File Handling

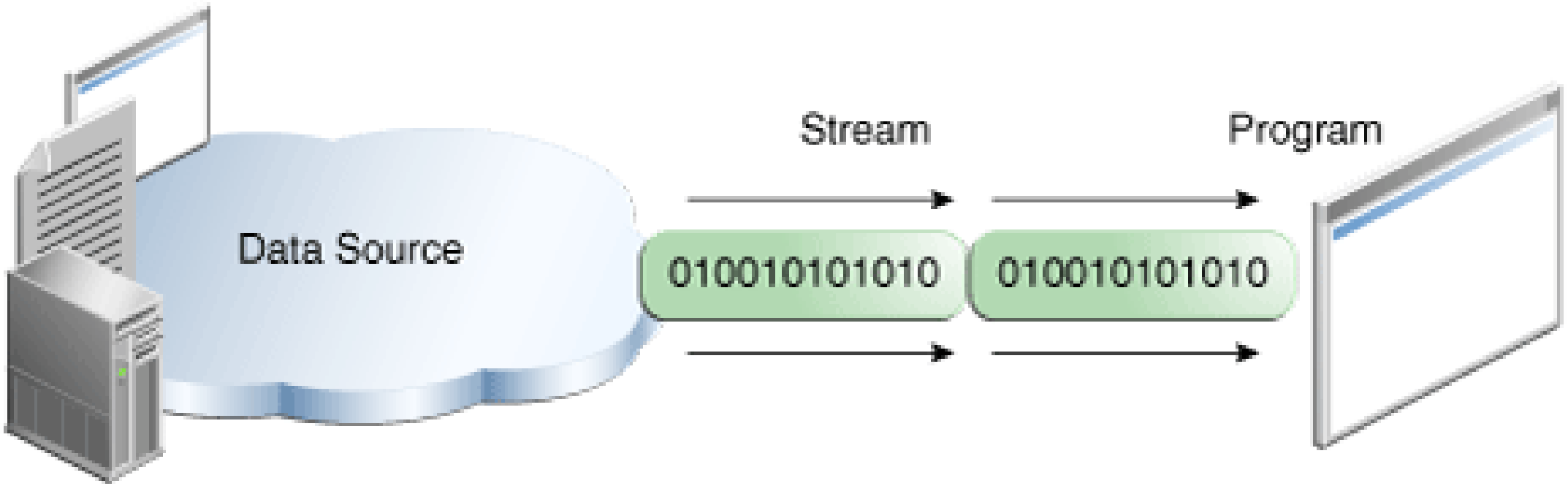
Byte & Character Stream

I/O Stream

- The java.io package provides an extensive library of classes for dealing with input and output.
- Java provides streams as a general mechanism for dealing with data I/O.
- Streams implement sequential access of data.
- There are two kinds of streams:
  1. byte streams (binary streams)
  2. character streams (text streams)
- An input stream is an object that an application can use to read a sequence of data.
- An output stream is an object that an application can use to write a sequence of data.
- An input stream acts as a source of data.
- An output stream acts as a destination of data.

