

MULTITHREADED PROGRAMMING AND INPUT/OUTPUT: EXPLORING JAVA.IO

4.1 INTRODUCTION TO MULTITHREADING

There are two distinct types of multitasking:

Process based and Thread-based.

- **Process** is a program that is executing.
 - Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently.
 - Ex: listening music and typing
 - In process based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
 - **Thread-based multitasking:** Thread is the smallest unit of dispatch-able code.
 - This means that a single program can perform two or more tasks simultaneously.
 - For instance, a text editor can format text at the same time that it is printing.
-
- Multitasking threads require less overhead than multitasking processes.
 - Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. because:
 - Local file system resources are read and written at a much slower pace than they can be processed by the CPU. And, of course, user input is much slower than the computer.
 - In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though the CPU is sitting idle most of the time.
 - Multithreading lets you gain access to this idle time and put it to good use.

INPUT/OUTPUT:

4.2 Exploring java.io

- Programs cannot accomplish their goals without accessing external data. Data is retrieved from an input source. The results of a program are sent to an output destination. In Java, these sources or destinations are defined very broadly.
- For example, a network connection, memory buffer, or disk file can be manipulated by the Java I/O classes.
- Although physically different, these devices are all handled by the same abstraction: **the stream**.
- *A stream is a logical entity that either produces or consumes information.* A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ.

The I/O classes defined by java.io are listed here:

BufferedInputStream	FileWriter	pipedReader
BufferedOutputStream	FilterInputStream	PipedReader
BufferedReader	FilterOutputStream	PipedWriter
BufferedWriter	FilterReader	PrintStream
ByteArrayInputStream	FilterWriter	PrintWriter
ByteArrayOutputStream	InputStream	PushbackInputStream
CharArrayReader	InputStreamReader	PushbackReader
CharArrayWriter	LineNumberReader	RandomAccessFile
Console	ObjectInputStream	Reader
DataInputStream	ObjectInputStream.GetField	SequenceInputStream
DataOutputStream	ObjectOutputStream	SerializablePermission
File	ObjectOutputStream.PutField	StreamTokenizer
FileDescriptor	ObjectStreamClass	StringReader
FileInputStream	ObjectStreamField	StringWriter
FileOutputStream	OutputStream	Writer
FilePermission	OutputStreamWriter	
FileReader	PipedInputStream	

The following interfaces are defined by java.io:

Closeable	FileFilter	ObjectInputValidation
DataInput	FilenameFilter	ObjectOutput
DataOutput	Flushable	ObjectStreamConstants
Externalizable	ObjectInput	Serializable

4.2 THE JAVA THREAD MODEL

- The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind.
- In fact, Java uses threads to enable the entire environment to be asynchronous.
- This helps reduce inefficiency by preventing the waste of CPU cycles.
- **Threads exist in several states.**
 - **A thread can be *running*.** *It can be ready to run as soon as it gets CPU time.*
 - **A running thread can be *suspended*,** *which temporarily suspends its activity.*
 - **A suspended thread can then be *resumed*,** *allowing it to pick up where it left off.*
 - **A thread can be *blocked* when waiting for a resource.**
 - **At any time, a thread can be *terminated*,** *which halts its execution immediately.*
 - Once terminated, a thread cannot be resumed.

The Thread Class and the Runnable Interface:

- To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

The **Thread** class defines several methods that help manage threads few among them are:

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
Join	Wait for a thread to terminate.
Run	Entry point for the thread.
Sleep	Suspend a thread for a period of time.
Start	Start a thread by calling its run method.

4.3 THE MAIN THREAD

- When a Java program starts up, one thread begins running immediately that is main.
- The main thread is important for two reasons:
 - This is the main thread from where other threads will be called
 - Often, it must be the last thread to finish execution because it performs various shutdown actions.

Note:

- Although the main thread is created automatically when your program is started, it can be controlled through a **Thread object**.
- To do so, you must obtain a reference to it by calling the method `currentThread()`, which is a public static member of `Thread`.

Its general form is shown here:

– **static Thread `currentThread()`**

- This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Example program of Main Thread:

```
class CurrentThreadDemo
{
    public static void main(String args[]) {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);
        // change the name of the thread
        t.setName("My Thread");
        System.out.println("After name change: " + t);
        try {
            for(int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
    }
}
```

Output of Program:

Current thread: Thread[main,5,main]

After name change: Thread[My Thread,5,main]

5

4

3

2

1

- Notice the output produced when `t` is used as an argument to `println()`.
- This displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is `main`.
- Its priority is 5, which is the default value, and `main` is also the name of the group of threads to which this thread belongs.

