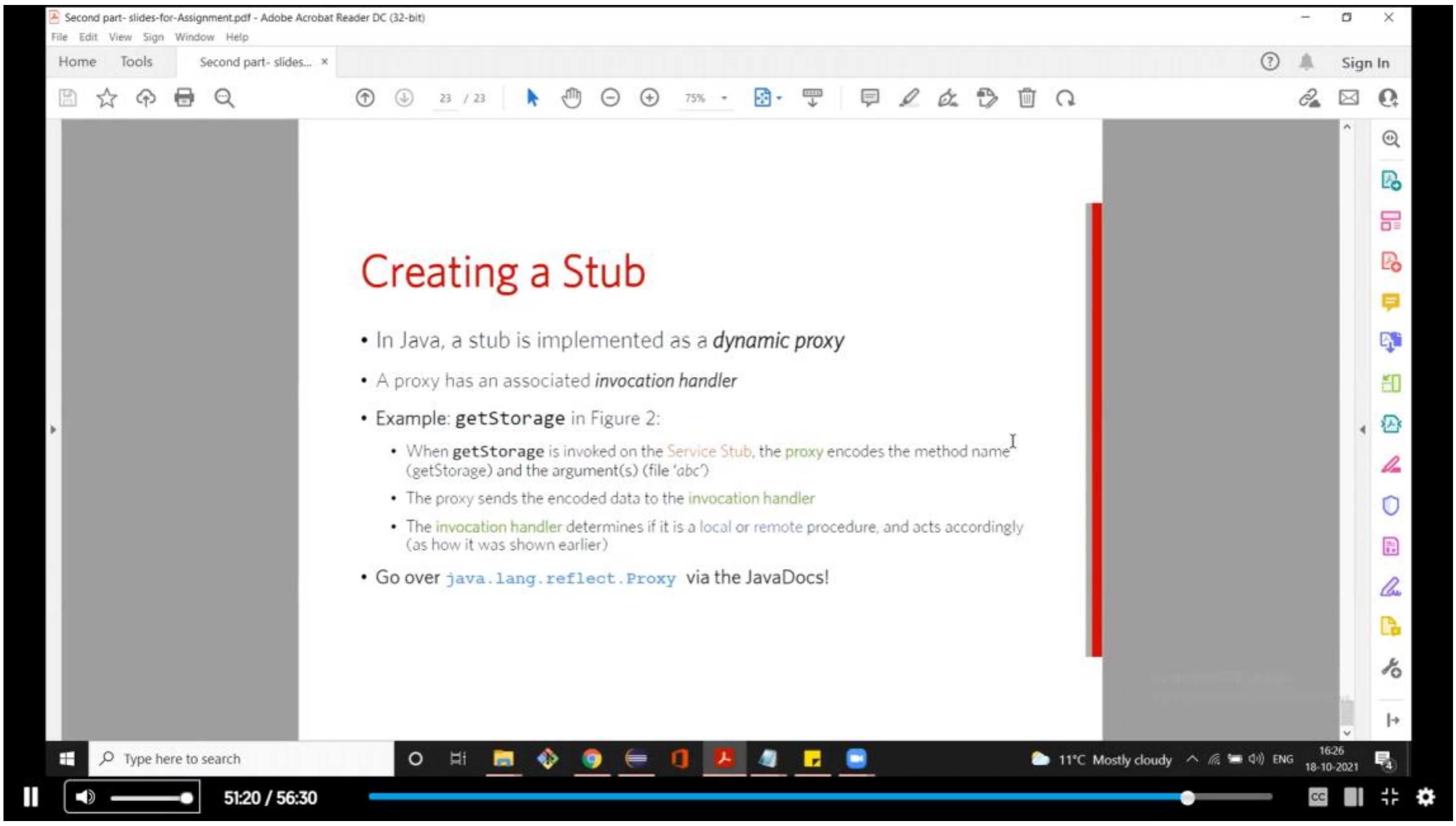
```
        if(directory.isRoot())
            return false;
        if(!treeDS.isDirectory(directory.parent())) {
            throw new FileNotFoundException("The parent of " + directory.t
        }
        return treeDS.createDirectory(directory);
    }

    @Override
    public boolean delete(Path path) throws FileNotFoundException {
        return (!path.isRoot()) && treeDS.delete(path);
    }

    @Override
    public Storage getStorage(Path file) throws FileNotFoundException {
        return treeDS.getStorage(file).storeStub;
    }
```



1. On first getStorage request by the client, it has received the storage stub.
2. SO for further ops like read, it does not need to connect to naming server anymore . In this case, storage stub by client is being used to communicate with Storage server using storage skeleton

Lab 1 - Dynamic proxy

30 October 2021 21:58



Creating a Stub

- In Java, a stub is implemented as a *dynamic proxy*
- A proxy has an associated *invocation handler*
- Example: `getStorage` in Figure 2:
  - When `getStorage` is invoked on the Service Stub, the proxy encodes the method name (getStorage) and the argument(s) (file '*abc*')
  - The proxy sends the encoded data to the invocation handler
  - The invocation handler determines if it is a local or remote procedure, and acts accordingly (as how it was shown earlier)
- Go over `java.lang.reflect.Proxy` via the JavaDocs!

Second part- slides-for-Assignment.pdf - Adobe Acrobat Reader DC (32-bit)
51:20 / 56:30

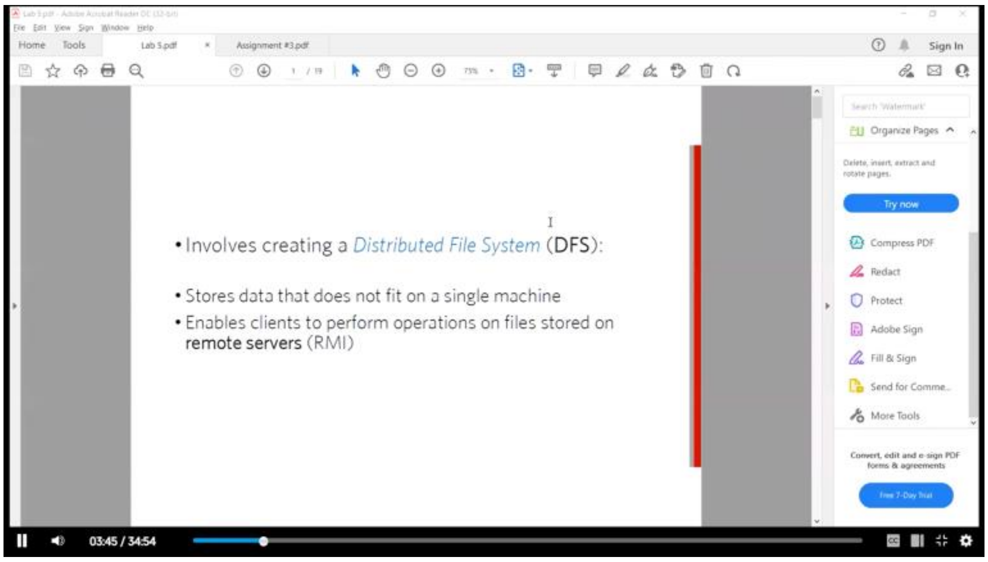
- Dynamic proxy is a class that implements the list of interfaces that is specified at run time
- Invocation handler provides communication between stub and skeleton

Design Patterns in Java : Dynamic Proxy for Logging

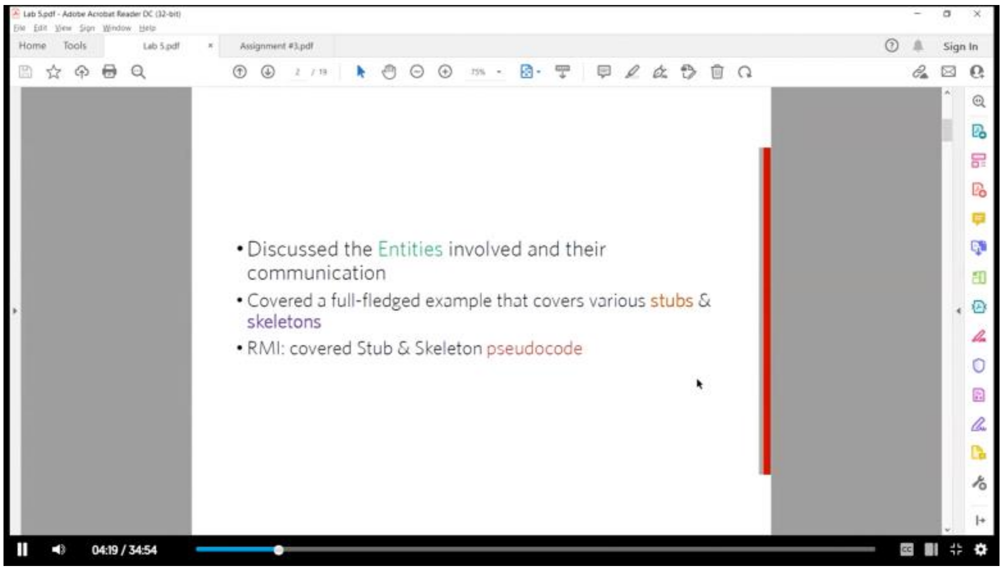
1. Dynamic proxy is created at runtime as supposed to compile time
2. At runtime you can take an object and build a wrapper around it and intercept all calls to its every single methods
3. InvocationHanlder is a reflection interface which allows us to intercept diff methods using invoke method
4. Invoke method is an idea that you want to invoke a method, with the given arguments

Lab 2

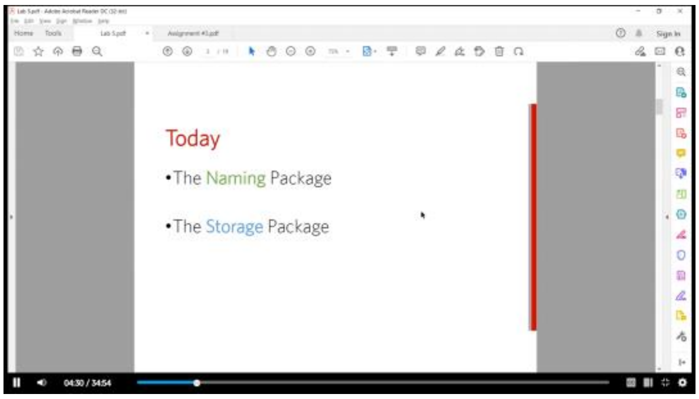
31 October 2021 14:46



- Involves creating a *Distributed File System* (**DFS**):
  - Stores data that does not fit on a single machine
  - Enables clients to perform operations on files stored on **remote servers** (RMI)

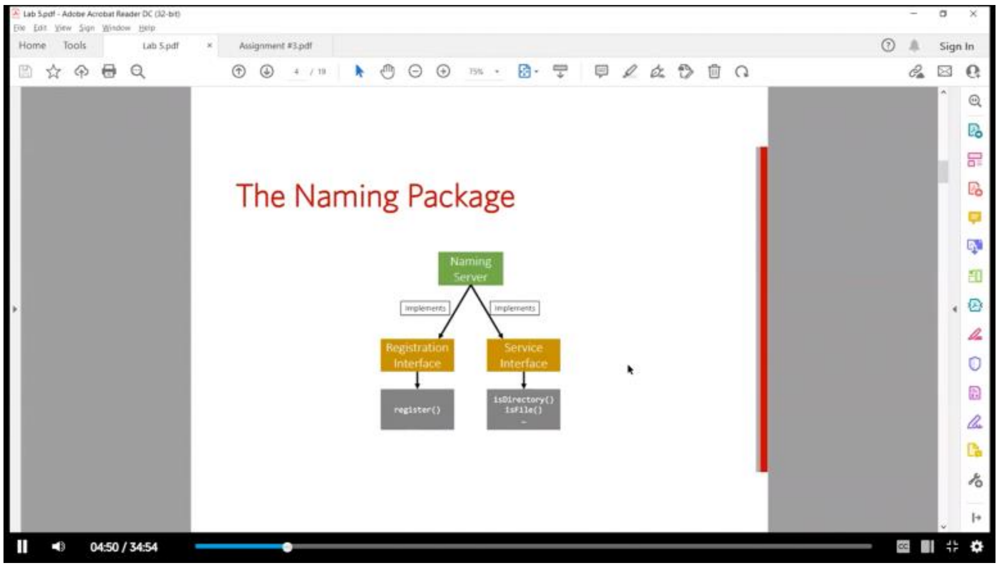


- Discussed the Entities involved and their communication
- Covered a full-fledged example that covers various stubs & skeletons
- RMI: covered Stub & Skeleton pseudocode

