

# Object-oriented Programming in Java

## Session: 6

### New Features in File Handling





- ◆ Describe the Console class
- ◆ Explain the DeflaterInputStream class
- ◆ Explain the InflaterOutputStream class
- ◆ Describe the java.nio Package
- ◆ Describe the file system

For Aptech Centre Use Only



- ◆ NIO in Java stands for New Input/Output operation.
- ◆ It is a collection of Java APIs that offers intensive I/O operations.
- ◆ The `java.nio.file` package provides comprehensive support for input and output operations.
- ◆ There are many classes present in the API, but in this session, only some of them will be discussed.
- ◆ The `java.nio` package mainly defines buffers which are containers for data.
- ◆ This API is easy to use.

For Aptech Centre Use Only



- ◆ The `Console` class provides various methods to access character-based console device.
- ◆ These devices should be associated with the current virtual machine.
- ◆ Automatic invocation of virtual machine will not have a console associated with it.
- ◆ There are no public constructors for the `Console` class.
- ◆ To obtain an instance of the `Console` class, you need to invoke the `System.console()` method.
- ◆ The `System.console()` method returns the `Console` object if it is available; otherwise, it returns null.
- ◆ At present, the methods of `Console` class can be invoked only from the command line and not from Integrated Development Environments (IDEs), such as Eclipse, NetBeans, and so on.



- ◆ The following lists various methods available in the `Console` class:
  - ◆ `format(String fmt, Object... args)`
  - ◆ `printf(String fmt, Object... args)`
  - ◆ `reader()`
  - ◆ `readLine()`
  - ◆ `readLine(String fmt, Object... args)`
- ◆ The following Code Snippet shows the use of `Console` class methods:

### Code Snippet

```
...  
public static void main(String [] args) {  
    Console cons = System.console();  
    if (cons == null) {  
        System.err.println("No console device is present!");  
        return;  
    }  
    try {  
        String username = cons.readLine("Enter your username: ");
```



### Code Snippet

```
char [] pwd = cons.readPassword("Enter your secret Password:");
System.out.println("Username = " + username);
System.out.println("Password entered was = " + new
String(pwd));
} catch(IOException ioe) {
    cons.printf("I/O problem: %s\n", ioe.getMessage());
}
}
...
```

- ◆ The code accepts user name and password from the user through a console using the `readLine()` and `readPassword()` methods.
- ◆ The `System.console()` method returns a `Console` object if it is available that reads the username and password.



Class Name	Description
CheckedInputStream	Maintains checksum of data that is being read.
CheckedOutputStream	Maintains checksum of data that is to be written.
Deflater	Performs data compression.
DeflaterInputStream	Reads source data and then compresses it in the 'deflate' compression format.
DeflaterOutputStream	Reads source data, compresses it in 'deflate' compression format, and then writes the compressed data to the output stream.
Inflater	Performs data decompression.
InflaterInputStream	Reads compressed data and then decompresses it in the 'deflate' compression format.
InflaterOutputStream	Reads compressed data, decompresses it in the 'deflate' compression format, and then writes the decompressed data in the output stream.
ZipInputStream	Implements an input stream filter to read files in the ZIP file format. It supports compressed and uncompressed entries.
ZipOutputStream	Reads the source data, compresses it in ZIP file format, and writes the data in the output stream.



- ◆ The `Deflater` class compresses the data present in an input stream. It compresses the data using the ZLIB compression library.
- ◆ The constructor of the `Deflater` class is used to create instances of the `Deflater` class.

### Syntax

```
public Deflater()
```

- ◆ The following lists various methods available in the `Deflater` class:
  - ◆ `deflate(byte[] buffer)`
  - ◆ `deflate(byte[] buffer, int offset, int len)`
  - ◆ `setInput(byte[] buffer)`
  - ◆ `setInput(byte[] buffer, int offset, int len)`
  - ◆ `finish()`
  - ◆ `end()`





The following Code Snippet shows the use of methods in Deflater class:

### Code Snippet

```
import java.util.zip.Deflater;
public class DeflaterApplication {
    public static void main(String[] args) throws Exception {
        // Encode a String into bytes
        String input = "The Deflater class compresses the data.";
        byte[] inputObj = input.getBytes("UTF-8");
        // Compress the bytes
        byte[] output = new byte[100];
        Deflater deflater = new Deflater();
        deflater.setInput(inputObj);
        deflater.finish();
        int compressDataLength = deflater.deflate(output);
        System.out.println(compressDataLength);
    }
}
```



- ◆ The `Inflater` class decompresses the compressed data.
- ◆ This class supports decompression using the ZLIB compression library.

