Cape Peninsula University of Technology

Software Design 2
SDN260S

Files, Streams & Object Serialization

H. Mataifa

Department of Electronic, Electrical and Computer Engineering

Outline

- Basics of files and streams in Java
- Files and directories
- Binary and character input/output streams
- Creating, reading, and writing files
- Classes for Input/Output:
  - Scanner and Formatter for text file processing
  - FileInputStream and FileOutputStream for file processing
  - ObjectInputStream and ObjectOutputStream for object serialization
- Additional Java **I/O** Classes

Basics of Files and Streams in Java (SE 6)

- **Temporary vs persistent data**:
  - Data stored in variables during program execution is temporary; it is lost when variable leaves scope or program terminates
  - Persistent data in the form of files (e.g. Word, Excel, PowerPoint) is used for long-term retention of data (typically in secondary storage devices, e.g. hard disks, optical disks, flashdrives, etc.)
- **Files** in Java are viewed as *streams of bytes*; the **OS** usually has a marker to indicate end-of-file (i.e. end of stream)
- **File streams**: used to *input/output data into/out of program*, either as:
  - **Bytes**: *byte-based streams: binary data/files*, or
  - **Characters**: *character-based streams: sequence of characters/text files*

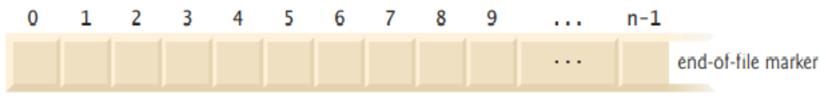


Fig. 1: Java's view of a file of n bytes

Basics of Files and Streams in Java

- A Java program opens an *input/output file* by creating an *object* and associating *byte/character input/output stream* with it
- Java creates three **stream objects** when a program begins executing:
  - **System.in**: standard *input stream*, normally *inputs bytes from keyboard*
  - **System.out**: standard *output stream*, normally *outputs character data to screen*
  - **System.err**: standard *error stream* object, normally *outputs character-based error messages to screen*
- Class **System** provides methods **setIn**, **setOut**, **setErr** to *redirect standard input, standard output, and standard error streams* (to other devices when required)

