MULTITHREADING

MULTITHREADING

- Multithreading in Java is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

MULTITHREADING

- **Multitasking**

- Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

  - Process-based Multitasking (Multiprocessing)
  - Thread-based Multitasking (Multithreading)

MULTITHREADING

- **1) Process-based Multitasking (Multiprocessing)**

- 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.

MULTITHREADING

- **2) Thread-based Multitasking (Multithreading)**

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

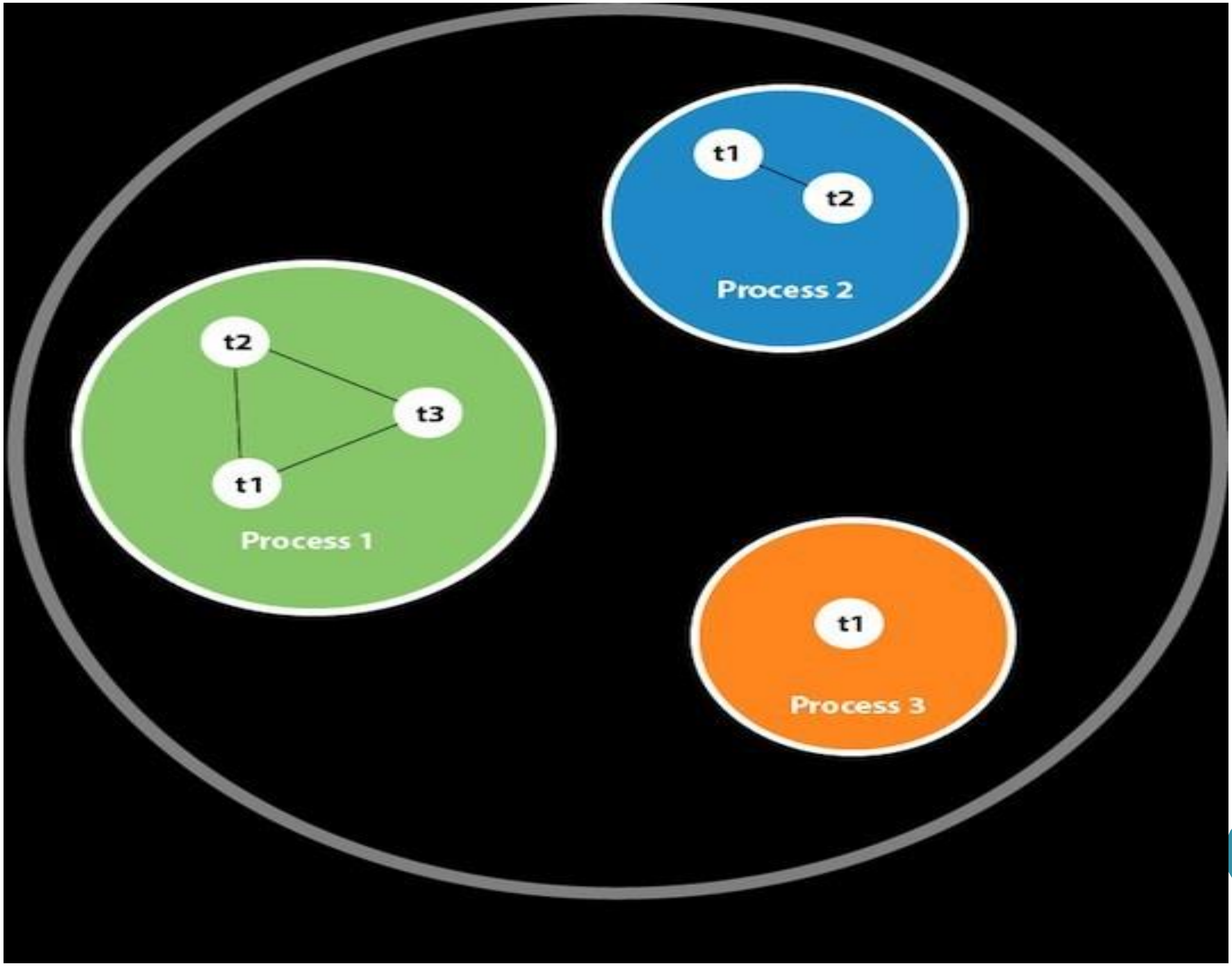
MULTITHREADING

- **What is Thread in java?**

- A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

- Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

MULTITHREADING



MULTITHREADING

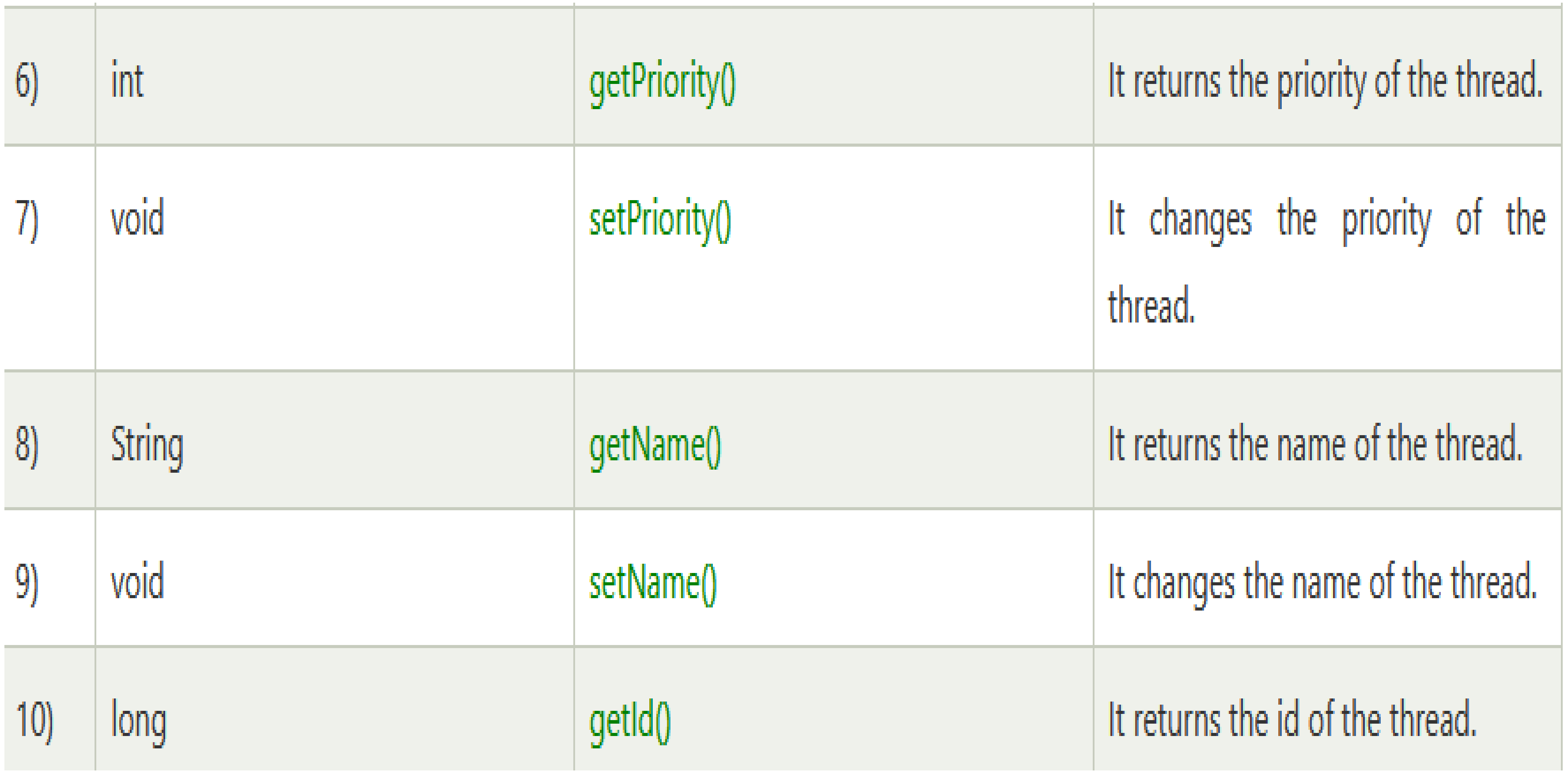
- **Java Thread class**
- Java provides Thread class to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread.

MULTITHREADING

- **Java Thread Methods**

1)	void	start()	It is used to start the execution of the thread.
2)	void	run()	It is used to do an action for a thread.
3)	static void	sleep()	It sleeps a thread for the specified amount of time.
4)	static Thread	currentThread()	It returns a reference to the currently executing thread object.
5)	void	join()	It waits for a thread to die.

MULTITHREADING



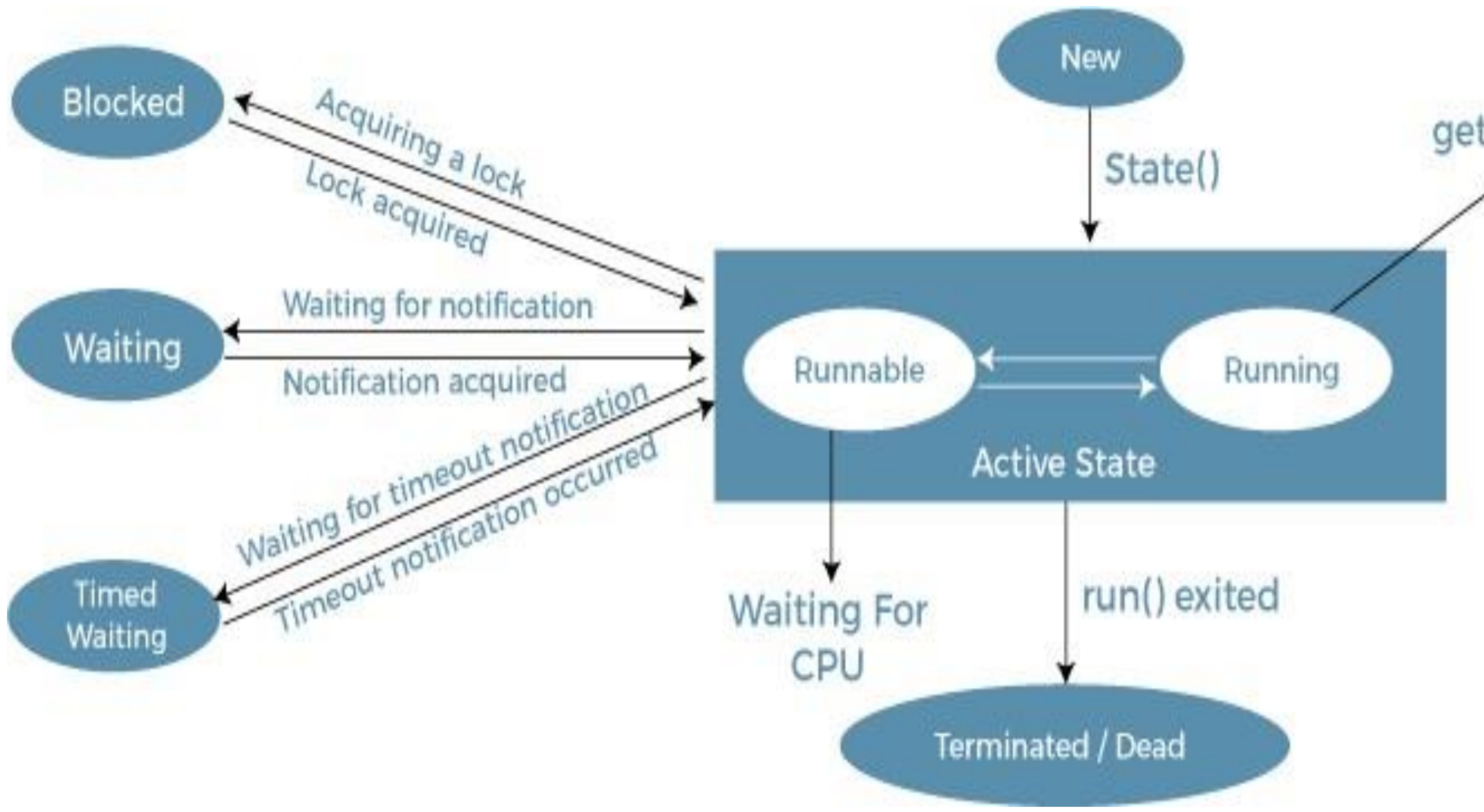
6)	int	getPriority()	It returns the priority of the thread.
7)	void	setPriority()	It changes the priority of the thread.
8)	String	getName()	It returns the name of the thread.
9)	void	setName()	It changes the name of the thread.
10)	long	getId()	It returns the id of the thread.

MULTITHREADING

- **Life cycle of a Thread (Thread States)**
- In Java, a thread always exists in any one of the following states. These states are:

- New
- Active
- Blocked / Waiting
- Timed Waiting
- Terminated

MULTITHREADING



MULTITHREADING

- **New:** Whenever a new thread is created, it is always in the new state.
- **Active:** When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is runnable, and the other is running.
- **Runnable:** A thread, that is ready to run is then moved to the runnable state.
- **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

MULTITHREADING

- **Blocked or Waiting:** Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.
- **Timed Waiting:** Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation.
- **Terminated:** When a thread has finished its job, then it exists or terminates normally.

MULTITHREADING

- **How to create a thread**
- There are two ways to create a thread:
  - By extending Thread class
  - By implementing Runnable interface

MULTITHREADING

- **Thread class:**

- Thread class provide constructors and methods to create and perform operations on a thread.
- **Commonly used methods of Thread class:**
- public void run(): is used to perform action for a thread.
- public void start(): starts the execution of the thread.JVM calls the run() method on the thread.
- public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

MULTITHREADING

- public void join(): waits for a thread to die.
- public void join(long miliseconds): waits for a thread to die for the specified miliseconds.
- public int getPriority(): returns the priority of the thread.
- public int setPriority(int priority): changes the priority of the thread.
- public String getName(): returns the name of the thread.
- public int getId(): returns the id of the thread.
- public Thread.State getState(): returns the state of the thread.
- public boolean isAlive(): tests if the thread is alive.

MULTITHREADING

- **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().

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

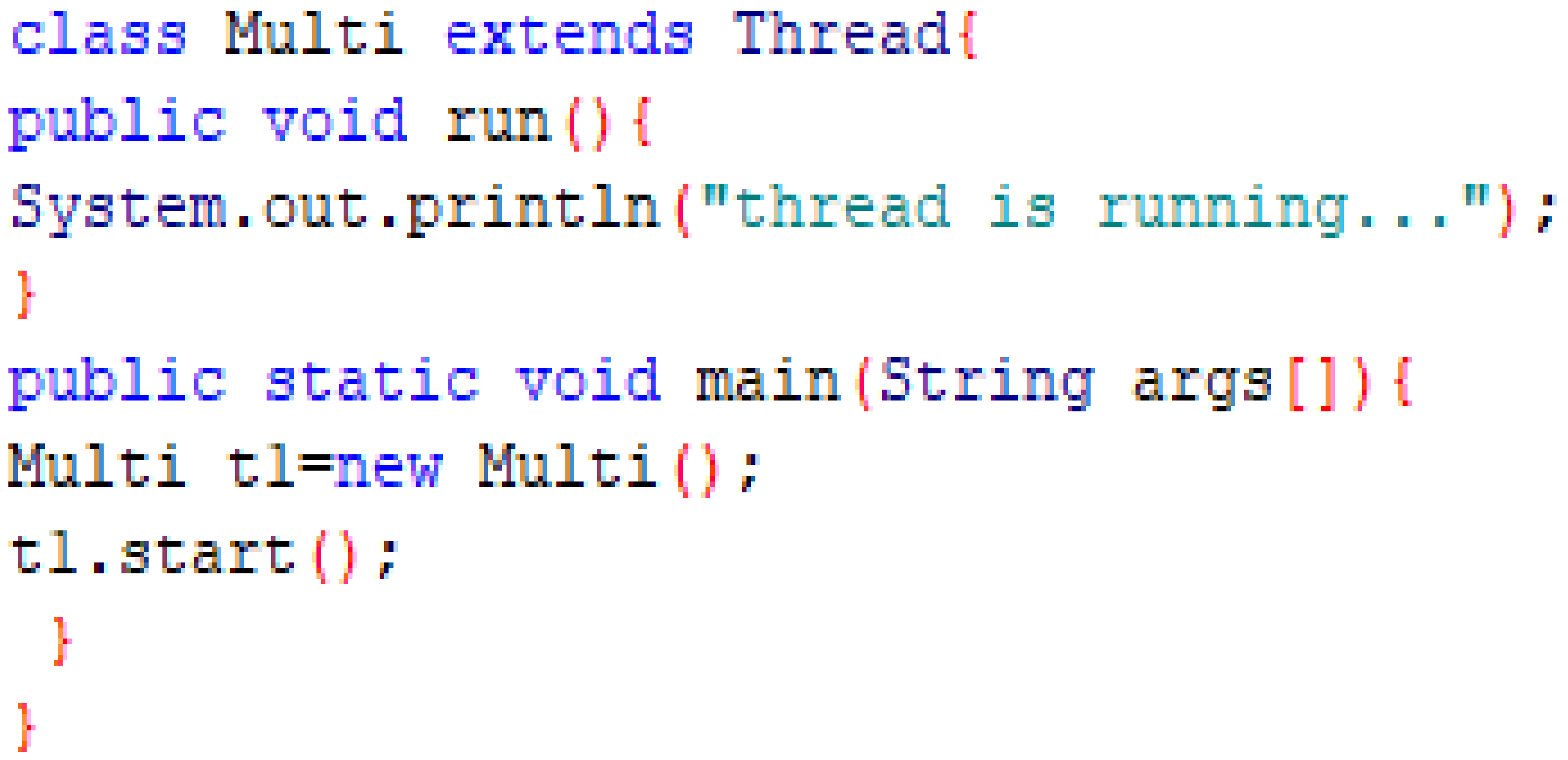
MULTITHREADING

- **Starting a thread:**

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

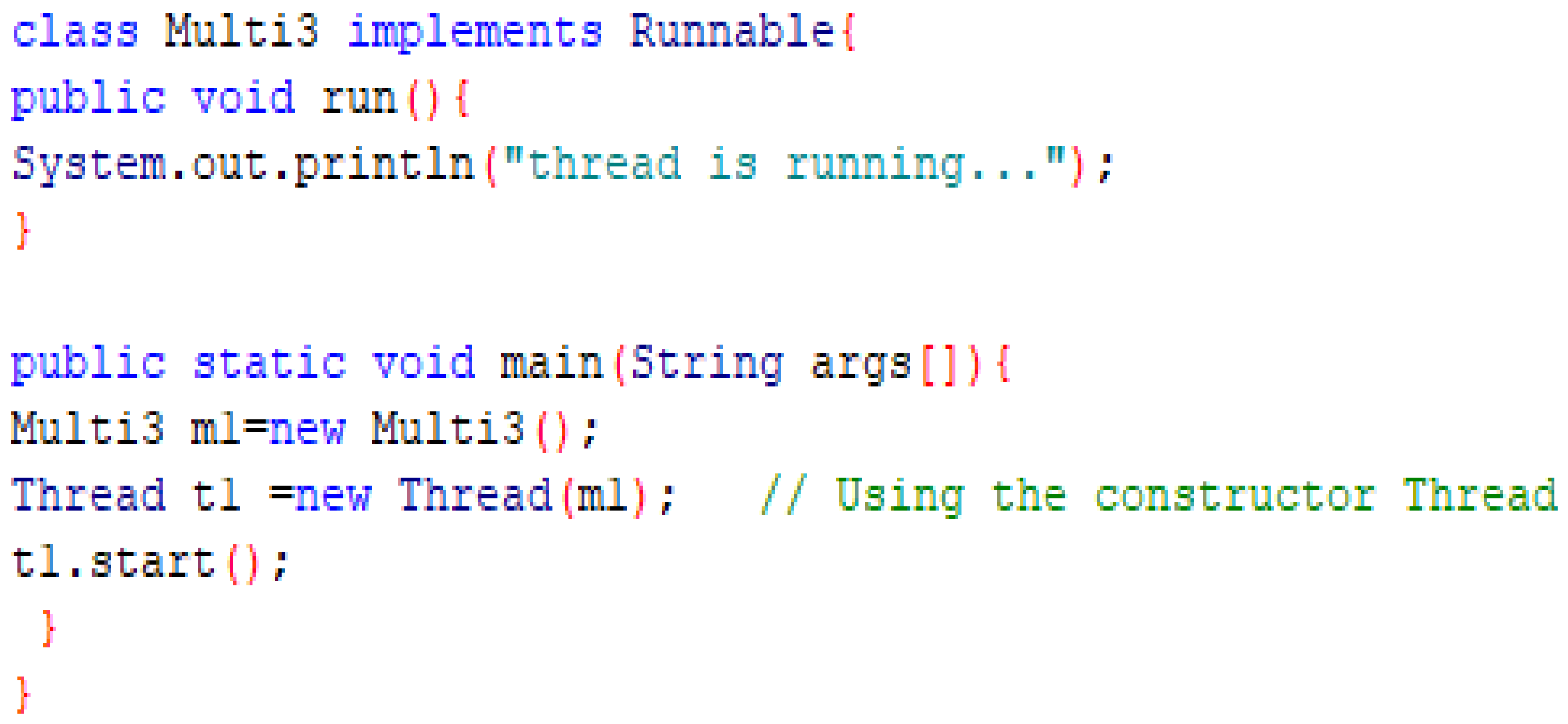
- A new thread starts.
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

