Multithreading

Multithreading in java is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.

2) You **can perform many operations together so it saves time**.

3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

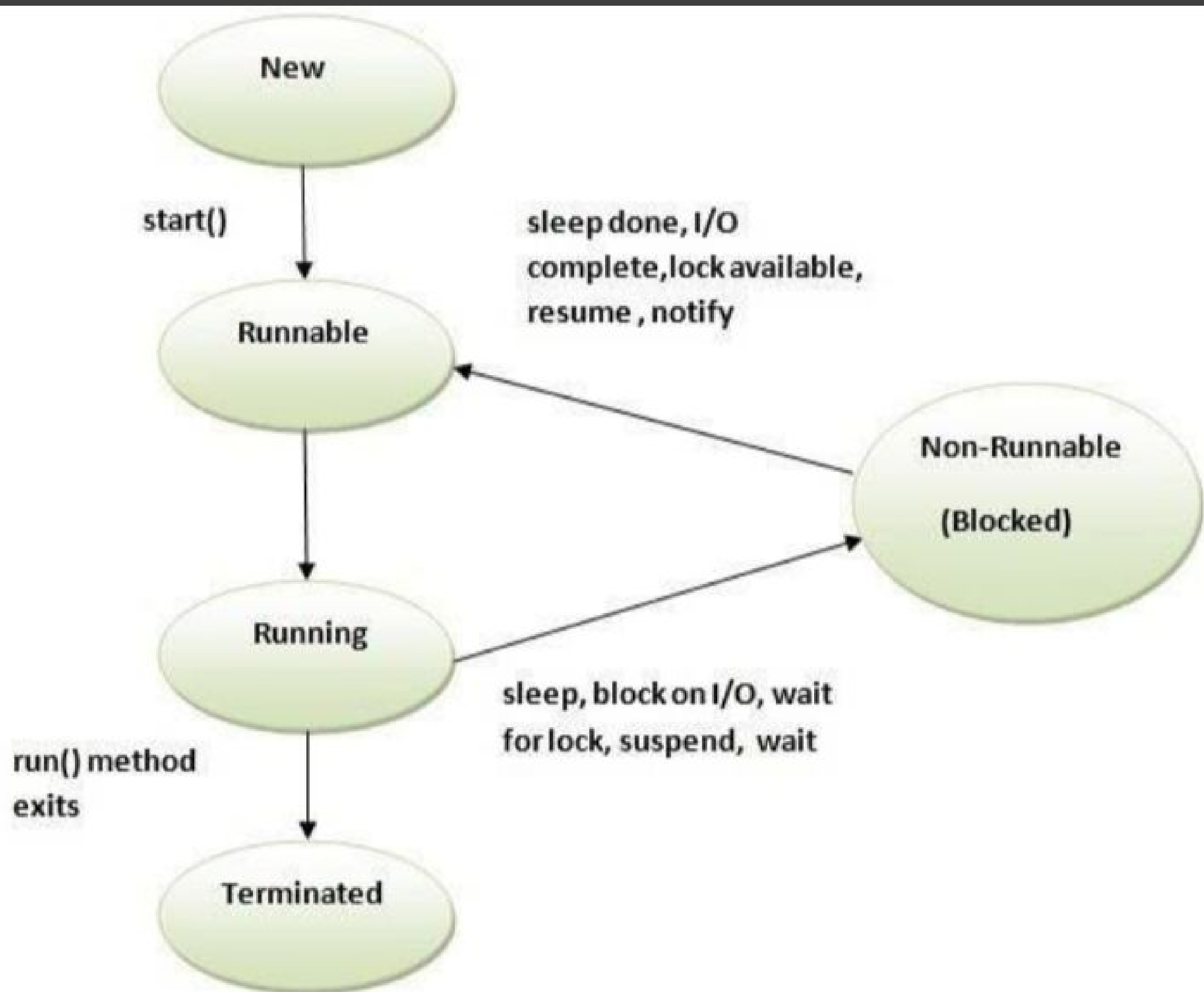
Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

Java Thread Example by extending Thread class

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 } }
```

Output:thread is running...

Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 } }
```

Output:thread is running...

Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
public void run(){
  System.out.println("running thread name is:"+Thread.currentThread().getName());
  System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
  }
 public static void main(String args[]){
```

```
TestMultiPriority1 m1=new TestMultiPriority1();
TestMultiPriority1 m2=new TestMultiPriority1();
m1.setPriority(Thread.MIN_PRIORITY);
m2.setPriority(Thread.MAX_PRIORITY);
m1.start();
m2.start();
} }
```

Output:running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1
running thread priority is:1

Java synchronized method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Example of inter thread communication in java

Let's see the simple example of inter thread communication.

```
class Customer{
int amount=10000;
synchronized void withdraw(int amount){
System.out.println("going to withdraw...");
if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}
synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}
class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
```

```
public void run(){c.deposit(10000);}
}
start();
}}
```

```
Output: going to withdraw...
       Less balance; waiting for deposit...
       going to deposit...
       deposit completed...
       withdraw completed
```

ThreadGroup in Java

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

Note: Now suspend(), resume() and stop() methods are deprecated.

Java thread group is implemented by *java.lang.ThreadGroup* class.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

ThreadGroup(String name)
ThreadGroup(ThreadGroup parent, String name)

Let's see a code to group multiple threads.

```
1. ThreadGroup tg1 = new ThreadGroup("Group A");
2. Thread t1 = new Thread(tg1,new MyRunnable(),"one");
3. Thread t2 = new Thread(tg1,new MyRunnable(),"two");
4. Thread t3 = new Thread(tg1,new MyRunnable(),"three");
```

Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

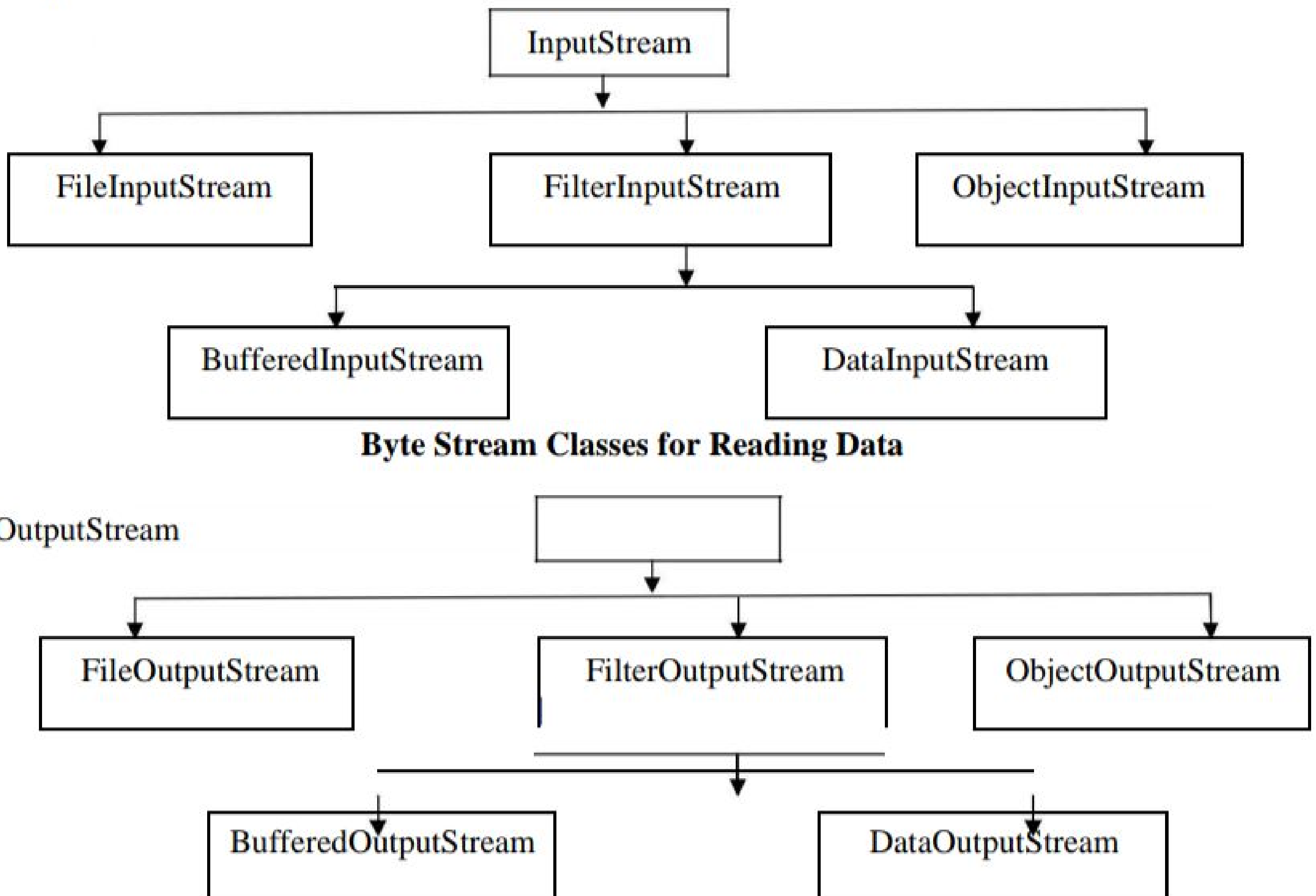
Now we can interrupt all threads by a single line of code only.

