Files

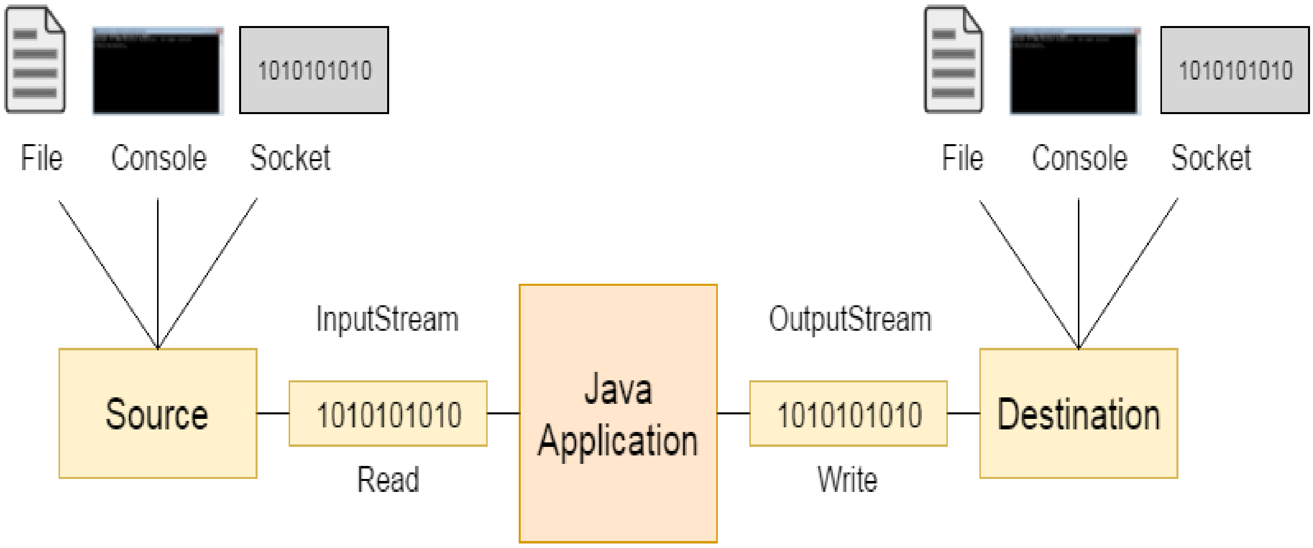
Introduction to I/O Streams: *Byte Streams, Character Streams. File I/O.*

Introduction to I/O Streams in Java

I/O (Input/Output) streams are an essential part of Java programming that allows the program to communicate with the outside world, whether it's reading data from files, writing data to files, or interacting with other input/output devices like keyboards, networks, or consoles.

In Java, the **java.io** package provides the classes needed to work with I/O operations. The primary classes in this package allow reading from and writing to files, consoles, memory, and other data sources.

Stream: A stream in Java is an abstraction that allows you to read or write data in a continuous flow. It represents the flow of data between your program and some external source (like a file, network connection, or keyboard).

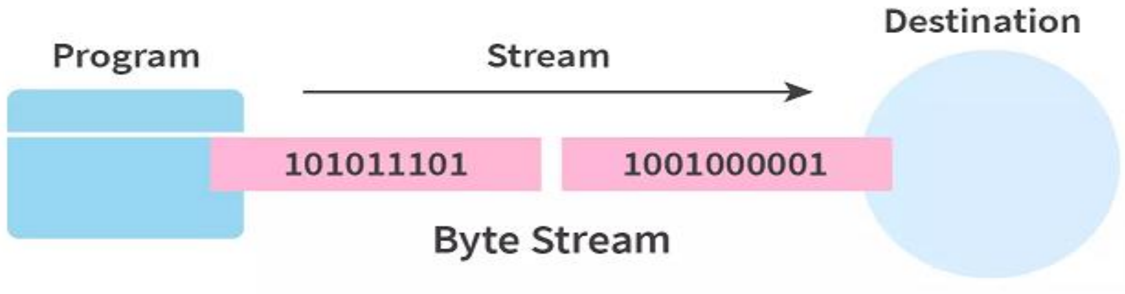


Types of Streams in Java:

1. Byte Streams
2. Character Streams

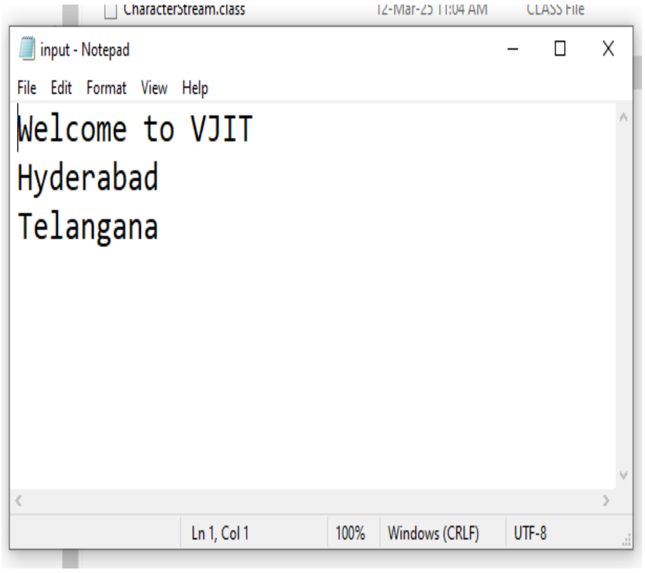
1. Byte Streams:

- Byte Streams works with raw binary data and this is used to process data byte by byte (8 bits).
- Used for handling raw binary data (e.g., image, audio, or video files).
- The basic classes for byte I/O are InputStream and OutputStream.
  - **InputStream**: Used to read bytes from an input source.
  - **OutputStream**: Used to write bytes to an output destination.

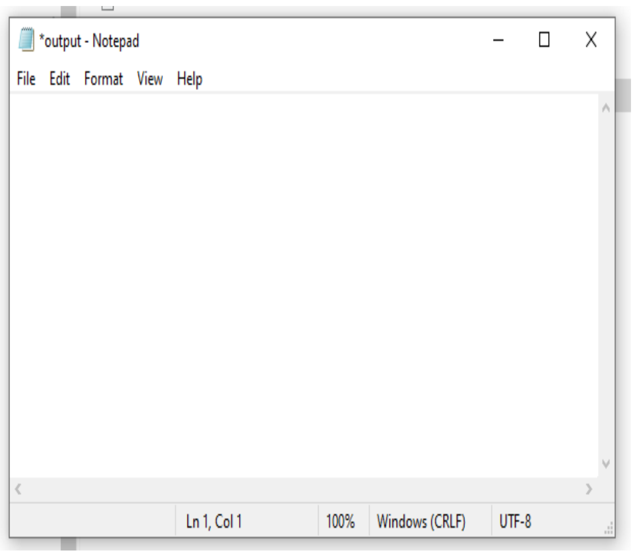


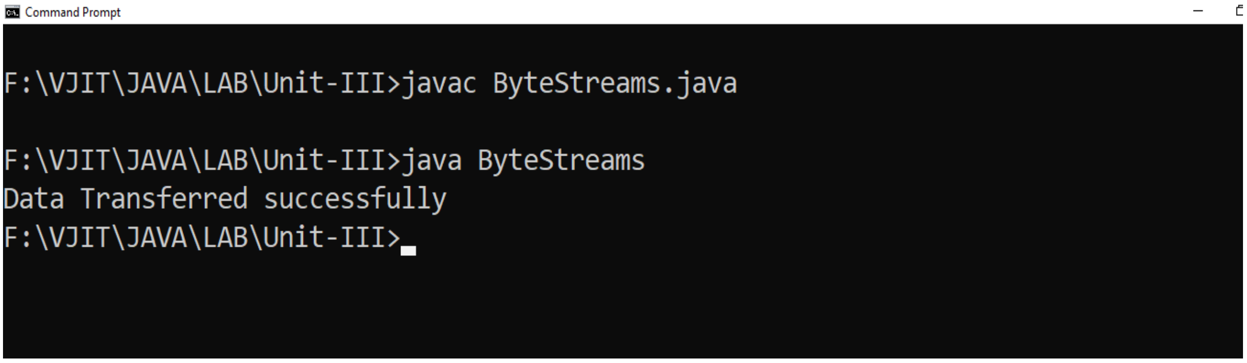
Stream class	Description
BufferedInputStream	It is used for Buffered Input Stream.
DataInputStream	It contains method for reading java standard datatypes.
FileInputStream	This is used to reads from a file
InputStream	This is an abstract class that describes stream input.
PrintStream	This contains the most used print() and println() method
BufferedOutputStream	This is used for Buffered Output Stream.
DataOutputStream	This contains method for writing java standard data types.
FileOutputStream	This is used to write to a file.
OutputStream	This is an abstract class that describe stream output.

```
//Program to demonstrate on ByteStreams
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class ByteStreams
{
   public static void main(String args[]) throws IOException
   {
      FileInputStream in = null;
      FileOutputStream out = null;

      try
      {
               in = new FileInputStream("input.txt");
               out = new FileOutputStream("output.txt");
               int c;
               while ((c = in.read()) != -1)
              {
                 out.write(c);
              }
              System.out.print("Data Transferred successfully");
      }
       finally
      {
                if (in != null)
                {
                  in.close();
                }
                if (out != null)
                {
                  out.close();
                }
      }
   }
}
```

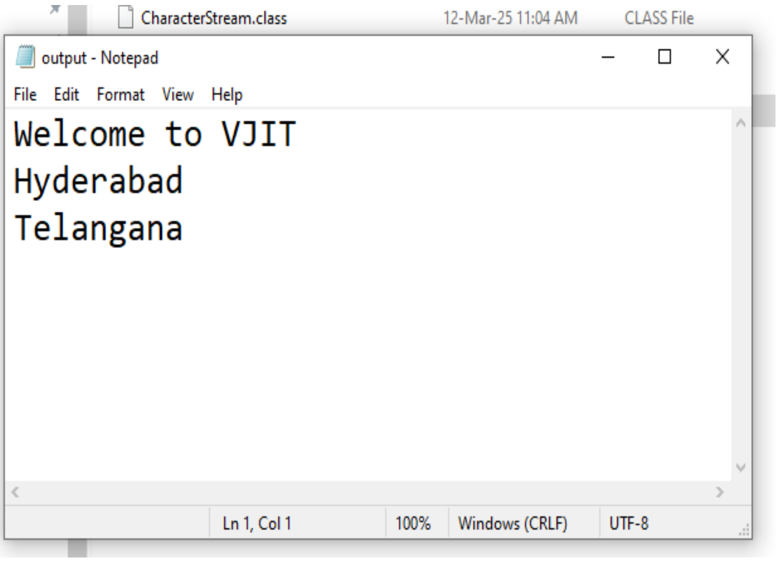


```
Welcome to VJIT
Hyderabad
Telangana
```





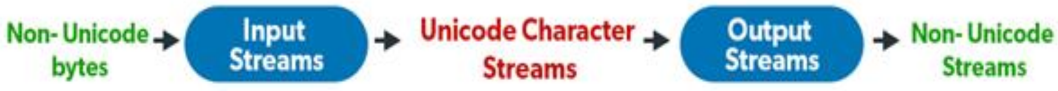
```
F:\VJIT\JAVA\LAB\Unit-III>javac ByteStreams.java

F:\VJIT\JAVA\LAB\Unit-III>java ByteStreams
Data Transferred successfully
F:\VJIT\JAVA\LAB\Unit-III>
```



```
Welcome to VJIT
Hyderabad
Telangana
```

2. Character Streams:

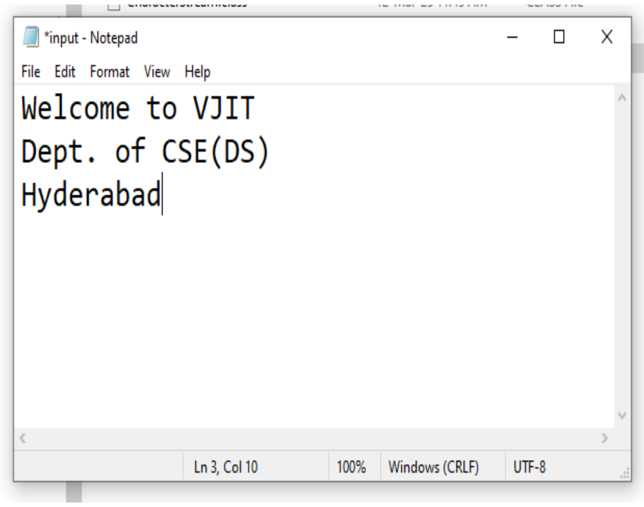
Characters are stored using Unicode conventions. Character stream automatically allows us to read/write data character by character. For example, FileReader and FileWriter are character streams used to read from the source and write to the destination.



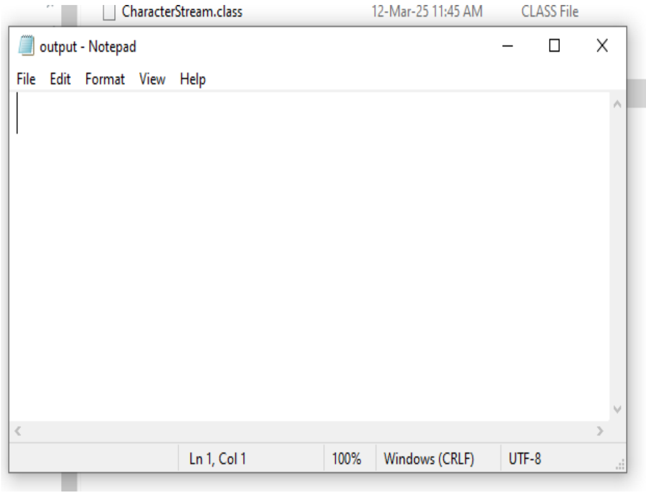
- Used for handling text data (characters).
- The basic classes for character I/O are Reader and Writer.
  - **Reader**: Used to read characters from an input source.
  - **Writer**: Used to write characters to an output destination.