MULTITHREADING



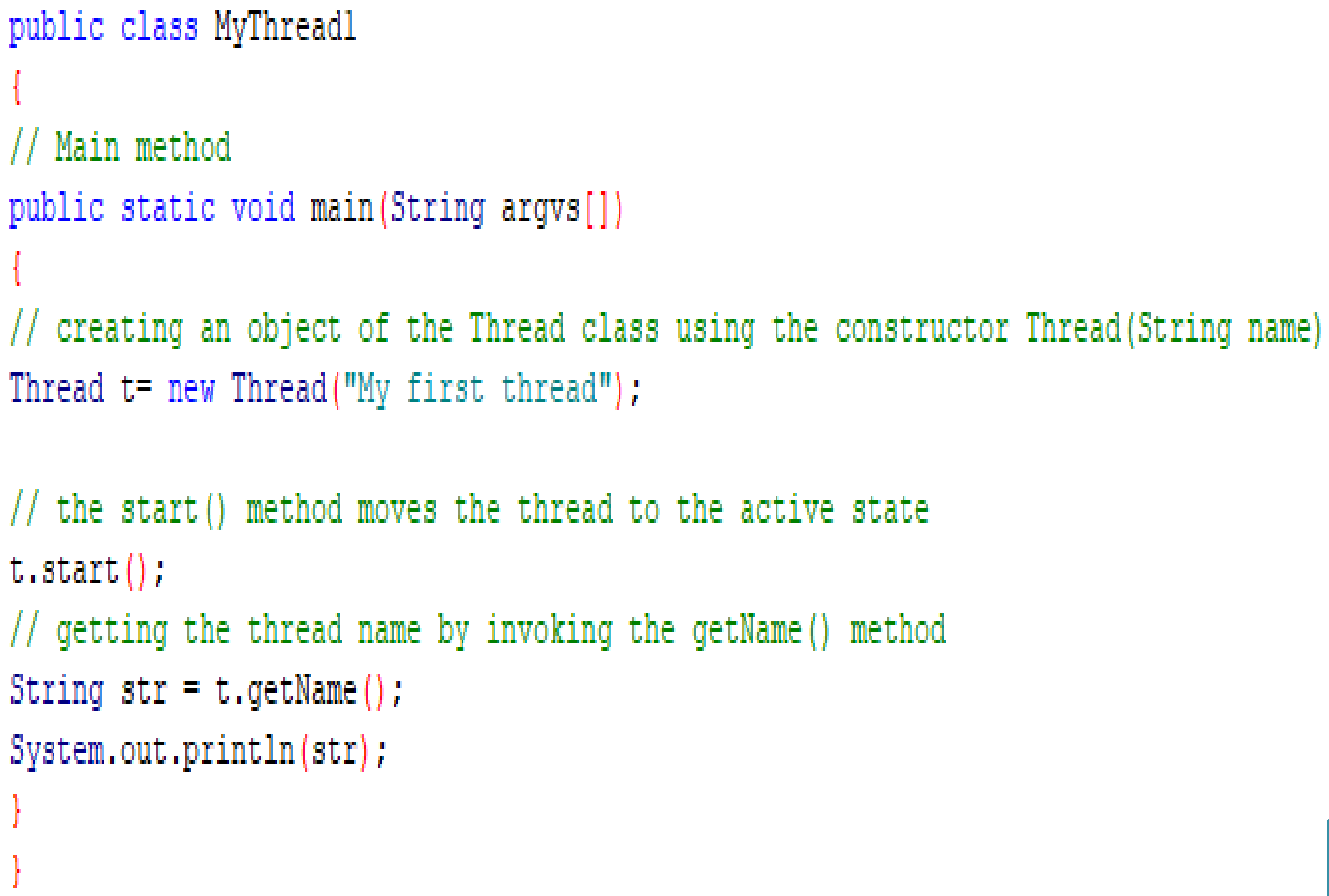
```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
  }
}
```

MULTITHREADING



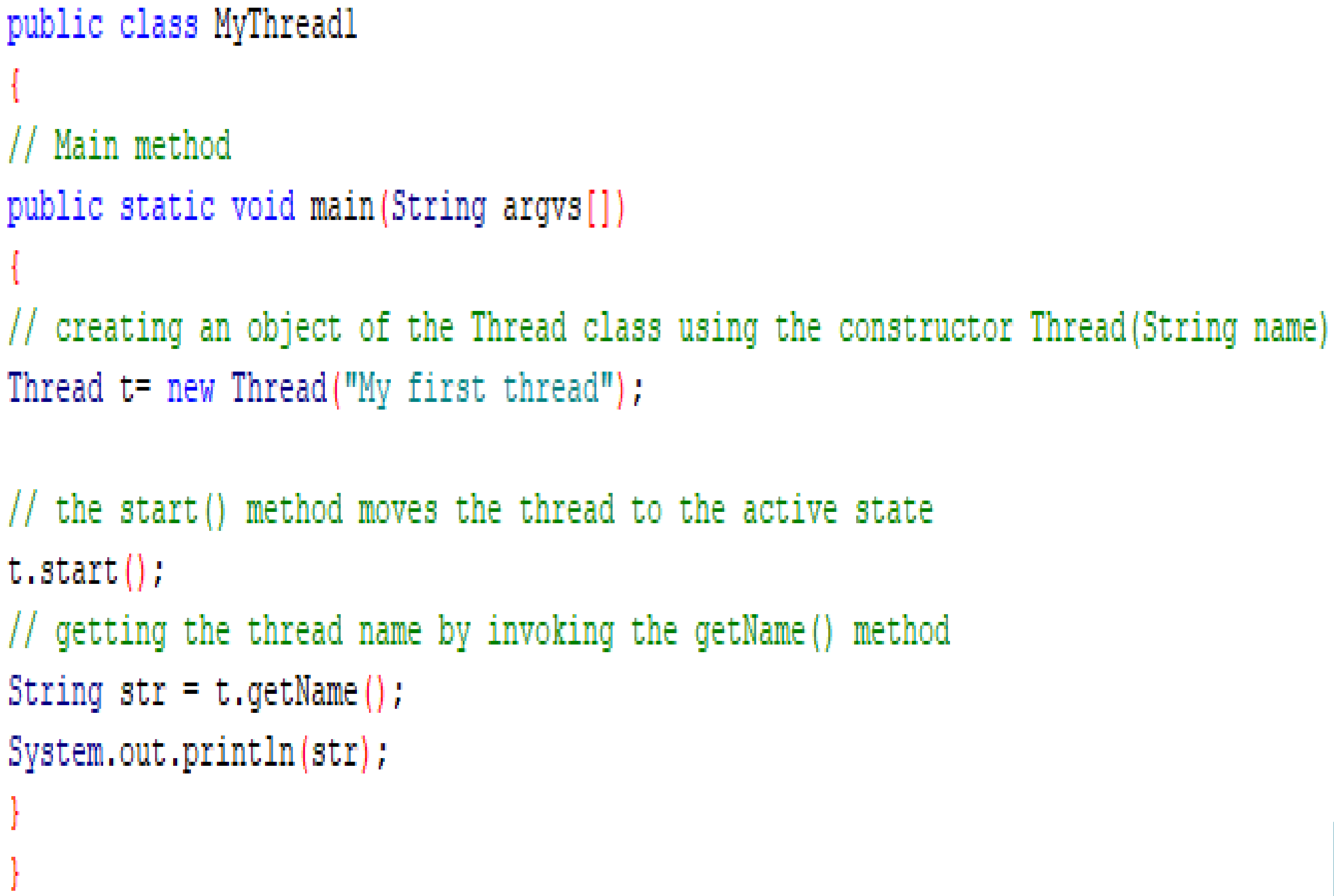
```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);    // Using the constructor Thread
t1.start();
 }
}
```

MULTITHREADING



```
public class MyThread1
{
// Main method
public static void main(String argvs[])
{
// creating an object of the Thread class using the constructor Thread(String name)
Thread t= new Thread("My first thread");

// the start() method moves the thread to the active state
t.start();
// getting the thread name by invoking the getName() method
String str = t.getName();
System.out.println(str);
}
}
```

MULTITHREADING



```
public class MyThread1
{
// Main method
public static void main(String argvs[])
{
// creating an object of the Thread class using the constructor Thread(String name)
Thread t= new Thread("My first thread");

// the start() method moves the thread to the active state
t.start();
// getting the thread name by invoking the getName() method
String str = t.getName();
System.out.println(str);
}
}
```

MULTITHREADING

Thread.sleep()

The method sleep() is being used to halt the working of a thread for a given amount of time.

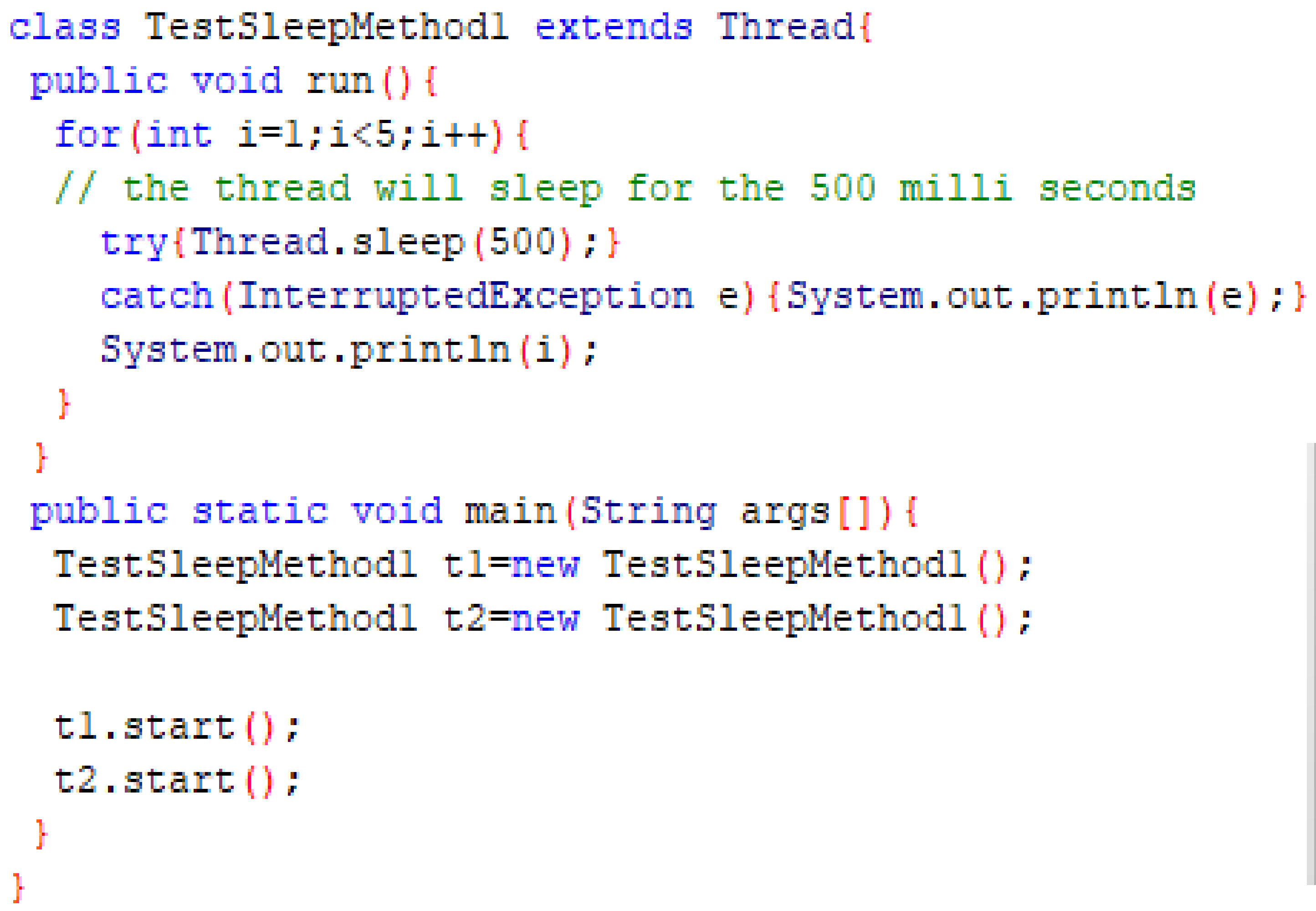
The time up to which the thread remains in the sleeping state is known as the sleeping time of the thread.

After the sleeping time is over, the thread starts its execution from where it has left.

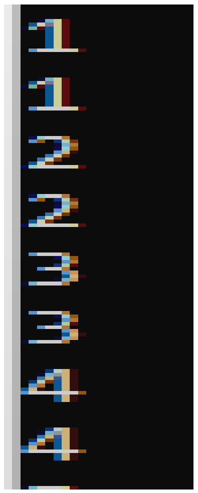
The sleep() Method Syntax:

public static void sleep(long mls) throws InterruptedException

MULTITHREADING



```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
  // the thread will sleep for the 500 milli seconds
    try{Thread.sleep(500);}
    catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();

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



```
1
1
2
2
3
3
4
4
```

MULTITHREADING

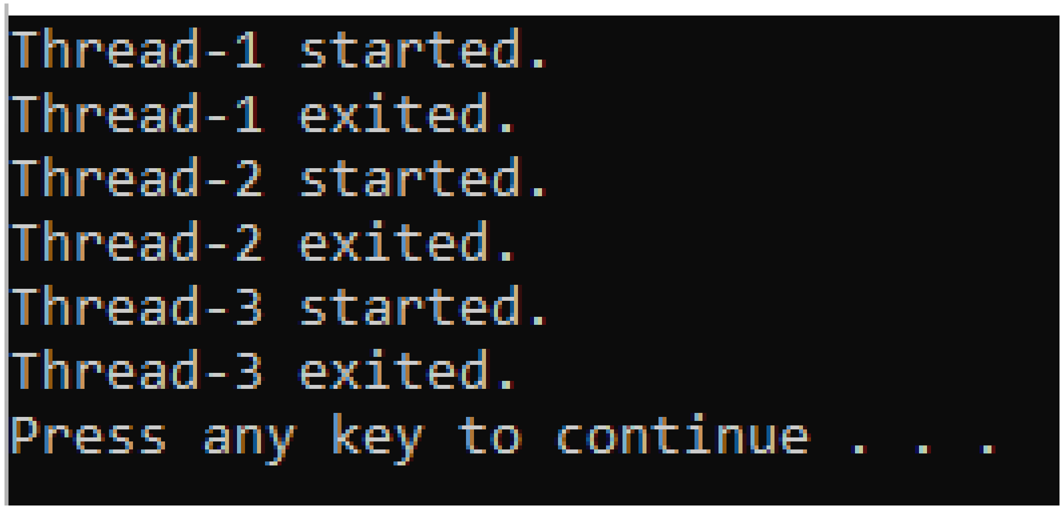
join() method

The join() method in Java is provided by the java.lang.Thread class that permits one thread to wait until the other thread to finish its execution.

Suppose th be the object the class Thread whose thread is doing its execution currently, then the th.join(); statement ensures that th is finished before the program does the execution of the next statement.

```
class CustomThread implements Runnable {
   public void run() {
      System.out.println(Thread.currentThread().getName() + " started.");
      try {
         Thread.sleep(500);
      } catch (InterruptedException e) {
         System.out.println(Thread.currentThread().getName() + " interrupted.");
      }
      System.out.println(Thread.currentThread().getName() + " exited.");
   }
}
public class Tester {
   public static void main(String args[]) throws InterruptedException {
      Thread t1 = new Thread( new CustomThread(), "Thread-1");
      t1.start();
      //main thread class the join on t1
      //and once t1 is finish then only t2 can start
      t1.join();
      Thread t2 = new Thread( new CustomThread(), "Thread-2");
      t2.start();
      //main thread class the join on t2
      //and once t2 is finish then only t3 can start
      t2.join();
      Thread t3 = new Thread( new CustomThread(), "Thread-3");
      t3.start();
   }
}
```

MULTITHREADING



```
Thread-1 started.
Thread-1 exited.
Thread-2 started.
Thread-2 exited.
Thread-3 started.
Thread-3 exited.
Press any key to continue . . .
```

MULTITHREADING

Naming Thread

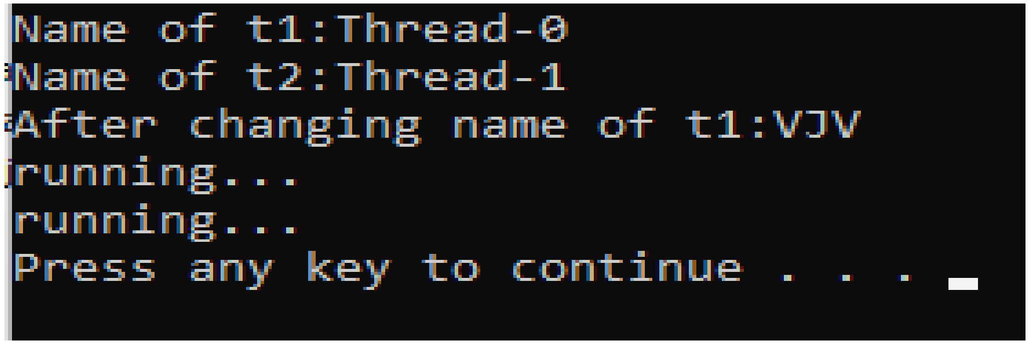
The Thread class provides methods to change and get the name of a thread. By default, each thread has a name, i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using the setName() method. The syntax of setName() and getName() methods are given below:

public String getName(): is used to return the name of a thread.

public void setName(String name): is used to change the name of a thread.

MULTITHREADING

```
public class TestMultiNaming1 extends Thread{
  public void run(){
   System.out.println("running...");
  }
 public static void main(String args[]){
  TestMultiNaming1 t1=new TestMultiNaming1();
  TestMultiNaming1 t2=new TestMultiNaming1();
  System.out.println("Name of t1:"+t1.getName());
  System.out.println("Name of t2:"+t2.getName());

  t1.start();
  t2.start();

  t1.setName("VJV");
  System.out.println("After changing name of t1:"+t1.getName());
 }
}
```



```
Name of t1:Thread-0
Name of t2:Thread-1
After changing name of t1:VJV
running...
running...
Press any key to continue . . .
```

MULTITHREADING

Thread Priority

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling).

MULTITHREADING

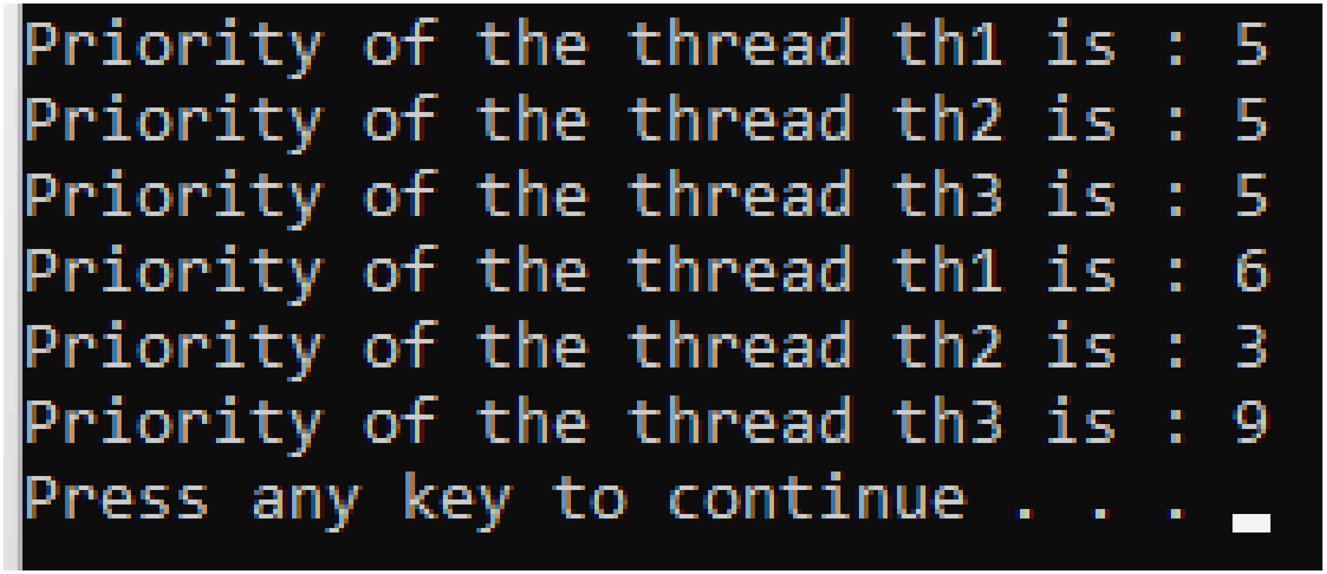
Setter & Getter Method of Thread Priority

public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority.