## Syntax

```
public Inflater()
```

- ◆ The following lists various methods available in the `Inflater` class:
  - ◆ `inflate(byte[] buffer)`
  - ◆ `inflate(byte[] buffer, int offset, int len)`
  - ◆ `setInput(byte[] buffer)`
  - ◆ `setInput(byte[] buffer, int offset, int len)`
  - ◆ `end()`



The following Code Snippet shows the use of methods in Inflater class:

### Code Snippet

```
public class DeflaterApplication {
    public static void main(String[] args) throws Exception {
        // Encode a String into bytes
        String input = "The Deflater class compresses the data.";
        byte[] inputObj = input.getBytes("UTF-8");
        // Compress the bytes
        byte[] output = new byte[100];
        Deflater deflaterObj = new Deflater();
        deflaterObj.setInput(inputObj);
        deflaterObj.finish();
        int compressDataLength = deflaterObj.deflate(output);
        System.out.println(compressDataLength);
        // Decompress the bytes
        Inflater inflaterObj = new Inflater();
        inflaterObj.setInput(output, 0, output.length);
```



```
byte[] resultObj = new byte[100];
int resultLength = inflaterObj.inflate(resultObj);
inflaterObj.end();
// Decode the bytes into a String
String strOutput = new String(resultObj, 0, resultLength, "UTF-8");
System.out.println("Recovered string is: " + strOutput);
}
```

For Apteck Centre Use Only



- ◆ The `DeflaterInputStream` class reads the source data from an input stream and then compresses it in the 'deflate' compression format.
- ◆ This class provides its own constructors and methods.
- ◆ The following lists various methods available in the `DeflaterInputStream` class:
  - ◆ `read()`
  - ◆ `read(byte[] buffer, int offset, int buffSize)` throws `IOException`
  - ◆ `close()`
  - ◆ `boolean markSupported()`
  - ◆ `int available()`
  - ◆ `long skip(long n)`

For Apteck Centre Use Only

## DeflaterInputStream Class [2-5]



The following Code Snippet shows the use of methods in DeflaterInputStream class:

### Code Snippet

```
package javaioapplication;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.zip.DeflaterInputStream;

public class DeflaterInputApplication {
    public static byte[] increaseArray(byte[] arrtemp) {
        byte[] temp = arrtemp;
```

## DeflaterInputStream Class [3-5]



```
arrtemp = new byte[arrtemp.length + 1];
// backs up the data
for (int itr = 0; itr < temp.length; itr++) {
    arrtemp[itr] = temp[itr];
}
return arrtemp;
}

public static void main(String args[]) throws IOException {
    FileOutputStream fos = null;
    try {
        File file = new File("C:/Java/Hello.txt");
        FileInputStream fis = new FileInputStream(file);
        DeflaterInputStream dis = new DeflaterInputStream(fis);
        // Creating byte array for deflating the data
```

## DeflaterInputStream Class [4-5]



```
byte input[] = new byte[0];
int iindex = -1;
// reads data from file
int iread = 0;
while ((iread = dis.read()) != -1) {
    input = increaseArray(input);
    input[++iindex] = (byte) iread;
}
fos = new FileOutputStream("C:/Java/DeflatedMain.dfl");
fos.write(input, 0, input.length);
fos.close();
System.out.println("File size after compression " +
input.length);
} catch (FileNotFoundException ex) {
```



## DeflaterInputStream Class [5-5]



```
        Logger.getLogger(DeflaterInputApplication.class.getName()).log(Level.SEVERE, null, ex);
    } finally {
        try {
            fos.close();
        } catch (IOException ex) {
            Logger.getLogger(DeflaterInputApplication.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

For Apteck Centre Use Only



- ◆ The `DeflaterOutputStream` class reads the source data, compresses it in the 'deflate' compression format, and then writes the compressed data to a predefined output stream.
- ◆ It also acts as the base for other types of compression filters, such as `GZIPOutputStream`.
- ◆ The following lists various methods available in `DeflaterOutputStream` class:
  - ◆ `write(int buffer)`
  - ◆ `write(byte[] buffer, int offset, int buffSize)`
  - ◆ `deflate()`
  - ◆ `close()`
  - ◆ `finish()`

For Apteck Centre Use Only

## DeflaterOutputStream Class [2-3]



The following Code Snippet shows the use of methods in DeflaterOutputStream class:

### Code Snippet

```
package javaioapplication;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.zip.DeflaterOutputStream;
public class DeflaterOutputApplication {
    public static void main(String args[]) {
        try {
            File filein = new File("C:/Java/Hello.txt");
            FileInputStream finRead = new FileInputStream(filein);