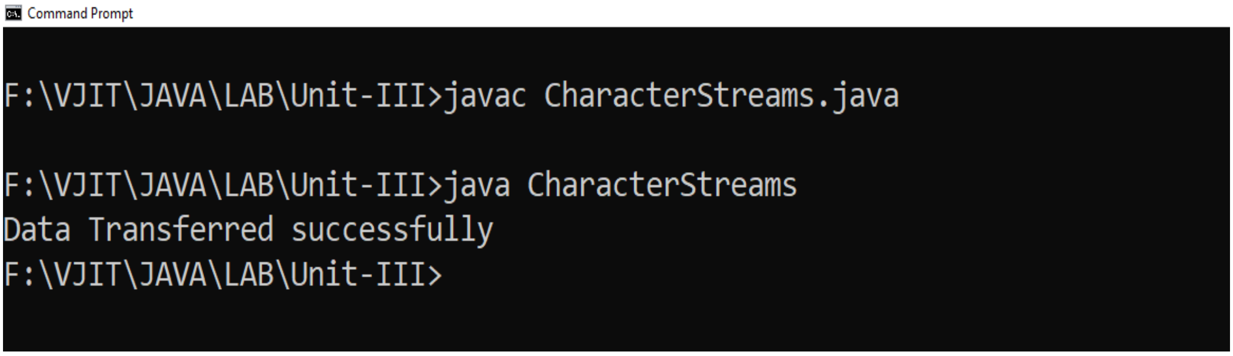
Stream class	Description
BufferedReader	It is used to handle buffered input stream.
FileReader	This is an input stream that reads from file.
InputStreamReader	This input stream is used to translate byte to character.
OutputStreamReader	This output stream is used to translate character to byte.
Reader	This is an abstract class that define character stream input.
PrintWriter	This contains the most used print() and println() method
Writer	This is an abstract class that define character stream output.
BufferedWriter	This is used to handle buffered output stream.
FileWriter	This is used to output stream that writes to fil//Program to

```
//Program to demonstrate on CharacterStreams
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CharacterStreams
{
  public static void main(String args[]) throws IOException
  {
    FileReader in = null;
    FileWriter out = null;

      try
      {
              in = new FileReader("input.txt");
              out = new FileWriter("output.txt");
              int c;
              while ((c = in.read()) != -1)
             {
                out.write(c);
              }
           System.out.print("Data Transferred successfully");
      }
      finally
      {
              if (in != null)
              {
                   in.close();
               }
                if (out != null)
                {
                    out.close();
                 }
        }
  }
}
```

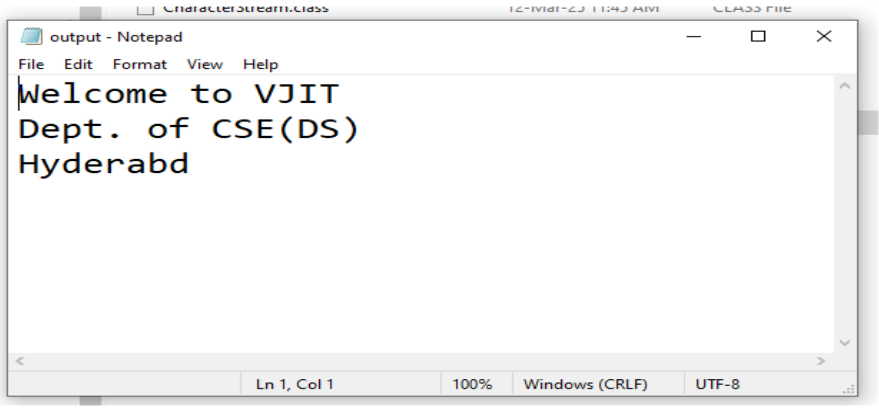


```
Welcome to VJIT
Dept. of CSE(DS)
Hyderabad
```





```
F:\VJIT\JAVA\LAB\Unit-III>javac CharacterStreams.java

F:\VJIT\JAVA\LAB\Unit-III>java CharacterStreams
Data Transferred successfully
F:\VJIT\JAVA\LAB\Unit-III>
```



```
Welcome to VJIT
Dept. of CSE(DS)
Hyderabd
```

Difference Between Byte Stream and Character Stream

Byte Stream	Character Stream
Byte stream is used to perform input and output operations of 8-bit bytes.	Character stream is used to perform input and output operations of 16-bit Unicode.
It processes data byte by byte.	It processes data character by character.
Common classes for Byte stream are FileInputStream and FileOutputStream.	Common classes for Character streams are FileReader and FileWriter.

File I/O

File I/O (Input/Output) operations in Java are crucial for reading from and writing to files. Java provides a comprehensive set of classes in the java.io package that enable file handling. These classes are capable of working with both text files and binary files.

The **Java File** class is an abstract representation of file.

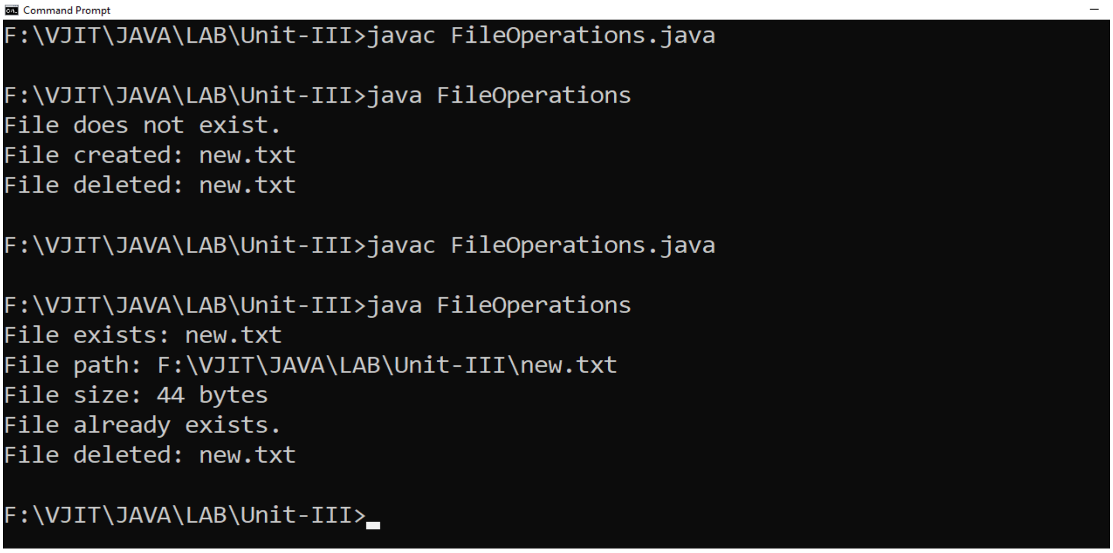
Basic File Operations

The primary operations related to file handling include:

- Reading from a file
- Writing to a file
- Creating, deleting, or renaming files
- Checking file properties (exists, isDirectory, etc.)

*//program to perform **File I/O Operations***

```
import java.io.File;
import java.io.IOException;
public class FileOperations
{
   public static void main(String[] args)
    {
      // Create a File object for the file
      File file = new File("new.txt");
      // Check if file exists
       if (file.exists())
       {
         System.out.println("File exists: " + file.getName());
         System.out.println("File path: " + file.getAbsolutePath());
         System.out.println("File size: " + file.length() + " bytes");
        }
       else
        {
         System.out.println("File does not exist.");
         }
        // Create a new file (if not already existing)
        try
        {
              if (file.createNewFile())
              {
                 System.out.println("File created: " + file.getName());
               }
               else
              {
                 System.out.println("File already exists.");
              }
        }
        catch (IOException e)
        {
           e.printStackTrace();
         }
               // Delete a file
               if (file.delete())
               {
                  System.out.println("File deleted: " + file.getName());
               }
               else
              {
                  System.out.println("Failed to delete the file.");
               }
      }
   }
```



```
F:\VJIT\JAVA\LAB\Unit-III>javac FileOperations.java

F:\VJIT\JAVA\LAB\Unit-III>java FileOperations
File does not exist.
File created: new.txt
File deleted: new.txt

F:\VJIT\JAVA\LAB\Unit-III>javac FileOperations.java

F:\VJIT\JAVA\LAB\Unit-III>java FileOperations
File exists: new.txt
File path: F:\VJIT\JAVA\LAB\Unit-III\new.txt
File size: 44 bytes
File already exists.
File deleted: new.txt

F:\VJIT\JAVA\LAB\Unit-III>_
```

Multi Threading PART-2

Differences between multi threading and multitasking, thread life cycle, creating threads, thread priorities, synchronizing threads, inter thread communication.

Multithreading

Multithreading in Java is a process of executing multiple threads simultaneously. The main reason for incorporating threads into real-world applications to improve performance, responsiveness and resource utilization.

A **thread** is a **lightweight process**, or the smallest component of the process, that enables software to work more effectively by doing numerous tasks concurrently.

Benefits of Multithreading in Java

Multithreading is a powerful feature in Java that enables multiple threads to run concurrently within a program. Here are the key benefits:

---

1. Gaming Applications

- Games require multiple threads for rendering graphics, handling user input, playing sounds, and managing AI logic.
- **Unity** and **Unreal Engine** use multithreading to optimize real-time game performance.

✓ **Example**: A racing game where one thread handles rendering, another handles physics, and another manages network communication.

---

2. Chat Applications & Messaging Services

- Applications like **WhatsApp, Slack, and Discord** use multithreading to manage multiple conversations in real time.
- One thread handles incoming messages, another updates the UI, and another sends messages to the server.

✓**Example**: Receiving and sending messages simultaneously in a chat app.

---

3. Database Operations

- Multithreading is used in **JDBC connections** to execute multiple database queries at the same time.
- Improves database transaction performance and reduces latency.

✓**Example**: A banking system processing multiple transactions concurrently.

4. Parallel Processing in Data Science & Machine Learning

- Multithreading speeds up computation-heavy tasks like **big data processing (Hadoop, Spark)** and **AI model training**.
- Enables parallel execution of matrix operations and large dataset analysis.

✓**Example**: Running multiple machine learning model training processes at once.

---

5. Real-Time Stock Trading Systems

- Stock trading platforms use multithreading to process thousands of trades per second.
- One thread updates stock prices, another executes trades, and another handles UI updates.

✓ **Example**: A real-time trading app like Bloomberg or Robinhood updating stock prices live.

---

6. Video Streaming & Media Players

- Streaming platforms like **YouTube, Netflix, and VLC** use multithreading for buffering, playback, and video compression.
- Separate threads handle downloading, decoding, and rendering frames.

✓**Example**: Watching a movie while buffering continues in the background.

---

7. Web Scraping & Crawling

- Search engines like **Google** use multithreading in their crawlers to scan multiple web pages simultaneously.
- This speeds up the process of indexing billions of websites.

✓**Example**: A multithreaded web scraper extracting data from multiple URLs at once.

---

8. IoT & Embedded Systems

- Smart home devices (e.g., Alexa, Google Nest) use multithreading to handle voice recognition, Wi-Fi communication, and sensor data processing simultaneously.

✓**Example**: A smart thermostat adjusting temperature while taking voice commands.

---

9. Cloud Computing & Distributed Systems

- Cloud services like **AWS Lambda** and **Google Cloud Functions** use multithreading to handle multiple user requests in parallel.
- Helps in load balancing and distributing computing power efficiently.

✓**Example**: A cloud server processing multiple API requests simultaneously.

10. Web Servers & Application Servers

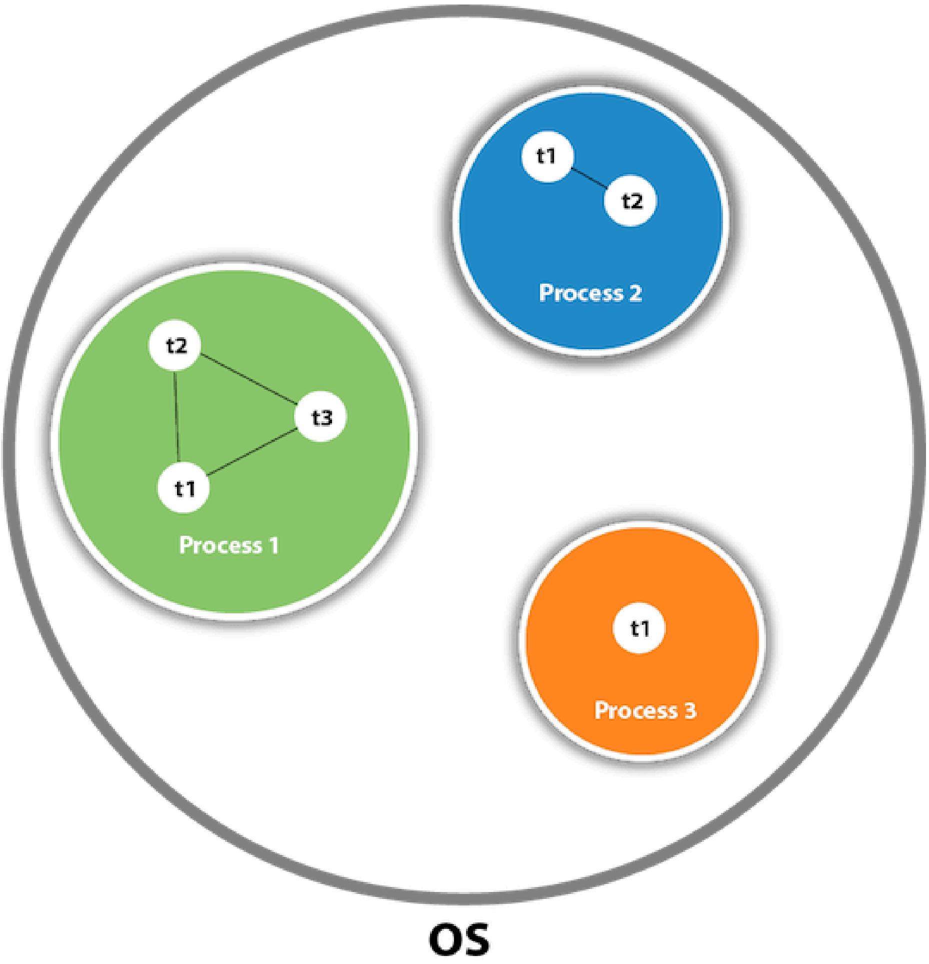
- Web servers like **Apache Tomcat** and **Jetty** use multithreading to handle multiple user requests concurrently.
- Each request (e.g., loading a webpage, submitting a form) runs on a separate thread, ensuring fast response times.