```
import java.lang.*;

public class ThreadPriorityExample extends Thread
{
public void run()
{
    System.out.println("Inside the run() method");
}
public static void main(String argvs[])
{
// Creating threads with the help of ThreadPriorityExample class
ThreadPriorityExample th1 = new ThreadPriorityExample();
ThreadPriorityExample th2 = new ThreadPriorityExample();
ThreadPriorityExample th3 = new ThreadPriorityExample();

// We did not mention the priority of the thread.
// Therefore, the priorities of the thread is 5, the default value

System.out.println("Priority of the thread th1 is : " + th1.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
System.out.println("Priority of the thread th3 is : " + th3.getPriority());

// Setting priorities of above threads by
// passing integer arguments
th1.setPriority(6);
th2.setPriority(3);
th3.setPriority(9);

System.out.println("Priority of the thread th1 is : " + th1.getPriority());
System.out.println("Priority of the thread th2 is : " + th2.getPriority());
System.out.println("Priority of the thread th3 is : " + th3.getPriority());

}
}
```

MULTITHREADING


```
Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
Priority of the thread th3 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Press any key to continue . . .
```

THREAD SYNCHRONIZATION

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time

The process by which this synchronization is achieved is called *thread synchronization*

“The synchronized keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock”

CONT..

Threads are synchronized in Java through the use of a monitor. Think of a monitor as an object that enables a thread to access a resource

Only one thread can use a monitor at any one time period. Programmers say that the thread *owns* the monitor for that period of time. The monitor is also called a *semaphore*

A thread can own a monitor only if no other thread owns the monitor

If the monitor is available, a thread can own the monitor and have exclusive access to the resource associated with the monitor

CONT..

If the monitor is not available, the thread is suspended until the monitor becomes available. Programmers say that the thread is *waiting* for the monitor

You have two ways in which you can synchronize threads:

- You can use the synchronized method or
- The synchronized statement.

SYNCHRONIZED STATEMENT

This is the general form of the synchronized statement:

- synchronized(object) { // statements to be synchronized}

Here, object is a reference to the object being synchronized

A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor

Calls to the methods contained in the synchronized block happen only after the thread enters the monitor of the object

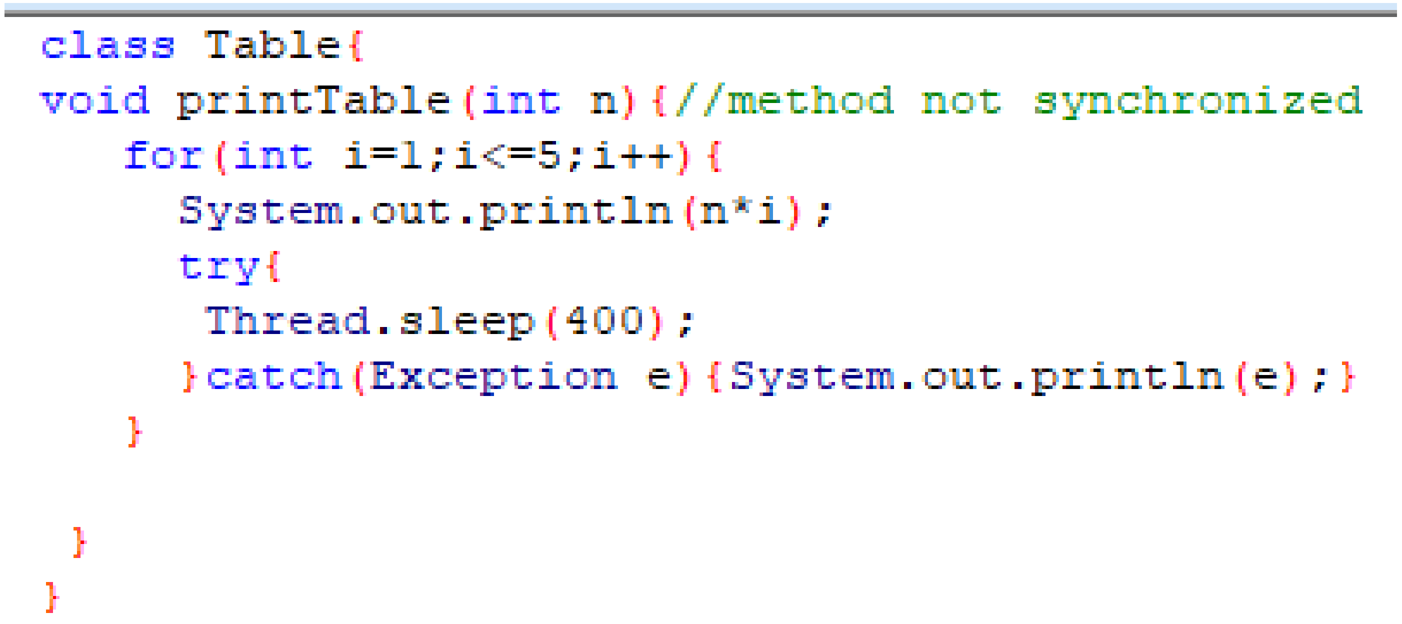
CONT..

Synchronizing a method is the best way to restrict the use of a method one thread at a time

However, there will be occasions when you won’t be able to synchronize a method, such as when you use a class that is provided to you by a third party

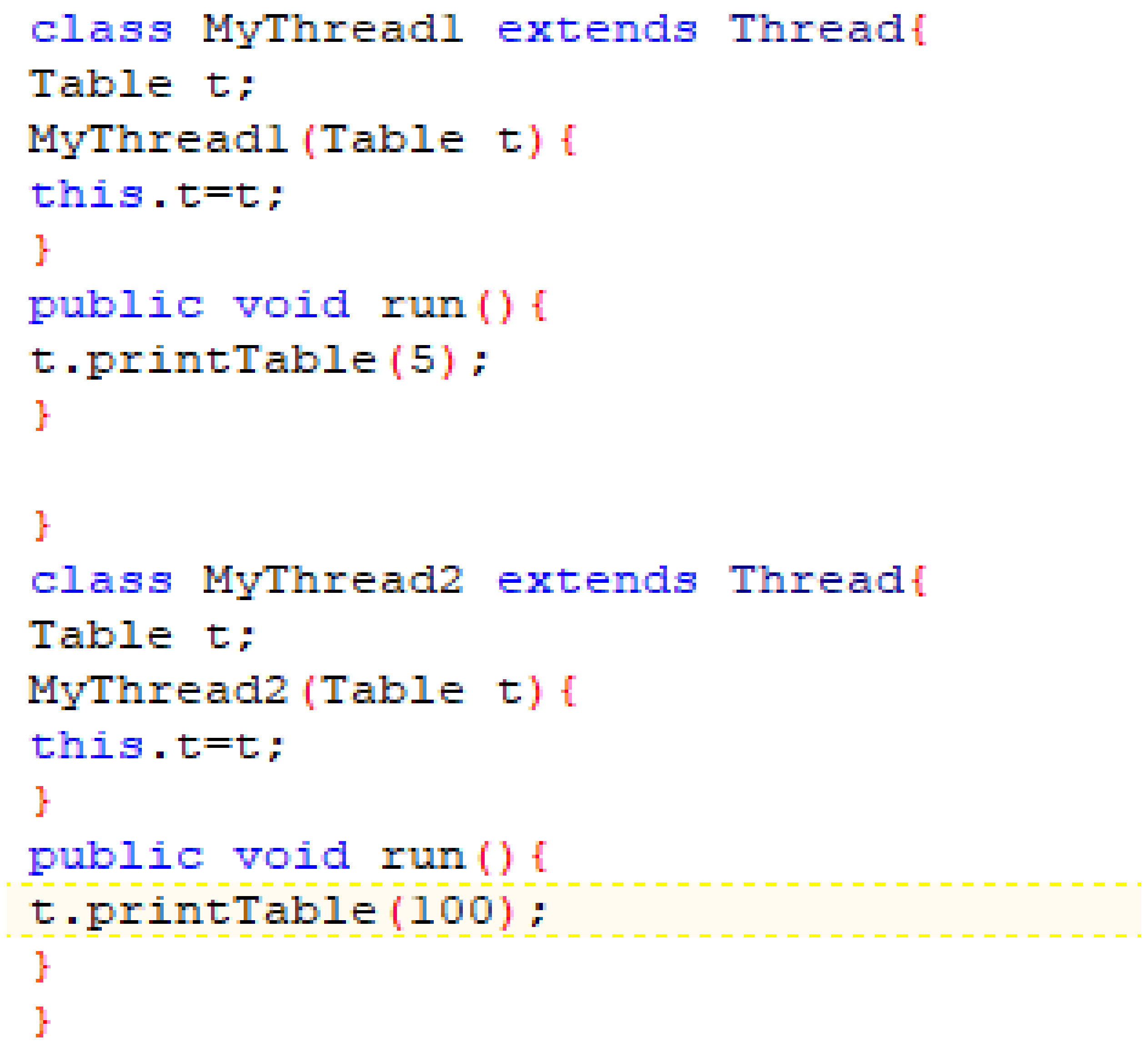
Although you can call methods within a synchronized block, the method declaration must be made outside a synchronized block

CONT..



```
class Table{
void printTable(int n){//method not synchronized
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }

 }
}
```

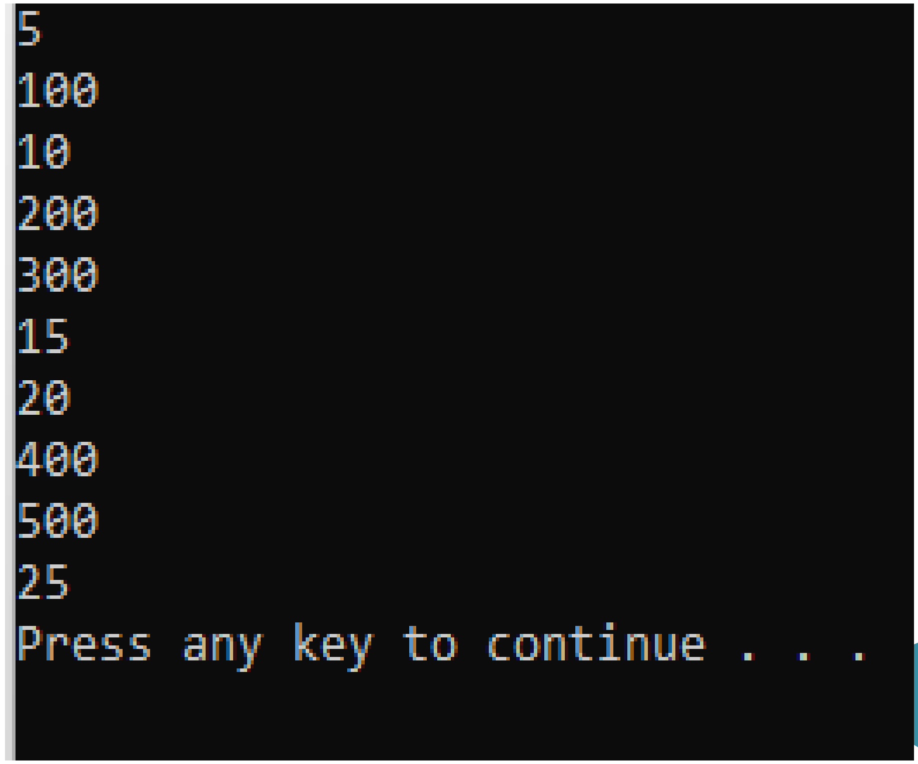
CONT..



```
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
```

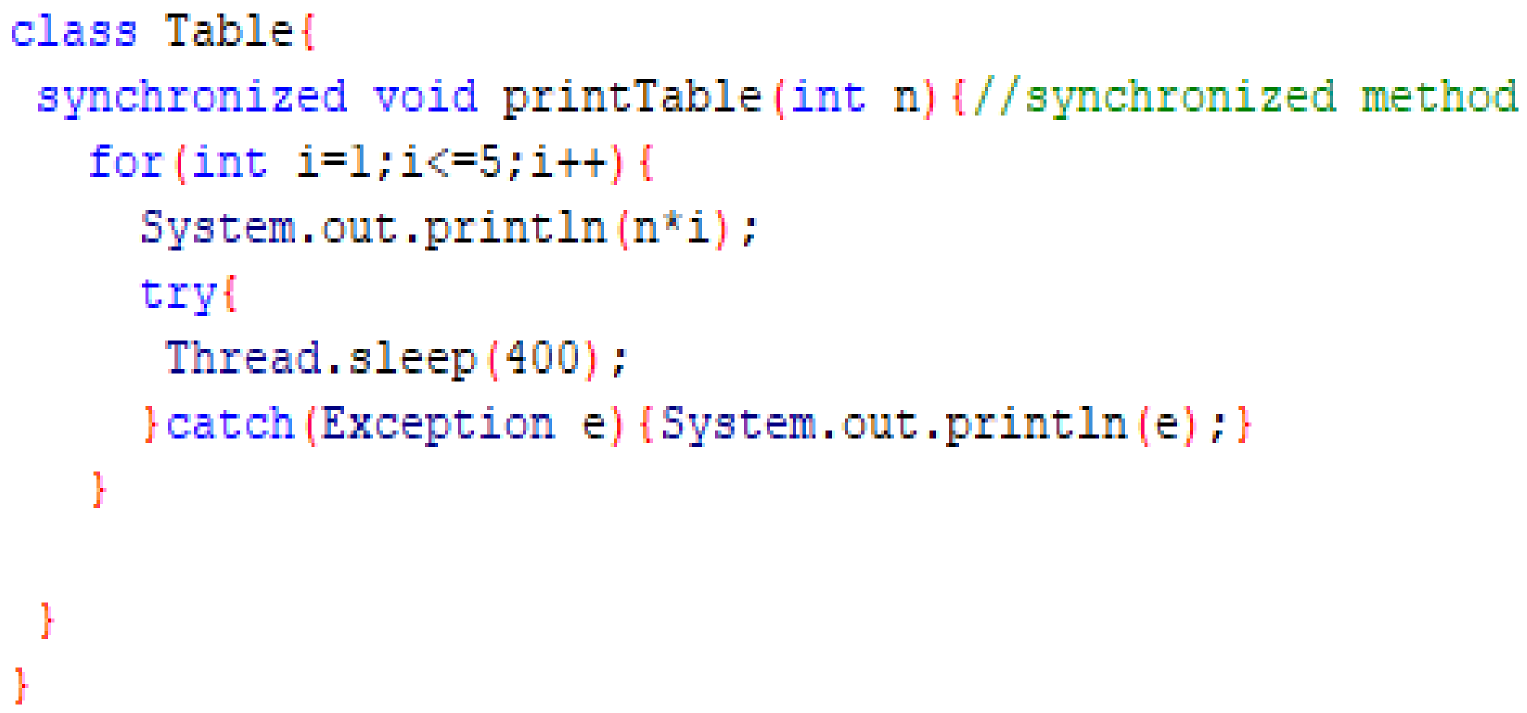
CONT..

```
class TestSynchronization1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```



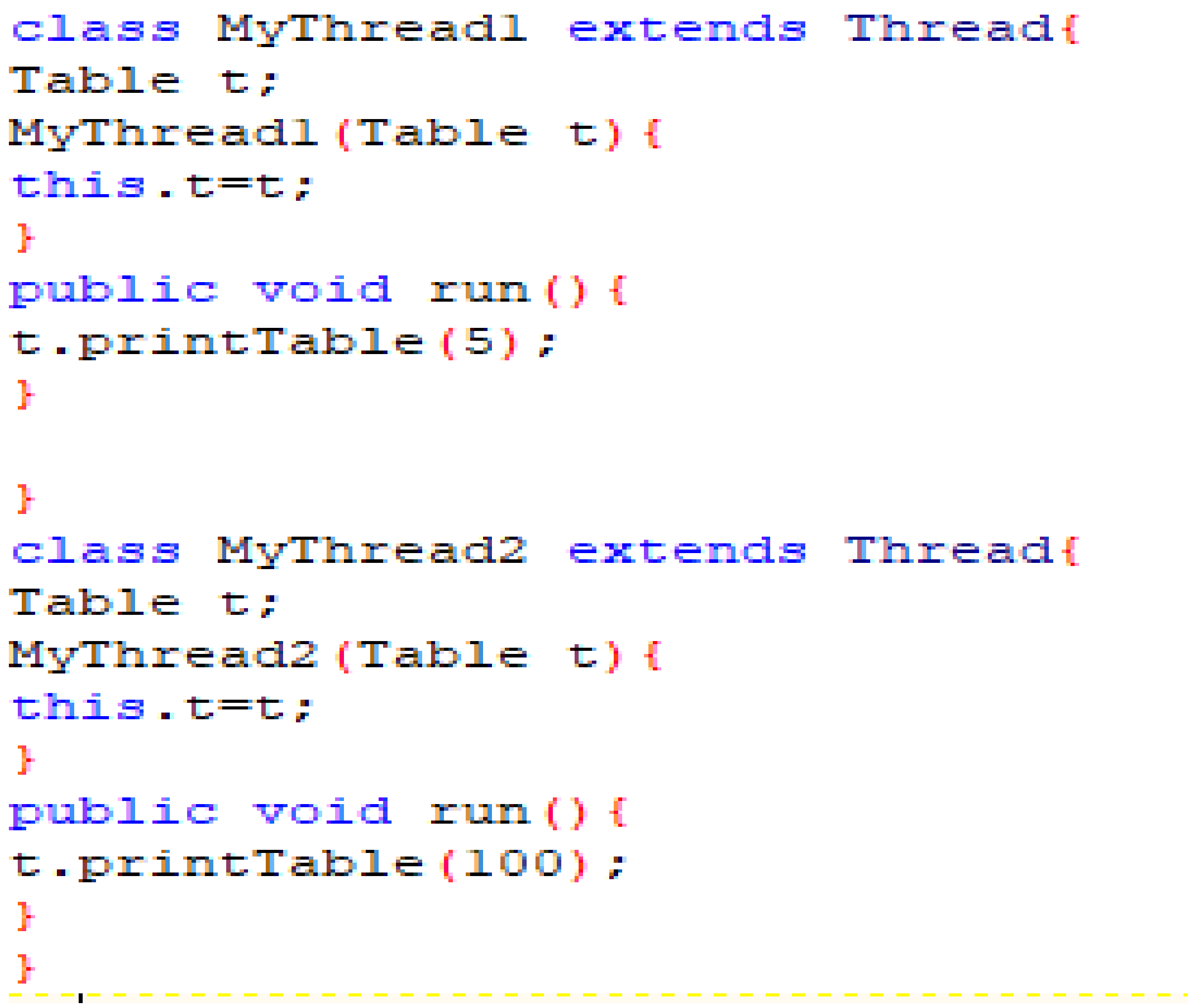
```
5
100
10
200
300
15
20
400
500
25
Press any key to continue . . .
```

JAVA SYNCHRONIZED METHOD



```
class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }

 }
}
```

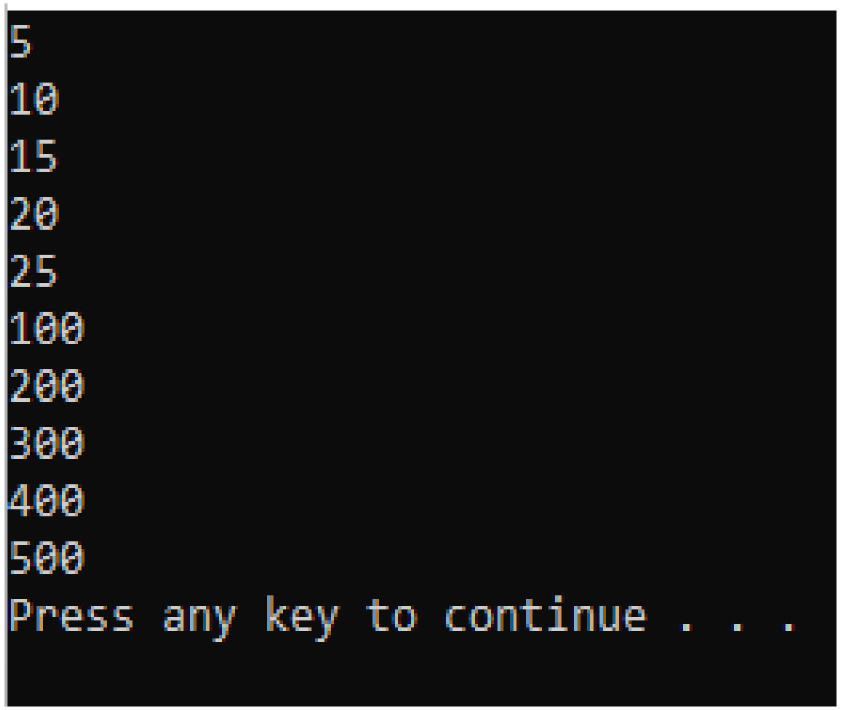
JAVA SYNCHRONIZED METHOD



```
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
```

JAVA SYNCHRONIZED METHOD

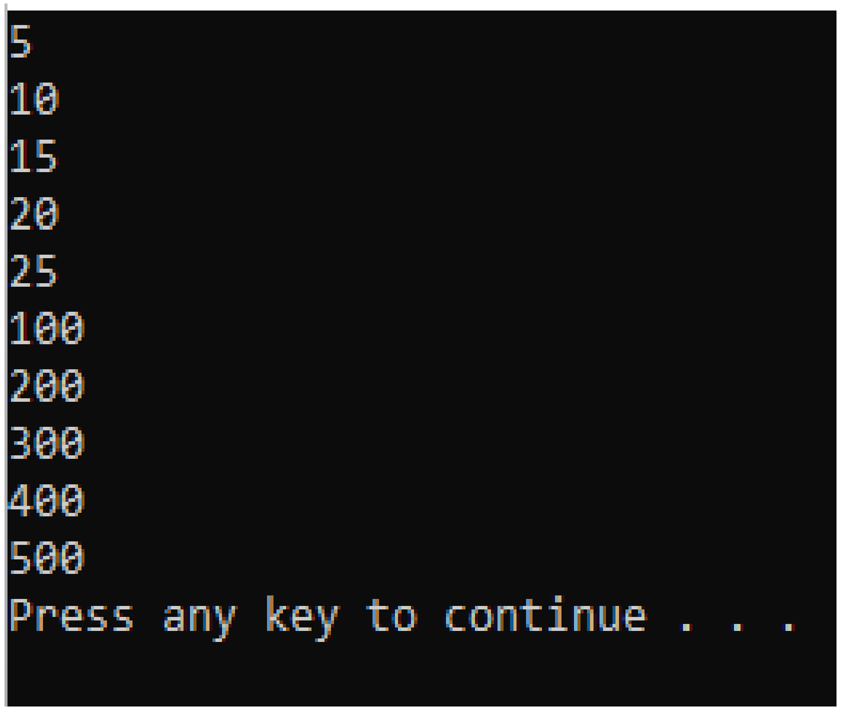
```
public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```



```
5
10
15
20
25
100
200
300
400
500
Press any key to continue . . .
```

JAVA SYNCHRONIZED METHOD

```java
public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```



```
5
10
15
20
25
100
200
300
400
500
Press any key to continue . . .
```

Files and I/O

Presented By : Prof.Honey Parmar

Introduction

- The java.io package, which provides support for I/O operations.
- A stream can be defined as a sequence of data.
- The InputStream is used to read data from a source.
- The OutputStream is used for writing data to a destination.
- The File class have several methods for working with directories and files such as creating new directories or files, deleting and renaming directories or files, listing the contents of a directory etc.

Introduction

In Java, streams are the sequence of data that are read from the source and written to the destination.