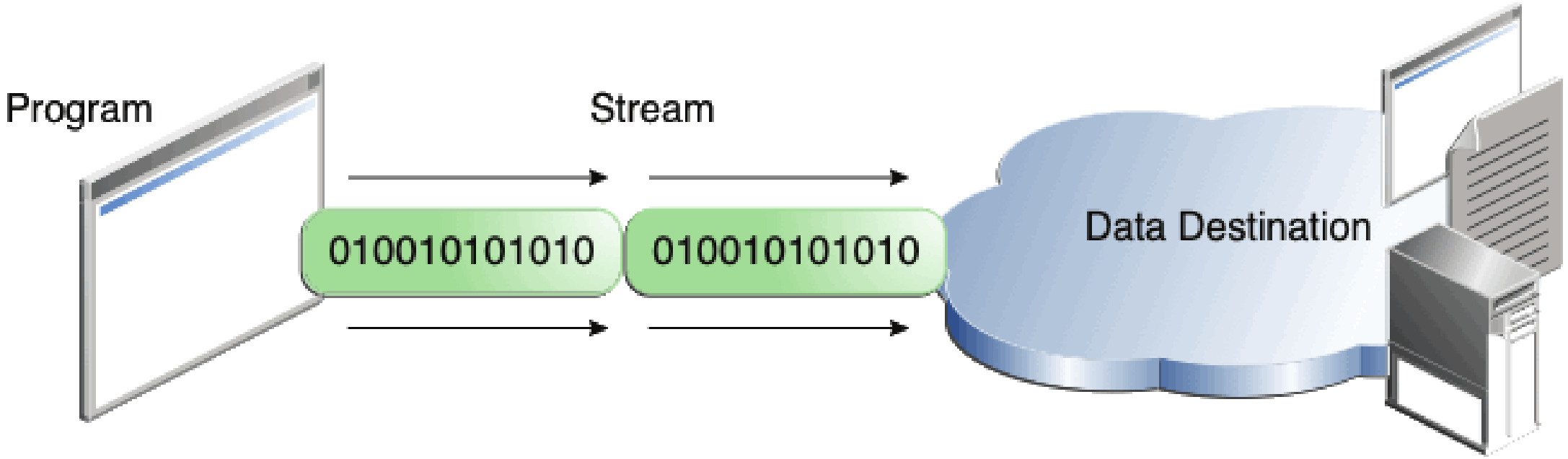
I/O Stream

- A program uses an input stream to read data from a source, one item at a time:



I/O Stream

- A program uses an output stream to write data to a destination, one item at time:



Byte Streams

- ByteStream classes are used to read bytes from the input stream and write bytes to the output stream.
- ByteStream classes read/write the data of 8-bits.
- The ByteStream classes are divided into two types of classes.
  - InputStream
  - OutputStream.
- There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, FileInputStream and FileOutputStream.

Byte Streams- InputStream

java.io.InputStream

- java.io.ByteArrayInputStream
- java.io.FileInputStream
- java.io.FilterInputStream
  - java.io.BufferedInputStream
  - java.io.DataInputStream
  - java.io.LineNumberInputStream
  - java.io.PushbackInputStream
- java.io.ObjectInputStream
- java.io.PipedInputStream
- java.io.SequenceInputStream
- java.io.StringBufferInputStream

Byte Streams- InputStream

- The InputStream class provides methods to read bytes from a file, console or memory.
- It is an abstract class and can't be instantiated.
- However, various classes inherit the InputStream class and override its methods.

Class	Description
BufferedInputStream	This class provides methods to read bytes from the buffer.
ByteArrayInputStream	This class provides methods to read bytes from the byte array.
DataInputStream	This class provides methods to read Java primitive data types.
FileInputStream	This class provides methods to read bytes from a file.
FilterInputStream	This class contains methods to read bytes from the other input streams, which are used as the primary source of data.
ObjectInputStream	This class provides methods to read objects. (java.io.Serializable or java.io.Externalizable)
PipedInputStream	This class provides methods to read from a piped output stream to which the piped input stream must be connected.
SequenceInputStream	This class provides methods to connect multiple Input Stream and read data from them.

Byte Streams- InputStream

- The InputStream class contains various methods to read the data from an input stream.
- These methods are overridden by the classes that inherit the InputStream class.

Method	Description
int read()	This method returns an integer, an integral representation of the next available byte of the input. The integer -1 is returned once the end of the input is encountered.
int read (byte buffer [])	This method is used to read the specified buffer length bytes from the input and returns the total number of bytes successfully read. It returns -1 once the end of the input is encountered.
int read (byte buffer [], int loc, int nBytes)	This method is used to read the 'nBytes' bytes from the buffer starting at a specified location, 'loc'. It returns the total number of bytes successfully read from the input. It returns -1 once the end of the input is encountered.
int available ()	This method returns the number of bytes that are available to read.
void mark(int nBytes)	This method is used to mark the current position in the input stream until the specified nBytes are read.
void reset ()	This method is used to reset the input pointer to the previously set mark.
long skip (long nBytes)	This method is used to skip the nBytes of the input stream and returns the total number of bytes that are skipped.
void close ()	This method is used to close the input source. If an attempt is made to read even after the closing, IOException is thrown by the method.