✅**Example**: A web server handling multiple users' requests simultaneously.

Multitasking

Multitasking refers to the ability of a computer to execute multiple tasks or processes at the same time. There are two types of multitasking:

1. **Process-Based Multitasking**
2. **Thread-Based Multitasking**



1. **Process-Based Multitasking**:
   - In this type of multitasking, the operating system manages multiple processes. Each process runs in its own memory space and the CPU switches between them.
   - **Examples**: Running multiple applications (like a web browser, a word processor, and a media player) at the same time.
   - In Java, process-based multitasking is handled by the **operating system** and Java interacts with the OS for this functionality.
   - Each process has an address in memory. In other words, each process allocates a separate memory area.
   - A process is heavyweight.
   - Cost of communication between the process is high.
   - Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.
2. **Thread-Based Multitasking**:
   - This involves multiple threads within the same process, where each thread can execute a part of the task concurrently.
   - This is where **multithreading** comes into play.
   - Threads share the same address space.
   - A thread is lightweight.
   - Cost of communication between the thread is low.

Differences between Multitasking and Multithreading

S. No	Multitasking	Multithreading
1	Multitasking enables users to perform multiple tasks concurrently using the CPU.	Multithreading involves creating multiple threads within a single process, enhancing computational power.
2	Multitasking often requires the CPU to switch between different tasks.	Multithreading also involves CPU context switching between threads.
3	In multitasking, processes have separate memory spaces.	In multithreading, threads share the same memory space within a process.
4	Multitasking can include multiprocessing, where multiple processes run independently.	Multithreading focuses on concurrent execution within a single process and doesn't necessarily involve multiprocessing.
5	Multitasking allocates CPU time for executing multiple tasks concurrently.	Multithreading provides CPU time for executing multiple threads within a process concurrently.
6	In multitasking, processes typically do not share resources; each has its own allocated resources.	In multithreading, threads share the same resources within a process.
7	Multitasking may be slower than multithreading, depending on the system and tasks.	Multithreading is generally faster due to reduced overhead in managing threads.
8	Terminating a process in multitasking can take more time.	Terminating a thread in multithreading is typically faster as it involves less cleanup.

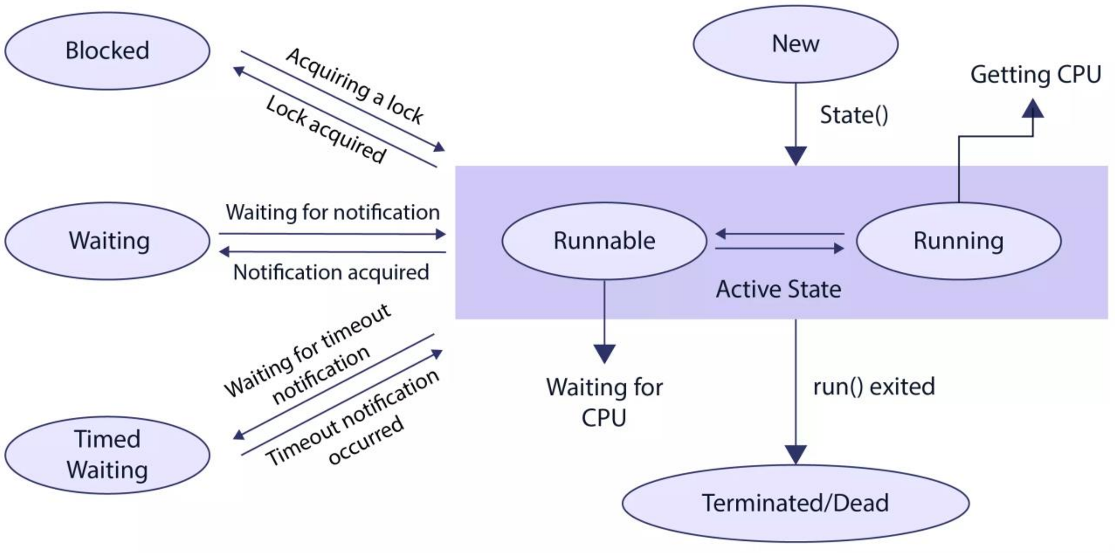
S. No	Multitasking	Multithreading
9	Multitasking provides isolation and memory protection between processes.	Multithreading lacks isolation and memory protection, as threads share the same memory space.
10	Multitasking is crucial for developing efficient programs to perform multiple concurrent tasks.	Multithreading is essential for developing efficient operating systems and applications.
11	**Examples** of multitasking include running multiple applications on a computer or multiple servers on a network.	**Examples** of multithreading include splitting a video streaming task into multiple threads in an application.

Thread Life Cycle

A **thread** is a **lightweight process**, or the smallest component of the process, that enables software to work more effectively by doing numerous tasks concurrently.

A thread has five states in the span of its creation to its termination:

1. **New State**
2. **Active State**
3. **Waiting/Blocked State**
4. **Timed Waiting State**
5. **Terminated State**



Life Cycle of a Thread

1. New (Created) State

- A thread is in the **NEW** state when it is created but **not yet started**.
- It remains in this state until the start() method is called.

✅**Example:**

```
Thread t = new Thread();  // Thread created but not started
```

2. Runnable State

- After calling start(), the thread moves to the **RUNNABLE** state.
- It is **ready to run** but waits for the CPU to schedule it.
- The OS **scheduling algorithm** decides when the thread will execute.

✅**Example:**

```
t.start();  // Thread is now Runnable
```

3. Running State

- When a thread gets CPU time, it moves to the **RUNNING** state.
- This is where the run() method executes.
- A thread stays in this state until it completes execution or is paused.

✓**Example:**

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

4. Blocked / Waiting / Timed Waiting State

A thread can be **temporarily paused** in different ways:

a) Blocked State

- A thread is **BLOCKED** when it tries to access a **locked resource** but has to wait.
- It remains in this state until the resource is released.

✓**Example:**

```
 synchronized (lock)
{
   // Thread trying to acquire lock
}
```

b) Waiting State

- A thread enters the **WAITING** state when it waits indefinitely for another thread's signal.
- It must be notified using notify() or notifyAll().

✓**Example:**

```
 synchronized (lock)
{
   lock.wait();  // Thread goes into waiting state
 }
```

c) Timed Waiting State

- A thread enters the **TIMED_WAITING** state when it waits for a **specific time** before resuming.
- Methods like Thread.sleep(), wait(time), and join(time) put a thread in this state.

✅**Example:**

```
Thread.sleep(5000);  // Thread sleeps for 5 seconds
```

5. Terminated (Dead) State

- A thread enters the **TERMINATED** state after completing execution or being stopped forcefully.
- Once terminated, it **cannot be restarted**.

✅**Example:**

```
System.out.println("Thread execution completed.");
```

Creating Threads

There are the two ways to create a thread:

- **By Extending Thread Class**
- **By Implementing Runnable Interface**

1. By Extending Thread Class

We declare a sub-class or a child class that inherits the Thread class. The child class should override the run() method of the Thread class.The new thread can be associated with the main thread by invoking the start() method.

After invoking the start() method, the new thread can start its execution. When the new thread starts its execution, the main thread is moved to the waiting state.

Methods of Thread Class

Let us now know some of the most commonly used methods of the Thread class.

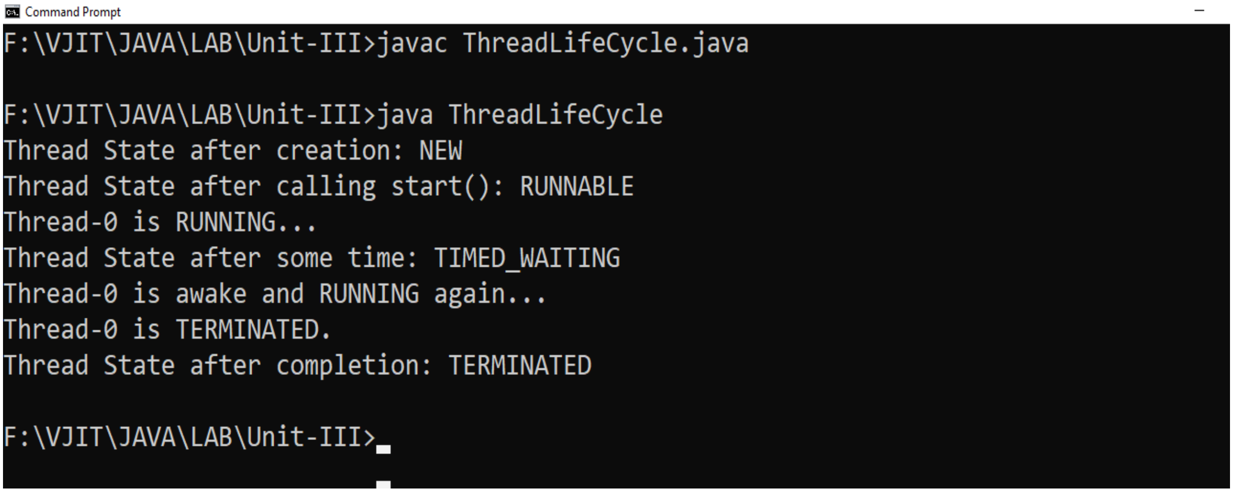
Method Name	Usage
void run()	used to run a thread.
void start()	used to start execution for a thread.
void sleep(long milliseconds)	used to temporarily terminate the invoking thread's execution for a specified duration of time.
void join()	used to wait for a thread to die.
void join(long milliseconds)	used to wait for a thread to die for a specified duration of time.
int getPriority()	used to get the priority of the thread.
int setPriority(int priority)	used to set or change the priority of the thread.
String getName()	used to get the name of the thread.
void setName(String name)	used to set or change the name of the thread.
Thread currentThread()	used to get the reference of currently executing thread.
int getId()	used to get the id of the thread.
Thread. State getState()	used to get the state of the thread.
boolean isAlive()	used to check if the thread is alive or not.
void yield()	used to temporarily pause the currently executing thread and allow other threads to execute.
boolean isDaemon()	used to check the thread is daemon thread.
void setDaemon(boolean b)	used to mark the thread as daemon or user thread.
void interrupt()	used to make interrupts the thread.
boolean isInterrupted()	used to check if the current thread has been interrupted or not.

Constructors of Thread Class

In java.lang.Thread class, several constructors have been declared for different purposes. Some of them are:

Constructor	Usage
Thread()	no-argument constructor.
Thread (String name)	takes a string as an argument.
Thread(Runnable r)	takes reference (**r**) of a Runnable object as an argument.

```
    class MyThread extends Thread
    {
       public void run()
      {
          try
         {
            System.out.println(Thread.currentThread().getName() + " is RUNNING...");
             // Simulate a waiting state
             Thread.sleep(2000);
System.out.println(Thread.currentThread().getName() + " awake &RUNNING again...");
          }
          catch (InterruptedException e)
         {
           System.out.println(Thread.currentThread().getName() + " was interrupted.");
          }
        System.out.println(Thread.currentThread().getName() + " is TERMINATED.");
       }
    }
    public class ThreadLifeCycle
    {
       public static void main(String[] args)
      {
          // NEW State - Thread is created but not started
          MyThread t1 = new MyThread();
          System.out.println("Thread State after creation: " + t1.getState());
          t1.start();
         System.out.println("Thread State after calling start(): " + t1.getState());
        // Let’s check the state after some time
            try
           {
               Thread.sleep(500);
               System.out.println("Thread State after some time: " + t1.getState());
            }
            catch (InterruptedException e)
            {
               e.printStackTrace();
            }
            // Wait for thread to complete
            try
            {
               t1.join();
            }
           catch (InterruptedException e)
           {
               e.printStackTrace();
            }
            // TERMINATED State - Thread execution is complete
            System.out.println("Thread State after completion: " + t1.getState());
         }
      }
```



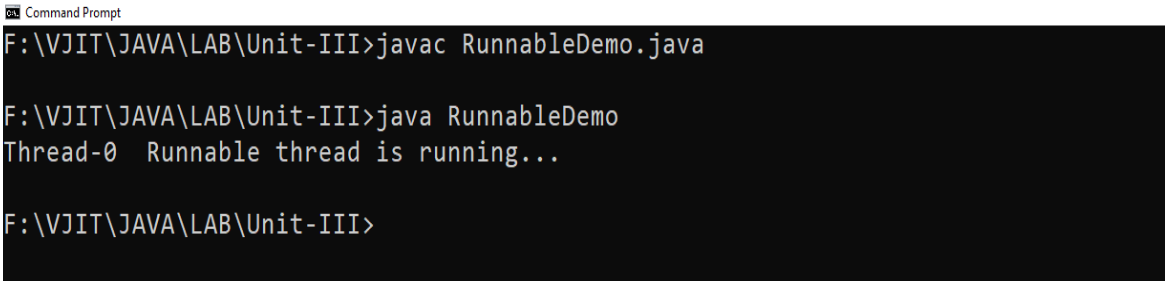
```
F:\VJIT\JAVA\LAB\Unit-III>javac ThreadLifeCycle.java

F:\VJIT\JAVA\LAB\Unit-III>java ThreadLifeCycle
Thread State after creation: NEW
Thread State after calling start(): RUNNABLE
Thread-0 is RUNNING...
Thread State after some time: TIMED_WAITING
Thread-0 is awake and RUNNING again...
Thread-0 is TERMINATED.
Thread State after completion: TERMINATED

F:\VJIT\JAVA\LAB\Unit-III>
```

2. By Implementing a Runnable Interface

We can also create a thread in Java by implementing the Runnable interface.