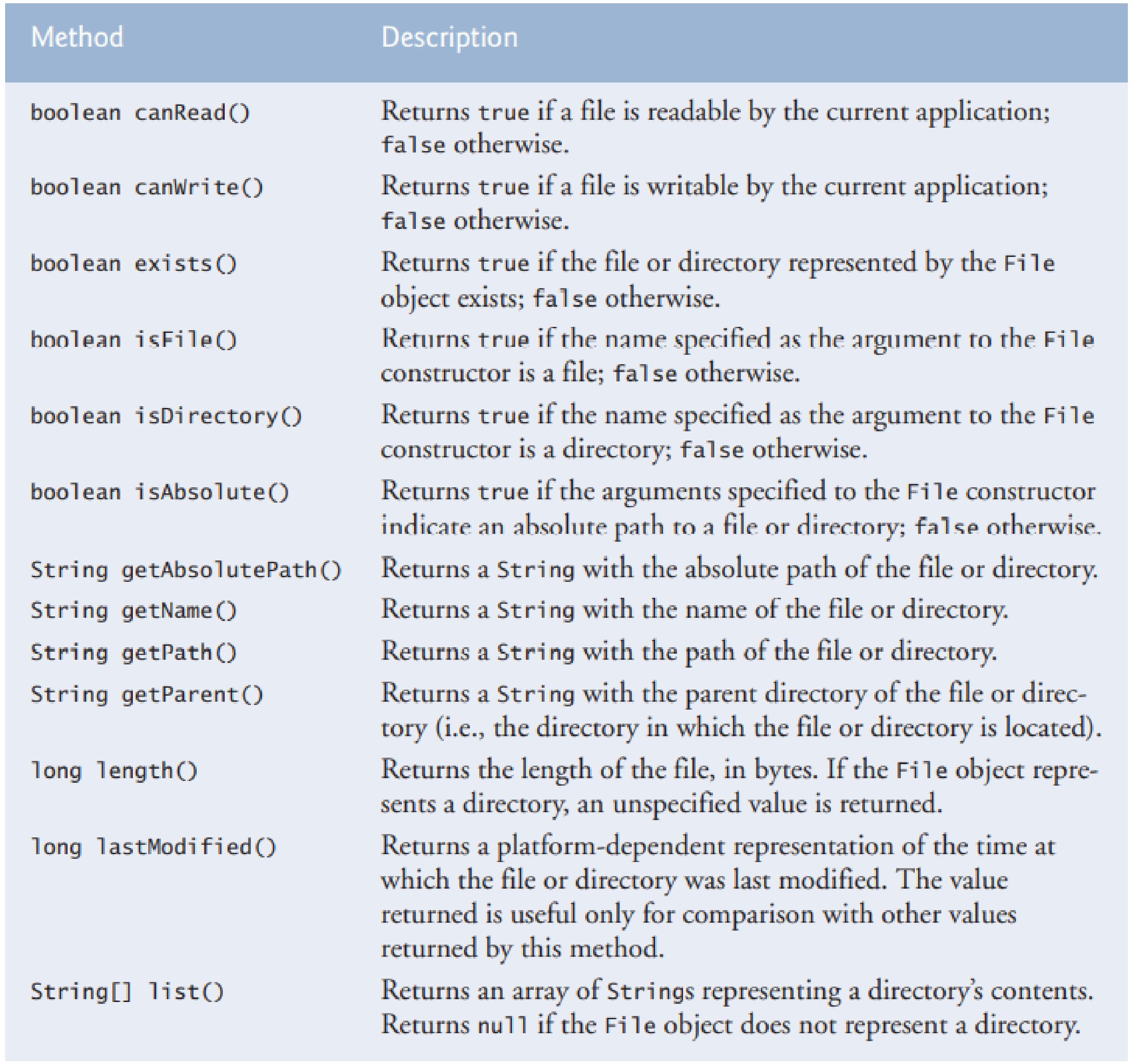
Basics of Files and Streams in Java

- **Main stream classes**:
  - **FileInputStream**: *byte-based input* from a file
  - **FileOutputStream**: *byte-based output* to a file
  - **FileReader**: *character-based input* from a file
  - **FileWriter**: *character-based output* to a file
  - **ObjectOutputStream** (in conjunction with **FileOutputStream**): *object serialization*
  - **ObjectInputStream** (in conjunction with **FileInputStream**): *object deserialization*
  - **Scanner**: *character-based input* from keyboard
  - **Formatter**: *formatted data output* to text-based stream (in a manner similar to *System.out.printf)*

Class **File**

- Class **File** *provides information about files and directories*
- Provides **four constructors**:
  - First with one String argument specifying *name of file/directory to associate with File object*:
    - Name can contain (relative/absolute) path information, as well as file/directory name
  - Second with two String arguments specifying *absolute/relative path*, and *file/directory to associate with File object*
  - Third with File and String arguments, uses an *existing File object that specifies the parent directory* of the *file/directory specified by String argument*
  - Fourth uses a **URI** (Uniform Resource Identifier) object to locate the file
- (File/directory) **path** specifies its location on disk:
  - Absolute path *contains all directories, starting with root directory* (e.g. **C:\** in Windows), that lead to a specific file/directory
  - Relative path normally starts from directory in which the application began executing, therefore *"relative" to current directory*
- **Separator character** *for separating path files/directories*: varies among **OS**'s; use **File.separator** to obtain local computer's proper separator

Class File Methods



Method	Description
`boolean canRead()`	Returns `true` if a file is readable by the current application; `false` otherwise.
`boolean canWrite()`	Returns `true` if a file is writable by the current application; `false` otherwise.
`boolean exists()`	Returns `true` if the file or directory represented by the `File` object exists; `false` otherwise.
`boolean isFile()`	Returns `true` if the name specified as the argument to the `File` constructor is a file; `false` otherwise.
`boolean isDirectory()`	Returns `true` if the name specified as the argument to the `File` constructor is a directory; `false` otherwise.
`boolean isAbsolute()`	Returns `true` if the arguments specified to the `File` constructor indicate an absolute path to a file or directory; `false` otherwise.
`String getAbsolutePath()`	Returns a `String` with the absolute path of the file or directory.
`String getName()`	Returns a `String` with the name of the file or directory.
`String getPath()`	Returns a `String` with the path of the file or directory.
`String getParent()`	Returns a `String` with the parent directory of the file or directory (i.e., the directory in which the file or directory is located).
`long length()`	Returns the length of the file, in bytes. If the `File` object represents a directory, an unspecified value is returned.
`long lastModified()`	Returns a platform-dependent representation of the time at which the file or directory was last modified. The value returned is useful only for comparison with other values returned by this method.
`String[] list()`	Returns an array of `Strings` representing a directory's contents. Returns `null` if the `File` object does not represent a directory.

Class **File** Methods

```java
// Fig. 17.3: FileDemonstration.java
// File class used to obtain file and directory information.
import java.io.File;
import java.util.Scanner;

public class FileDemonstration
{
   public static void main( String[] args )
   {
      Scanner input = new Scanner( System.in );

      System.out.print( "Enter file or directory name: " );
      analyzePath( input.nextLine() );
   } // end main

   // display information about file user specifies
   public static void analyzePath( String path )
   {
      // create File object based on user input
      File name = new File( path );

      if ( name.exists() ) // if name exists, output information about it
      {
         // display file (or directory) information
         System.out.printf(
            "%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",
            name.getName(), " exists",
            ( name.isFile() ? "is a file" : "is not a file" ),
            ( name.isDirectory() ? "is a directory" :
               "is not a directory" ),
            ( name.isAbsolute() ? "is absolute path" :
               "is not absolute path" ), "Last modified: ",
            name.lastModified(), "Length: ", name.length(),
            "Path: ", name.getPath(), "Absolute path: ",
            name.getAbsolutePath(), "Parent: ", name.getParent() );

         if ( name.isDirectory() ) // output directory listing
         {
            String[] directory = name.list();
            System.out.println( "\n\nDirectory contents:\n" );

            for ( String directoryName : directory )
               System.out.println( directoryName );
         } // end if
      } // end outer if
      else // not file or directory, output error message
      {
         System.out.printf( "%s %s", path, "does not exist." );
      } // end else
   } // end method analyzePath
} // end class FileDemonstration
```

Note: *a single \ denotes escape character; use \\ to insert \ in a file path*

Sequential-Access Text Files

- **Sequential-access files**:
  - Store records in order by record-key field
  - Java has no such notion as records, and imposes no structure on a file; thus, programmer needs to structure files to meet application requirements
  - Text files are human-readable files (as opposed to binary files)
- Creating a *sequential-access text file*:
  - **Formatter** used to output formatted data to text-based stream
- Involves three operations:

Character-based output stream

  - **Open file**:
    - **Formatter** constructor with one String argument receives name of file, including its path; if path is not supplied, **JVM** assumes file to be in directory from which program was executed; file created if not existing already; if existing file is opened, its contents are truncated
  - **Write data to file**:
    - Data to be written to file passed to **Formatter** object
  - **Close file**:
    - Closes **Formatter** object

Sequential-Access Text Files

- **Exceptions** associated with *writing to file*:
  - **SecurityException**: occurs when user has *no permission to write* data to file
  - **FileNotFoundException**: occurs if *file does not exist*, and a new one can't be created
  - **FormatterClosedException**: occurs on attempt to output to file when *Formatter has been closed*
- **Other:**
  - System.exit(int): *static method that terminates an application*; argument of **0** indicates successful program termination, nonzero value normally indicates an error (exception)
  - Different platforms use different *line-separator characters*; use %n in a format-control string to output a platform-specific line separator

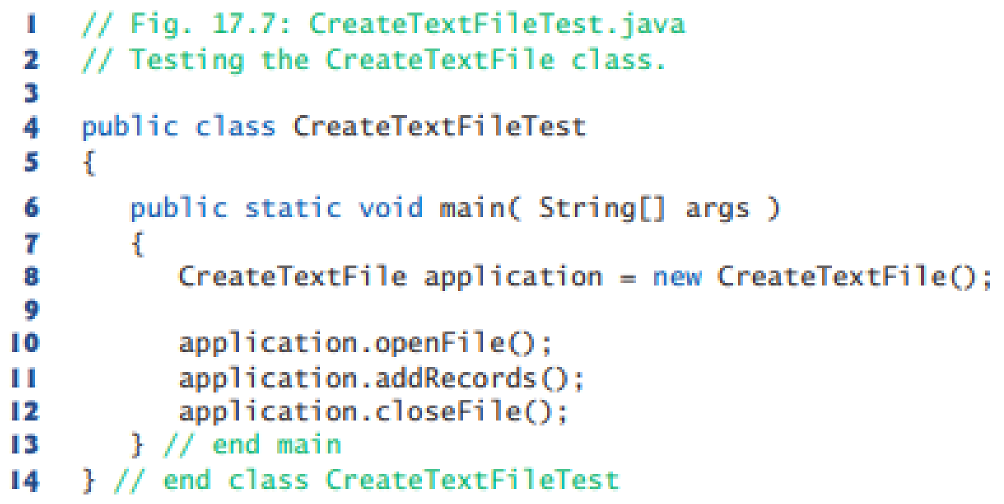
Sequential-Access Text File (AccountRecord class)

```java
// Fig. 17.4: AccountRecord.java
// AccountRecord class maintains information for one account.
package com.deitel.ch17; // packaged for reuse

public class AccountRecord
{
   private int account;
   private String firstName;
   private String lastName;
   private double balance;

   // no-argument constructor calls other constructor with default values
   public AccountRecord()
   {
      this( 0, "", "", 0.0 ); // call four-argument constructor
   } // end no-argument AccountRecord constructor

   // initialize a record
   public AccountRecord( int acct, String first, String last, double bal )
   {
      setAccount( acct );
      setFirstName( first );
      setLastName( last );
      setBalance( bal );
   } // end four-argument AccountRecord constructor

   // set account number
   public void setAccount( int acct )
   {
      account = acct;
   } // end method setAccount

   // get account number
   public int getAccount()
   {
      return account;
   } // end method getAccount

   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

   // get first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // get last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set balance
   public void setBalance( double bal )
   {
      balance = bal;
   } // end method setBalance

   // get balance
   public double getBalance()
   {
      return balance;
   } // end method getBalance
} // end class AccountRecord
```

Sequential-Access Text File (CreateTextFile class)

```java
// Fig. 17.5: CreateTextFile.java
// Writing data to a sequential text file with class Formatter.
import java.io.FileNotFoundException;
import java.lang.SecurityException;
import java.util.Formatter;
import java.util.FormatterClosedException;
import java.util.NoSuchElementException;
import java.util.Scanner;

import com.deitel.ch17.AccountRecord;

public class CreateTextFile
{
   private Formatter output; // object used to output text to file

   // enable user to open file
   public void openFile()
   {
      try
      {
         output = new Formatter( "clients.txt" ); // open the file
      } // end try
      catch ( SecurityException securityException )
      {
         System.err.println(
            "You do not have write access to this file." );
         System.exit( 1 ); // terminate the program
      } // end catch
      catch ( FileNotFoundException fileNotFoundException )
      {
         System.err.println( "Error opening or creating file." );
         System.exit( 1 ); // terminate the program
      } // end catch
   } // end method openFile

   // add records to file
   public void addRecords()
   {
      // object to be written to file
      AccountRecord record = new AccountRecord();

      Scanner input = new Scanner( System.in );

      System.out.printf( "%s\n%s\n%s\n%s\n\n",
         "To terminate input, type the end-of-file indicator ",
         "when you are prompted to enter input.",
         "On UNIX/Linux/Mac OS X type <ctrl> d then press Enter",
         "On Windows type <ctrl> z then press Enter" );

      System.out.printf( "%s\n%s",
         "Enter account number (> 0), first name, last name and balance.",
         "? " );

      while ( input.hasNext() ) // loop until end-of-file indicator
      {
         try // output values to file
         {
            // retrieve data to be output
            record.setAccount( input.nextInt() ); // read account number
            record.setFirstName( input.next() ); // read first name
            record.setLastName( input.next() ); // read last name
            record.setBalance( input.nextDouble() ); // read balance

            if ( record.getAccount() > 0 )
            {
               // write new record
               output.format( "%d %s %s %.2f\n", record.getAccount(),
                  record.getFirstName(), record.getLastName(),
                  record.getBalance() );
            } // end if
            else
            {
               System.out.println(
                  "Account number must be greater than 0." );
            } // end else
         } // end try
         catch ( FormatterClosedException formatterClosedException )
         {
            System.err.println( "Error writing to file." );
            return;
         } // end catch
         catch ( NoSuchElementException elementException )
         {
            System.err.println( "Invalid input. Please try again." );
            input.nextLine(); // discard input so user can try again
         } // end catch

         System.out.printf( "%s %s\n%s", "Enter account number (>0),",
            "first name, last name and balance.", "? " );
      } // end while
   } // end method addRecords

   // close file
   public void closeFile()
   {
      if ( output != null )
         output.close();
   } // end method closeFile
} // end class CreateTextFile
```