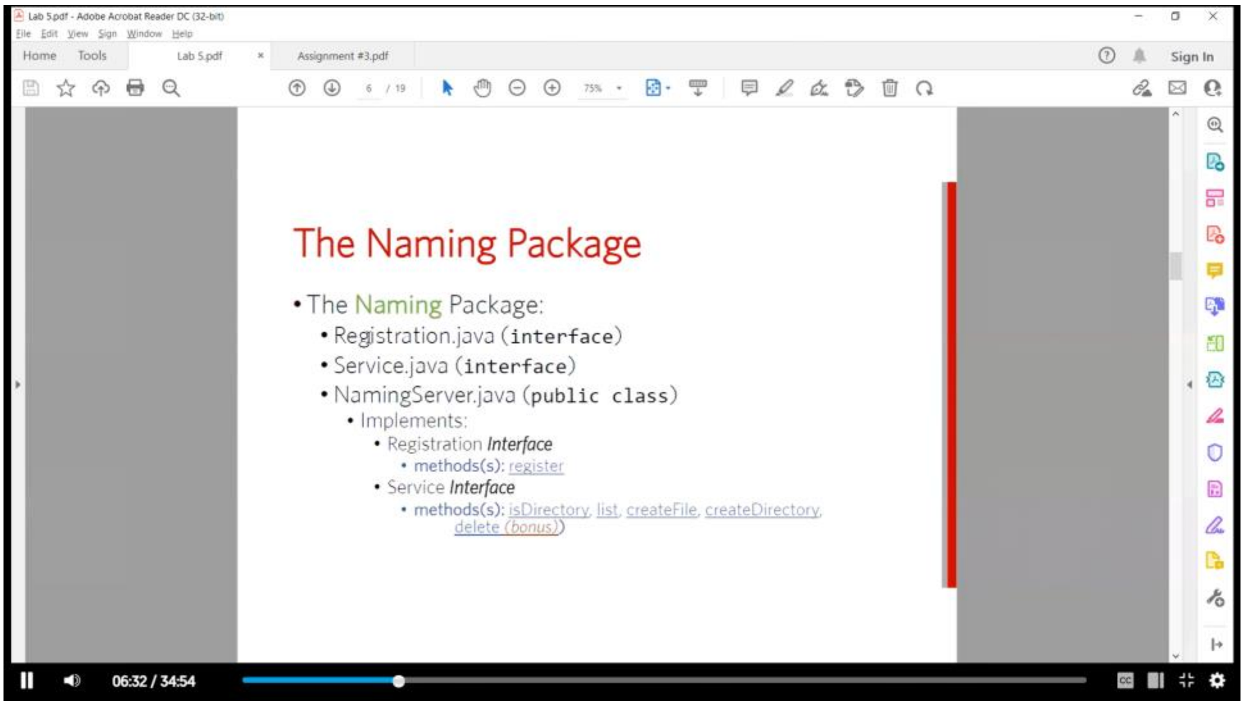


Today

- The Naming Package
- The Storage Package



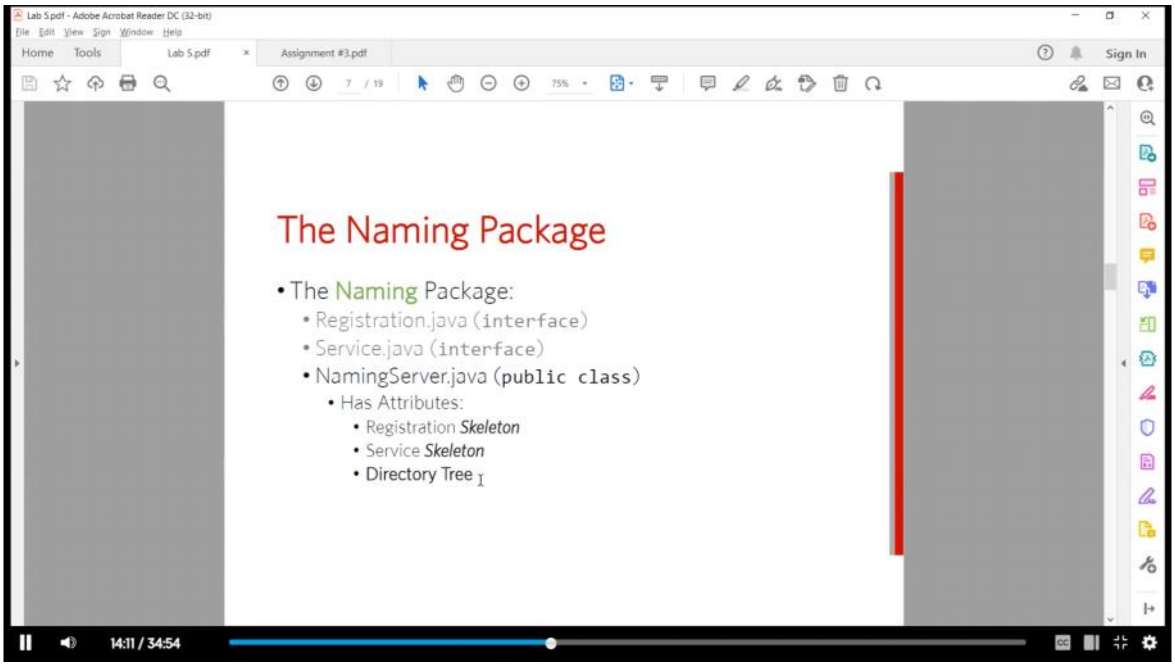
The Naming Package



The Naming Package

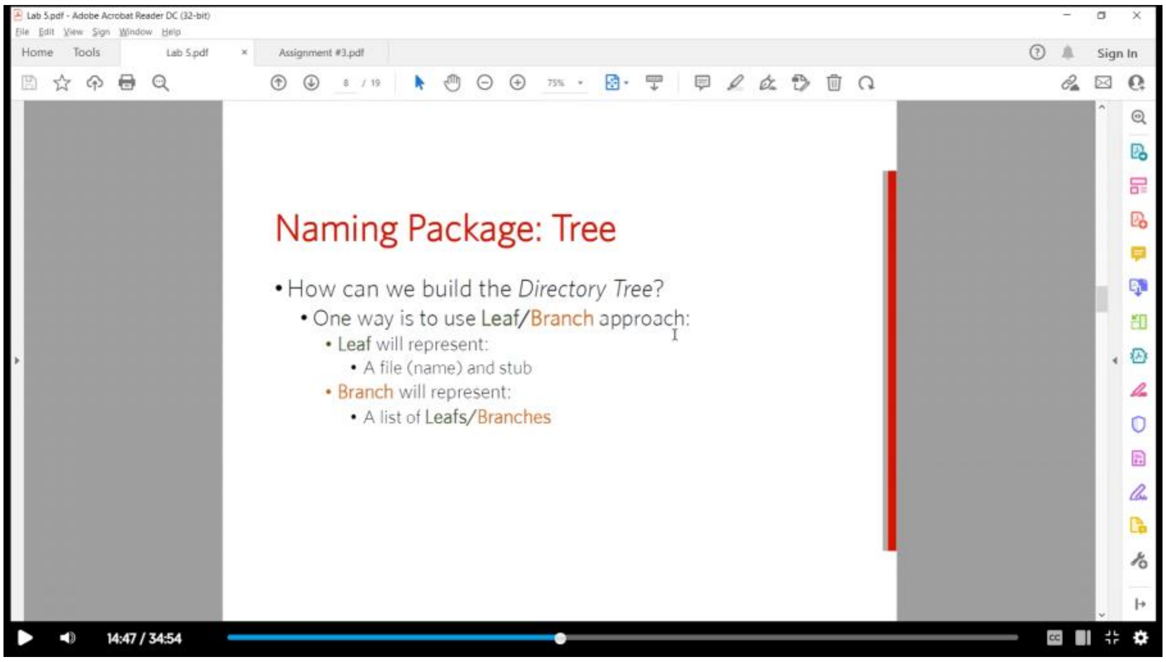
- The Naming Package:
  - Registration.java (interface)
  - Service.java (interface)
  - NamingServer.java (public class)
    - Implements:
      - Registration *Interface*
        - methods(s): register
      - Service *Interface*
        - methods(s): isDirectory, list, createFile, createDirectory, delete (bonus))

- Delete method in service interface uses the same delete method used in command interface
- Directory tree -> inner node are representing directories -> Leaves are rep files
- The metadata of storage that gets store din NS is **file-stub tuple** (file and a stub)



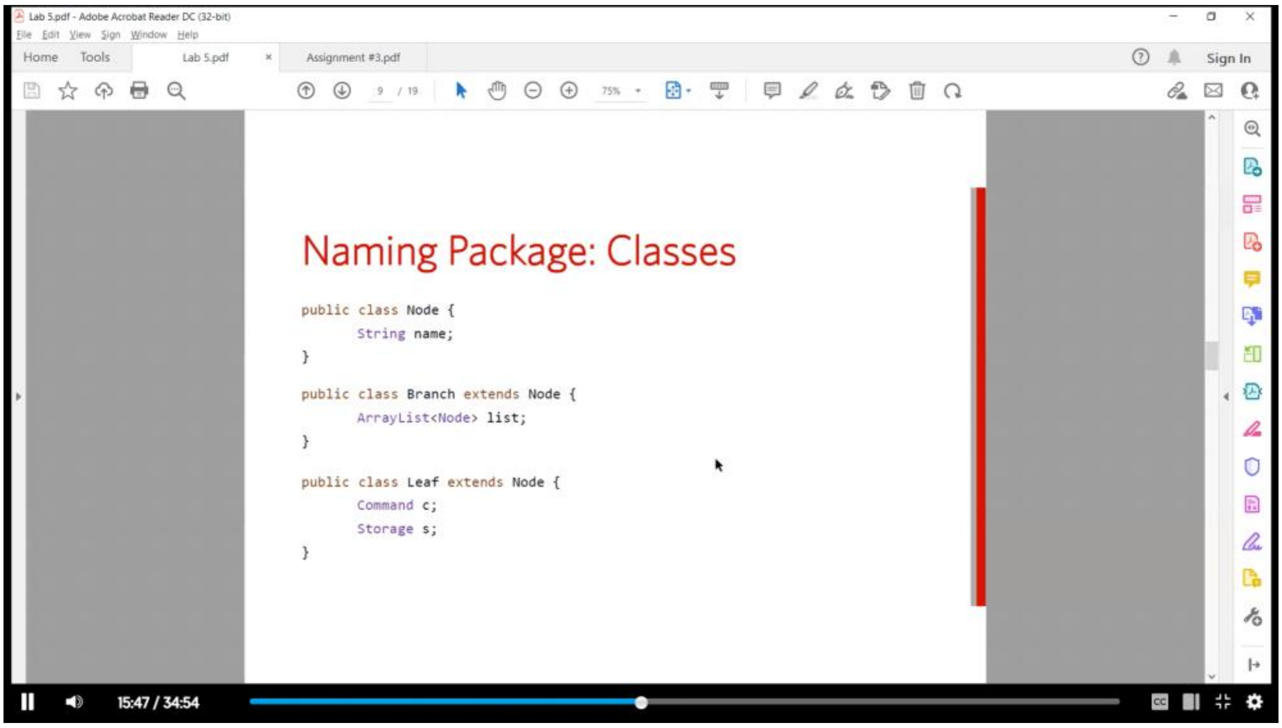
The Naming Package

- The Naming Package:
  - Registration.java (interface)
  - Service.java (interface)
  - NamingServer.java (public class)
    - Has Attributes:
      - Registration *Skeleton*
      - Service *Skeleton*
      - Directory Tree


Naming Package: Tree

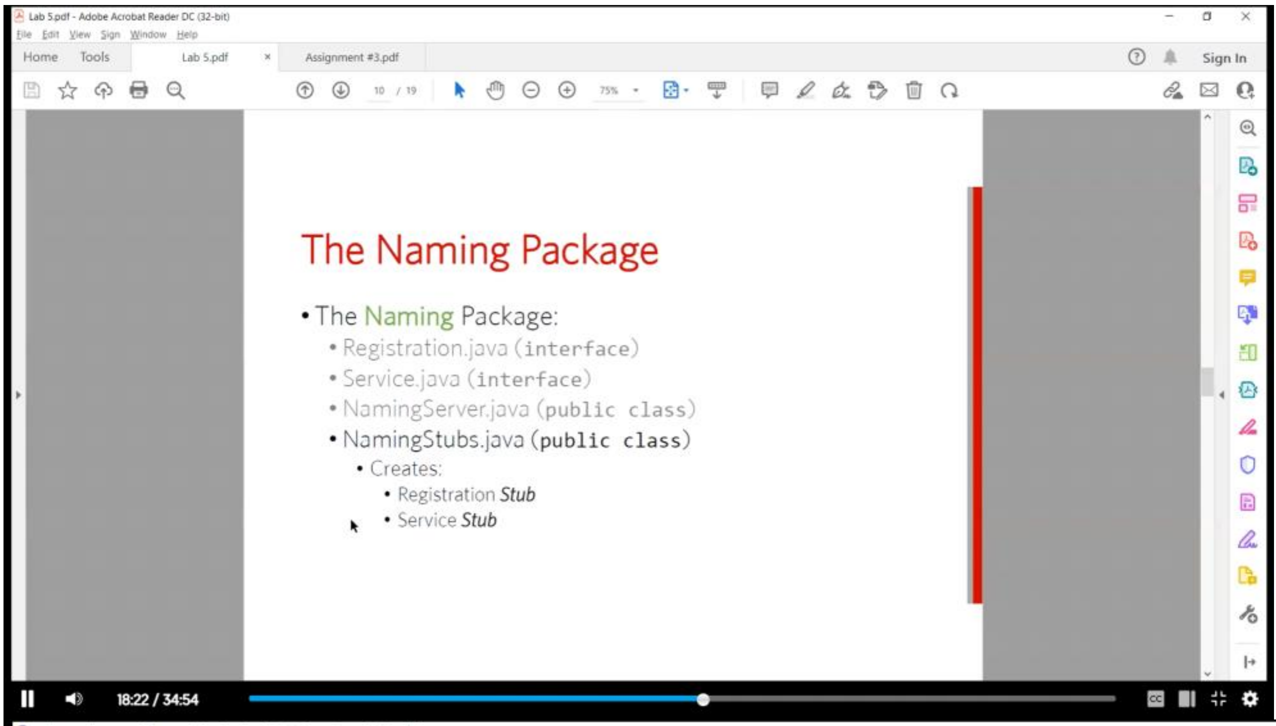
- How can we build the *Directory Tree*?
  - One way is to use Leaf/Branch approach:
    - Leaf will represent:
      - A file (name) and stub
    - Branch will represent:
      - A list of Leafs/Branches

- Node -> leaf


Naming Package: Classes

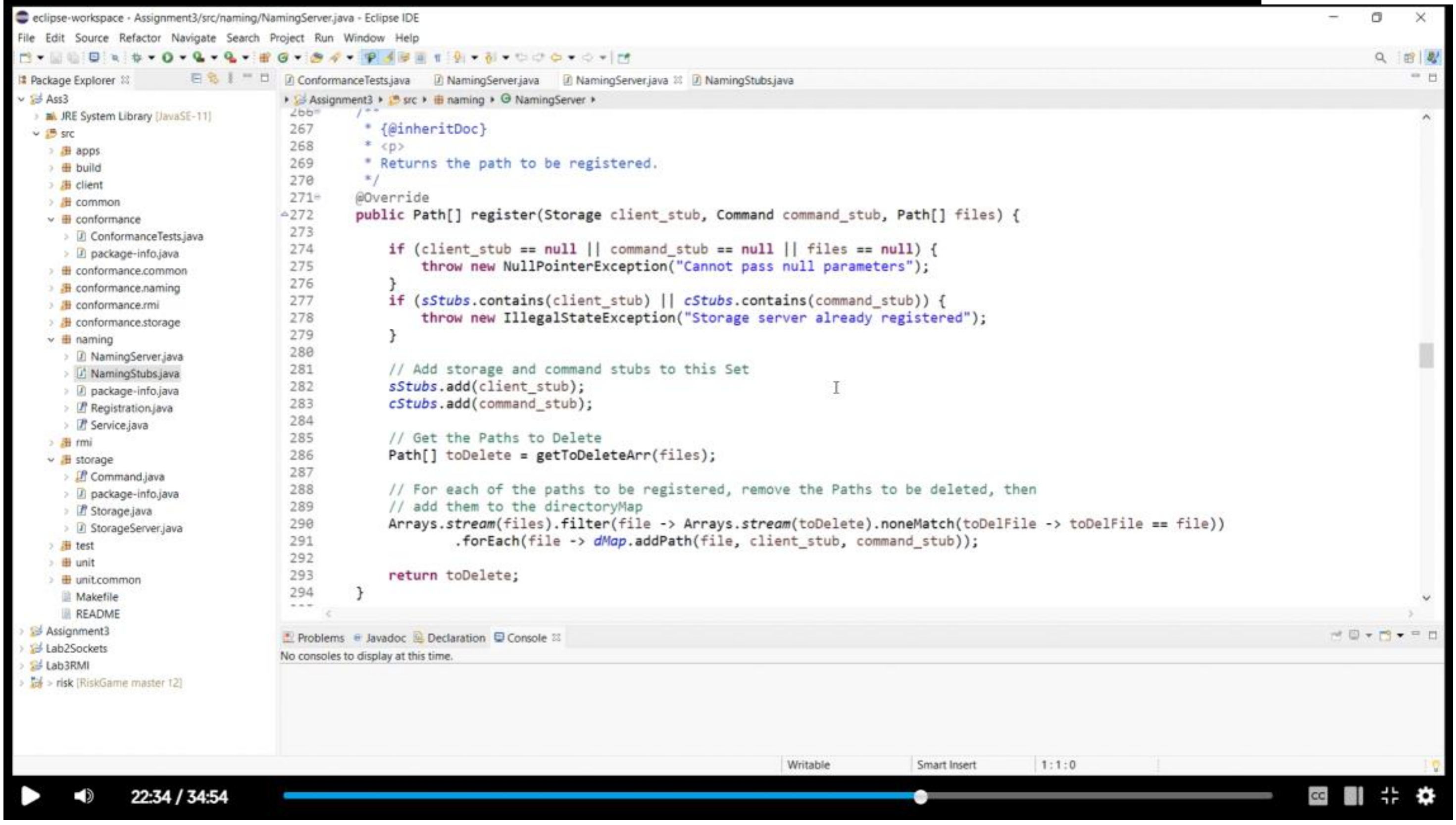
```
public class Node {
       String name;
}

public class Branch extends Node {
       ArrayList<Node> list;
}

public class Leaf extends Node {
       Command c;
       Storage s;
}
```