- The **`sleep()`** method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is shown here:

static void `sleep(long milliseconds)` throws `InterruptedException`

- The number of milliseconds to suspend is specified in *milliseconds*. This method may throw an `InterruptedException`.

- The **sleep()** method has a second form, shown next, which allows you to specify the period in terms of milliseconds and nanoseconds:

static void sleep(long milliseconds, int nanoseconds) throws InterruptedException

- This second form is useful only in environments that allow timing periods as short as nanoseconds.
- You can obtain the name of a thread by calling **getName()**.

These methods are members of the Thread class and are declared like this:

- **final void setName(String threadName)**
- **final String getName()**

4.4 **CREATING A THREAD**

- In the most general sense, you create a thread by instantiating an object of type Thread.
- You can **implement the Runnable interface**.
- You can **extend the Thread class**, itself.

4.4.1 **Implementing Runnable**

- The easiest way to create a thread is to create a class that implements the **Runnable interface**.
- To implement Runnable, a class need only implement a single method called **run()**, which is declared like this:

public void run()

- Inside run(), you will define the code that constitutes the new thread.
- It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can.
- The only difference is that run() establishes the entry point for another, concurrent thread of execution within your program.
- This thread will end when run() returns.
- After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class.
- Thread defines several constructors.
- The one that we will use is shown here:

Thread(Runnable threadOb, String threadName)

Example program showing Creation of Thread using Runnable Interface:

```
class NewThread implements Runnable {  
    Thread t;  
    NewThread()  
    {  
        t = new Thread(this, "Demo Thread");  
        System.out.println("Child thread: " + t);  
        t.start(); // Start the thread  
    }  
    public void run() {  
        try {  
            for(int i = 5; i > 0; i--) {  
                System.out.println("Child Thread: " + i);  
                Thread.sleep(500);  
            }  
        } catch (InterruptedException e) {  
            System.out.println("Child interrupted.");  
        }  
        System.out.println("Exiting child thread.");  
    }  
}
```

```
class ThreadDemo {  
    public static void main(String args[]) {  
        new NewThread(); // create a new thread  
        try {  
            for(int i = 5; i > 0; i--) {  
                System.out.println("Main Thread: " + i);  
                Thread.sleep(1000);  
            }  
        }  
        catch (InterruptedException e) {  
            System.out.println("Main thread interrupted.");  
        }  
        System.out.println("Main thread exiting.");  
    }  
}
```

Output:

```
Child thread: Thread[Demo Thread,5,main]  
Main Thread: 5  
Child Thread: 5  
Child Thread: 4  
Main Thread: 4  
Child Thread: 3  
Child Thread: 2  
Main Thread: 3  
Child Thread: 1  
Exiting child thread.  
Main Thread: 2  
Main Thread: 1  
Main thread exiting.
```

4.4.2 Extending Thread

```
class NewThread extends Thread {
    NewThread()
    {
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Start the thread
    }

    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}
```

```
class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // create a new thread
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

Notice the call to **super()** inside NewThread.
This invokes the following form of the Thread constructor:

public Thread(String threadName)

Here, *threadName* specifies the name of the thread.

4.5 Creating Multiple Threads