We pass the reference of the Runnable implemented class to the Thread object's constructor to create a new thread. After passing the reference, we invoke the start() method to start the execution of the newly created thread.

```
class MyThread implements Runnable
{
   public void run()
{
System.out.println(Thread.currentThread().getName()+"  Runnable thread is running.");
   }
}

public class RunnableDemo
{
   public static void main(String[] args)
  {
      MyThread mt = new MyThread();
      Thread t1 = new Thread(mt); // Pass Runnable instance to Thread
      t1.start();
   }
}
```



```
F:\VJIT\JAVA\LAB\Unit-III>javac RunnableDemo.java

F:\VJIT\JAVA\LAB\Unit-III>java RunnableDemo
Thread-0  Runnable thread is running...

F:\VJIT\JAVA\LAB\Unit-III>
```

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled. You can get and set the priority of a Thread. Thread class provides methods and constants for working with the priorities of a Thread.

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. Priority can either be given by JVM while creating the thread or it can be given by the programmer explicitly.

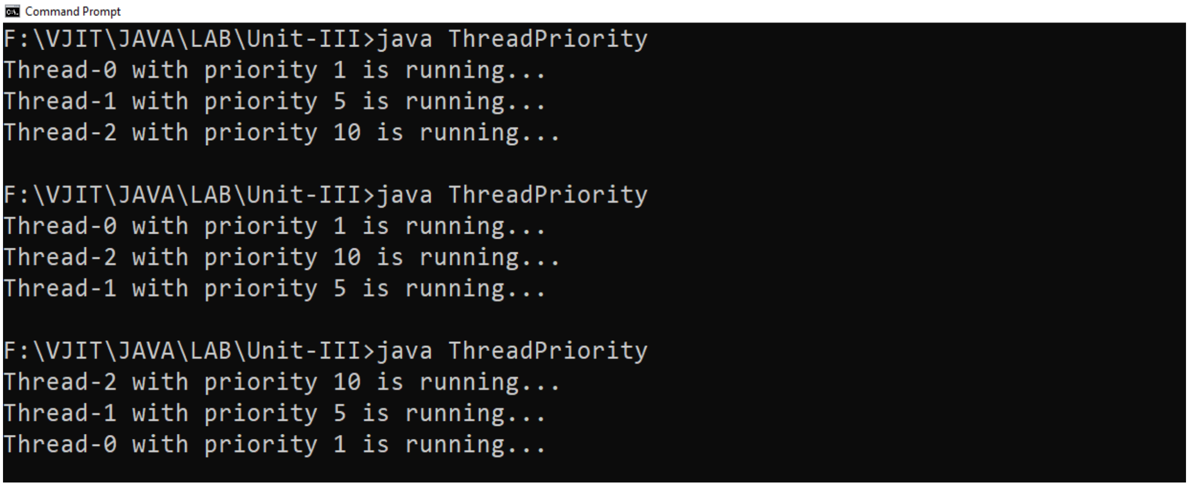
Built-in Property Constants of Thread Class

Constant	Description
NORM_PRIORITY	Sets the default priority for the Thread. (Priority: 5)
MIN_PRIORITY	Sets the Minimum Priority for the Thread. (Priority: 1)
MAX_PRIORITY	Sets the Maximum Priority for the Thread. (Priority: 10)

Thread Priority Setter and Getter Methods

Method	Description
Thread.getPriority()	used to get the priority of a thread.
Thread.setPriority()	used to set the priority of a thread, it accepts the priority value and updates an existing priority with the given priority.

```
class MyThread extends Thread
{
   public void run()
   {
      System.out.println(Thread.currentThread().getName() + " with priority " +
                     Thread.currentThread().getPriority() + " is running...");
   }
}
public class ThreadPriority
{
   public static void main(String[] args)
  {
      // Creating three threads
      MyThread t1 = new MyThread();
      MyThread t2 = new MyThread();
      MyThread t3 = new MyThread();
      // Setting priorities
      t1.setPriority(Thread.MIN_PRIORITY);  // Priority 1
      t2.setPriority(Thread.NORM_PRIORITY); // Priority 5 (Default)
      t3.setPriority(Thread.MAX_PRIORITY);  // Priority 10
      // Starting threads
      t1.start();
      t2.start();
      t3.start();
   }
}
```



```
F:\VJIT\JAVA\LAB\Unit-III>java ThreadPriority
Thread-0 with priority 1 is running...
Thread-1 with priority 5 is running...
Thread-2 with priority 10 is running...

F:\VJIT\JAVA\LAB\Unit-III>java ThreadPriority
Thread-0 with priority 1 is running...
Thread-2 with priority 10 is running...
Thread-1 with priority 5 is running...

F:\VJIT\JAVA\LAB\Unit-III>java ThreadPriority
Thread-2 with priority 10 is running...
Thread-1 with priority 5 is running...
Thread-0 with priority 1 is running...
```

Synchronizing Threads

Synchronization in Java is used to control the access of multiple threads to shared resources. It prevents race conditions and ensures data consistency.

When multiple threads try to access a shared resource simultaneously, **race conditions** can occur. Synchronization helps in **avoiding thread interference** and maintaining **data integrity**.

Types of Synchronization

There are the following two types of synchronization:

1. **Process Synchronization**
2. **Thread Synchronization**

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of Thread Synchronization in Java.

1. **Mutual Exclusive**
    1. Synchronized Method.
    2. Synchronized Block.
    3. Static Synchronization.
2. **Cooperation** (Inter-Thread Communication)

1. Synchronized Methods

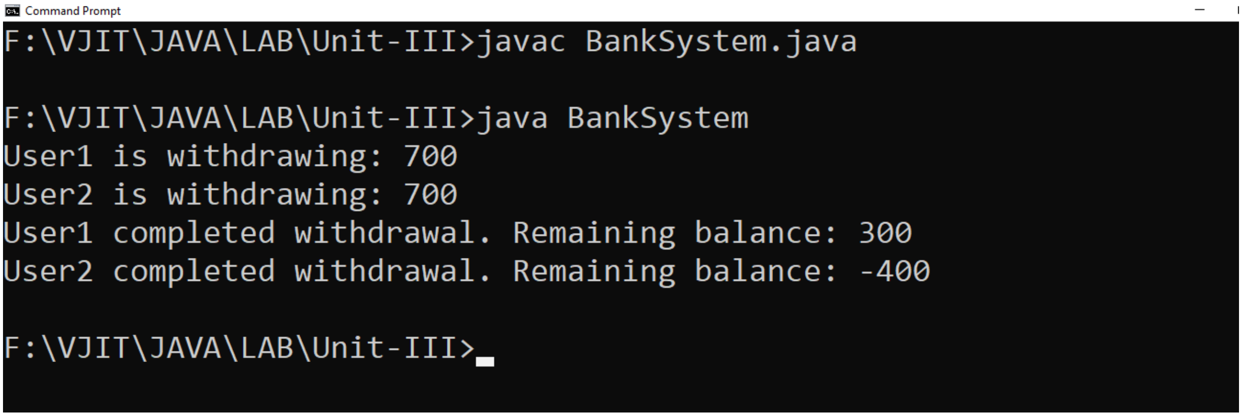
- The **synchronized** keyword ensures that only one thread at a time can execute a method.
- It locks the entire method on the object.

```
class Example
{
   synchronized void methodName()
 {
       // Critical section (only one thread can execute this at a time)
    }
}
```

```
    class BankAccount
    {
        private int balance = 1000; // Initial balance

        synchronized void withdraw(int amount)
       {
            if (balance >= amount)
           {
System.out.println(Thread.currentThread().getName() + " is withdrawing: " + amount);
                try
              {
                    Thread.sleep(500); // Simulate processing time
                }
               catch (InterruptedException e)
              {
                    System.out.println(e);
                }
                balance =balance-amount;
System.out.println(Thread.currentThread().getName() + " completed withdrawal. Remaining balance: " + balance);
            }
         else
       {
          System.out.println(Thread.currentThread().getName() + " attempted to withdraw but insufficient funds!");
            }
        }
    }
```

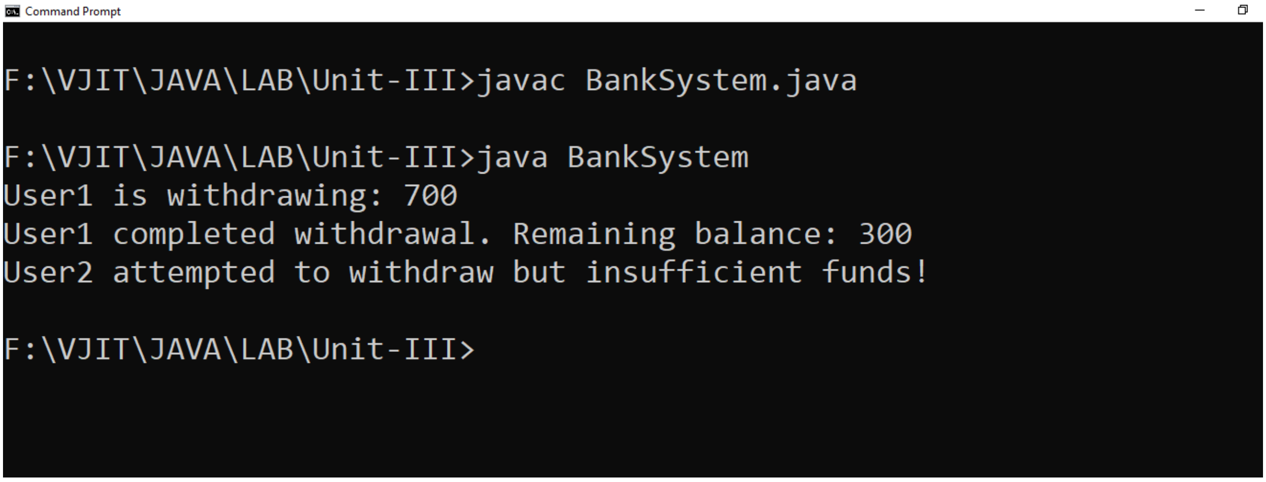
```
public class BankSystem
{
    public static void main(String[] args)
    {
        BankAccount account = new BankAccount();

        // Creating two threads that try to withdraw money
        Thread t1 = new Thread(() -> account.withdraw(700), "User1");
        Thread t2 = new Thread(() -> account.withdraw(700), "User2");

        t1.start();
        t2.start();
    }
}
```

Without Synchronized method



```
F:\VJIT\JAVA\LAB\Unit-III>javac BankSystem.java

F:\VJIT\JAVA\LAB\Unit-III>java BankSystem
User1 is withdrawing: 700
User2 is withdrawing: 700
User1 completed withdrawal. Remaining balance: 300
User2 completed withdrawal. Remaining balance: -400

F:\VJIT\JAVA\LAB\Unit-III>
```

With Synchronized method



```
F:\VJIT\JAVA\LAB\Unit-III>javac BankSystem.java

F:\VJIT\JAVA\LAB\Unit-III>java BankSystem
User1 is withdrawing: 700
User1 completed withdrawal. Remaining balance: 300
User2 attempted to withdraw but insufficient funds!

F:\VJIT\JAVA\LAB\Unit-III>
```

2. Synchronized Block

A **synchronized block** in Java is used to synchronize a specific section of code rather than an entire method. This improves **performance** by allowing other parts of the method to be executed concurrently by multiple threads.