            File fileout = new File("C:/Java/DeflatedMain.dfl");
            FileOutputStream foutWrite = new FileOutputStream(fileout);
            DeflaterOutputStream deflWrite = new
            DeflaterOutputStream(foutWrite);
```

## DeflaterOutputStream Class [3-3]



```
System.out.println("Original file size " +
filein.length());
// Reading and writing the compressed data
int bread = 0;
while ((bread = finRead.read()) != -1) {
    deflWrite.write(bread);
}
// Closing objects
deflWrite.close();
finRead.close();
System.out.println("File size after compression " +
fileout.length());
} catch (Exception e) {
    e.printStackTrace();
}
}
```



- ◆ The `InflaterInputStream` class reads the compressed data and decompresses it in the 'deflate' compression format.

## Syntax

```
public InflaterInputStream(InputStream in)
```

The constructor creates an input stream of bytes to read the compressed data with a default decompressor and buffer size.

- ◆ The following lists various methods available in `InflaterInputStream` class:
  - ◆ `read()`
  - ◆ `read(byte[] buffer, int offset, int buffSize)`

## InflaterInputStream Class [2-3]



The following Code Snippet shows the use of methods in InflaterInputStream class:

### Code Snippet

```
...
public static void main(String args[]) {
try {
    File finf = new File("C:\\DeflatedMain.dfl");
    FileOutputStream foutWrite = new FileOutputStream(finf);
    File fout = new File("C:\\InflatedMain.java");
    FileInputStream finRead = new FileInputStream(fout);
    InflaterInputStream defRead = new
InflaterInputStream(finRead);
    System.out.println("File size before Inflation " +
fout.length());
    // Inflating the file to original size
    int bread = 0;
    while ((bread = defRead.read()) != -1) {
        foutWrite.write(bread);
    }
}
```



```
foutWrite.close();  
    System.out.println("File size after Inflation " +  
finf.length());  
    } catch (IOException ioe) {  
        ioe.printStackTrace();  
    }  
}  
...  
...  
...
```

- ◆ The code creates two File objects, `fout` and `finf` where, `fout` holds the location of compressed file and `finf` holds the location of decompressed file.
- ◆ The object, `fout` is passed as a reference to the `FileInputStream` and the object, `finf` is passed as a reference to the `FileOutputStream`.
- ◆ The `InflaterInputStream` object reads the data in the `FileInputStream` object, decompresses the compressed data, and then invokes the `write()` method to write the decompressed data to the output file named `InflatedMain.java`.



- ◆ The `InflaterOutputStream` class reads the compressed data, decompresses the data stored in the deflate compression format, and then writes the decompressed data to an output stream.
- ◆ This class also serves as the base class for the decompression class named `GZIPInputStream`.

## Syntax

```
public InflaterOutputStream(OutputStream out)
```

The constructor creates an output stream of bytes to write decompressed data with a default decompressor and buffer size.

- ◆ The following lists various methods available in the `InflaterOutputStream` class:
  - ◆ `write(int buffer)`
  - ◆ `write(byte[] buffer, int offset, int buffSize)`
  - ◆ `close()`
  - ◆ `finish()`



## InflaterOutputStream Class [2-3]



The following Code Snippet shows decompression of data using the methods of InflaterOutputStream class:

### Code Snippet

```
public class InflaterOutputStreamApplication {
    public static void main(String args[]) {
        try {
            // Writing decompressed data
            File fin = new File("C:/Java/DeflatedMain.dfl");
            FileInputStream finWrite = new FileInputStream(fin);
            File fout = new File("C:/Java/InflatedMain.java");
            FileOutputStream foutWrite = new FileOutputStream(fout);
            InflaterOutputStream infWrite = new
InflaterOutputStream(foutWrite);
            System.out.println("Original file size " + fin.length());
            // Reading and writing the decompressed data
            int bread = 0;
            while ((bread = finWrite.read()) != -1) {
```

## InflaterOutputStream Class [3-3]



