OBJECT ORIENTED PROGRAMMING WITH JAVA

BCA II SEM(NEP)

NOTES

Prepared By

Mr. Prabhu Kichadi, BE, MTECH

9880437187

UNIT – IV

Multithreading in java:

Thread life cycle and methods, Runnable interface, Thread priorities, Exception handling mechanism with try catch-finally, Introduction to JavaBeans.

I/O programming: Java Input Output: Java IO package, File, Byte/Character Stream, File reader / writer.

Threads: **Thread** is a light-weight process, it is a separate flow of execution runs independently within the program.

As java is a sequential programming as it takes execution from main() method, hence main() is a parent thread, from main() we can create any number of threads for multiple flows of executions.

Multithreading in java: Multithreading is a feature of Java that allows you to create multiple threads of execution within a single program. Each thread runs independently of the others and can perform a different task at the same time.

How to create threads in java:

1. By extending T**hread** Class and overriding run() method.
2. By implementing **Runnable** interface and overriding run() method.

1. By extending Thread Class and overriding run() method.

To create a thread by extending the Thread class, you need to create a subclass of the Thread class and override its run() method. The run() method contains the code that will be executed when the thread is started. To call run() method we have to call start() method from thread object.

Ex:

```
class MyThread extends Thread
{
  public void run()
  {
    // Code to be executed in this thread
    System.out.println("MyThread t1 task");
  }
}
public class MainClass
{
  public static void main(String args[])
  {
    MyThread t1 = new MyThread();
    t1.start(); // start the thread
    System.out.println("main() method task");
  }
}
```

Output:

```
MyThread t1 task
main() method task
```

2. By implementing Runnable interface and overriding run() method.:

To create a thread by implementing the Runnable interface, you need to create a class that implements the Runnable interface and override its run() method. The run() method contains the code that will be executed when the thread is started.

Example:

```
class MyRunnable implements Runnable
{
  public void run()
  {
    // Code to be executed in this thread
    System.out.println("Thread MyRunnable running");
  }
}

public class MainClass
{
  public static void main(String args[])
  {
    MyRunnable runnable = new MyRunnable();
    Thread t1 = new Thread(runnable);
    t1.start(); // start the thread
  }
}
```

Output:

Thread life cycle and methods:

In Java, a thread has a well-defined life cycle that starts when the thread is created and ends when the thread completes its execution.
Here are the different stages of the thread life cycle:

New: When a new thread is created, it is in the new state. The thread has not yet started to execute, and all its variables and resources are not yet allocated.

Runnable: Once the thread has been started using the start() method, it enters the runnable state. The thread is now ready to run, but it may not be currently executing because the operating system has not yet scheduled it.

Running: When the thread is scheduled by the operating system, it enters the running state. The code inside the run() method is now executing.

Blocked: A thread may enter the blocked state if it is waiting for a monitor lock, for example, if it has called the wait() method.

Waiting: A thread may enter the waiting state if it has called the wait() method or the join() method.

Timed waiting: A thread may enter the timed waiting state if it has called the sleep() method or the wait() method with a timeout value.

Terminated: When the run() method completes, the thread enters the terminated state.

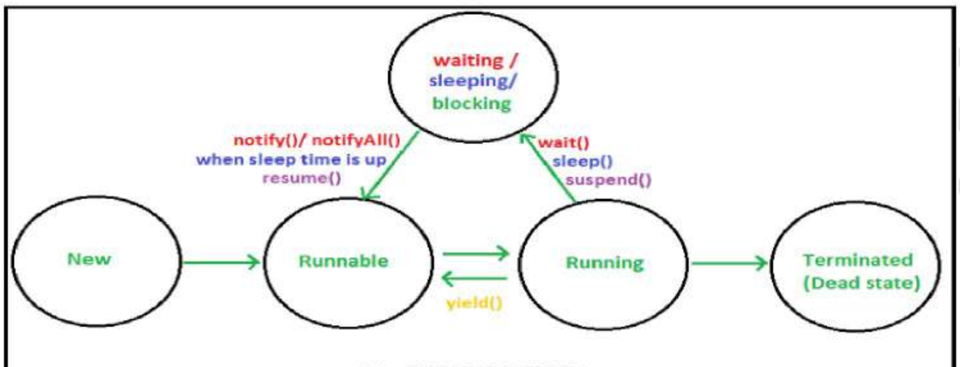

Fig. THREAD STATES

Here are some important methods that you can use to interact with the thread life cycle:

start(): This method starts the execution of the thread. When this method is called, the thread enters the runnable state.
run(): This method contains the code that will be executed by the thread. You must override this method when you create a new thread.
sleep(): This method causes the thread to sleep for a specified amount of time.
wait(): This method causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method on the same object.
join(): This method waits for the thread to complete its execution.
yield(): This method causes the thread to yield the CPU to other threads.
interrupt(): This method interrupts the thread.
isAlive(): This method returns true if the thread is currently running or if it has not yet started.

Thread priorities: Threads can have different priorities that determine the order in which they are scheduled to run by the operating system. Thread priorities are integers that range from 1 (lowest) to 10 (highest). By default, all threads have a priority of 5.

You can set the priority of a thread using the **setPriority()** method, which takes an integer value between 1 and 10 as its argument.

Here is an example:

```
Thread t1 = new Thread();
t1.setPriority(Thread.MIN_PRIORITY); // set the priority to 1 (minimum)
Thread t2 = new Thread();
t2.setPriority(Thread.MAX_PRIORITY); // set the priority to 10 (maximum)
```

In this example, we create two threads and set their priorities using the setPriority() method. The first thread has the lowest priority, while the second thread has the highest priority.

The thread scheduler uses the priority of a thread to decide which thread to schedule next. However, the exact behavior depends on the operating system and the JVM implementation.

Exception handling mechanism with try catch-finally:

Error: error is a mistake in program, not following the rules of the grammar of the language. These are called syntax errors, like accessing a variable which is not declared, missing semicolon, missing closing parenthesis.

Runtime Errors: Errors which occurs at program execution time are runtime errors.

Exceptions: Exceptions are unwanted, unexpected events/statements occurs at run time, that disturbs the normal flow of execution and terminates program abnormally.

Ex: ArithmeticException:

Without exception handling:

```
class TestException
{
        public static void main(String[] args)
        {
                int x,y,z;
                x = 10;
                y = 0;
                System.out.println(x/y);
        }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at TestException.main(TestException.java:9)
```

In this example The code above will result in a runtime exception: java.lang.ArithmeticException: / by zero

This is because it is trying to divide 10 by 0, which is not allowed in Java (or any other programming language). In Java, dividing any number by zero will result in an ArithmeticException being thrown. The code above will result in a runtime exception: java.lang.ArithmeticException: / by zero

This is because it is trying to divide 10 by 0, which is not allowed in Java (or any other programming language). In Java, dividing any number by zero will result in an ArithmeticException being thrown.

Exception handling: Exception handling is a mechanism when an exception is occurred what alternative code can be executed and so that program terminates normally.

In java exception is handled using try, catch and finally block.

try: The try block contains the code that may throw an exception. If an exception is thrown, control is transferred to the catch block. Every try block must follow at-least one catch or one finally block.

catch: The catch block catches the exception and handles it appropriately. It contains the code that handles the exception, for a single try block, we can surround with multiple catch blocks, the order should be more specific to more generic exceptions.

finally: The finally block is executed regardless of whether an exception is thrown or not. It is used to release resources or perform cleanup operations.

Ex1: ArrayIndexOutOfBoundsException

```
public class DemoException
{
  public static void main(String[] args)
   {
     try
     {
        int[] arr = {1, 2, 3};
        System.out.println(arr[3]); // throws ArrayIndexOutOfBoundsException
```

```
      }
     catch (ArrayIndexOutOfBoundsException e)
      {
         System.out.println("Index out of bounds");
       }
      finally
     {
         System.out.println("Done");
       }
    }
}
```

Output:

```
Index out of bounds
Done
```

Ex2: ArithemticException

```
class TestException
{
        public static void main(String[] args)
        {
                int x,y,z;

                try
                {
                        x = 10;
                        y = 0;
                        System.out.println(x/y);
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Exception caught - / by zero and handled");
                        y = 2;
                }
                finally
                {
                        System.out.println("Always executes");
                }
        }
}
```

