Advanced Object-Oriented Programming

Streams and Files

Dr. Kulwadee Somboonviwat

International College, KMITL

kskulwad@kmitl.ac.th

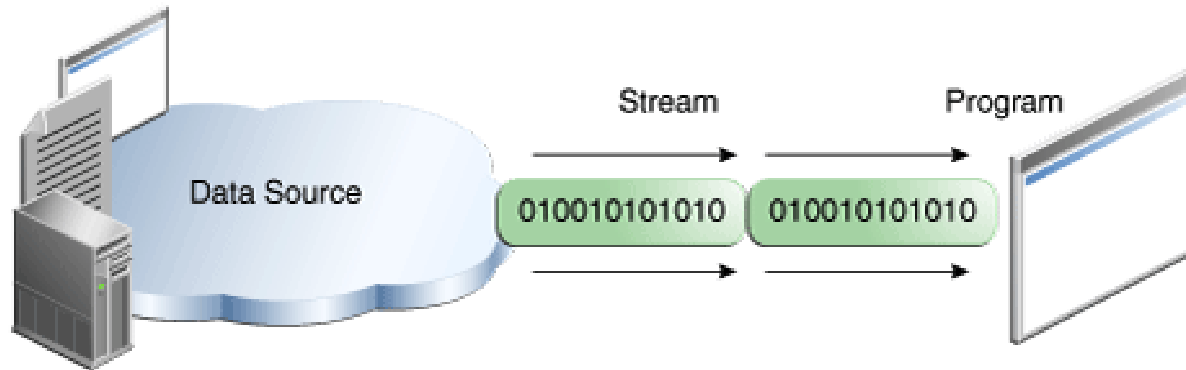
Streams and Files

- Streams
- File I/O
  - Binary I/O
  - Text I/O
  - Object I/O (object serialization)
- Working with files
  - JFileChooser
  - Path and Files classes

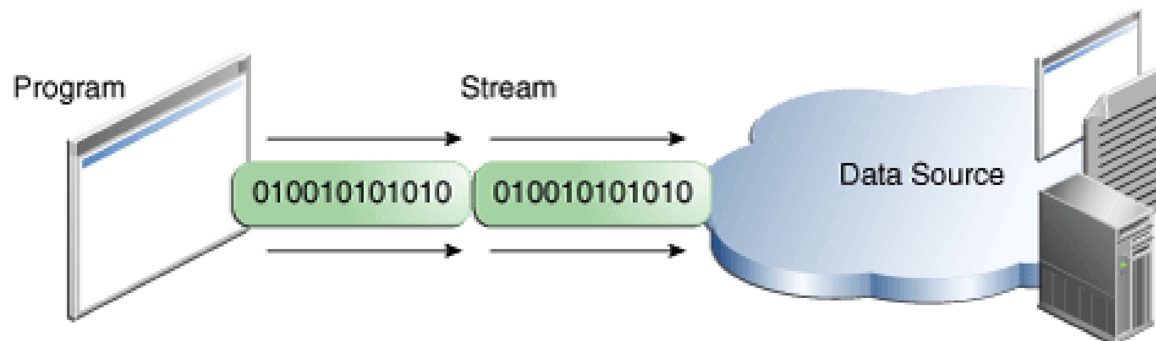
Streams

- A *stream* is a sequence of data items, usually 8-bit bytes: e.g. files, network connections, blocks of memory
- The Java API has two types of streams: an *input stream* and an *output stream*.