```
1. Thread.currentThread().getThreadGroup().interrupt();
```

Streams and Files

A Stream represents flow of data from one place to another place Input Streams reads or accepts data Output Streams sends or writes data to some other place All streams are represented as classes in javaio package The main advantage of using stream concept is to achieve hardware independence This is because we need not change the stream in our program even though we change the hardware Streams are of two types in Java:

Byte Streams: Handle data in the form of bits and bytes Byte streams are used to handle any characters (text), images, audio and video files For example, to store an image file (gif or
jpg), we should go for a byte stream To handle data in the form of 'bytes' the abstract classes: InputStream and OutputStream are used The important classes of byte streams are:



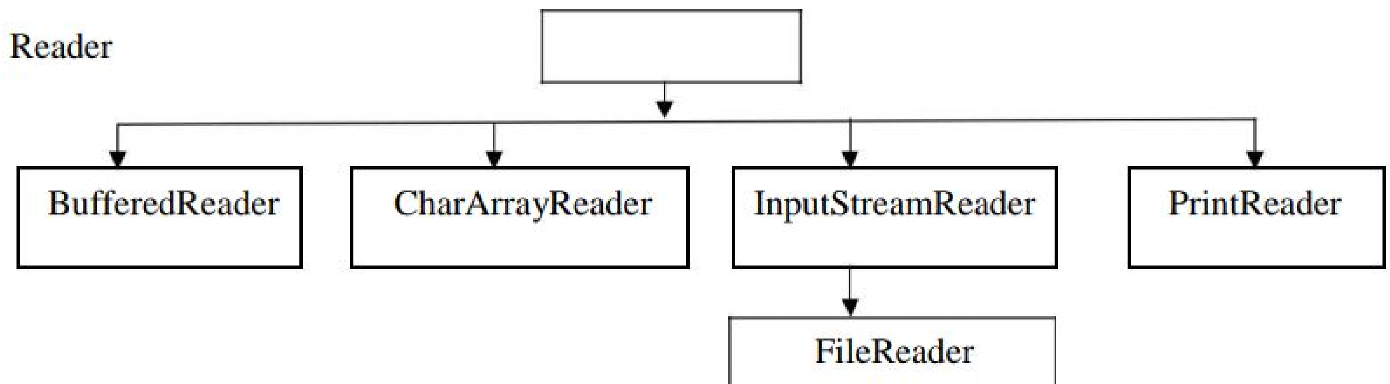
Byte Stream Classes for Reading Data

Byte Stream Classes for Writing Data

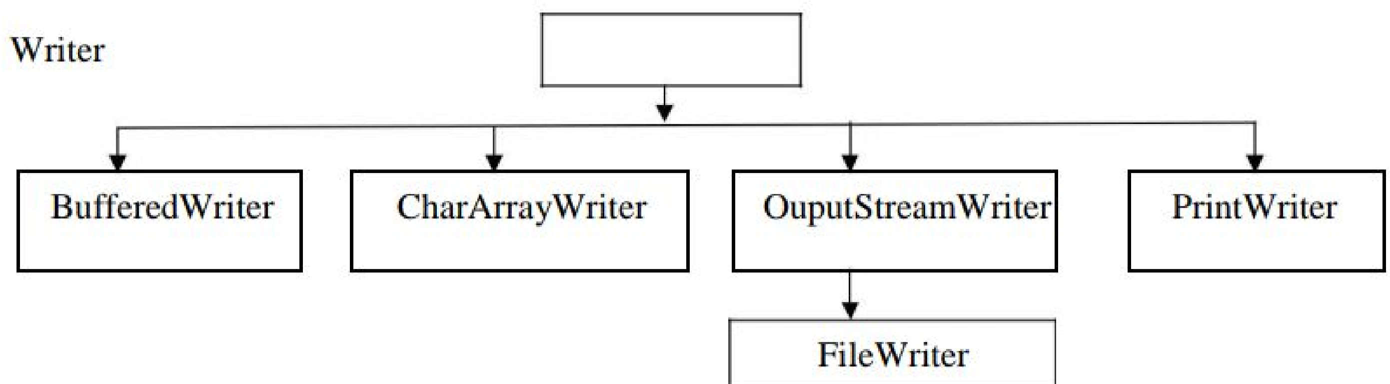
FileInputStream/FileOutputStream: They handle data to be read or written to disk files
FilterInputStream/FilterOutputStream: They read data from one stream and write it to another stream
ObjectInputStream/ObjectOutputStream: They handle storage of objects and primitive data

Character or Text Streams: Handle data in the form of characters Character or text streams can always store and retrieve data in the form of characters (or text) only It means text streams are more suitable for handling text files like the ones we create in Notepad They are not suitable to handle the images, audio or video files To handle data in the form of 'text'

the abstract classes: Reader and Writer are used The important classes of character streams are:



Text Stream Classes for Reading Data



Text Stream Classes for Writing Data

BufferedReader/BufferedWriter: - Handles characters (text) by buffering them They provide efficiency
CharArrayReader/CharArrayWriter: - Handles array of characters
InputStreamReader/OutputStreamWriter: - They are bridge between byte streams and character streams Reader reads bytes and then decodes them into 16-bit unicode characters Writer decodes characters into bytes and then writes
PrintReader/PrintWriter: - Handle printing of characters on the screen

File: A file represents organized collection of data Data is stored permanently in the file Once data is stored in the form of a file we can use it in different programs

Program 1: Write a program to read data from the keyboard and write it to a text file using byte stream classes
//Creating a text file using byte stream classes