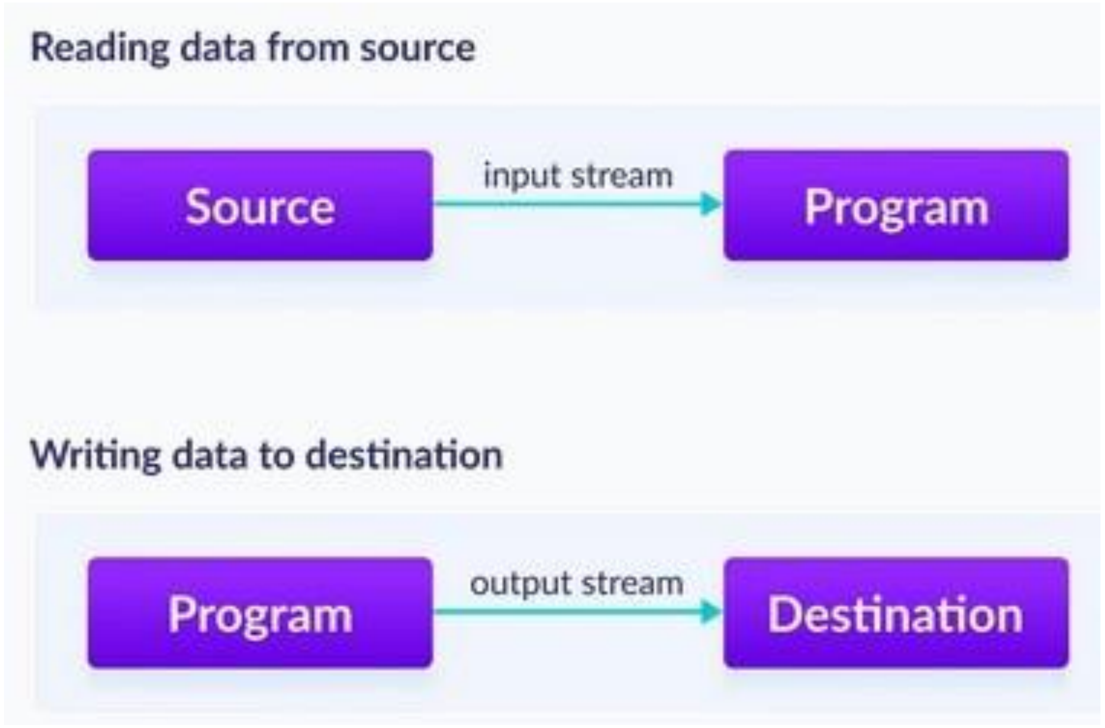
An input stream is used to read data from the source. And, an output stream is used to write data to the destination.

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Introduction

For example, in our first Hello World example, we have used System.out to print a string. Here, the System.out is a type of output stream.

Similarly, there are input streams to take input.



Introduction

Types of Streams

Depending upon the data a stream holds, it can be classified two types:

- Byte Stream
- Character Stream

Introduction

1. Byte Stream

We use Byte Stream to read and write a single byte (8 bits) of data.

All byte stream classes are derived from base abstract classes called InputStream and OutputStream.

Introduction

2. Character Stream

We use Character Stream to read and write a single character of data.

All the character stream classes are derived from base abstract classes Reader and Writer.

Java InputStream Class

The InputStream class of the java.io package is an abstract superclass that represents an input stream of bytes.

Since InputStream is an abstract class, it is not useful by itself. However, its subclasses can be used to read data.

Subclasses of InputStream

In order to use the functionality of InputStream, we can use its subclasses. Some of them are:

FileInputStream
ByteArrayInputStream
ObjectInputStream

Java OutputStream Class

The OutputStream class of the java.io package is an abstract superclass that represents an output stream of bytes.

Since OutputStream is an abstract class, it is not useful by itself. However, its subclasses can be used to write data.

Subclasses of OutputStream

In order to use the functionality of OutputStream, we can use its subclasses. Some of them are:

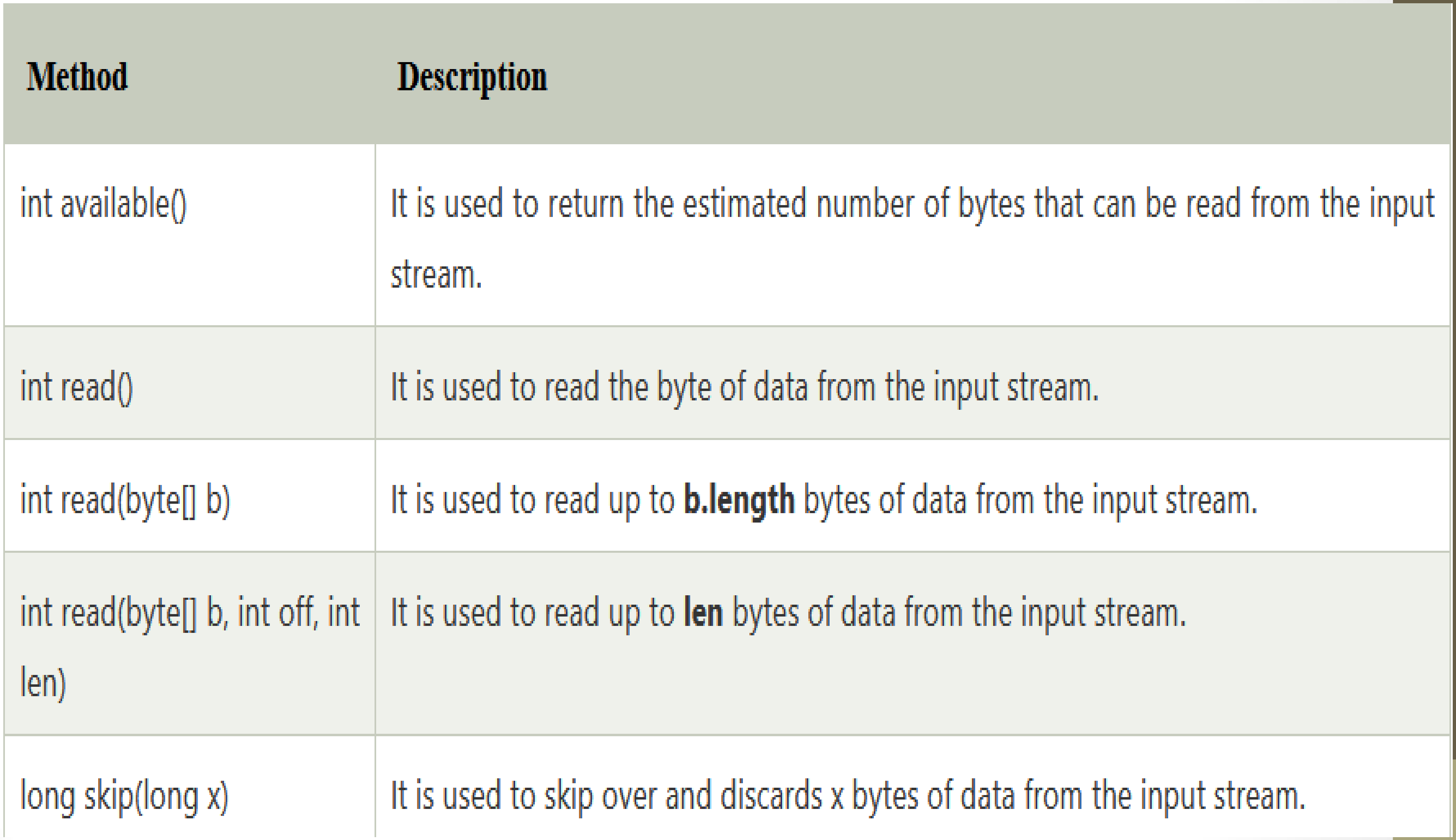
- FileOutputStream
- ByteArrayOutputStream
- ObjectOutputStream

Java FileInputStream Class

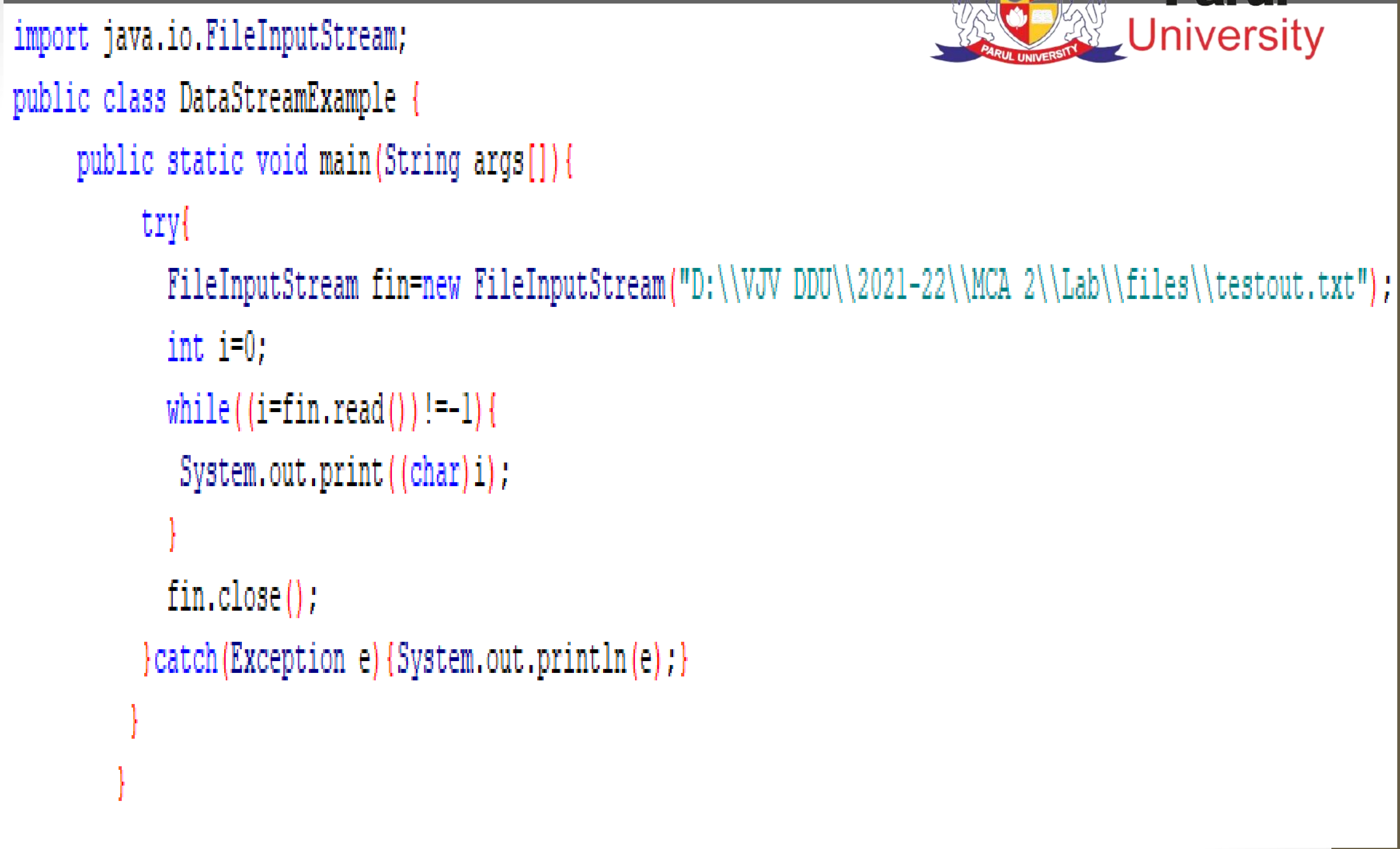
- The FileInputStream class of the java.io package can be used to read data (in bytes) from files.

- It extends the InputStream abstract class.

Java FileInputStream Class



Method	Description
int available()	It is used to return the estimated number of bytes that can be read from the input stream.
int read()	It is used to read the byte of data from the input stream.
int read(byte[] b)	It is used to read up to **b.length** bytes of data from the input stream.
int read(byte[] b, int off, int len)	It is used to read up to **len** bytes of data from the input stream.
long skip(long x)	It is used to skip over and discards x bytes of data from the input stream.



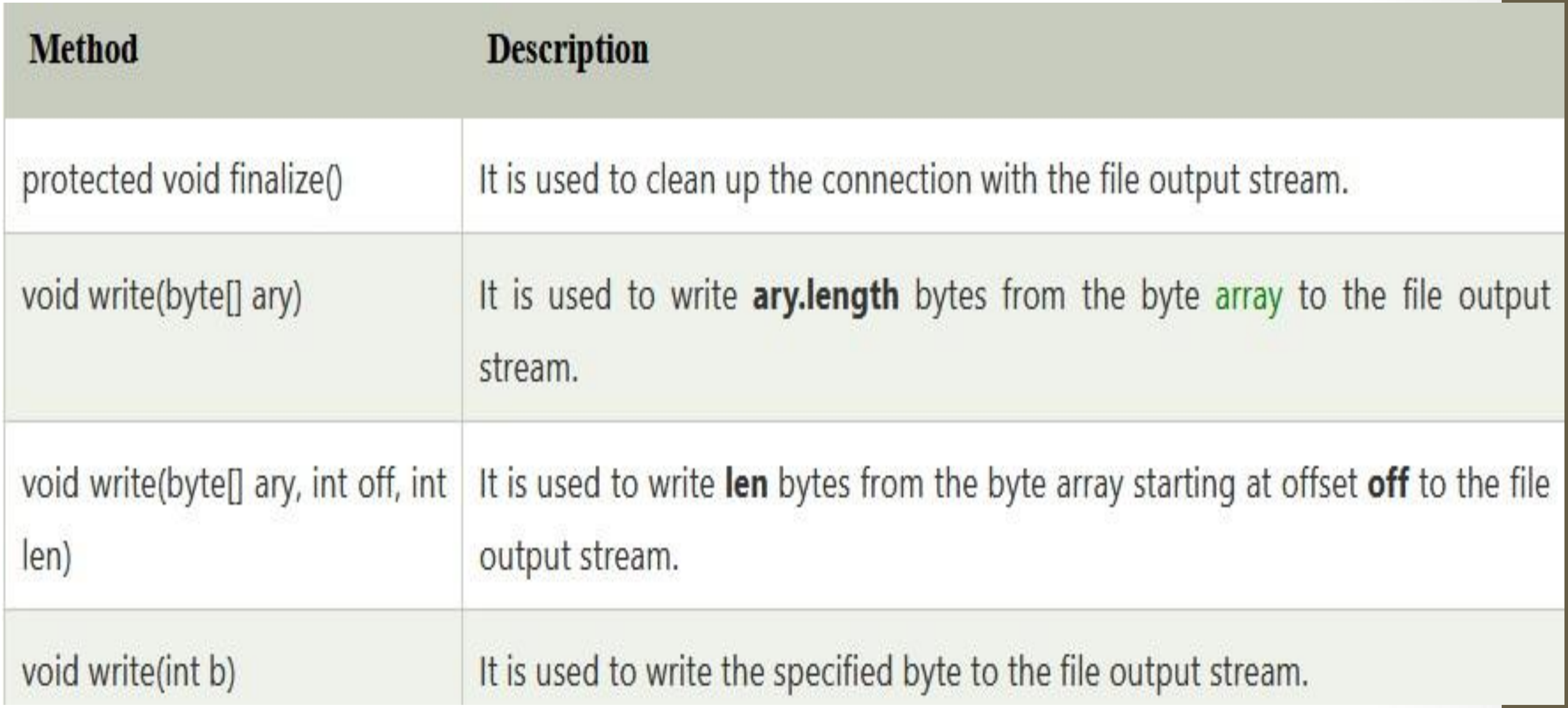
```
import java.io.FileInputStream;
public class DataStreamExample {
    public static void main(String args[]){
        try{
          FileInputStream fin=new FileInputStream("D:\\VJV DDU\\2021-22\\MCA 2\\Lab\\files\\testout.txt");
          int i=0;
          while((i=fin.read())!=-1){
           System.out.print((char)i);
          }
          fin.close();
        }catch(Exception e){System.out.println(e);}
      }
     }
```

Java FileOutputStream Class

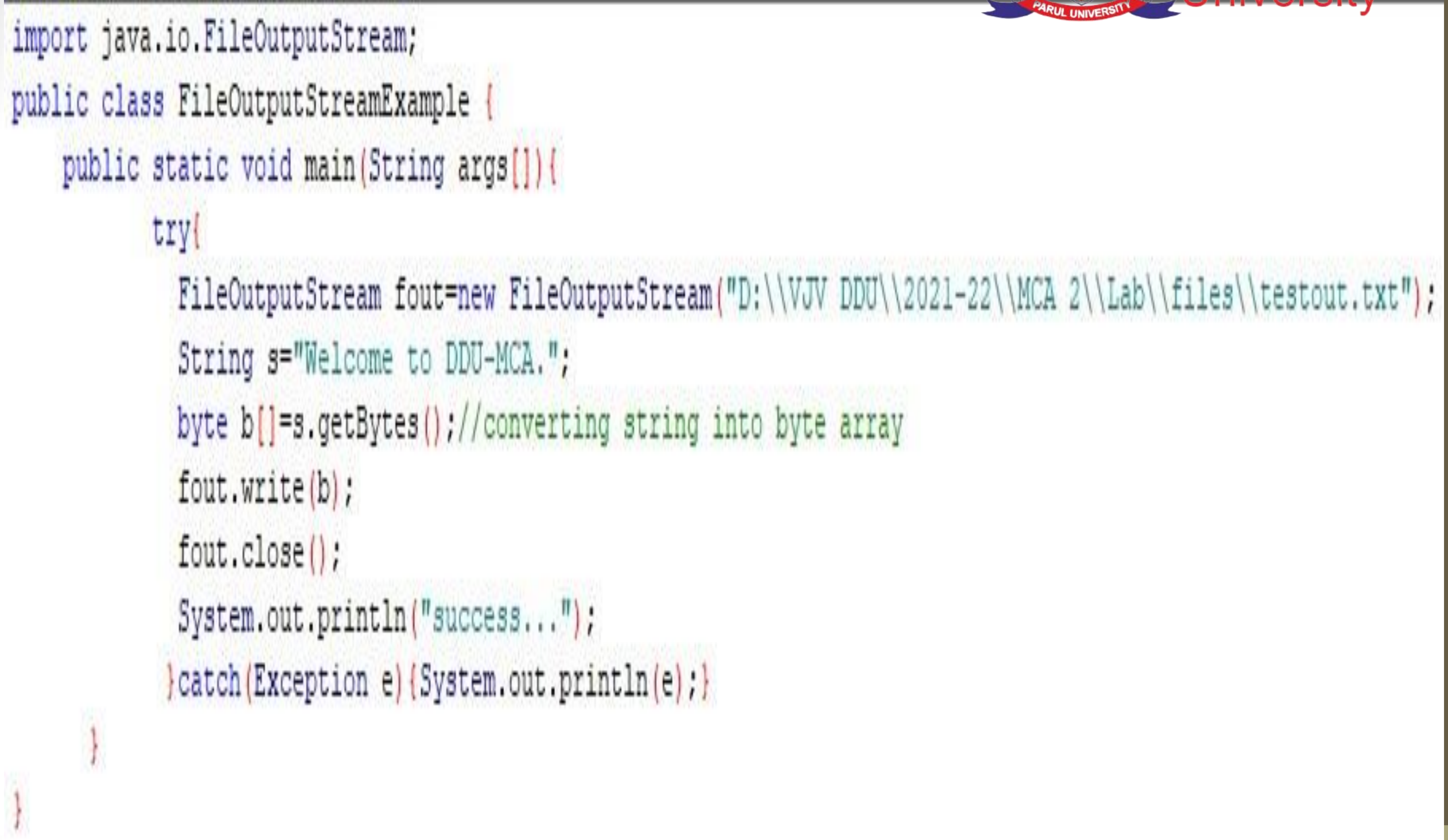
- The FileOutputStream class of the java.io package can be used to write data (in bytes) to the files.

- It extends the OutputStream abstract class.

Java FileOutputStream Class



Method	Description
protected void finalize()	It is used to clean up the connection with the file output stream.
void write(byte[] ary)	It is used to write **ary.length** bytes from the byte array to the file output stream.
void write(byte[] ary, int off, int len)	It is used to write **len** bytes from the byte array starting at offset **off** to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.