```
        infWrite.write(bread);  
    }  
    // Inflating the file to original size  
    infWrite.close();  
    System.out.println("File size after Inflation " +  
fout.length());  
    } catch (IOException ioe) {  
        ioe.printStackTrace();  
    }  
    }  
}
```

For Apteck Centre Use Only



- ◆ NIO is platform dependent.
- ◆ Its ability to enhance application performance depends on the following:
  - ◆ OS
  - ◆ Specific JVM
  - ◆ Mass storage characteristics
  - ◆ Data
  - ◆ Host virtualization context

## **Central features of the NIO APIs:**

- ◆ Charsets and their Associated Decoders and Encoders translate the data between bytes and Unicode characters. The charset API is defined in the `java.nio.charset` package.
- ◆ Buffers are containers for data. The buffer classes are defined in the `java.nio` package and are used by all NIO APIs.
- ◆ Channels of Various Types represent connections to entities that can perform I/O operations.
- ◆ Selectors and Selection Keys define a multiplexed, non-blocking I/O facility.



## File Systems

- ◆ A file system stores and organizes files on media, typically hard drives.
- ◆ Typically, files are stored in a hierarchical structure, where there is a root node.
- ◆ Below this node exist files and directories.
- ◆ Each directory can contain files and subdirectories, which can contain files and subdirectories and so on.
- ◆ There is no limit to the hierarchical structure.
- ◆ File systems can have one or more root directories.
- ◆ File systems have different characteristics for path separators.

## Path

- ◆ Every file is identified through its path.
- ◆ It starts from the root node.
- ◆ File system include different characteristics for path separators.



## Files

NIO.2 includes the following new package and classes:

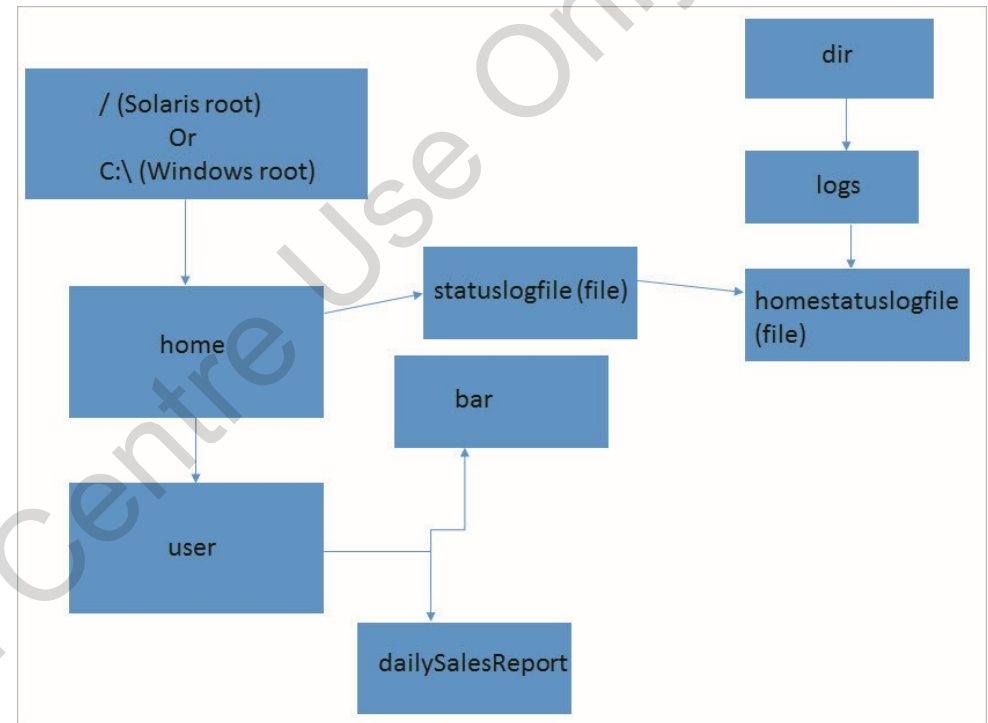
- ◆ `java.nio.file.Path`: This uses a system dependent path to locate a file or a directory.
- ◆ `Java.nio.file.Files`: This uses a `Path` object to perform operations on files and directories.
- ◆ `java.nio.file.FileSystem`: This provides an interface to a file system. This also helps to create a `Path` object and other objects to access a file system.

For Apteck Centre Use Only

# Symbolic Links



- ◆ A symbolic link is a reference to another file and is transparent to applications and users.
- ◆ Operations on symbolic links are automatically redirected to the target of the link.
- ◆ Here, the target is the file or directory that is pointed to.



**Symbolic Link**



- ◆ The `java.nio.file.Path` interface object can help to locate a file in a file system.
- ◆ Typically, the interface represents a system dependent file path.
- ◆ A Path is hierarchical.
- ◆ It includes a sequence of directory and file name elements.
- ◆ These are separated by a delimiter.
- ◆ Following are the features of a Path:
  - ◆ There could be a root component. This represents a file system hierarchy.
  - ◆ The name of a file or directory is the name element that is extreme far from the root of the directory hierarchy.
  - ◆ The other name elements include directory names.
- ◆ A Path can represent the following:
  - ◆ A root
  - ◆ A root and a sequence of names
  - ◆ One or more name elements



- ◆ Following are some of the methods of the Path interface that can be grouped based on their common functionalities:
  - ◆ To access the path components or a subsequence of its name elements, the `getFileName()`, `getParent()`, `getRoot()`, and `subpath()` methods can be used.
  - ◆ To combine paths, the Path interface defines `resolve()` and `resolveSibling()` methods can be used.
  - ◆ To construct a relative path between two paths, the `relativize()` method can be used.
  - ◆ To compare and test paths, the `startsWith()` and `endsWith()` methods can be used.
- ◆ To obtain a Path object, obtain an instance of the default file system. Next, invoke the `getPath()` method.

### Code Snippet

