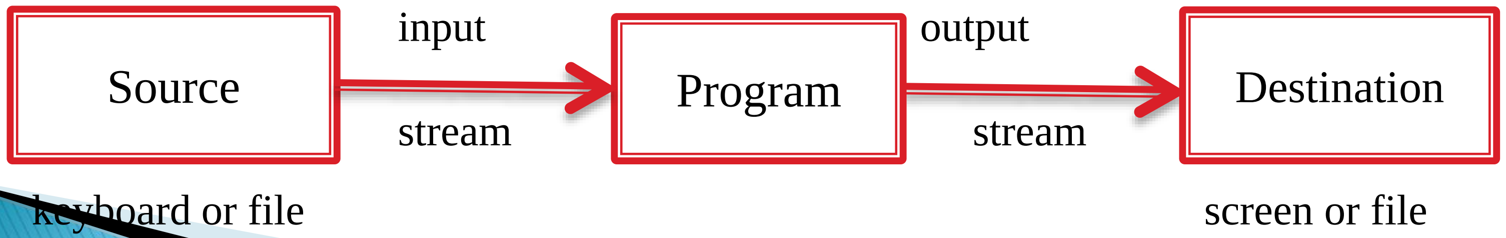
Chapter 2
Systems Programming

Introduction

- Java uses the concept of a stream to make I/O operation fast.
- I/O means Input/Output.
- The java.io package contains all the classes required for input and output operations.
- A stream can be defined as a sequence of data.
- An I/O Stream is a sequence of data that is read from a source and write to a destination.
- All these streams represent an input source and an output destination.



2.1 File Descriptors

- Files
  - collections of data stored on a storage device, such as a hard disk or flash drive.
  - In Java, files can contain various types of data, including text, images, audio, video, and more.
- File Descriptors
  - File descriptors are unique identifiers assigned to open files by the operating system.
  - Serve as handles that enable Java applications to interact with files at a low level.
  - File descriptors are crucial for performing file operations in Java, including reading, writing, closing, and seeking within files.

2.2 Reading and Writing Files

- File streams can be used to input and output data as bytes or characters.
  - Streams that input and output bytes are known as **byte-based streams**, representing data in its binary format.
  - Streams that input and output characters are known as **character-based streams**, representing data as a sequence of characters.
- Files that are created using byte-based streams are referred to as **binary files.**
- Files created using character-based streams are referred to as **text files.** Text files can be read by text editors.
- Data stored in a Text file are represented in human-readable form.
- Data stored in a binary file are represented in binary form. You cannot read **binary files**.
- Binary files are designed to be **read by programs**.

2.2 Reading and Writing Files

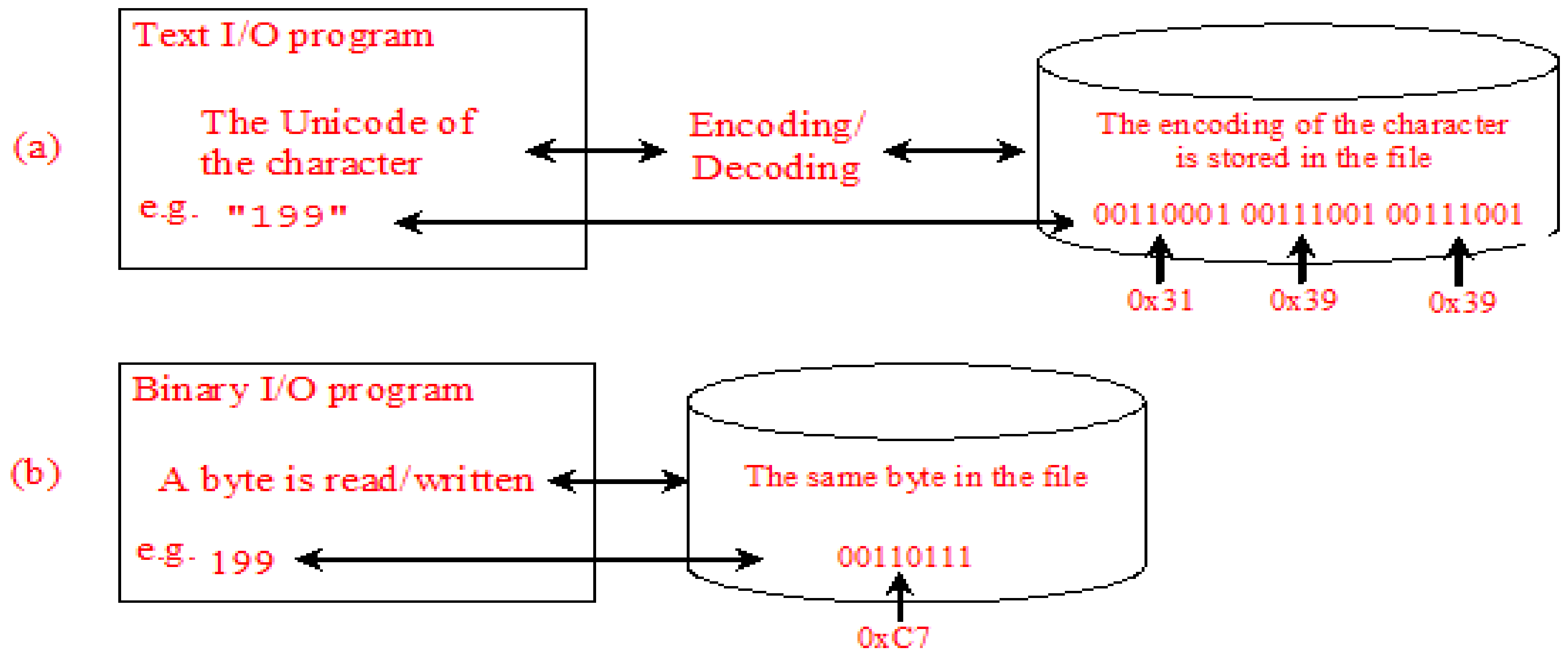
- The advantage of binary files
  - I/O are more efficient to process than text files I/O, because binary file I/O does not require encoding and decoding.
- But Text file I/O requires encoding and decoding.
- For example:
  - the decimal integer 199 is stored as the sequence of three characters: '1', '9', '9' in a text file
  - the same integer is stored as a byte-type value C7 in a binary file, because decimal 199 equals to hex C7.
- Computers do not differentiate between binary files & text files.
- All files are stored in binary format, and thus all files are essentially binary files. Text I/O requires encoding and decoding.

