I\O Streams

I\O - Introduction

- A stream is a flow of data. The Java programming language has a huge range of classes that their instances represent streams.
- The stream can also be thought of as a pipe through which the data flows.

I\O Streams- Categories

	Input		Output	
	byte	char	byte	char
node streams	InputStream FileInputStream PipedInputStream…	Reader FileReader PipedReader …	OutputStream, FileOutputStream, PipedOutputStream…	Writer FileWriter PipedWriter …
filter streams	BuferedInputStream DataInputStream ObjectInputStream…	FilterReader InputStreamReader BufferedReader ...	BuferedOutputStream DataOutputStream ObjectOutputStream ...	FilterWriter, InputStreamWriter BufferedWriter …

Byte & Char streams

- The streams that java supports are categorized into two different categories: byte streams and character streams.
- Input and Output of character data is handled by readers and writers.
- Input and Output of byte data is handled by input streams and output streams.

The InputStream Abstract Class

- The super class of all the input streams. It includes the following methods:

```
public abstract int read()
public int read(byte []vec)
public int read(byte []vec, int offset, int length)
public void close()
```

The InputStream Abstract Class

```
public int available()
public void skip (long n)
public boolean markSupported()
void mark(int readLimit)
void reset()
```

The OutputStream Abstract Class

- The super class of all output streams. It includes the following methods:

```
public abstract void write(int val)
public void write(byte[] vec)
public void write(byte[] vec, int offset, int lengt)
public void close()
public flush()
```

Writing\Reading To\From Files

- The I\O classes include specific classes that describe streams to\from files.
- The following example presents a stand alone application that copies a given file.
- Note the `FileInputStream` and `FileOutputStream` classes that extend `InputStream` and `OutputStream` respectively.

Writing\Reading To\From Files

```
import java.io.*;
public class CopyFile
{
    public static void main(String args[])
    {
        FileInputStream fis = null;
        FileOutputStream fos = null;
        try
        {
            fis = new FileInputStream(args[0]);
            fos = new FileOutputStream(args[1]);
            int data;
            data = fis.read();
```

Writing\Reading To\From Files

```
        while(data!=-1)
        {
                fos.write(data);
                data = fis.read();
        }
        fos.flush();
    }
    catch(IOException e)
    {...}
    finally
    {
        if(fos!=null) try{fos.close()} catch(IOException e) {}
        if(fis!=null) try{fis.close()} catch(IOException e) {}
    }
}
```

The Reader Methods

- The `Reader` class includes the following methods:

```
public int read()
public int read(char[] buffer)
public int read(char[] buffer, int offset, int lengt)
public void close()
public boolean ready()
public void skip(long num)
public boolean markSupported()
public void mark(int limit)
public void reset()
```

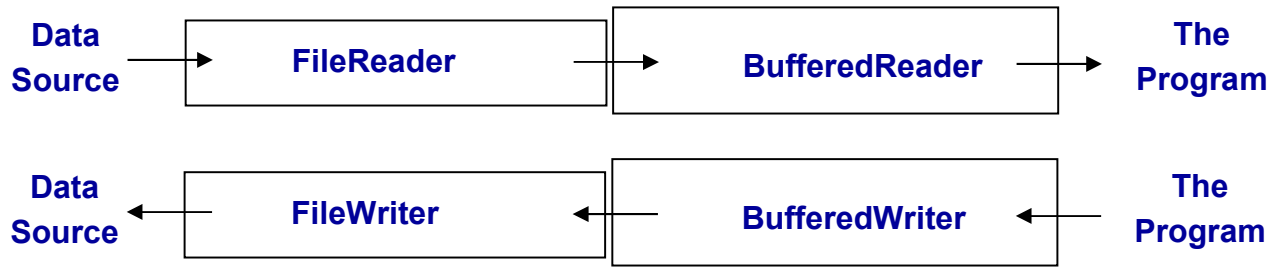
The Writer Methods

- The `Writer` class includes the following methods:

```
public void write(int c)
public void write(char[] buffer)
public void write(char[] buffer, int offset, int leng)
public void write(String str)
public void write(String str, int offSet, int leng)
public void close()
public void flush()
```

Streams Chaining

- Programs in Java usually chains a series of streams together to process the data.
- Each stream contributes something to the streams chain.



Streams Chaining

```
import java.io.*;
public class CopyFileV2
{
    public static void main(String args[])
    {
        FileReader in = null;
        FileWriter out = null;
        BufferedReader br = null;
        BufferedWriter bw = null;
        try
        {
            in = new FileReader(args[0]);
            out = new FileWriter(args[1]);
```

Streams Chaining

```
    br = new BufferedReader(in);
    bw = new BufferedWriter(out);
    String currentLine = null;
    currentLine = br.readLine();
    while(currentLine!=null)
    {
        bw.write(currentLine,0,
        currentLine.length());
        bw.newLine();
        currentLine = br.readLine();
    }
}
```

Streams Chaining

```
        catch(IOException e)
        {
            System.out.println("exception appened");
        }
        finally
        {
            if(br!=null) try{br.close();} catch(Exception e){ }
            if(bw!=null) try{bw.close();} catch(Exception e){}
        }
    }
}
```

The `InputStreamReader` and `OutputStreamWriter` classes

- The readers and writers flowing data include chars.
- The input streams and output streams flowing data include bytes.
- The `InputStreamReader` and the `OutputStreamWriter` serve as a bridge between characters flowing and bytes flowing.

The `URL` class

- The `URL` class represents a Uniform Resource Locator, a pointer to a "resource" on the web.
- A resource can be something as simple as a file or a directory, or it can be a reference to a more complicated object, such as a servelt, JSP, ASP or a CGI program.

The `URL` class

- Calling the `openStream()` on the `URL` object we shall get a reference for an input stream connected directly with the represented resource.

The URL class

```
import java.io.*;
import java.net.*;
public class URLExample
{
    public static void main(String args[])
    {
        InputStream in = null;
        URL url = null;
        try
        {
            url = new URL("http://www.yahoo.com/index.html");
            in = url.openStream();
            int tmp = in.read();
```

The URL class

```
            while(tmp!=-1)
            {
                System.out.print((char)tmp);
                tmp = in.read();
            }
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
        finally
        {
            if (in!=null) try {in.close();} catch(Exception e) {}
        }
}
```

Object Serialization

- Java enables reading\writing an object to a stream.
- The written\read object must be `Serializable`.
- The written\read object's class should include the `serialVersionUID` static variable declaration.
- When an object can be stored to disk we can describe the object as a *persistent capable* one.

Object Serialization

- When an object is serialized, only the data in its instance variables is preserved. Methods and static variables are not part of the serialized stream.
- When a data member of the serialized object is also a `Serializable` object then it is also serialized. The whole objects graph is serialized.

Object Serialization

- The following code writes an object to specific file.

```
..
ObjectOutputStream oos = new ObjectOutputStream( new
            FileOutputStream("birthday.ser"));
oos.writeObject(new Date());
..
```

Object Serialization

- The following code reads an object from a specific file.

```
..
ObjectIntputStream ois = new ObjectIntputStream( new
            FileInputStream(“birthday.ser”));
Date dat = (Date)ois.readObject();
..
```

Object Serialization

- The variables we don't want to include within the serialization process should be marked with `transient`.

The `File` Class

- Instantiating the class `File`:

```
File file = new File("myKkk.txt");
```

- Once we have a `File` instance we can call various methods on it:

```
public String getName()
public String getParent()
public String getAbsolutePath()
public void renameTo(String str)
ublic boolean canWrite()
...
```

The `RandomAccessFile` Class

- The `RandomAccessFile` enables to access a file without reading it from its beginning.
- The `RandomAccessFile` doesn't belong to the input\output streams hierarchy neither to the reader\writer hierarchy.

The `RandomAccessFile` Class

- The main methods this class includes are:

```
public long getFilePointer()

public void seek(long position)

public long length()

public int read()
```

The `Path` Class

- The `Path` class is used for representing paths in the files system.

  Object of type 'Path' can also represent a path that doesn't exist.

- Each object instantiated from `Path` contains the file name and a list of all directories that construct the path.

  Using an object of type Path we can examine, locate and manipulate files.

The `Path` Class

- The `Path` object isn't system independent. We cannot compare a `Path` object constructed on one operation system with a `Path` object constructed on another.

  Even when the directory structure is the same, there are differences between operation systems in how the path looks.

The `Path` Class

- Once we have a `Path` object we can manipulate it in various ways.

  We can append other paths to it, extract pieces of it, compare it with others etc. It is also possible to check the existence of the file the path refers to, create the file if it still doesn't exist, open it, delete it, change its permissions etc.

Getting a `Path` Object

- The `Paths` class includes two get static factory methods allowing us to get a `Path` object.

```
public static Path get(String path)
public static Path get(URI uri)
```

Path Operations

- The `Path` class includes methods that allow us to perform various operations on it.

  Those operations include the capability to obtain information about the path, access its elements, extract parts of it and convert it into something else.

- The available methods treat the paths a Path object holds as if they are indexed starting with 0 for the top element ending with n-1 for the last one.