An *input stream* is an object from which we can read a sequence of bytes



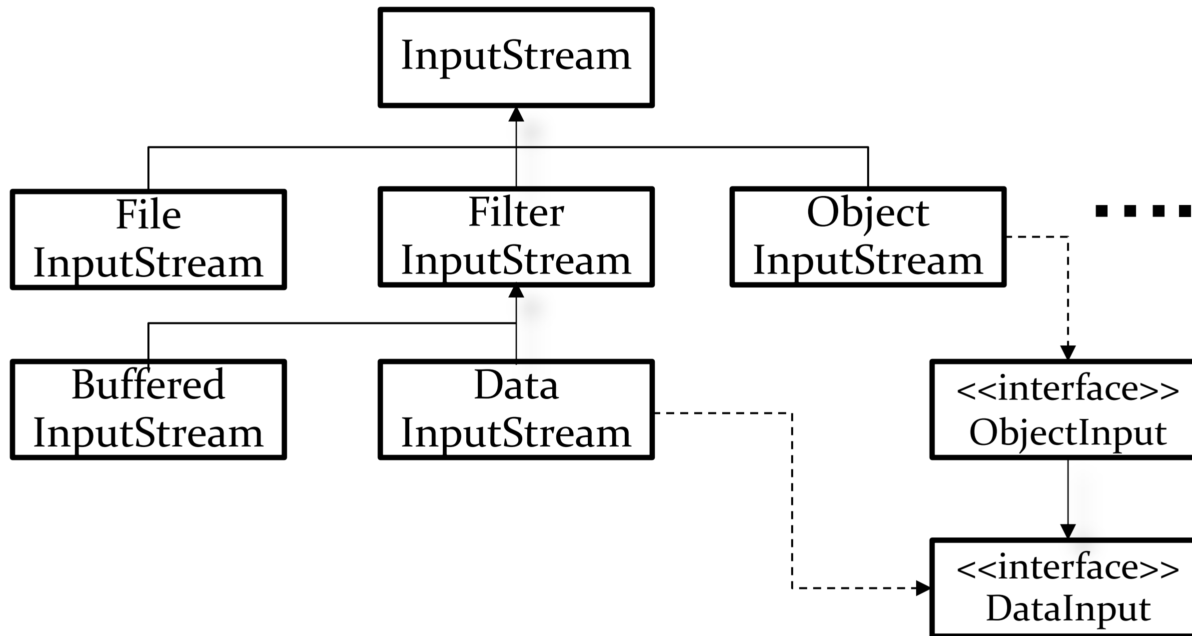
An *output stream* is an object to which we can write a sequence of bytes



Types of Streams supported by Java API

- Byte-oriented Streams
    - InputStream and OutputStream hierarchies
- Unicode characters Streams
    - Reader and Writer hierarchies

InputStream Hierarchy (partial)