2.2 Reading and Writing Files

Byte I/O Streams

- For example, suppose you write the string "199" using text I/O to a file. The ASCII or Unicode code for character 1 is 49 (0x31 in hex) and for character 9 is 57 (0x39 in hex).
- Thus, to write the characters 199, three bytes— 0x31, 0x39, and 0x39—are sent to the output, as shown in Figure (a) below.
- Binary I/O does not require conversions.
- When you write a byte to a file, the original byte is copied into the file. When you read a byte from a file, the exact byte in the file is returned.
- For example, a byte-type value 199 is represented as 0xC7 in the memory and appears exactly as 0xC7 in the file, as shown in Figure (b) below.

2.2 Reading and Writing Files



Binary I/O is more efficient than text I/O, because binary I/O does not require **encoding** and **decoding**. Binary files are **independent** of the encoding scheme on the host machine and thus are portable. Java programs on any machine can read a binary file created by a Java program. This is why Java class files are binary files. Java class files can run on a JVM on any machine.

2.2 Reading and Writing Files

Byte I/O Streams

- It is an input and output data in its binary format.
- Java byte streams are used to perform input and output of **8-bit** bytes.
- All byte stream classes are descended from two abstract classes: **InputStream** and **OutputStream**.
- The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.
- There are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**.

Example: FileInputStream/FileOutputStream

```
import java.io.*;
public class TestFileStream {
  public static void main(String[] args) thorw IOException {
  // Create an output stream to the file
  FileOutputStream output = new
  FileOutputStream("temp.dat");
  // Output values to the file
   for (int i = 1; i <= 10; i++)
   output.write(i);
  // Close the output stream
  output.close();
}
```

This program uses binary I/O to write ten byte values from **1** to **10** to a file named **temp.dat** and reads them back from the file.

Example: FileInputStream/FileOutputStream

```
   // Create an input stream for the file
    FileInputStream input = new FileInputStream("temp.dat");
 // Read values from the file
    int value;
    while ((value = input.read() )!= -1)
    System.out.print(value + " ");

    // Close the output stream
    input.close();
     }
}
```

This program uses binary I/O to write ten byte values from **1** to **10** to a file named **temp.dat** and reads them back from the file.

Characters and Strings in Byte I/O

- A Unicode consists of two bytes. The writeChar(char c) method writes the Unicode of character c to the output.
- The writeChars(String s) method writes the Unicode for each character in the string s to the output.
- The writeBytes(String s) method writes the lower byte of the Unicode for each character in the string s to the output.
- The high byte of the Unicode is discarded.
- The writeBytes method is suitable for strings that consist of ASCII characters, since an ASCII code is stored only in the lower byte of a Unicode.
- If a string consists of non-ASCII characters, you have to use the writeChars method to write the string.

Characters and Strings in Byte I/O

Why UTF-8? What is UTF-8?

- UTF-8 is a coding scheme that allows systems to operate with both ASCII and Unicode efficiently.
- Most operating systems use ASCII. Java uses Unicode. The ASCII character set is a subset of the Unicode character set.
- Since most applications need only the ASCII character set, it is a waste to represent an 8-bit ASCII character as a 16-bit Unicode character.
- The UTF-8 is an alternative scheme that stores a character using 1, 2, or 3 bytes.
- ASCII values (less than 0x7F) are coded in one byte. Unicode values less than 0x7FF are coded in two bytes. Other Unicode values are coded in three bytes.

DataInputStream/DataOutputStream

- Data streams are used as wrappers on existing input and output streams to filter data in the original stream. They are created using the following constructors:

```
public DataInputStream(InputStream instream)
public DataOutputStream(OutputStream outstream)
```

- The statements given below create data streams. The first statement creates an input stream for file **in.dat**; the second statement creates an output stream for file **out.dat**.

```
DataInputStream infile =  new DataInputStream(new FileInputStream("in.dat"));
DataOutputStream outfile =  new DataOutputStream(new
FileOutputStream("out.dat"));
```

Example: DataInputStream/DataOutputStream

```
import java.io.*;
  public class TestDataStream {
   public static void main(String[] args) throws IOException {
   // Create an output stream for file temp.dat
  DataOutputStream output = new DataOutputStream(new
  FileOutputStream("temp.dat"));
  // Write student test scores to the file
   output.writeUTF("John");
   output.writeDouble(85.5);
   output.writeUTF("Susan");
   output.writeDouble(185.5);
   output.writeUTF("Kim");
   output.writeDouble(105.25);
   // Close output stream
   output.close();
  }
```

This program writes student names and scores to a file named **temp.dat** and reads the data back from the file.