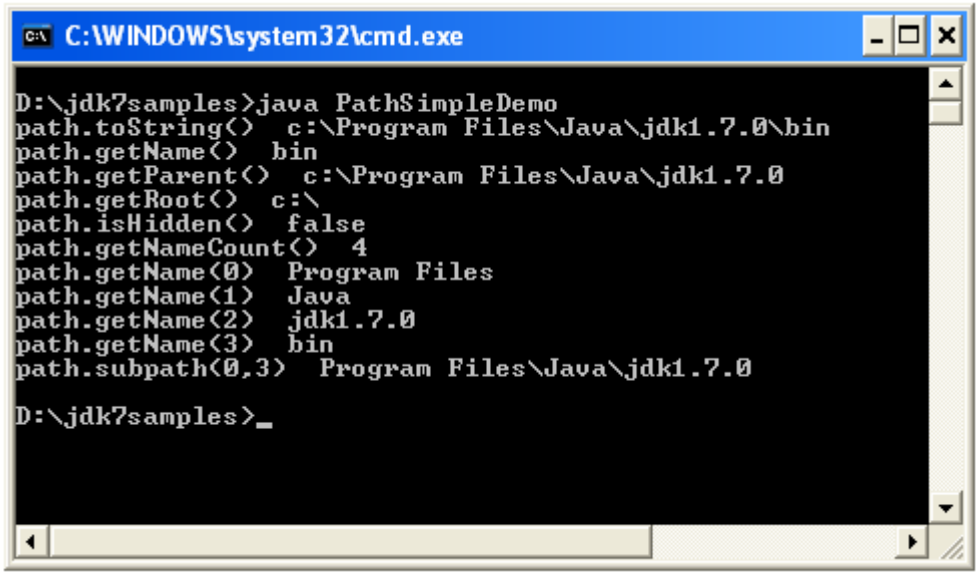
Path Operations

```
import java.nio.file.*;
import java.io.*;


public class PathSimpleDemo
{
  public static void main(String args[])
  {
    Path path = Paths.get("c:\\Program Files\\Java\\jdk1.7.0\\bin");
    System.out.println("path.toString()  "+path.toString());
    System.out.println("path.getName()  "+path.getName());
    System.out.println("path.getParent()  "+path.getParent());
    System.out.println("path.getRoot()  "+path.getRoot());
    try
    {
      System.out.println("path.isHidden()  "+path.isHidden());
    }
    catch(IOException e) {e.printStackTrace();}
```

Path Operations

```
    System.out.println("path.getNameCount()  "+path.getNameCount());
    int count = path.getNameCount();
    for(int i=0;i<count;i++)
    {
      System.out.println("path.getName("+i+")  "+path.getName(i));
    }
    System.out.println("path.subpath(0,3)  "+path.subpath(0,3));
  }
}
```

Path Operations



The `normalize()` Method

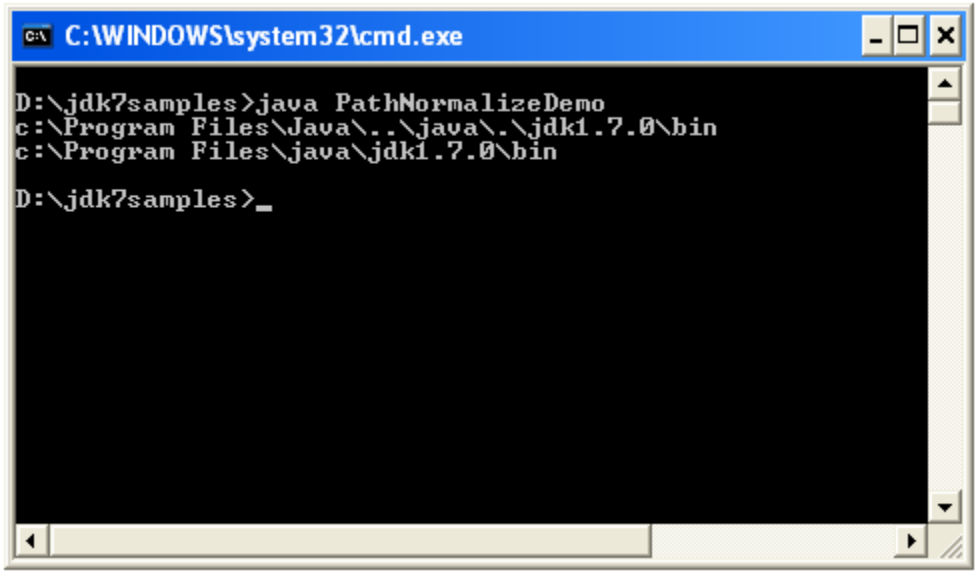
- This method removes redundant parts of the given path.

```
public abstract Path normalize()
```

The `normalize()` Method

```
import java.nio.file.*;
import java.io.*;


public class PathNormalizeDemo
{
  public static void main(String args[])
  {
    Path pathBefore = Paths.get(
       "c:\\Program Files\\Java\\..\\java\\.\\jdk1.7.0\\bin");
    Path pathAfter = pathBefore.normalize();
    System.out.println(pathBefore);
    System.out.println(pathAfter);
  }
}
```

The `normalize()` Method



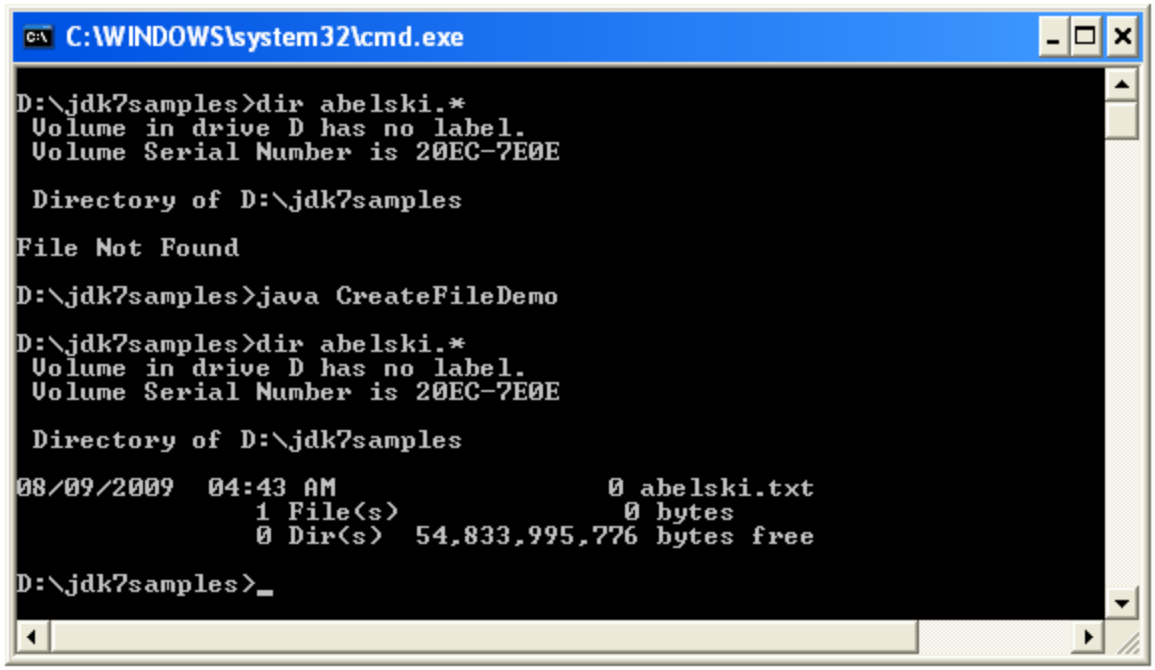
The `createFile` Method

- Calling the `createFile` method we can create a new file based on its `Path` representation.

The `createFile` Method

```
import java.nio.file.*;
import java.io.*;


public class CreateFileDemo
{
  public static void main(String args[])
  {
    try
    {
      Path path = Paths.get("d:\\jdk7samples\\abelski.txt");
      path.createFile();
    }
    catch(IOException e)
    {
      e.printStackTrace();
    }
  }
}
```

The `createFile` Method



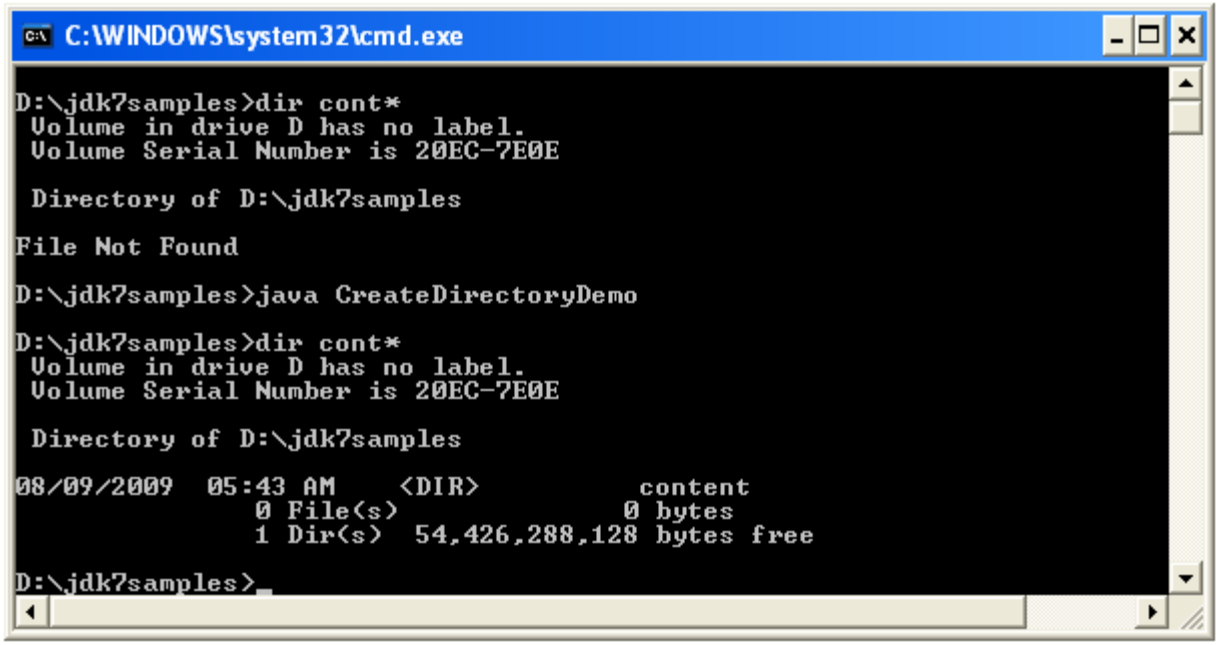
The `createDirectory` Method

- Calling the `createDirectory` method we can create a new directory based on its Path representation.

The `createDirectory` Method

```
import java.nio.file.*;
import java.io.*;


public class CreateDirectoryDemo
{
  public static void main(String args[])
  {
    try
    {
        Path path = Paths.get("d:\\jdk7samples\\content");
        path.createDirectory();
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
  }
}
```

The `createDirectory` Method



The `FileRef` Interface

- The `Path` class implements the `FileRef` interface.