```
import javaio*;
class Create1
{public static void main(String args[]) throws IOException
	{//attach keyboard to DataInputStream DataInputStream dis =
		new DataInputStream (Systemin); //attach the file to
		FileOutputStream FileOutputStream fout = new
		FileOutputStream ("myfile");
		//read data from DataInputStream and write into
		FileOutputStream char ch;
		Systemoutprintln ("Enter @ at end : " ) ;
		while( (ch = (char) disread() ) != '@' )
			foutwrite (ch);
		foutclose ();
	}
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Create1.java

D:\JQR>java  Create1
Enter @ at end :
I am writing first line
I am writing second line
@

D:\JQR>
```

Program 2: Write a program to improve the efficiency of writing data into a file using BufferedOutputStream

```
//Creating a text file using byte stream classes
import javaio*;
class Create2
{public static void main(String args[]) throws IOException
	{//attach keyboard to DataInputStream DataInputStream dis =
		new DataInputStream (Systemin);
		//attach file to FileOutputStream, if we use true then it will open in append
		mode FileOutputStream fout = new FileOutputStream ("myfile", true);
		BufferedOutputStream bout = new BufferedOutputStream (fout, 1024); //Buffer
		size is declared as 1024 otherwise default buffer size of 512 bytes is used //read
		data from DataInputStream and write into FileOutputStream
		char ch;
		Systemoutprintln ("Enter @ at end : " ) ;
		while ( (ch = (char) disread() ) != '@' )
			boutwrite (ch);
		boutclose ();
		foutclose ();
	}
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Create2.java