Byte Streams- OutputStream

java.io.OutputStream

- java.io.ByteArrayOutputStream
- java.io.FileOutputStream
- java.io.FilterOutputStream
  - java.io.BufferedOutputStream
  - java.io.DataOutputStream
  - java.io.PrintStream
- java.io.ObjectOutputStream
- java.io.PipedOutputStream

Byte Streams- OutputStream

- The OutputStream is an abstract class that is used to write 8-bit bytes to the stream.
- It is the superclass of all the output stream classes.
- However, various classes inherit the OutputStream class and override its methods.

Class	**Description**
BufferedOutputStream	This class provides methods to write the bytes to the buffer.
ByteArrayOutputStream	This class provides methods to write bytes to the byte array.
DataOutputStream	This class provides methods to write the java primitive data types.
FileOutputStream	This class provides methods to write bytes to a file.
FilterOutputStream	This class provides methods to write to other output streams.
ObjectOutputStream	This class provides methods to write objects.
PipedOutputStream	It provides methods to write bytes to a piped output stream.
PrintStream	It provides methods to print Java primitive data types.

Byte Streams- OutputStream

- The OutputStream class provides various methods to write bytes to the output streams.
- These methods are overridden by the classes that inherit the OutputStream class.

Method	**Description**
void write (int i)	This method is used to write the specified single byte to the output stream.
void write (byte buffer [])	It is used to write a byte array to the output stream.
void write(bytes buffer[], int loc, int nBytes)	It is used to write nByte bytes to the output stream from the buffer starting at the specified location.
void flush ()	It is used to flush the output stream and writes the pending buffered bytes.
void close ()	It is used to close the output stream. However, if we try to close the already closed output stream, the IOException will be thrown by this method.

Byte Streams

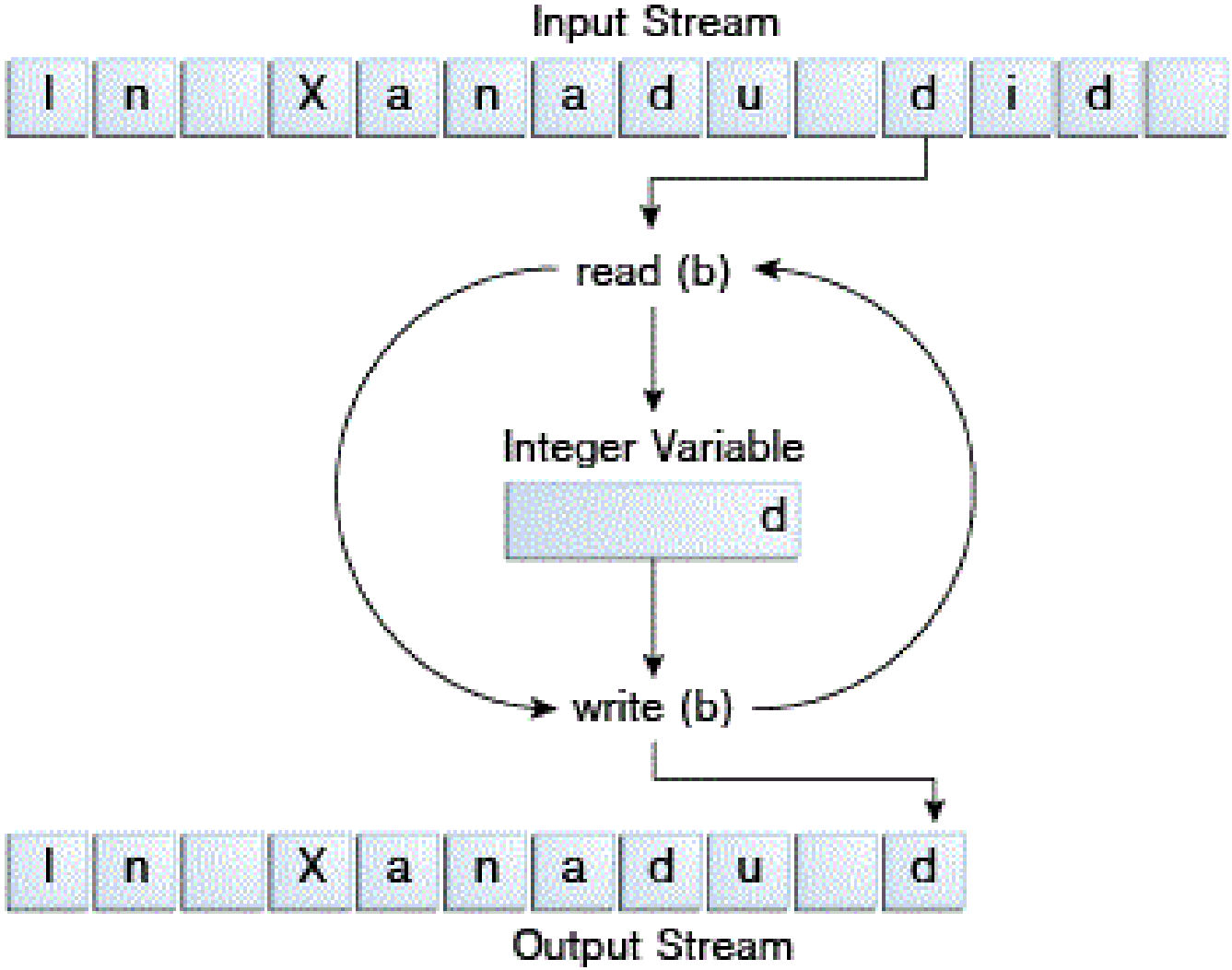
```
import java.io.*;
public class CopyBytes {
  public static void main(String args[]) throws IOException {
    FileInputStream in = null;
    FileOutputStream out = null;
    try {
      in = new FileInputStream("input.txt");
      out = new FileOutputStream("output.txt");
      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
      }
    }finally {
      if (in != null) {
        in.close();        }
      if (out != null) {
        out.close();       }
  }  } }
```