```
FileSystem fs = FileSystem.getDefault();  
Path pathObj = fsObj.getPath("C:/Java/Hello.txt");
```





The following Code Snippet displays the use of the Path interface:

### Code Snippet

```
package javaioapplication;
import java.nio.file.Path;
import java.nio.file.Paths;
public class PathApplication {
public static void main(String[] args){
    Path pathObj = Paths.get("C:/Java/Hello.txt");
    System.out.printf("FileName is: %s\n", pathObj.getFileName());
    System.out.printf("Parent is: %s\n", pathObj.getParent());
    System.out.printf("Name count is: %d\n",
pathObj.getNameCount());
    System.out.printf("Root directory is: %s\n",
pathObj.getRoot());
    System.out.printf("Is Absolute: %s\n", pathObj.isAbsolute());
    }
}
```



- ◆ Path interface is link aware and every method either detects what to do or provides an option to configure the behavior when a symbolic link is encountered.
- ◆ While certain file systems support symbolic link, certain support hard links.
- ◆ Hard links differ from symbolic links.
- ◆ The following defines a hard link:
  - ◆ It is not allowed on directories and not allowed to cross partitions or volumes.
  - ◆ It is hard to find as it behaves like a regular file.
  - ◆ It should include the target of the link.

For Apteck Centre Use Only



- ◆ Static methods in the `java.nio.file.Files` class perform primary functions for the Path objects.
- ◆ The methods in the class can identify and automatically manage symbolic links.
- ◆ Following are the various `File` operations:
  - ◆ Copying a file or directory
  - ◆ Moving a file or directory
  - ◆ Checking a file or directory
  - ◆ Deleting a file or directory
  - ◆ Listing a Directory's Content
  - ◆ Creating and Reading Directories
  - ◆ Reading and Writing from Files
  - ◆ Reading a File by Using Buffered Stream I/O
  - ◆ Accessing a File Randomly



## Copying a file or directory

- ◆ To do so, the `copy(Path, Path, CopyOption...)` method can be used.
- ◆ The following Code Snippet shows how to use the `copy()` method:

### Code Snippet

```
import static java.nio.file.StandardCopyOption.*;
...
Files.copy(source, target, REPLACE_EXISTING);
```



## Moving a file or directory

- ◆ To do so, use the `move(Path, Path, CopyOption...)` method.
- ◆ If the target file exists, the `REPLACE_EXISTING` option should be used.
- ◆ Otherwise, the move fails.
- ◆ The method takes a varargs argument.
- ◆ The following Syntax shows how to use the `move()` method:

### Syntax

```
import static java.nio.file.StandardCopyOption.*;
...
Files.move(source, target, REPLACE_EXISTING);
```



## Checking a file or directory

- ◆ To do so, the file system should be accessed using the `Files` methods to determine if a particular `Path` exists.
- ◆ The methods in the `Path` class operate on the `Path` instance.
- ◆ Following are the `Files` methods for checking the existence of `Path` instance:
  - ◆ `exists(Path, LinkOption...opt)`: These check if the file exists. By default, it uses symbolic links.
  - ◆ `notExists(Path, LinkOption...)`: These check if the file does not exist.
  - ◆ `Files.exists(path)` is not equivalent to `Files.notExists(path)`.
- ◆ When testing a file's existence, one of the following is the possible outcome:
  - ◆ The file is verified to not exist.
  - ◆ The file is verified to exist.
  - ◆ The existence of the file cannot be verified.
  - ◆ The file's status is unknown.



## Deleting a file or directory

- ◆ The `delete(Path)` method can be used to delete a file, directories, or links.
- ◆ The deletion will fail if the directory is not empty.
- ◆ The method throws an exception if the deletion fails.
- ◆ If the file does not exist, a `NoSuchFileException` is thrown.

For Apteck Centre Use Only



To determine the reason for the failure, the following exception can be caught as shown in the Code Snippet:

### Code Snippet