D:\JQR>java  Create2
Enter @ at end :
This is new line in my file
@

D:\JQR>type myfile
I am writing first line
I am writing second line
This is new line in my file

D:\JQR>
```

Program 3: Write a program to read data from *myfile* using FileInputStream

```
//Reading a text file using byte stream classes
import javaio*;
class Read1
{public static void main (String args[]) throws IOException
        {//attach the file to FileInputStream FileInputStream fin =
                new FileInputStream ("myfile");
                //read data from FileInputStream and display it on the
                monitor int ch;
                while ( (ch = finread() ) != -1 )
                        Systemoutprint ((char) ch);
                finclose ();
        }
}
```

Output:

Program 4: Write a program to improve the efficiency while reading data from a file using BufferedInputStream

```
//Reading a text file using byte stream classes
import javaio*;
class Read2
{public static void main(String args[]) throws IOException
        {//attach the file to FileInputStream FileInputStream fin = new
                FileInputStream ("myfile"); BufferedInputStream bin = new
                BufferedInputStream (fin); //read data from FileInputStream
                and display it on the monitor int ch;

                while ( (ch = binread() ) != -1 )
                        Systemoutprint ( (char) ch);
                finclose ();
        }
}
```

Output:

Program 5: Write a program to create a text file using character or text stream classes //Creating a text file using character (text) stream classes import javaio*;

```
class Create3
{public static void main(String args[]) throws IOException
    {       String str = "This is an Institute" + "\n You are a student";     // take a String
            //Connect a file to FileWriter
            FileWriter fw = new FileWriter ("textfile");
            //read chars from str and send to fw
            for (int i = 0; i<strlength () ; i++)
                    fwwrite (strcharAt (i) );
            fwclose ();
    }
}
```

Output:

Program 6: Write a program to read a text file using character or text stream classes //Reading data from file using character (text) stream classes import javaio*;

```
class Read3
{public static void main(String args[]) throws IOException
    {//attach file to FileReader
            FileReader fr = new FileReader
            ("textfile"); //read data from fr and display
            int ch;
            while ((ch = frread()) != -1)
                    Systemoutprint ((char) ch);
            //close the
            file frclose ();
    }
}
```

Output:

Note: Use BufferedReader and BufferedWriter to improve the efficiency of the above two programs

A String is a sequence of characters (e.g. "Hello World").

A String is an object in java, and not a primitive.

Creating Strings

We can create a String object in two ways:

1. Assigning a String literal to a String variable
   - e.g. **String greeting = "Hello world!";**
   - Here, "Hello world!" is a string literal.
   - Java keep only one copy of a string literal object and reuses them. This process is called String interning.
   - In this approach, string objects are not created again and again, but reused.
2. Creating a String object using the new keyword and one of the String constructors like any other object
   - e.g. **String greeting = new String("Hello world!");**
   - In this approach, minimum one new String object is created every time.

♣ Whenever the new keyword is used, an object is created allocating memory from the heap.

♣ The String literal is also placed in the String pool if this string literal is used for first time. Thus if the String object is not already present in the pool two objects will be created in total.

Important Note!

1. It is preferred to use String literal (approach 1) when possible as it will reuse objects and won't create a new object every time.
2. In some cases it might be needed to use approach two (where you use the new keyword and a String constructor).
   - The String class has thirteen constructors that allow you to provide the initial value using different sources, such as an array of characters.
   - There are two constructors that accept the StringBuffer and StringBuilder classes.

Immutability of String objects

1. A String object is immutable; this means that once an Object is created it, cannot be changed.
2. If you call a function on the string object, it will return you the string object; but it will not change the string object.

1. However you may assign this returned value to the reference variable overriting the previous value.

Example: Calling methods on String object