- Create instances of command and storage



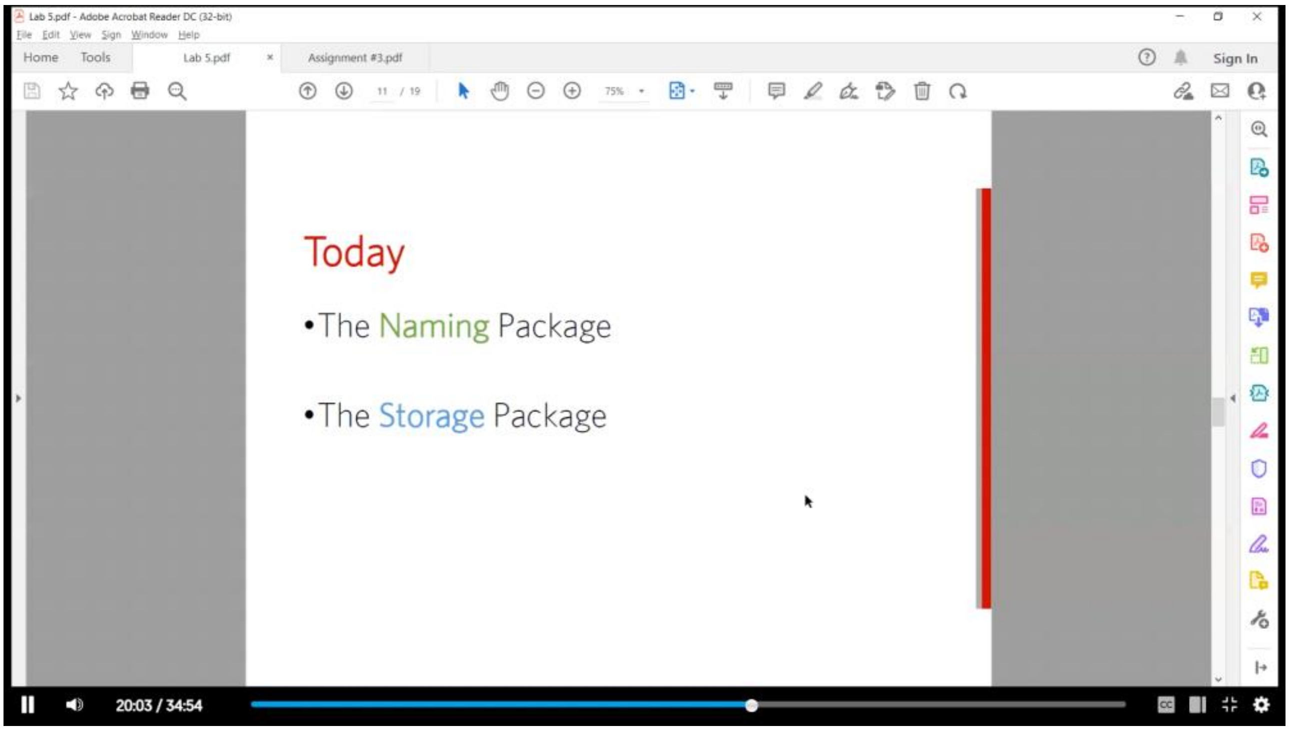
The Naming Package

- The Naming Package:
  - Registration.java (`interface`)
  - Service.java (`interface`)
  - NamingServer.java (`public class`)
  - NamingStubs.java (`public class`)
    - Creates:
      - Registration *Stub*
      - Service *Stub*



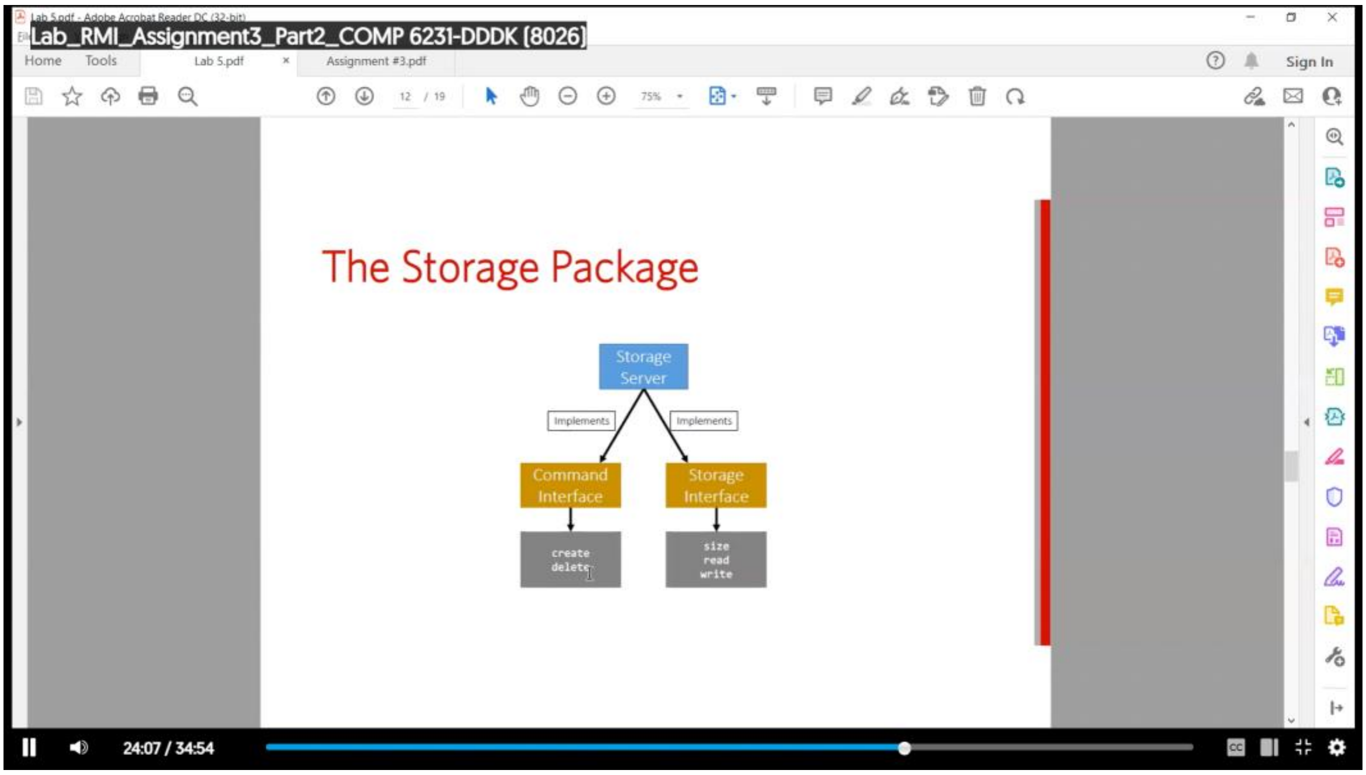
```java
    /**
     * {@inheritDoc}
     * <p>
     * Returns the path to be registered.
     */
    @Override
    public Path[] register(Storage client_stub, Command command_stub, Path[] files) {

        if (client_stub == null || command_stub == null || files == null) {
            throw new NullPointerException("Cannot pass null parameters");
        }
        if (sStubs.contains(client_stub) || cStubs.contains(command_stub)) {
            throw new IllegalStateException("Storage server already registered");
        }

        // Add storage and command stubs to this Set
        sStubs.add(client_stub);
        cStubs.add(command_stub);

        // Get the Paths to Delete
        Path[] toDelete = getToDeleteArr(files);

        // For each of the paths to be registered, remove the Paths to be deleted, then
        // add them to the directoryMap
        Arrays.stream(files).filter(file -> Arrays.stream(toDelete).noneMatch(toDelFile -> toDelFile == file))
                .forEach(file -> dMap.addPath(file, client_stub, command_stub));

        return toDelete;
    }
```

No consoles to display at this time.



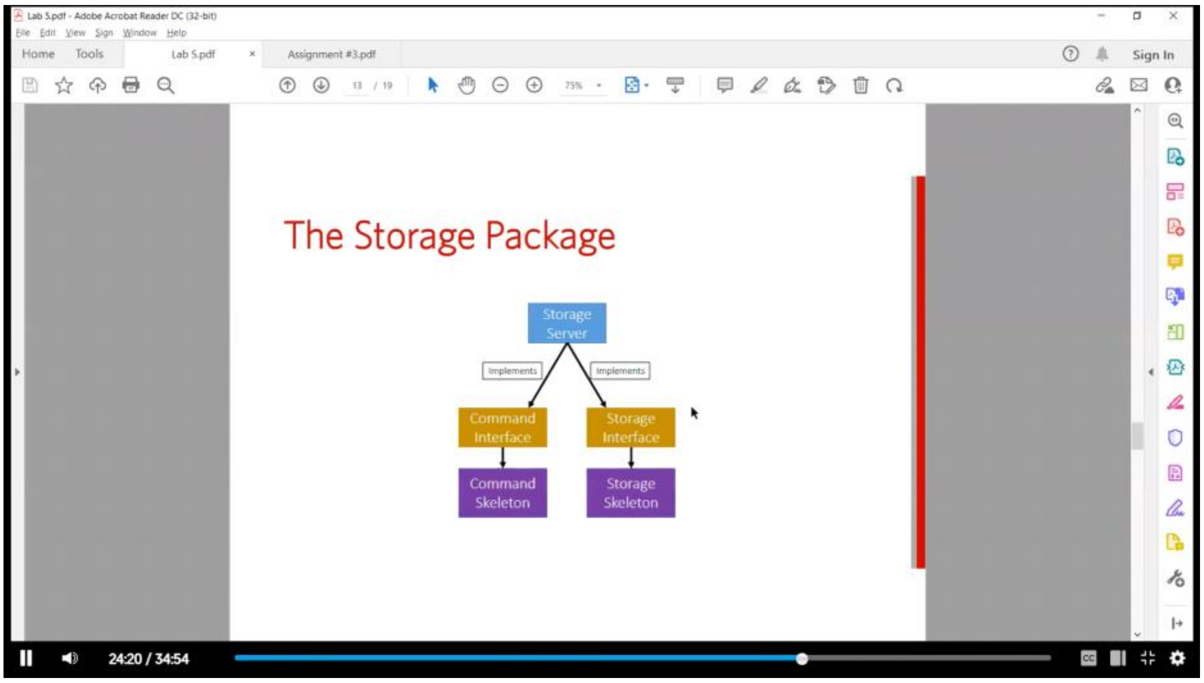
Today

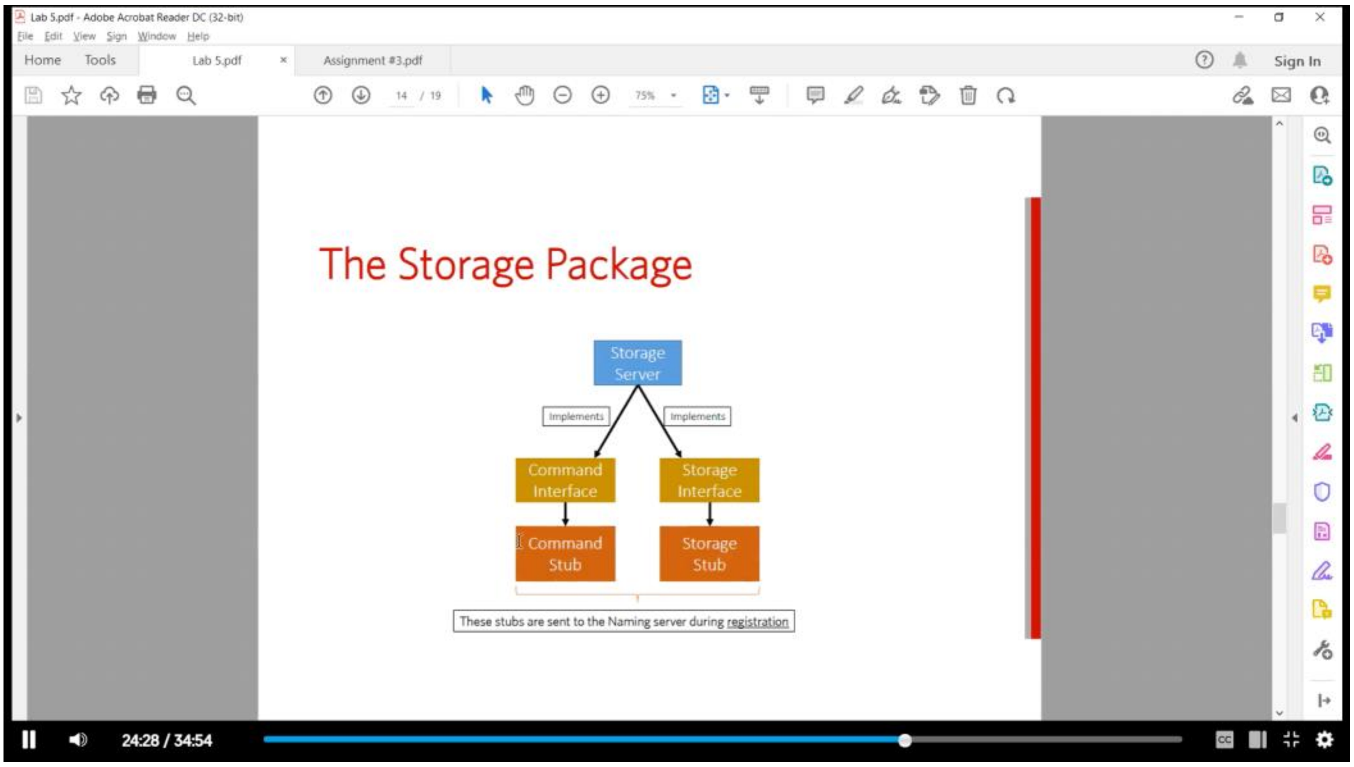
- The Naming Package
- The Storage Package



Lab_RMI_Assignment3_Part2_COMP 6231-DDDK (8026)