Example: DataInputStream/DataOutputStream

```
  // Create an input stream for file temp.dat
 DataInputStream input = new DataInputStream(new
 FileInputStream("temp.dat"));

  // Read student test scores from the file
  System.out.println(input.readUTF()+ " " + input.readDouble());
  System.out.println(input.readUTF() + " " + input.readDouble());
  System.out.println(input.readUTF() + " " + input.readDouble());
  }
}
```

This program writes student names and scores to a file named **temp.dat** and reads the data back from the file.

Order and Format

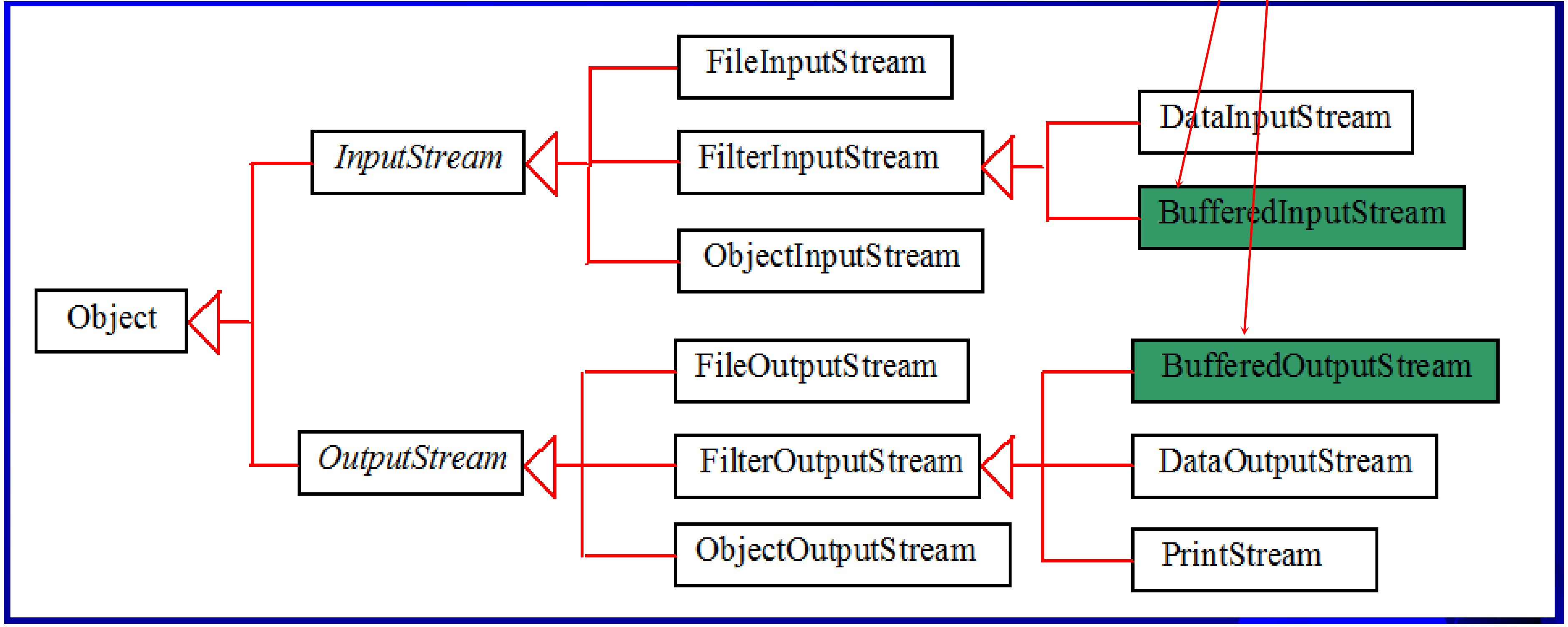
CAUTION: You have to read the data in the same order and same format in which they are stored. For example, since names are written in UTF-8 using writeUTF, you must read names using readUTF.

Checking End of File

TIP: If you keep reading data at the end of a stream, an EOFException would occur. So how do you check the end of a file? You can use input.available() to check it. input.available() == 0 indicates that it is the end of a file.

BufferedInputStream/BufferedOutputStream

Using buffers to speed up I/O

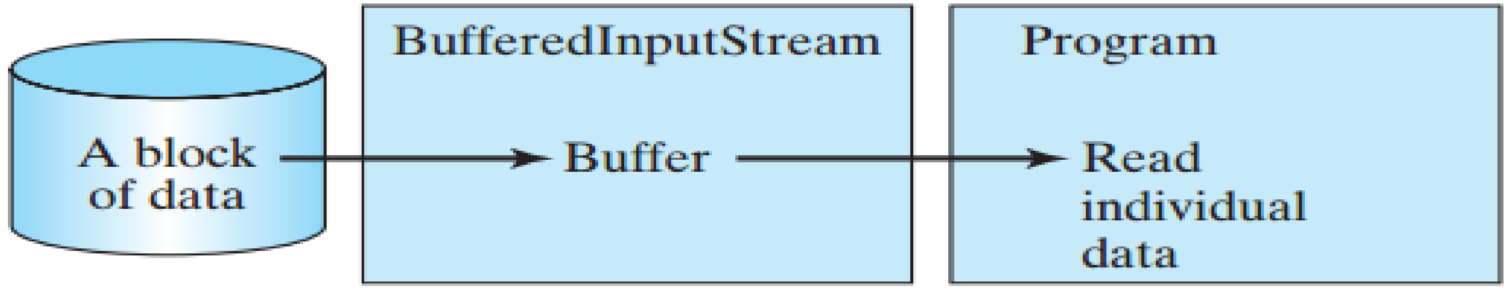


BufferedInputStream/BufferedOutputStream does not contain new methods. All the methods BufferedInputStream/BufferedOutputStream are inherited from the InputStream/OutputStream classes. **BufferedInputStream/BufferedOutputStream** manages a buffer behind the scene and automatically reads/writes data from/to disk on demand.