Sequential-Access Text File (CreateTextFileTest class)



```java
// Fig. 17.7: CreateTextFileTest.java
// Testing the CreateTextFile class.

public class CreateTextFileTest
{
   public static void main( String[] args )
   {
      CreateTextFile application = new CreateTextFile();

      application.openFile();
      application.addRecords();
      application.closeFile();
   } // end main
} // end class CreateTextFileTest
```

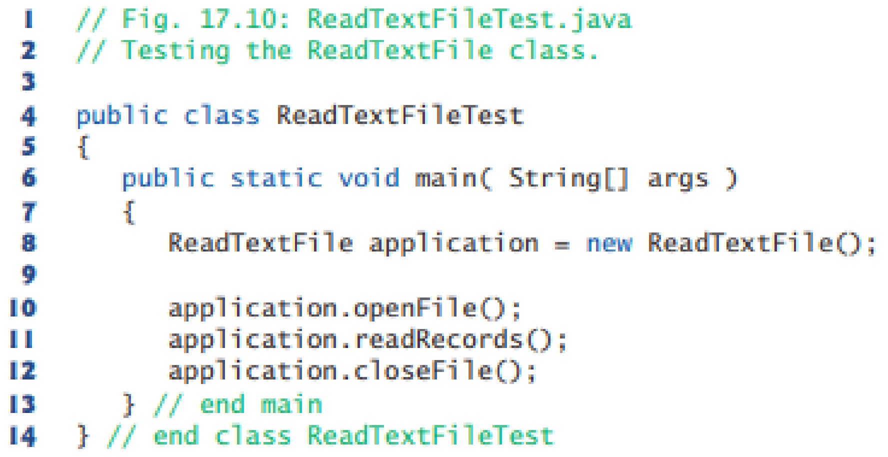
Reading Data from Sequential-Access Text File

- Data that was written to file in previous section is read back into a program to demonstrate sequential-access character-based stream input
- Has similar steps to write-to-file, that is, three operations:
  - **Open file**:
    - **Scanner** constructor takes file path of the file to be read from
  - **Read data from file**:
    - **Scanner** object receives data file passed to it by constructor, which can then be used by the program
  - **Close file**:
    - Closes **Scanner** object
- **Exceptions** associated with *reading from file*:
  - **FileNotFoundException** occurs when file does not exist or somehow can't be read
  - **NoSuchElementException** occurs if data being read by a Scanner method is in wrong format or if no data is left to input
  - **IllegalStateException** occurs if **Scanner** is closed before data is input

Reading Data from Sequential-Access Text File (ReadTextFile class)

```java
// Fig. 17.9: ReadTextFile.java
// This program reads a text file and displays each record.
import java.io.File;
import java.io.FileNotFoundException;
import java.lang.IllegalStateException;
import java.util.NoSuchElementException;
import java.util.Scanner;

import com.deitel.ch17.AccountRecord;

public class ReadTextFile
{
   private Scanner input;

   // enable user to open file
   public void openFile()
   {
      try
      {
         input = new Scanner( new File( "clients.txt" ) );
      } // end try
      catch ( FileNotFoundException fileNotFoundException )
      {
         System.err.println( "Error opening file." );
         System.exit( 1 );
      } // end catch
   } // end method openFile

   // read record from file
   public void readRecords()
   {
      // object to be written to screen
      AccountRecord record = new AccountRecord();

      System.out.printf( "%-10s%-12s%-12s%10s\n", "Account",
         "First Name", "Last Name", "Balance" );

      try // read records from file using Scanner object
      {
         while ( input.hasNext() )
         {
            record.setAccount( input.nextInt() ); // read account number
            record.setFirstName( input.next() ); // read first name
            record.setLastName( input.next() ); // read last name
            record.setBalance( input.nextDouble() ); // read balance

            // display record contents
            System.out.printf( "%-10d%-12s%-12s%10.2f\n",
               record.getAccount(), record.getFirstName(),
               record.getLastName(), record.getBalance() );
         } // end while
      } // end try
      catch ( NoSuchElementException elementException )
      {
         System.err.println( "File improperly formed." );
         input.close();
         System.exit( 1 );
      } // end catch
      catch ( IllegalStateException stateException )
      {
         System.err.println( "Error reading from file." )
         System.exit( 1 );
      } // end catch
   } // end method readRecords

   // close file and terminate application
   public void closeFile()
   {
      if ( input != null )
         input.close(); // close file
   } // end method closeFile
} // end class ReadTextFile
```