Input.txt
Ram Chander

Output:
Output.txt
Ram Chander

Byte Streams

- CopyBytes spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time.



Byte Streams

Close Streams:

- Closing a stream when it's no longer needed is very important — so important that CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs.
- This practice helps avoid serious resource leaks.
- One possible error is that CopyBytes was unable to open one or both files.
- When that happens, the stream variable corresponding to the file never changes from its initial null value.
- That's why CopyBytes makes sure that each stream variable contains an object reference before invoking close.

Character Streams

- ByteStream can only handle the 8-bit bytes and is not compatible to work directly with the Unicode characters.
- The java.io package provides CharacterStream classes to overcome the limitations of ByteStream classes.
- CharacterStream classes are used to work with 16-bit Unicode characters.
- They can perform operations on characters, char arrays and Strings.
- The CharacterStream classes are divided into two types of classes.
  - Reader class
  - Writer class.

Character Streams- Reader Class

java.io.Reader

- java.io.BufferedReader
- java.io.CharArrayReader
- java.io.FilterReader
- java.io.InputStreamReader
  - java.io.FileReader
- java.io.PipedReader
- java.io.StringReader

Character Streams- Reader Class

- Reader class is used to read the 16-bit characters from the input stream.
- It is an abstract class and can't be instantiated.
- However, various classes inherit the Reader class and override its methods.
- All methods of the Reader class throw an IOException.

Class	Description
BufferedReader	This class provides methods to read characters from the buffer.
CharArrayReader	This class provides methods to read characters from the char array.
FileReader	This class provides methods to read characters from the file.
FilterReader	This class provides methods to read characters from the underlying character input stream.
InputStreamReader	This class provides methods to convert bytes to characters.
PipedReader	This class provides methods to read characters from the connected piped output stream.
StringReader	This class provides methods to read characters from a string.