Output:

```
Exception caught - / by zero and handled
Always executes
```

Exception with multiple catch blocks:

```
class TestException
{
        public static void main(String[] args)
        {
                int x,y,z;
                int a[] = new int[5];

                try
                {
                        x = 10;
                        y = 2;
                        System.out.println(x/y);
                        System.out.println(a[5]);

                }
                catch(ArithmeticException e)
                {
                        System.out.println("Exception caught - / by zero and handled");
                        y = 2;
                }
                catch (ArrayIndexOutOfBoundsException e)
                {
                       System.out.println("Index out of bounds");
                }
                finally
                {
                        System.out.println("Always executes");
                }
        }
}
```

Output:

```
5
Index out of bounds
Always executes
```

Some of the exceptions:

1. NullPointerException - This exception is thrown when attempting to access or modify a null object reference. For example:
String str = null;
System.out.println(str.length()); // throws NullPointerException

2.ArrayIndexOutOfBoundsException - This exception is thrown when attempting to access an array element with an index that is out of bounds. For example:

```
int[] arr = {1, 2, 3};
System.out.println(arr[3]); // throws ArrayIndexOutOfBoundsException
```

3. ClassCastException - This exception is thrown when attempting to cast an object to a type that is not compatible with the actual type of the object. For example:

```
Object obj = "hello";
Integer num = (Integer) obj; // throws ClassCastException
```

4. ArithmeticException - This exception is thrown when an arithmetic operation results in an exceptional condition. For example:

```
int x = 10, y = 0;
int z = x / y; // throws ArithmeticException
```

Types of Exceptions:

Checked Exceptions: Checked exceptions are the exceptions that are checked at compile time. It means if a method throws a checked exception, the programmer needs to handle it or declare it in the method signature using the throws keyword. Examples of checked exceptions include FileNotFoundException, IOException, ClassNotFoundException, etc.

```
public void readFromFile(String fileName) throws FileNotFoundException
{
   File file = new File(fileName);
   FileInputStream fis = new FileInputStream(file);
   // Code to read from file
   fis.close();
}
```

Unchecked Exceptions: Unchecked exceptions are the exceptions that are not checked at compile time. It means the programmer is not required to handle or declare them. Examples of unchecked exceptions include ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, IllegalArgumentException, etc.

```
int x = 10, y = 0;
int z = x / y; // throws ArithmeticException
```

Introduction to JavaBeans.

I/O programming: Input/Output (I/O) programming in Java involves reading data from an input source(files on disk), processing it, and writing the output to an output destination(files on the disk).

Java provides a rich set of classes and interfaces for performing I/O operations.

Java IO package:

Java I/O: **java.io package** provides a set of classes & interfaces for IO programming, some of the classes are,

InputStream and OutputStream: These are the basic abstract classes for reading and writing byte-oriented data. All other I/O classes are built on these classes.

Reader and Writer: These are the abstract classes for reading and writing character-oriented data. They are used to handle text-based input/output.

BufferedReader and BufferedWriter: These classes are used for buffered input/output. They are more efficient than the basic I/O classes because they minimize the number of system calls.

Scanner: This class is used for parsing primitive types and strings from a stream. It can also be used to read input from the console.

File: This class represents a file or directory path. It provides methods for creating, deleting, and renaming files, as well as determining file properties such as size, last modified date, and permissions.

FileInputStream and FileOutputStream: These classes are used for reading and writing binary data from and to files.

FileReader and FileWriter: These classes are used for reading and writing character data from and to files.

To perform input/output operations in Java, you generally follow these steps:

1. Open the input/output stream, file or connection.
2. Read or write data.
3. Close the input/output stream, file or connection.

File: File is a collection of related data which is stored on a secondary storage(disk). File formats can be .txt, .docx, .pdf, ,xls, .csv etc.

File Types:

1. **Text files**
2. **Binary Files:**

In Java, a File object represents a file or directory path. It provides methods for creating, deleting, and renaming files, as well as determining file properties such as size, last modified date, and permissions.

To create a File object, you need to specify the file path. The file path can be absolute or relative. Here are some examples:

File Class: The File class represents a file or directory path in a platform-independent way. It provides methods to access and manipulate files and directories on the file system.