```
try {  
    Files.delete(path);  
} catch (NoSuchFileException x) {  
    System.err.format("%s: no such" + " file or directory%n",  
path);  
} catch (DirectoryNotEmptyException x) {  
    System.err.format("%s not empty%n", path);  
} catch (IOException x) {  
    // File permission problems are caught here.  
    System.err.println(x);  
}
```





## Listing a Directory's Content

- ◆ To do so, use the `DirectoryStream` class that iterates over all the files and directories from any `Path` directory.
- ◆ Consider the following:
  - ◆ `DirectoryIteratorException` is thrown if there is an I/O error while iterating over the entries in the specified directory.
  - ◆ `PatternSyntaxException` is thrown when the pattern is invalid.
- ◆ The following Code Snippet displays the use of `DirectoryStream` class:

### Code Snippet

```
import java.io.IOException;
import java.nio.file.DirectoryIteratorException;
import java.nio.file.DirectoryStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
```

# Tracking File System and Directory Changes [8-17]



```
import java.util.Iterator;
public class ListDirApplication {
    public static void main(String[] args) {
        Path pathObj = Paths.get("D:/resources");
        try (DirectoryStream<Path> dirStreamObj =
Files.newDirectoryStream(pathObj, "*.java")) {
            for (Iterator<Path> itrObj = dirStreamObj.iterator(); itrObj.
hasNext();) {
                Path fileObj = itrObj.next();
                System.out.println(fileObj.getFileName());
            }
        } catch (IOException | DirectoryIteratorException ex) {
            // IOException can never be thrown by the iteration.
            // In this snippet, itObj can only be thrown by
            newDirectoryStream.
            System.err.println(ex.getMessage());
        }
    }
}
```



## Creating and Reading Directories

- ◆ The `createDirectory(Path dir)` method is used to create a new directory.
- ◆ The `createDirectories()` method can be used to create directories from top to bottom.
- ◆ The following Code Snippet illustrates this:

### Code Snippet

```
Files.createDirectories(Paths.get("C:/Java/test/example"));
```

For Apteck Centre Use Only



## Reading and Writing from Files

- ◆ To read from files, use the `readAllBytes` or `ReadAllLines` methods that will read the entire content of the file.
- ◆ The following Code Snippet shows the use of `ReadAllLines` method:

### Code Snippet

```
Files.createDirectories(Paths.get("C:/Java/test/example"));
```

For Aptech Centre Use Only



## Reading a File by Using Buffered Stream I/O

- ◆ The `newBufferedReader(Path, Charset)` method opens a file for reading.
- ◆ This returns a `BufferedReader` object that helps to read text from a file efficiently.
- ◆ The following Code Snippet demonstrates the use of `newBufferedReader()` method:

### Code Snippet

```
Import java.io.BufferedReader;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
public class BufferedIOApplication {
```

# Tracking File System and Directory Changes [12-17]



```
public static void main(String[] args) {  
    Path pathObj = Paths.get("C:/Java/Hello.txt");  
    Charset charset = Charset.forName("US-ASCII");  
    try (BufferedReader buffReadObj =  
Files.newBufferedReader(pathObj, charset)) {  
        String lineObj = null;  
        while ((lineObj = buffReadObj.readLine()) != null) {  
            System.out.println(lineObj);  
        }  
    } catch (IOException e) {  
        System.err.format("IOException: %s%n", e);  
    }  
}
```

For Aptech Centre Use Only



## Accessing a File Randomly

- ◆ Files can be nonsequentially or randomly accessed using the `SeekableByteChannel` interface.
- ◆ To randomly access a file, perform the following:
  - ◆ Open the file.
  - ◆ Find the particular location.
  - ◆ Read from the file or write to the file.
- ◆ The `SeekableByteChannel` interface extends channel I/O and includes various methods to set the position.
- ◆ The data can then be read from the location or written to it.
- ◆ The interface includes the following methods:
  - ◆ `read(ByteBuffer)`
  - ◆ `write(ByteBuffer)`
  - ◆ `truncate(long)`



- ◆ position
- ◆ position(long)
- ◆ The following Code Snippet displays the use of the methods of random access file:

## Code Snippet

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.Path;
import java.nio.file.Paths;
import static java.nio.file.StandardOpenOption.*;
public class RandomAccessApplication {
```