Character Streams- Reader Class

- The Reader class methods are

Method	Description
int read()	This method returns the integral representation of the next character present in the input. It returns -1 if the end of the input is encountered.
int read(char buffer[])	This method is used to read from the specified buffer. It returns the total number of characters successfully read. It returns -1 if the end of the input is encountered.
int read(char buffer[], int loc, int nChars)	This method is used to read the specified nChars from the buffer at the specified location. It returns the total number of characters successfully read.
void mark(int nchars)	This method is used to mark the current position in the input stream until nChars characters are read.
void reset()	This method is used to reset the input pointer to the previous set mark.
long skip(long nChars)	This method is used to skip the specified nChars characters from the input stream and returns the number of characters skipped.
boolean ready()	This method returns a boolean value true if the next request of input is ready. Otherwise, it returns false.
void close()	This method is used to close the input stream. However, if the program attempts to access the input, it generates IOException.

Character Streams- Writer Class

java.io.Reader

- java.io.BufferedWriter
- java.io.CharArrayWriter
- java.io.FilterWriter
- java.io.OutputStreamWriter
  - java.io.FileWriter
- java.io.PipedWriter
- java.io.PrintWriter
- java.io.StringWriter

Character Streams- Writer Class

- Writer class is used to write 16-bit Unicode characters to the output stream.
- The methods of the Writer class generate IOException.
- Like Reader class, Writer class is also an abstract class that cannot be instantiated

Class	**Description**
BufferedWriter	This class provides methods to write characters to the buffer.
FileWriter	This class provides methods to write characters to the file.
CharArrayWriter	This class provides methods to write the characters to the character array.
OutpuStreamWriter	This class provides methods to convert from bytes to characters.
PipedWriter	This class provides methods to write the characters to the piped output stream.
StringWriter	This class provides methods to write the characters to the string.
PrintWriter	It provides methods to print Java primitive data types.

Character Streams- Writer Class

- The Writer class provides various methods to write characters to the output streams.
- These methods are overridden by the classes that inherit the Writer class.

Method	**Description**
void write()	This method is used to write the data to the output stream.
void write(int i)	This method is used to write a single character to the output stream.
void write(char buffer[])	This method is used to write the array of characters to the output stream.
void write(char buffer [], int loc, int nChars)	This method is used to write the nChars characters to the character array from the specified location.
void close ()	This method is used to close the output stream. However, this generates the IOException if an attempt is made to write to the output stream after closing the stream.
void flush ()	This method is used to flush the output stream and writes the waiting buffered characters.

Character Streams

```
import java.io.*;
public class CopyCharacters {
   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;
      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");
         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();         }
         if (out != null) {
            out.close();         }
    }   } }
```

Input.txt
Ram Chander

Output:
Output.txt
Ram Chander

Character Streams

- CopyCharacters is very similar to CopyBytes.
- The most important difference is that CopyCharacters uses FileReader and FileWriter for input and output in place of FileInputStream and FileOutputStream.
- Notice that both CopyBytes and CopyCharacters use an int variable to read to and write from.
- However, in CopyCharacters, the int variable holds a character value in its last **16 bits**; in CopyBytes, the int variable holds a byte value in its last **8 bits**.

Buffered Streams

- Unbuffered I/O read or write request is handled directly by the underlying OS. This can make a program much less efficient.
- To reduce this kind of overhead, the Java platform implements buffered I/O streams.
- There are four buffered stream classes used to wrap unbuffered streams:
- BufferedInputStream and BufferedOutputStream create buffered byte streams, while BufferedReader and BufferedWriter create buffered character streams.

Buffered Streams

- Some buffered output classes support autoflush, specified by an optional constructor argument.
- When autoflush is enabled, certain key events cause the buffer to be flushed.
- For example, an autoflush PrintWriter object flushes the buffer on every invocation of println or format.
- To flush a stream manually, invoke its flush method.
- The flush method is valid on any output stream.