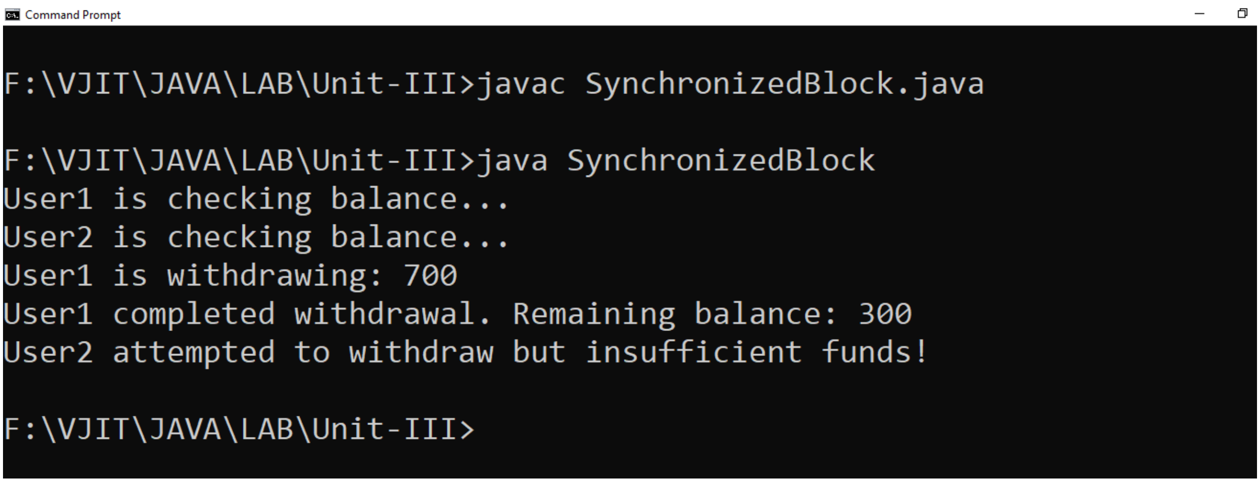
- Instead of locking the **entire method**, a **synchronized block** locks only a specific **critical section**.
- **Improves performance** by allowing non-critical sections to be executed by multiple threads.
- Reduces **thread contention**, as smaller blocks of code are locked.

```
synchronized (lockObject)
 {
   // Critical section - only one thread can execute this at a time
}
```

- lockObject: The object on which the lock is applied.
- The lock ensures that **only one thread at a time** executes the code inside the block.

```
class BankAccount
{
   private int balance = 1000; // Initial balance
   void withdraw(int amount)
   {
      System.out.println(Thread.currentThread().getName() + " is checking balance...");
    // Synchronizing only the critical section
    synchronized (this)
     {
         if (balance >= amount)
       {
System.out.println(Thread.currentThread().getName() + " is withdrawing: " + amount);
            try
             {
               Thread.sleep(500); // Simulating processing time
            }
             catch (InterruptedException e)
             {
               System.out.println(e);
            }
```

```
                balance = balance-amount;
                System.out.println(Thread.currentThread().getName() + " completed withdrawal. Remaining balance: " + balance);
            }
             else
             {
                System.out.println(Thread.currentThread().getName() + " attempted to withdraw but insufficient funds!");
            }
        }
    }
}

public class SynchronizedBlock
{
    public static void main(String[] args)
    {
        BankAccount account = new BankAccount();

        // Creating two threads that try to withdraw money
        Thread t1 = new Thread(() -> account.withdraw(700), "User1");
        Thread t2 = new Thread(() -> account.withdraw(700), "User2");

        t1.start();
        t2.start();
    }
}
```



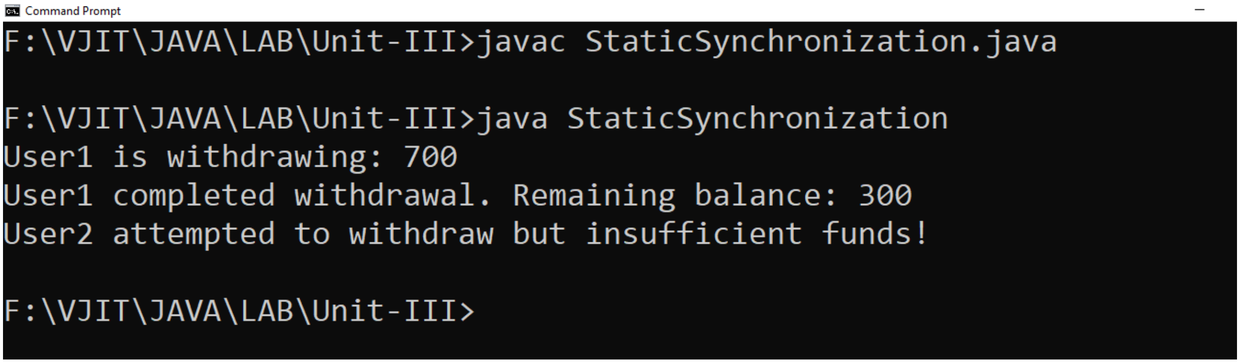
```
F:\VJIT\JAVA\LAB\Unit-III>javac SynchronizedBlock.java

F:\VJIT\JAVA\LAB\Unit-III>java SynchronizedBlock
User1 is checking balance...
User2 is checking balance...
User1 is withdrawing: 700
User1 completed withdrawal. Remaining balance: 300
User2 attempted to withdraw but insufficient funds!

F:\VJIT\JAVA\LAB\Unit-III>
```

3. Static Synchronization

Static synchronization is used to synchronize **static methods** in Java. It ensures that **only one thread at a time** can access a static synchronized method, even if multiple instances of the class exist.

- **When multiple threads access static data** (class-level data), race conditions can occur.
- Static methods belong to the **class, not instances**, so synchronization should be on the **class level** (className.class).
- It prevents **inconsistent data modifications** when shared static resources are used.
- Instead of synchronizing on this (instance-level lock), **static synchronization** locks the **class object**.
- The lock is applied to **ClassName.class**, ensuring only one thread executes any static synchronized method at a time.

```
class BankAccount
{
    private static int balance = 1000; // Shared static balance
// Static Synchronized Method
    static synchronized void withdraw(int amount)
   {
        if (balance >= amount)
       {
            System.out.println(Thread.currentThread().getName() + " is withdrawing: " + amount);
            try
           {
                Thread.sleep(500); // Simulating delay
            }
           catch (InterruptedException e)
          {
                System.out.println(e);
            }
            balance =balance-amount;
            System.out.println(Thread.currentThread().getName() + " completed withdrawal. Remaining balance: " + balance);
        }
        else
        {
            System.out.println(Thread.currentThread().getName() + " attempted to withdraw but insufficient funds!");
        }
    }
}

public class StaticSynchronization
{
    public static void main(String[] args)
    {
        // Multiple users trying to withdraw simultaneously
        Thread t1 = new Thread(() -> BankAccount.withdraw(700), "User1");
        Thread t2 = new Thread(() -> BankAccount.withdraw(700), "User2");

        t1.start();
        t2.start();
    }
}
```



```
F:\VJIT\JAVA\LAB\Unit-III>javac StaticSynchronization.java

F:\VJIT\JAVA\LAB\Unit-III>java StaticSynchronization
User1 is withdrawing: 700
User1 completed withdrawal. Remaining balance: 300
User2 attempted to withdraw but insufficient funds!

F:\VJIT\JAVA\LAB\Unit-III>
```

Inter-Thread Communication

Inter-Thread Communication allows multiple threads to communicate and coordinate their execution using **wait()**, **notify()**, and **notifyAll()** methods in Java.

- **Avoids busy waiting**: Instead of checking conditions continuously, a thread can **wait** and be **notified** when required.
- **Efficient resource usage**: A producer thread can wait when the buffer is full, and a consumer thread can wait when the buffer is empty.
- **Synchronization improvement**: Helps threads coordinate execution by waiting for updates from other threads.

Methods for Inter-Thread Communication

These methods must be used within a **synchronized block/method**:

1. **wait()** → Makes a thread wait until another thread calls notify() or notifyAll().
2. **notify()** → Wakes up a single waiting thread.
3. **notifyAll()** → Wakes up all waiting threads.

Example Program: **Producer-Consumer Problem**

Scenario:

- A **Producer** thread produces items.
- A **Consumer** thread consumes items.
- If the **buffer is full**, the producer **waits**.
- If the **buffer is empty**, the consumer **waits**.

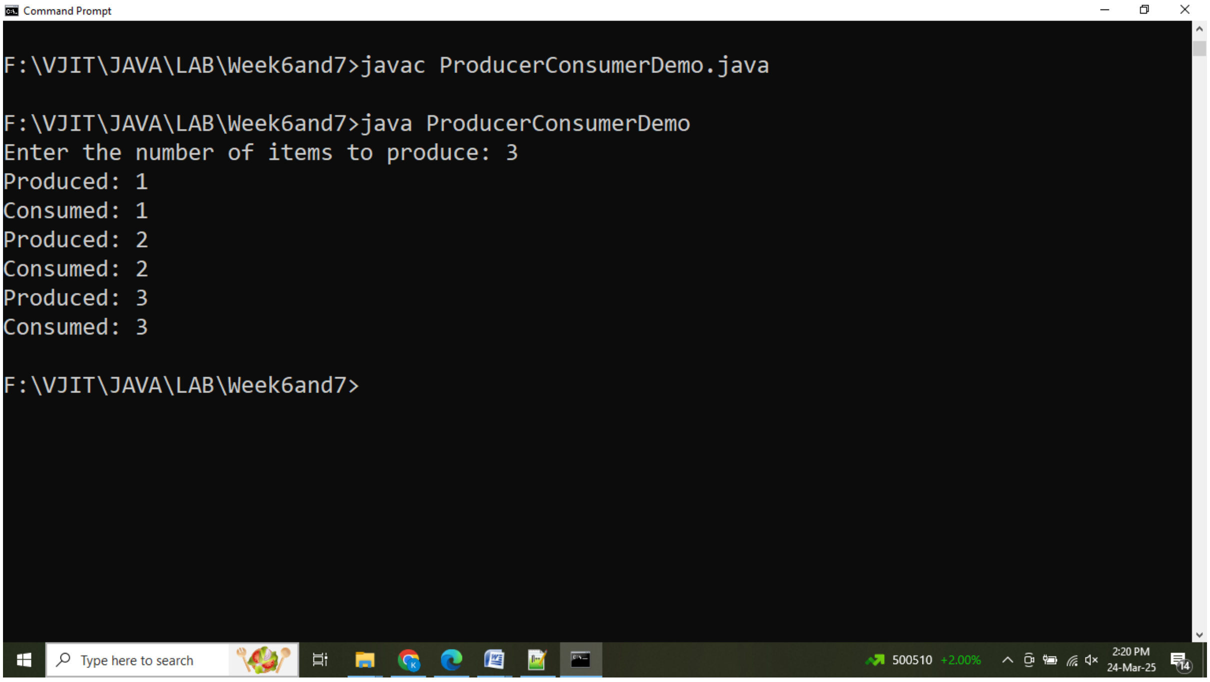
```
import java.util.LinkedList;
import java.util.Scanner;

class SharedBuffer
{
    private LinkedList<Integer> buffer = new LinkedList<>();
    private int capacity;
    // Constructor to set the buffer capacity
    public SharedBuffer(int capacity)
   {
        this.capacity = capacity;
    }
    // Produce an item and add it to the buffer
  public synchronized void produce(int item) throws InterruptedException
  {
        while (buffer.size() == capacity)
       {
            wait(); // Wait if the buffer is full
        }
        buffer.add(item);
        System.out.println("Produced: " + item);
        notify(); // Notify the consumer that an item is available
    }
    // Consume an item from the buffer
    public synchronized void consume() throws InterruptedException
    {
        while (buffer.isEmpty())
       {
            wait(); // Wait if the buffer is empty
        }
        int item = buffer.removeFirst();
        System.out.println("Consumed: " + item);
        notify(); // Notify the producer that space is available
    }
}
```

```
class Producer extends Thread
{
    private SharedBuffer buffer;
    private int itemsToProduce;
    public Producer(SharedBuffer buffer, int itemsToProduce)
   {
        this.buffer = buffer;
        this.itemsToProduce = itemsToProduce;
    }

    @Override
    public void run()
  {
        try
        {
            for (int i = 1; i <= itemsToProduce; i++)
            {
                buffer.produce(i);
                Thread.sleep(100); // Simulating some delay in producing
            }
        }
       catch (InterruptedException e)
       {
            Thread.currentThread().interrupt();
        }
    }
}
class Consumer extends Thread
{
    private SharedBuffer buffer;

    public Consumer(SharedBuffer buffer)
   {
        this.buffer = buffer;
    }
```

```
@Override
   public void run()
   {
       try
          {
          // Continue consuming until the producer has finished producing
          while (true)
           {
              buffer.consume();
              Thread.sleep(100); // Simulating some delay in consuming
          }
       }
      catch (InterruptedException e)
      {
          Thread.currentThread().interrupt();
       }
    }
}

public class ProducerConsumerDemo
{
 public static void main(String[] args)
 {
       Scanner scanner = new Scanner(System.in);
       // Get user input for the number of items to produce
       System.out.print("Enter the number of items to produce: ");
       int itemsToProduce = scanner.nextInt();
       // Create a shared buffer
       SharedBuffer buffer = new SharedBuffer(itemsToProduce);
       // Create producer and consumer threads
       Producer producer = new Producer(buffer, itemsToProduce);
       Consumer consumer = new Consumer(buffer);
       // Start the threads
       producer.start();
       consumer.start();
```

```
            // Wait for producer to finish producing before terminating the consumer
            try
           {
               producer.join();  // Wait for producer to finish
            }
              catch (InterruptedException e)
              {
               Thread.currentThread().interrupt();
            }
          // Interrupt the consumer to gracefully stop the thread after production
            consumer.interrupt();
            // Close the scanner
            scanner.close();
        }
    }
```

Output:



```
F:\VJIT\JAVA\LAB\Week6and7>javac ProducerConsumerDemo.java

F:\VJIT\JAVA\LAB\Week6and7>java ProducerConsumerDemo
Enter the number of items to produce: 3
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3

F:\VJIT\JAVA\LAB\Week6and7>
```

Java.util Package- Collection Interfaces & Collection Classes PART-3

Collection Interfaces: *List, Map, Set.*

Collection Classes: *LinkedList, HashMap, TreeSet, StringTokenizer, Date, Random, Scanner.*

The **Java Collections Framework (JCF)** provides a standardized way to store, retrieve, and manipulate collections of objects.

The three key **collection interfaces** are:

1. **List** → Ordered collection, allows duplicates.
2. **Set** → Unordered collection, no duplicates.
3. **Map** → Stores key-value pairs, keys must be unique.