Reading Data from Sequential-Access Text File (ReadTextFileTest class)



```java
// Fig. 17.10: ReadTextFileTest.java
// Testing the ReadTextFile class.

public class ReadTextFileTest
{
   public static void main( String[] args )
   {
      ReadTextFile application = new ReadTextFile();

      application.openFile();
      application.readRecords();
      application.closeFile();
   } // end main
} // end class ReadTextFileTest
```

Case Study: Credit Enquiry Program (Section 17.4.3)

- Program further demonstrates *character-based stream data input*, using **Scanner** object and **AccountRecord** class from previous section
- Credit manager obtains a list of customers based on credit balance, and displays the list as output of the program
- **Updating sequential-access files**:
  - Poses risk of “corrupting” existing data
  - Often involves rewriting and overwriting existing data, to combine with new data
  - Other file-access methods may permit “appending” new data to existing data

Object Serialization

- Reading an entire object from file (or writing to file) is referred to as Object serialization
  - Writing to file is referred to as serialization, reading from file is referred to as deserialization (object recreated in memory)
- Serialized object is represented as a sequence of bytes that includes the object's data and its type information:
  - **FileOutputStream** used in conjunction with **ObjectOutputStream** for serialization
    - **ObjectOutput** interface method writeObject takes Object to be serialized as input argument and writes it to **OutputStream**
  - **FileInputStream** used in conjunction with **ObjectInputStream** for deserialization
    - **ObjectInput** interface method readObject reads and returns a reference to an Object from **InputStream**
    - Upon deserialization, the reference to the object is cast to the object's actual type
- An object has to be tagged as Serializable to be able to serialize it:
  - implements Serializable should appear in class definition header
  - Interface Serializable is a tagging interface, does not contain any methods
  - If any variable in a class that implements Serializable is not serializable, it must be declared transient (then it's ignored during serialization)

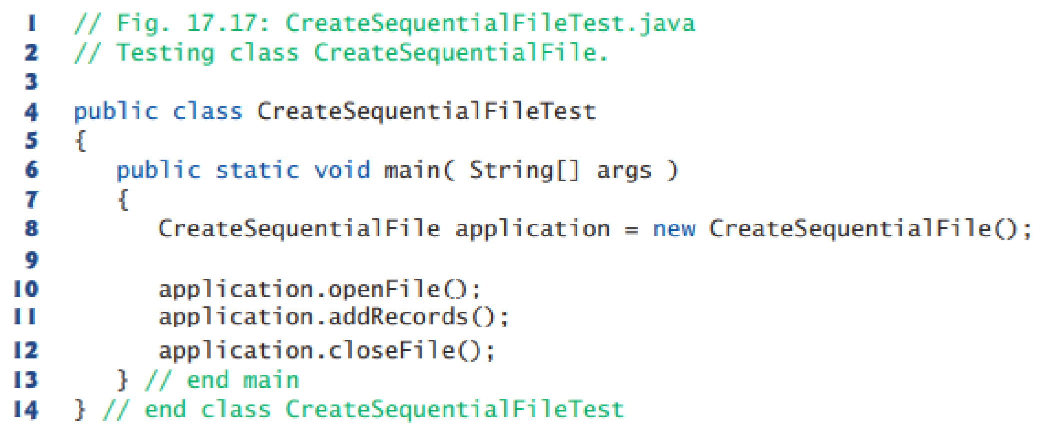
Object Serialization (class AccountRecordSerializable)

```java
// Fig. 17.15: AccountRecordSerializable.java
// AccountRecordSerializable class for serializable objects.
package com.deitel.ch17; // packaged for reuse

import java.io.Serializable;

public class AccountRecordSerializable implements Serializable
{
   private int account;
   private String firstName;
   private String lastName;
   private double balance;

   // no-argument constructor calls other constructor with default values
   public AccountRecordSerializable()
   {
      this( 0, "", "", 0.0 );
   } // end no-argument AccountRecordSerializable constructor

   // four-argument constructor initializes a record
   public AccountRecordSerializable(
      int acct, String first, String last, double bal )
   {
      setAccount( acct );
      setFirstName( first );
      setLastName( last );
      setBalance( bal );
   } // end four-argument AccountRecordSerializable constructor

   // set account number
   public void setAccount( int acct )
   {
      account = acct;
   } // end method setAccount

   // get account number
   public int getAccount()
   {
      return account;
   } // end method getAccount

   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

   // get first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // get last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set balance
   public void setBalance( double bal )
   {
      balance = bal;
   } // end method setBalance

   // get balance
   public double getBalance()
   {
      return balance;
   } // end method getBalance
} // end class AccountRecordSerializable
```

