DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

Program: BCA

SUBJECT NAME: JAVA PROGRAMMING
SUBJECT CODE: 23BCA2C03
SEM: II

MODULE-4

- **Multi-Threading:** Multithreading and String Handling:
- Multithreading: Java thread model, creating multiple threads, thread priorities, synchronization, inter-thread communication, suspending, resuming and stopping threads. String Handling – string operations, character extraction, string comparison, searching and modifying strings, String Buffer, String Builder, StringTokenzier; File I/O: Reading from and Writing to File.

Introduction to Java thread model

Java provides built-in support for multi threaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.
In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously.
Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

Java Thread Model:

The Java run-time system depends on threads for many things. Threads reduce inefficiency by preventing the waste of CPU cycles.

Threads exist in several states. Following are those states:

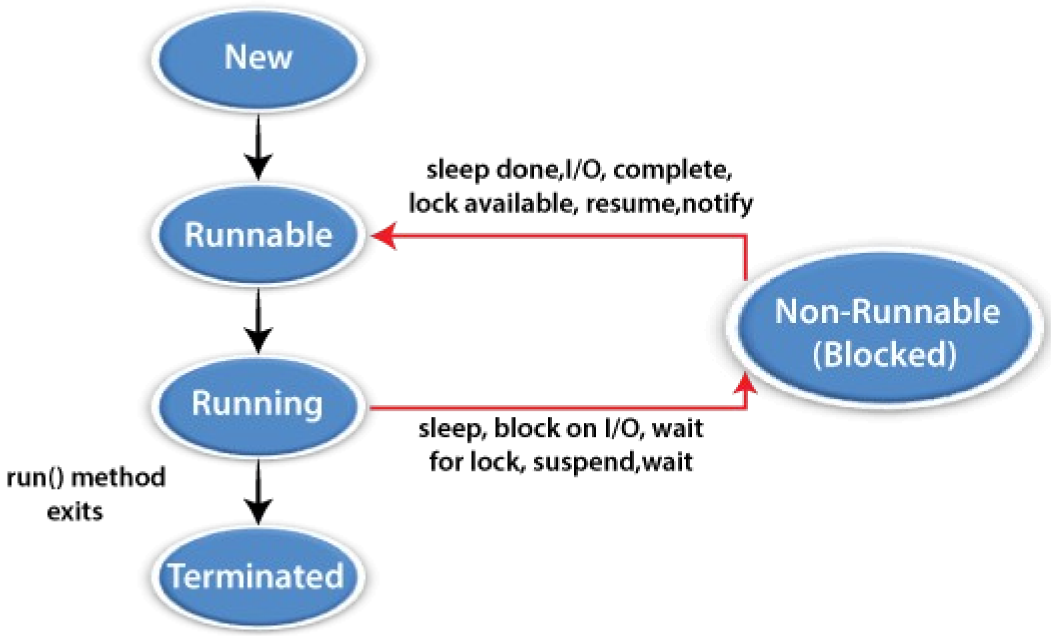
•**New** — When we create an instance of Thread class, a thread is in a new state.

•**Runnable** — The Java thread is in running state.

•**Suspended** — 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.

•**Blocked** — A java thread can be blocked when waiting for a resource.