  This interface includes various methods that can provide more information about a given file and allow us even getting InputStream and OutputStream of that file.

```
Object getAttribute(String attribute,
     LinkOption... options) throws IOException

void setAttribute(String attribute,
     Object value, LinkOption... options) throws IOException
```

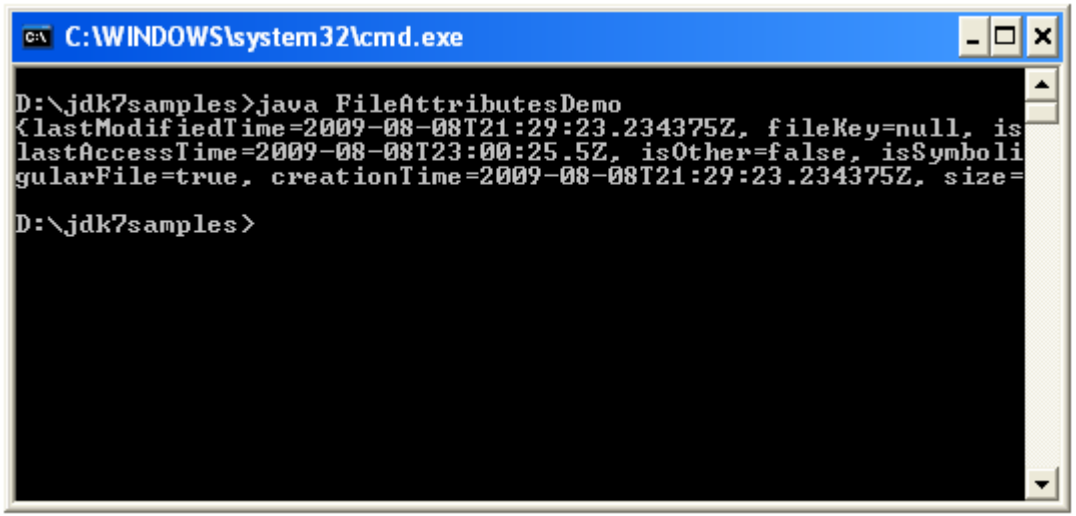
The `FileRef` Interface

```
Map<String,?> readAttributes(String attributes,
     LinkOption... options) throws IOException

<V extends FileAttributeView> V getFileAttributeView(
     Class<V> type, LinkOption... options) throws IOException

InputStream newInputStream(OpenOption... options)
     throws IOException

OutputStream newOutputStream(OpenOption... options)
     throws IOException
```

The `FileRef` Interface

```
import java.nio.file.*;
import java.io.*;
import java.util.*;


public class FileAttributesDemo
{
  public static void main(String args[])
  {
    try
    {
      Path path = Paths.get(
        "c:\\Program Files\\Java\\\jdk1.7.0\\bin\\java.exe");
      Map map = path.readAttributes("*");
      System.out.println(map);
    }
    catch(IOException e) {e.printStackTrace();}
  }
}
```

The `FileRef` Interface



Checking Files

- You can check a file by calling the `checkAccess` method on its `Path` object.

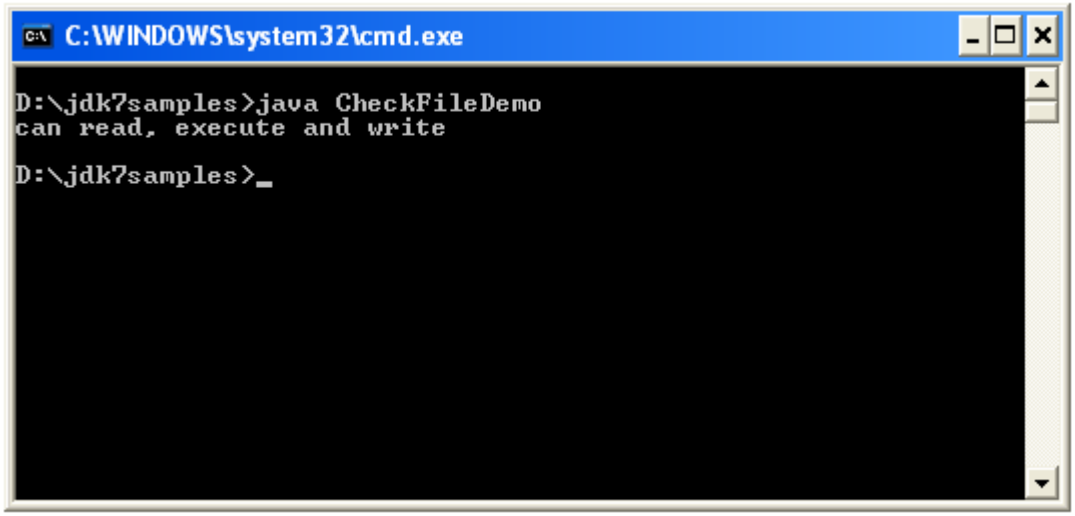
```
public abstract void checkAccess(AccessMode... modes)
                                                throws IOException
```

If the length of modes is 0 then the performed check is of the file very existence. Alternatively, we can pass any combination of the enum AccessMode possible values: EXECUTE, WRITE and READ.

Checking Files

```
import java.nio.file.*;
import java.io.*;
import java.util.*;
import static java.nio.file.AccessMode.*;


public class CheckFileDemo
{
  public static void main(String args[])
  {
    try
    {
      Path path = Paths.get(
         "c:\\Program Files\\Java\\jdk1.7.0\\bin\\java.exe");
      path.checkAccess(READ,EXECUTE,WRITE);
      System.out.println("can read, execute and write");
    }
    catch(IOException e) {e.printStackTrace();}
  }
}
```

Checking Files



Comparing Paths

- It is possible to compare two paths in order to determine whether they point at the same file using the `isSameFile()` method.

```
public abstract boolean isSameFile(Path other)
                                                throws IOException
```

The `delete()` Method

- Calling this method the file the Path object refers will be deleted. If for any reason this deletion fails (e.g. the file doesn't exist) then the method throws an exception.

```
public abstract void delete() throws IOException
```

The `copyTo()` Method

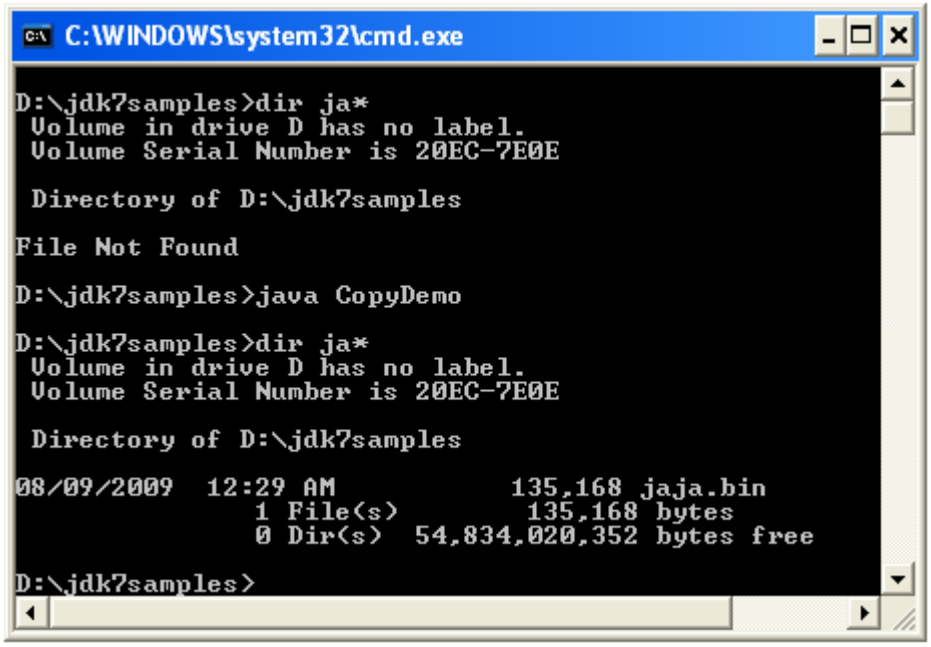
- Calling this method we can copy a file (or directory). If the target already exists then the method fails (unless the REPLACE_EXISTING option is specified).

```
public abstract Path copyTo(Path target,
                            CopyOption... options) throws IOException
```

The `copyTo()` Method

```
import java.nio.file.*;
import java.io.*;
import java.util.*;
import static java.nio.file.AccessMode.*;


public class CopyDemo
{
  public static void main(String args[])
  {
    try
    {
      Path path = Paths.get(
        "c:\\Program Files\\Java\\jdk1.7.0\\bin\\java.exe");
      Path newPath = Paths.get("d:\\jdk7samples\\jaja.bin");
      path.copyTo(newPath);
    }
    catch(IOException e) {e.printStackTrace();}
  }
}
```

The `copyTo()` Method



The `moveTo()` Method

- Calling this method we can move a file (or directory) from its current location into another one.

```
public abstract Path moveTo(Path target,
                            CopyOption... options) throws IOException
```

The `FileChannel` Class

- Using this class it is possible to randomly access a file's content and read\write its content.

  The `FileChannel` class is an abstract one. We shall actually work with a concrete class that extends it. There are various ways for getting a `FileChannel` object.