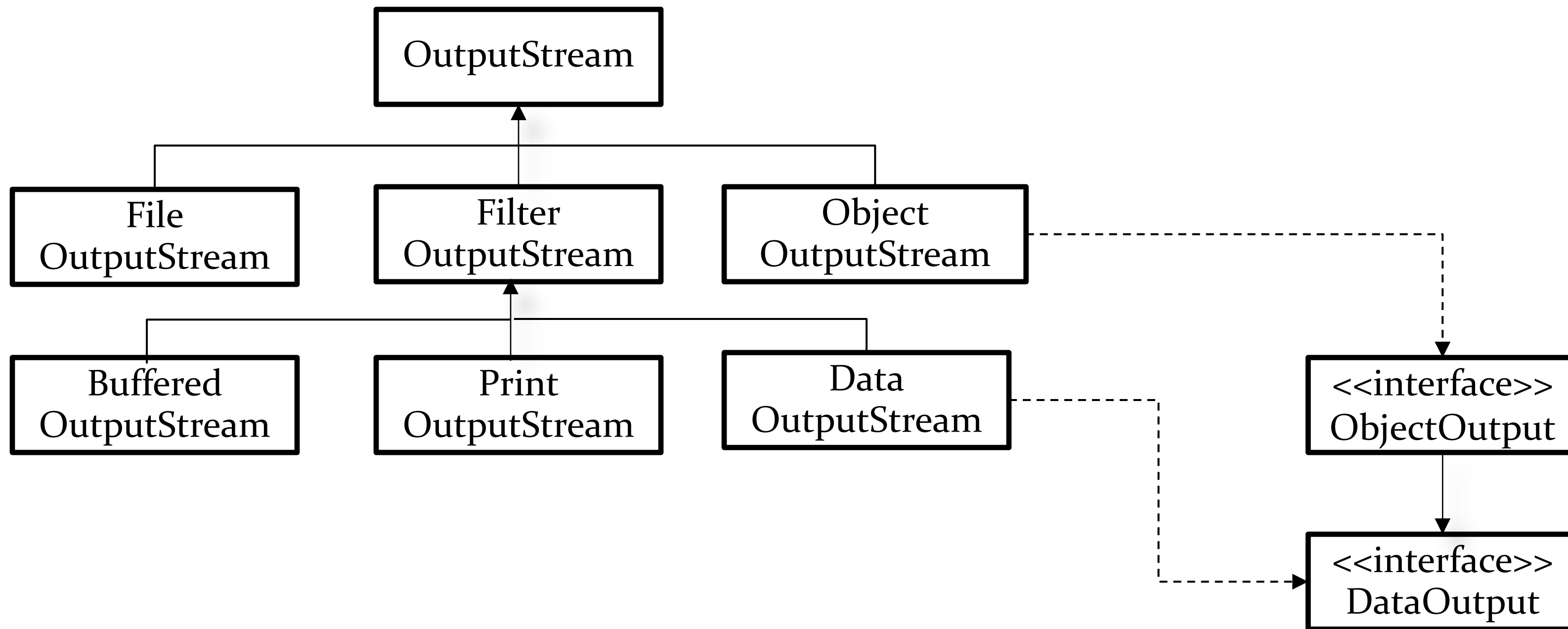


java.io.InputStream has an abstract method:

abstract int read()

The designer of a concrete input stream class overrides this method to provide useful functionality

OutputStream Hierarchy (partial)

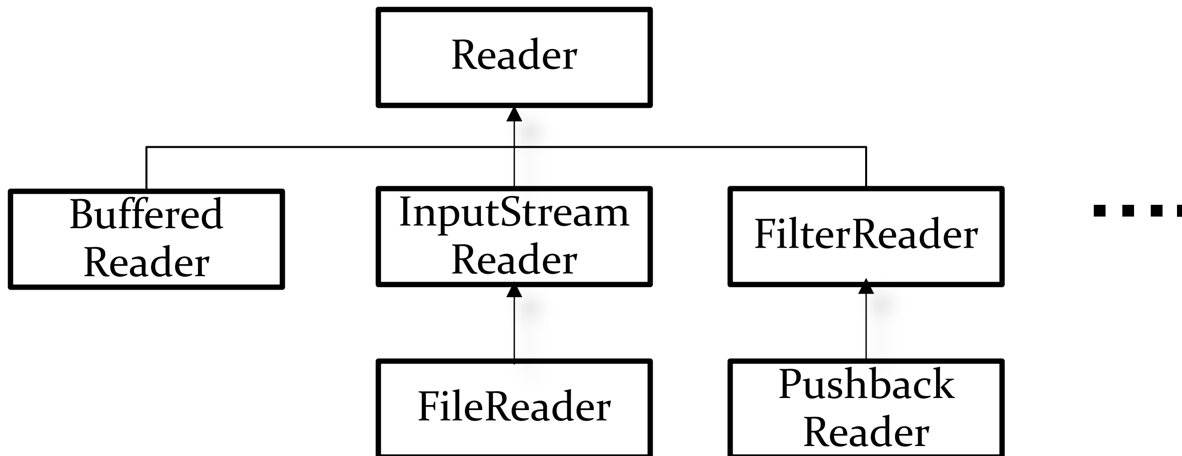


java.io.OutputStream has an abstract method:

abstract void write(int b)

The designer of a concrete output stream class overrides this method to provide useful functionality

Reader Hierarchy (partial)