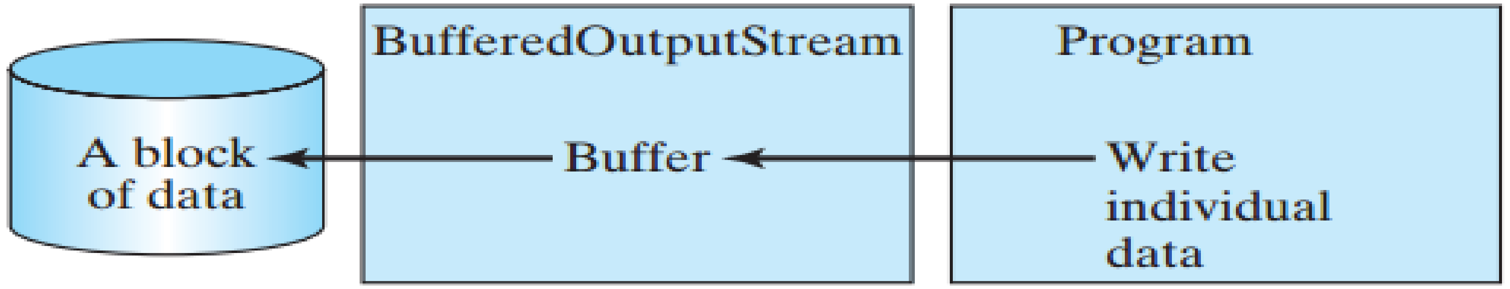
BufferedInputStream/BufferedOutputStream

- **BufferedInputStream/BufferedOutputStream** can be used to speed up input and output by reducing the number of disk reads and writes.
- Using **BufferedInputStream**, the whole block of data on the disk is read into the buffer in the memory once.
- The individual data are then delivered to your program from the buffer, as shown in Figure (a) below.
- Using **BufferedOutputStream**, the individual data are first written to the buffer in the memory.
- When the buffer is full, all data in the buffer is written to the disk once, as shown in Figure (b) below.

BufferedInputStream/BufferedOutputStream



(a)