Object Serialization (class CreateSequentialFile)

```java
// Fig. 17.16: CreateSequentialFile.java
// Writing objects sequentially to a file with class ObjectOutputStream.
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.NoSuchElementException;
import java.util.Scanner;

import com.deitel.ch17.AccountRecordSerializable;

public class CreateSequentialFile
{
   private ObjectOutputStream output; // outputs data to file

   // allow user to specify file name
   public void openFile()
   {
      try // open file
      {
         output = new ObjectOutputStream(
            new FileOutputStream( "clients.ser" ) );
      } // end try
      catch ( IOException ioException )
      {
         System.err.println( "Error opening file." );
      } // end catch
   } // end method openFile

   // add records to file
   public void addRecords()
   {
      AccountRecordSerializable record; // object to be written to file
      int accountNumber = 0; // account number for record object
      String firstName; // first name for record object
      String lastName; // last name for record object
      double balance; // balance for record object

      Scanner input = new Scanner( System.in );

      System.out.printf( "%s\n%s\n%s\n%s\n\n",
         "To terminate input, type the end-of-file indicator ",
         "when you are prompted to enter input.",
         "On UNIX/Linux/Mac OS X type <ctrl> d then press Enter",
         "On Windows type <ctrl> z then press Enter" );

      System.out.printf( "%s\n%s",
         "Enter account number (> 0), first name, last name and balance.",
         "? " );

      while ( input.hasNext() ) // loop until end-of-file indicator
      {
         try // output values to file
         {
            accountNumber = input.nextInt(); // read account number
            firstName = input.next(); // read first name
            lastName = input.next(); // read last name
            balance = input.nextDouble(); // read balance

            if ( accountNumber > 0 )
            {
               // create new record
               record = new AccountRecordSerializable( accountNumber,
                  firstName, lastName, balance );
               output.writeObject( record ); // output record
            } // end if
            else
            {
               System.out.println(
                  "Account number must be greater than 0." );
            } // end else
         } // end try
         catch ( IOException ioException )
         {
            System.err.println( "Error writing to file." );
            return;
         } // end catch
         catch ( NoSuchElementException elementException )
         {
            System.err.println( "Invalid input. Please try again." );
            input.nextLine(); // discard input so user can try again
         } // end catch

         System.out.printf( "%s %s\n%s", "Enter account number (>0),",
            "first name, last name and balance.", "? " );
      } // end while
   } // end method addRecords

   // close file and terminate application
   public void closeFile()
   {
      try // close file
      {
         if ( output != null )
            output.close();
      } // end try
      catch ( IOException ioException )
      {
         System.err.println( "Error closing file." );
         System.exit( 1 );
      } // end catch
   } // end method closeFile
} // end class CreateSequentialFile
```

Object Serialization (class CreateSequentialFileTest)



```java
// Fig. 17.17: CreateSequentialFileTest.java
// Testing class CreateSequentialFile.

public class CreateSequentialFileTest
{
   public static void main( String[] args )
   {
      CreateSequentialFile application = new CreateSequentialFile();

      application.openFile();
      application.addRecords();
      application.closeFile();
   } // end main
} // end class CreateSequentialFileTest
```

Object deserialization

- Example of reading objects from file that was created in previous section;
- Quite similar to procedure followed in previous section (*open file for read* >> *read data* >> *close file*)
- **ObjectInputStream** used in conjunction with **FileInputStream**
- Exception handling included to take care of exceptions associated with read-from-file (*failure-to-open-file*, *end-of-file*, *class-not-found*)

Object Deserialization (class ReadSequentialFile)

```java
// Fig. 17.18: ReadSequentialFile.java
// Reading a file of objects sequentially with ObjectInputStream
// and displaying each record.
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

import com.deitel.ch17.AccountRecordSerializable;

public class ReadSequentialFile
{
   private ObjectInputStream input;

   // enable user to select file to open
   public void openFile()
   {
      try // open file
      {
         input = new ObjectInputStream(
            new FileInputStream( "clients.ser" ) );
      } // end try
      catch ( IOException ioException )
      {
         System.err.println( "Error opening file." );
      } // end catch
   } // end method openFile

   // read record from file
   public void readRecords()
   {
      AccountRecordSerializable record;
      System.out.printf( "%-10s%-12s%-12s%10s\n", "Account",
         "First Name", "Last Name", "Balance" );

      try // input the values from the file
      {
         while ( true )
         {
            record = ( AccountRecordSerializable ) input.readObject();

            // display record contents
            System.out.printf( "%-10d%-12s%-12s%10.2f\n",
               record.getAccount(), record.getFirstName(),
               record.getLastName(), record.getBalance() );
         } // end while
      } // end try
      catch ( EOFException endOfFileException )
      {
         return; // end of file was reached
      } // end catch
      catch ( ClassNotFoundException classNotFoundException )
      {
         System.err.println( "Unable to create object." );
      } // end catch
      catch ( IOException ioException )
      {
         System.err.println( "Error during read from file." );
      } // end catch
   } // end method readRecords

   // close file and terminate application
   public void closeFile()
   {
      try // close file and exit
      {
         if ( input != null )
            input.close();
      } // end try
      catch ( IOException ioException )
      {
         System.err.println( "Error closing file." );
         System.exit( 1 );
      } // end catch
   } // end method closeFile
} // end class ReadSequentialFile
```

class ReadSequentialFileTest

```java
// Fig. 17.19: ReadSequentialFileTest.java
// Testing class ReadSequentialFile.

public class ReadSequentialFileTest
{
   public static void main( String[] args )
   {
      ReadSequentialFile application = new ReadSequentialFile();

      application.openFile();
      application.readRecords();
      application.closeFile();
   } // end main
} // end class ReadSequentialFileTest
```