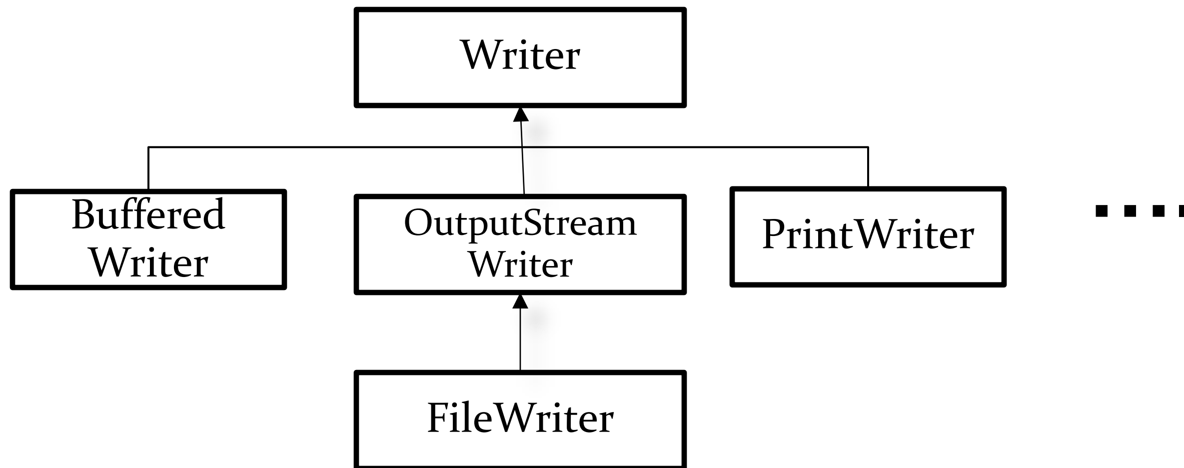


java.io.Reader has an abstract method:

abstract int read()

The designer of a concrete input stream class overrides this method to provide useful functionality

Writer Hierarchy (partial)



java.io.Writer has an abstract method:

abstract void write(int c)

The designer of a concrete output stream class overrides this method to provide useful functionality

Low-level File I/O

- **FileOutputStream** and **FileInputStream** are two stream objects that facilitate file access.
- **FileOutputStream** allows us to output a sequence of bytes; values of data type **byte**.
- **FileInputStream** allows us to read in an array of bytes.

Low-level File Output

```
//set up file and stream
File  outFile   = new File("sample1.data");

FileOutputStream
      outStream = new FileOutputStream( outFile );

//data to save
byte[] byteArray = {10, 20, 30, 40,
                    50, 60, 70, 80};

//write data to the stream
outStream.write( byteArray );

//output done, so close the stream
outStream.close();
```

Low-level File Input

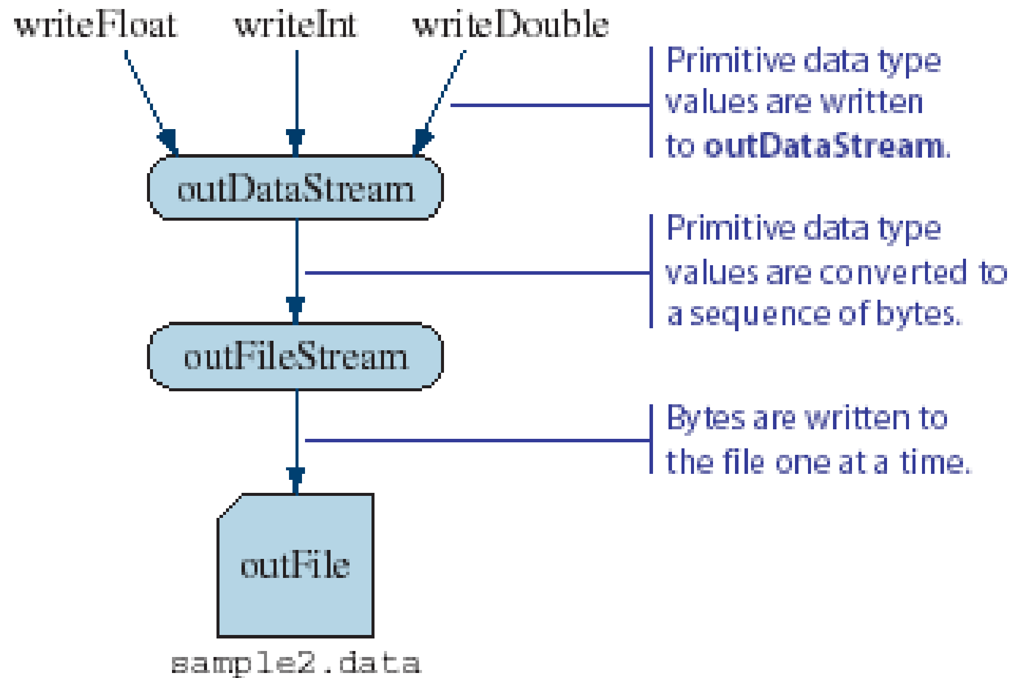
```
//set up file and stream
File            inFile   = new File("sample1.data");
FileInputStream inStream = new FileInputStream(inFile);

//set up an array to read data in
int    fileSize  = (int)inFile.length();
byte[] byteArray = new byte[fileSize];

//read data in and display them
inStream.read(byteArray);
for (int i = 0; i < fileSize; i++) {
       System.out.println(byteArray[i]);
}

//input done, so close the stream
inStream.close();
```