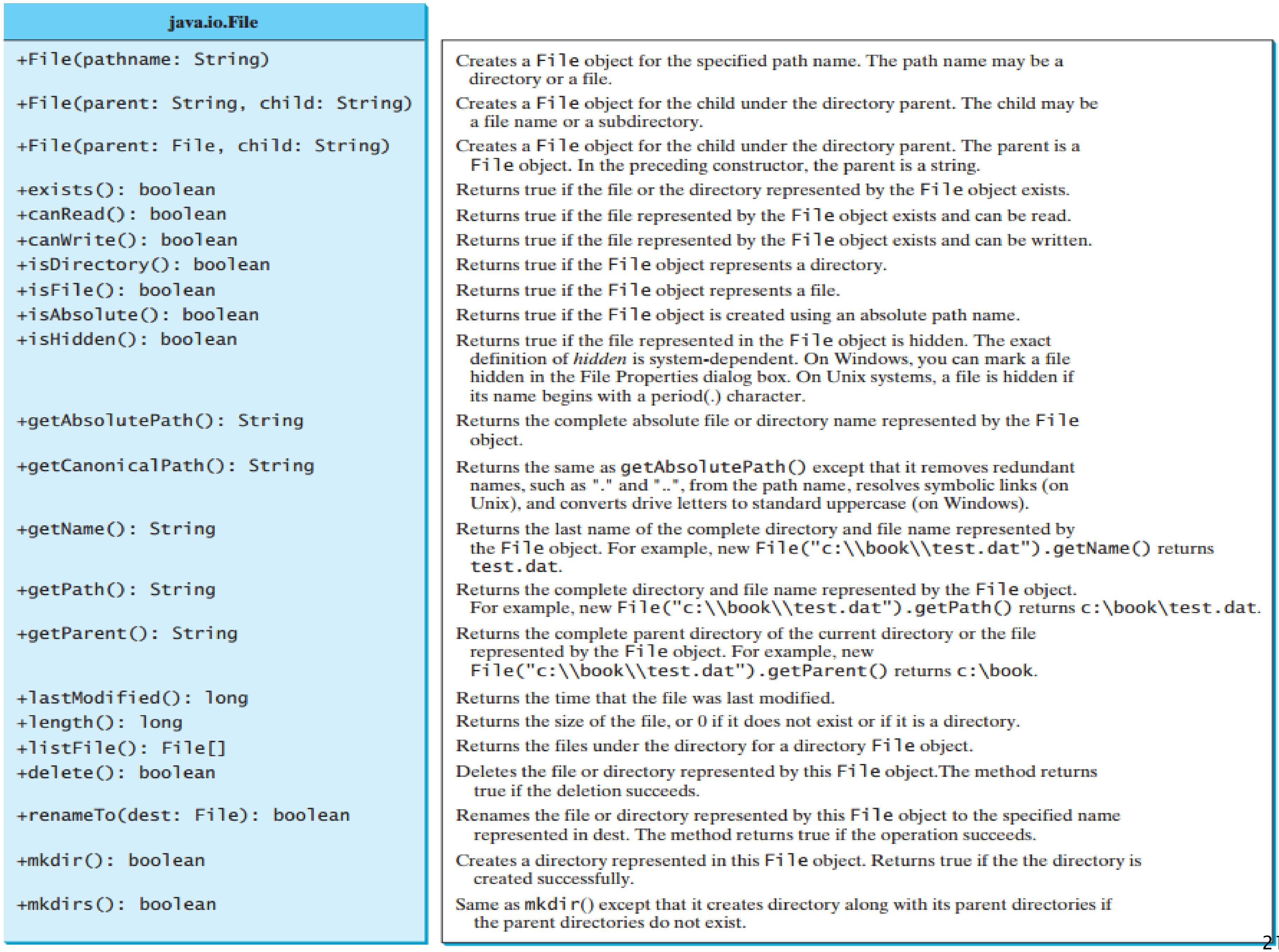


(b)

2.3 Files and Directories

The Java File Class

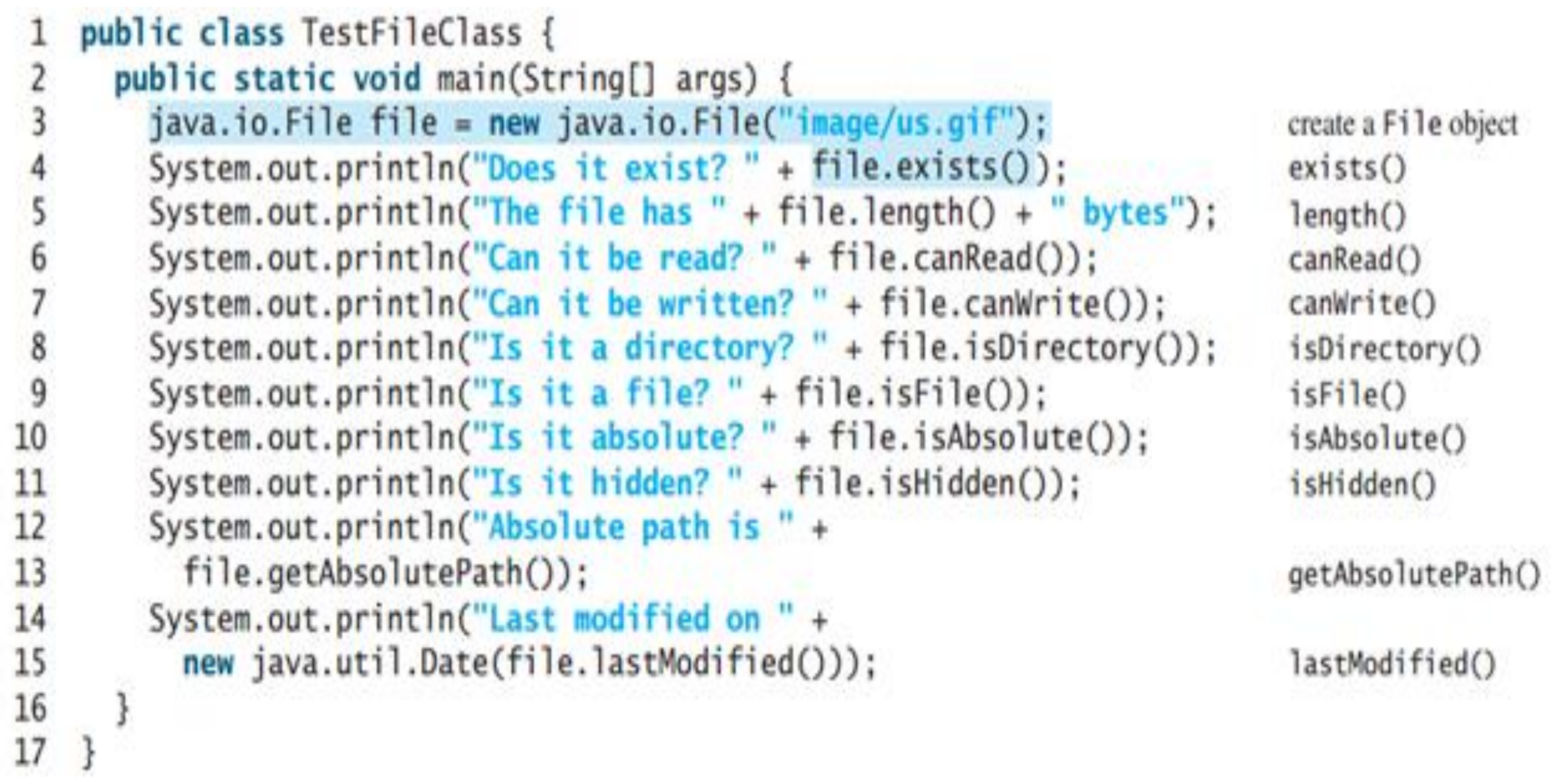
- Java File class represents the files & directory pathnames in an abstract manner.
- The File class is useful for retrieving information about files or directories from disk.
- The **File** class contains the methods for obtaining the properties of a file/directory and for renaming and deleting a file/directory.
- The File class has different methods which can be used to manipulate the files.
  - An **absolute path** contains all the directories, starting with the **root directory**, that lead to a specific file or directory.
  - A **relative path** normally starts from the directory in which the application began executing and is therefore "relative" to the current directory.