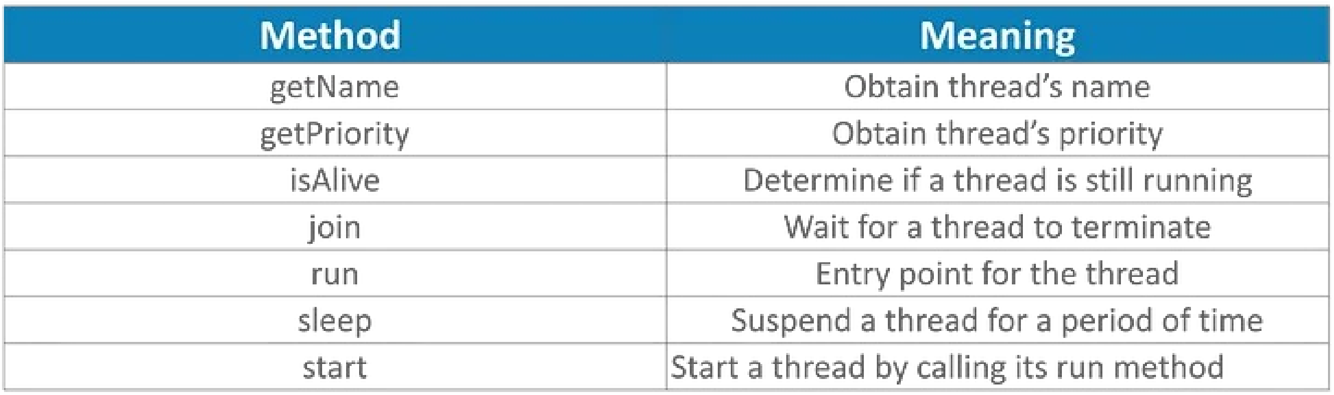
•**Terminated** — A thread can be terminated, which halts its execution immediately at any given time. Once a thread is terminated, it cannot be resumed.



Thread Class and Runnable Interface

Java’s multithreading system is built upon the Thread class, its methods, and its companion interface, **Runnable**. 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. The table below displays the same:



Method	Meaning
getName	Obtain thread’s name
getPriority	Obtain 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

Main Java Thread

Now let us see how to use Thread and Runnable interface to create and manage threads, beginning with the **main java thread,** that all Java programs have. So, let us discuss the main thread.

Why is Main Thread so important?

- Because this thread affects the other ‘child’ threads
- Because it performs various shutdown actions
- It is created automatically when your program is started.

So, this was the main thread. Let’s see how we can create a java thread?

How to Create a Java Thread?

Java lets you create a thread in following two ways:-

- By **implementing** the **Runnable interface**.
- By **extending** the **Thread**

Let’s see how both the ways help in implementing Java thread.

Runnable Interface

The easiest way to create a thread is to create a class that implements the **Runnable** interface.

To implement the Runnable interface, a class need only implement a single method called run( ), which is declared like this:

```
public void run( )
```

Inside run( ), we will define the code that constitutes the new thread

Example:

```
public class MyClass implements Runnable {
public void run(){
System.out.println("MyClass running");
   }
}
```

To execute the run() method by a thread, pass an instance of MyClass to a Thread in its constructor(*A **constructor in Java** is a block of code similar to a method that's called when an instance of an object is created*). Here is how that is done:

```
Thread t1 = new Thread(new MyClass ());
t1.start();
```

When the thread is started it will call the run() method of the MyClass instance instead of executing its own run() method. The above example would print out the text "**MyClass running**".

Extending Java Thread

The second way to create a thread is to create a new class that extends Thread, then override the run() method and then to create an instance of that class. The run() method is what is executed by the thread after you call start(). Here is an example of creating a Java Thread subclass:

```
public class MyClass extends Thread {
    public void run(){
    System.out.println("MyClass running");
  }
}
```

To create and start the above thread you can do like this:

```
MyClass t1 = new MyClass ();


t1.start();
```

When the run() method executes it will print out the text “**MyClass running**“.

*So far, we have been using only two threads: the **main** thread and one **child** thread. However, our program can affect as many threads as it needs. Let’s see how we can create multiple threads.*

Creating Multiple Threads

```
class MyThread implements Runnable {


String name;


Thread t;


    MyThread (String thread){


    name = threadname;


    t = new Thread(this, name);


System.out.println("New thread: " + t);


t.start();


}
```

```
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 MultiThread {

public static void main(String args[]) {

     new MyThread("One");

     new MyThread("Two");

     new NewThread("Three");

try {

     Thread.sleep(10000);

} catch (InterruptedException e) {

      System.out.println("Main thread Interrupted");

}

      System.out.println("Main thread exiting.");
```

```
    }



}
```

The output from the program is shown here:

```
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.
```

Synchronization and Inter-Thread Communication

It is already mentioned that threads in Java are running in the same memory space, and hence it is easy to communicate between two threads. Inter-thread communications allow threads to talk to or wait on each other. Again, because all the threads in a program share the same memory space, it is possible for two threads to access the same variables and methods in object. Problems may occur when two or more threads are accessing the same data concurrently, for example, one thread stores data into the shared object and the other threads reads data, there can be synchronization problem if the first thread has not finished storing the data before the second one goes to read it. So we need to take care to access the data by only one thread process at a time. Java provides unique language level support for such synchronization. in this Section we will learn how synchronization mechanism and inter-thread communications are possible in Java.