```
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }

    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        }
    }
}
```

```
catch (InterruptedException e) {
    System.out.println(name + "Interrupted");
}
System.out.println(name + " exiting.");
}} // end of NewThread class
```

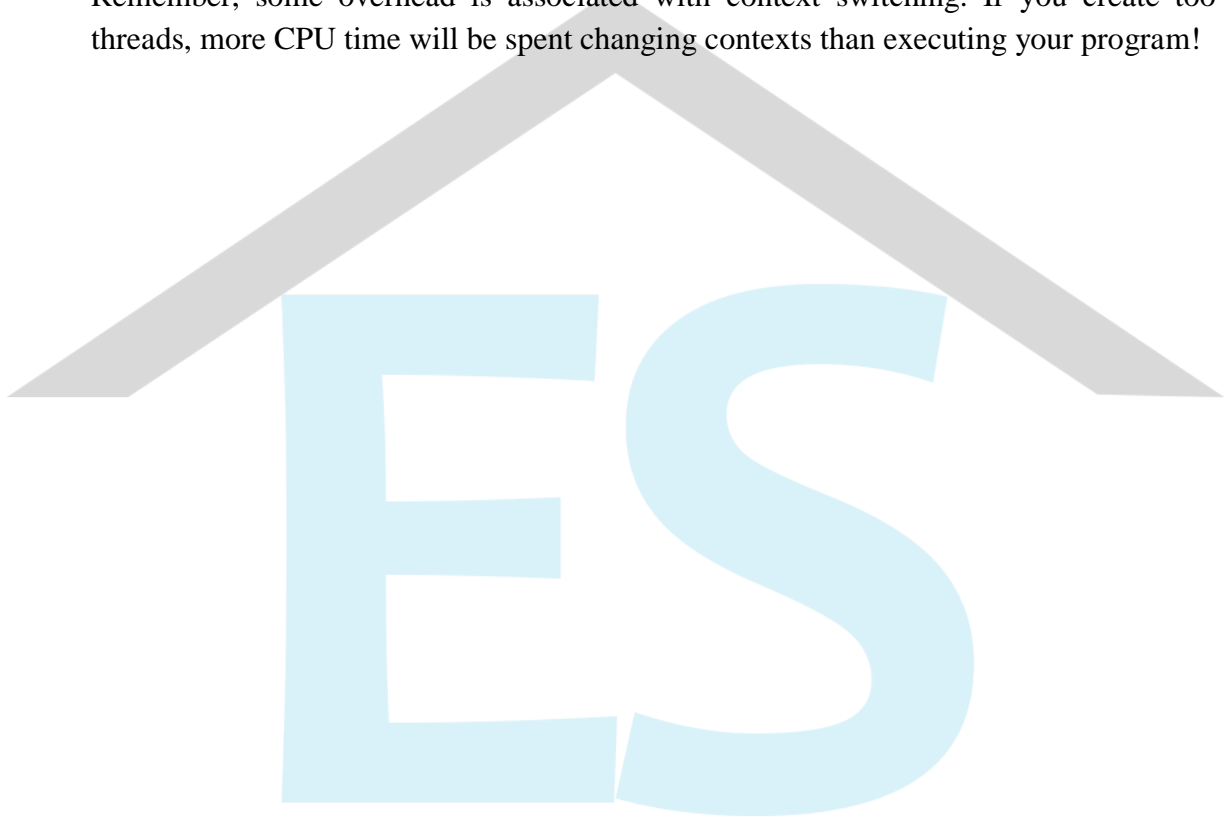
```
class MultiThreadDemo {
    public static void main(String args[]) {
        new NewThread("One");// start threads
        new NewThread("Two");
        new NewThread("Three");
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}
```

OUTPUT:

New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.

4.11 Using Multithreading

- The key to utilizing Java's multithreading features effectively is to think concurrently rather than serially.
- For example, when you have two subsystems within a program that can execute concurrently, make them individual threads.
- With the careful use of multithreading, you can create very efficient programs.
- **A word of caution:** *If you create too many threads, you can actually degrade the performance of your program rather than enhance it.*
- Remember, some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program!



Edusources.in

4.2.1 File

- **Files are a primary source and destination for data within many programs.**
- Although there are severe restrictions on their use within applets for security reasons, files are still a central resource for storing persistent and shared information.
- A directory in Java is treated simply as a File with one additional property—a list of filenames that can be examined by the `list()` method.

The following constructors can be used to create File objects:

`File(String directoryPath)`

`File(String directoryPath, String filename)`

`File(File dirObj, String filename)`

`File(URI uriObj)`

directoryPath→ It is the path name of the file, filename is the name of the file or subdirectory

dirObj→ It is a File object that specifies a directory

uriObj→ It is a URI object that describes a file.

- The following example creates three files: f1, f2, and f3.
- The first File object is constructed with a directory path as the only argument.
- The second includes two arguments—the path and the filename.
- The third includes the file path assigned to f1 and a filename; f3 refers to the same file as f2.

```
File f1 = new File("/");
```

```
File f2 = new File("/", "autoexec.bat");
```

```
File f3 = new File(f1, "autoexec.bat");
```

EduSources.in

The Program below demonstrates details about a File:

```
// Demonstrate File.
import java.io.File;
class FileDemo {
    static void p(String s) {
        System.out.println(s);
    }
    public static void main(String args[]) {
        File f1 = new File("/java/COPYRIGHT");
        p("File Name: " + f1.getName());
        p("Path: " + f1.getPath());
        p("Abs Path: " + f1.getAbsolutePath());
    }
}
```

```
p("Parent: " + f1.getParent());
p(f1.exists() ? "exists" : "does not exist");
p(f1.canWrite() ? "is writeable" : "is not writeable");
p(f1.canRead() ? "is readable" : "is not readable");
p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
p(f1.isFile() ? "is normal file" : "might be a named
pipe");
p(f1.isAbsolute() ? "is absolute" : "is not absolute");
p("File last modified: " + f1.lastModified());
p("File size: " + f1.length() + " Bytes");
}}
```

OUTPUT:

```
File Name: COPYRIGHT
Path: /java/COPYRIGHT
Abs Path: /java/COPYRIGHT
Parent: /java
exists
is writeable
is readable
is not a directory
is normal file
is absolute
File last modified: 812465204000
File size: 695 Bytes
```

- Most of the File methods are self-explanatory. `isFile()` and `isAbsolute()` are not.
- `isFile()` returns true if called on a file and false if called on a directory. Also, `isFile()` returns false for some special files, such as device drivers and named pipes, so this method can be used to make sure the file will behave as a file.
- The `isAbsolute()` method returns true if the file has an absolute path and false if its path is relative.

File also includes two useful utility methods:

1) **renameTo()**

2) **delete()**

1) **renameTo()** → used to rename the file

Syntax: `boolean renameTo(File newName)`

- Here, the filename specified by `newName` becomes the new name of the invoking File object.

- It will return true upon success and false if the file cannot be renamed

2) delete()

- It deletes the disk file represented by the path of the invoking File object

Syntax: boolean delete()

- *You can also use delete() to delete a directory if the directory is empty.*
- delete() returns true if it deletes the file and false if the file cannot be removed.

File Method Description:

void deleteOnExit() → Removes the file associated with the invoking object when the JVM terminates.

long getFreeSpace() → Returns the number of free bytes of storage available on the partition associated with the invoking object.

long getTotalSpace() → Returns the storage capacity of the partition associated with the invoking object. (Added by Java SE 6.)

long getUsableSpace() → Returns the number of usable free bytes of storage available on the partition associated with the invoking object. (Added by Java SE 6.)

isFile() → returns true if called on a file and false if called on a directory.

Also, isFile() returns false for some special files, such as device drivers and named pipes, so this method can be used to make sure the file will behave as a file.

isAbsolute() → method returns true if the file has an absolute path and false if its path is relative. File also includes two useful utility methods.

Edusources.in

4.2.2 The Closeable and Flushable Interfaces:

- The interfaces are implemented by several of the I/O classes.
- Their inclusion does not add new functionality to the stream classes.
- They simply offer a uniform way of specifying that a stream can be closed or flushed.

Objects of a class that implements Closeable can be closed

It defines the `close()` method, shown here:

`void close()` throws `IOException`

- This method closes the invoking stream, releasing any resources that it may hold.
- This interface is implemented by all of the I/O classes that open a stream that can be closed.

Objects of a class that implements Flushable can force buffered output to be written to the stream to which the object is attached.

It defines the `flush()` method, shown here:

`void flush()` throws `IOException`

- Flushing a stream typically causes buffered output to be physically written to the underlying device.
- This interface is implemented by all of the I/O classes that write to a stream.

4.2.3 The Stream Classes:

Java's stream-based I/O is built upon four abstract classes:

- 1) **`InputStream`**
- 2) **`OutputStream`**
- 3) **`Reader`**
- 4) **`Writer`**

- They are used to create several concrete stream subclasses.
- Although your programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.
- **`InputStream`** and **`OutputStream`** are designed for byte streams.
- **`Reader`** and **`Writer`** are designed for character streams.
- The byte stream classes and the character stream classes form separate hierarchies.

- In general, you should use the character stream classes when working with characters or strings, and use the byte stream classes when working with bytes or other binary objects.

4.2.4 The Byte Streams:

- The byte stream classes provide a rich environment for handling byte-oriented I/O.
- A byte stream can be used with any type of object, including binary data.

1) InputStream:

- InputStream is an abstract class that defines Java's model of streaming byte input.
- It implements the Closeable interface.
- Most of the methods in this class will throw an IOException on error conditions.

2) OutputStream:

- OutputStream is an abstract class that defines streaming byte output.
- It implements the Closeable and Flushable interfaces.
- Most of the methods in this class return void and throw an IOException in the case of errors.

Methods Defines by InputStream class:

<u>Method</u>	<u>Description</u>
int available()	Returns the number of bytes of input currently available for reading.
void close()	Closes the input source. Further read attempts will generate an IOException
void mark(int numBytes)	Places a mark at the current point in the input stream that will remain valid until numBytes bytes are read.
boolean markSupported()	Returns true if mark()/reset() are supported by the invoking stream.
int read()	Returns an integer representation of the next available byte of input. -1 is returned when the end of the file is encountered.
int read(byte buffer[])	Attempts to read up to buffer.length bytes into buffer and returns the actual number of bytes that were successfully read. -1 is returned when the end of the file is encountered.
int read(byte buffer[], int offset, int numBytes)	Attempts to read up to numBytes bytes into buffer starting at buffer[offset], returning the number of bytes successfully read. -1 is returned when the end of the file is encountered.
Void reset()	Resets the input pointer to the previously set mark
long skip(long numBytes)	Ignores (that is, skips) numBytes bytes of input, returning the number of bytes actually ignored.

Methods Defines by OutputStream class:

Method	Description
void close()	Closes the output stream. Further write attempts will generate an IOException
void flush()	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int b)	Writes a single byte to an output stream. Note that the parameter is an int, which allows you to call write() with expressions without having to cast them back to byte.
void write(byte buffer[])	Writes a complete array of bytes to an output stream.
void write(byte buffer[], int offse int numBytes)	Writes a subrange of numBytes bytes from the array buffer, beginning at buffer[offset].

4.2.5 FileInputStream

- The FileInputStream class creates an InputStream that you can use to read bytes from a file Its two most common constructors are shown here:

FileInputStream(String filepath)

FileInputStream(File fileObj)

- Either can throw a FileNotFoundException.
- Here, filepath is the full path name of a file, and fileObj is a File object that describes the file.

The Program below demonstrates an example of FileInputStream:

```
import java.io.*;
class FileInputStreamDemo {
public static void main(String args[]) throws IOException {
int size;
InputStream f =
new FileInputStream("FileInputStreamDemo.java");
System.out.println("Total Available Bytes: " + (size = f.available()));
int n = size/40;
System.out.println("First " + n + " bytes of the file one read() at a time");
for (int i=0; i < n; i++) {
System.out.print((char) f.read());
}
System.out.println("\nStill Available: " + f.available());
}
```

```

System.out.println("Reading the next " + n + " with one read(b[])");
byte b[] = new byte[n];
if (f.read(b) != n) {
System.err.println("couldn't read " + n + " bytes.");
}
System.out.println(new String(b, 0, n));
System.out.println("\nStill Available: " + (size = f.available()));
System.out.println("Skipping half of remaining bytes with skip()");
f.skip(size/2);
System.out.println("Still Available: " + f.available());
System.out.println("Reading " + n/2 + " into the end of array");
if (f.read(b, n/2, n/2) != n/2) {
System.err.println("couldn't read " + n/2 + " bytes.");
}
System.out.println(new String(b, 0, b.length));
System.out.println("\nStill Available: " + f.available());
f.close();
}}

```

OUTPUT:

```

Total Available Bytes: 1433
First 35 bytes of the file one read() at a time
// Demonstrate FileInputStream.
im
Still Available: 1398
Reading the next 35 with one read(b[])
port java.io.*;
class FileInputS
Still Available: 1363
Skipping half of remaining bytes with skip()
Still Available: 682
Reading 17 into the end of array
port java.io.*;
read(b) != n) {
S
Still Available: 665

```

Edusources.in

4.2.6 FileOutputStream

- FileOutputStream creates an OutputStream that you can use to write bytes to a file.

- Its most commonly used constructors are shown here:

FileOutputStream(String filePath)

FileOutputStream(File fileObj)

FileOutputStream(String filePath, boolean append)

FileOutputStream(File fileObj, boolean append)

- They can throw a FileNotFoundException.
- Here, filePath is the full path name of a file, and fileObj is a File object that describes the file.
- If append is true, the file is opened in append mode.
- Creation of a FileOutputStream is not dependent on the file already existing.
- FileOutputStream will create the file before opening it for output when you create the object.
- In the case where you attempt to open a read-only file, an IOException will be thrown.
- The following example creates a sample buffer of bytes by first making a String and then using the getBytes() method to extract the byte array equivalent.
- It then creates three files. The first, file1.txt, will contain every other byte from the sample. The second, file2.txt, will contain the entire set of bytes. The third and last, file3.txt, will contain only the last quarter.

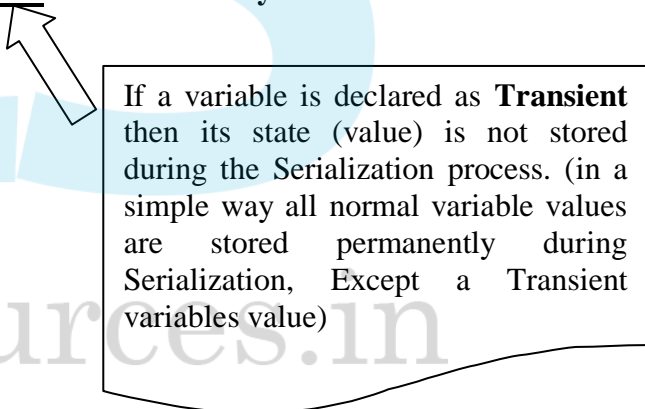
```
// Demonstrate FileOutputStream.
import java.io.*;
class FileOutputStreamDemo {
public static void main(String args[]) throws IOException {
String source = "Now is the time for all good men\n" + " to come to the aid of their country\n" +
" and pay their due taxes.";
byte buf[] = source.getBytes();
OutputStream f0 = new FileOutputStream("file1.txt");
for (int i=0; i < buf.length; i += 2) {
f0.write(buf[i]);
}
f0.close();
OutputStream f1 = new FileOutputStream("file2.txt");
f1.write(buf);
f1.close();
OutputStream f2 = new FileOutputStream("file3.txt");
f2.write(buf,buf.length-buf.length/4,buf.length/4);
f2.close();
}}
```

4.2.7 Serialization

- **Serialization is the process of writing the state of an object to a byte stream.**
- This is useful when you want to save the state of your program to a persistent storage area, such as a file.
- At a later time, you may restore these objects by using the process of deserialization.
- Serialization is also needed to implement Remote Method Invocation (RMI).
- RMI allows a Java object on one machine to invoke a method of a Java object on a different machine.
- An object may be supplied as an argument to that remote method.
- The sending machine serializes the object and transmits it. The receiving machine deserializes it.

Serializable

- object that implements the Serializable interface can be saved and restored by the serialization facilities.
- The Serializable interface defines no members. It is simply used to indicate that a class may be serialized.
- If a class is serializable, all of its subclasses are also serializable.
- **Variables that are declared as transient are not saved by the serialization facilities.** Also,
- static variables are not saved.



If a variable is declared as **Transient** then its state (value) is not stored during the Serialization process. (in a simple way all normal variable values are stored permanently during Serialization, Except a Transient variables value)

Program below shows Serialization Example