1. List Interface (java.util.List)

The List interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements.

public interface List<E> extends Collection<E>;

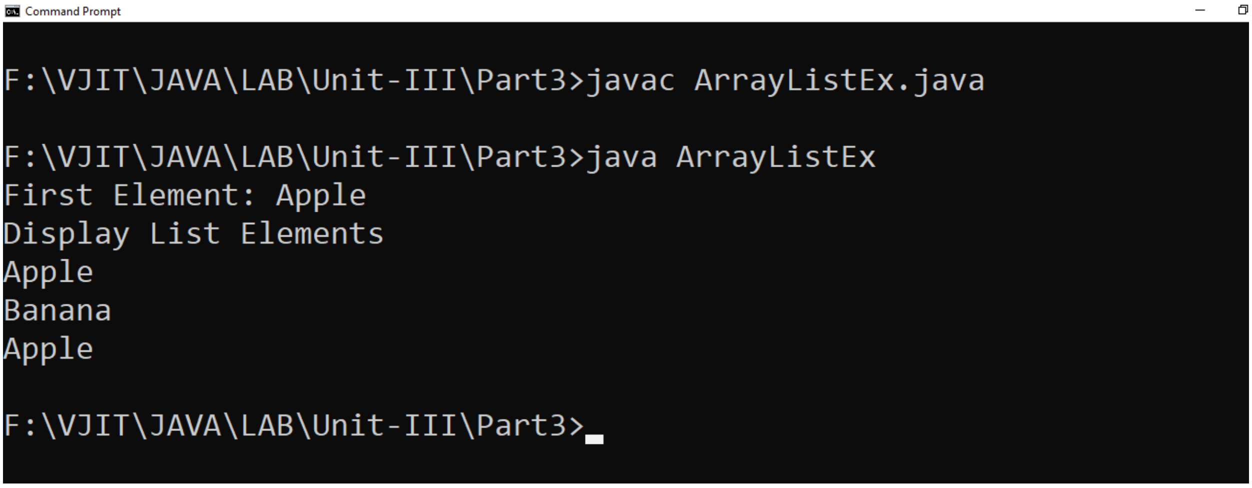
- Allows duplicate elements
- Maintains insertion order
- Provides index-based access
- Allows null values

Implementations of List:

Implementation	Description
1. ArrayList	**Fast random access**, slow insert/delete
2. LinkedList	**Fast insert/delete**, slow random access
3. Vector	**Thread-safe**, legacy class
4. Stack	LIFO (Last-In-First-Out) structure

1. ArrayList

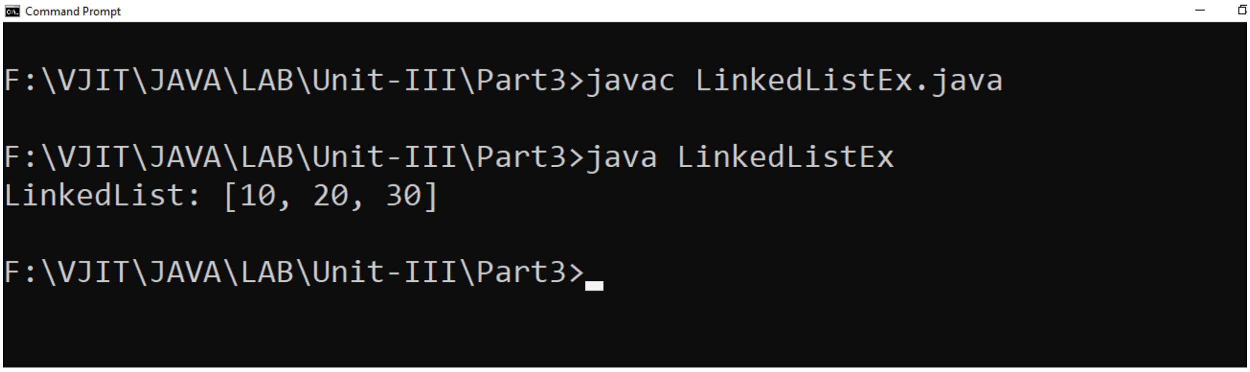
```
import java.util.*;

public class ArrayListEx
{
    public static void main(String[] args)
   {
        List<String> list = new ArrayList<>();

        // Adding elements
        list.add("Apple");
        list.add("Banana");
        list.add("Apple"); // Duplicates allowed

        // Accessing elements
        System.out.println("First Element: " + list.get(0));
        System.out.println("Display List Elements ");
        // Iterating over elements
        for (String fruit : list)
       {
            System.out.println(fruit);
        }
    }
}
```



```
F:\VJIT\JAVA\LAB\Unit-III\Part3>javac ArrayListEx.java

F:\VJIT\JAVA\LAB\Unit-III\Part3>java ArrayListEx
First Element: Apple
Display List Elements
Apple
Banana
Apple

F:\VJIT\JAVA\LAB\Unit-III\Part3>
```

2. LinkedList

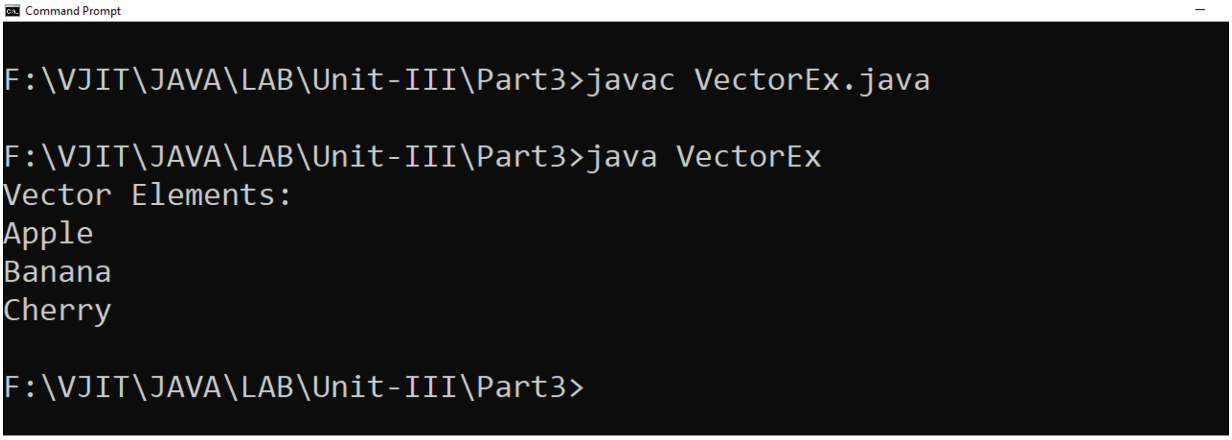
```
import java.util.*;

public class LinkedListEx
{
  public static void main(String[] args)
  {
    List<Integer> numbers = new LinkedList<>();
    numbers.add(10);
    numbers.add(20);
    numbers.add(30);

    //numbers.remove(1); // Removes element at index 1

    System.out.println("LinkedList: " + numbers);
  }
}
```



```
F:\VJIT\JAVA\LAB\Unit-III\Part3>javac LinkedListEx.java

F:\VJIT\JAVA\LAB\Unit-III\Part3>java LinkedListEx
LinkedList: [10, 20, 30]

F:\VJIT\JAVA\LAB\Unit-III\Part3>_
```

3. Vector

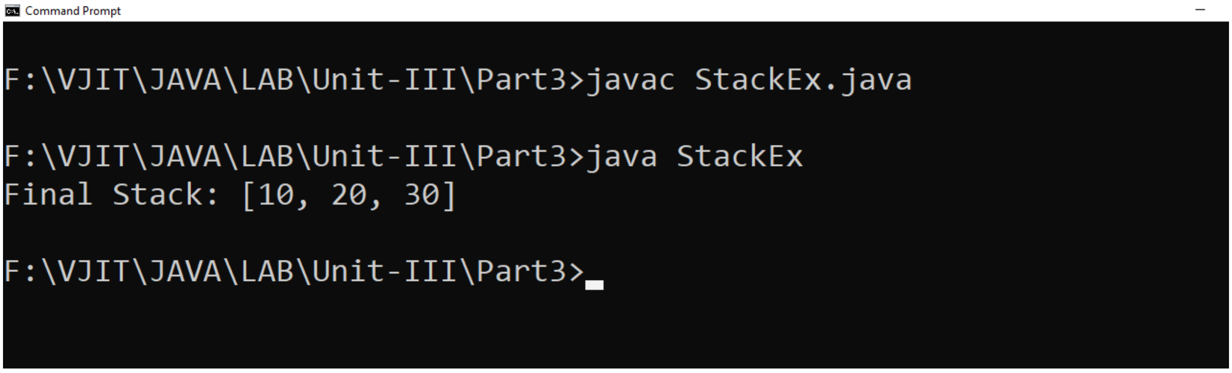
```
import java.util.*;

public class VectorEx
{
   public static void main(String[] args)
   {
      Vector<String> vector = new Vector<>();

      // Adding elements
      vector.add("Apple");
      vector.add("Banana");
      vector.add("Cherry");

      // Accessing elements using index
      //System.out.println("First Element: " + vector.get(0));

      // Removing an element
      //vector.remove("Banana");
      // Iterating over elements
      System.out.println("Vector Elements:");
      for (String fruit : vector)
     {
         System.out.println(fruit);
      }
   }
}
```



```
F:\VJIT\JAVA\LAB\Unit-III\Part3>javac VectorEx.java

F:\VJIT\JAVA\LAB\Unit-III\Part3>java VectorEx
Vector Elements:
Apple
Banana
Cherry

F:\VJIT\JAVA\LAB\Unit-III\Part3>
```

4. Stack

```
import java.util.*;

public class StackEx
{
   public static void main(String[] args)
  {
       Stack<Integer> stack = new Stack<>();

       // Pushing elements onto the stack
       stack.push(10);
       stack.push(20);
       stack.push(30);

       // Peeking (viewing top element)
       //System.out.println("Top Element: " + stack.peek());

       // Popping (removing) elements
       //System.out.println("Popped: " + stack.pop());

       // Checking if stack is empty
      // System.out.println("Is Stack Empty? " + stack.isEmpty());

       // Printing final stack
       System.out.println("Final Stack: " + stack);
   }
}
```



```
F:\VJIT\JAVA\LAB\Unit-III\Part3>javac StackEx.java

F:\VJIT\JAVA\LAB\Unit-III\Part3>java StackEx
Final Stack: [10, 20, 30]

F:\VJIT\JAVA\LAB\Unit-III\Part3>
```

2. Set Interface (java.util.Set)

A **Set** is an **unordered collection** that:

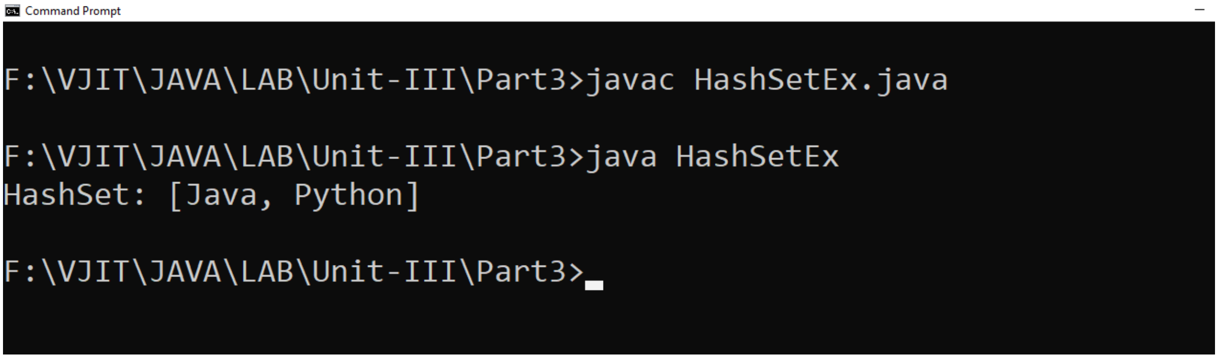
- Does not allow duplicate elements
- Does not guarantee order (except TreeSet)
- Allows null (only once)

Implementations of Set:

Implementation	Description
1. HashSet	**Fastest**, unordered
2. TreeSet	**Sorted order**, slower than HashSet
3. LinkedHashSet	**Maintains insertion order**

1. HashSet

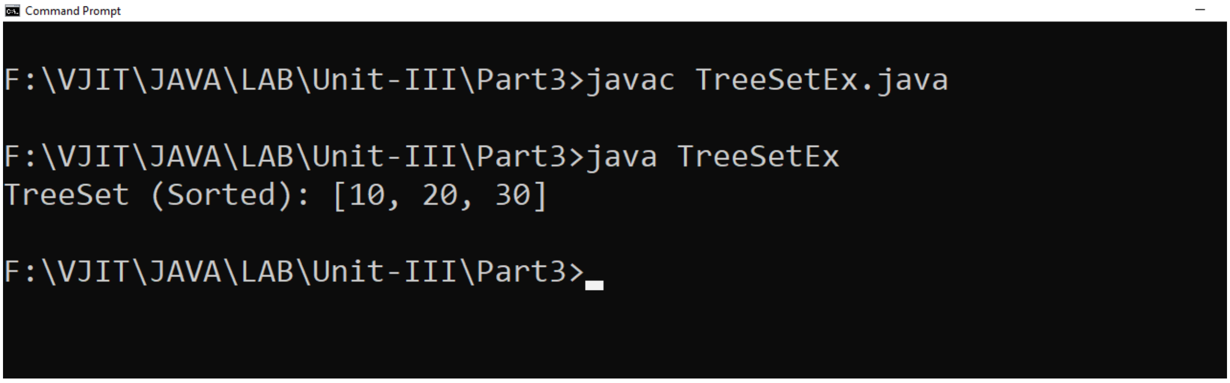
```
import java.util.*;

public class HashSetEx
{
    public static void main(String[] args)
   {
        Set<String> set = new HashSet<>();

        // Adding elements
        set.add("Java");
        set.add("Python");
        set.add("Java"); // Ignored (No duplicates allowed)

        System.out.println("HashSet: " + set);
    }
}
```



```
F:\VJIT\JAVA\LAB\Unit-III\Part3>javac HashSetEx.java

F:\VJIT\JAVA\LAB\Unit-III\Part3>java HashSetEx
HashSet: [Java, Python]

F:\VJIT\JAVA\LAB\Unit-III\Part3>
```