The `FileChannel` Class

- The most important methods this class includes are:

```
public long position() throws IOException

FileChannel position(long newPosition) throws IOException

public int read(ByteBuffer ob) throws IOException

public int write(ByteBuffer ob) throws IOException

public FileChannel truncate(long size) throws IOException
```

The `FileChannel` Class

- Getting a new `FileChannel` object can be done by calling one of the available static factory methods.

```
public static FileChannel open(Path file,
     OpenOption... options) throws IOExceptionn

public static FileChannel open(Path file,
     Set<? extends OpenOption> options,
     FileAttribute<?>... attrs) throws IOException
```

The `FileChannel` Class

- Other classes include methods for getting a `FileChannel` object... one of them is the `Path` class that includes the `newByteChannel`.

```
public abstract SeekableByteChannel newByteChannel
                (OpenOption... options) throws IOException
```

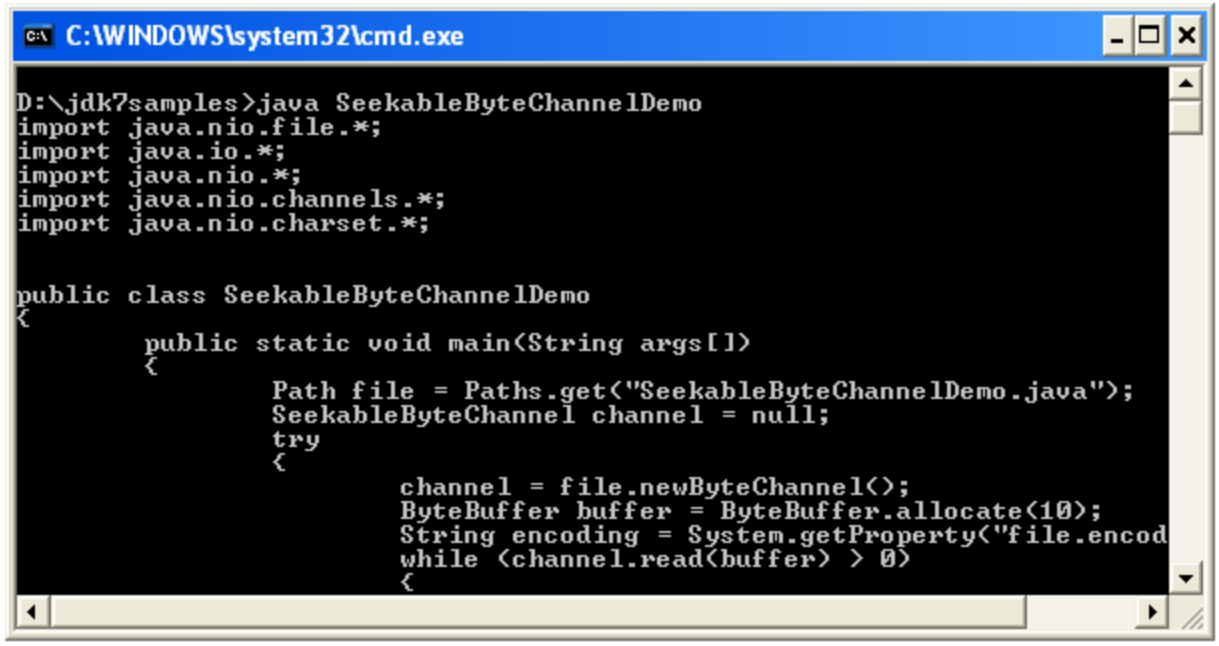
The `FileChannel` Class

```
import java.nio.file.*;
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;


public class SeekableByteChannelDemo
{
  public static void main(String args[])
  {
    Path file = Paths.get("SeekableByteChannelDemo.java");
    SeekableByteChannel channel = null;
    try
    {
       channel = file.newByteChannel();
       ByteBuffer buffer = ByteBuffer.allocate(10);
       String encoding = System.getProperty("file.encoding");
```

The `FileChannel` Class

```
            while (channel.read(buffer) > 0)
            {
               buffer.rewind();
               System.out.print(Charset.forName(encoding).decode(buffer));
               buffer.rewind();
            }
         }
         catch (IOException e) {e.printStackTrace();}
         finally {if (channel != null)
         try{channel.close();}catch(IOException e){e.printStackTrace();}}
      }
   }
```

The `FileChannel` Class



The `Attributes` Class

- The `Attributes` class provides various static convenience methods for reading and setting file's attributes.

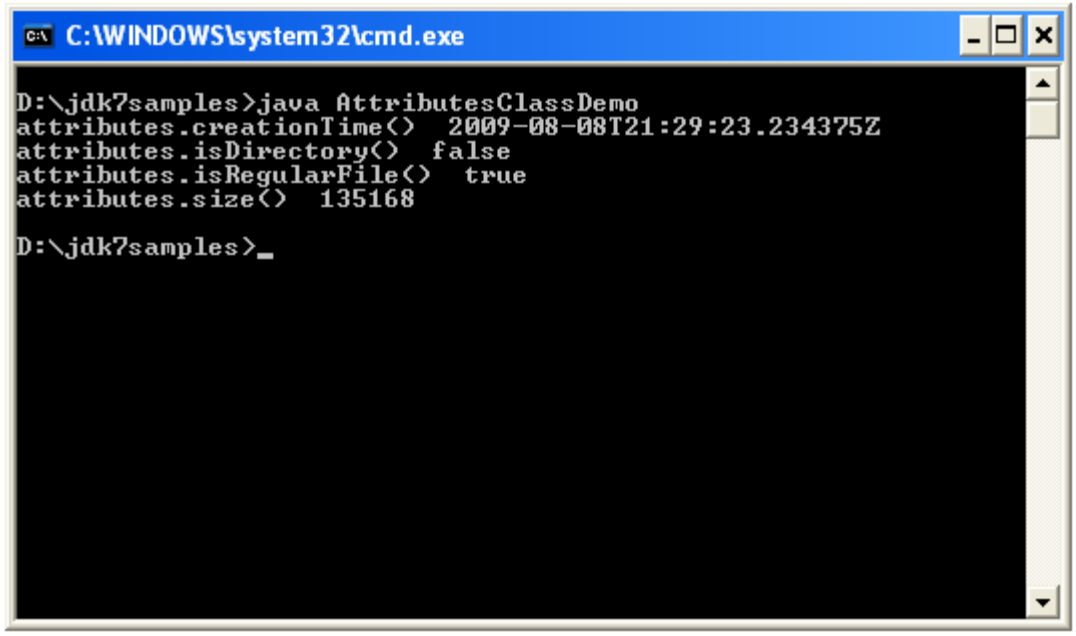
The `Attributes` Class

```
import java.nio.file.*;
import java.io.*;
import java.util.*;
import java.nio.file.attribute.*;


public class AttributesClassDemo
{
  public static void main(String args[])
  {
    try
    {
      Path path = Paths.get(
        "c:\\Program Files\\Java\\jdk1.7.0\\bin\\java.exe");
      BasicFileAttributes attributes =
         Attributes.readBasicFileAttributes(path);
      System.out.println("attributes.creationTime()  " +
         attributes.creationTime());
```

The `Attributes` Class

```
      System.out.println("attributes.isDirectory()  " +
         attributes.isDirectory());
      System.out.println("attributes.isRegularFile()  " +
         attributes.isRegularFile());
      System.out.println("attributes.size()  " +
         attributes.size());
    }
    catch(IOException e)
    {
      e.printStackTrace();
    }
  }
}
```

The `Attributes` Class



The `FileSystem` Class

- The `FileSystem` class describes the files system.

  This class was added in JDK 1.7. This class includes various methods we can call in order to get information about the file system.

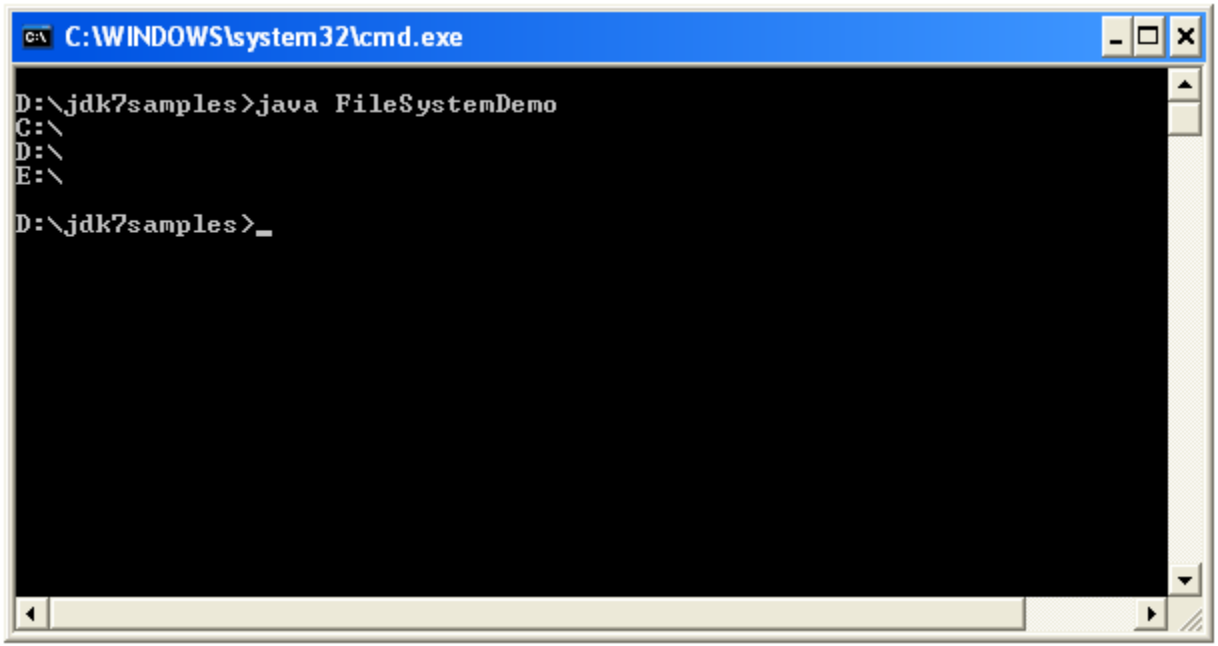
- One of the ways for getting a `FileSystem` object is by calling the `FileSystems.getDefault()` method.

  As with Paths that serves as a factory class for `Path`, `FileSystems` serves as a factory class for `FileSystem`.