Buffered Streams

```
import java.io.*;
public class CopyLines {
   public static void main(String args[]) throws IOException {
      BufferedReader inputStream = null;
       BufferedWriter outputStream = null;
       try {
          inputStream = new BufferedReader(new FileReader("input.txt"));
          outputStream = new BufferedWriter(new FileWriter("output.txt"));
          String l;
          while ((l = inputStream.readLine()) != null) {
             outputStream.write(l);
          }
       } finally {
          if (inputStream != null) {
             inputStream.close();
          }
          if (outputStream != null) {
             outputStream.close();
          }   }  } }
```

Input.txt
Ram Chander

Output:
Output.txt
Ram Chander

Buffered Streams

```
import java.io.*;
public class CopyLines {
   public static void main(String args[]) throws IOException {
      BufferedReader inputStream = null;
       PrintWriter outputStream = null;
       try {
          inputStream = new BufferedReader(new FileReader("input.txt"));
          outputStream = new PrintWriter(new FileWriter("output.txt"));
          String l;
          while ((l = inputStream.readLine()) != null) {
             outputStream.write(l);
          }
       } finally {
          if (inputStream != null) {
             inputStream.close();
          }
          if (outputStream != null) {
             outputStream.close();
          }   }  } }
```

Input.txt
Ram Chander

Output:
Output.txt
Ram Chander

Buffered Streams

- Invoking readLine() returns a line of text with the line.
- CopyLines outputs each line using println.
- PrintWriter is one of the character-based classes.
- PrintWriter supports the print( ) and println( ) methods.
- Thus, you can use these methods in the same way as you used them with System.out.

Scanning and Formatting

- The scanner API breaks input into individual tokens associated with bits of data.
- The formatting API assembles data into nicely formatted, human-readable form.

Scanning

- Objects of type Scanner are useful for breaking down formatted input into tokens and translating individual tokens according to their data type.
- By default, a scanner uses white space to separate tokens.
- White space characters include blanks, tabs, and line terminators.

Scanning

```
import java.io.*;
import java.util.Scanner;
public class ScanData {
public static void main(String args[]) throws IOException {
        Scanner s = null;
       try {
          s = new Scanner(new BufferedReader(new FileReader("input.txt")));
          while (s.hasNext()) {
             System.out.println(s.next());
          }
       } finally {
          if (s != null) {
             s.close();
          }
    }   } }
```

Input.txt
Ram Chander

Output:
?

Scanning

```
import java.io.*;
import java.util.Scanner;
public class ScanData {
public static void main(String args[]) throws IOException {
        Scanner s = null;
       try {
          s = new Scanner(new BufferedReader(new FileReader("input.txt")));
          while (s.hasNext()) {
             System.out.println(s.next());
          }
       } finally {
          if (s != null) {
             s.close();
          }
    }   } }
```

Input.txt
Ram Chander

Output:
Ram
Chander

Scanning

- The program that reads the individual words in input.txt and prints them out, one per line.
- Notice that ScanData invokes Scanner's close method when it is done with the scanner object. Even though a scanner is not a stream, you need to close it to indicate that you're done with its underlying stream.
- The ScanData example treats all input tokens as simple String values. Scanner also supports tokens for all of the Java language's primitive types except for char.

Scanning

```
import java.io.*;
import java.util.Scanner;
public class ScanData {
public static void main(String args[]) throws IOException {
        Scanner s = null;
       try {
          s = new Scanner(new BufferedReader(new FileReader("input.txt")));
          while (s.hasNextInt()) {
             System.out.println(s.nextInt());
          }
       } finally {
          if (s != null) {
             s.close();
          }
    }   } }
```

Input.txt

10
20
30

Output:

10
20
30

Scanning

- boolean hasNext()
- Returns true if this scanner boolean has another token in its input.
- boolean hasNextByte()
- boolean hasNextDouble()
- boolean hasNextFloat()
- boolean hasNextInt()
- boolean hasNextLine()
- boolean hasNextLong()
- boolean hasNextShort()
- boolean hasNextBoolean()
- Returns true if this scanner has another token corresponding input.

Scanning

- String next()
- Finds and returns the next complete token from this scanner.
- byte nextByte()
- double nextDouble()
- float nextFloat()
- int nextInt()
- String nextLine()
- long nextLong()
- short nextShort()
- boolean nextBoolean()
- Scans the next token of the input as a corresponding type and return that.