Synchronization

To solve the critical section problem, one usual concept is known what is called monitor. A monitor is an object which is used as a mutually exclusive lock (called mutex). Only one thread may own a monitor at a given time. When a thread acquires a lock it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the owner thread exits the monitor. But in Java, there is no such monitor. In fact, all Java object have their own implicit monitor associated with them. Here, the key word synchronized is used by which method (s) or block of statements can be made protected from the simultaneous access. With this, when a class is designed with threads in mind, the class designer decides which methods should not be allowed to execute concurrently. when a class with synchronized method is instantiated, the new object is given its own implicit monitor

In the above example, the keyword synchronized is used for the methods **void deposite(..) and void withdraw**so that these two methods will never run for the same object instance simultaneously.
Alternatively, if one wants to design a class that was not designed for multi-thread access and thus has non-synchronized methods, then it can be wrapped the call to the methods in a synchronized block. Here is the general form of the synchronized statement :

synchronized (Object) { block of statement(s) }

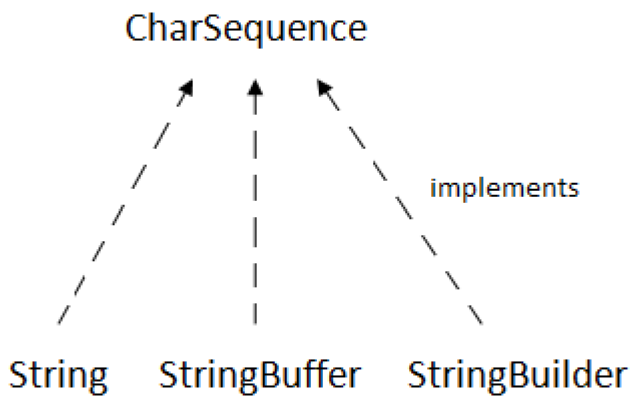
```
public void run( ) {           // in TransactionDeposite
           synchronized (accountX ) {

                 accountX.deposite(amount );

                }

        }



   public void run( )   {               // in TransactionWithdraw

          synchronized (accountY )       {

                   accountY.withdraw(amount );

                }

        }
```

```
public void run( ) {           // in TransactionDeposite
           synchronized (accountX ) {

                 accountX.deposite(amount );

                }

        }



   public void run( )   {               // in TransactionWithdraw

          synchronized (accountY )       {

                   accountY.withdraw(amount );
        }

    }
```

Java Thread suspend () method: The **suspend()** method of thread class puts the thread from running to waiting state. This method is used if you want to stop the thread execution and start it again when a certain event occurs. This method allows a thread to temporarily cease execution. The suspended thread can be resumed using the resume() method.

Syntax: public final void suspend()