The `FileSystem` Class

```
import java.nio.file.*;
import java.io.*;


public class FileSystemDemo
{
        public static void main(String args[])
        {
                FileSystem system = FileSystems.getDefault();
                Iterable<Path> dirs = system.getRootDirectories();
                for (Path path: dirs)
                {
                        System.out.println(path);
                }
        }
}
```

The `FileSystem` Class



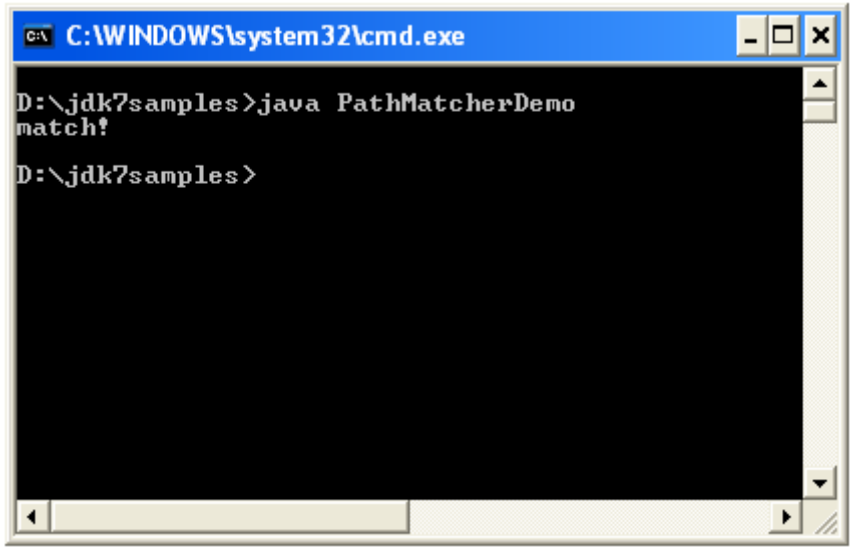
The `PathMatcher` Class

- The `FileSystem` class includes the `getPathMatcher()` method through which you can get a `PathMatcher` object, that describes a matching rule for files and directories filtering

The `PathMatcher` Class

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


public class PathMatcherDemo
{
  public static void main(String args[])
  {
    PathMatcher matcher = FileSystems.getDefault().
      getPathMatcher("glob:*.{java,txt}");
    Path path = Paths.get("PathMatcherDemo.java");
    if(matcher.matches(path))
    {
      System.out.println("match!");
    }
  }
}
```

The `PathMatcher` Class



Files Iteration

- We can iterate all files by calling the `Files.walkFileTree()` static method passing over the Path object representing the starting point and a `FileVisitor` instance.

  The `FileVisitor` is an interface. We should define a class that implements that interface and instantiate it. We can define a class that extends `SimpleFileVisitor`, a class that implements this interface and was already defined.

Files Iteration

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


public class FilesIterationDemo
{
  public static class MyVisitor extends SimpleFileVisitor<Path>
  {
    public FileVisitResult visitFile(Path file,
      BasicFileAttributes attr)
    {
      System.out.println("visiting file "+file.getName());
      return CONTINUE;
    }
```

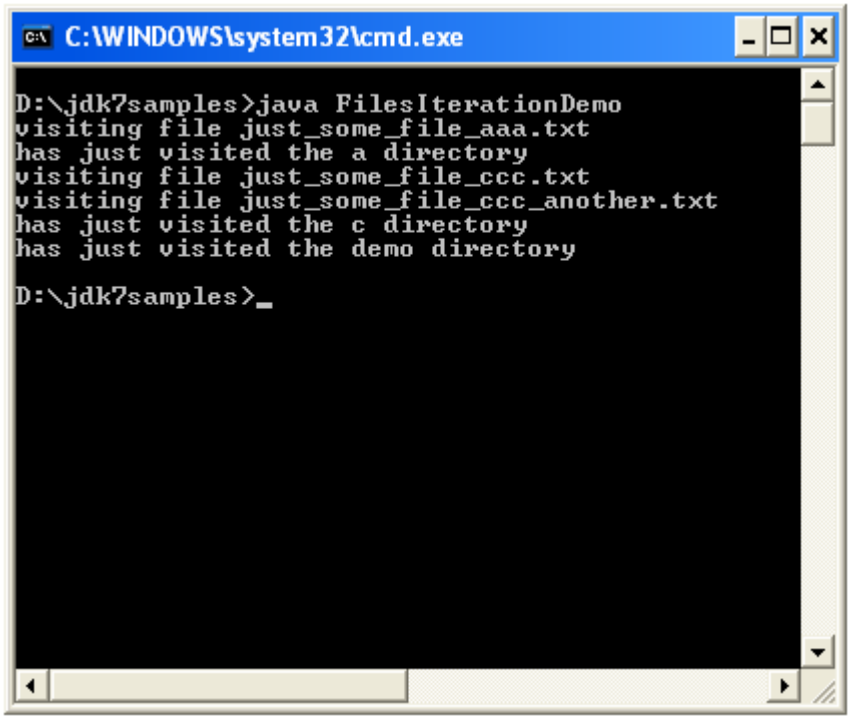
Files Iteration

```
public FileVisitResult postVisitDirectory(Path dir,
  IOException exc)
{
  System.out.println("has just visited the " +
    dir.getName()+" directory");
  return CONTINUE;
}
public FileVisitResult preVisitDirectoryFailed(Path dir,
  IOException exc)
{
  System.out.println("failed to visit directory "+dir.getName());
  return CONTINUE;
}
public FileVisitResult visitFileFailed(Path file,
  IOException exc)
{
  System.out.println("failed to visit file "+file.getName());
  return CONTINUE;
}
```

Files Iteration

```
  }

  public static void main(String args[])
  {
    Path path = Paths.get("d:\\jdk7samples\\content\\demo");
    Files.walkFileTree(path,new MyVisitor());
  }
}
```

Files Iteration



The WatchService API

- Using this API we can register a directory (or directories) telling the service which types of events are of your interest (e.g. file creation, file deletion etc.) and when the service detects any of these events it is forward to the registered process.

  We will usually have the registered process using a thread or even a pool of threads for getting these notifications.

The WatchService API

The following code registers a specific path with the watch service.

```
WatchService service = FileSystems.getDefault().newWatchService();

Path path = Paths.get(...);

WatchKey key = null;
try
{
    key = path.register(service, ENTRY_CREATE,
        ENTRY_DELETE, ENTRY_MODIFY);
}
catch (IOException x) { System.err.println(x); }
```

The WatchService API

The following code will most likely be within a separated thread. Calling the `take()` method stops the thread till a key is delivered by the watch service.

```
try
{
    key = service.take();
}
catch (InterruptedException x)
{
    return;
}
```

I\O Streams

I\O - Introduction

- A stream is a flow of data. The Java programming language has a huge range of classes that their instances represent streams.
- The stream can also be thought of as a pipe through which the data flows.

Java has a ‘jungle’ of various different stream classes. Different classes for different purposes. When learning this module try to focus on the concept (less on the details).

I\O Streams- Categories

	Input		Output	
	byte	char	byte	char
node streams	InputStream FileInputStream PipedInputStream…	Reader FileReader PipedReader …	OutputStream, FileOutputStream, PipedOutputStream…	Writer FileWriter PipedWriter …
filter streams	BuferedInputStream DataInputStream ObjectInputStream…	FilterReader InputStreamReader BufferedReader ...	BuferedOutputStream DataOutputStream ObjectOutputStream ...	FilterWriter, InputStreamWriter BufferedWriter …

The different streams can be categorized into different categories:

Input & Output:

Either the stream is used for input or it is used for output.

Byte & Char:

Conceptually, each stream can be treated either as a stream through which the flowing data are bytes or as a stream through which the flowing data are chars.

Node & Filter:

Streams can also be categorized by having a specific data source\destination (node streams) or by having another stream connected (filter streams).

Byte & Char streams

- The streams that java supports are categorized into two different categories: byte streams and character streams.
- Input and Output of character data is handled by readers and writers.
- Input and Output of byte data is handled by input streams and output streams.

The flowing data is always bytes. The difference between byte streams and char streams in solely conceptually. The methods that can be invoked on a char stream allow you writing\reading character data (strings, chars etc…). The methods that can be invoked on a byte stream allow you writing\reading byte data (integers, floating-point numbers etc…).

Readers and Writers are streams that their flowing data is chars. These kind of streams can be created by instantiating classes that extend the Reader and Writer classes.

Input streams and Output streams are streams that their flowing data is bytes. These kind of streams can be created by instantiating the classes that extend the InputSteam and OutputStream classes.

The InputStream Abstract Class

- The super class of all the input streams. It includes the following methods:

```
public abstract int read()
public int read(byte []vec)
public int read(byte []vec, int offset, int length)
public void close()
```

It is important knowing the methods that were declared within the InputStream class. These methods can be invoked on every input stream (every object that was instantiated from a class that extends InputStream).

The first read method returns an int which contains a byte value read from the stream. The first read method returns –1 when the end of the stream is reached.

The other two read methods read the bytes from the stream into an array of bytes and return the number of bytes that they read. For efficiency reasons these two methods are usually preferred comparing the first read method that read one byte at a time.

The close() method Closes this input stream and releases any system resources associated with it. If the stream has other streams connected to it, then invoking the close() method invokes the close() methods of the other streams.

The InputStream Abstract Class

```
public int available()
public void skip (long n)
public boolean markSupported()
void mark(int readLimit)
void reset()
```

The available() method returns the number of bytes that are immediately available to be read from the stream.

The method markSupported() returns true or false according to the question whether the stream support (or doesn't support) the mark & reset mechanism (methods).The mark() method is used to indicate that the current point in the stream should be noted and a buffer which is big enough for at least the argument the method received will be allocated. This argument specifies the number of bytes that can be re-read by calling the reset() method. Calling the reset() method returns the input stream to the point that was marked.

The OutputStream Abstract Class

- The super class of all output streams. It includes the following methods:

```
public abstract void write(int val)
public void write(byte[] vec)
public void write(byte[] vec, int offset, int lengt)
public void close()
public flush()
```

It is important knowing the methods that were declared inside the OutputStream class. These methods can be called on every output stream (every object that was instantiated from a class that extends OutputStream).

The three write methods are used to write bytes through the output stream. The first write() method writes the first byte of the int value it receives.

The close() method Closes this output stream and releases any system resources associated with it. If the stream has other streams connected to, then the close() methods of the other streams are called as well.

Some of the output streams accumulates the bytes before writing them. The flush() method forces the output stream writing the bytes.

Writing\Reading To\From Files

- The I\O classes include specific classes that describe streams to\from files.
- The following example presents a stand alone application that copies a given file.
- Note the `FileInputStream` and `FileOutputStream` classes that extend `InputStream` and `OutputStream` respectively.