java.io.File	
`+File(pathname: String)`	Creates a `File` object for the specified path name. The path name may be a directory or a file.
`+File(parent: String, child: String)`	Creates a `File` object for the child under the directory parent. The child may be a file name or a subdirectory.
`+File(parent: File, child: String)`	Creates a `File` object for the child under the directory parent. The parent is a `File` object. In the preceding constructor, the parent is a string.
`+exists(): boolean`	Returns true if the file or the directory represented by the `File` object exists.
`+canRead(): boolean`	Returns true if the file represented by the `File` object exists and can be read.
`+canWrite(): boolean`	Returns true if the file represented by the `File` object exists and can be written.
`+isDirectory(): boolean`	Returns true if the `File` object represents a directory.
`+isFile(): boolean`	Returns true if the `File` object represents a file.
`+isAbsolute(): boolean`	Returns true if the `File` object is created using an absolute path name.
`+isHidden(): boolean`	Returns true if the file represented in the `File` object is hidden. The exact definition of *hidden* is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period(.) character.
`+getAbsolutePath(): String`	Returns the complete absolute file or directory name represented by the `File` object.
`+getCanonicalPath(): String`	Returns the same as `getAbsolutePath()` except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).
`+getName(): String`	Returns the last name of the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getName()` returns `test.dat`.
`+getPath(): String`	Returns the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getPath()` returns `c:\book\test.dat`.
`+getParent(): String`	Returns the complete parent directory of the current directory or the file represented by the `File` object. For example, new `File("c:\\book\\test.dat").getParent()` returns `c:\book`.
`+lastModified(): long`	Returns the time that the file was last modified.
`+length(): long`	Returns the size of the file, or 0 if it does not exist or if it is a directory.
`+listFile(): File[]`	Returns the files under the directory for a directory `File` object.
`+delete(): boolean`	Deletes the file or directory represented by this `File` object.The method returns true if the deletion succeeds.
`+renameTo(dest: File): boolean`	Renames the file or directory represented by this `File` object to the specified name represented in dest. The method returns true if the operation succeeds.
`+mkdir(): boolean`	Creates a directory represented in this `File` object. Returns true if the the directory is created successfully.
`+mkdirs(): boolean`	Same as `mkdir()` except that it creates directory along with its parent directories if the parent directories do not exist.

2.3 Files and Directories

This program demonstrates how to create a **File** object and use the methods in the **File** class to obtain its properties.



```java
public class TestFileClass {
  public static void main(String[] args) {
    java.io.File file = new java.io.File("image/us.gif");
    System.out.println("Does it exist? " + file.exists());
    System.out.println("The file has " + file.length() + " bytes");
    System.out.println("Can it be read? " + file.canRead());
    System.out.println("Can it be written? " + file.canWrite());
    System.out.println("Is it a directory? " + file.isDirectory());
    System.out.println("Is it a file? " + file.isFile());
    System.out.println("Is it absolute? " + file.isAbsolute());
    System.out.println("Is it hidden? " + file.isHidden());
    System.out.println("Absolute path is " +
      file.getAbsolutePath());
    System.out.println("Last modified on " +
      new java.util.Date(file.lastModified()));
  }
}
```

The **lastModified()** method returns the date and time when the file was last modified.

2.4 File locking

- File locking is a mechanism that allows you to prevent multiple processes or threads from concurrently accessing or modifying the same file.
- In Java, you can use the java.nio.channels package to lock a file or a portion of a file.
- There are two types of file locks: shared locks and exclusive locks.
- A shared lock allows multiple processes or threads to read from the locked file, but only one process or thread can hold an exclusive lock, which allows it to read from and write to the locked file.

2.4 File locking

```
import java.io.*;
import java.nio.channels.*;  import java.nio.file.*;
public class SharedLockExample {
    public static void main(String[] args) {
        Path filePath = Paths.get("example.txt");
        try (FileChannel channel = FileChannel.open(filePath,
StandardOpenOption.READ)) { // Acquire a shared lock
            FileLock lock = channel.lock(0, Long.MAX_VALUE, true); // Shared
lock
            // Perform read operations on the locked file, // Multiple threads or
processes can read from the file concurrently,  // Release the lock after use
            lock.release();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }}
```

2.4 File locking

```
import java.io.*;
import java.nio.channels.*; import java.nio.file.*;
public class ExclusiveLockExample {
    public static void main(String[] args) {
        Path filePath = Paths.get("example.txt");
        try (FileChannel channel = FileChannel.open(filePath,
StandardOpenOption.READ, StandardOpenOption.WRITE)) {
            // Acquire an exclusive lock
            FileLock lock = channel.lock(); // Exclusive lock
            // Perform read and write operations on the locked file, // Only one
thread or process can access the file at a time      // Release the lock after use
            lock.release();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }}
```

2.5 Memory-mapped I/O

- Memory-mapped I/O (MMIO) enables accessing hardware devices from a program by mapping their registers into the computer's address space.
- Java supports MMIO through the java.nio package, simplifying interaction with hardware.
- This technique treats hardware devices as normal memory locations, enhancing software-hardware communication.
- Memory-mapped buffers in Java are created using the java.nio.MappedByteBuffer class.
- This class provides methods for reading from and writing to mapped buffers, offering a convenient interface for hardware interaction.