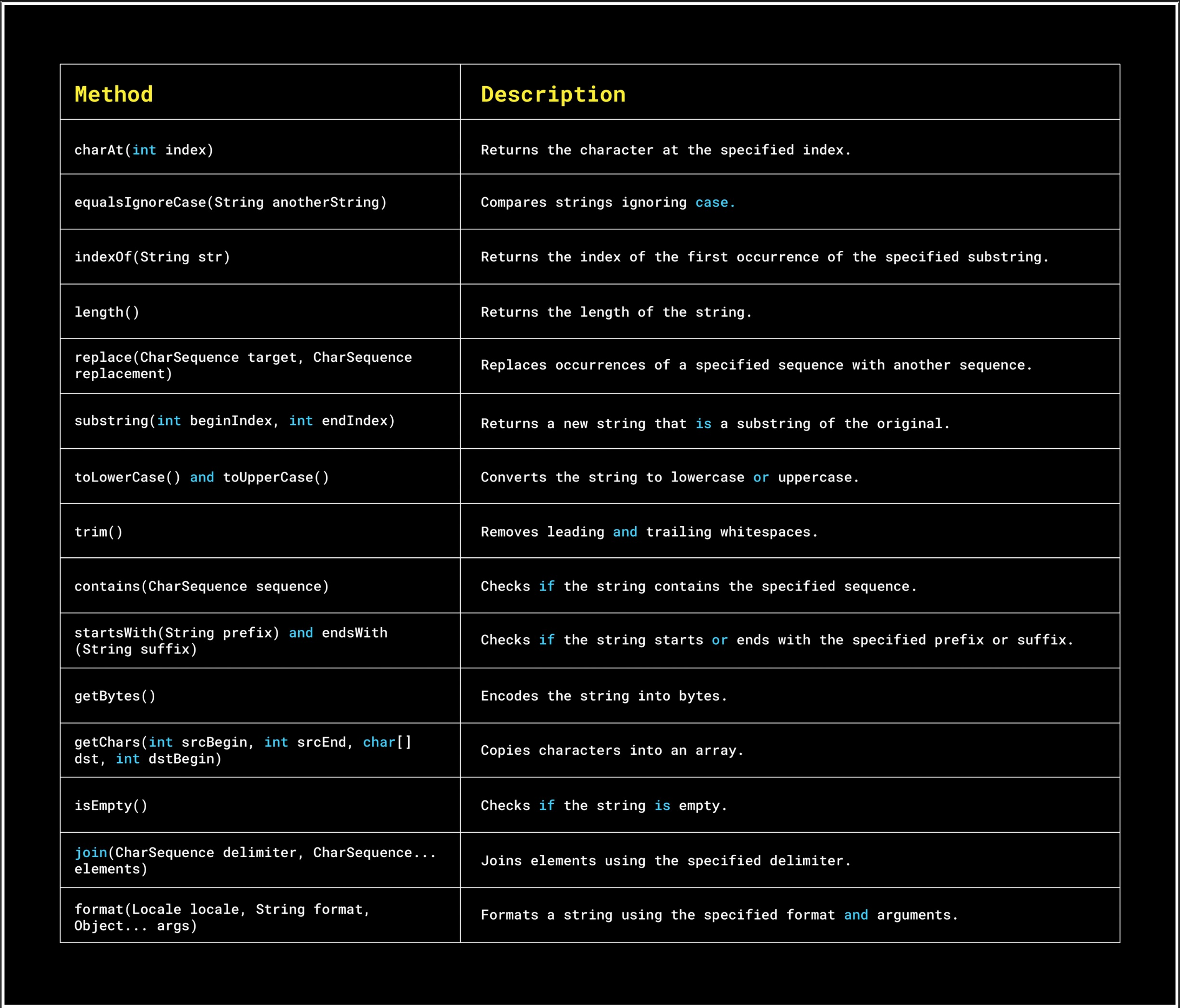
String Handling in Java: Java provides a vast range of string handling techniques that programmers can use to perform various operations on strings.
The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.



There are two ways to create String object:

1.By string literal

2.By new keyword



Method	Description
charAt(int index)	Returns the character at the specified index.
equalsIgnoreCase(String anotherString)	Compares strings ignoring case.
indexOf(String str)	Returns the index of the first occurrence of the specified substring.
length()	Returns the length of the string.
replace(CharSequence target, CharSequence replacement)	Replaces occurrences of a specified sequence with another sequence.
substring(int beginIndex, int endIndex)	Returns a new string that is a substring of the original.
toLowerCase() and toUpperCase()	Converts the string to lowercase or uppercase.
trim()	Removes leading and trailing whitespaces.
contains(CharSequence sequence)	Checks if the string contains the specified sequence.
startsWith(String prefix) and endsWith (String suffix)	Checks if the string starts or ends with the specified prefix or suffix.
getBytes()	Encodes the string into bytes.
getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	Copies characters into an array.
isEmpty()	Checks if the string is empty.
join(CharSequence delimiter, CharSequence... elements)	Joins elements using the specified delimiter.
format(Locale locale, String format, Object... args)	Formats a string using the specified format and arguments.