2. TreeSet

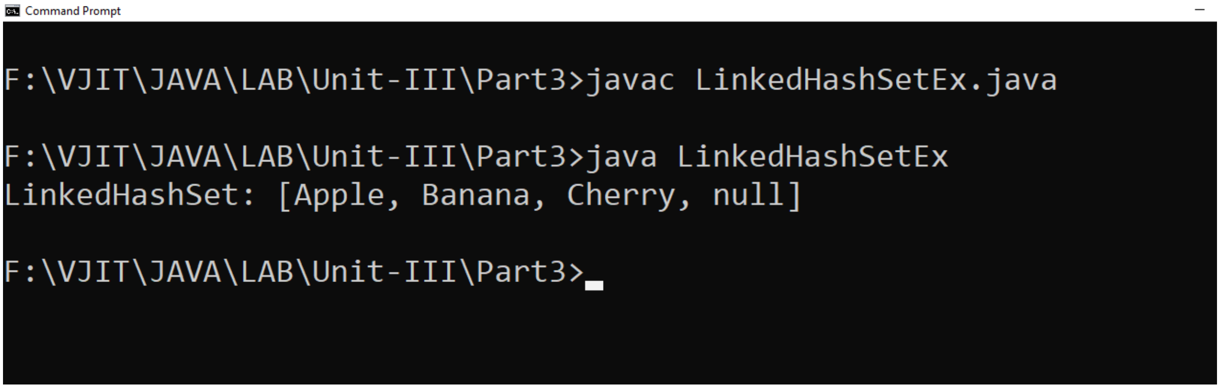
```
import java.util.*;

public class TreeSetEx
{
   public static void main(String[] args)
  {
      Set<Integer> set = new TreeSet<>();
      set.add(30);
      set.add(10);
      set.add(20);

      System.out.println("TreeSet (Sorted): " + set);
   }
}
```



```
F:\VJIT\JAVA\LAB\Unit-III\Part3>javac TreeSetEx.java

F:\VJIT\JAVA\LAB\Unit-III\Part3>java TreeSetEx
TreeSet (Sorted): [10, 20, 30]

F:\VJIT\JAVA\LAB\Unit-III\Part3>
```

3. LinkedHashSet

```
import java.util.*;

public class LinkedHashSetEx
{
    public static void main(String[] args)
   {
        // Creating a LinkedHashSet
        LinkedHashSet<String> set = new LinkedHashSet<>();

        // Adding elements
        set.add("Apple");
        set.add("Banana");
        set.add("Cherry");
        set.add("Apple"); // Duplicate, will be ignored
        set.add(null); // Allows one null value
        //set.add(null);
        // Printing the elements
        System.out.println("LinkedHashSet: " + set);


    }
}
```



```
F:\VJIT\JAVA\LAB\Unit-III\Part3>javac LinkedHashSetEx.java

F:\VJIT\JAVA\LAB\Unit-III\Part3>java LinkedHashSetEx
LinkedHashSet: [Apple, Banana, Cherry, null]

F:\VJIT\JAVA\LAB\Unit-III\Part3>
```

3. Map Interface (java.util.Map)

A **Map** is a **Key-Value Pair Collection** that:

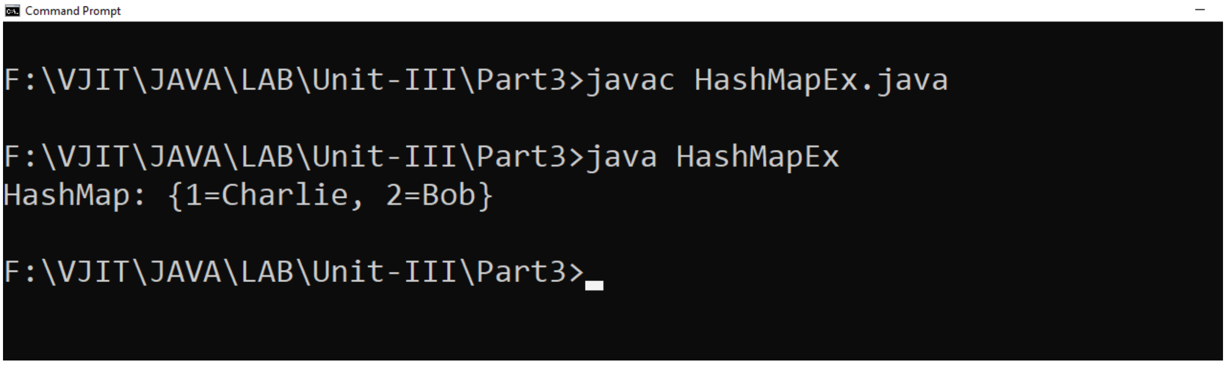
- Does not allow duplicate keys
- Keys must be unique, but values can repeat
- Allows null key (only in HashMap)

Implementations of Map:

Implementation	Description
1. HashMap	**Fastest**, unordered
2. TreeMap	**Sorted by keys**
3. LinkedHashMap	**Maintains insertion order**

1. HashMap

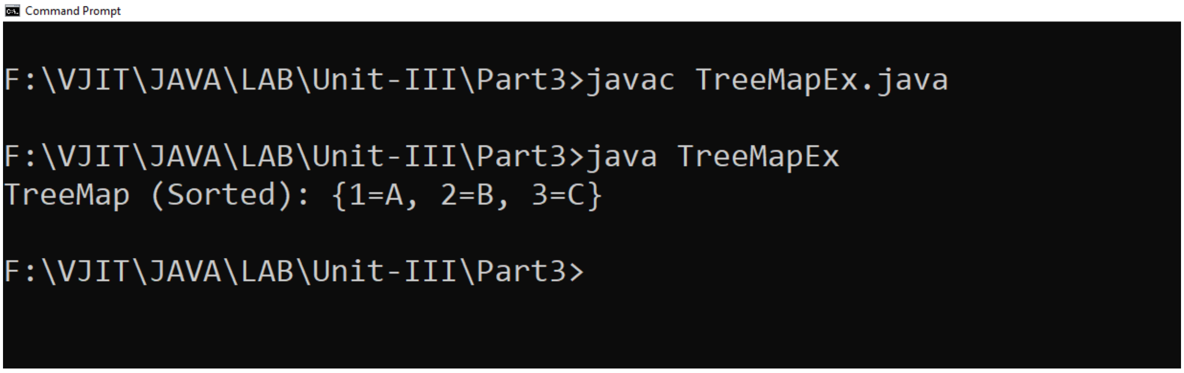
```
import java.util.*;

public class HashMapEx
{
   public static void main(String[] args)
  {
      Map<Integer, String> map = new HashMap<>();

      // Adding key-value pairs
      map.put(1, "Alice");
      map.put(2, "Bob");
      map.put(1, "Charlie"); // Overwrites previous value for key 1

      System.out.println("HashMap: " + map);
   }
}
```



```
F:\VJIT\JAVA\LAB\Unit-III\Part3>javac HashMapEx.java

F:\VJIT\JAVA\LAB\Unit-III\Part3>java HashMapEx
HashMap: {1=Charlie, 2=Bob}

F:\VJIT\JAVA\LAB\Unit-III\Part3>_
```

2. TreeMap

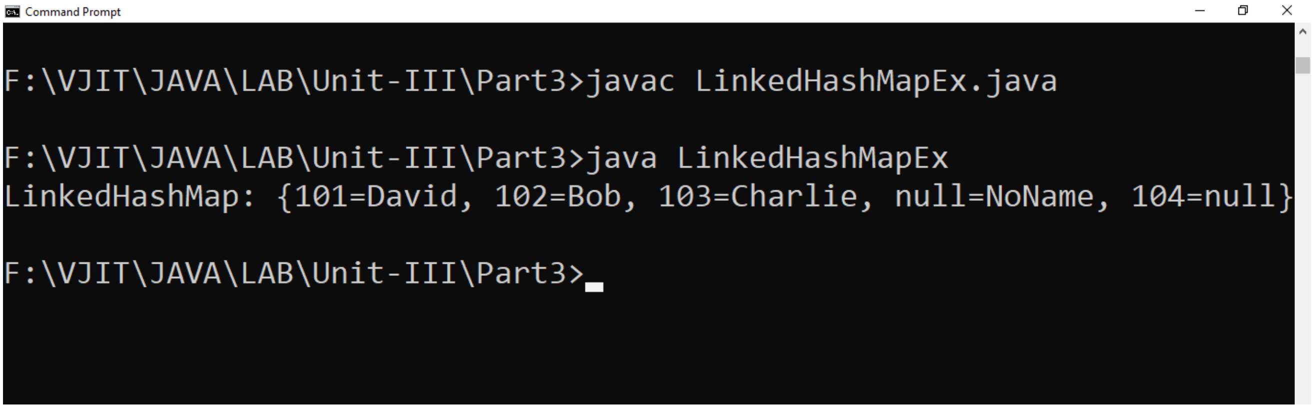
```
import java.util.*;

public class TreeMapEx
{
    public static void main(String[] args)
   {
        Map<Integer, String> map = new TreeMap<>();
        map.put(3, "C");
        map.put(1, "A");
        map.put(2, "B");

        System.out.println("TreeMap (Sorted): " + map);
    }
}
```



```
F:\VJIT\JAVA\LAB\Unit-III\Part3>javac TreeMapEx.java

F:\VJIT\JAVA\LAB\Unit-III\Part3>java TreeMapEx
TreeMap (Sorted): {1=A, 2=B, 3=C}

F:\VJIT\JAVA\LAB\Unit-III\Part3>
```

3. LinkedHashMap

```
import java.util.*;

public class LinkedHashMapEx
{
    public static void main(String[] args)
   {
        // Creating a LinkedHashMap
        LinkedHashMap<Integer, String> map = new LinkedHashMap<>();

        // Adding key-value pairs
        map.put(101, "Alice");
        map.put(102, "Bob");
        map.put(103, "Charlie");
        map.put(101, "David"); // Overwrites value for key 101
        map.put(null, "NoName"); // Allows one null key
        map.put(104, null); // Allows null values

        // Printing the LinkedHashMap
        System.out.println("LinkedHashMap: " + map);


    }
}
```



```
F:\VJIT\JAVA\LAB\Unit-III\Part3>javac LinkedHashMapEx.java

F:\VJIT\JAVA\LAB\Unit-III\Part3>java LinkedHashMapEx
LinkedHashMap: {101=David, 102=Bob, 103=Charlie, null=NoName, 104=null}

F:\VJIT\JAVA\LAB\Unit-III\Part3>
```

Comparison Table

Feature	List	Set	Map
1. Allows Duplicates	Yes	No	Keys - No Values - Yes
2. Maintains Insertion Order	Yes	No (LinkedHashSet)	Yes (LinkedHashMap)
3. Access via Index	Yes	No	No
4. Allows Null	Yes	One Null value	Yes (One Null Key)
5. Sorted Order	No	Yes (TreeSet)	Yes (TreeMap)

Collection Classes in Java

Java provides several **Collection Classes** in the java.util package that help in storing, managing, and processing data efficiently. These classes implement interfaces such as **List, Set, Queue, and Map**, offering various functionalities like ordering, uniqueness, and sorting.

1. StringTokenizer Class

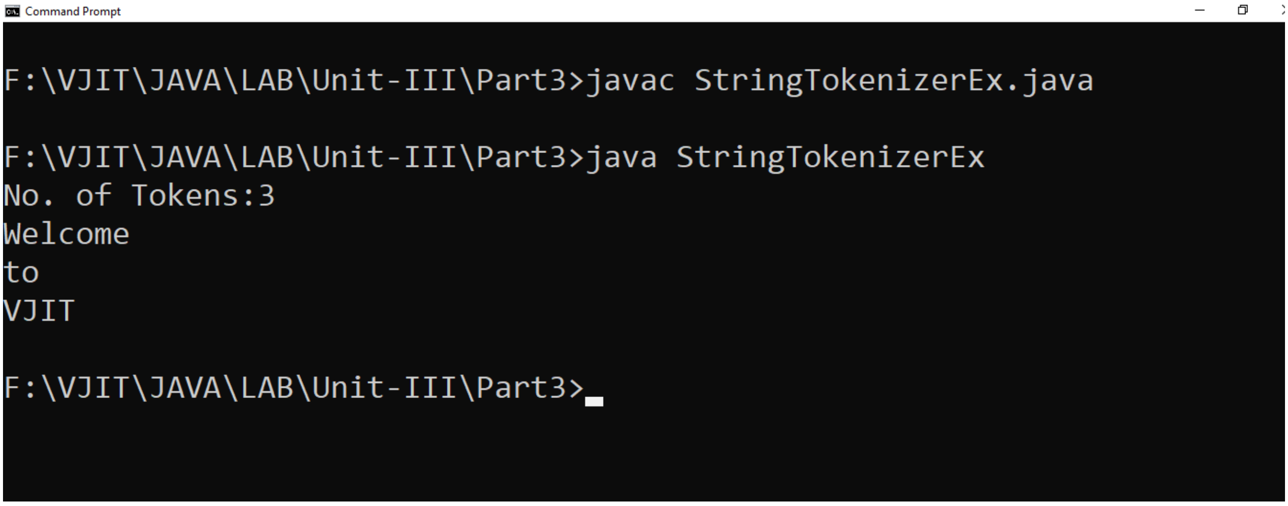
- Used to **break a string into tokens**
- **Faster than split() method**

Constructors of the StringTokenizer Class

Constructor	Description
StringTokenizer(String str)	It creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	It creates StringTokenizer with specified string and delimiter.