```
String helloString = "Hello";

helloString.concat(" world");

System.out.println(helloString);
```

- This will print only Hello.
- The statement "*helloString.concat(" world");*" will create a new String object with content as "Hello World" and return it, but will not change the original String.
  To print "Hello world", you have to assign the concat result to the helloString reference variable as:

  helloString = helloString.concat(" world");.

StringBuffer and StringBuilder

- Any modification on a String object will create a new string object that contains the result of the operation.
  - Therefore, if you try to append 100 string literals to a String object, 100 String objects will be created, which is not a good thing.
  - Java hence provide two alternatives to String called StringBuffer and StringBuilder which will modify the current object and will not create a new object like String every time the object value is modified.

Important methods of String class

1. The **length()** method returns the number of characters contained in the string object.
2. The String class includes a **concat method** for concatenating two strings.
   - E.g. string1.concat(string2);
     - ♣ This returns a new string that is string1 with string2 added to it at the end, but not that since a String object is immutable, it will not change the values of string1 or string2.

3. The **static format() method** allows you to create a formatted string that you can reuse, as opposed to a one-time print statement.
   - Example:
     - String fs;
     - fs = String.format("The value of the float variable is %f", floatVar);
     - System.out.println(fs);
   - You could also write the same as:
     - System.out.printf("The value of the float variable is %f", floatVar);
4. The **static valueOf method** will convert a number to a string:
   - E.g. String s1 = String.valueOf(777);
5. The **charAt(int index) method** returns the char value at the specified index. The index ranges from 0 to length() - 1.
   - **StringIndexOutOfBoundsException** is thrown by String methods to indicate that an index is either negative or greater than the size of the string.
     - For some methods such as the charAt method, this exception also is thrown when the index is equal to the size of the string.
   - Example:
     - String str = "Hello World";
     - System.out.println(str.charAt(1));
     - str.charAt(1) will print the second element, which is e.
     - System.out.println(str.charAt(11));
     - str.charAt(11) will throw a StringIndexOutOfBoundsException as the size of the string is 11 and str.charAt(11) refers to the 12th element.
     - Note that the index ranges from 0 to length() - 1.
6. String class overrides the **equals method of the Object class**.
   - The equals method see if two String objects has the same value whereas == checks if two String objects are actually referring to same memory location.
   - Example
     - String h1 = "Java";
     - String h2 = new String ("Java");

- System.out.println(h1==h2);
- This will print false and second print statement will print true. Refer to String interning in Java for more details.
- System.out.println(h1.equals(h2);
  - The **equals** method is **case sensitive.** String contains a method **equalsIgnoreCase()** which is similar to equals, but compares value **case insensitively**.

Chaining

Most methods of String class (e.g. concat) return a new string object; we can call any of the String methods on that new object returned through chaining.

Example: Chaining of string methods that return a string object

String s = new String("Hello").concat("World").concat("Program");

System.out.println(s);

This will print:

Hello World Program

- Even with Chaining we have to assign the final object to a reference or the changes will not be saved (as String objects are immutable).

Collator class

- The Collator class can be used to manipulate strings in a locale-specific manner removing the problems of different internal string representations.
- For instance, many languages use the accent to differentiate or emphasize a character. If these different representations are compared using the equals method, the method would return false.

Examples

Example: Empty String

Find the output:

```
String s = "HelloWorld";

if(s.startsWith(""))

  System.out.println("Strings in java start with an empty string");
```

else

System.out.println("Strings in java DOES NOT start with an empty string");

- This will print:
  - Strings in java start with an empty string

Example: Empty String check with == and equals
Find the output:

```
String s=new String();

if(s =="")

  System.out.println("==");

if(s.equals(""))

  System.out.println(".equals");
```

- This will print:
  - .equals
- Always use .equals() for comparison. == may return false even if two strings contain same value. This can happen even for empty strings.

Example: String concatenation
Find the output:

```
System.out.println(5 + 3);

System.out.println("Hello"+5 + 3);

System.out.println(5 + 3 + "Hello");
```

- This will print:
  - 8
  - Hello53
  - 8Hello