Example 1: String str1 = "Hello";

```
String str2 = "Hello";
if (str1.equals(str2)) {
    System.out.println("Strings are equal");
} else {
    System.out.println("Strings are not equal");
}
```

Output for the Above program is as follows

```
Strings are equal
```

Example2:

```
public class Demo{

public static void main(String[] args) {
   String s = "Hell";
   String s1 = "Hello";
   String s2 = "Hello";
   boolean b = s1.equals(s2);    //true
   System.out.println(b);
   b =     s.equals(s1) ;   //false
   System.out.println(b);
```

String Builder –

- •StringBuilder objects are String objects that can be modified. Hence to create a mutable (modifiable) string object, we use the Java StringBuilder class.
- •StringBuilder is not thread-safe or, in other words, not synchronized. One should prefer using StringBuffer if synchronization is mandatory.
- •The class is designed for use as a drop-in replacement for StringBuffer in places where the String Buffer used by a single thread.
- •Class Hierarchy – java.lang.Object -> java.lang-> Class StringBuilder
- •Constructors of StringBuilder Class –
  - •StringBuilder ( ) – This constructor constructs a string builder with no characters in it and an initial capacity of 16 characters.
  - •StringBuilder (int capacity) – This constructor constructs a string builder with no characters in it and an initial capacity specified by the capacity argument.
  - •StringBuilder(CharSequence seq) – This constructor constructs a string builder that contains the same characters as the specified CharSequence.
  - •StringBuilder (String str) – This constructor constructs a string builder initialized to the specified string's contents. Also, the initial capacity of the string builder is 16 plus the length of the string argument.
- •Some methods of StringBuilder class –
  - •append () – It concatenates the given argument(string representation) to the end of the invoking StringBuilder object.
  - •insert() – It inserts the given argument(string representation) into the invoking StringBuilder object at the given position.
  - •replace() – It replaces the string from a specified start index to the end index.
  - •reverse() – It reverses the characters within a StringBuilder object.

- •delete() – It deletes the string from the specified beginIndex to the endIndex.
- •capacity() – It returns the current capacity of the StringBuilder object.

String Buffer –

•StringBuffer is a peer class of String that creates mutable (modifiable) string, i.e., it represents growable and writable character sequences.
•One may insert characters and substrings in the middle or append them to the end in StringBuffer. It will automatically grow to make room for such additions and often pre-allocate more characters than we need to allow room for growth.
•StringBuffer is thread-safe; most of its methods are synchronized. So multiple threads cannot access or use the StringBuffer object at the same time.
•Constructors of StringBuffer Class –

- •StringBuffer( ) – This constructor reserves room for 16 characters without reallocation.
- •StringBuffer(int size) – This constructor accepts an integer argument that explicitly sets the buffer's size.
- •StringBuffer(String str) – This constructor accepts a String argument and sets the StringBuffer object's initial contents. Also, it reserves room for 16 more characters without reallocation.

•Some methods of StringBuffer class –

- •length() – It returns the StringBuffer object's length.
- •capacity() – It returns the capacity of the StringBuffer object.
- •append( ) – It adds text at the end of the existing text.
- •insert( ) – It inserts text at the specified index position.
- •reverse( ) – It reverses the characters within a StringBuffer object. It will return the reversed object on which this method was called.
- •ensureCapacity( ) – It increases the capacity of a StringBuffer object and ensures that the capacity is equal to the given minimum.

String Tokenizer –

•An application can break a string into tokens with the help of StringTokenizer class. It is a simple way to break a string.
•It does not have the facility to differentiate between numbers, quoted strings, identifiers, etc.
•A current position is maintained internally by the StringTokenizer object within the string to be tokenized.
•Constructors of StringTokenizer Class –

- •StringTokenizer(String str) – It accepts a string that needs to be tokenized. The default delimiters used are new line, space, tab, carriage return, and form feed.
- •StringTokenizer(String str, String delim) – It accepts a string and a set of delimiters used to tokenize the given string.
- •StringTokenizer(String str, String delim, boolean flag) -It accepts a string, set of delimiters, and a flag. If the flag is true, delimiter characters are considered to be tokens. If the flag is false, delimiter characters serve to separate tokens.