High-level I/O (binary file)

- FileInputStream and FileOutputStream support only reading and writing (from a file) at **byte-level**
- To read/write at higher level (e.g. int, float,..), the Java programmer has to combine the file based streams with some filter based streams:

High-level File Output

```
File             outFile       = new File( "sample2.data" );
FileOutputStream outFileStream = new FileOutputStream(outFile);
DataOutputStream outDataStream = new DataOutputSteam(outFileStream);
```

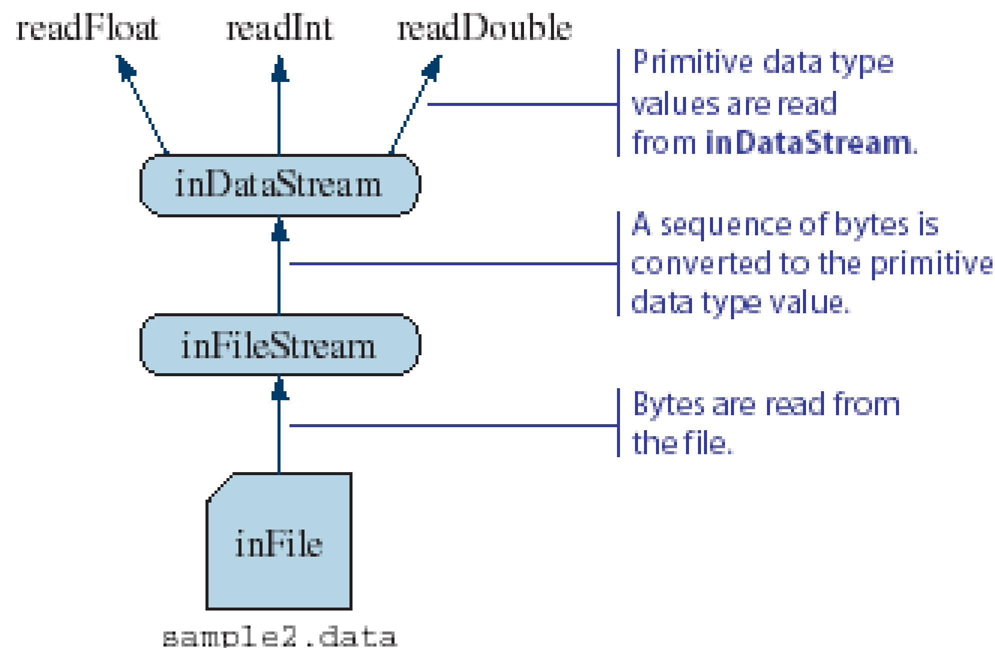


High-level File Output (cont’)

```
import java.io.*;
class TestDataOutputStream {
    public static void main (String[] args) throws IOException {

        . . . //set up outDataStream

        //write values of primitive data types to the stream
        outDataStream.writeInt(987654321);
        outDataStream.writeLong(11111111L);
        outDataStream.writeFloat(22222222F);
        outDataStream.writeDouble(3333333D);
        outDataStream.writeChar('A');
        outDataStream.writeBoolean(true);

        //output done, so close the stream
        outDataStream.close();
    }
}
```