Formatting

- The format method formats multiple arguments based on a format string.
- The format string consists of static text embedded with format specifiers except for the format specifiers, the format string is output unchanged.

```
public PrintStream format(String format, Object... args)
```

Example:

```
public class Root2 {
   public static void main(String[] args) {
      int i = 2;
      double r = Math.sqrt(i);

      System.out.format("The square root of %d is %f.%n", i, r);
   }
}
```

Formatting

- The format method formats multiple arguments based on a format string.
- The format string consists of static text embedded with format specifiers except for the format specifiers, the format string is output unchanged.

```
public PrintStream format(String format, Object... args)
```

Example:

```
public class Root2 {
   public static void main(String[] args) {
      int i = 2;
      double r = Math.sqrt(i);

      System.out.format("The square root of %d is %f.%n", i, r);
   }
}
```

Output:

The square root of 2 is 1.414214.

Formatting

- IllegalFormatException - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions.
  - *d* formats an integer value as a decimal value.
  - *f* formats a floating point value as a decimal value.
  - *n* outputs a platform-specific line terminator.

https://docs.oracle.com/javase/tutorial/essential/io/formatting.html

Standard Streams

- The Java platform supports three Standard Streams:
  - Standard Input, accessed through System.in
  - Standard Output, accessed through System.out
  - Standard Error, accessed through System.err.
- System.out and System.err are defined as PrintStream objects.
- System.in is defined as InputStream object.

Standard Streams

```
import java.io.*;
    class Input{
        public static void main(String args[])throws Exception{
                InputStreamReader r=new InputStreamReader(System.in);
                BufferedReader br=new BufferedReader(r);
                System.out.println("Enter your name");
                String name=br.readLine();
                System.out.println("Welcome "+name);
        }
    }
```

Console:
Enter your name
Ram Chander *//input*
Welcome Ram Chander

Wrapper classes

- The eight classes of the java.lang package are known as wrapper classes in Java.

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Wrapper classes

- Boolean, Character, Byte, Short, Integer, Long, Float and Double
- The wrapper classes have a static method:

Primitive_Type **parse**NameOfClass(String s)

public static boolean parseBoolean(String s)

public static byte parseByte(String s)

public static short parseShort(String s)

public static int parseInt(String s)

public static Long parseLong(String s)

public static float parseFloat(String s)

public static double parseDouble(String s)

Example

```
import java.io.*;
public class Input {
  public static void main(String args[]) throws IOException {
        InputStreamReader r=new InputStreamReader(System.in);
        BufferedReader br=new BufferedReader(r);
        System.out.println("Enter the value:");
        String value=br.readLine();
        int a=Integer.parseInt(value);
        System.out.println("Square of the value is: "+a*a);

  }
}
```

Console:
Enter the value:
3
Square of the value is: 9

Example

```
import java.io.*;
public class Input {
  public static void main(String args[]) throws IOException {
        InputStreamReader r=new InputStreamReader(System.in);
        BufferedReader br=new BufferedReader(r);
        System.out.println("Enter the value:");
        String value=br.readLine();
        int a=Integer.parseInt(value);
        System.out.println("Square of the value is: "+a*a);

  }
}
```

```
while(!value.equals("stop"))
```

Console:
Enter the value:
3
Square of the value is: 9

Example

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

        Scanner my_scan = new Scanner(System.in);
        System.out.println("Enter the value:");
        String value=my_scan.nextLine();
        int a=Integer.parseInt(value);
        System.out.println("Square of the value is: "+a*a);

  }
}
```

Console:
Enter the value:
3
Square of the value is: 9

File class

- The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java.
- The File class is not meant for handling the contents of files.
- An object of the File class provides a handle to a file or directory in the file system, and can be used to create, rename, and delete the entry.
- A File object can also be used to query the file system for information about a file or directory:
  - whether the entry exists
  - whether the File object represents a file or directory
  - get and set read, write, or execute permissions for the entry
  - get pathname information about the file or directory

File class

- Many methods of the File class throw a SecurityException in the case of a security violation, for example if read or write access is denied.
- SecurityException is a subclass of RuntimeException.
- Some methods also return a boolean value to indicate whether the operation was successful.

Creating New Files

- The File class can be used to create files and directories. A file can be created whose pathname is specified in a File object.

  boolean createNewFile() throws IOException

- It creates a new, empty file named by the abstract pathname if, and only if, a file with this name does not already exist.
- The returned value is true if the file was successfully created, false if the file already exists.
- Any I/O error results in an IOException.
- Constructor for creating abstract pathname.

  public File(String pathname)

- Creates a new File instance by converting the given pathname string into an abstract pathname. If the given string is the empty string, then the result is the empty abstract pathname.

Creating New Files