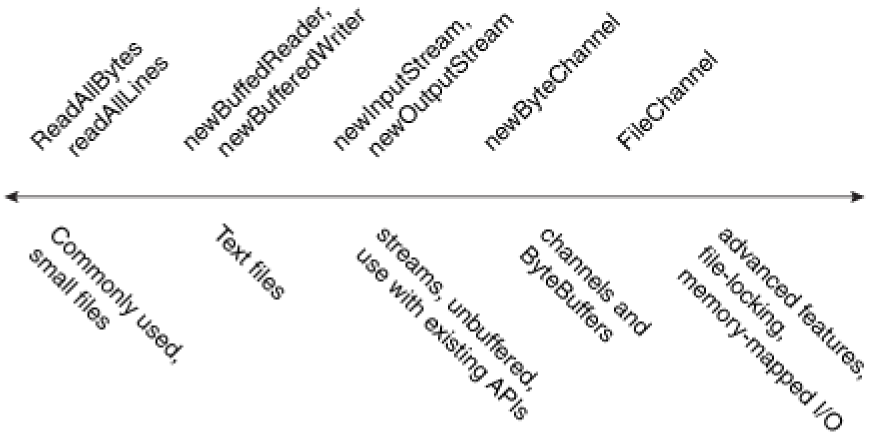
•Some methods of StringTokenizer Class –

- •hasMoreTokens() – It checks if there are more tokens available.
- •nextToken() – It returns the next token from the StringTokenizer object.
- •nextToken(String delim) – It returns the next token based on the delimiter.
- •hasMoreElements() – It is same as hasMoreTokens() method.
- •nextElement() – It is same as nextToken() but its return type is Object.
- •countTokens() – It returns the total number of tokens.

```
}


}
```

Reading, Writing, and Creating Files

There are a wide array of file I/O methods to choose from. To help make sense of the API, the following diagram arranges the file I/O methods by complexity.


ReadAllBytes readAllLines
newBuffedReader, newBufferedWriter
newInputStream, newOutputStream
newByteChannel
FileChannel
Commonly used, small files
Text files
streams, unbuffered, use with existing APIs
channels and ByteBuffers
advanced features, file-locking, memory-mapped I/O

OutputStream vs InputStream

The explanation of OutputStream and InputStream classes are given below:

OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Java File Writer Class: Java FileWriter class is used to write character-oriented data to a file. It is character-oriented class which is used for file handling in java.

Constructor	Description
FileWriter(String file)	Creates a new file. It gets file name in string.
FileWriter(File file)	Creates a new file. It gets file name in File object.

Method	Description
void write(String text)	It is used to write the string into FileWriter.
void write(char c)	It is used to write the char into FileWriter.
void write(char[] c)	It is used to write char array into FileWriter.
void flush()	It is used to flushes the data of FileWriter.
void close()	It is used to close the FileWriter.

Example:
```
try {
    FileWriter writer1 = new FileWriter("path/to/your/file1.txt");

    File file = new File("path/to/your/file2.txt");

    FileWriter writer2 = new FileWriter(file);

    // code to write...

} catch (IOException ex) {
    System.err.println(ex);
}
```

Java FileReader Class

Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.

Constructor	Description
FileReader(String file)	It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
FileReader(File file)	It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

Method	Description
int read()	It is used to return a character in ASCII form. It returns -1 at the end of file.

void close()	It is used to close the FileReader class.

Example:

```
try {

 FileReader reader1 = new FileReader("/path/to/yourfile1.txt");

 File file = new File("/path/to/yourfile2.txt");
 FileReader reader2 = new FileReader(file);
 // code to read…

} catch (FileNotFoundException ex) {
 System.err.println(ex);
}
```