File(String path) - creates a new File instance by converting the given pathname string into an abstract pathname.

Ex:

```
File file = new File("example.txt");
```

File(String parent, String child) - creates a new File instance by converting the given parent pathname string and child pathname string into an abstract pathname.

Here are some common methods provided by the File class:

1. exists() - returns a boolean indicating whether the file or directory exists.
2. getName() - returns the name of the file or directory.
3.getPath() - returns the path of the file or directory.
4. isFile() - returns a boolean indicating whether the path represents a file.
5. isDirectory() - returns a boolean indicating whether the path represents a directory.
6. mkdir() - creates a directory.
7. createNewFile() - creates a new empty file.
8. delete() - deletes the file or directory.

Ex:

```
import java.io.*;
class DemoFile
{
        public static void main(String[] args) throws IOException
        {
                File f1 = new File("sample.txt");
                System.out.println("File Exists : "+f1.exists());
                System.out.println("Created New File : "+f1.createNewFile());
                System.out.println("Get File/Path Name : "+f1.getName());
                f1 = new File("IO");
                System.out.println("Directory Created : "+f1.mkdir());
                System.out.println("File? : "+f1.isFile());
                System.out.println("Directory? : "+f1.isDirectory());
                System.out.println("Deleted? : "+f1.delete());
        }
}
```

Output:

```
File Exists : true
Created New File : false
Get File/Path Name : sample.txt
Directory Created : false
File? : false
Directory? : true
Deleted? : true
```

Byte/Character Stream & File reader / writer:

A byte stream; is used to handle input and output of bytes (8-bit binary data), Byte streams are mainly used for handling binary data such as images, audio, and video files.
Java provides InputStream and OutputStream for byte streams.
Some commonly used byte stream classes in Java include
FileInputStream,
FileOutputStream,
BufferedInputStream, and
BufferedOutputStream.

Character stream is used to handle input and output of characters (16-bit Unicode data). Character streams, on the other hand, are used to handle textual data, such as files containing text, web pages, and documents.
Reader and Writer for character streams.

Commonly used character stream classes include
FileReader,
FileWriter,
BufferedReader, and
BufferedWriter.

Byte stream:

Some commonly used byte stream classes in Java include
FileInputStream,
FileOutputStream,
BufferedInputStream, and
BufferedOutputStream.

1. FileInputStream: This class is used for reading data from a file as a sequence of bytes.
Constructors:

FileInputStream(File file) : Creates a FileInputStream object by opening a connection to the file specified by the File object parameter.
FileInputStream(String filename) : Creates a FileInputStream object by opening a connection to the file specified by the string parameter.

Both constructors can throw a FileNotFoundException if the file does not exist or cannot be opened.

FileInputStream has several methods that can be used to read data from the file:

read(): Reads a single byte of data from the input stream and returns it as an integer in the range 0 to 255. Returns -1 if the end of the stream has been reached.
read(byte[] b): Reads up to b.length bytes of data from the input stream into an array of bytes. Returns the total number of bytes read, or -1 if the end of the stream has been reached.
read(byte[] b, int off, int len): Reads up to len bytes of data from the input stream into an array of bytes starting at offset off. Returns the total number of bytes read, or -1 if the end of the stream has been reached.
skip(long n): Skips over and discards n bytes of data from the input stream. Returns the actual number of bytes skipped.
available(): Returns an estimate of the number of bytes that can be read from the input stream without blocking.
close(): Closes the input stream and releases any resources associated with it.

All of these methods can throw an IOException if an error occurs while reading from the input stream.

Example:

```
import java.io.*;
class DemoFile
{
	public static void main(String[] args) throws IOException
	{
		FileInputStream fis = new FileInputStream("myfile.txt");
		int ch;
		ch = fis.read();
		while (ch != -1)
		{
			System.out.print((char)ch);
			ch = fis.read();
		}
		fis.close();
	}
}
```

2. FileOutputStream: This class is used for writing data to a file as a sequence of bytes.

FileOutputStream is a class in Java's java.io package that is used for writing data to a file in byte format. It is used to create a file and write data to it.