```
import java.io.File;
import java.io.IOException;
 class CreateFile {
            public static void main(String args[]) {
           try{
             File f=new File("C:\\Users\\KSB\\Desktop\\Java
             Programming\\Program\\FileCreateExample.txt");
             if(f.createNewFile())
             {
                        System.out.println("File"+ f.getName()+" is created successfully.");}
             else{
                        System.out.println("File is already exist in the directory.");
             }}
             catch(IOException exception){
                        System.out.println("An unexpected error is occurred.");
                        exception.printStackTrace();}
             }
}
```

Creating New Files

```
import java.io.File;
import java.io.IOException;
 class CreateFile {
        public static void main(String args[]) {
        try{
         File f=new File("C:\\Users\\KSB\\Desktop\\Java Programming\\Program\\FileCreateExample.txt");
         if(f.createNewFile())
         {
                System.out.println("File"+ f.getName()+" is created successfully.");}
         else{
                System.out.println("File is already exist in the directory.");
         }}
         catch(IOException exception){
                System.out.println("An unexpected error is occurred.");
                exception.printStackTrace();}
         }
}
```

Output:

File FileCreateExample.txt is created successfully.

Creating New Directories

- A directory whose pathname is specified in a File object can be created.

  boolean mkdir()

- The mkdirs() method creates any intervening parent directories in the pathname of the directory to be created.

Renaming Files and Directories

- A file or a directory can be renamed, using the following method which takes the new pathname from its argument.
- It throws a SecurityException if access is denied.

```
boolean renameTo(File dest)
```

Deleting Files and Directories

- A file or a directory can be deleted using the following method. In the case of a directory, it **must be empty** before it can be deleted.
- It throws a SecurityException if access is denied.

boolean delete()

Querying the File System

- The File class provides a number of methods for obtaining the platform-dependent representation of a pathname and its components.

  String getName()

- Returns the name of the file entry, excluding the specification of the directory in which it resides.

  String getPath()

- The method returns the pathname of the file represented by the File object.

Querying the File System

String getParent()

- The parent part of the pathname of this File object is returned if one exists, otherwise the null value is returned.

long lastModified()

- The modification time returned is encoded as a long value, and should only be compared with other values returned by this method.

long length()

- Returns the size (in bytes) of the file represented by the File object.

Querying the File System

boolean equals(Object obj)

- This method just compares the pathnames of the File objects, and returns true if they are identical.
- On Unix systems, alphabetic case is significant in comparing pathnames; on Windows systems it is not.

File or Directory Existence

boolean exists()

- A File object is created using a pathname. Whether this pathname denotes an entry that actually exists in the file system can be checked using the exists() method.

boolean isFile()

boolean isDirectory()

- File object can represent a file or a directory, boolean isFile() and boolean isDirectory() method can be used to distinguish whether a given File object represents a file or a directory.

File and Directory Permissions

boolean setReadable(boolean readable)

boolean setReadable(boolean readable, boolean owner)

boolean setWritable(boolean writable)

boolean setWritable(boolean writable, boolean owner)

boolean setExecutable(boolean executable)

boolean setExecutable(boolean executable, boolean owner)

- Write, read and execute permissions can be set.
- If the first argument is true, the operation permission is set; otherwise it is cleared.
- If the second argument is true, the permission only affects the owner; otherwise it affects all users.
- These methods throw a SecurityException if permission cannot be changed.

File and Directory Permissions

boolean canWrite()

boolean canRead()

boolean canExecute()

- To check whether the specified file has write, read, or execute permissions.
- They throw a SecurityException if general access is not allowed, i.e., the application is not even allowed to check whether it can read, write or execute a file..