Since the FileOutputStream and FileInputStream extend the OutputStream and InputStream classes, the methods that were describes in the last slides (the ones that covered InputStream & OutputStream) can be called on FileOutputStream and FileInputStream instances as well.

Writing\Reading To\From Files

```
import java.io.*;
public class CopyFile
{
    public static void main(String args[])
    {
        FileInputStream fis = null;
        FileOutputStream fos = null;
        try
        {
            fis = new FileInputStream(args[0]);
            fos = new FileOutputStream(args[1]);
            int data;
            data = fis.read();
```

Since the FileOutputStream and FileInputStream extend the OutputStream and InputStream classes, the methods that were describes in the last slides can be invoked on FileOutputStream and FileInputStream instances too.

This example gets the names of the two files in its invocation from the command line.

Writing\Reading To\From Files

```
            while(data!=-1)
            {
                  fos.write(data);
                  data = fis.read();
            }
            fos.flush();
        }
        catch(IOException e)
        {...}
        finally
        {
            if(fos!=null) try{fos.close()} catch(IOException e) {}
            if(fis!=null) try{fis.close()} catch(IOException e) {}
        }
    }
```

Since the FileOutputStream and FileInputStream extend the OutputStream and InputStream classes, the methods that were describes in the last slides can be invoked on FileOutputStream and FileInputStream instances too.

This example gets the names of the two files in its invocation from the command line.

The Reader Methods

- The `Reader` class includes the following methods:

```
public int read()
public int read(char[] buffer)
public int read(char[] buffer, int offset, int lengt)
public void close()
public boolean ready()
public void skip(long num)
public boolean markSupported()
public void mark(int limit)
public void reset()
```

The read methods enable reading the character data from the given reader. The first read() method returns an int which contains the unicode value read from the stream or –1, which indicates the end of the stream.

The other two read() methods return the number of bytes read. The characters themselves are filled into the array that its reference was sent as an argument.

The other methods do the same as in input streams.

The Writer Methods

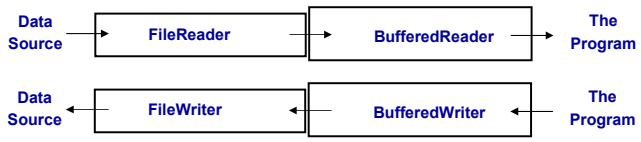
- The `Writer` class includes the following methods:

```
public void write(int c)
public void write(char[] buffer)
public void write(char[] buffer, int offset, int leng)
public void write(String str)
public void write(String str, int offSet, int leng)
public void close()
public void flush()
```

Writer includes similar methods to those you can find in OutputStream.

Streams Chaining

- Programs in Java usually chains a series of streams together to process the data.
- Each stream contributes something to the streams chain.



Each stream in a chain of streams effects in its own way on the overall data flow. The BufferedWrite & BufferedReader, for instance, add a buffer functionality to the data flow (which improves the data flowing).

The next example presents this chaining possibility.

Streams Chaining

```
import java.io.*;
public class CopyFileV2
{
    public static void main(String args[])
    {
        FileReader in = null;
        FileWriter out = null;
        BufferedReader br = null;
        BufferedWriter bw = null;
        try
        {
            in = new FileReader(args[0]);
            out = new FileWriter(args[1]);
```


Streams Chaining

```
    br = new BufferedReader(in);
    bw = new BufferedWriter(out);
    String currentLine = null;
    currentLine = br.readLine();
    while(currentLine!=null)
    {
        bw.write(currentLine,0,
        currentLine.length());
        bw.newLine();
        currentLine = br.readLine();
    }
}
```


Streams Chaining

```
        catch(IOException e)
        {
            System.out.println("exception appened");
        }
        finally
        {
            if(br!=null) try{br.close();} catch(Exception e){ }
            if(bw!=null) try{bw.close();} catch(Exception e){}
        }
    }
}
```


The `InputStreamReader` and `OutputStreamWriter` classes

- The readers and writers flowing data include chars.
- The input streams and output streams flowing data include bytes.
- The `InputStreamReader` and the `OutputStreamWriter` serve as a bridge between characters flowing and bytes flowing.

These classes are used to interface between byte streams and character readers and writers. When instantiating the InputStreamReader and the OutputStreamWriter classes, conversion rules are defined to change between 16 bit Unicode and other platform specific representations.

By default, the conversion rule between input\output streams and characters readers\writers that take place in English-speaking countries is IOS 8859-1.

A list of the supported encoding forms is given at http://java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.html. When the InputStreamReader or the OutputStreamWriter are instantiated the required encoding form can be specified.

The `URL` class

- The `URL` class represents a Uniform Resource Locator, a pointer to a "resource" on the web.
- A resource can be something as simple as a file or a directory, or it can be a reference to a more complicated object, such as a servelt, JSP, ASP or a CGI program.

The URL object can simply represents a file (on this computer or on another computer on the web).

The `URL` class

- Calling the `openStream()` on the `URL` object we shall get a reference for an input stream connected directly with the represented resource.

Remember this class for more advanced topics. This class, among other things, enables communicating between an applet and a servlet\JSP\ASP or event a CGI program.

The URL class

```
import java.io.*;
import java.net.*;
public class URLExample
{
    public static void main(String args[])
    {
        InputStream in = null;
        URL url = null;
        try
        {
            url = new URL("http://www.yahoo.com/index.html");
            in = url.openStream();
            int tmp = in.read();
```


The URL class

```
            while(tmp!=-1)
            {
                System.out.print((char)tmp);
                tmp = in.read();
            }
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
        finally
        {
            if (in!=null) try {in.close();} catch(Exception e) {}
        }
}
```


Object Serialization

- Java enables reading\writing an object to a stream.
- The written\read object must be `Serializable`.
- The written\read object's class should include the `serialVersionUID` static variable declaration.
- When an object can be stored to disk we can describe the object as a *persistent capable* one.

The Serializable interface that must be implemented in a class from which the written\read object was instantiated, has no methods. The Serializable interface serves as a “marker” indicating that instances from a specific class can be serialized (are persistent capable).

The serialVersionUID static variable should be declared within the class from which the written\read object was instantiated. This variable (of type long) includes “kind of” a version number of this class. This version number is included within the stream of bytes created when the written/read object is serialized. If we don't include this variable in our class declaration it will be automatically added for us.

Object Serialization

- When an object is serialized, only the data in its instance variables is preserved. Methods and static variables are not part of the serialized stream.
- When a data member of the serialized object is also a `Serializable` object then it is also serialized. The whole objects graph is serialized.

If, for instance, the head of a linked list is serialized then all of the other objects in that list will be serialized as well. Reading back the object that was serialized will reconstruct the whole graph of objects we had.

It is possible marking one (or more) of the object's data members as 'transient' and prevent that data member from being serialized.This might be a solution for cases in which the given data member isn't Serializable.

Object Serialization

- The following code writes an object to specific file.

```
..
ObjectOutputStream oos = new ObjectOutputStream( new
            FileOutputStream("birthday.ser"));
oos.writeObject(new Date());
..
```

It is common to name the file to which the object is serialized, with a name that has the ".ser" extension.

Object Serialization

- The following code reads an object from a specific file.

```
..
ObjectIntputStream ois = new ObjectIntputStream( new
           FileInputStream("birthday.ser"));
Date dat = (Date)ois.readObject();
..
```

Note that the readObject() method returns a reference which its type is Object. Therefore, an explicit casting is needed. Since we know the read object we can cast its reference to Date.

Object Serialization

- The variables we don't want to include within the serialization process should be marked with `transient`.

Note that the readObject() method returns a reference which its type is Object. Therefore, an explicit casting is needed. Since we know the read object we can cast its reference to Date.

The `File` Class

- Instantiating the class `File`:

```
File file = new File("myKkk.txt");
```

- Once we have a `File` instance we can call various methods on it:

```
public String getName()
public String getParent()
public String getAbsolutePath()
public void renameTo(String str)
ublic boolean canWrite()
...
```

The class File has several useful constructors that allow us instantiating it in different ways.

The `RandomAccessFile` Class

- The `RandomAccessFile` enables to access a file without reading it from its beginning.
- The `RandomAccessFile` doesn't belong to the input\output streams hierarchy neither to the reader\writer hierarchy.

The `RandomAccessFile` Class

- The main methods this class includes are:

```
public long getFilePointer()
public void seek(long position)
public long length()
public int read()
```

Using the RandomAccessFile you can access a file at any location and read/write… as well as move to another location within that file… backward & forward.

The `Path` Class

- The `Path` class is used for representing paths in the files system.
  Object of type 'Path' can also represent a path that doesn't exist.
- Each object instantiated from `Path` contains the file name and a list of all directories that construct the path.
  Using an object of type Path we can examine, locate and manipulate files.

The Path class was introduced in JDK 7.

The `Path` Class

- The `Path` object isn't system independent. We cannot compare a `Path` object constructed on one operation system with a `Path` object constructed on another.

  Even when the directory structure is the same, there are differences between operation systems in how the path looks.

The `Path` Class

- Once we have a `Path` object we can manipulate it in various ways.

  We can append other paths to it, extract pieces of it, compare it with others etc. It is also possible to check the existence of the file the path refers to, create the file if it still doesn't exist, open it, delete it, change its permissions etc.

Getting a `Path` Object

- The `Paths` class includes two get static factory methods allowing us to get a `Path` object.

```
public static Path get(String path)
public static Path get(URI uri)
```

Path Operations

- The `Path` class includes methods that allow us to perform various operations on it.

  Those operations include the capability to obtain information about the path, access its elements, extract parts of it and convert it into something else.

- The available methods treat the paths a Path object holds as if they are indexed starting with 0 for the top element ending with n-1 for the last one.