```
import java.io.*;
public class SerializationDemo {
    public static void main(String args[]) {
        // Object serialization
        try {
            MyClass object1 = new MyClass("Hello", -7, 2.7e10);
            System.out.println("object1: " + object1);
            FileOutputStream fos = new FileOutputStream("serial.txt");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(object1);
            oos.flush();
            oos.close();
        }
        catch(IOException e) {
            System.out.println("Exception during serialization: " + e);
            System.exit(0);
        }
        // Object deserialization
        try {
            MyClass object2;
            FileInputStream fis = new FileInputStream("serial.txt");
            ObjectInputStream ois = new ObjectInputStream(fis);
            object2 = (MyClass) ois.readObject();
            ois.close();
            System.out.println("object2: " + object2);
        }
        catch(Exception e) {
            System.out.println("Exception during deserialization: " + e);
            System.exit(0);
        }
    }
}
```

```
class MyClass implements Serializable {
    String s;
    int i;
    double d;
    public MyClass(String s, int i, double d) {
        this.s = s;
        this.i = i;
        this.d = d;
    }
    public String toString() {
        return "s=" + s + "; i=" + i + "; d=" + d;
    }
}
```

This program demonstrates that the instance variables of object1 and object2 are identical.

OUTPUT:

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

4.6 Using `isAlive()` and `join()`

Note: main method should finish last compare to all the threads.

This is accomplished by calling **`sleep()`** within **`main()`**, with a long enough delay to ensure that all child threads terminate prior to the main thread.

It also raises a larger question:

How can one thread know when another thread has ended?

Two ways exist to determine whether a thread has finished execution:

1) **`isAlive()`**

2) **`join()`**

`isAlive()` :

- Call **`isAlive()`** on the thread.
- This method is defined by Thread, and its general form is shown here:

`final boolean isAlive()`

- *The **`isAlive()`** method returns true if the thread called is still running*

join() :

- commonly use to wait for a thread to finish is called join(), shown here:

final void join() throws InterruptedException

- This method waits until the thread on which it is called terminates.
- **Ex: with join function main will wait until all the threads finish their execution.**

Program showing the use of isAlive() and join()

```
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }
}
```

```
class DemoJoin {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        System.out.println("Thread One is alive: "
            + ob1.t.isAlive());
        System.out.println("Thread Two is alive: "
            + ob2.t.isAlive());
        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
    }
}
```

OUTPUT:-of Program showing the use of isAlive() and join():

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
Thread One is alive: true
Thread Two is alive: true
Waiting for threads to finish.
One: 5
Two: 5
One: 4
Two: 4
One: 3
Two: 3
One: 2
Two: 2
One: 1
Two: 1
Two exiting.
One exiting.
```

NOTE: As you can see, after the calls to **join()** return, the threads have stopped executing.

4.7 THREAD PRIORITIES

Thread Priorities:

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.
- Thread priorities are integers that specify the relative priority of one thread to another.
- A higher-priority thread doesn't run any faster than a lower-priority thread.
- Thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch*.