Constructors
FileOutputStream(File file): Creates a new FileOutputStream instance that writes to the specified File object.

FileOutputStream(String fileName): Creates a new FileOutputStream instance that writes to the file with the specified name.

FileOutputStream(File file, boolean append): Creates a new FileOutputStream instance that writes to the specified File object. If the append parameter is true, the data will be appended to the end of the file. Otherwise, the file will be truncated before writing.

FileOutputStream(String fileName, boolean append): Creates a new FileOutputStream instance that writes to the file with the specified name. If the append parameter is true, the data will be appended to the end of the file. Otherwise, the file will be truncated before writing.

Methods

write(byte[] b): Writes the bytes in the specified byte array to the file.

write(byte[] b, int off, int len): Writes len bytes from the specified byte array starting at offset off to the file.

write(int b): Writes the specified byte to the file.

flush(): Flushes the output stream, ensuring that any buffered data is written to the file.

close(): Closes the output stream, releasing any system resources associated with it. Once the stream is closed, further calls to its methods will throw an IOException.

getChannel(): Returns the unique FileChannel object associated with this file output stream.

Ex:

```
import java.io.*;
class DemoFile
{
        public static void main(String[] args) throws IOException
        {
                try
                {
                FileOutputStream fos = new FileOutputStream("myfile.txt");
                  fos.write("Hello, world!".getBytes());
                 fos.close();
                System.out.println("Data written to the file successfully.");
              }
                catch (IOException e)
                {
                System.out.println("An error occurred while writing data to the
file.");
```

```
                e.printStackTrace();
        }
          }
}
```

3. BufferedInputStream: This class is used for buffering the input data to improve the performance of byte stream reading.

BufferedInputStream is a class in Java that provides buffered input functionality. It reads data from an InputStream and stores it in an internal buffer to increase efficiency and reduce the number of reads from the underlying input stream. BufferedInputStream extends the FilterInputStream class, which means that it inherits all of its methods and constructors.

Constructors:

BufferedInputStream(InputStream in) - This constructor creates a new BufferedInputStream object that reads from the specified InputStream.

BufferedInputStream(InputStream in, int size) - This constructor creates a new BufferedInputStream object that reads from the specified InputStream with the specified buffer size.

Methods:

int read() - Reads the next byte of data from the input stream.

int read(byte[] b, int off, int len) - Reads up to len bytes of data from the input stream into an array of bytes.

long skip(long n) - Skips over and discards n bytes of data from the input stream.

int available() - Returns the number of bytes that can be read from the input stream without blocking.

void mark(int readlimit) - Sets a mark position in the input stream.

void reset() - Resets the input stream to the last marked position.

boolean markSupported() - Returns true if the input stream supports the mark() and reset() methods.

Example:

```
import java.io.*;
class DemoFile
{
        public static void main(String[] args) throws IOException
        {
                try
                {
                         FileInputStream fis = new FileInputStream ("sample.txt");
                        BufferedInputStream bis = new BufferedInputStream(fis);
                        int c;
                        while ((c = bis.read()) != -1)
                       {
                                System.out.print((char) c);
                         }
                         bis.close();
                 }
                catch (IOException e)
                {
                        System.out.println(e);
                 }
        }
}
```

Output:

```
Good morning
```

4. BufferedOutputStream: This class is used for buffering the output data to improve the performance of byte stream writing.

BufferedOutputStream is a class in Java that provides buffered output functionality. It writes data to an OutputStream and stores it in an internal buffer to increase efficiency and reduce the number of writes to the underlying output stream. BufferedOutputStream extends the FilterOutputStream class, which means that it inherits all of its methods and constructors.

Constructors:

BufferedOutputStream(OutputStream out) - This constructor creates a new BufferedOutputStream object that writes to the specified OutputStream.

BufferedOutputStream(OutputStream out, int size) - This constructor creates a new BufferedOutputStream object that writes to the specified OutputStream with the specified buffer size.

Methods:

void write(int b) - Writes a byte of data to the output stream.

void write(byte[] b, int off, int len) - Writes len bytes of data from an array of bytes to the output stream.

void flush() - Flushes the output stream, forcing any buffered output bytes to be written out.

void close() - Closes the output stream and releases any system resources associated with it.