Additional java.io Classes

- Additional interfaces and classes from package java.io for byte-based and character-based input/output streams
- **InputStream**/**OutputStream**: abstract classes that declare methods for performing byte-based input/output
- **Pipes**: synchronized communication channels between threads:
  - **PipedInputStream**/**PipedOutputStream**: subclasses of **InputStream**/ **OutputStream**, establish pipes between two threads in a program
  - One thread sends data to another by writing to a **PipedOutputStream**
  - Target thread reads information from the pipe via a **PipedInputStream**
- **Filtering** (an input/output stream): providing additional functionality or modifying data to facilitate processing
  - **FilterInputStream**/**FilterOutputStream**: filtered versions of **InputStream**/**OutputStream**; usually extended (i.e. subclassed) to provide additional capabilities
- **PrintStream**: a subclass of **FilterOutputStream**, performs text output to specified stream
  - **System.out** and **System.err** are **PrintStream** objects

Additional java.io Classes

- Interfaces **DataInput**/**DataOutput**: define methods for reading/writing primitive types (e.g. int, float, double, etc.) from/to an input/output stream
  - Implemented by classes **DataInputStream**/**DataOutputStream** and **RandomAccessFile** to read sets of bytes and process them as primitive-type values (respectively write primitive-type values as bytes)
- **Buffering**: an I/O performance-enhancement technique, providing transfer of large amounts of data in/out of programs with one large operation:
  - **BufferedOutputStream**: each output operation directed to a buffer, then transfer to output device done in one large operation whenever buffer fills (stream method flush can be used to force data out to device when buffer is only partially filled)
  - **BufferedInputStream**: many “logical” chunks of data from a file read as one large physical input operation into a memory buffer, then program reads from the buffer whenever data input is required; request for data directed to input device only when buffer is empty
- **ByteArrayInputStream**/**ByteArrayOutputStream**: subclasses of **InputStream**/**OutputStream** allowing to read from byte array into memory (respectively output byte array to memory)

Additional java.io Classes

- **SequenceInputStream** (subclass of **InputStream**) logically concatenates several **InputStreams**, so that program sees one continuous stream from several (when program reaches end of one input stream, that stream closes, the next in sequence opens)
- **BufferedReader**/**BufferedWriter**: subclasses of **Reader**/**Writer** abstract classes (Unicode two-byte, character-based streams) for buffering of character-based streams
- **CharArrayReader**/**CharArrayWriter**: for reading/writing a stream of characters to/from a char array
- **LineNumberReader** (subclass of **BufferedReader**): a buffered character stream that keeps track of the number of lines read
- **InputStreamReader**/**InputStreamWriter**: used to convert **InputStream**/**OutputStream** to **Reader**/**Writer**
- **FileReader**/**FileWriter** read characters from (respectively write characters to) a file
- **PipedReader**/**PipedWriter**: implement piped-character streams for transferring data between threads
- **StringReader**/**StringWriter**: read characters from (write characters to) Strings
- **PrintWriter** writes characters to a stream