Rules that determine when a context switch takes place are simple:

- *A thread can voluntarily relinquish control. This is done by explicitly yielding, sleeping, or blocking on pending I/O.*
- In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- *A thread can be preempted by a higher-priority thread.* In this case, a lower-priority thread that does not yield the processor is simply preempted no matter what it is doing by a higher-priority thread.
- Basically, as soon as a higher-priority thread wants to run, it does. **This is called *preemptive multitasking*.**
- In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated.
- For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion.
- For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.
- **CAUTION:** *Portability problems can arise from the differences in the way operating systems context-switch threads of equal priority.*

How to achieve Priority setting for thread:

To set a thread's priority, use the `setPriority()` method, which is a member of `Thread`.

This is its general form:

`final void setPriority(int level)`

Here, `level` specifies the new priority setting for the calling thread. The value of `level` must be within the range **`MIN_PRIORITY`** and **`MAX_PRIORITY`**

Thread Priority ranges from `MIN_PRIORITY` and `MAX_PRIORITY` level i.e. from 1 to 10, Default priority, specify `NORM_PRIORITY`, which is currently 5

Calling thread priority method:

Syntax: `ThreadName.setPriority(int Number);`

Ex: `A ThreadA = new A();`

`ThreadA.setPriority(Thread.Max_PRIORITY);`

You can obtain the current priority setting by calling the `getPriority()` method of `Thread`, shown here:

`final int getPriority()`

Example program showing setting thread Priority

```
class A extends Thread {
    public void run() {
        System.out.println("threadA started");
        for(int i=1;i<=4;i++) {
            System.out.println("from Thread A:i="+i);
        }
        System.out.println("Exit from A");
    }
}
class B extends Thread {
    public void run() {
        System.out.println("threadB started");
        for(int j=1;j<=4;j++) {
            System.out.println("from Thread B:j="+j);
        }
        System.out.println("Exit from B");
    }
}
```

```
class thread_priority {
    public static void main(String args[]) {
        A threadA = new A();
        B threadB = new B();
        threadB.setPriority(Thread.MAX_PRIORITY);
        threadA.setPriority(Thread.MIN_PRIORITY);
        System.out.println("Start thread A");
        threadA.start();
        System.out.println("Start thread B");
        threadB.start();
        System.out.println("End of main Thread");
    }
}
```

OUTPUT:

```
Start thread A
Start thread B
threadA started
End of main Thread
threadB started
from Thread A:i=1
from Thread B:j=1
from Thread A:i=2
from Thread B:j=2
from Thread B:j=3
from Thread B:j=4
Exit from B
from Thread A:i=3
from Thread A:i=4
Exit from A
```


4.9 INTER-THREAD COMMUNICATION

- Classic queuing problem, where one thread is producing some data and another is consuming it.
- To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.
- In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce.
- Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on.
- **Note: Polling means repeatedly in a loop one thread checks whether the other has done his work.**
- To avoid polling, Java includes inter-process communication via **wait(), notify(), and notifyAll() methods**
- **All methods are part of Object class**
- All three methods can be called only from within a **synchronized context**.

Rules for using these methods are actually quite simple:

- **wait()** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify()**.
- **notify()** wakes up a thread that called **wait()** on the same object.
- **notifyAll()** wakes up all the threads that called **wait()** on the same object. One of the threads will be granted access.

wait() method:

- **wait()** normally waits until **notify()** or **notifyAll()** is called,
- But there may be situations where it may be awakened due to some unknown reason. So to avoid this sun says always put wait in a loop which checks some condition on which the thread is waiting.

Example program showing inter-Thread communication (Producer Consumer)

```
class prodConDemo
{
    public static void main(String args[])
    {
        Messenger c=new Messenger();
        Producer p1=new Producer(c);
        Consumer c1=new Consumer(c);
        p1.start();
        c1.start();
    }
}
```

class Producer extends Thread

```
{
    private Messenger messenger;
    public Producer(Messenger c)
    {
        messenger=c;
    }
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            messenger.put(i);
            System.out.println("Put:"+i);
            try
            {
                sleep((int)(Math.random() *100));
            }catch(InterruptedException e) {}
        }
    }
}
```

class Consumer extends Thread {

```
private Messenger Messenger;
public Consumer(Messenger c)
{
    Messenger=c;
}
public void run(){
    int value=0;
    for(int i=0;i<10;i++){
        value=Messenger.get();
        System.out.println("Got:"+value);
    }
}
```

```
class Messenger{
private int contents;
private boolean available=false;
public synchronized int get(){
    while(available==false)
    {
        try {
            wait();
        } catch(InterruptedException e) {}
    }
    available=false;
    notifyAll();
    return contents;
}
```

```
public synchronized void put(int value){
    while(available==true)
    {
        try
        {
            wait();
        } catch(InterruptedException e) {}
    }
    contents=value;
    available=true;
    notifyAll();
}}
```

OUTPUT:

```
Put: 0
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
...
```

Deadlock

- Occurs when two threads have a circular dependency on a pair of synchronized objects.
- For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y.
- If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

4.10 Suspending, Resuming, and Stopping Threads

- Sometimes, suspending execution of a thread is useful.
 - For example, a separate thread can be used to display the time of day. If the user doesn't want a clock, then its thread can be suspended.
 - Once suspended, restarting the thread is also a simple matter.