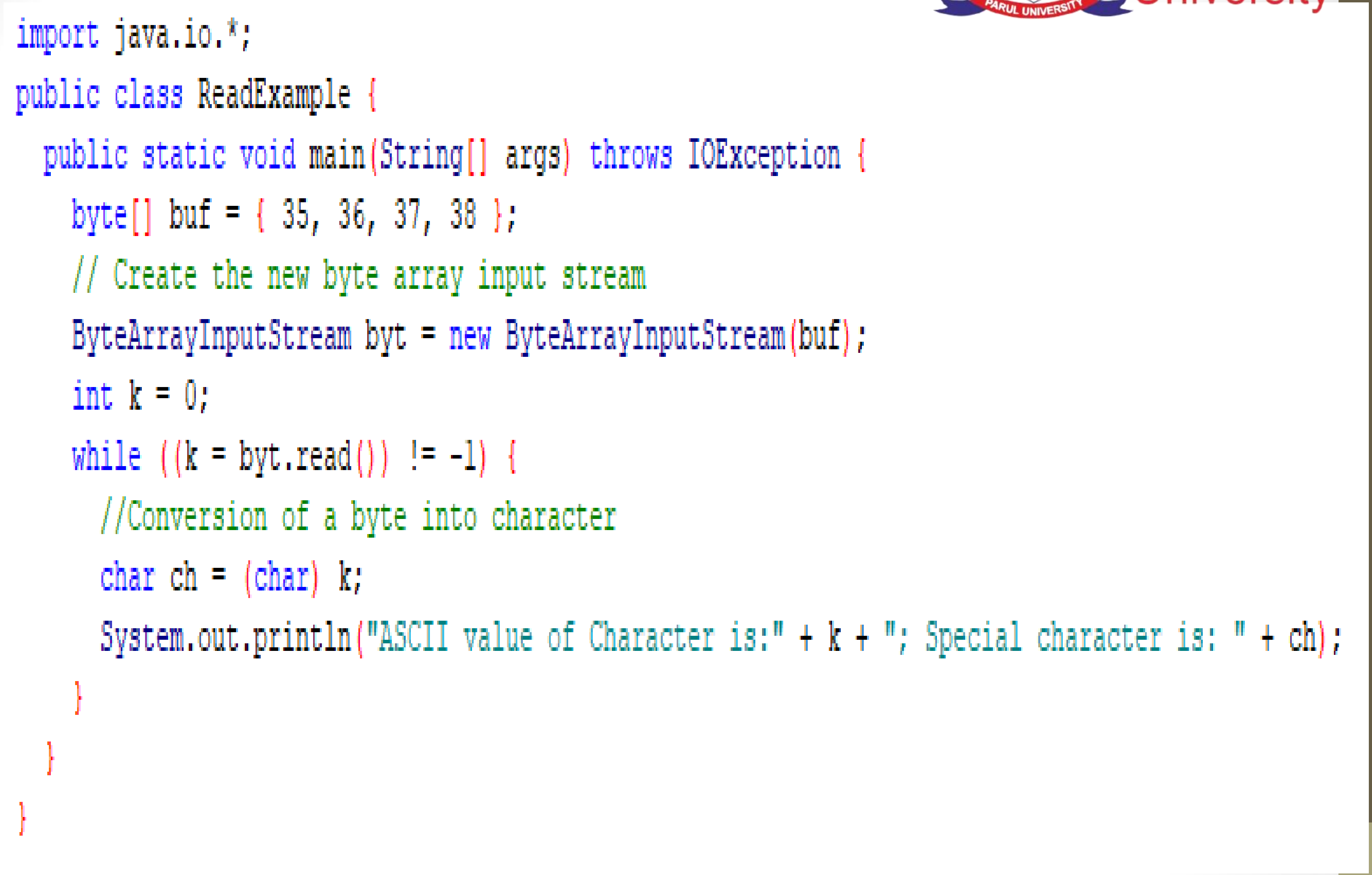


```
import java.io.FileOutputStream;
public class FileOutputStreamExample {
    public static void main(String args[]){
            try{
              FileOutputStream fout=new FileOutputStream("D:\\VJV DDU\\2021-22\\MCA 2\\Lab\\files\\testout.txt");
              String s="Welcome to DDU-MCA.";
              byte b[]=s.getBytes();//converting string into byte array
              fout.write(b);
              fout.close();
              System.out.println("success...");
             }catch(Exception e){System.out.println(e);}
      }
}
```

Java ByteArrayInputStream Class

- The ByteArrayInputStream class of the java.io package can be used to read an array of input data (in bytes).
- It extends the InputStream abstract class.
- Note: In ByteArrayInputStream, the input stream is created using the array of bytes. It includes an internal array to store data of that particular byte array.

Methods	Description
int available()	It is used to return the number of remaining bytes that can be read from the input stream.
int read()	It is used to read the next byte of data from the input stream.
int read(byte[] ary, int off, int len)	It is used to read up to len bytes of data from an array of bytes in the input stream.



```
import java.io.*;
public class ReadExample {
  public static void main(String[] args) throws IOException {
    byte[] buf = { 35, 36, 37, 38 };
    // Create the new byte array input stream
    ByteArrayInputStream byt = new ByteArrayInputStream(buf);
    int k = 0;
    while ((k = byt.read()) != -1) {
      //Conversion of a byte into character
      char ch = (char) k;
      System.out.println("ASCII value of Character is:" + k + "; Special character is: " + ch);
    }
  }
}
```

Java ByteArrayOutputStream

- The ByteArrayOutputStream class of the java.io package can be used to write an array of output data (in bytes).

- It extends the OutputStream abstract class.

Java ByteArrayOutputStream

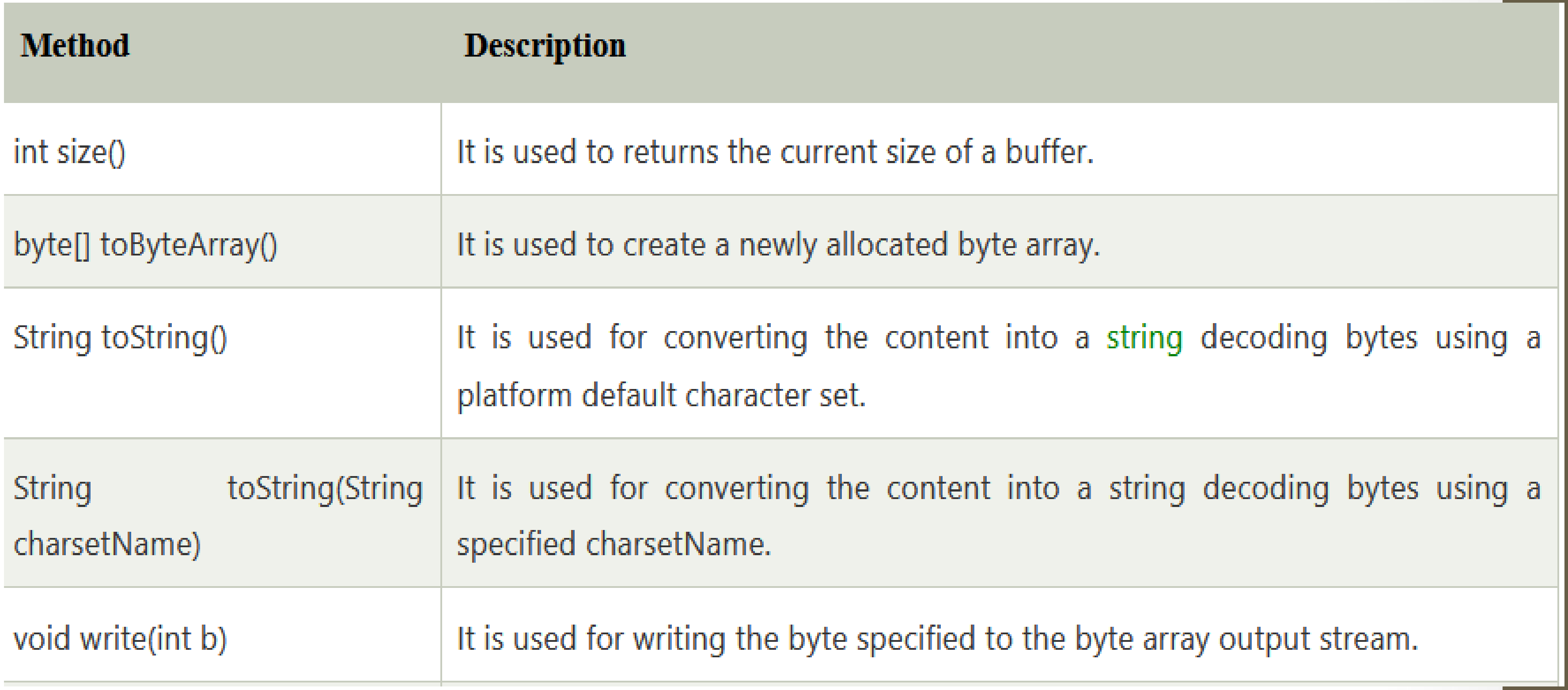
Java ByteArrayOutputStream Class

Java ByteArrayOutputStream class is used to write common data into multiple files. In this stream, the data is written into a byte array which can be written to multiple streams later.

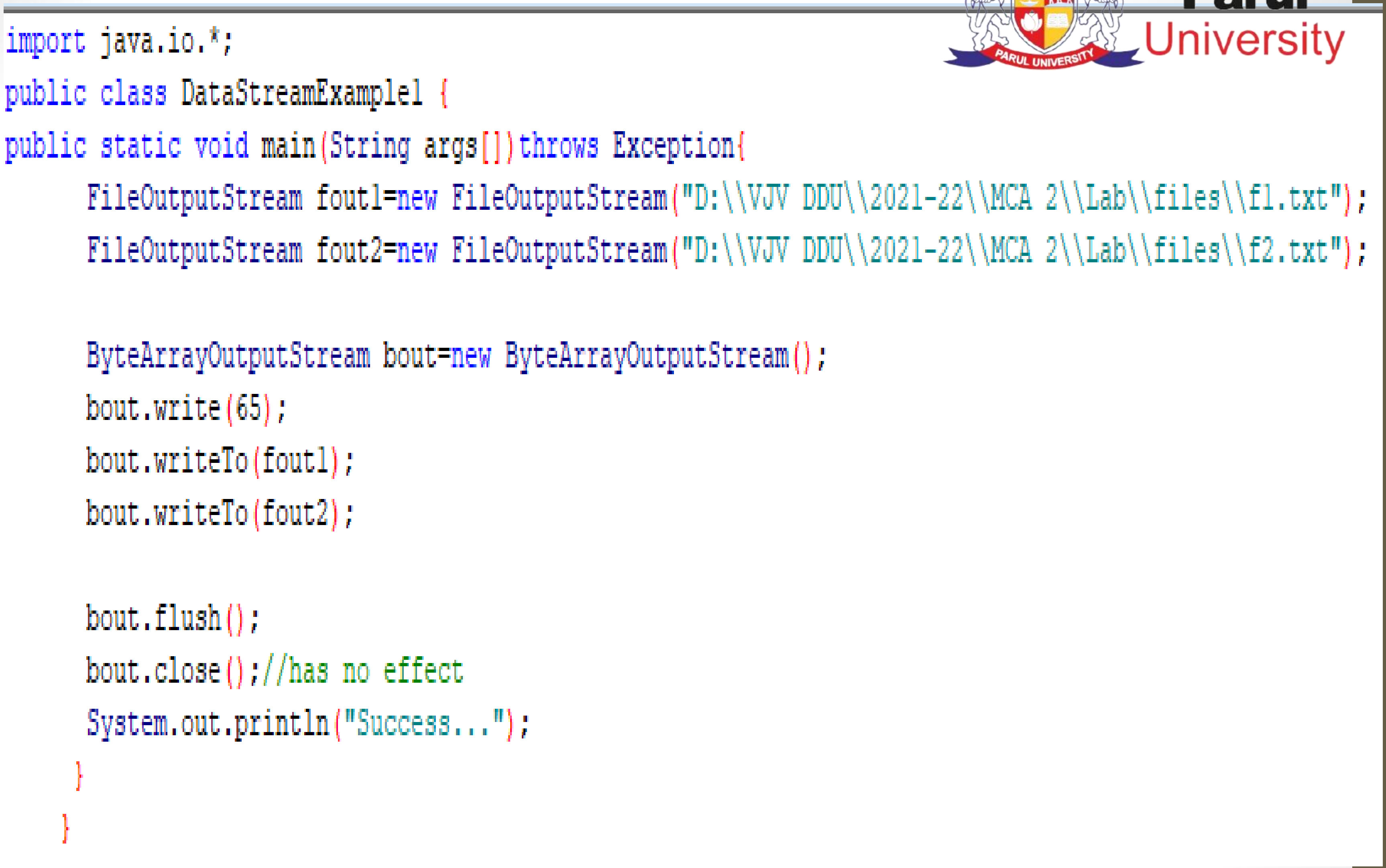
The ByteArrayOutputStream holds a copy of data and forwards it to multiple streams.

The buffer of ByteArrayOutputStream automatically grows according to data.

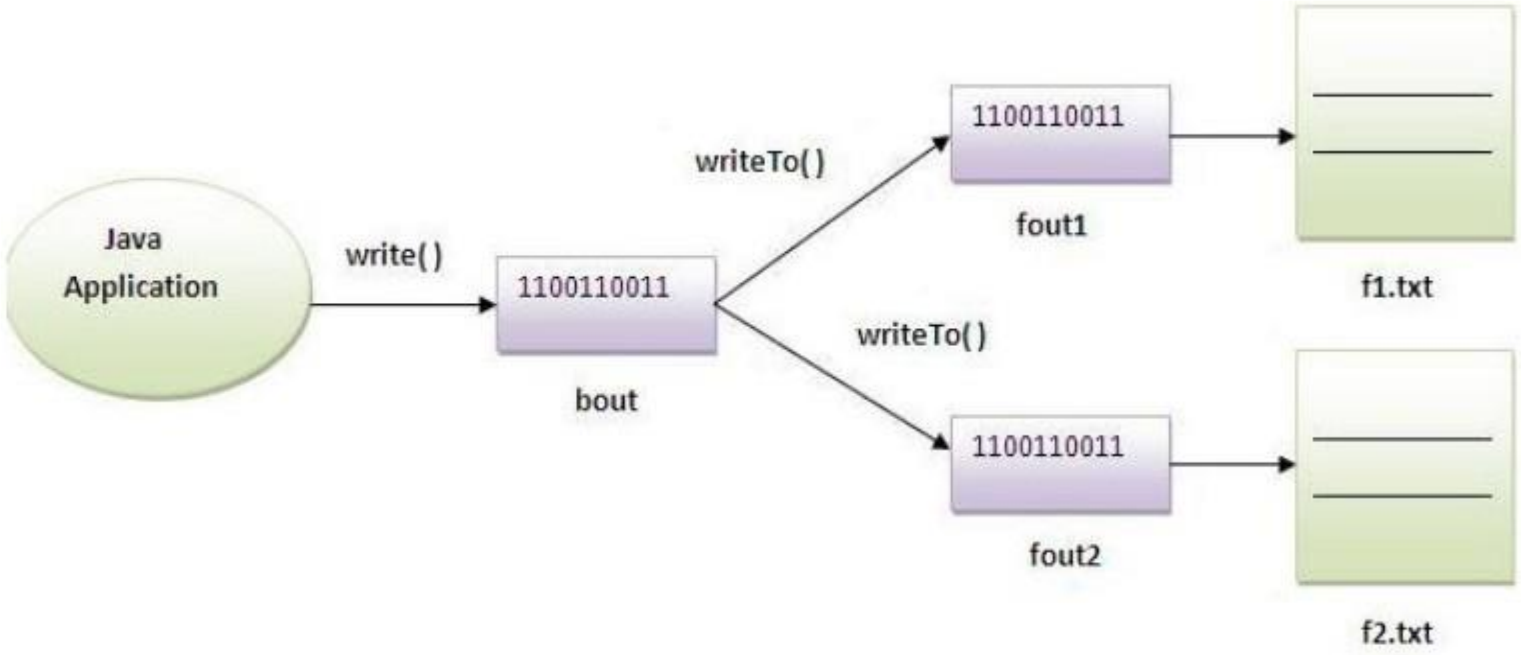
Introduction



Method	Description
int size()	It is used to returns the current size of a buffer.
byte[] toByteArray()	It is used to create a newly allocated byte array.
String toString()	It is used for converting the content into a string decoding bytes using a platform default character set.
String toString(String charsetName)	It is used for converting the content into a string decoding bytes using a specified charsetName.
void write(int b)	It is used for writing the byte specified to the byte array output stream.



```
import java.io.*;
public class DataStreamExample1 {
public static void main(String args[])throws Exception{
      FileOutputStream fout1=new FileOutputStream("D:\\VJV DDU\\2021-22\\MCA 2\\Lab\\files\\f1.txt");
      FileOutputStream fout2=new FileOutputStream("D:\\VJV DDU\\2021-22\\MCA 2\\Lab\\files\\f2.txt");

      ByteArrayOutputStream bout=new ByteArrayOutputStream();
      bout.write(65);
      bout.writeTo(fout1);
      bout.writeTo(fout2);

      bout.flush();
      bout.close();//has no effect
      System.out.println("Success...");
     }
    }
```


Java Application
write( )
1100110011
bout
writeTo( )
1100110011
fout1
writeTo( )
1100110011
fout2
f1.txt
f2.txt

Java ObjectInputStream

- The ObjectInputStream class of the java.io package can be used to read objects that were previously written by ObjectOutputStream.

- It extends the InputStream abstract class.

Java ObjectInputStream

- Working of ObjectInputStream
- The ObjectInputStream is mainly used to read data written by the ObjectOutputStream.
- Basically, the ObjectOutputStream converts Java objects into corresponding streams. This is known as serialization. Those converted streams can be stored in files or transferred through networks.

Java ObjectInputStream

- Now, if we need to read those objects, we will use the ObjectInputStream that will convert the streams back to corresponding objects. This is known as deserialization.

Java ObjectInputStream

- Create an ObjectInputStream
- In order to create an object input stream, we must import the java.io.ObjectInputStream package first. Once we import the package, here is how we can create an input stream.
- // Creates a file input stream linked with specified file
- FileInputStream fileStream = new FileInputStream(String file);
- // Creates an object input stream using the file input stream
- ObjectInputStream objStream = new ObjectInputStream(fileStream);
- In the above example, we have created an object input stream named objStream that is linked with the file input stream named fileStream.
- Now, the objStream can be used to read objects from the file.

Java ObjectOutputStream

- The ObjectOutputStream class of the java.io package can be used to write objects that can be read by ObjectInputStream.
- It extends the OutputStream abstract class.
- Working of ObjectOutputStream
- Basically, the ObjectOutputStream encodes Java objects using the class name and object values. And, hence generates corresponding streams. This process is known as serialization.
- Those converted streams can be stored in files and can be transferred among networks.
- Note: The ObjectOutputStream class only writes those objects that implement the Serializable interface. This is because objects need to be serialized while writing to the stream

Java ObjectOutputStream

- Create an ObjectOutputStream
- In order to create an object output stream, we must import the java.io.ObjectOutputStream package first. Once we import the package, here is how we can create an output stream.
- // Creates a FileOutputStream where objects from ObjectOutputStream are written
- FileOutputStream fileStream = new FileOutputStream(String file);
- // Creates the ObjectOutputStream
- ObjectOutputStream objStream = new ObjectOutputStream(fileStream);
- In the above example, we have created an object output stream named objStream that is linked with the file output stream named fileStream.

Serialization and Deserialization

- **Serialization in Java** is a mechanism of *writing the state of an object into a byte-stream*. It is mainly used in Hibernate, RMI (Remote Method Invocation), JPA (Java Persistence API), EJB (Enterprise JavaBeans)and JMS (Java Message Service) technologies.

- The reverse operation of serialization is called *deserialization* where byte-stream is converted into an object.

- The serialization and deserialization process is platform-independent, it means you can serialize an object on one platform and deserialize it on a different platform.

Serialization and Deserialization

For serializing the object, we call the **writeObject()** method of *ObjectOutputStream* class, and for deserialization we call the **readObject()** method of *ObjectInputStream* class.

We must have to implement the *Serializable* interface for serializing the object.

Advantages of Java Serialization
It is mainly used to travel object's state on the network (that is known as marshalling).

Serialization and Deserialization

java.io.Serializable interface

The **Serializable** interface must be implemented by the class whose object needs to be persisted.

The String class and all the wrapper classes implement the *java.io.Serializable* interface by default.

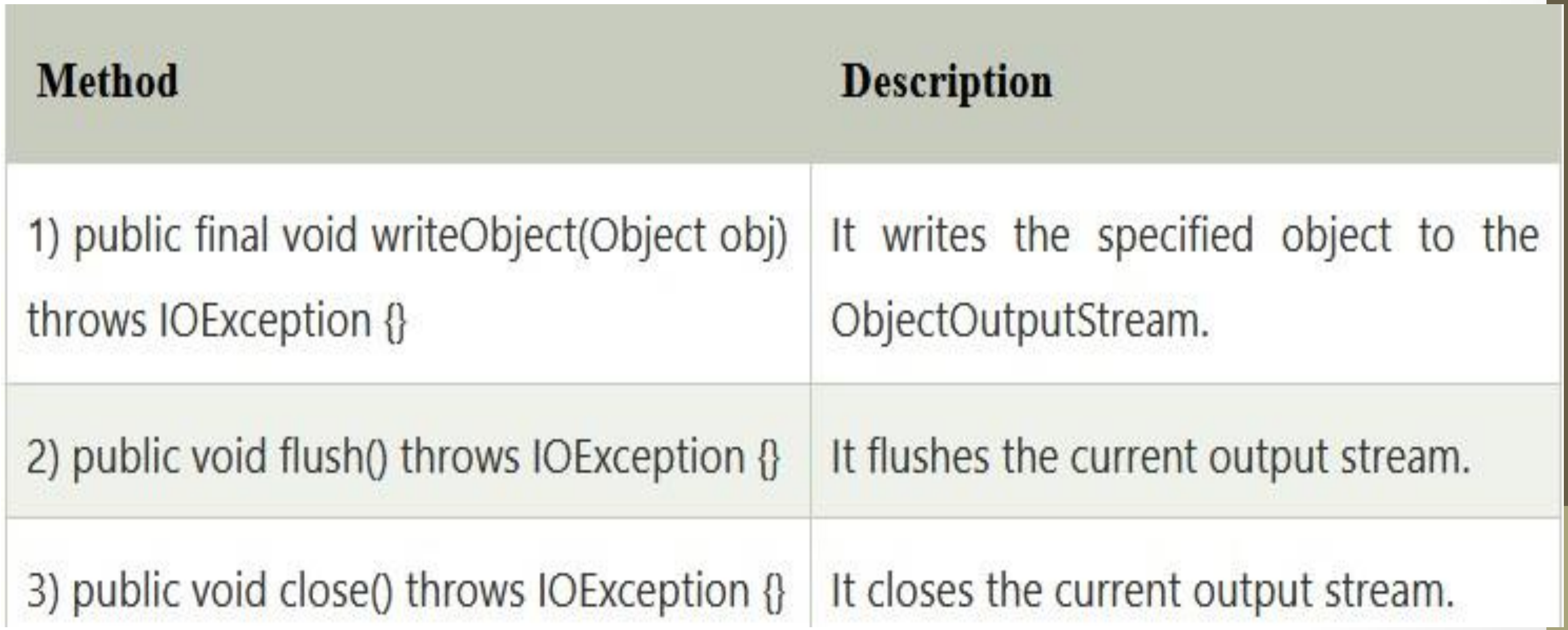
Serialization and Deserialization

```
import java.io.Serializable;
public class Student implements Serializable{
  int id;
 String name;
 public Student(int id, String name) {
 this.id = id;
 this.name = name;
 }
}
```

In the above example, ***Student*** class implements Serializable interface. Now its objects can be converted into stream.

Serialization and Deserialization

ObjectOutputStream class
The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream.