High-level File Input

```
File             inFile       = new File( "sample2.data" );
FileOutputStream inFileStream = new FileOutputStream(inFile);
DataOutputStream inDataStream = new DataOutputSteam(inFileStream);
```



High-level File Input (cont’)

```
import java.io.*;
class TestDataInputStream {
    public static void main (String[] args) throws IOException {

        . . . //set up inDataStream

        //read values back from the stream and display them
        System.out.println(inDataStream.readInt());
        System.out.println(inDataStream.readLong());
        System.out.println(inDataStream.readFloat());
        System.out.println(inDataStream.readDouble());
        System.out.println(inDataStream.readChar());
        System.out.println(inDataStream.readBoolean());

        //input done, so close the stream
        inDataStream.close();
    }
}
```

Text file input and output

- To output data as a string to file, we use a PrintWriter object

- To input data from a textfile, we use a FileReader and BufferedReader classes
  - From Java 5.0 (SDK 1.5), we can also use the Scanner class for inputting textfiles

Text file output

```
import java.io.*;
class TestPrintWriter {
    public static void main (String[] args) throws IOException {

        //set up file and stream
        File outFile = new File("sample3.data");
        FileOutputStream outFileStream
                       = new FileOutputStream(outFile);
        PrintWriter outStream = new PrintWriter(outFileStream);

        //write values of primitive data types to the stream
        outStream.println(987654321);
        outStream.println("Hello, world.");
        outStream.println(true);

        //output done, so close the stream
        outStream.close();
    }
}
```

Text file input

```
import java.io.*;
class TestBufferedReader {

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

        //set up file and stream
        File inFile = new File("sample3.data");
        FileReader fileReader = new FileReader(inFile);
        BufferedReader bufReader = new BufferedReader(fileReader);
        String str;

        str = bufReader.readLine();
        int i = Integer.parseInt(str);

        //similar process for other data types

        bufReader.close();
    }
}
```

Text file input with Scanner class

```
import java.io.*;

class TestScanner {

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

        //open the Scanner
        Scanner scanner = new Scanner(new File("sample3.data"));

        //get integer
        int i = scanner.nextInt();

        //similar process for other data types

        scanner.close();
    }
}
```

Object file input and output

- It is possible to store objects just as easily as you store primitive data values.
- We use ObjectOutputStream and ObjectInputStream to save to and load objects from a file.
- To save objects from a given class, the class declaration must include the phrase implements Serializable. For example,

```
class Person implements Serializable {
  . . .
}
```

Saving Objects

```
File                 outFile
                         = new File("objects.data");

FileOutputStream    outFileStream
                         = new FileOutputStream(outFile);

ObjectOutputStream outObjectStream
                         = new ObjectOutputStream(outFileStream);
```

```
Person person = new Person("Mr. Espresso", 20, 'M');

outObjectStream.writeObject( person );
```

```
account1    = new Account();
bank1       = new Bank();

outObjectStream.writeObject( account1 );
outObjectStream.writeObject( bank1    );
```

Could save objects from the different classes.

Reading Objects

```
File                inFile
                       = new File("objects.data");

FileInputStream     inFileStream
                       = new FileInputStream(inFile);

ObjectInputStream inObjectStream
                       = new ObjectInputStream(inFileStream);
```

```
Person person

      = (Person) inObjectStream.readObject( );
```

Must type cast to the correct object type.

```
Account account1
       = (Account) inObjectStream.readObject( );

Bank    bank1
       = (Bank) inObjectStream.readObject( );
```

Must read in the correct order.