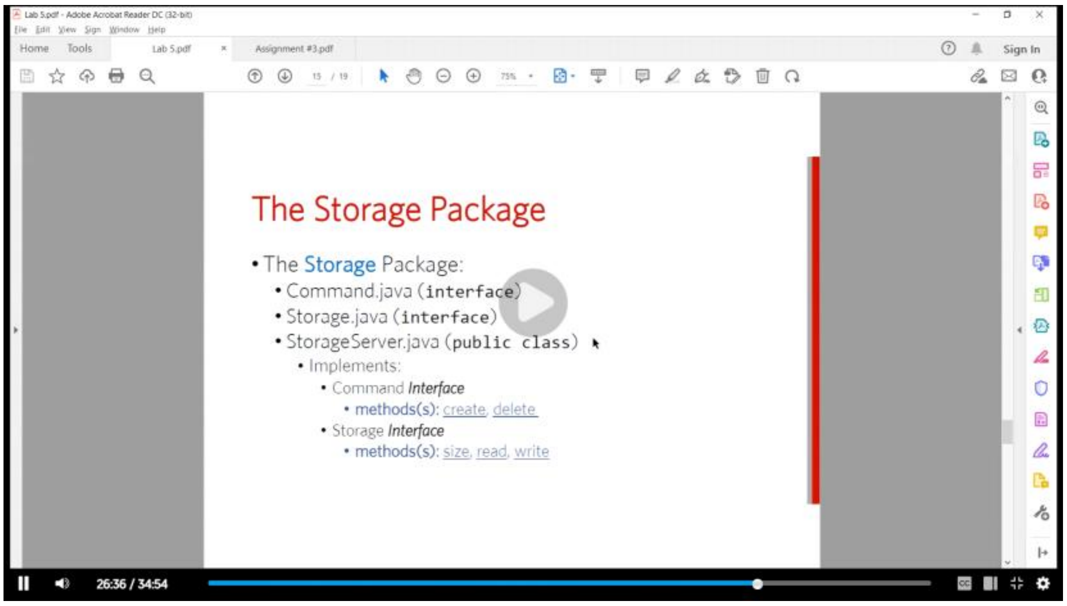
The Storage Package





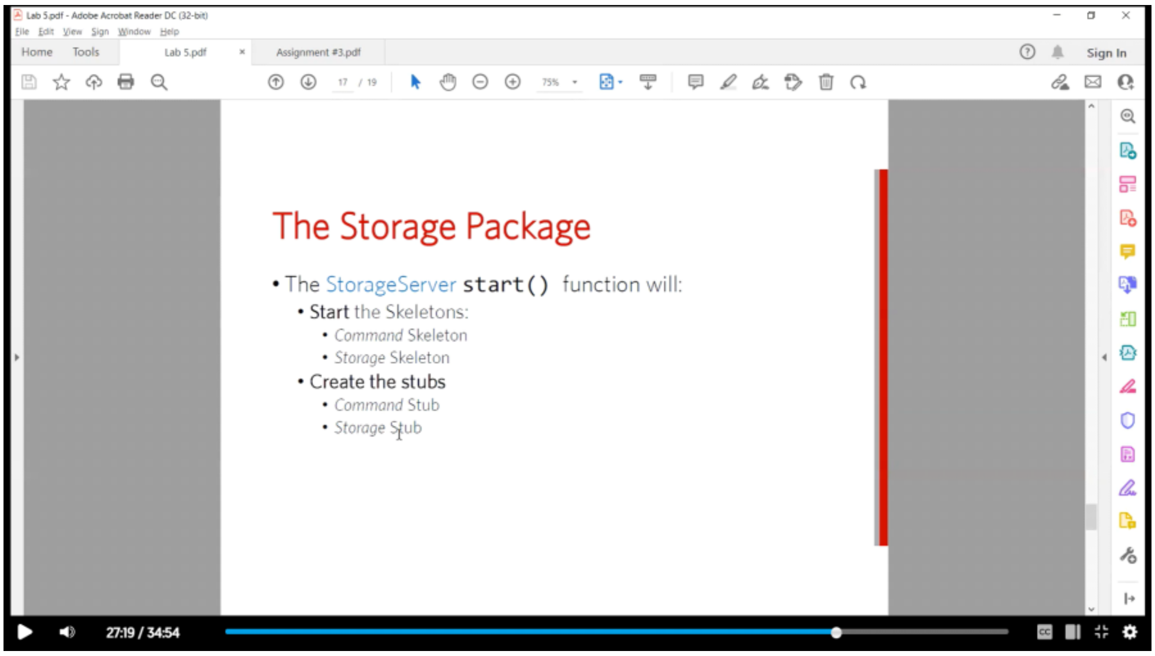
- Seek() for read and write

1. Duplicate files are returned back to the storage
2. Storage you need to delete files and empty folders



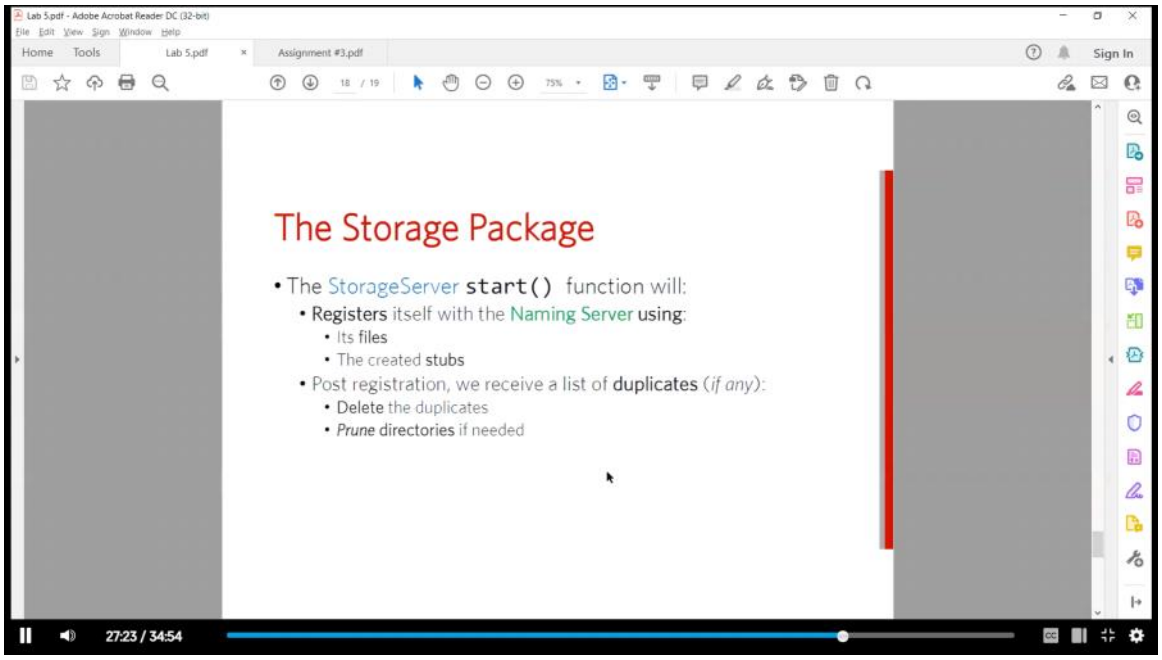
The Storage Package

- The Storage Package:
  - Command.java (`interface`)
  - Storage.java (`interface`)
  - StorageServer.java (`public class`)
    - Implements:
      - Command *Interface*
        - methods(s): create, delete
      - Storage *Interface*
        - methods(s): size, read, write



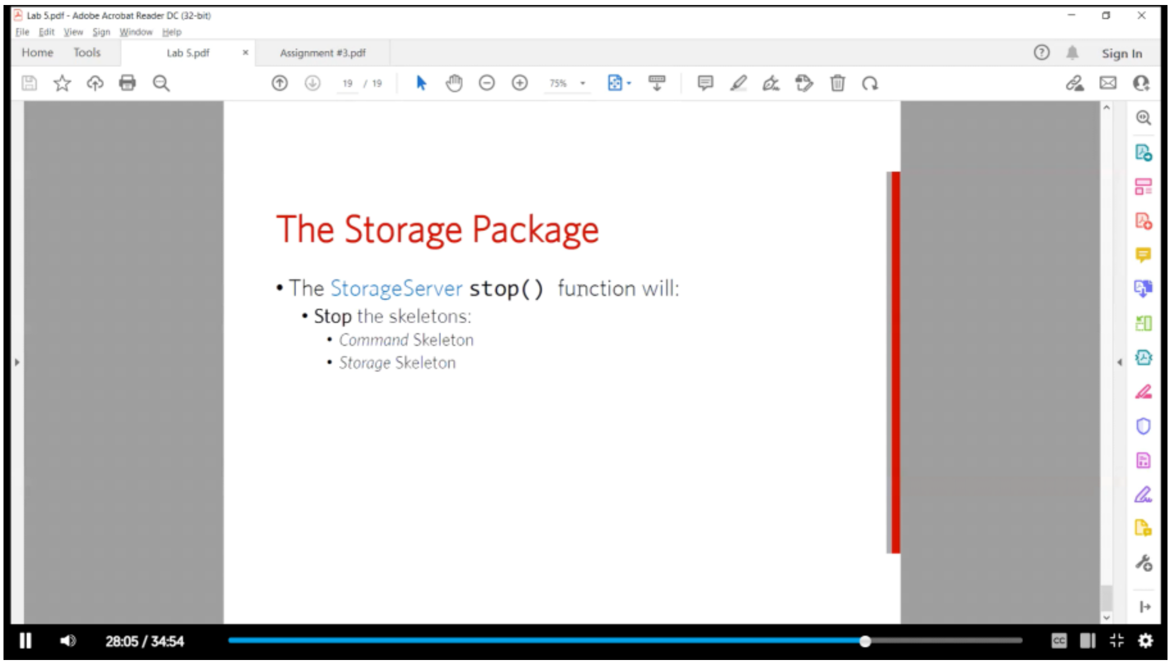
The Storage Package

- The StorageServer `start()` function will:
  - Start the Skeletons:
    - *Command* Skeleton
    - *Storage* Skeleton
  - Create the stubs
    - *Command* Stub
    - *Storage* Stub



The Storage Package

- The StorageServer `start()` function will:
  - Registers itself with the Naming Server using:
    - Its files
    - The created stubs
  - Post registration, we receive a list of duplicates (*if any*):
    - Delete the duplicates
    - *Prune* directories if needed



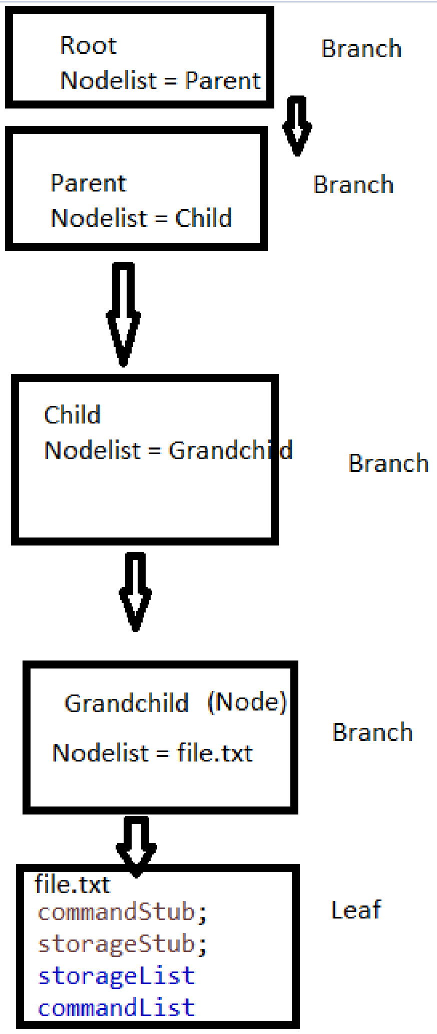
The Storage Package

- The StorageServer `stop()` function will:
  - Stop the skeletons:
    - *Command* Skeleton
    - *Storage* Skeleton

Tree - NamingServer

31 October 2021 20:00

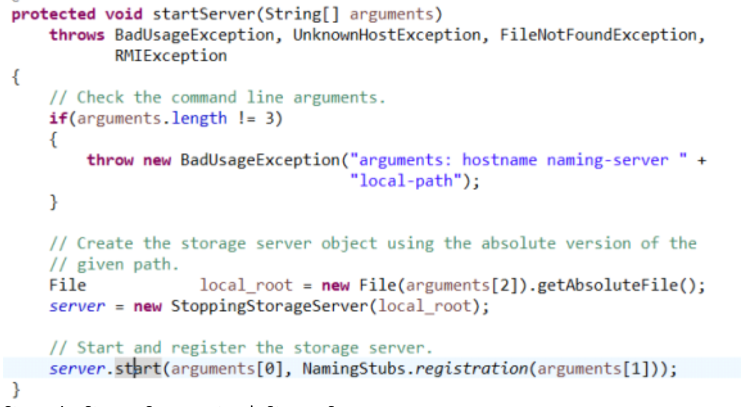
Root/Parent/Child/Grandchild/file.txt



Register Storage Server

07 November 2021 21:28

1. Launcher runs
2. Starts NamingsServerApp and Storage server app
3. Storage server app start StorageServer

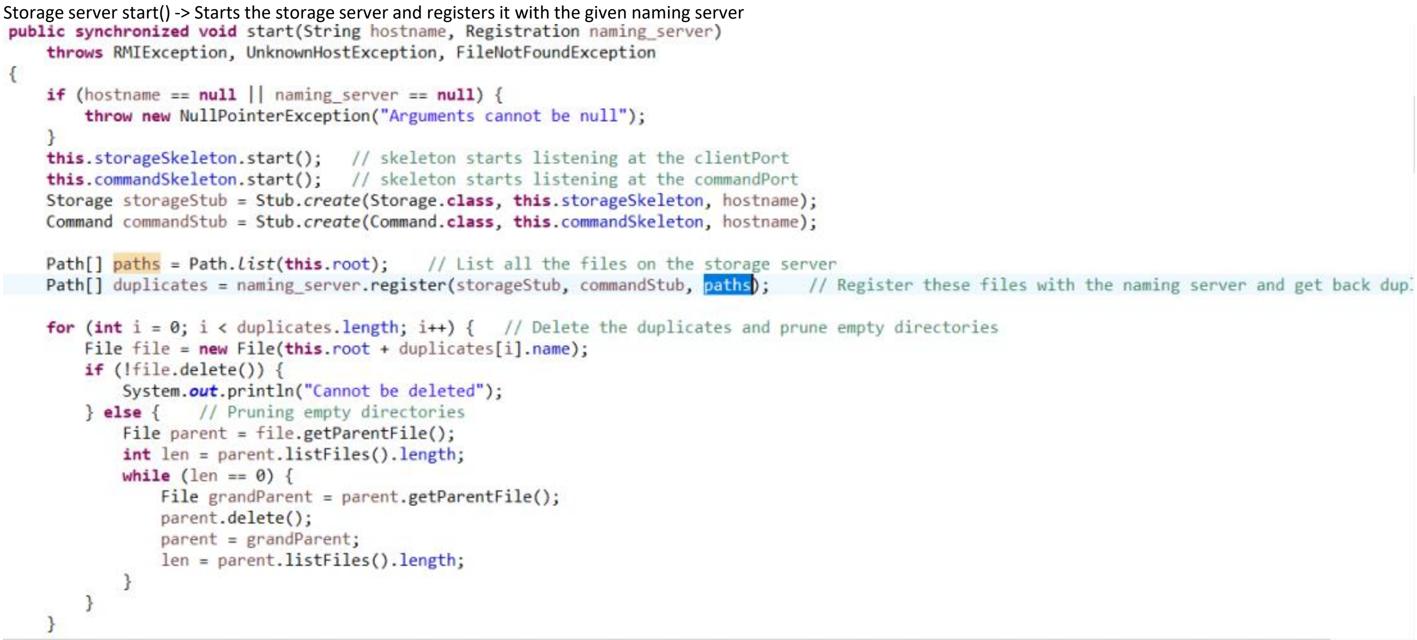


```
protected void startServer(String[] arguments)
    throws BadUsageException, UnknownHostException, FileNotFoundException,
           RMIException
{
    // Check the command line arguments.
    if(arguments.length != 3)
    {
        throw new BadUsageException("arguments: hostname naming-server " +
                                    "local-path");
    }

    // Create the storage server object using the absolute version of the
    // given path.
    File            local_root = new File(arguments[2]).getAbsoluteFile();
    server = new StoppingStorageServer(local_root);

    // Start and register the storage server.
    server.start(arguments[0], NamingStubs.registration(arguments[1]));
}
```

StopppingStorageServer extends StorageServer
Local_root -> absolute value of "storage-test"

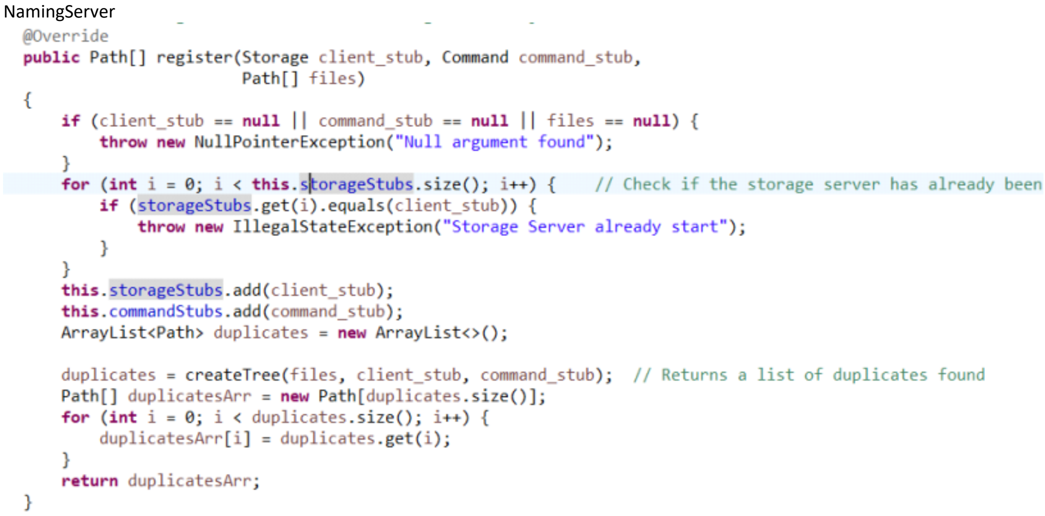
Cmd args: local hostname, the hostname of the remote naming server, and the directory that the storage server will use as its local storage for files.