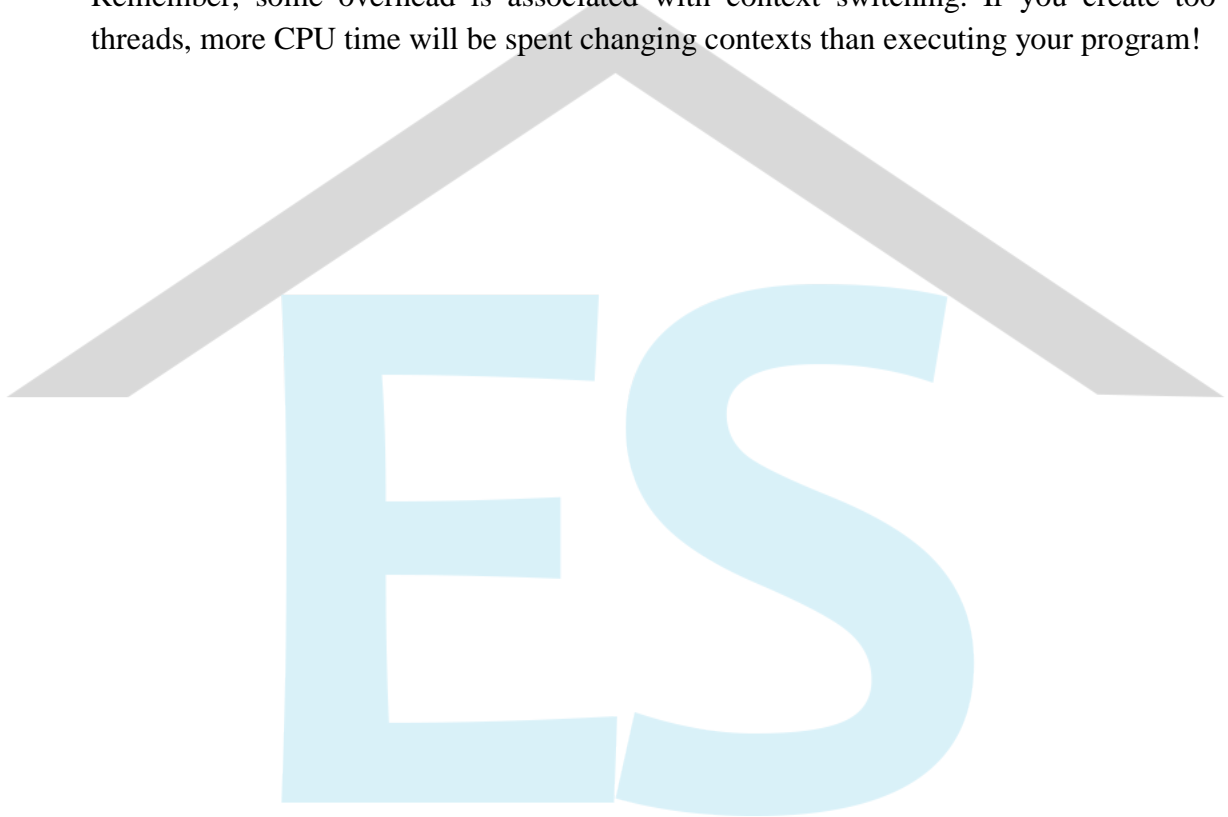
An Example program showing Suspending and Resuming of Threads:

```
// Using suspend() and resume().
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }
    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 15; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(200);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }
}
```

```
class SuspendResume {
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        try {
            Thread.sleep(1000);
            ob1.t.suspend();
            System.out.println("Suspending thread One");
            Thread.sleep(1000);
            ob1.t.resume();
            System.out.println("Resuming thread One");
            ob2.t.suspend();
            System.out.println("Suspending thread Two");
            Thread.sleep(1000);
            ob2.t.resume();
            System.out.println("Resuming thread Two");
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}
```

4.11 Using Multithreading

- The key to utilizing Java's multithreading features effectively is to think concurrently rather than serially.
- For example, when you have two subsystems within a program that can execute concurrently, make them individual threads.
- With the careful use of multithreading, you can create very efficient programs.
- **A word of caution:** *If you create too many threads, you can actually degrade the performance of your program rather than enhance it.*
- Remember, some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program!



Edusources.in