Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 2

Lecture 13

Lecture 13

- **The Idea behind Exception**
- **Exceptions & Errors**
- **Types of Exception, Checked and Un-Checked Exceptions**

Exception Handling

Exception Handling in Java is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved.

Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Control Flow in Exceptions

The core advantage of exception handling is to maintain the normal flow of the application.

An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions.

Let's consider a scenario

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

The Idea behind Exception

- Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions.
- Exceptions can be caught and handled by the program.
- When an exception occurs within a method, it creates an object.

- This object is called the exception object.
- It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

Major reasons why an exception Occurs

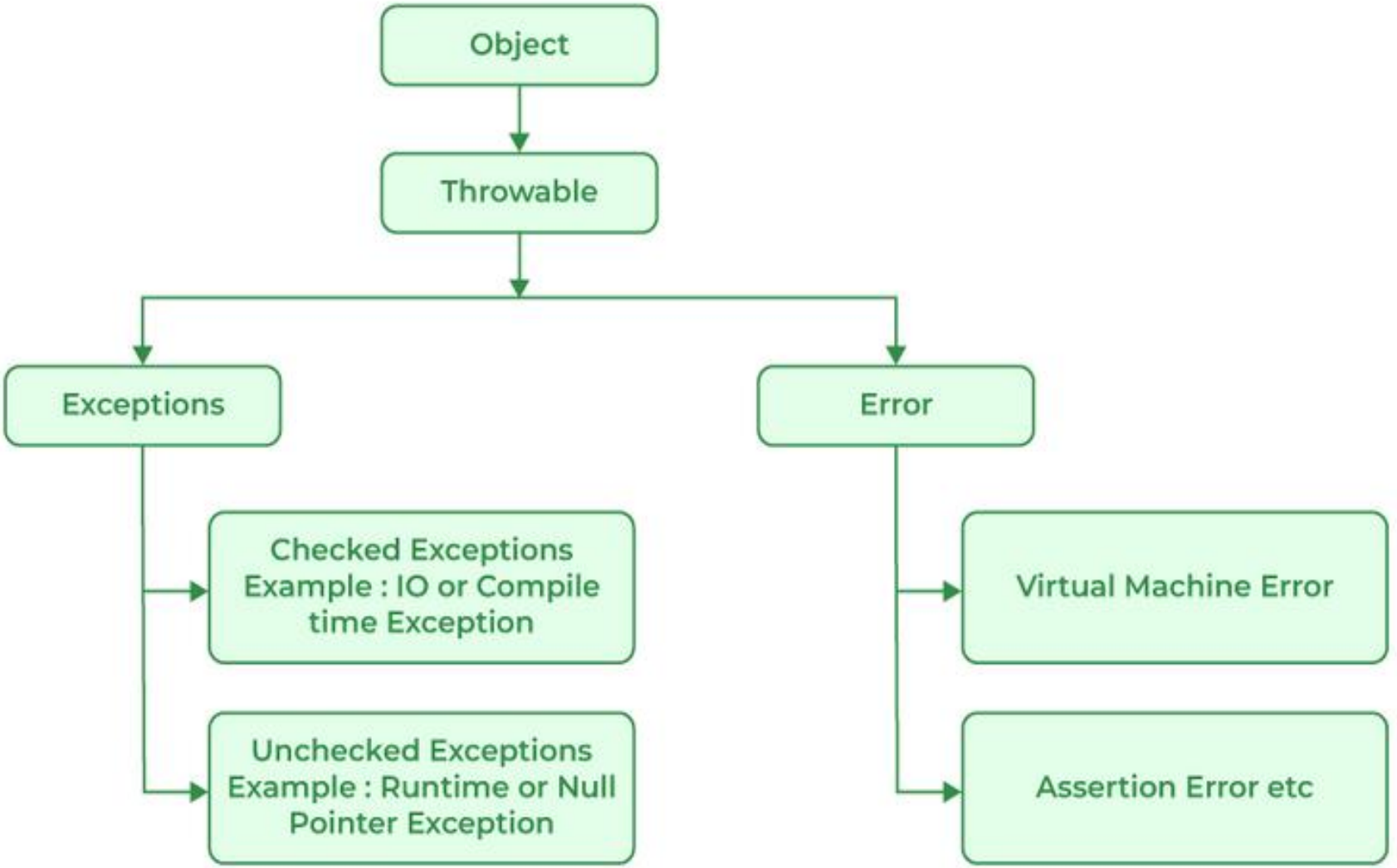
- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out-of-disk memory)
- Code errors
- Opening an unavailable file

Errors

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.

Errors are usually beyond the control of the programmer, and we should not try to handle errors.