4. Storage server start() -> Starts the storage server and registers it with the given naming server



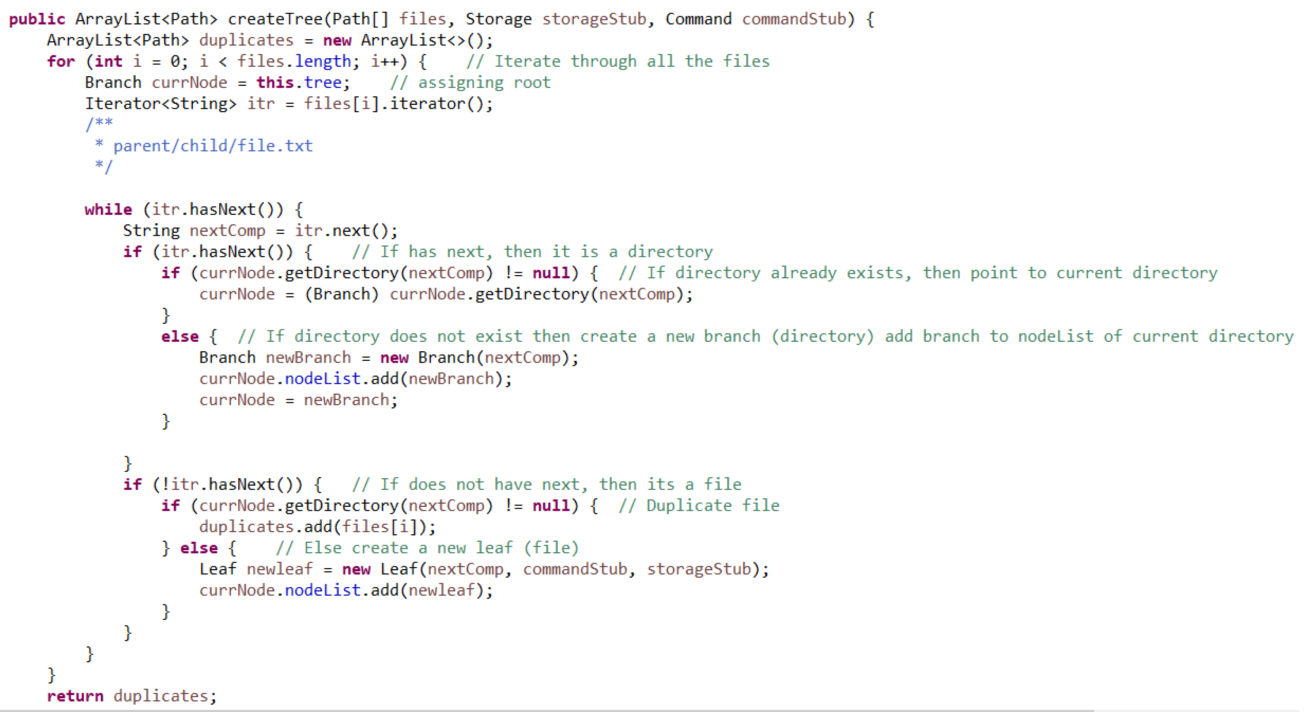
```
public synchronized void start(String hostname, Registration naming_server)
    throws RMIException, UnknownHostException, FileNotFoundException
{
    if (hostname == null || naming_server == null) {
        throw new NullPointerException("Arguments cannot be null");
    }
    this.storageSkeleton.start();   // skeleton starts listening at the clientPort
    this.commandSkeleton.start();   // skeleton starts listening at the commandPort
    Storage storageStub = Stub.create(Storage.class, this.storageSkeleton, hostname);
    Command commandStub = Stub.create(Command.class, this.commandSkeleton, hostname);

    Path[] paths = Path.list(this.root);    // List all the files on the storage server
    Path[] duplicates = naming_server.register(storageStub, commandStub, paths);    // Register these files with the naming server and get back dup

    for (int i = 0; i < duplicates.length; i++) {   // Delete the duplicates and prune empty directories
        File file = new File(this.root + duplicates[i].name);
        if (!file.delete()) {
            System.out.println("Cannot be deleted");
        } else {    // Pruning empty directories
            File parent = file.getParentFile();
            int len = parent.listFiles().length;
            while (len == 0) {
                File grandParent = parent.getParentFile();
                parent.delete();
                parent = grandParent;
                len = parent.listFiles().length;
            }
        }
    }
```

6. NamingServer

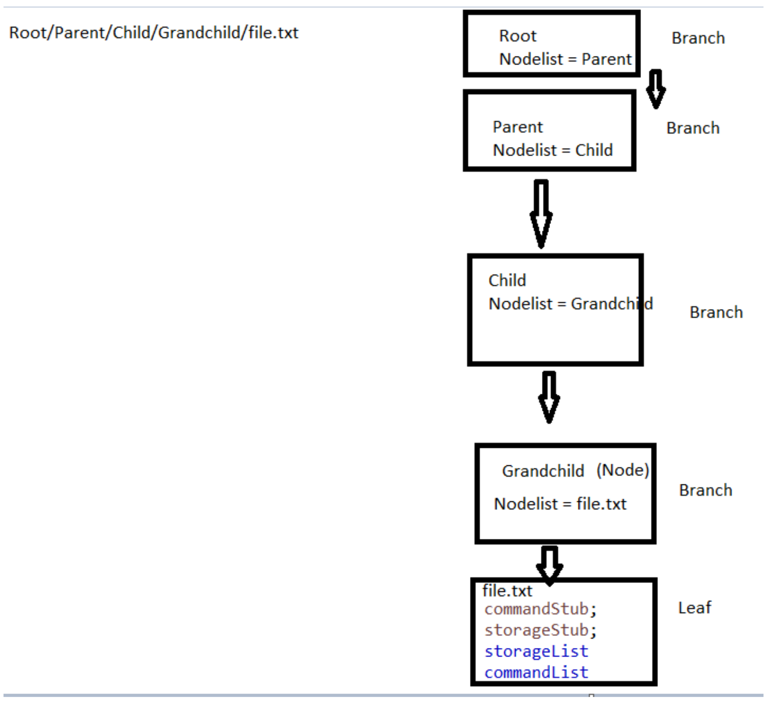


```
@Override
public Path[] register(Storage client_stub, Command command_stub,
                       Path[] files)
{
    if (client_stub == null || command_stub == null || files == null) {
        throw new NullPointerException("Null argument found");
    }
    for (int i = 0; i < this.storageStubs.size(); i++) {    // Check if the storage server has already been
        if (storageStubs.get(i).equals(client_stub)) {
            throw new IllegalStateException("Storage Server already start");
        }
    }
    this.storageStubs.add(client_stub);
    this.commandStubs.add(command_stub);
    ArrayList<Path> duplicates = new ArrayList<>();

    duplicates = createTree(files, client_stub, command_stub);  // Returns a list of duplicates found
    Path[] duplicatesArr = new Path[duplicates.size()];
    for (int i = 0; i < duplicates.size(); i++) {
        duplicatesArr[i] = duplicates.get(i);
    }
    return duplicatesArr;
}
```

7.



```
public ArrayList<Path> createTree(Path[] files, Storage storageStub, Command commandStub) {
    ArrayList<Path> duplicates = new ArrayList<>();
    for (int i = 0; i < files.length; i++) {    // Iterate through all the files
        Branch currNode = this.tree;    // assigning root
        Iterator<String> itr = files[i].iterator();
        /**
         * parent/child/file.txt
         */

        while (itr.hasNext()) {
            String nextComp = itr.next();
            if (itr.hasNext()) {    // If has next, then it is a directory
                if (currNode.getDirectory(nextComp) != null) {  // If directory already exists, then point to current directory
                    currNode = (Branch) currNode.getDirectory(nextComp);
                }
                else {  // If directory does not exist then create a new branch (directory) add branch to nodeList of current directory
                    Branch newBranch = new Branch(nextComp);
                    currNode.nodeList.add(newBranch);
                    currNode = newBranch;
                }

            }
            if (!itr.hasNext()) {   // If does not have next, then its a file
                if (currNode.getDirectory(nextComp) != null) {  // Duplicate file
                    duplicates.add(files[i]);
                } else {    // Else create a new leaf (file)
                    Leaf newleaf = new Leaf(nextComp, commandStub, storageStub);
                    currNode.nodeList.add(newleaf);
                }
            }
        }
    }
    return duplicates;
```

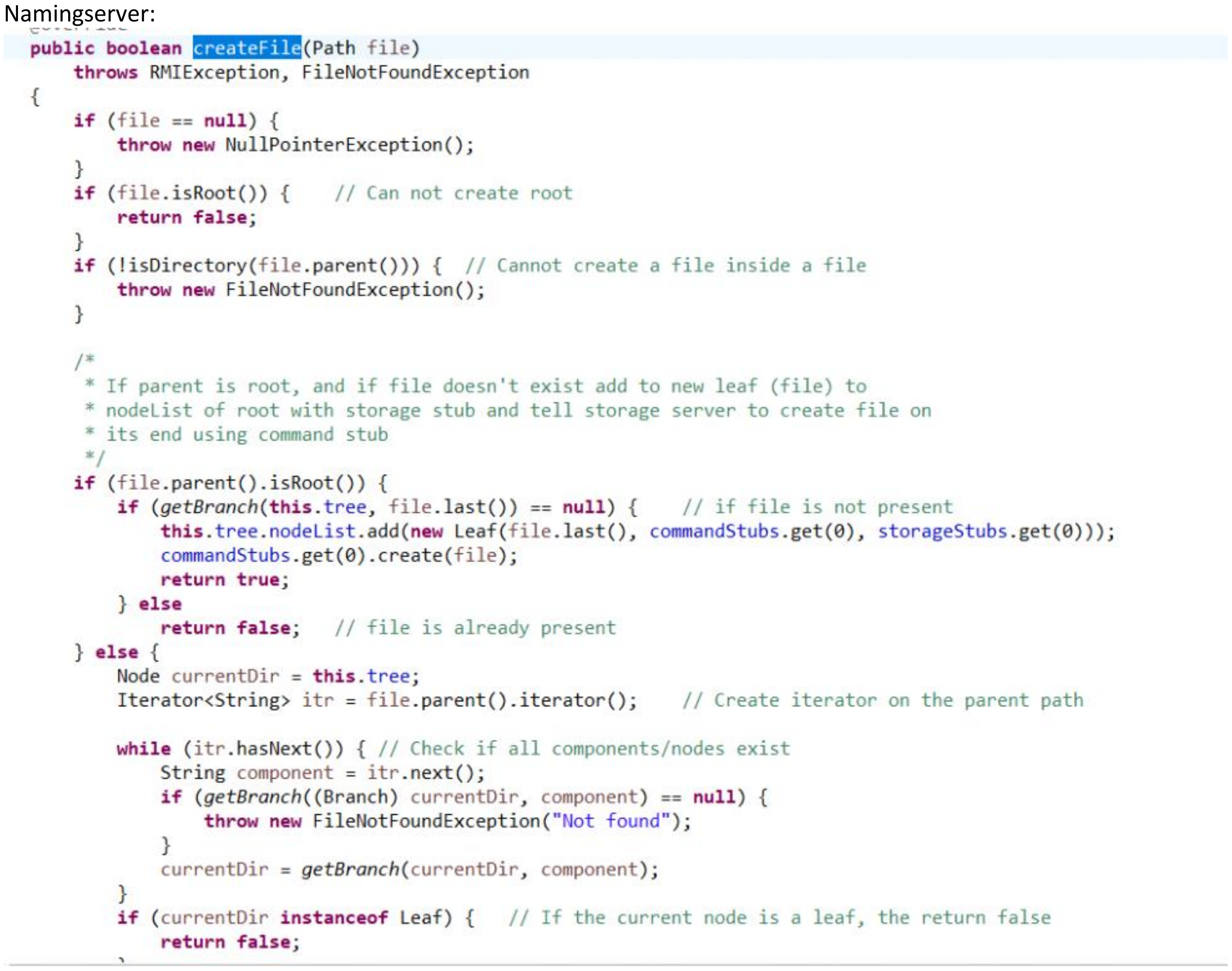


CreateFile(path): create the file referred to by path1.

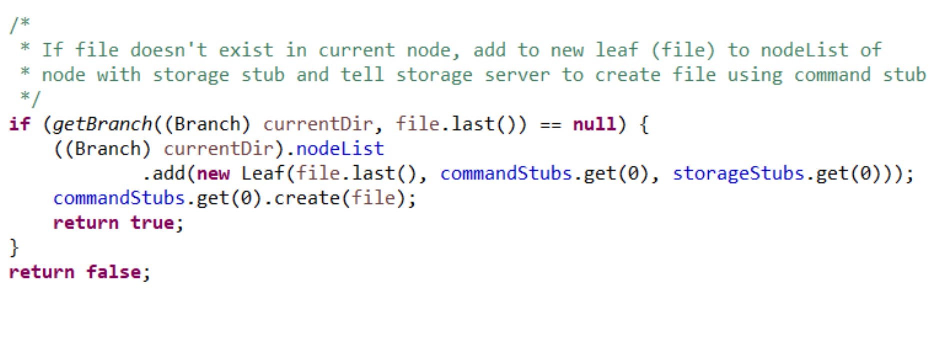
07 November 2021 21:23

The operations (or functionalities) that are available to the Clients of DFS are:

1. Namingserver:

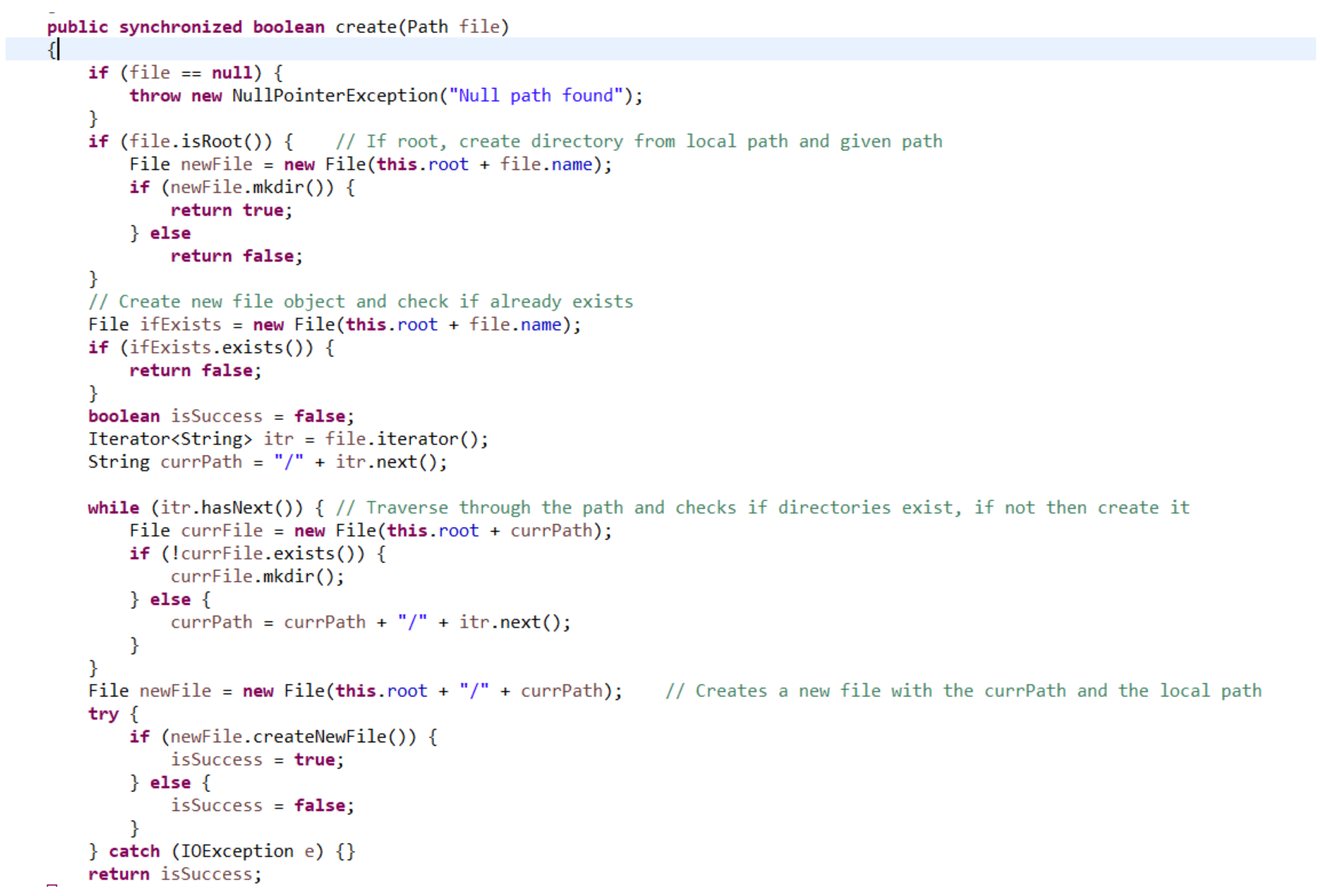


```java
public boolean createFile(Path file)
    throws RMIException, FileNotFoundException
{
    if (file == null) {
        throw new NullPointerException();
    }
    if (file.isRoot()) {    // Can not create root
        return false;
    }
    if (!isDirectory(file.parent())) {  // Cannot create a file inside a file
        throw new FileNotFoundException();
    }

    /*
     * If parent is root, and if file doesn't exist add to new leaf (file) to
     * nodeList of root with storage stub and tell storage server to create file on
     * its end using command stub
     */
    if (file.parent().isRoot()) {
        if (getBranch(this.tree, file.last()) == null) {    // if file is not present
            this.tree.nodeList.add(new Leaf(file.last(), commandStubs.get(0), storageStubs.get(0)));
            commandStubs.get(0).create(file);
            return true;
        } else
            return false;    // file is already present
    } else {
        Node currentDir = this.tree;
        Iterator<String> itr = file.parent().iterator();    // Create iterator on the parent path

        while (itr.hasNext()) { // Check if all components/nodes exist
            String component = itr.next();
            if (getBranch((Branch) currentDir, component) == null) {
                throw new FileNotFoundException("Not found");
            }
            currentDir = getBranch(currentDir, component);
        }
        if (currentDir instanceof Leaf) {   // If the current node is a leaf, the return false
            return false;
```



```java
        /*
         * If file doesn't exist in current node, add to new leaf (file) to nodeList of
         * node with storage stub and tell storage server to create file using command stub
         */
        if (getBranch((Branch) currentDir, file.last()) == null) {
            ((Branch) currentDir).nodeList
                    .add(new Leaf(file.last(), commandStubs.get(0), storageStubs.get(0)));
            commandStubs.get(0).create(file);
            return true;
        }
        return false;
    }
}
```

2. Create happens in storageserver:



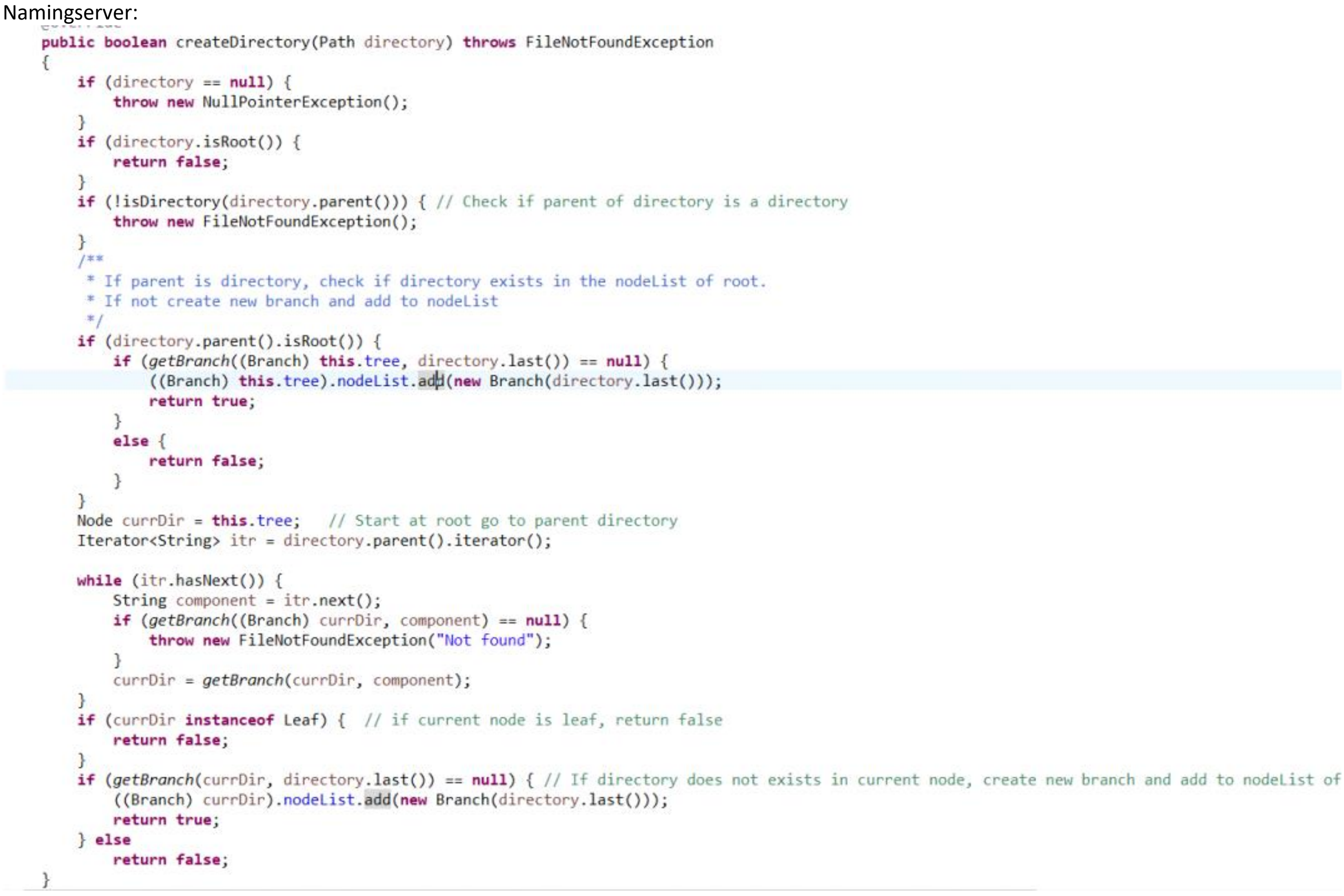
```java
public synchronized boolean create(Path file)
{
    if (file == null) {
        throw new NullPointerException("Null path found");
    }
    if (file.isRoot()) {    // If root, create directory from local path and given path
        File newFile = new File(this.root + file.name);
        if (newFile.mkdir()) {
            return true;
        } else
            return false;
    }
    // Create new file object and check if already exists
    File ifExists = new File(this.root + file.name);
    if (ifExists.exists()) {
        return false;
    }
    boolean isSuccess = false;
    Iterator<String> itr = file.iterator();
    String currPath = "/" + itr.next();

    while (itr.hasNext()) { // Traverse through the path and checks if directories exist, if not then create it
        File currFile = new File(this.root + currPath);
        if (!currFile.exists()) {
            currFile.mkdir();
        } else {
            currPath = currPath + "/" + itr.next();
        }
    }
    File newFile = new File(this.root + "/" + currPath);    // Creates a new file with the currPath and the local path
    try {
        if (newFile.createNewFile()) {
            isSuccess = true;
        } else {
            isSuccess = false;
        }
    } catch (IOException e) {}
    return isSuccess;
```

CreateDirectory(path) : create the directory referred to by path.

07 November 2021 21:25

The operations (or functionalities) that are available to the Clients of DFS are:

1. Namingserver:

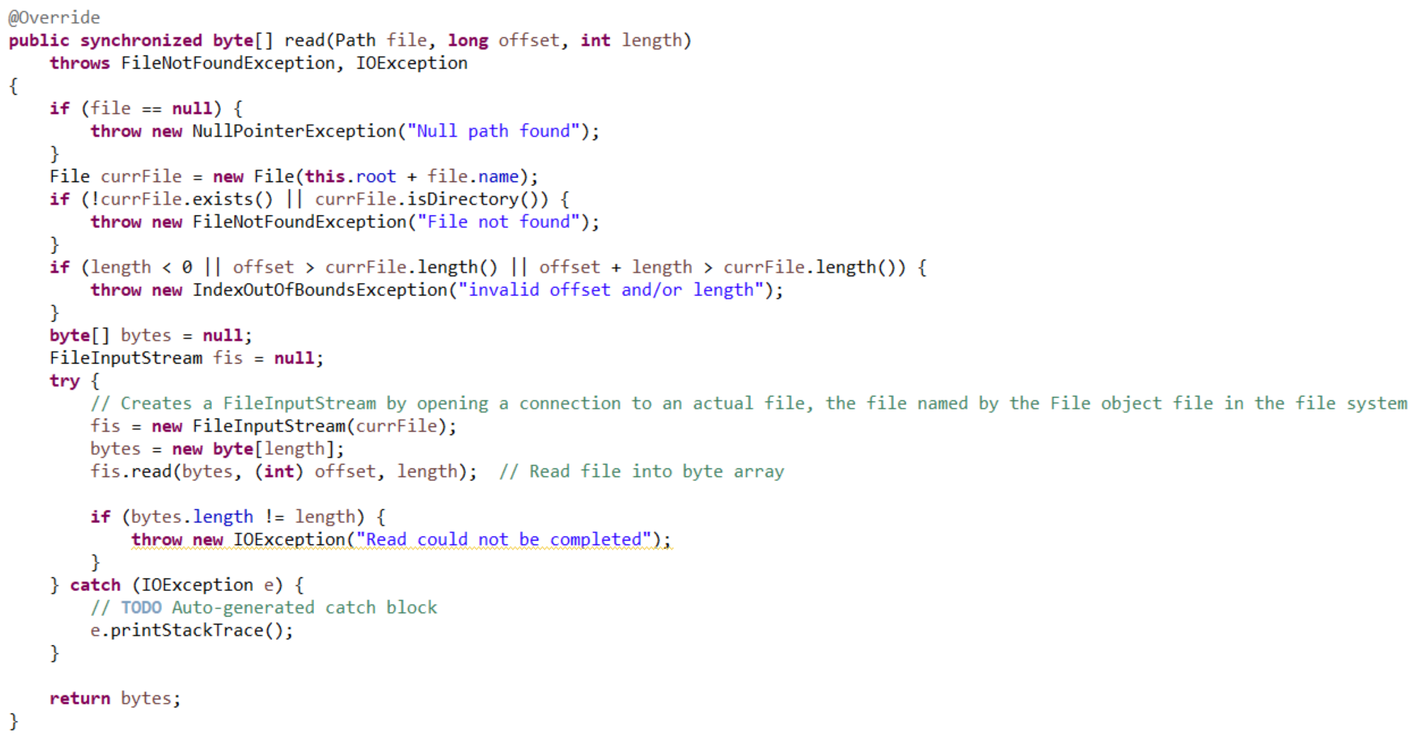


```java
public boolean createDirectory(Path directory) throws FileNotFoundException
{
    if (directory == null) {
        throw new NullPointerException();
    }
    if (directory.isRoot()) {
        return false;
    }
    if (!isDirectory(directory.parent())) { // Check if parent of directory is a directory
        throw new FileNotFoundException();
    }
    /**
     * If parent is directory, check if directory exists in the nodeList of root.
     * If not create new branch and add to nodeList
     */
    if (directory.parent().isRoot()) {
        if (getBranch((Branch) this.tree, directory.last()) == null) {
            ((Branch) this.tree).nodeList.add(new Branch(directory.last()));
            return true;
        }
        else {
            return false;
        }
    }
    Node currDir = this.tree;   // Start at root go to parent directory
    Iterator<String> itr = directory.parent().iterator();

    while (itr.hasNext()) {
        String component = itr.next();
        if (getBranch((Branch) currDir, component) == null) {
            throw new FileNotFoundException("Not found");
        }
        currDir = getBranch(currDir, component);
    }
    if (currDir instanceof Leaf) {  // if current node is leaf, return false
        return false;
    }
    if (getBranch(currDir, directory.last()) == null) { // If directory does not exists in current node, create new branch and add to nodeList of
        ((Branch) currDir).nodeList.add(new Branch(directory.last()));
        return true;
    } else
        return false;
}
```

Read(path, o, n)

07 November 2021 21:25

Read(path, o, n) : read n bytes of data from the file referred to by path starting at an offset o.