Saving and Loading an Array

- Instead of processing array elements individually, it is possible to save and load the whole array at once.

```
Person[] people = new Person[ N ];
           //assume N already has a value

//build the people array
. . .
//save the array
outObjectStream.writeObject ( people );
```

```
//read the array

Person[ ] people = (Person[]) inObjectStream.readObject( );
```

The JFileChooser Class

- A **javax.swing.JFileChooser** object allows the user to select a file.

```
JFileChooser chooser = new JFileChooser( );

chooser.showOpenDialog(null);
```

To start the listing from a specific directory:

```
JFileChooser chooser = new JFileChooser("C:/JavaPrograms/Ch12");

chooser.showOpenDialog(null);
```

Getting information from JFileChooser

```
int status = chooser.showOpenDialog(null);
if (status == JFileChooser.APPROVE_OPTION) {
    System.out.println("Open is clicked");

} else { //== JFileChooser.CANCEL_OPTION
    System.out.println("Cancel is clicked");
}
```

```
File selectedFile  = chooser.getSelectedFile();
```

```
File currentDirectory = chooser.getCurrentDirectory();
```

Applying a File Filter to JFileChooser

- A *file filter* may be used to restrict the listing in JFileChooser to only those files/directories that meet the designated filtering criteria.
- To apply a file, we define a subclass of the **javax.swing.filechooser.FileFilter** class and provide the **accept** and **getDescription** methods.

```
public boolean accept(File file)

public String getDescription( )
```

* example available at:

http://docs.oracle.com/javase/tutorial/uiswing/examples/components/index.html#FileChooserDemo2

Working with the file system

- The **stream classes** (e.g. DataInputStream, PrintWriter) that we discussed earlier deals with the **content of files**
- Java SE 7 adds two new classes for file system management
  - The Path and Files classes encapsulate the functionality required to work with the file system on the user's machine.

Paths

- A Path is a sequence of directory names, optionally followed by a file name.

```
Path absolute = Paths.get(“/home”, “ann”);
Path relative = Paths.get(“myprog”, “conf”, “user.properties”);
```

- Frequently used methods:

static Path java.nio.file.Paths.get(String first, String... more)

Path java.nio.file.Path.resolve(Path other)

Path java.nio.file.Path.resolve(String other)

Path java.nio.file.Path.relativize(Path other)

Path java.nio.file.Path.getParent()

Path java.nio.file.Path.getFileName()

File java.nio.file.Path.toFile()

Path java.io.File.toPath()

Files

- Common file operations for text file I/O

```
// read the entire file content
byte[] bytes = Files.readAllBytes(path);
// read the file as a sequence of lines
List<String> lines = Files.readAllLines(path, charset);

// write a string to file
Files.write(path, content.getBytes(charset));
// append to file
Files.write(path, content.getBytes(charset), StandardOpenOption.APPEND);
// write a collection of lines
Files.write(path, lines);
```

Files

- Common file operations for binary file I/O

```
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);

Reader in = Files.newBufferedReader(path, charset);
Writer out = Files.newBufferedWriter(path, charset);
```

Files

- Copying, moving, deleting files

```
Files.copy(fromPath, toPath);
Files.move(fromPath, toPath);
Files.delete(path);
Boolean deleted = Files.deleteIfExists(path);
```

- Creating files and directories

```
Files.createDirectory(path);
Files.createDirectories(path);   // also creates intermediate directory
Files.createFile(path);
Files.createTempFile(dir, prefix, suffix);
```

Files

- Getting file information

```
Files.exists(path);
Files.isHidden(path);
Files.isReadable(path);
Files.isWritable(path);
Files.isExecutable(path);
Files.isDirectory(path);
Files.size(path); // return size in bytes

// getting file attributes
BasicFileAttributes attributes = Files.readAttributes(path, BasicFileAttributes.class);
attributes.creationTime();
attributes.lastModifiedTime();
```