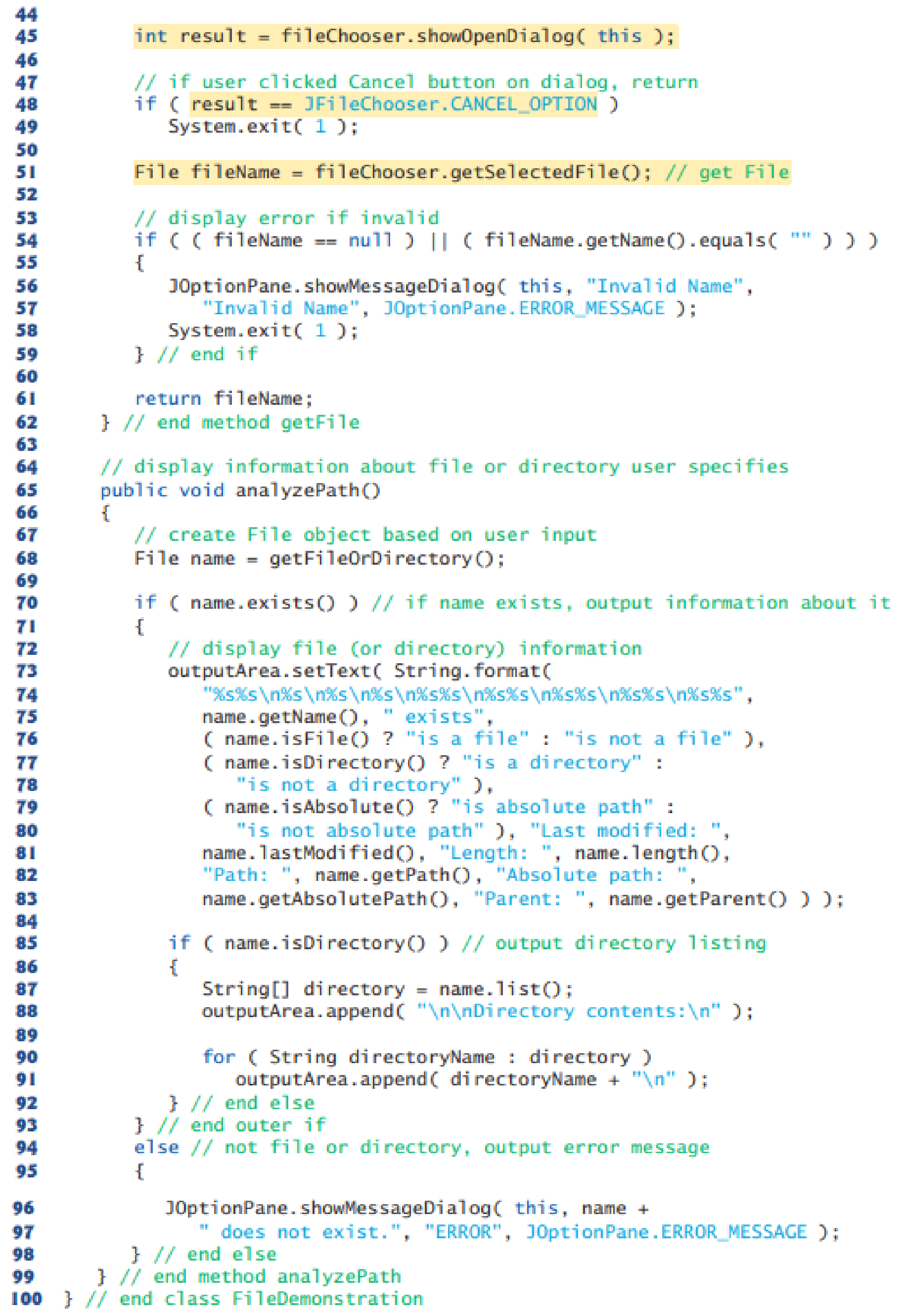
Opening Files with JFileChooser

- Class **JFileChooser** displays a dialog that enables the user to easily select files/directories
- **JFileChooser** methods/variables:
  - setFileSelectionMode specifies what user can select for opening; options are static constants:
    - FILES-ONLY (default)
    - DIRECTORIES_ONLY
    - FILES_AND_DIRECTORIES
  - showOpenDialog displays **JFileChooser** dialog titled Open; returns an integer specifying whether user pressed Open (static constant APPROVE_OPTION) or Cancel (static constant CANCEL_OPTION)
  - getSelectedFile retrieves file selected by user

Opening Files with JFileChooser

```java
// Fig. 17.20: FileDemonstration.java
// Demonstrating JFileChooser.
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class FileDemonstration extends JFrame
{
   private JTextArea outputArea; // used for output
   private JScrollPane scrollPane; // used to provide scrolling to output

   // set up GUI
   public FileDemonstration()
   {
      super( "Testing class File" );

      outputArea = new JTextArea();

      // add outputArea to scrollPane
      scrollPane = new JScrollPane( outputArea );

      add( scrollPane, BorderLayout.CENTER ); // add scrollPane to GUI

      setSize( 400, 400 ); // set GUI size
      setVisible( true ); // display GUI

      analyzePath(); // create and analyze File object
   } // end FileDemonstration constructor

   // allow user to specify file or directory name
   private File getFileOrDirectory()
   {
      // display file dialog, so user can choose file or directory to open
      JFileChooser fileChooser = new JFileChooser();
      fileChooser.setFileSelectionMode(
         JFileChooser.FILES_AND_DIRECTORIES );
```

Opening Files with JFileChooser



```
      int result = fileChooser.showOpenDialog( this );

      // if user clicked Cancel button on dialog, return
      if ( result == JFileChooser.CANCEL_OPTION )
         System.exit( 1 );

      File fileName = fileChooser.getSelectedFile(); // get File

      // display error if invalid
      if ( ( fileName == null ) || ( fileName.getName().equals( "" ) ) )
      {
         JOptionPane.showMessageDialog( this, "Invalid Name",
            "Invalid Name", JOptionPane.ERROR_MESSAGE );
         System.exit( 1 );
      } // end if

      return fileName;
   } // end method getFile

   // display information about file or directory user specifies
   public void analyzePath()
   {
      // create File object based on user input
      File name = getFileOrDirectory();

      if ( name.exists() ) // if name exists, output information about it
      {
         // display file (or directory) information
         outputArea.setText( String.format(
            "%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",
            name.getName(), " exists",
            ( name.isFile() ? "is a file" : "is not a file" ),
            ( name.isDirectory() ? "is a directory" :
               "is not a directory" ),
            ( name.isAbsolute() ? "is absolute path" :
               "is not absolute path" ), "Last modified: ",
            name.lastModified(), "Length: ", name.length(),
            "Path: ", name.getPath(), "Absolute path: ",
            name.getAbsolutePath(), "Parent: ", name.getParent() ) );

         if ( name.isDirectory() ) // output directory listing
         {
            String[] directory = name.list();
            outputArea.append( "\n\nDirectory contents:\n" );

            for ( String directoryName : directory )
               outputArea.append( directoryName + "\n" );
         } // end else
      } // end outer if
      else // not file or directory, output error message
      {
         JOptionPane.showMessageDialog( this, name +
            " does not exist.", "ERROR", JOptionPane.ERROR_MESSAGE );
      } // end else
   } // end method analyzePath
} // end class FileDemonstration
```

Opening Files with JFileChooser

```java
// Fig. 17.21: FileDemonstrationTest.java
// Testing class FileDemonstration.
import javax.swing.JFrame;

public class FileDemonstrationTest
{
   public static void main( String[] args )
   {
      FileDemonstration application = new FileDemonstration();
      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
   } // end main
} // end class FileDemonstrationTest
```