2.5 Memory-mapped I/O

```
// Java code example demonstrating creation of a memory-
mapped buffer
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
public class MemoryMappedIOExample {
    public static void main(String[] args) {
        // Specify the file to be memory-mapped
        File file = new File("example.txt");
        try (RandomAccessFile raf = new
RandomAccessFile(file, "rw");
             FileChannel channel = raf.getChannel()) {
```

2.5 Memory-mapped I/O

```
// Map the file into memory
            MappedByteBuffer buffer =
channel.map(FileChannel.MapMode.READ_WRITE, 0,
channel.size());
            // Perform read and write operations on the mapped buffer
            // Unmap the buffer after use
            buffer.force();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2.5 Memory-mapped I/O

```
// Another Example
import java.io.RandomAccessFile;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;
public class MemoryMappedIOExample {
  public static void main(String[] args) throws
    Exception {
   RandomAccessFile file = new
     RandomAccessFile("example.txt", "rw");
   FileChannel channel = file.getChannel();
   // Map the file into memory
  MappedByteBuffer buffer =
  channel.map(FileChannel.MapMode.READ_WRITE, 0,
     channel.size());
```

2.5 Memory-mapped I/O

```
// Write to the buffer
buffer.putChar('H');
buffer.putChar('e');
buffer.putChar('l');
buffer.putChar('l');
buffer.putChar('o');
// Flush changes to disk
buffer.force();
// Read from the buffer
buffer.position(0);
while (buffer.hasRemaining()) {
System.out.print(buffer.getChar()); }
// Unmap the buffer
channel.close();
file.close(); } }
```

2.5 Memory-mapped I/O

In this example, we create a **MappedByteBuffer** by mapping a file channel to memory using the **map** method.
We then write characters to the buffer and flush the changes to disk using the **force** method.
Finally, we read the characters from the buffer and unmap the buffer by closing the channel and file.

2.6 Creating Processes

- A process is an instance of a running program on a computer system.
- Characteristics:
  - Represents the execution of a set of instructions.
  - Has its own memory space containing code, data, and resources.
  - Managed by the operating system.
  - Executes independently and can perform tasks concurrently with other processes.
- Java provides mechanisms for creating and managing external processes, enabling interaction with system commands and external applications.
- Two common approaches for creating processes in Java are using the ProcessBuilder class and the Runtime.exec() method.

2.6 Creating Processes

- ProcessBuilder Class
  - The ProcessBuilder class in Java provides a flexible and convenient way to create and manage processes.
  - With ProcessBuilder, developers can define various attributes of the process, such as the command, arguments, environment variables, and working directory.
  - This class offers greater control and customization options for process creation compared to other methods.
  - ProcessBuilder enables the creation of robust and versatile applications that interact with external processes seamlessly.

```
import java.io.IOException;
public class ProcessBuilderExample {
    public static void main(String[] args) {
        // Create a new ProcessBuilder instance
        ProcessBuilder processBuilder = new ProcessBuilder();
        // Set the command and arguments
        processBuilder.command("ls", "-l");
        try {
            // Start the process
            Process process = processBuilder.start();
            // Wait for the process to exit and get the exit code
            int exitCode = process.waitFor();
            // Print the exit code
            System.out.println("Process exited with code " + exitCode);
        } catch (IOException | InterruptedException e) {
            // Handle any exceptions
            e.printStackTrace();
        }
    }
}
```

2.6 Creating Processes

- In this example, we create a ProcessBuilder instance for the "notepad.exe" program and pass it the "test.txt" file as an argument. We then start the external program using the start() method of the ProcessBuilder class. Finally, we wait for the process to finish using the waitFor() method of the Process class, and print the exit code of the process.
- Alternatively, you can use the Runtime class to create a new process. Here's an example:

```
total 8
-rw-r--r-- 1 user user    0 Mar 23 14:28 file1.txt
-rw-r--r-- 1 user user    0 Mar 23 14:28 file2.txt
Process exited with code 0
```

The exit code 0 indicates that the process exited successfully.

2.7 Process management

- It encompasses the creation, scheduling, execution, and termination of processes.

Key Components of Process Management

- Process Control Block (PCB):
  - Contains essential information about each process, such as process ID, state, priority, and resource allocation.
- Process Scheduling:
  - Determines the order in which processes are executed on the CPU, using scheduling algorithms like round-robin or priority-based scheduling.
- Inter-Process Communication (IPC):
  - Facilitates communication and data exchange between processes, employing mechanisms such as shared memory, message passing, and synchronization.

2.7 Process management

Process Lifecycle

- Creation:
  - A process is initiated when a program is loaded into memory and commences execution.
- Execution:
  - During execution, the process executes its instructions, utilizing system resources and interacting with other processes.
- Termination:
  - The process concludes its task or is terminated by the operating system, leading to the release of allocated resources.

2.7 Process management

Process States

- Running:
  - The process is actively executing on the CPU.
- Ready:
  - The process is primed for execution but awaits CPU allocation.
- Blocked:
  - The process is in a wait state, pending an external event, such as I/O completion, before it can resume execution.
- Terminated:
  - The process has finalized its execution and is designated for termination by the operating system.

2.7 Process management

In Java, you can manage processes using the Process class, which represents an external process that is running on the operating system. The Process class provides methods for controlling and monitoring the execution of the process.

Here are some of the methods provided by the Process class:

•**destroy()**: Terminates the process abruptly.

•**waitFor()**: Waits for the process to terminate and returns the exit code of the process.

•**exitValue()**: Returns the exit code of the process if it has already terminated.

•**isAlive()**: Returns true if the process is still running.

•**getInputStream()**, **getOutputStream()**, and **getErrorStream()**: Provide input, output, and error streams for the process, respectively.

2.7 Process management