# Tracking File System and Directory Changes [15-17]



```
public static void main(String[] args) throws
FileNotFoundException {
    String strObj = "I love Java programming!\n";
    byte byteDataObj[] = strObj.getBytes();
    ByteBuffer bufObj = ByteBuffer.wrap(byteDataObj);

    ByteBuffer bufCopyObj = ByteBuffer.allocate(20);
    Path pathObj = Paths.get("C:/Java/NewHello.txt");
    //creates a new file if it is not existing
    try (FileChannel fcObj = (FileChannel.open(pathObj, CREATE,
    READ, WRITE))) {
        //reads the first 20 bytes of the file
        int nreadChar;
        do {
            nreadChar = fcObj.read(bufCopyObj);
            System.out.println(nreadChar);
        } while (nreadChar != -1 && bufCopyObj.hasRemaining());
```

# Tracking File System and Directory Changes [16-17]



```
// writes the string at the beginning of the file.
fcObj.position(0);
while (bufObj.hasRemaining()) {
    fcObj.write(bufObj);
}
bufObj.rewind();

// Moves to the end of the file and copies the first 20 bytes to
// the end of the file.
long length = fcObj.size();
fcObj.position(length - 1);
//flips the buffer and sets the limit to the current position
bufCopyObj.flip();
while (bufCopyObj.hasRemaining()) {
    fcObj.write(bufCopyObj);
}
```

# Tracking File System and Directory Changes [17-17]



```
while (bufObj.hasRemaining()) {  
    fcObj.write(bufObj);  
}  
} catch (IOException ex) {  
    System.out.println("I/O Exception: " + ex.getMessage());  
}  
}  
}
```

For Apteck Centre Use Only



- ◆ The `FileSystem` class provides an interface to a file system and is the factory for objects to access files and other objects in the file system.
- ◆ It includes several types of objects.
- ◆ Following are some of the objects:
  - ◆ `getUserPrincipalLookupService()`
  - ◆ `newWatchService()`
  - ◆ `getPath()`
  - ◆ `getPathMatcher()`
  - ◆ `getFileStores()`
- ◆ A file system can have a single hierarchy of files or several distinct file hierarchies.
- ◆ A single hierarchy of files includes one top-level root directory.
- ◆ A file system can include one or more `file-stores`.



- ◆ A WatchService watches registered objects for changes and events.
- ◆ Multiple concurrent consumers can use a WatchService.
- ◆ A file manager can use a WatchService to check a directory for changes such as when files are created or deleted.
- ◆ The `register()` method is invoked to register a `Watchable` object with a WatchService.
- ◆ This method returns a `WatchKey` to represent the registration.
- ◆ When a change or event for an object occurs, the key is signaled or queued to the WatchService.
- ◆ After the events are processed, the consumer invokes the key's `reset()` method.
- ◆ This resets the key. The key is then signaled and re-queued with further events.
- ◆ To cancel a registration with a WatchService, the key's `cancel()` method is invoked.



- ◆ To locate a file one would search directory.
- ◆ One could use a search tool or the `PathMatcher` interface which has a `match` method that determines whether a `Path` object matches a specified string.
- ◆ The `PathMatcher` interface is implemented by objects to match operations on paths.
- ◆ The following is the syntax for the `syntaxAndPattern` string:

## Syntax

`Syntax:pattern`

- ◆ The following lists certain patterns:
  - ◆ `*.*{java,class}`
  - ◆ `Client.?`
  - ◆ `C:\\*`
  - ◆ `*.java`
  - ◆ `*.*`



- ◆ To find a file, the user will search a directory recursively.
- ◆ The `java.nio.PathMatcher` interface has a `match` method to determine whether a `Path` object matches the specified search string.
- ◆ The `FileSystem` factory methods can be used to retrieve the `PathMatcher` instance.
- ◆ To walk a file tree, the `FileVisitor` interface needs to be implemented.
- ◆ This interface specifies the behavior in traversal process.
- ◆ The key points in the traversal process includes when a file is visited, before accessing a directory, after accessing a directory, or when a failure happens.
- ◆ The methods corresponding to these situations are as follows:
  - ◆ `preVisitDirectory()` – invoked before visiting a directory
  - ◆ `postVisitDirectory()` – invoked after all entries in a directory is visited
  - ◆ `visitFile()` – invoked when a file is visited
  - ◆ `visitFileFailed()` – invoked when a file cannot be accessed



- ◆ The `SimpleFileVisitor` class implements the `FileVisitor` interface and overrides all the methods of this class.
- ◆ This class visits all the files and when it encounters an error, it throws an `IOException`.
- ◆ This class can be extended and only the required methods is required to be overridden.
- ◆ The Code Snippet displays the use of `PatternMatcher` and the `SimpleFileVisitor` class to search for a file based on a pattern:

### Code Snippet

```
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.PathMatcher;
import java.nio.file.Paths;
import java.nio.file.SimpleFileVisitor;
import static java.nio.file.FileVisitResult.*;
```