Methods	Description
boolean hasMoreTokens()	It checks if there is more tokens available.
String nextToken()	It returns the next token from the StringTokenizer object.
String nextToken(String delim)	It returns the next token based on the delimiter.
int countTokens()	It returns the total number of tokens.

```
import java.util.*;

public class StringTokenizerEx
{
   public static void main(String[] args)
  {
      String str = "Welcome to VJIT";
      StringTokenizer st = new StringTokenizer(str, " ");
      System.out.println("No. of Tokens:"+st.countTokens());
      // Iterating through tokens
      while (st.hasMoreTokens()) {
         System.out.println(st.nextToken());
      }
   }
}
```



```
F:\VJIT\JAVA\LAB\Unit-III\Part3>javac StringTokenizerEx.java

F:\VJIT\JAVA\LAB\Unit-III\Part3>java StringTokenizerEx
No. of Tokens:3
Welcome
to
VJIT

F:\VJIT\JAVA\LAB\Unit-III\Part3>
```

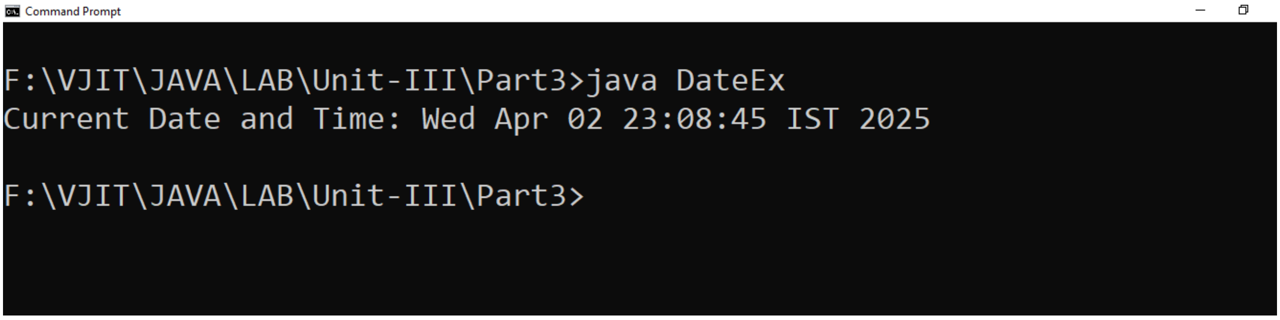
2. Date Class

Date (Working with Dates)

- Represents **date and time**
- The java.util.Date is a class that is inside java.util package.

No.	Constructor	Description
1.	Date()	It is used to create a date object to represent the current date and time.
2.	Date(long milliseconds)	It is used to create a date object for the given milliseconds since January 1, 1970, 00:00:00 GMT.

```
import java.util.Date;

public class DateEx
{
    public static void main(String[] args)
   {
        // Creating a Date object representing the current date and time
        Date dt = new Date();
        System.out.println("Current Date and Time: " + dt);
    }
}
```



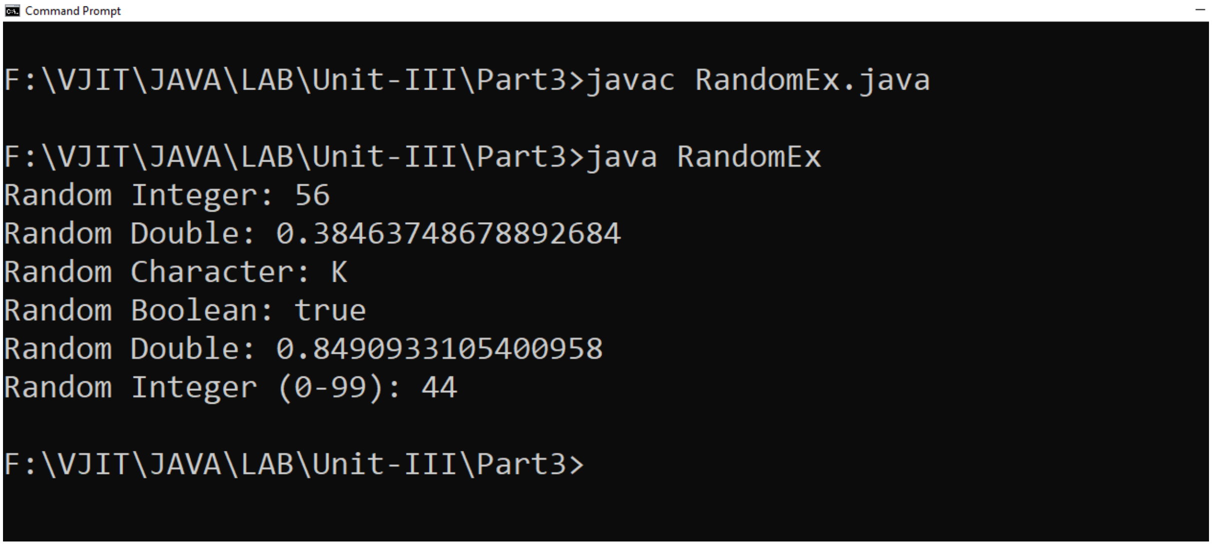
```
F:\VJIT\JAVA\LAB\Unit-III\Part3>java DateEx
Current Date and Time: Wed Apr 02 23:08:45 IST 2025

F:\VJIT\JAVA\LAB\Unit-III\Part3>
```

3. Random Class

The **Random** class in Java is part of the java.util package and is used to **generate random numbers** of different data types such as int, double, float, long, and boolean.

It is commonly used in **games, simulations, cryptography, and random sampling**.

```
import java.util.*;

public class RandomEx
{
    public static void main(String[] args)
   {
        Random random = new Random();

        // Generating random numbers
        System.out.println("Random Integer: " + random.nextInt(100));
        System.out.println("Random Double: " + random.nextDouble());

        // Generates a random uppercase letter
        char randomChar = (char) ('A' + random.nextInt(26));
        System.out.println("Random Character: " + randomChar);

        // Generates a random Boolean Value
        boolean randomBoolean = random.nextBoolean();
        System.out.println("Random Boolean: " + randomBoolean);

            // Using Math.random() method
       double randomDouble = Math.random(); // Generates between 0.0 and 1.0
        int randomInt = (int) (Math.random() * 100); // 0 to 99

        System.out.println("Random Double: " + randomDouble);
        System.out.println("Random Integer (0-99): " + randomInt);

          }
}
```



```
F:\VJIT\JAVA\LAB\Unit-III\Part3>javac RandomEx.java

F:\VJIT\JAVA\LAB\Unit-III\Part3>java RandomEx
Random Integer: 56
Random Double: 0.38463748678892684
Random Character: K
Random Boolean: true
Random Double: 0.8490933105400958
Random Integer (0-99): 44

F:\VJIT\JAVA\LAB\Unit-III\Part3>
```

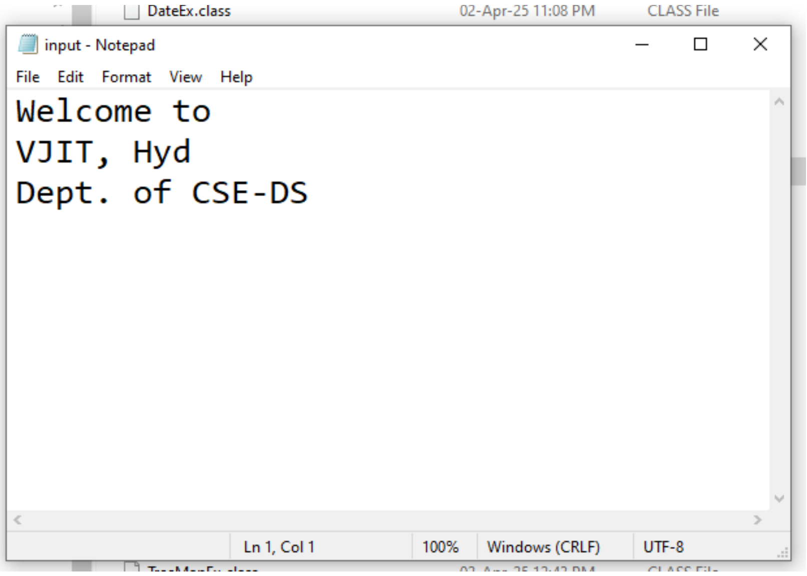
4. Scanner Class

The **Scanner** class in Java is part of the java.util package and is used to **take input from various sources**, such as:

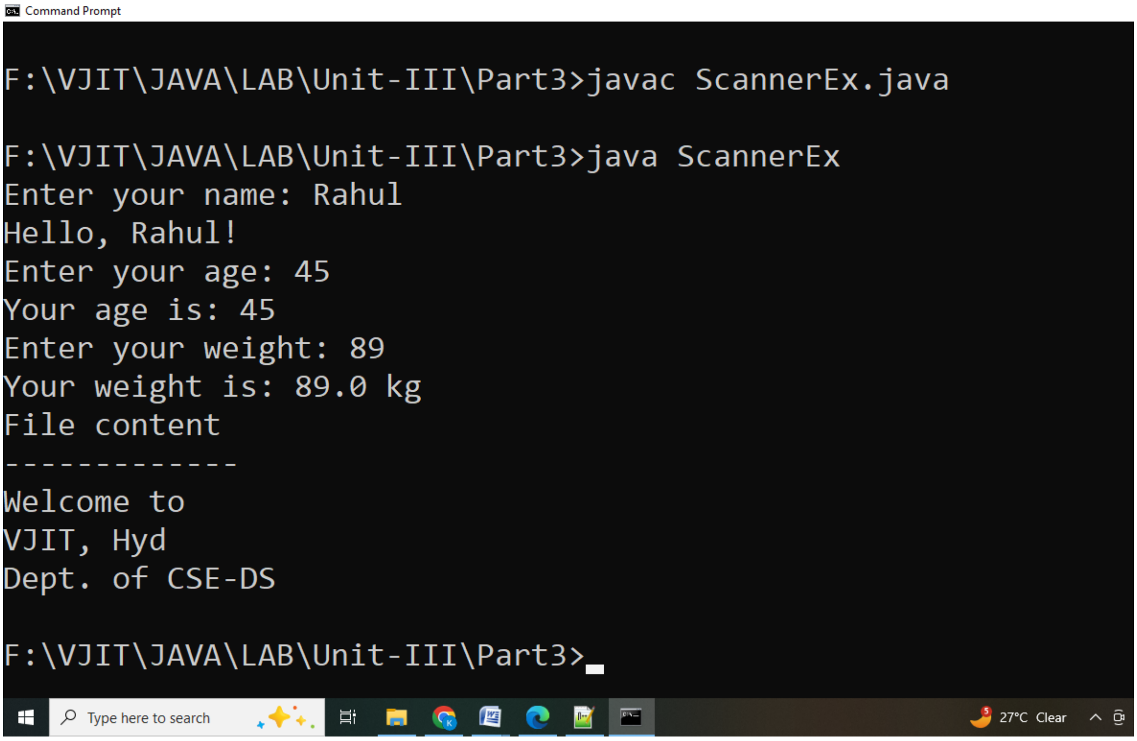
- **Keyboard (User Input)**
- **Files**
- **Strings**
- **Streams (System input/output)**

It can read **integers, floats, strings, characters, and more**.

```
import java.util.*;
import java.io.File;
import java.io.FileNotFoundException;
public class ScannerEx
{
   public static void main(String[] args)
   {
      Scanner sc = new Scanner(System.in);
      // Reading input
      System.out.print("Enter your name: ");
      String name = sc.nextLine();
      System.out.println("Hello, " + name + "!");
      System.out.print("Enter your age: ");
       int age = sc.nextInt(); // Reads an integer
       System.out.println("Your age is: " + age);

       System.out.print("Enter your weight: ");
       float weight = sc.nextFloat(); // Reads a float
       System.out.println("Your weight is: " + weight + " kg");

        System.out.println("File content");
        System.out.println("-------------");
         //file reading
         try
         {
           // Create a File object
           File file = new File("input.txt");

           // Create a Scanner object to read the file
           Scanner s = new Scanner(file);

           // Read file line by line
           while (s.hasNextLine()) {
               String line = s.nextLine();
               System.out.println(line);
           }

           // Close the Scanner
           s.close();
       }
       catch (FileNotFoundException e)
      {
           System.out.println("File not found. Please check the file path.");
       }
         sc.close();
    }
}
```


DateEx.class 02-Apr-25 11:08 PM CLASS File
input - Notepad
File Edit Format View Help
Ln 1, Col 1 100% Windows (CRLF) UTF-8

```
Welcome to
VJIT, Hyd
Dept. of CSE-DS
```



```
F:\VJIT\JAVA\LAB\Unit-III\Part3>javac ScannerEx.java

F:\VJIT\JAVA\LAB\Unit-III\Part3>java ScannerEx
Enter your name: Rahul
Hello, Rahul!
Enter your age: 45
Your age is: 45
Enter your weight: 89
Your weight is: 89.0 kg
File content
-------------
Welcome to
VJIT, Hyd
Dept. of CSE-DS

F:\VJIT\JAVA\LAB\Unit-III\Part3>_
```