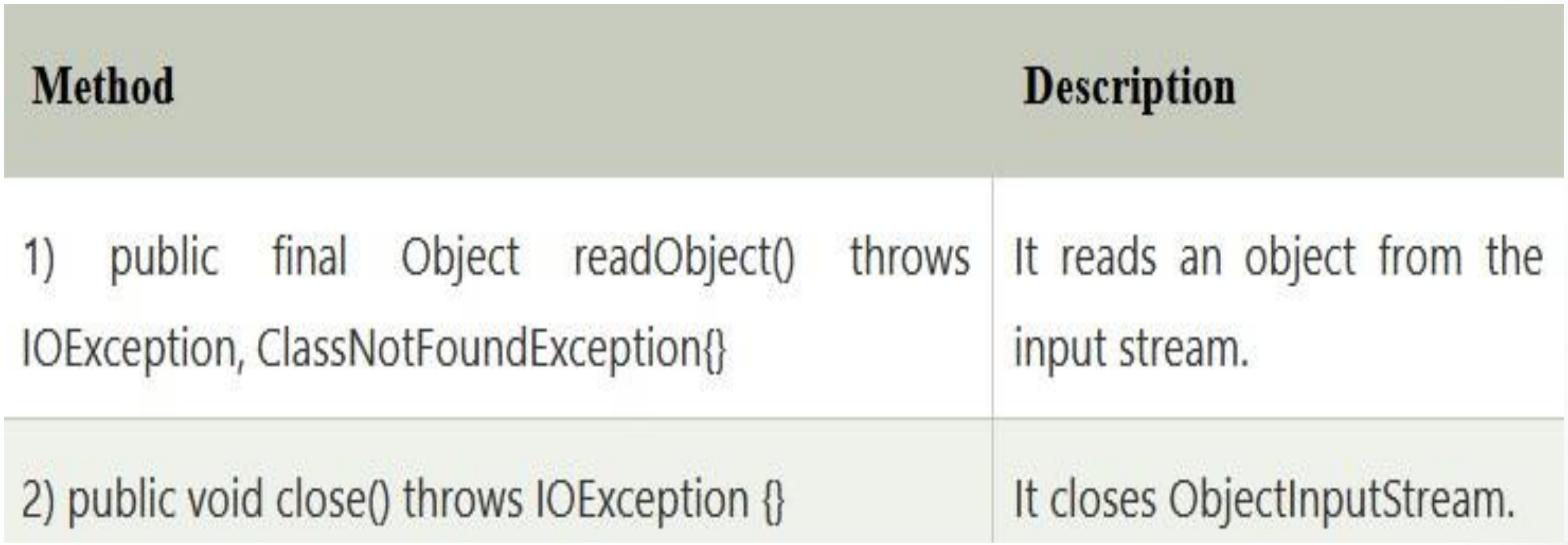


Method	Description
1) public final void writeObject(Object obj) throws IOException {}	It writes the specified object to the ObjectOutputStream.
2) public void flush() throws IOException {}	It flushes the current output stream.
3) public void close() throws IOException {}	It closes the current output stream.

Serialization and Deserialization

ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

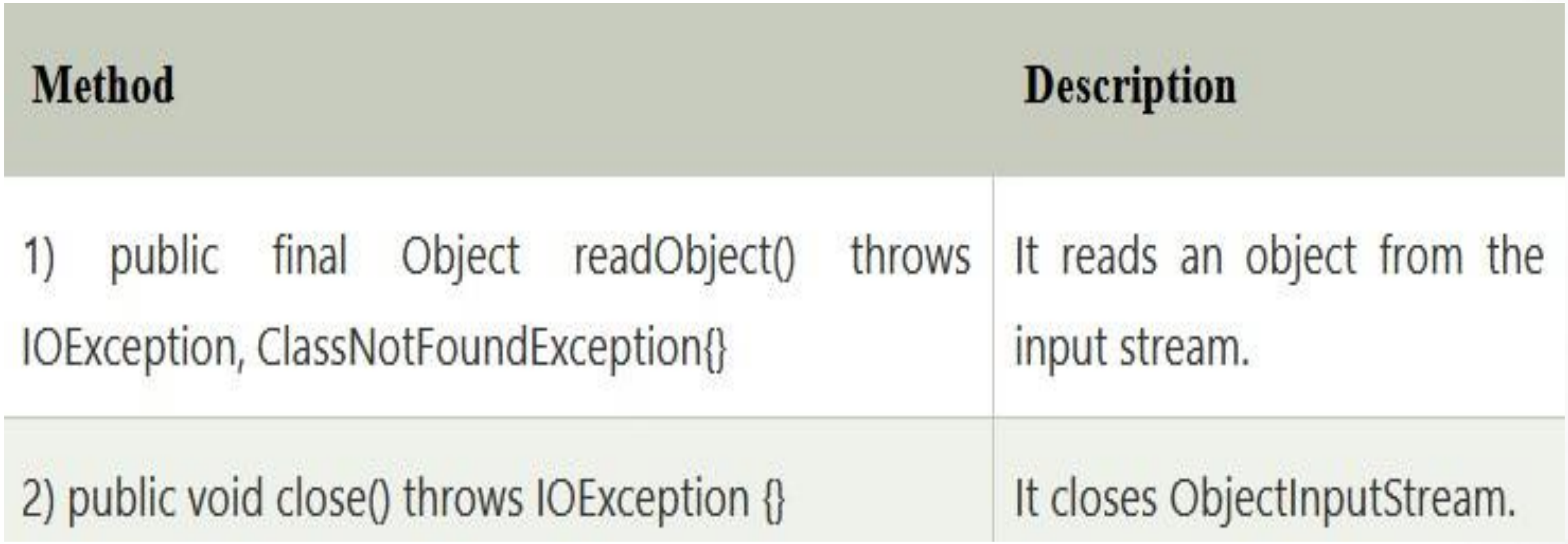


Method	Description
1) public final Object readObject() throws IOException, ClassNotFoundException{}	It reads an object from the input stream.
2) public void close() throws IOException {}	It closes ObjectInputStream.

Serialization and Deserialization

ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.



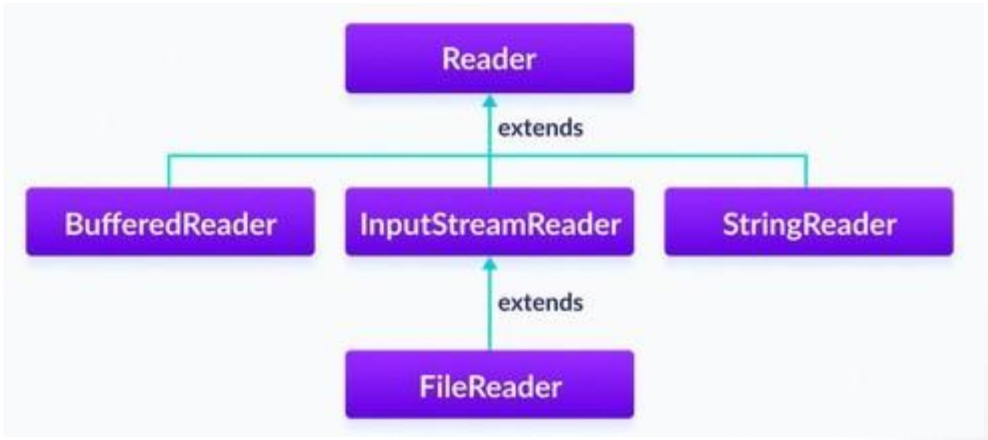
Method	Description
1) public final Object readObject() throws IOException, ClassNotFoundException{}	It reads an object from the input stream.
2) public void close() throws IOException {}	It closes ObjectInputStream.

```
import java.io.*;
class Persist{
 public static void main(String args[]){
  try{
  //Creating the object
  Student s1 =new Student(211,"ravi");
  //Creating stream and writing the object
  FileOutputStream fout=new FileOutputStream("f.txt");
  ObjectOutputStream out=new ObjectOutputStream(fout);

  out.writeObject(s1);
  out.flush();
  //closing the stream
  out.close();
  System.out.println("success");
  }catch(Exception e){System.out.println(e);}
 }
}
```

```
import java.io.*;
class Depersist{
 public static void main(String args[]){
  try{
  //Creating stream to read the object
  ObjectInputStream in=new ObjectInputStream(new FileInput
Stream("f.txt"));
  Student s=(Student)in.readObject();
  //printing the data of the serialized object
  System.out.println(s.id+" "+s.name);
  //closing the stream
  in.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```

Java Reader Class

- The Reader class of the java.io package is an abstract superclass that represents a stream of characters.
- Since Reader is an abstract class, it is not useful by itself. However, its subclasses can be used to read data.
- Subclasses of Reader
- In order to use the functionality of Reader, we can use its subclasses. Some of them are:
- BufferedReader
- InputStreamReader
- FileReader
- StringReader



Java Writer Class

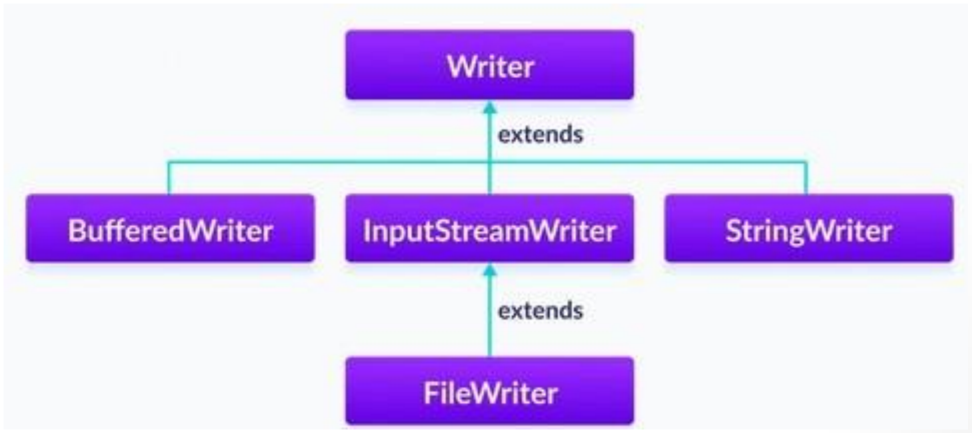
- The Writer class of the java.io package is an abstract superclass that represents a stream of characters.

- Since Writer is an abstract class, it is not useful by itself. However, its subclasses can be used to write data.

- Subclasses of Writer

- In order to use the functionality of the Writer, we can use its subclasses. Some of them are:

- BufferedWriter
- OutputStreamWriter
- FileWriter
- StringWriter



Java InputStreamReader Class

- The InputStreamReader class of the java.io package can be used to convert data in bytes into data in characters.

- It extends the abstract class Reader.

- The InputStreamReader class works with other input streams. It is also known as a bridge between byte streams and character streams. This is because the InputStreamReader reads bytes from the input stream as characters.

- For example, some characters required 2 bytes to be stored in the storage. To read such data we can use the input stream reader that reads the 2 bytes together and converts into the corresponding character.

Java InputStreamReader Class

- Create an InputStreamReader

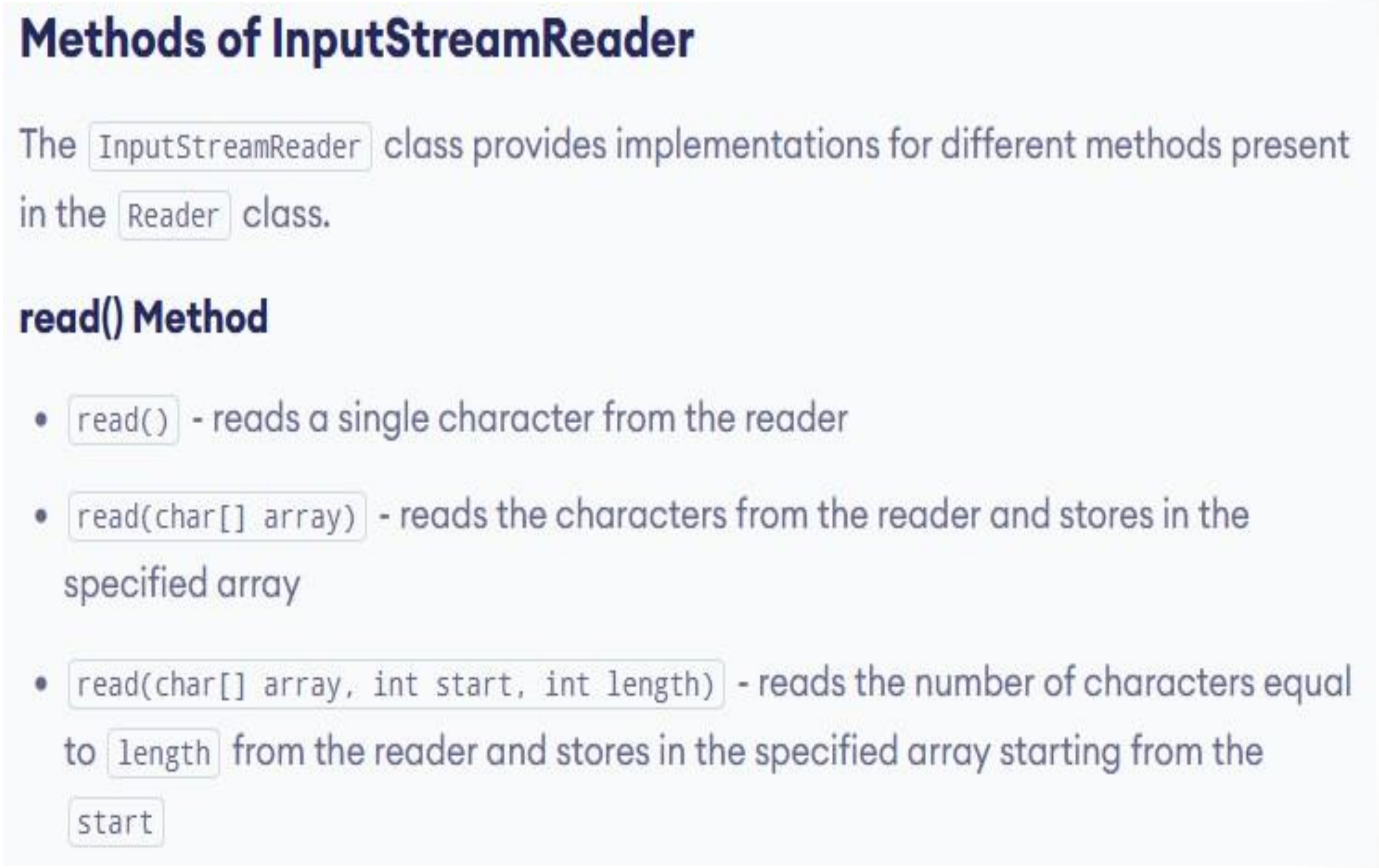
- In order to create an InputStreamReader, we must import the java.io.InputStreamReader package first. Once we import the package here is how we can create the input stream reader.

- // Creates an InputStream
- FileInputStream file = new FileInputStream(String path);

- // Creates an InputStreamReader
- InputStreamReader input = new InputStreamReader(file);

- In the above example, we have created an InputStreamReader named input along with the FileInputStream named file.

Java InputStreamReader Class



Methods of InputStreamReader

The `InputStreamReader` class provides implementations for different methods present in the `Reader` class.

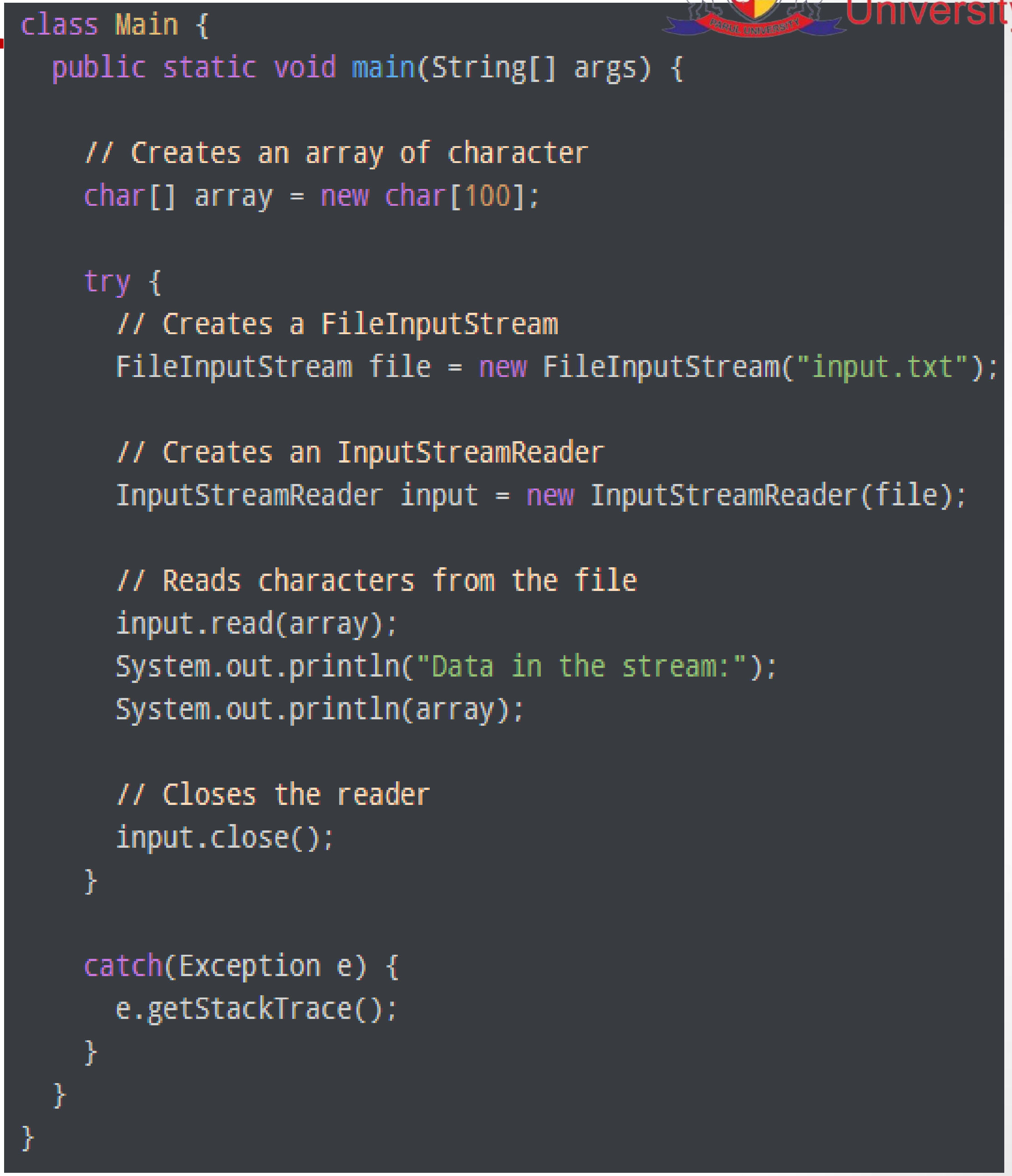
read() Method

- `read()` - reads a single character from the reader
- `read(char[] array)` - reads the characters from the reader and stores in the specified array
- `read(char[] array, int start, int length)` - reads the number of characters equal to `length` from the reader and stores in the specified array starting from the `start`

Java InputStr

- For example, suppose we have a file named input.txt with the following content.

- This is a line of text inside the file.