## PathMatcher Interface [4-6]



```
import java.nio.file.FileVisitResult;
import java.nio.file.attribute.BasicFileAttributes;
class Finder extends SimpleFileVisitor<Path> {

    private Path file;
    private PathMatcher matcher;
    private int num;

    public Finder(Path path, PathMatcher matcher) {
        file = path;
        this.matcher = matcher;
    }

    private void find(Path file) {
        Path name = file.getFileName();
        if (name != null && matcher.matches(name)) {
            num++;
            System.out.println(file);
        }
    }
}
```

## PathMatcher Interface [5-6]



```
void done() {
System.out.println("Matched: " + num);
}
@Override
public FileVisitResult visitFile(Path file, BasicFileAttributes
attr) {
    find(file);
    return CONTINUE;
}

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

## PathMatcher Interface [6-6]



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

public class PathMatcherApplication {
    public static void main(String[] args) throws IOException {
        Path pathObj;
        pathObj = Paths.get("D:/resources");
        PathMatcher matcherObj = FileSystems.getDefault().getPathMatcher("glob:" +
        "*.java");
        Finder finder = new Finder(pathObj, matcherObj);
        try {
            Files.walkFileTree(pathObj, finder);
        } catch (IOException ex) {
            System.out.println(ex);
        }
    }
}
```



- ◆ In a file system, the files and directories include data.
- ◆ It is the metadata that tracks information about each object in the file system.
- ◆ Information can be such as the file creation date, the last file modified date, file owner, and so on.
- ◆ Metadata refers to file attributes of a file system.
- ◆ These file attributes can be achieved by the following methods of the `Files` class:
  - ◆ `isRegularFile(Path, LinkOption...)`
  - ◆ `isSymbolicLink(Path)`
  - ◆ `size(Path)`
  - ◆ `isDirectory(Path, LinkOption)`
  - ◆ `isHidden(Path)`
  - ◆ `getPosixFilePermissions(Path, LinkOption...)`
  - ◆ `setPosixFilePermissions(Path, Set<Pos ixFilePermission>)`
  - ◆ `getAttribute(Path, String, LinkOption...)`



- ◆ `setAttribute(Path, String, Object, LinkOption...)`
- ◆ `getLastModifiedTime (Path, LinkOption...)`
- ◆ `setLastModifiedTime(Path, FileTime)`
- ◆ `getOwner(Path, LinkOption...)`
- ◆ `setOwner(Path, UserPrincipal)`
- ◆ To obtain a set of attributes, the `Files` class includes the following two `readAttributes()` methods to get a file's attributes in one bulk operation:
  - ◆ `readAttributes(Path p, Class<A> type, LinkOption... option)`
  - ◆ `readAttributes(Path p, String str, LinkOption... option)`
- ◆ The basic attributes of a file system includes the following time stamps:
  - ◆ `creationTime`
  - ◆ `lastModifiedTime`
  - ◆ `lastAccessTime`



- ◆ File systems other than DOS such as Samba also support DOS file attributes.
- ◆ The following Code Snippet shows the use of the methods of the `DosFileAttributes` class:

## Code Snippet

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.attribute.DosFileAttributes;
public class DOSFileAttriApplication {
    /**
     *
     * @param args
     * @throws IOException
     */
}
```

## DOS File Attributes [2-2]



```
public static void main(String[] args) throws IOException {
    Path pathObj;
    pathObj = Paths.get("C:/Java/Hello.txt");
    try {
        DosFileAttributes attrObj =
            Files.readAttributes(pathObj, DosFileAttributes.class);
        System.out.println("Is ReadOnly: " + attrObj.isReadOnly());
        System.out.println("Is Hidden: " + attrObj.isHidden());
        System.out.println("Is Archive: " + attrObj.isArchive());
        System.out.println("Is System: " + attrObj.isSystem());
    } catch (UnsupportedOperationException ex) {
        System.err.println("DOS file" + " attributes not supported:"
+ ex);
    }
}
```



- ◆ Java SE 6 has introduced the Console class to enhance and simplify command line applications.
- ◆ The Console class provides various methods to access character-based console device.
- ◆ Java in its java.util.zip package provides classes that can compress and decompress files.
- ◆ The Deflater and Inflater classes extend from the Object class.
- ◆ The DeflaterInputStream and DeflaterOutputStream classes are inherited from the FilterInputStream and FilterOutputStream classes respectively.
- ◆ The InflaterInputStream and InflaterOutputStream classes are inherited from the FilterInputStream and FilterOutputStream classes respectively.
- ◆ NIO is platform dependent. A file system stores and organizes files on media, typically hard drives.