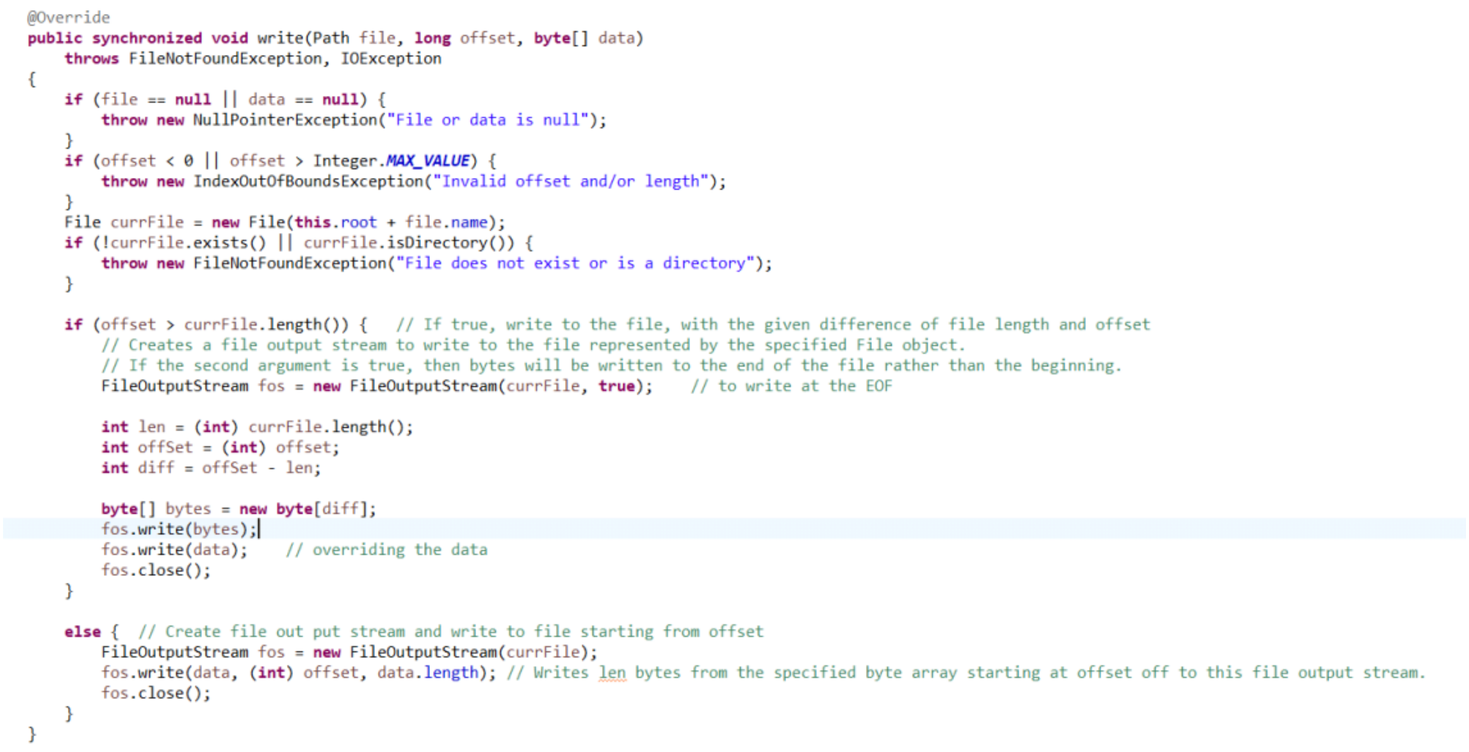


```java
@Override
public synchronized byte[] read(Path file, long offset, int length)
    throws FileNotFoundException, IOException
{
    if (file == null) {
        throw new NullPointerException("Null path found");
    }
    File currFile = new File(this.root + file.name);
    if (!currFile.exists() || currFile.isDirectory()) {
        throw new FileNotFoundException("File not found");
    }
    if (length < 0 || offset > currFile.length() || offset + length > currFile.length()) {
        throw new IndexOutOfBoundsException("invalid offset and/or length");
    }
    byte[] bytes = null;
    FileInputStream fis = null;
    try {
        // Creates a FileInputStream by opening a connection to an actual file, the file named by the File object file in the file system
        fis = new FileInputStream(currFile);
        bytes = new byte[length];
        fis.read(bytes, (int) offset, length);  // Read file into byte array

        if (bytes.length != length) {
            throw new IOException("Read could not be completed");
        }
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    return bytes;
}
```

Write(path, o, data) :

07 November 2021 21:26

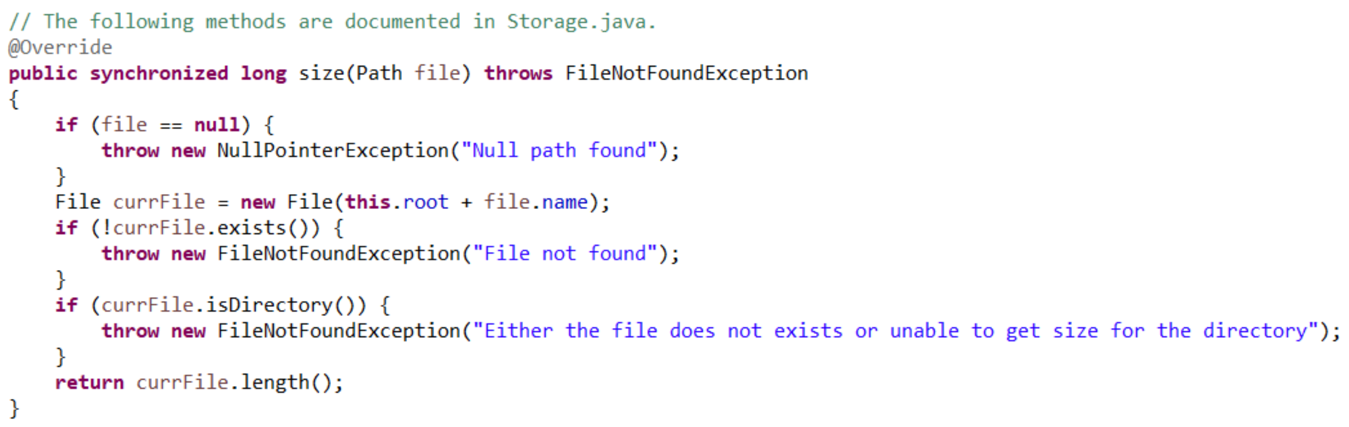
write n bytes of data to the file referred to by path starting at an
offset o.



```java
@Override
public synchronized void write(Path file, long offset, byte[] data)
    throws FileNotFoundException, IOException
{
    if (file == null || data == null) {
        throw new NullPointerException("File or data is null");
    }
    if (offset < 0 || offset > Integer.MAX_VALUE) {
        throw new IndexOutOfBoundsException("Invalid offset and/or length");
    }
    File currFile = new File(this.root + file.name);
    if (!currFile.exists() || currFile.isDirectory()) {
        throw new FileNotFoundException("File does not exist or is a directory");
    }

    if (offset > currFile.length()) {   // If true, write to the file, with the given difference of file length and offset
        // Creates a file output stream to write to the file represented by the specified File object.
        // If the second argument is true, then bytes will be written to the end of the file rather than the beginning.
        FileOutputStream fos = new FileOutputStream(currFile, true);    // to write at the EOF

        int len = (int) currFile.length();
        int offSet = (int) offset;
        int diff = offSet - len;

        byte[] bytes = new byte[diff];
        fos.write(bytes);
        fos.write(data);    // overriding the data
        fos.close();
    }

    else {  // Create file out put stream and write to file starting from offset
        FileOutputStream fos = new FileOutputStream(currFile);
        fos.write(data, (int) offset, data.length); // Writes len bytes from the specified byte array starting at offset off to this file output stream.
        fos.close();
    }
}
```

Size(path): return the size, in bytes, of the file referred to by path.

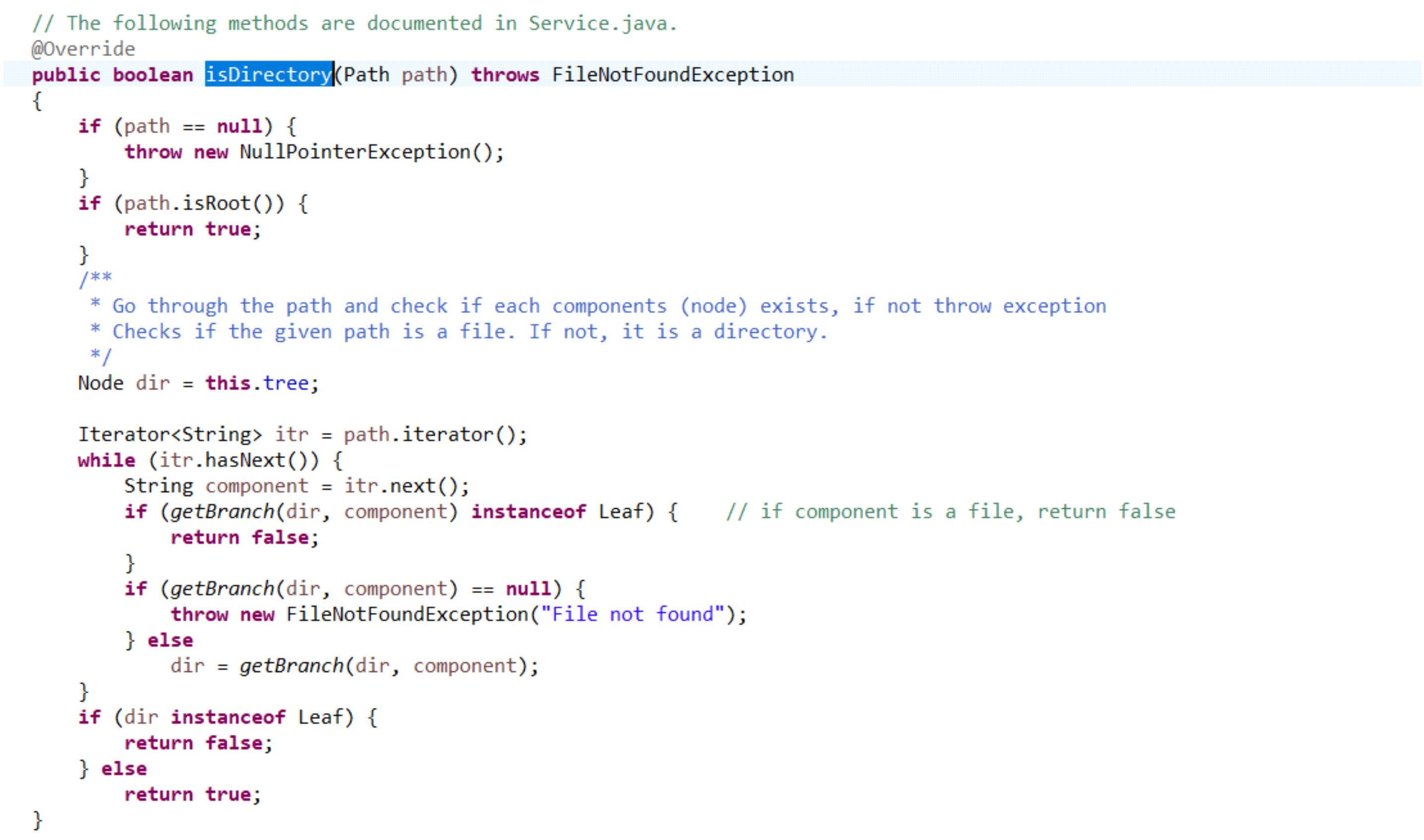
07 November 2021 21:27



```java
// The following methods are documented in Storage.java.
@Override
public synchronized long size(Path file) throws FileNotFoundException
{
    if (file == null) {
        throw new NullPointerException("Null path found");
    }
    File currFile = new File(this.root + file.name);
    if (!currFile.exists()) {
        throw new FileNotFoundException("File not found");
    }
    if (currFile.isDirectory()) {
        throw new FileNotFoundException("Either the file does not exists or unable to get size for the directory");
    }
    return currFile.length();
}
```

IsDirectory(path) : return true if path refers to a directory.

07 November 2021 21:27

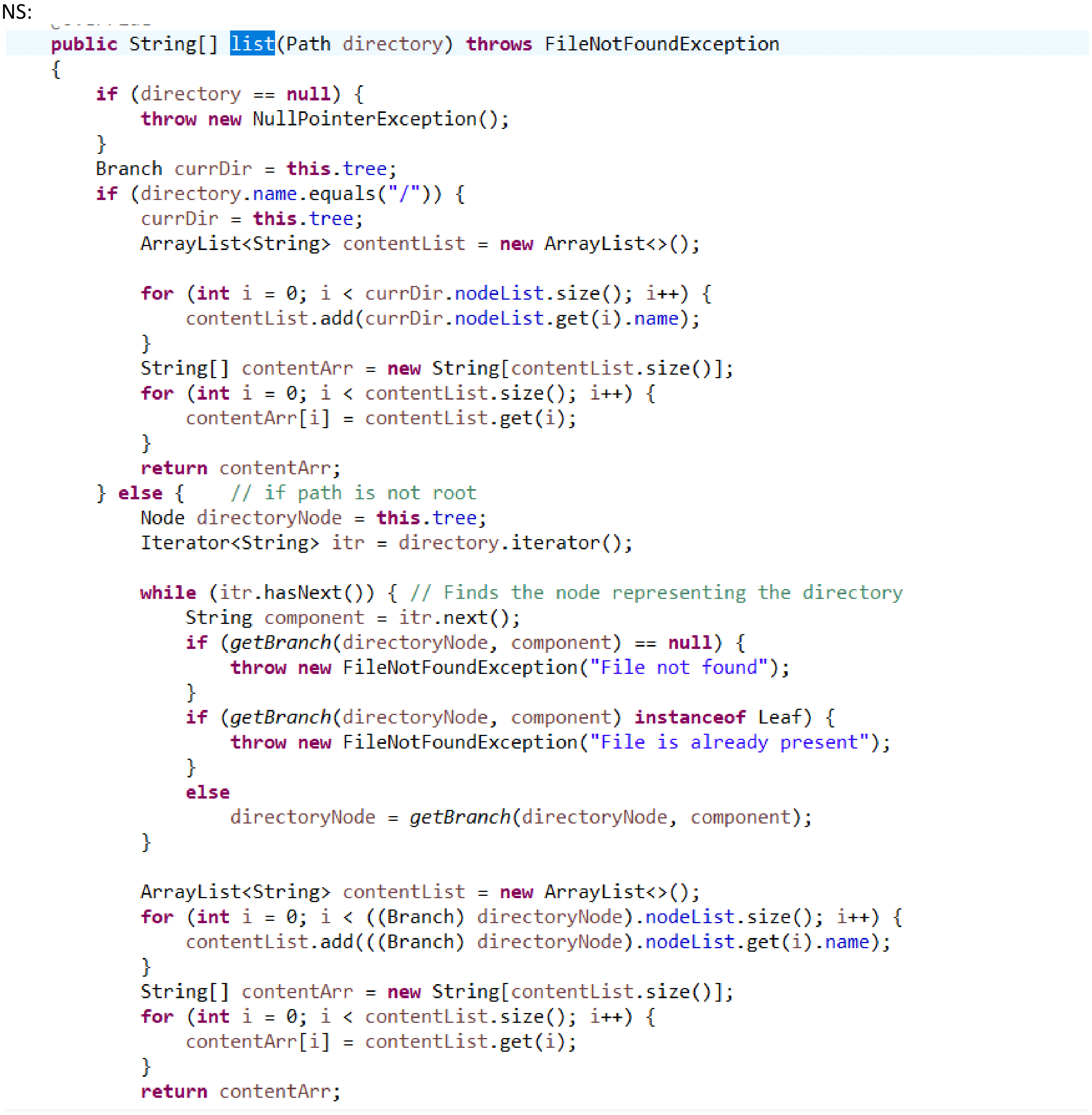


```java
// The following methods are documented in Service.java.
@Override
public boolean isDirectory(Path path) throws FileNotFoundException
{
    if (path == null) {
        throw new NullPointerException();
    }
    if (path.isRoot()) {
        return true;
    }
    /**
     * Go through the path and check if each components (node) exists, if not throw exception
     * Checks if the given path is a file. If not, it is a directory.
     */
    Node dir = this.tree;

    Iterator<String> itr = path.iterator();
    while (itr.hasNext()) {
        String component = itr.next();
        if (getBranch(dir, component) instanceof Leaf) {    // if component is a file, return false
            return false;
        }
        if (getBranch(dir, component) == null) {
            throw new FileNotFoundException("File not found");
        } else
            dir = getBranch(dir, component);
    }
    if (dir instanceof Leaf) {
        return false;
    } else
        return true;
}
```

List(path): list the contents of the directory referred to by path.