```java
class Main {
  public static void main(String[] args) {

    // Creates an array of character
    char[] array = new char[100];

    try {
      // Creates a FileInputStream
      FileInputStream file = new FileInputStream("input.txt");

      // Creates an InputStreamReader
      InputStreamReader input = new InputStreamReader(file);

      // Reads characters from the file
      input.read(array);
      System.out.println("Data in the stream:");
      System.out.println(array);

      // Closes the reader
      input.close();
    }

    catch(Exception e) {
      e.getStackTrace();
    }
  }
}
```

Java OutputStreamWriter Class

- The OutputStreamWriter class of the java.io package can be used to convert data in character form into data in bytes form.

- It extends the abstract class Writer.

- The OutputStreamWriter class works with other output streams. It is also known as a bridge between byte streams and character streams. This is because the OutputStreamWriter converts its characters into bytes.

- For example, some characters require 2 bytes to be stored in the storage. To write such data we can use the output stream writer that converts the character into corresponding bytes and stores the bytes together.

Java OutputStreamWriter Class

- Create an OutputStreamWriter

- In order to create an OutputStreamWriter, we must import the java.io.OutputStreamWriter package first. Once we import the package here is how we can create the output stream writer.

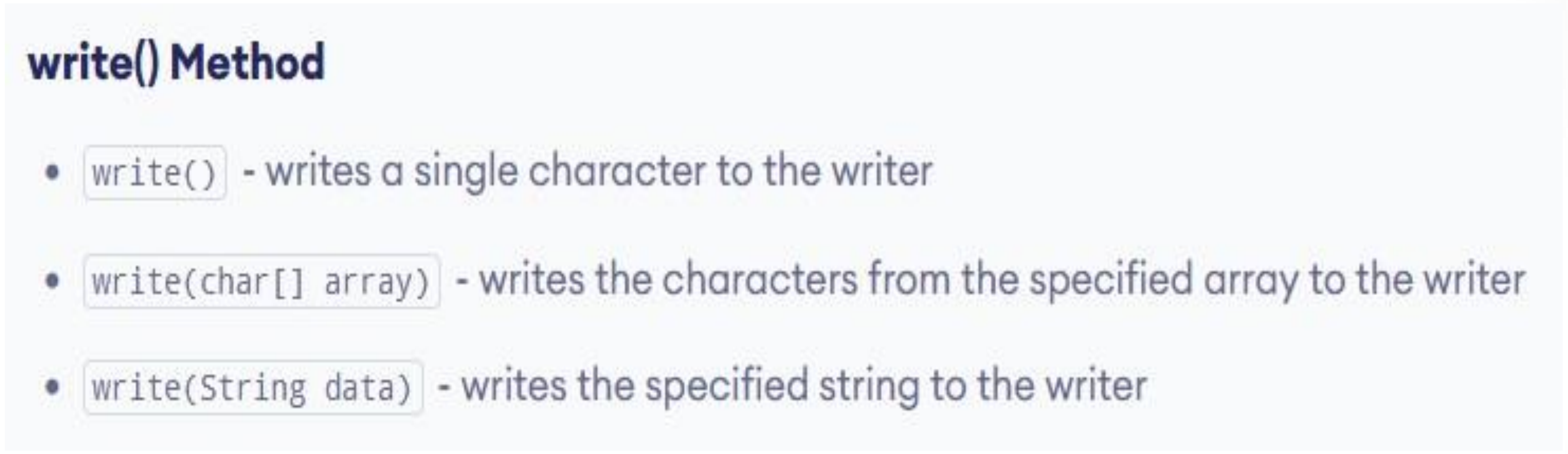
- // Creates an OutputStream
- FileOutputStream file = new FileOutputStream(String path);

- // Creates an OutputStreamWriter
- OutputStreamWriter output = new OutputStreamWriter(file);

- In the above example, we have created an OutputStreamWriter named output along with the FileOutputStream named file.

Java OutputStreamWriter Class

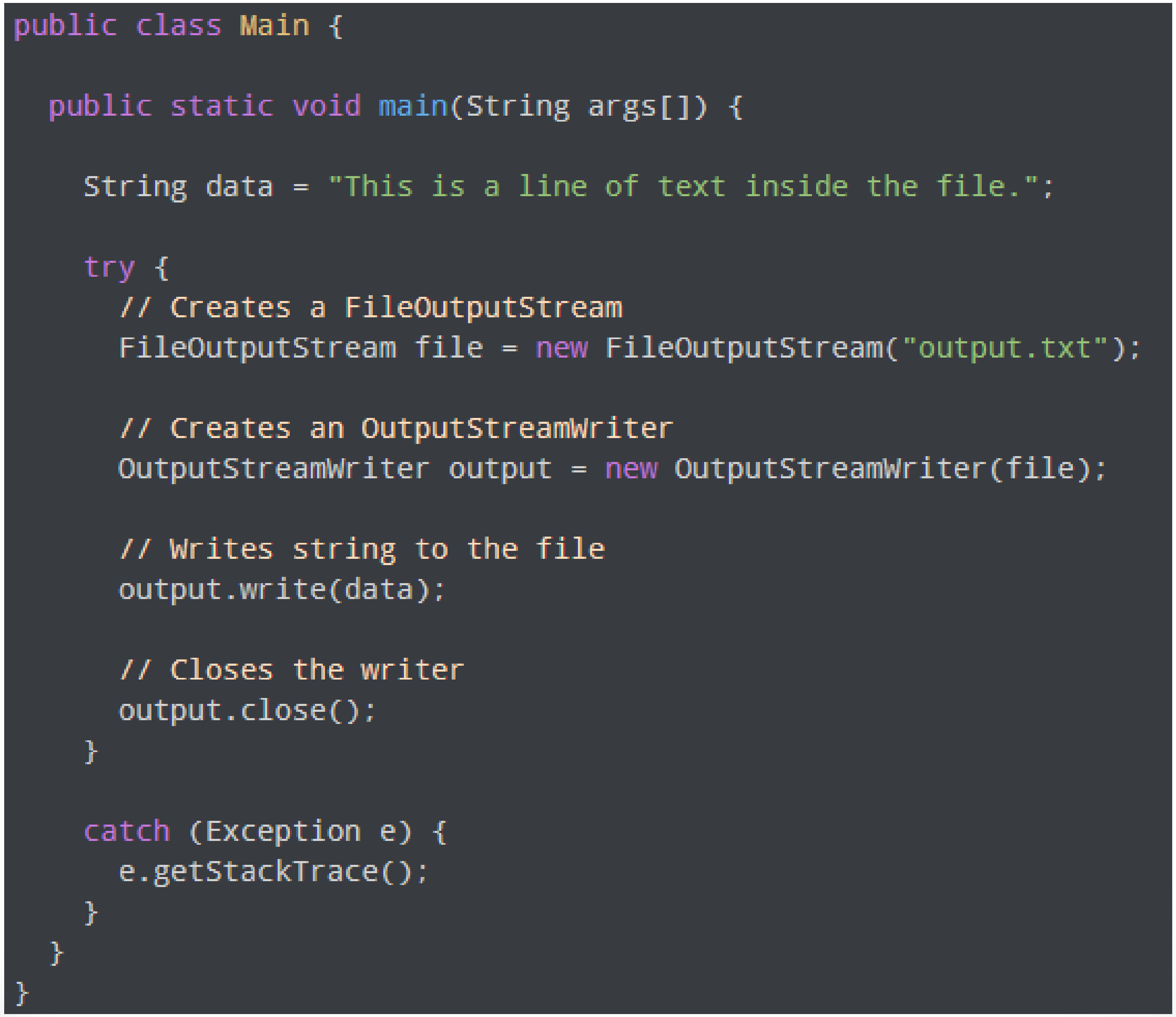
- **Methods of OutputStreamWriter**



write() Method

- `write()` - writes a single character to the writer
- `write(char[] array)` - writes the characters from the specified array to the writer
- `write(String data)` - writes the specified string to the writer

Java OutputStreamWriter Class

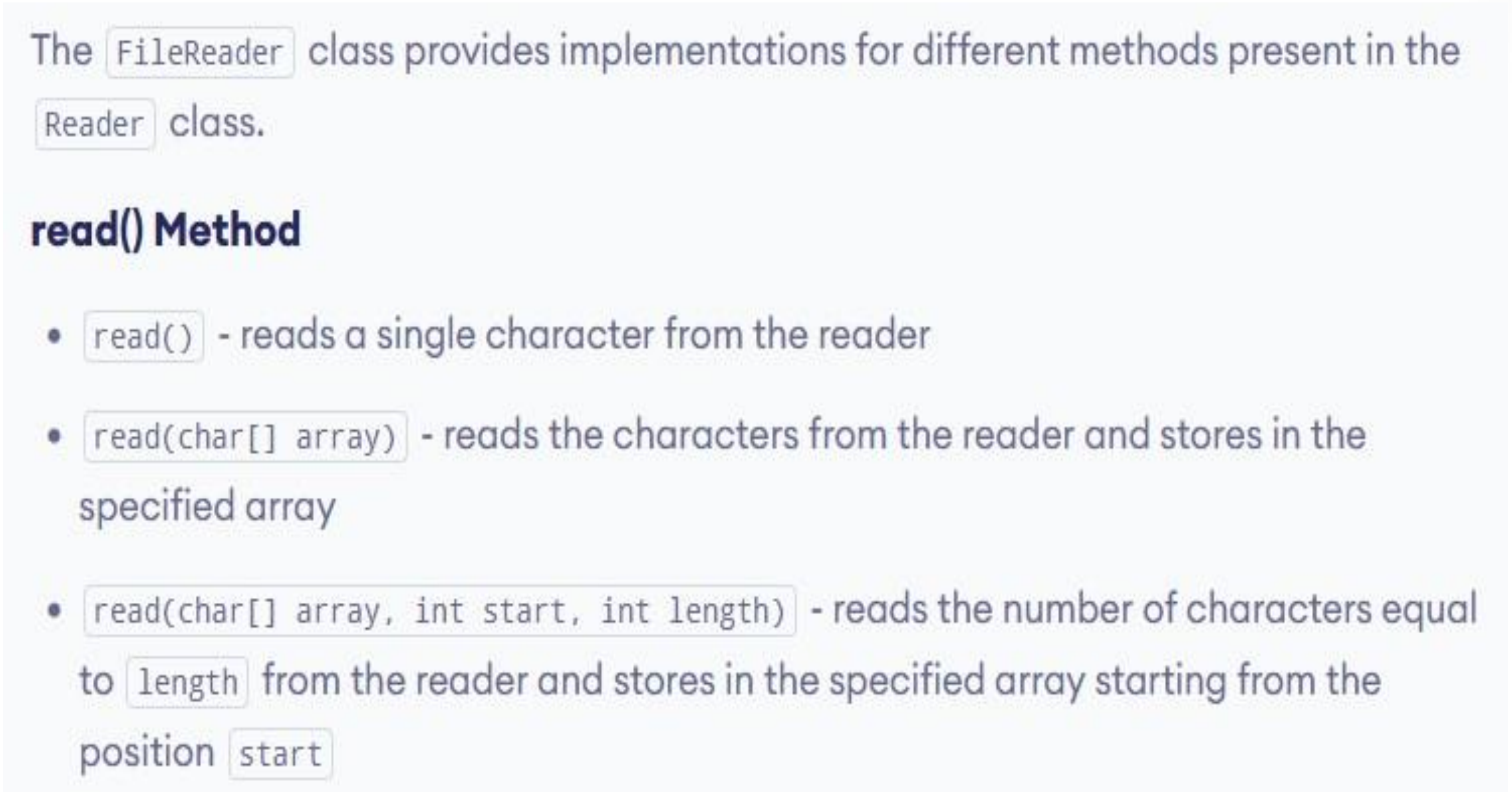


```java
public class Main {

  public static void main(String args[]) {

    String data = "This is a line of text inside the file.";

    try {
      // Creates a FileOutputStream
      FileOutputStream file = new FileOutputStream("output.txt");

      // Creates an OutputStreamWriter
      OutputStreamWriter output = new OutputStreamWriter(file);

      // Writes string to the file
      output.write(data);

      // Closes the writer
      output.close();
    }

    catch (Exception e) {
      e.getStackTrace();
    }
  }
}
```

Java FileReader Class

- The FileReader class of the java.io package can be used to read data (in characters) from files.
- It extends the InputSreamReader class.
- Create a FileReader
- In order to create a file reader, we must import the java.io.FileReader package first. Once we import the package, here is how we can create the file reader.
- Using the name of the file
- FileReader input = new FileReader(String name);
- Here, we have created a file reader that will be linked to the file specified by the name

Java FileReader Class

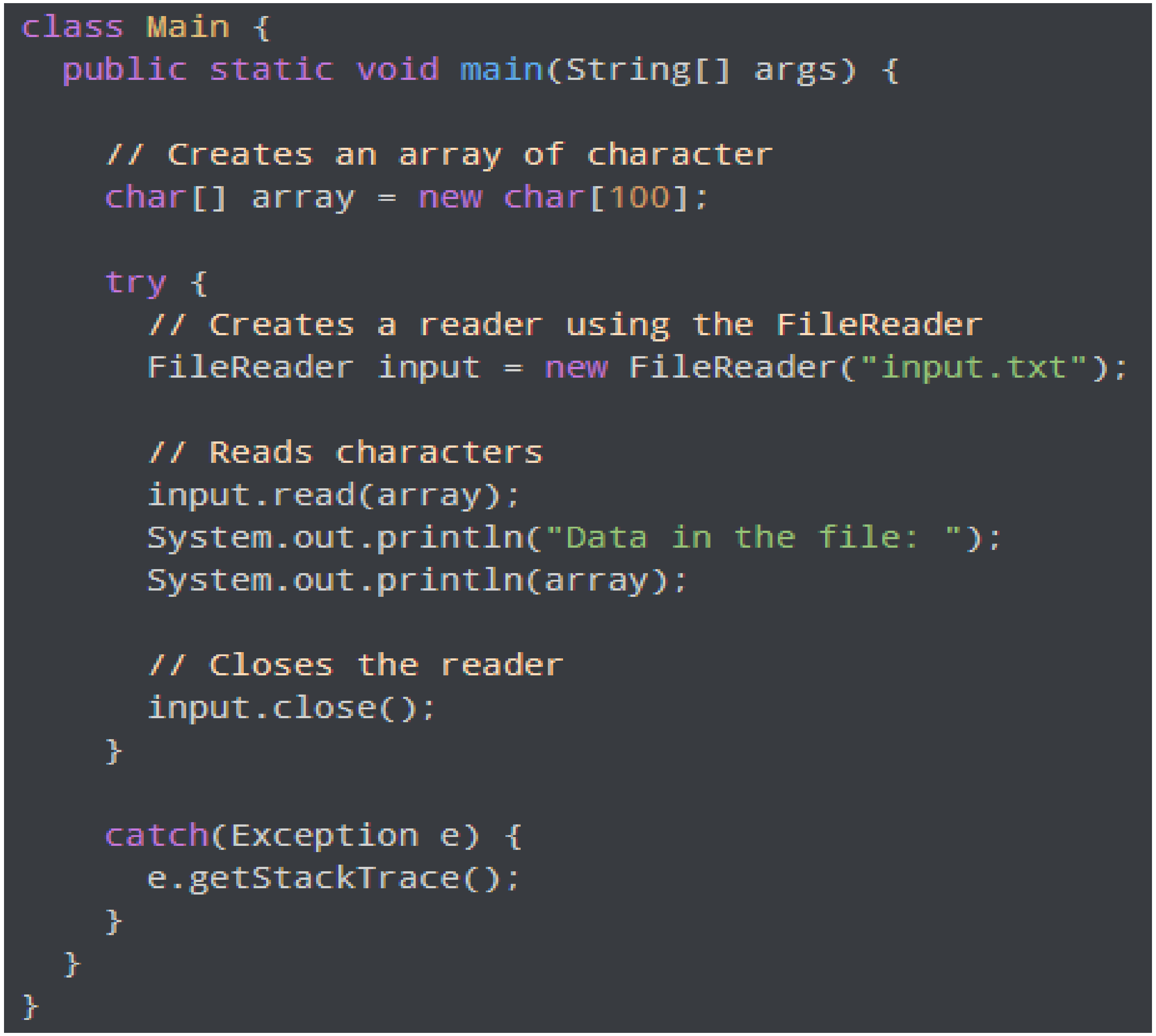


The `FileReader` class provides implementations for different methods present in the `Reader` class.

read() Method

- `read()` - reads a single character from the reader
- `read(char[] array)` - reads the characters from the reader and stores in the specified array
- `read(char[] array, int start, int length)` - reads the number of characters equal to `length` from the reader and stores in the specified array starting from the position `start`

Java FileReader Class

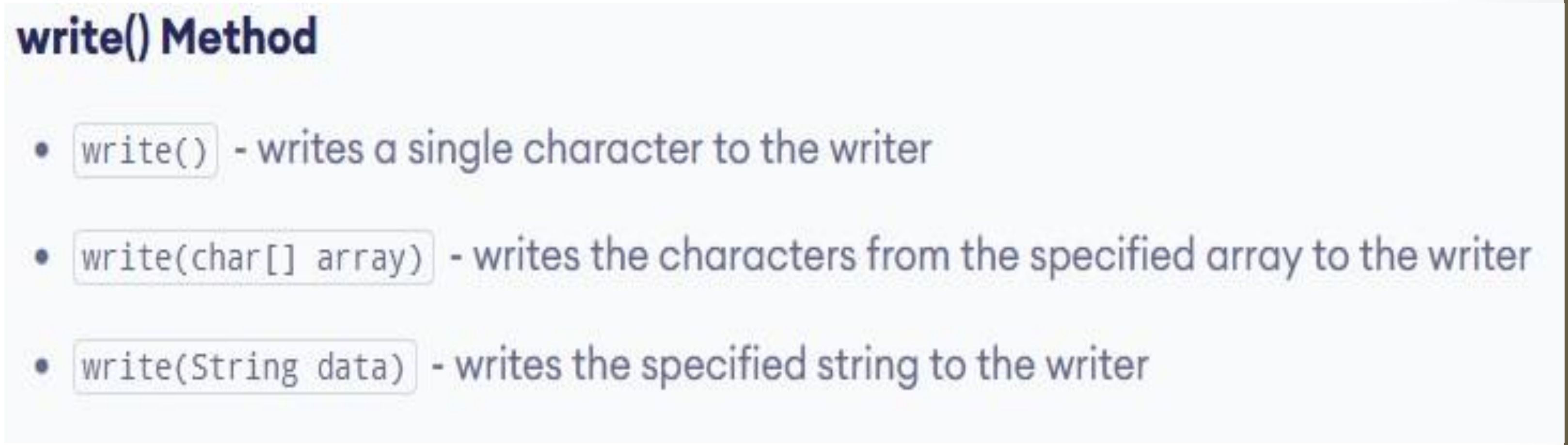


```java
class Main {
  public static void main(String[] args) {

    // Creates an array of character
    char[] array = new char[100];

    try {
      // Creates a reader using the FileReader
      FileReader input = new FileReader("input.txt");

      // Reads characters
      input.read(array);
      System.out.println("Data in the file: ");
      System.out.println(array);

      // Closes the reader
      input.close();
    }

    catch(Exception e) {
      e.getStackTrace();
    }
  }
}
```

Java FileWriter Class

- The FileWriter class of the java.io package can be used to write data (in characters) to files.
- It extends the OutputStreamWriter class.
- Create a FileWriter
- In order to create a file writer, we must import the Java.io.FileWriter package first. Once we import the package, here is how we can create the file writer.
- Using the name of the file
- FileWriter output = new FileWriter(String name);

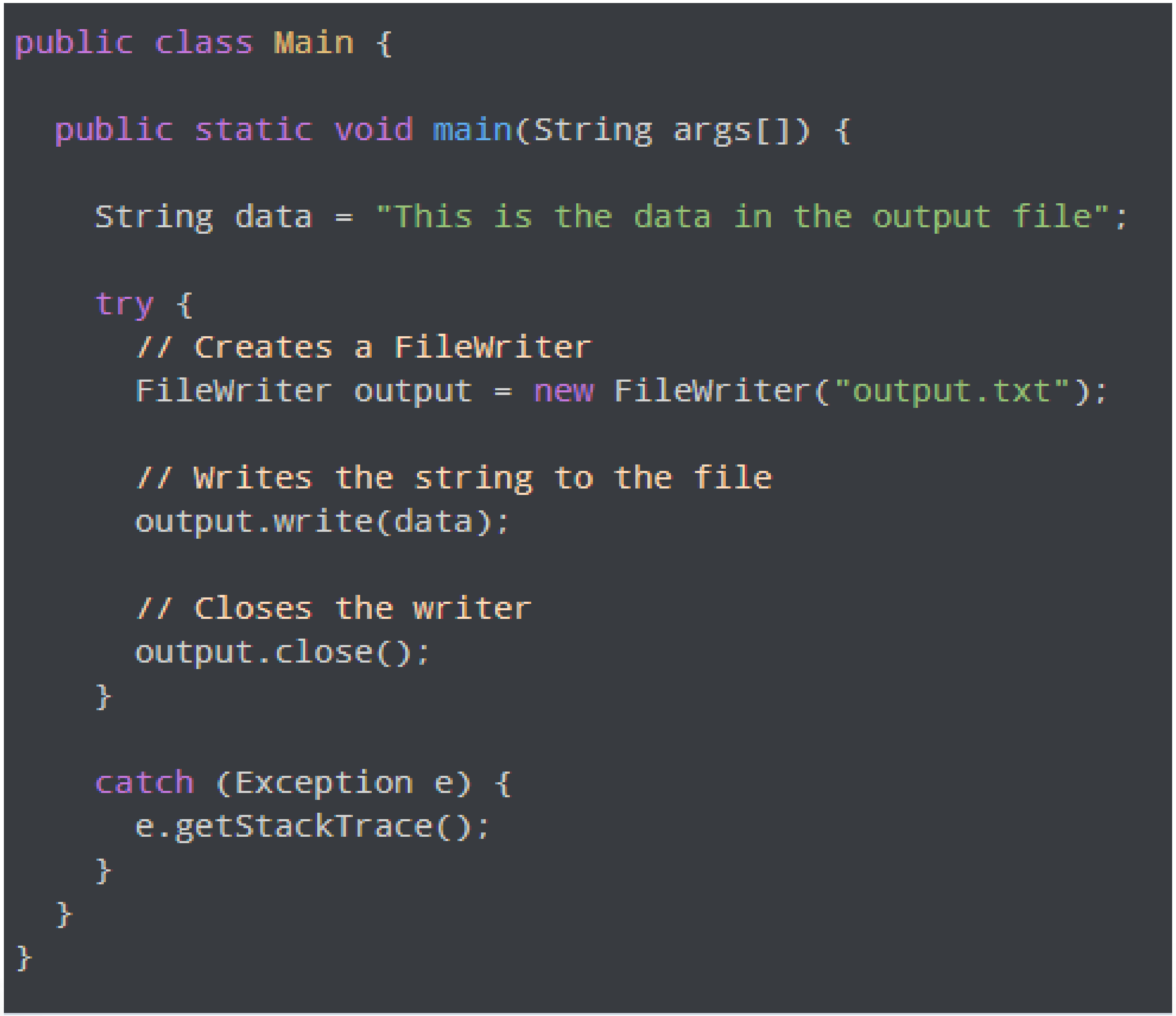
Java FileWriter Class



write() Method

- `write()` - writes a single character to the writer
- `write(char[] array)` - writes the characters from the specified array to the writer
- `write(String data)` - writes the specified string to the writer

Java FileWriter Class



```java
public class Main {

  public static void main(String args[]) {

    String data = "This is the data in the output file";

    try {
      // Creates a FileWriter
      FileWriter output = new FileWriter("output.txt");

      // Writes the string to the file
      output.write(data);