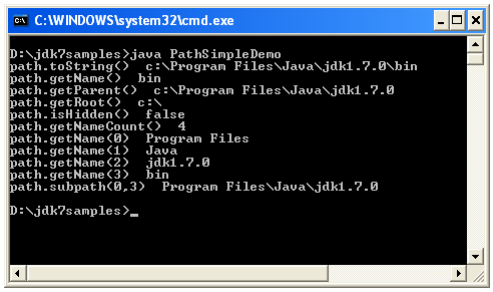
Path Operations

```
import java.nio.file.*;
import java.io.*;


public class PathSimpleDemo
{
  public static void main(String args[])
  {
    Path path = Paths.get("c:\\Program Files\\Java\\jdk1.7.0\\bin");
    System.out.println("path.toString()  "+path.toString());
    System.out.println("path.getName()  "+path.getName());
    System.out.println("path.getParent()  "+path.getParent());
    System.out.println("path.getRoot()  "+path.getRoot());
    try
    {
      System.out.println("path.isHidden()  "+path.isHidden());
    }
    catch(IOException e) {e.printStackTrace();}
```


Path Operations

```
    System.out.println("path.getNameCount()  "+path.getNameCount());
    int count = path.getNameCount();
    for(int i=0;i<count;i++)
    {
      System.out.println("path.getName("+i+")  "+path.getName(i));
    }
    System.out.println("path.subpath(0,3)  "+path.subpath(0,3));
  }
}
```


Path Operations



The `normalize()` Method

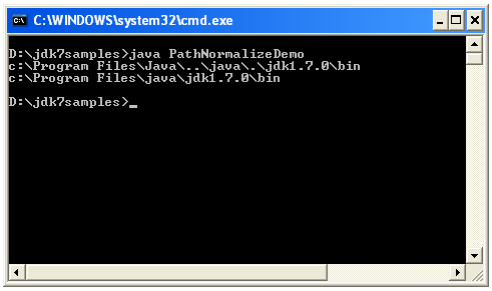
- This method removes redundant parts of the given path.

```
public abstract Path normalize()
```


The `normalize()` Method

```
import java.nio.file.*;
import java.io.*;


public class PathNormalizeDemo
{
  public static void main(String args[])
  {
    Path pathBefore = Paths.get(
        "c:\\Program Files\\Java\\..\\java\\.\\jdk1.7.0\\bin");
    Path pathAfter = pathBefore.normalize();
    System.out.println(pathBefore);
    System.out.println(pathAfter);
  }
}
```


The `normalize()` Method



The `createFile` Method

- Calling the `createFile` method we can create a new file based on its `Path` representation.

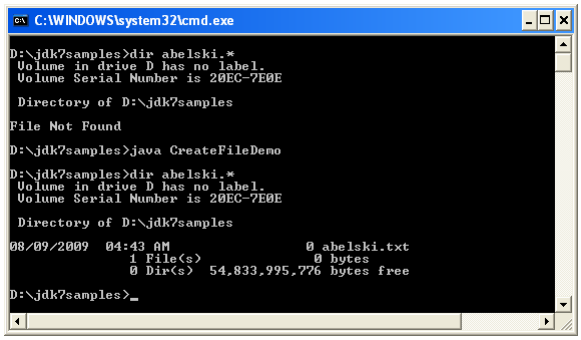
It is possible to check the existence of a file by calling exists() or notExists() as well.

The `createFile` Method

```
import java.nio.file.*;
import java.io.*;


public class CreateFileDemo
{
  public static void main(String args[])
  {
    try
    {
      Path path = Paths.get("d:\\jdk7samples\\abelski.txt");
      path.createFile();
    }
    catch(IOException e)
    {
      e.printStackTrace();
    }
  }
}
```

It is possible to check the existence of a file by calling exists() or notExists() as well.

The `createFile` Method



It is possible to check the existence of a file by calling exists() or notExists() as well.

The `createDirectory` Method

- Calling the `createDirectory` method we can create a new directory based on its Path representation.

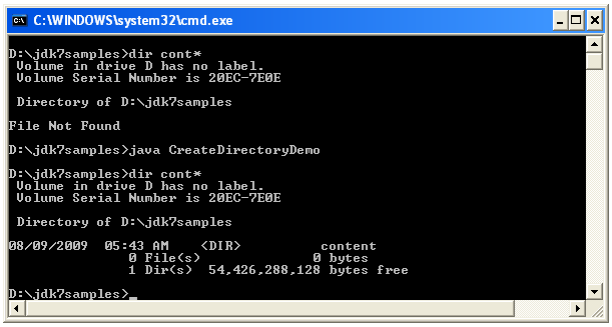
It is possible to check the existence of a file by calling exists() or notExists() as well.

The `createDirectory` Method

```
import java.nio.file.*;
import java.io.*;


public class CreateDirectoryDemo
{
  public static void main(String args[])
  {
    try
    {
        Path path = Paths.get("d:\\jdk7samples\\content");
        path.createDirectory();
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
  }
}
```

It is possible to check the existence of a file by calling exists() or notExists() as well.

The `createDirectory` Method



It is possible to check the existence of a file by calling exists() or notExists() as well.

The `FileRef` Interface

- The `Path` class implements the `FileRef` interface.

  This interface includes various methods that can provide more information about a given file and allow us even getting InputStream and OutputStream of that file.

```
Object getAttribute(String attribute,
     LinkOption... options) throws IOException

void setAttribute(String attribute,
     Object value, LinkOption... options) throws IOException
```

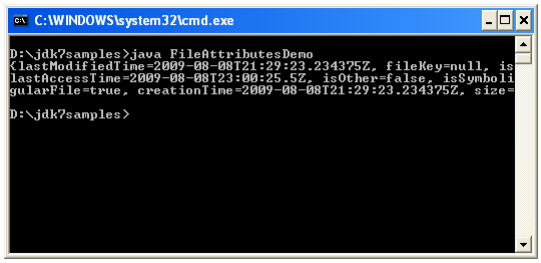

The `FileRef` Interface

```
Map<String,?> readAttributes(String attributes,
     LinkOption... options) throws IOException

<V extends FileAttributeView> V getFileAttributeView(
     Class<V> type, LinkOption... options) throws IOException

InputStream newInputStream(OpenOption... options)
     throws IOException

OutputStream newOutputStream(OpenOption... options)
     throws IOException
```


The `FileRef` Interface

```
import java.nio.file.*;
import java.io.*;
import java.util.*;


public class FileAttributesDemo
{
  public static void main(String args[])
  {
    try
    {
      Path path = Paths.get(
        "c:\\Program Files\\Java\\\jdk1.7.0\\bin\\java.exe");
      Map map = path.readAttributes("*");
      System.out.println(map);
    }
    catch(IOException e) {e.printStackTrace();}
  }
}
```


The `FileRef` Interface



Checking Files

- You can check a file by calling the `checkAccess` method on its `Path` object.

```
public abstract void checkAccess(AccessMode... modes)
                                        throws IOException
```

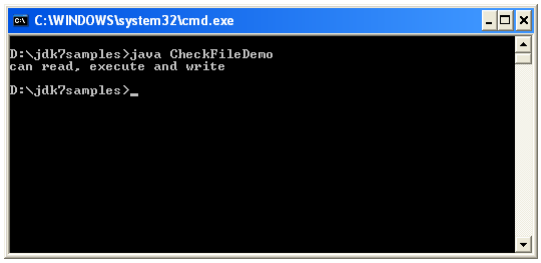
If the length of modes is 0 then the performed check is of the file very existence. Alternatively, we can pass any combination of the enum AccessMode possible values: EXECUTE, WRITE and READ.

It is possible to check the existence of a file by calling exists() or notExists() as well.

Checking Files

```
import java.nio.file.*;
import java.io.*;
import java.util.*;
import static java.nio.file.AccessMode.*;


public class CheckFileDemo
{
  public static void main(String args[])
  {
    try
    {
      Path path = Paths.get(
         "c:\\Program Files\\Java\\jdk1.7.0\\bin\\java.exe");
      path.checkAccess(READ,EXECUTE,WRITE);
      System.out.println("can read, execute and write");
    }
    catch(IOException e) {e.printStackTrace();}
  }
}
```


Checking Files



Comparing Paths

- It is possible to compare two paths in order to determine whether they point at the same file using the `isSameFile()` method.

```
public abstract boolean isSameFile(Path other)
                                        throws IOException
```

It is possible to check the existence of a file by calling exists() or notExists() as well.

The `delete()` Method

- Calling this method the file the Path object refers will be deleted. If for any reason this deletion fails (e.g. the file doesn't exist) then the method throws an exception.

```
public abstract void delete() throws IOException
```

It is possible to check the existence of a file by calling exists() or notExists() as well.

The `copyTo()` Method

- Calling this method we can copy a file (or directory). If the target already exists then the method fails (unless the REPLACE_EXISTING option is specified).

```
public abstract Path copyTo(Path target,
            CopyOption... options) throws IOException
```

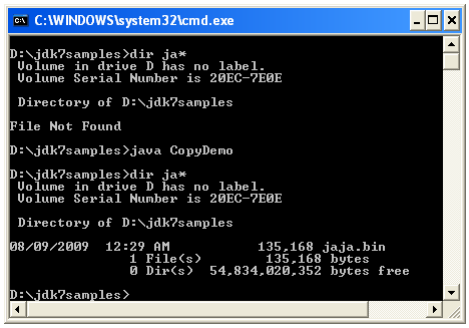
It is possible to check the existence of a file by calling exists() or notExists() as well.

The `copyTo()` Method

```
import java.nio.file.*;
import java.io.*;
import java.util.*;
import static java.nio.file.AccessMode.*;


public class CopyDemo
{
  public static void main(String args[])
  {
    try
    {
      Path path = Paths.get(
        "c:\\Program Files\\Java\\jdk1.7.0\\bin\\java.exe");
      Path newPath = Paths.get("d:\\jdk7samples\\jaja.bin");
      path.copyTo(newPath);
    }
    catch(IOException e) {e.printStackTrace();}
  }
}
```

It is possible to check the existence of a file by calling exists() or notExists() as well.

The `copyTo()` Method



It is possible to check the existence of a file by calling exists() or notExists() as well.

The `moveTo()` Method

- Calling this method we can move a file (or directory) from its current location into another one.

```
public abstract Path moveTo(Path target,
            CopyOption... options) throws IOException
```

It is possible to check the existence of a file by calling exists() or notExists() as well.

The `FileChannel` Class

- Using this class it is possible to randomly access a file's content and read\write its content.