```
import java.io.*;
public class ProcessManagementExample {
    public static void main(String[] args) throws IOException,
InterruptedException {
        // Start the external program
        Process process = Runtime.getRuntime().exec("notepad.exe
test.txt");
        // Check if the process is still running
        if (process.isAlive()) {
            // Print the process ID
            System.out.println("Process ID: " + process.pid());
            // Get the input and error streams of the process
```

2.7 Process management

```
BufferedReader input = new BufferedReader(new
InputStreamReader(process.getInputStream()));
            BufferedReader error = new BufferedReader(new
InputStreamReader(process.getErrorStream()));
            // Print the output and error messages of the process
            String line;
            while ((line = input.readLine()) != null) {
                System.out.println("Output: " + line);
            }
            while ((line = error.readLine()) != null) {
                System.out.println("Error: " + line);
            }
            // Wait for the process to finish
            int exitCode = process.waitFor();
            System.out.println("Process exited with code " + exitCode);
        }
    }
```

2.7 Process management

- In this example, we start the external program "notepad.exe" and check if it is still running using the isAlive() method.
- We then print the process ID, and get the input and error streams of the process using the getInputStream() and getErrorStream() methods.
- We then print the output and error messages of the process, and wait for it to finish using the waitFor() method. Finally, we print the exit code of the process.

2.8 Pipes and Signals

- Pipes
- Inter-process communication (IPC) is vital for data exchange between concurrent processes.
- Pipes serve as a mechanism to facilitate IPC, allowing seamless data transmission.
- Pipes enable unidirectional communication between two processes.
- In Java, pipes are implemented using the PipedInputStream and PipedOutputStream classes

2.8 Pipes and Signals

```
import java.io.*;
public class PipeExample {
 public static void main(String[] args) throws IOException {
   // Create a PipedInputStream and a PipedOutputStream
   PipedInputStream in = new PipedInputStream();
   PipedOutputStream out = new PipedOutputStream(in);
   // Write some data to the output stream
   out.write("Hello, world!".getBytes());
```

2.8 Pipes and Signals

```
    // Read the data from the input stream
    byte[] buffer = new byte[1024];
    int bytesRead = in.read(buffer);
    String message = new String(buffer, 0, bytesRead);
    System.out.println(message);
    // Close the streams out.close();
    in.close();
    }
}
```

2.8 Pipes and Signals

- In this example, we create a PipedInputStream and a PipedOutputStream. We write some data to the output stream using the write() method of the PipedOutputStream, and then read the data from the input stream using the read() method of the PipedInputStream.
- Signals allow you to send notifications between processes. In Java, you can use the Signal class to send and receive signals.
- However, note that signal handling is platform-specific and may not work on all operating systems. Here's an example:

2.8 Pipes and Signals

1. Introduction to Inter-Process Communication (IPC):
   - IPC enables communication and synchronization between concurrent processes in a system.
   - Signals serve as lightweight notifications for processes to respond to external events or requests.
2. Understanding Signals:
   - Signals are asynchronous notifications sent to processes by the operating system.
   - In Java, signal handling is facilitated through the **sun.misc.Signal** class.
3. Signal Handling Process:
   - Register signal handlers using the Signal.handle() method to specify actions for signal reception.
   - Implement signal handler methods to define custom behavior for processing specific signals.

2.8 Pipes and Signals

- Standard Signals:
  - a. SIGINT (Interrupt): Indicates a request to interrupt the process, typically triggered by pressing Ctrl+C.
  - b. SIGTERM (Termination): Requests the process to terminate gracefully, allowing cleanup operations.
  - c. SIGILL (Illegal Instruction): Indicates an attempt to execute an invalid or illegal CPU instruction.
  - d. SIGSEGV (Segmentation Violation): Indicates an attempt to access memory that is not allowed.
- Additional Standard Signals:
  - a. SIGABRT (Abort): Indicates an abnormal termination of the process.
  - b. SIGFPE (Floating Point Exception): Indicates an error in floating-point arithmetic operations.

2.8 Pipes and Signals

```
import sun.misc.Signal;
import sun.misc.SignalHandler;
public class SignalExample {
   public static void main(String[] args) throws InterruptedException
  {
     // Create a signal handler
     SignalHandler handler = new SignalHandler() {
    @Override
    public void handle(Signal signal) {
      System.out.println("Received signal: " + signal.getName());
   }
};
```

2.8 Pipes and Signals

```
    // Register the signal handler for the TERM signal
    Signal.handle(new Signal("TERM"), handler);
    // Wait for the TERM signal
    Thread.sleep(Long.MAX_VALUE);
  }
}
```

2.8 Pipes and Signals

- In this example, we create a signal handler that prints a message when a signal is received.
- We register the signal handler for the TERM signal using the Signal.handle() method, and then wait for the TERM signal using the Thread.sleep() method.
- Note that signal handling is not recommended for general use in Java, and should be used only when necessary.
- Also note that the Signal class is not part of the public API and may not be available on all platforms.