      // Closes the writer
      output.close();
    }

    catch (Exception e) {
      e.getStackTrace();
    }
  }
}
```

Java BufferedReader

- The BufferedReader class of the java.io package can be used with other readers to read data (in characters) more efficiently.
- It extends the abstract class Reader.
- Working of BufferedReader
- The BufferedReader maintains an internal buffer of 8192 characters.
- During the read operation in BufferedReader, a chunk of characters is read from the disk and stored in the internal buffer. And from the internal buffer characters are read individually.
- Hence, the number of communication to the disk is reduced. This is why reading characters is faster using BufferedReader.

Java BufferedReader

- Create a BufferedReader

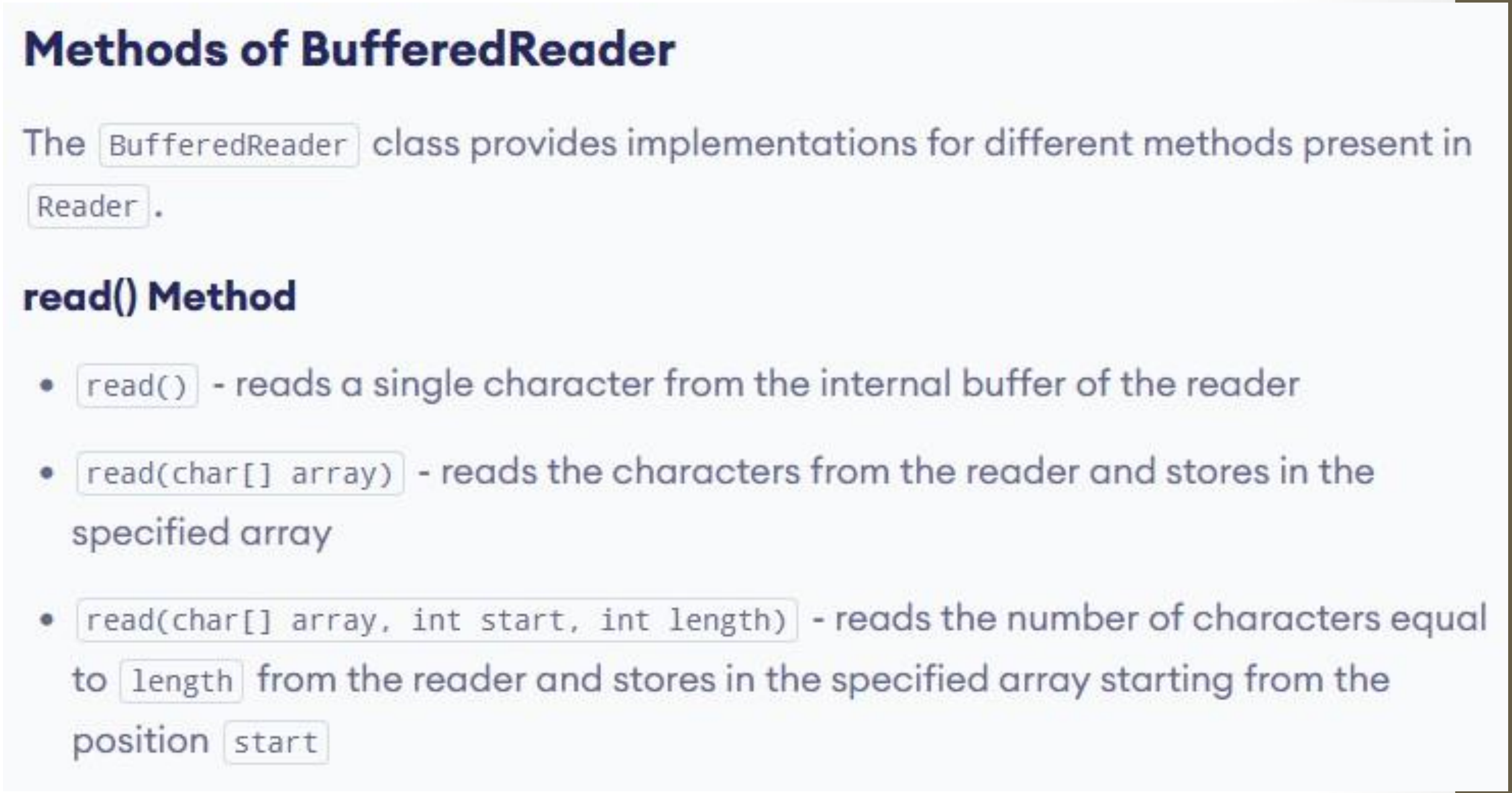
- In order to create a BufferedReader, we must import the java.io.BuferedReader package first. Once we import the package, here is how we can create the reader.

- // Creates a FileReader
- FileReader file = new FileReader(String file);

- // Creates a BufferedReader
- BufferedReader buffer = new BufferedReader(file);

- In the above example, we have created a BufferedReader named buffer with the FileReader named file.

Java BufferedReader



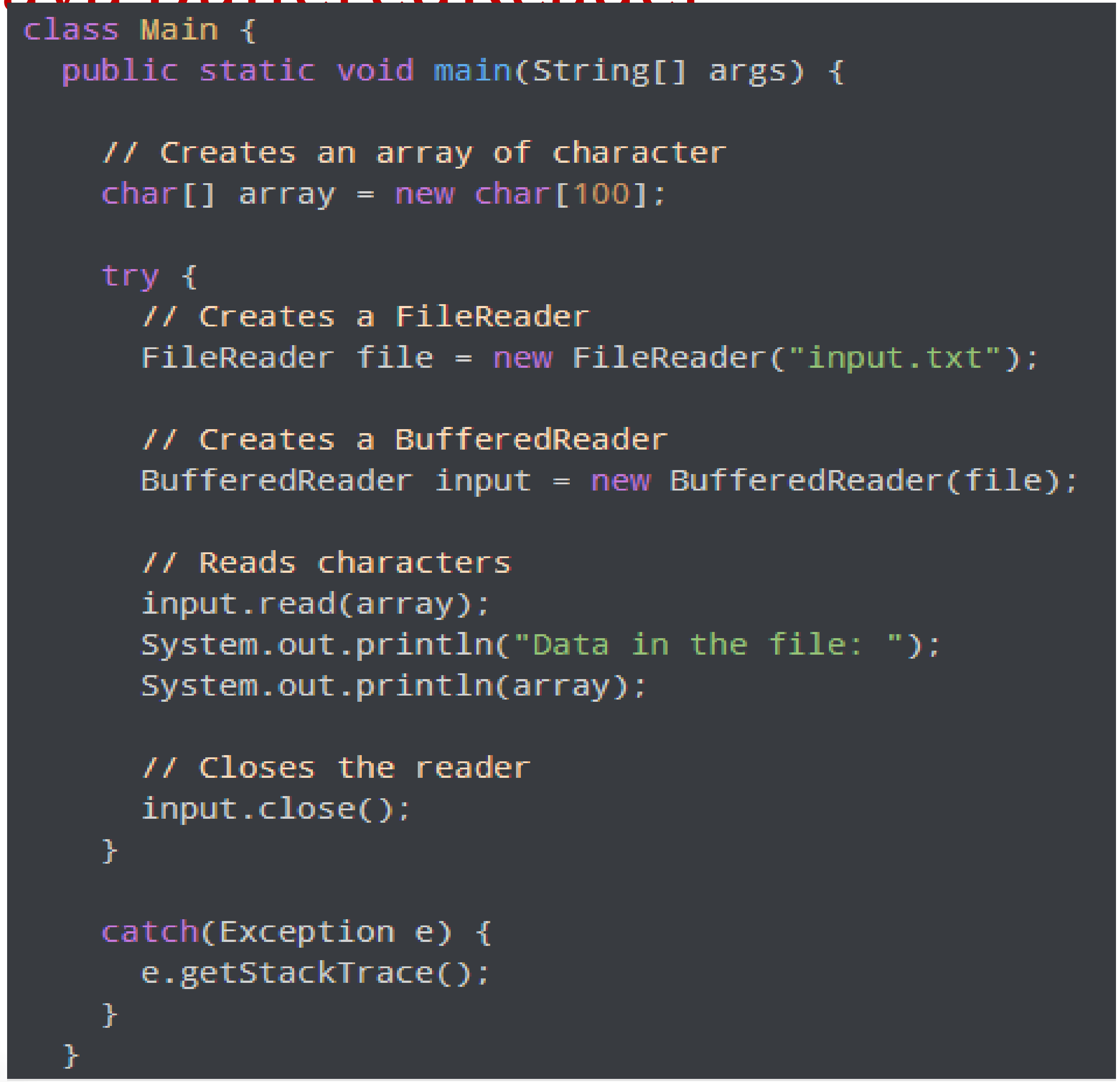
Methods of BufferedReader

The `BufferedReader` class provides implementations for different methods present in `Reader`.

read() Method

- `read()` - reads a single character from the internal buffer of the reader
- `read(char[] array)` - reads the characters from the reader and stores in the specified array
- `read(char[] array, int start, int length)` - reads the number of characters equal to `length` from the reader and stores in the specified array starting from the position `start`

Java BufferedReader



```java
class Main {
  public static void main(String[] args) {

    // Creates an array of character
    char[] array = new char[100];

    try {
      // Creates a FileReader
      FileReader file = new FileReader("input.txt");

      // Creates a BufferedReader
      BufferedReader input = new BufferedReader(file);

      // Reads characters
      input.read(array);
      System.out.println("Data in the file: ");
      System.out.println(array);

      // Closes the reader
      input.close();
    }

    catch(Exception e) {
      e.getStackTrace();
    }
  }
```

Java BufferedWriter Class

- The BufferedWriter class of the java.io package can be used with other writers to write data (in characters) more efficiently.
- It extends the abstract class Writer.
- Working of BufferedWriter
- The BufferedWriter maintains an internal buffer of 8192 characters.
- During the write operation, the characters are written to the internal buffer instead of the disk. Once the buffer is filled or the writer is closed, the whole characters in the buffer are written to the disk.
- Hence, the number of communication to the disk is reduced. This is why writing characters is faster using BufferedWriter.

Java BufferedWriter Class

- Create a BufferedWriter

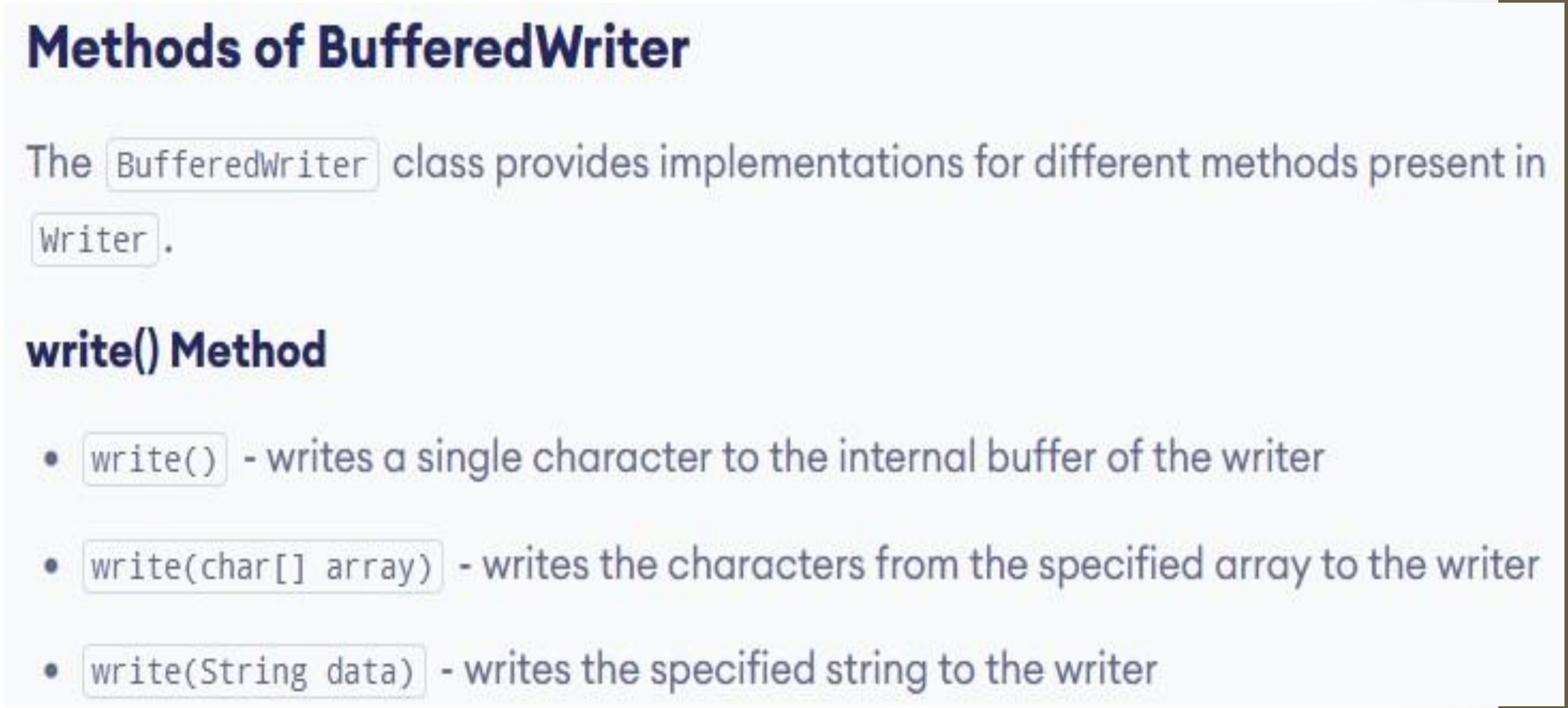
- In order to create a BufferedWriter, we must import the java.io.BufferedWriter package first. Once we import the package here is how we can create the buffered writer.

- // Creates a FileWriter
- FileWriter file = new FileWriter(String name);

- // Creates a BufferedWriter
- BufferedWriter buffer = new BufferedWriter(file);

- In the above example, we have created a BufferedWriter named buffer with the FileWriter named file.

Java BufferedWriter Class



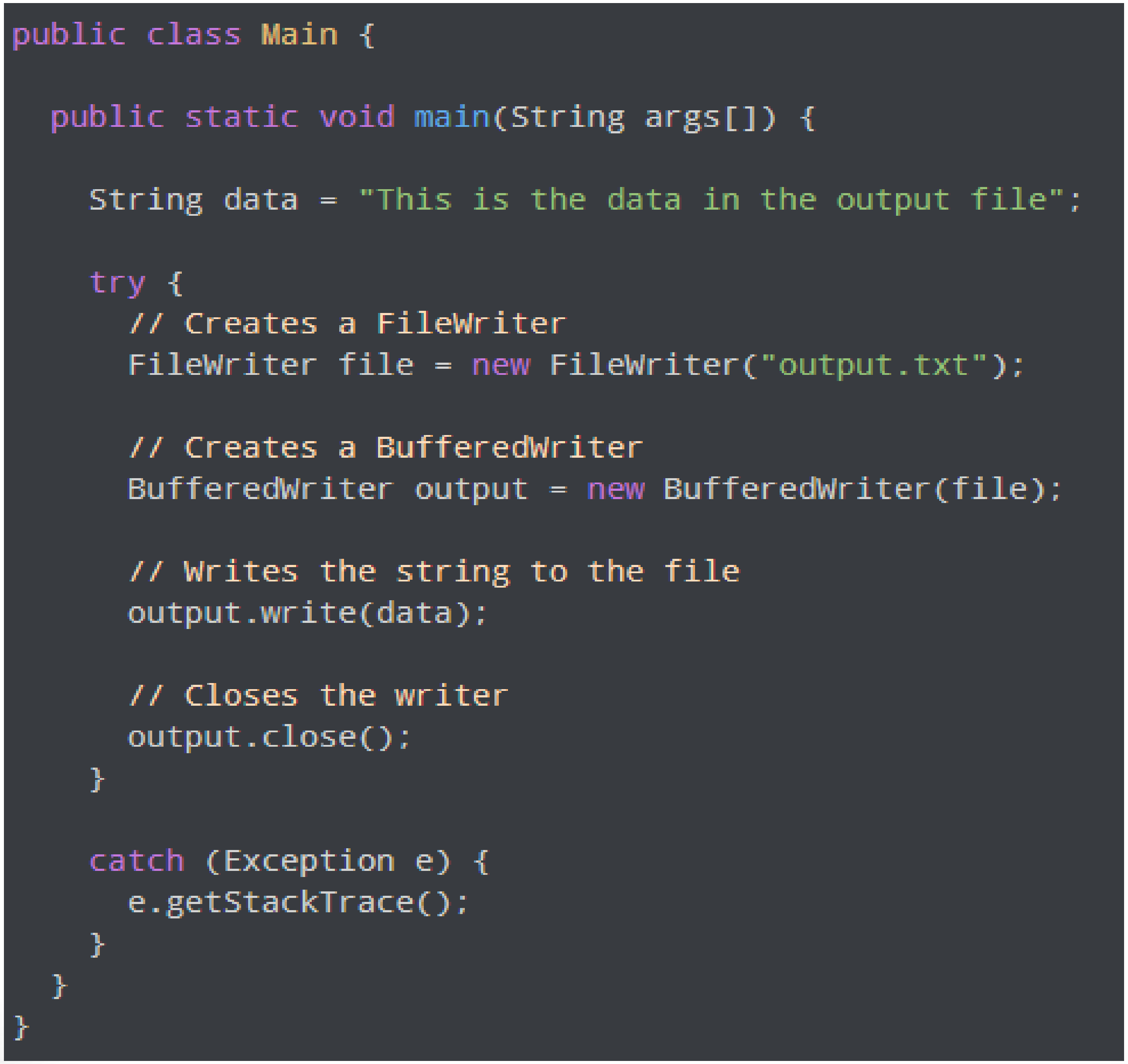
Methods of BufferedWriter

The `BufferedWriter` class provides implementations for different methods present in `Writer`.

write() Method

- `write()` - writes a single character to the internal buffer of the writer
- `write(char[] array)` - writes the characters from the specified array to the writer
- `write(String data)` - writes the specified string to the writer

Java BufferedWriter Class



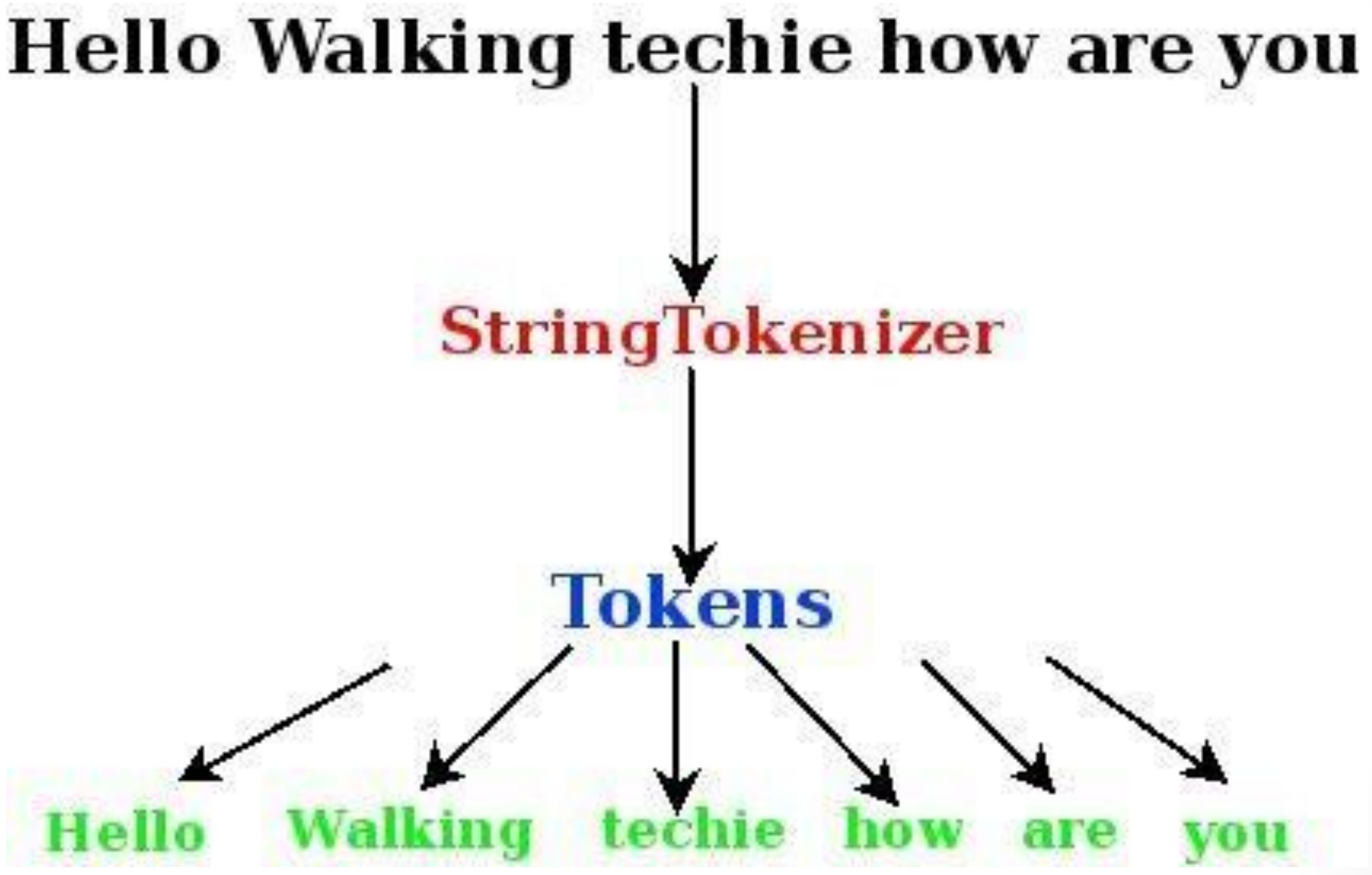
```java
public class Main {

  public static void main(String args[]) {

    String data = "This is the data in the output file";

    try {
      // Creates a FileWriter
      FileWriter file = new FileWriter("output.txt");

      // Creates a BufferedWriter
      BufferedWriter output = new BufferedWriter(file);

      // Writes the string to the file
      output.write(data);

      // Closes the writer
      output.close();
    }

    catch (Exception e) {
      e.getStackTrace();
    }
  }
}
```

StringTokenizer

The **java.util.StringTokenizer** class allows you to break a String into tokens. It is simple way to break a String.

In the StringTokenizer class, the delimiters can be provided at the time of creation or one by one to the tokens.

StringTokenizer



StringTokenizer

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.

StringTokenizer

```
import java.util.StringTokenizer;
public class Simple{
 public static void main(String args[]){
   StringTokenizer st = new StringTokenizer("my name is 
vivek"," ");

   while (st.hasMoreTokens()) {
         System.out.println(st.nextToken());
     }
   }
}
```

StringTokenizer

```
import java.util.*;

public class Test {
   public static void main(String[] args) {
        StringTokenizer st = new StringTokenizer("my,name,is,vivek
");


      // printing next token
      System.out.println("Next token is : " + st.nextToken(","));
   }
}
```