  The `FileChannel` class is an abstract one. We shall actually work with a concrete class that extends it. There are various ways for getting a `FileChannel` object.

It is possible to check the existence of a file by calling exists() or notExists() as well.

The `FileChannel` Class

- The most important methods this class includes are:

```
public long position() throws IOException

FileChannel position(long newPosition) throws IOException

public int read(ByteBuffer ob) throws IOException

public int write(ByteBuffer ob) throws IOException

public FileChannel truncate(long size) throws IOException
```

It is possible to check the existence of a file by calling exists() or notExists() as well.

The `FileChannel` Class

- Getting a new `FileChannel` object can be done by calling one of the available static factory methods.

```
public static FileChannel open(Path file,
     OpenOption... options) throws IOExceptionn

public static FileChannel open(Path file,
     Set<? extends OpenOption> options,
     FileAttribute<?>... attrs) throws IOException
```


The `FileChannel` Class

- Other classes include methods for getting a `FileChannel` object... one of them is the `Path` class that includes the `newByteChannel`.

```
public abstract SeekableByteChannel newByteChannel
            (OpenOption... options) throws IOException
```

It is possible to check the existence of a file by calling exists() or notExists() as well.

The `FileChannel` Class

```
import java.nio.file.*;
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;


public class SeekableByteChannelDemo
{
  public static void main(String args[])
  {
    Path file = Paths.get("SeekableByteChannelDemo.java");
    SeekableByteChannel channel = null;
    try
    {
       channel = file.newByteChannel();
       ByteBuffer buffer = ByteBuffer.allocate(10);
       String encoding = System.getProperty("file.encoding");
```

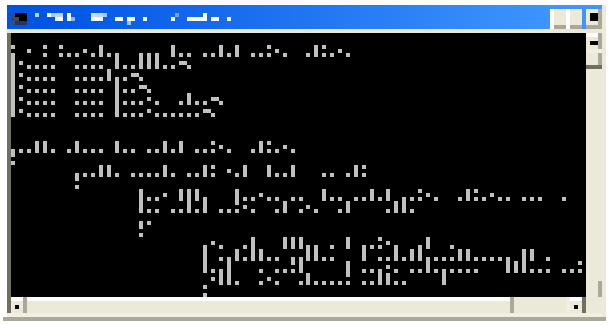
It is possible to check the existence of a file by calling exists() or notExists() as well.

The `FileChannel` Class

```
            while (channel.read(buffer) > 0)
            {
               buffer.rewind();
               System.out.print(Charset.forName(encoding).decode(buffer));
               buffer.rewind();
            }
         }
         catch (IOException e) {e.printStackTrace();}
         finally {if (channel != null)
         try{channel.close();}catch(IOException e){e.printStackTrace();}}
      }
   }
```

It is possible to check the existence of a file by calling exists() or notExists() as well.

The `FileChannel` Class



It is possible to check the existence of a file by calling exists() or notExists() as well.

The `Attributes` Class

- The `Attributes` class provides various static convenience methods for reading and setting file's attributes.

It is possible to check the existence of a file by calling exists() or notExists() as well.

The `Attributes` Class

```
import java.nio.file.*;
import java.io.*;
import java.util.*;
import java.nio.file.attribute.*;


public class AttributesClassDemo
{
  public static void main(String args[])
  {
    try
    {
      Path path = Paths.get(
        "c:\\Program Files\\Java\\jdk1.7.0\\bin\\java.exe");
      BasicFileAttributes attributes =
        Attributes.readBasicFileAttributes(path);
      System.out.println("attributes.creationTime()  " +
        attributes.creationTime());
```

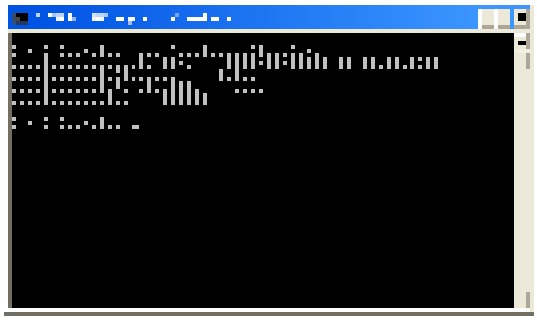
It is possible to check the existence of a file by calling exists() or notExists() as well.

The `Attributes` Class

```
      System.out.println("attributes.isDirectory()  " +
        attributes.isDirectory());
      System.out.println("attributes.isRegularFile()  " +
        attributes.isRegularFile());
      System.out.println("attributes.size()  " +
        attributes.size());
    }
    catch(IOException e)
    {
      e.printStackTrace();
    }
  }
}
```

It is possible to check the existence of a file by calling exists() or notExists() as well.

The `Attributes` Class



It is possible to check the existence of a file by calling exists() or notExists() as well.

The `FileSystem` Class

- The `FileSystem` class describes the files system.
  This class was added in JDK 1.7. This class includes various methods we can call in order to get information about the file system.
- One of the ways for getting a `FileSystem` object is by calling the `FileSystems.getDefault()` method.
  As with Paths that serves as a factory class for `Path`, `FileSystems` serves as a factory class for `FileSystem`.

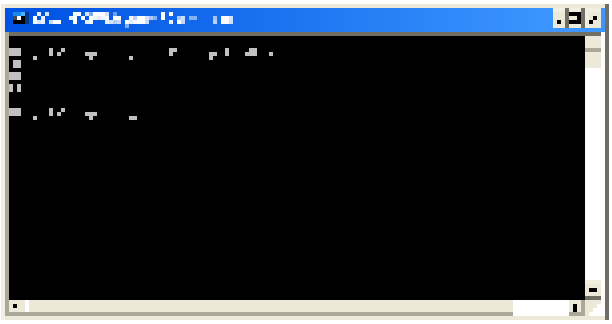
It is possible to check the existence of a file by calling exists() or notExists() as well.

The `FileSystem` Class

```
import java.nio.file.*;
import java.io.*;


public class FileSystemDemo
{
        public static void main(String args[])
        {
                FileSystem system = FileSystems.getDefault();
                Iterable<Path> dirs = system.getRootDirectories();
                for (Path path: dirs)
                {
                        System.out.println(path);
                }
        }
}
```

It is possible to check the existence of a file by calling exists() or notExists() as well.

The `FileSystem` Class



It is possible to check the existence of a file by calling exists() or notExists() as well.

The `PathMatcher` Class

- The `FileSystem` class includes the `getPathMatcher()` method through which you can get a `PathMatcher` object, that describes a matching rule for files and directories filtering

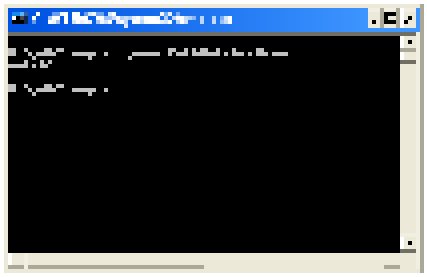
It is possible to check the existence of a file by calling exists() or notExists() as well.

The `PathMatcher` Class

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


public class PathMatcherDemo
{
  public static void main(String args[])
  {
    PathMatcher matcher = FileSystems.getDefault().
      getPathMatcher("glob:*.{java,txt}");
    Path path = Paths.get("PathMatcherDemo.java");
    if(matcher.matches(path))
    {
      System.out.println("match!");
    }
  }
}
```

It is possible to check the existence of a file by calling exists() or notExists() as well.

The `PathMatcher` Class



It is possible to check the existence of a file by calling exists() or notExists() as well.

Files Iteration

- We can iterate all files by calling the `Files.walkFileTree()` static method passing over the Path object representing the starting point and a `FileVisitor` instance.

  The `FileVisitor` is an interface. We should define a class that implements that interface and instantiate it. We can define a class that extends `SimpleFileVisitor`, a class that implements this interface and was already defined.

Files Iteration

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


public class FilesIterationDemo
{
  public static class MyVisitor extends SimpleFileVisitor<Path>
  {
    public FileVisitResult visitFile(Path file,
      BasicFileAttributes attr)
    {
      System.out.println("visiting file "+file.getName());
      return CONTINUE;
    }
```

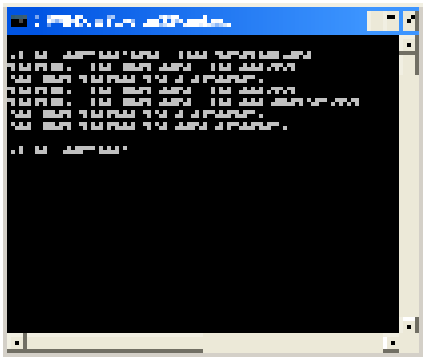

Files Iteration

```
public FileVisitResult postVisitDirectory(Path dir,
  IOException exc)
{
  System.out.println("has just visited the " +
    dir.getName()+" directory");
  return CONTINUE;
}
public FileVisitResult preVisitDirectoryFailed(Path dir,
  IOException exc)
{
  System.out.println("failed to visit directory "+dir.getName());
  return CONTINUE;
}
public FileVisitResult visitFileFailed(Path file,
  IOException exc)
{
  System.out.println("failed to visit file "+file.getName());
  return CONTINUE;
}
```


Files Iteration

```
  }

  public static void main(String args[])
  {
    Path path = Paths.get("d:\\jdk7samples\\content\\demo");
    Files.walkFileTree(path,new MyVisitor());
  }
}
```


Files Iteration



The WatchService API

- Using this API we can register a directory (or directories) telling the service which types of events are of your interest (e.g. file creation, file deletion etc.) and when the service detects any of these events it is forward to the registered process.

  We will usually have the registered process using a thread or even a pool of threads for getting these notifications.

The WatchService API

The following code registers a specific path with the watch service.

```
WatchService service = FileSystems.getDefault().newWatchService();

Path path = Paths.get(...);

WatchKey key = null;
try
{
    key = path.register(service, ENTRY_CREATE,
        ENTRY_DELETE, ENTRY_MODIFY);
}
catch (IOException x) { System.err.println(x); }
```


The WatchService API

The following code will most likely be within a separated thread. Calling the `take()` method stops the thread till a key is delivered by the watch service.

```
try
{
    key = service.take();
}
catch (InterruptedException x)
{
    return;
}
```