07 November 2021 21:27

NS:

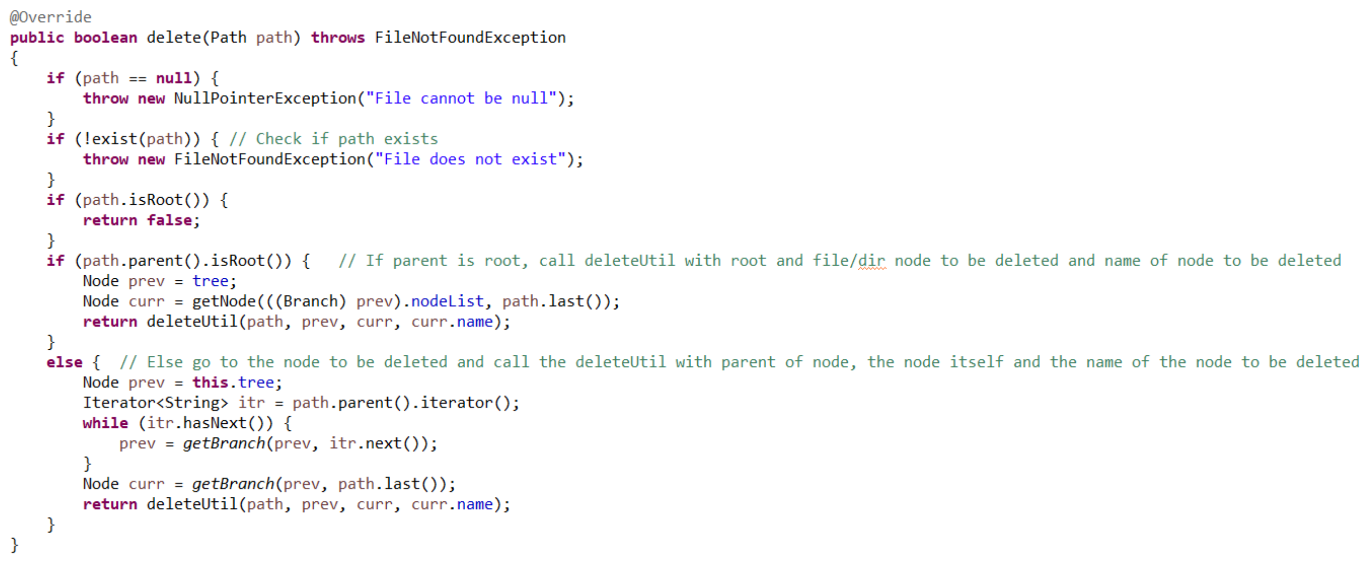


```java
    public String[] list(Path directory) throws FileNotFoundException
    {
        if (directory == null) {
            throw new NullPointerException();
        }
        Branch currDir = this.tree;
        if (directory.name.equals("/")) {
            currDir = this.tree;
            ArrayList<String> contentList = new ArrayList<>();

            for (int i = 0; i < currDir.nodeList.size(); i++) {
                contentList.add(currDir.nodeList.get(i).name);
            }
            String[] contentArr = new String[contentList.size()];
            for (int i = 0; i < contentList.size(); i++) {
                contentArr[i] = contentList.get(i);
            }
            return contentArr;
        } else {    // if path is not root
            Node directoryNode = this.tree;
            Iterator<String> itr = directory.iterator();

            while (itr.hasNext()) { // Finds the node representing the directory
                String component = itr.next();
                if (getBranch(directoryNode, component) == null) {
                    throw new FileNotFoundException("File not found");
                }
                if (getBranch(directoryNode, component) instanceof Leaf) {
                    throw new FileNotFoundException("File is already present");
                }
                else
                    directoryNode = getBranch(directoryNode, component);
            }

            ArrayList<String> contentList = new ArrayList<>();
            for (int i = 0; i < ((Branch) directoryNode).nodeList.size(); i++) {
                contentList.add(((Branch) directoryNode).nodeList.get(i).name);
            }
            String[] contentArr = new String[contentList.size()];
            for (int i = 0; i < contentList.size(); i++) {
                contentArr[i] = contentList.get(i);
            }
            return contentArr;
```

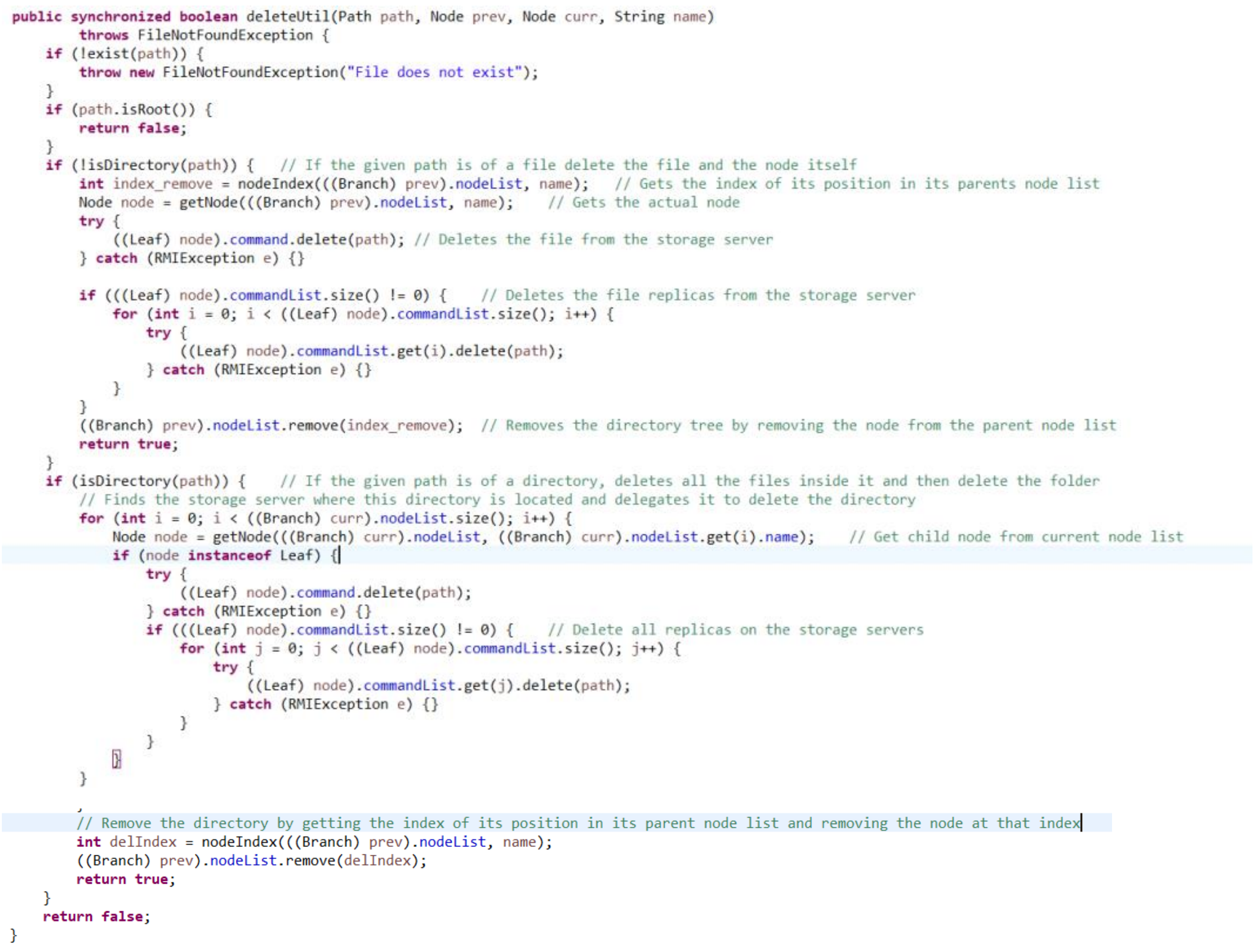
Delete(path): delete the file or directory referred to by path.

07 November 2021 21:27

1. Naming server:



```java
@Override
public boolean delete(Path path) throws FileNotFoundException
{
    if (path == null) {
        throw new NullPointerException("File cannot be null");
    }
    if (!exist(path)) { // Check if path exists
        throw new FileNotFoundException("File does not exist");
    }
    if (path.isRoot()) {
        return false;
    }
    if (path.parent().isRoot()) {   // If parent is root, call deleteUtil with root and file/dir node to be deleted and name of node to be deleted
        Node prev = tree;
        Node curr = getNode(((Branch) prev).nodeList, path.last());
        return deleteUtil(path, prev, curr, curr.name);
    }
    else {  // Else go to the node to be deleted and call the deleteUtil with parent of node, the node itself and the name of the node to be deleted
        Node prev = this.tree;
        Iterator<String> itr = path.parent().iterator();
        while (itr.hasNext()) {
            prev = getBranch(prev, itr.next());
        }
        Node curr = getBranch(prev, path.last());
        return deleteUtil(path, prev, curr, curr.name);
    }
}

/**
```



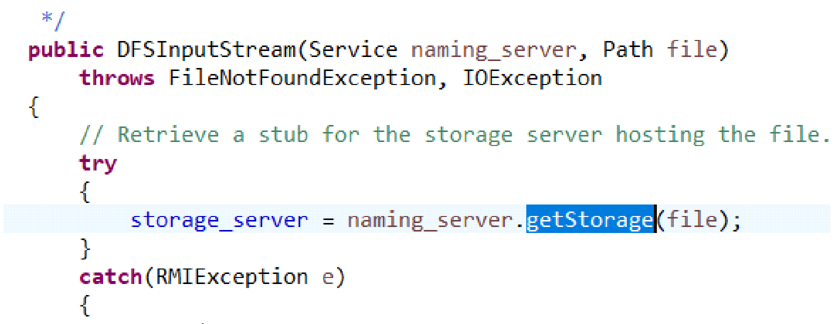
```java
public synchronized boolean deleteUtil(Path path, Node prev, Node curr, String name)
        throws FileNotFoundException {
    if (!exist(path)) {
        throw new FileNotFoundException("File does not exist");
    }
    if (path.isRoot()) {
        return false;
    }
    if (!isDirectory(path)) {   // If the given path is of a file delete the file and the node itself
        int index_remove = nodeIndex(((Branch) prev).nodeList, name);   // Gets the index of its position in its parents node list
        Node node = getNode(((Branch) prev).nodeList, name);    // Gets the actual node
        try {
            ((Leaf) node).command.delete(path); // Deletes the file from the storage server
        } catch (RMIException e) {}

        if (((Leaf) node).commandList.size() != 0) {    // Deletes the file replicas from the storage server
            for (int i = 0; i < ((Leaf) node).commandList.size(); i++) {
                try {
                    ((Leaf) node).commandList.get(i).delete(path);
                } catch (RMIException e) {}
            }
        }
        ((Branch) prev).nodeList.remove(index_remove);  // Removes the directory tree by removing the node from the parent node list
        return true;
    }
    if (isDirectory(path)) {    // If the given path is of a directory, deletes all the files inside it and then delete the folder
        // Finds the storage server where this directory is located and delegates it to delete the directory
        for (int i = 0; i < ((Branch) curr).nodeList.size(); i++) {
            Node node = getNode(((Branch) curr).nodeList, ((Branch) curr).nodeList.get(i).name);    // Get child node from current node list
            if (node instanceof Leaf) {
                try {
                    ((Leaf) node).command.delete(path);
                } catch (RMIException e) {}
                if (((Leaf) node).commandList.size() != 0) {    // Delete all replicas on the storage servers
                    for (int j = 0; j < ((Leaf) node).commandList.size(); j++) {
                        try {
                            ((Leaf) node).commandList.get(j).delete(path);
                        } catch (RMIException e) {}
                    }
                }
            }
        }

        // Remove the directory by getting the index of its position in its parent node list and removing the node at that index
        int delIndex = nodeIndex(((Branch) prev).nodeList, name);
        ((Branch) prev).nodeList.remove(delIndex);
        return true;
    }
    return false;
}
```

GetStorage(path) :

07 November 2021 21:27

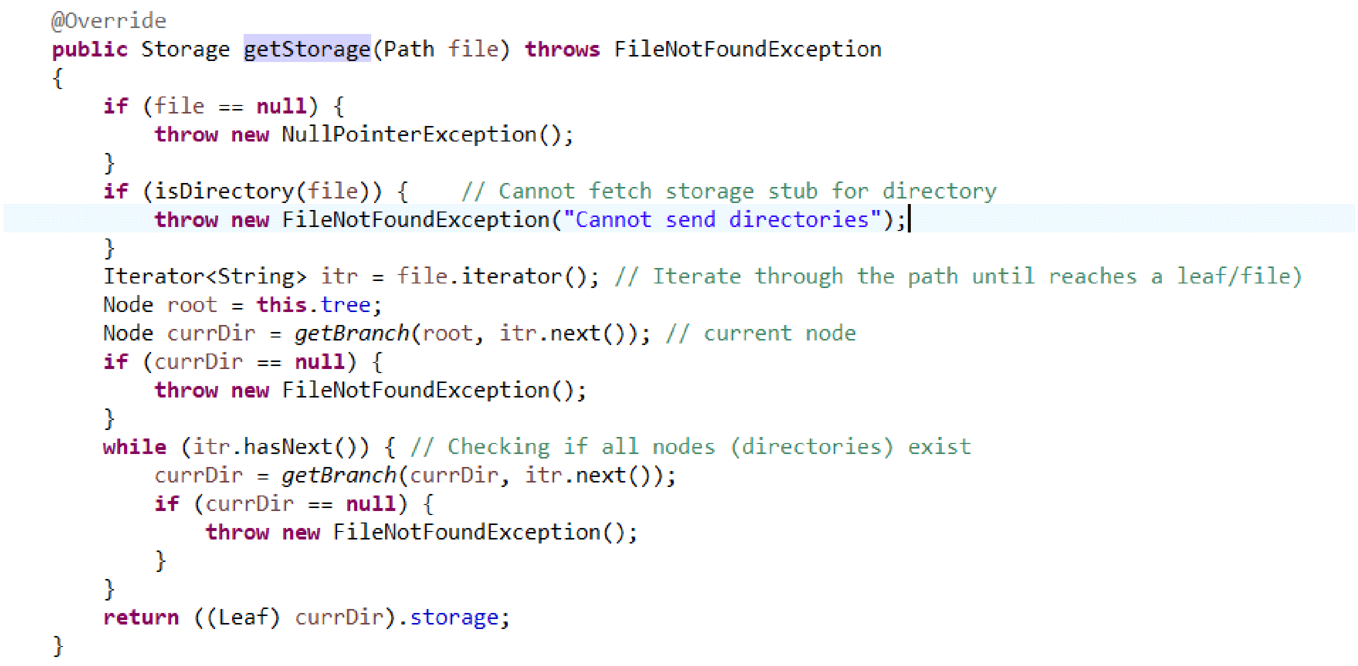
GetStorage(path) : get the Storage Server (or more precisely, a representing stub) hosting the file referred to by path.

1. Client fetches storage of the storageserver using naming server's getStorage



```
     */
    public DFSInputStream(Service naming_server, Path file)
        throws FileNotFoundException, IOException
    {
        // Retrieve a stub for the storage server hosting the file.
        try
        {
            storage_server = naming_server.getStorage(file);
        }
        catch(RMIException e)
        {
```

2. NamingServer implements getStorage



```
    @Override
    public Storage getStorage(Path file) throws FileNotFoundException
    {
        if (file == null) {
            throw new NullPointerException();
        }
        if (isDirectory(file)) {     // Cannot fetch storage stub for directory
            throw new FileNotFoundException("Cannot send directories");
        }
        Iterator<String> itr = file.iterator(); // Iterate through the path until reaches a leaf/file)
        Node root = this.tree;
        Node currDir = getBranch(root, itr.next()); // current node
        if (currDir == null) {
            throw new FileNotFoundException();
        }
        while (itr.hasNext()) { // Checking if all nodes (directories) exist
            currDir = getBranch(currDir, itr.next());
            if (currDir == null) {
                throw new FileNotFoundException();
            }
        }
        return ((Leaf) currDir).storage;
    }
```

3.