Example:

```
import java.io.*;
class DemoFile
{
        public static void main(String[] args) throws IOException
        {
        try
        {
         FileOutputStream fos = new FileOutputStream("example.txt");
         BufferedOutputStream bos = new BufferedOutputStream(fos);
         String data = "This is an example of using bos in Java.";
         bos.write(data.getBytes());
         bos.flush();
        bos.close();
         }
        catch (IOException e)
        {
        System.out.println(e);
         }
      }
}
```

File Reader/Writer:

FileReader:are used for reading from and writing to text files, respectively. They are both subclasses of the abstract class Reader and Writer, respectively.

FileReader:

FileReader is used to read character data from a file. It reads the file character by character and returns them as Unicode values.

Constructors:

FileReader(String fileName) - This constructor creates a new FileReader object with the specified file name.
FileReader(File file) - This constructor creates a new FileReader object with the specified File object.

Methods:

int read() - Reads a single character from the file.
int read(char[] cbuf) - Reads characters from the file into an array.
void close() - Closes the FileReader object and releases any system resources associated with it.

Example:

```
import java.io.*;
class DemoFile
{
        public static void main(String[] args) throws IOException
        {
        try
        {
         FileReader fr = new FileReader("example.txt");
         int c;
         while ((c = fr.read()) != -1)
         {
            System.out.print((char) c);
         }
         fr.close();
         }
        catch (IOException e)
        {
         System.out.println(e);
       }
       }
}
```

FileWriter:

FileWriter is used to write character data to a file. It writes the data character by character to the file.

Constructors:

FileWriter(String fileName) - This constructor creates a new FileWriter object with the specified file name.

FileWriter(File file) - This constructor creates a new FileWriter object with the specified File object.

Methods:

void write(int c) - Writes a single character to the file.

void write(char[] cbuf) - Writes an array of characters to the file.

void flush() - Flushes the FileWriter object, forcing any buffered output characters to be written out.

void close() - Closes the FileWriter object and releases any system resources associated with it.

Example:

```
import java.io.*;
class DemoFile
{
        public static void main(String[] args) throws IOException
        {
        try
        {
         FileWriter fileWriter = new FileWriter("example.txt");
         String data = "This is an example of using FileWriter in Java.";
         fileWriter.write(data);
         fileWriter.flush();
         fileWriter.close();
      }
       catch (IOException e)
      {
         System.out.println(e);
      }
     }
}
```

BufferedReader & BufferedWriter:

BufferedReader is a class that reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. It inherits from the Reader class and can be used to read data from various input sources such as files, network sockets, and standard input/output streams.

Constructors:

BufferedReader(Reader in) - creates a new BufferedReader object to read data from the specified input stream.

Methods:

int read() - reads a single character from the input stream and returns it as an integer.
int read(char[] cbuf) - reads characters into an array from the input stream and returns the number of characters read.
String readLine() - reads a line of text from the input stream and returns it as a string.
void close() - closes the input stream.

Ex:

```
try
{
   BufferedReader br = new BufferedReader(new FileReader("example.txt"))
   String line;
   while ((line = br.readLine()) != null)
  {
    System.out.println(line);
   }
}
```

BufferedWriter is a class that writes text to a character-output stream, buffering characters so as to provide for the efficient writing of characters, arrays, and lines. It inherits from the Writer class and can be used to write data to various output destinations such as files, network sockets, and standard input/output streams.

Constructors:

BufferedWriter(Writer out) - creates a new BufferedWriter object to write data to the specified output stream.

Methods:

void write(int c) - writes a single character to the output stream.
void write(String str) - writes a string of characters to the output stream.
void newLine() - writes a platform-specific line separator to the output stream.
void flush() - flushes the output stream.
void close() - closes the output stream.

Ex:

```
Try
{
   BufferedReader br = new BufferedReader(new FileReader("example.txt"))
   BufferedWriter bw = new BufferedWriter(new FileWriter("output.txt")))
   String line;
   while ((line = br.readLine()) != null)
  {
     bw.write(line);
     bw.newLine();
  }
}
catch (IOException e)
 {
   e.printStackTrace();
 }
```

Note: Refer all Java Lab journal programs on this unit.