Exception Hierarchy

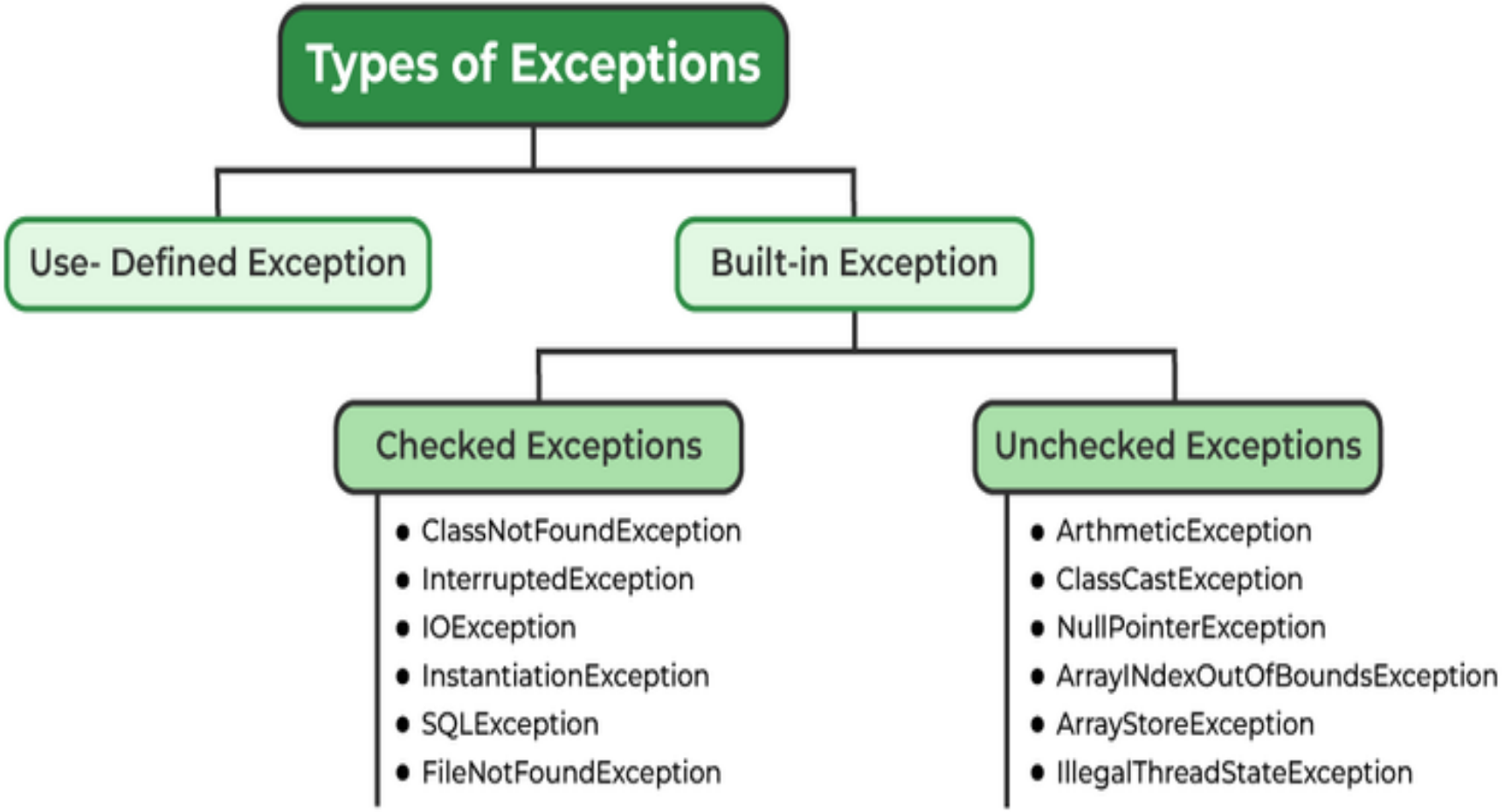


Difference between Error and Exception

Error: An Error indicates a serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch.

Types of Exceptions



Exceptions can be categorized in two ways:

1. **Built-in Exceptions**
   - Checked Exception
   - Unchecked Exception
2. **User-Defined Exceptions**

1. Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

Checked Exceptions: Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

Unchecked Exceptions:

The unchecked exceptions are just opposite to the checked exceptions.

The compiler will not check these exceptions at compile time.

In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

2. User-Defined Exceptions:

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called ‘user-defined Exceptions’.

The advantages of Exception Handling in Java are as follows:

- Provision to Complete Program Execution
- Easy Identification of Program Code and Error-Handling Code
- Propagation of Errors
- Meaningful Error Reporting
- Identifying Error Types

Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 2

Lecture 14

Lecture 14

- Control Flow in Exceptions
- JVM Reaction to Exceptions
- Use of try, catch and finally

Flow control in try catch finally in Java

1. **Control flow in try-catch clause OR try-catch-finally clause**
    1. **Case 1:** Exception occurs in try block and handled in catch block
    2. **Case 2:** Exception occurs in try-block is not handled in catch block
    3. **Case 3:** Exception doesn't occur in try-block
2. **try-finally clause**
    1. **Case 1:** Exception occurs in try block
    2. **Case 2:** Exception doesn't occur in try-block

Exception occurs in try block and handled in catch block

If a statement in try block raised an exception, then the rest of the try block doesn't execute and control passes to the **corresponding** catch block.

After executing the catch block, the control will be transferred to finally block(if present) and then the rest program will be executed.

Exception occurred in try-block is not handled in catch block

In this case, the default handling mechanism is followed.

If finally block is present, it will be executed followed by the default handling mechanism.

Exception doesn't occur in try-block:

In this case catch block never runs as they are only meant to be run when an exception occurs. finally block(if present) will be executed followed by rest of the program.

```
class A
{
    public static void main (String[] args)
    {
    try
    {
      String str = "123";
       int num = Integer.parseInt(str);
       System.out.println("try block fully executed");
     }
    catch(NumberFormatException ex)
    {
     System.out.println("catch block executed...");
     }
     finally
    {
        System.out.println("finally block executed");
    }
   System.out.println("Outside try-catch-finally clause");
    }
}
```

Control flow in try-finally

In this case, no matter whether an exception occurs in try-block or not finally will always be executed. But control flow will depend on whether an exception has occurred in the try block or not.

1. Exception raised: If an exception has occurred in the try block then the control flow will be finally block followed by the default exception handling mechanism.

```
class A
{
    public static void main (String[] args)
    {
       int[] arr = new int[4];
        try
        {
            int i = arr[4];
            System.out.println("Inside try block");
        }
        finally
        {
            System.out.println("finally block executed");
        }
        // rest program will not execute
        System.out.println("Outside try-finally clause");
    }
}
```

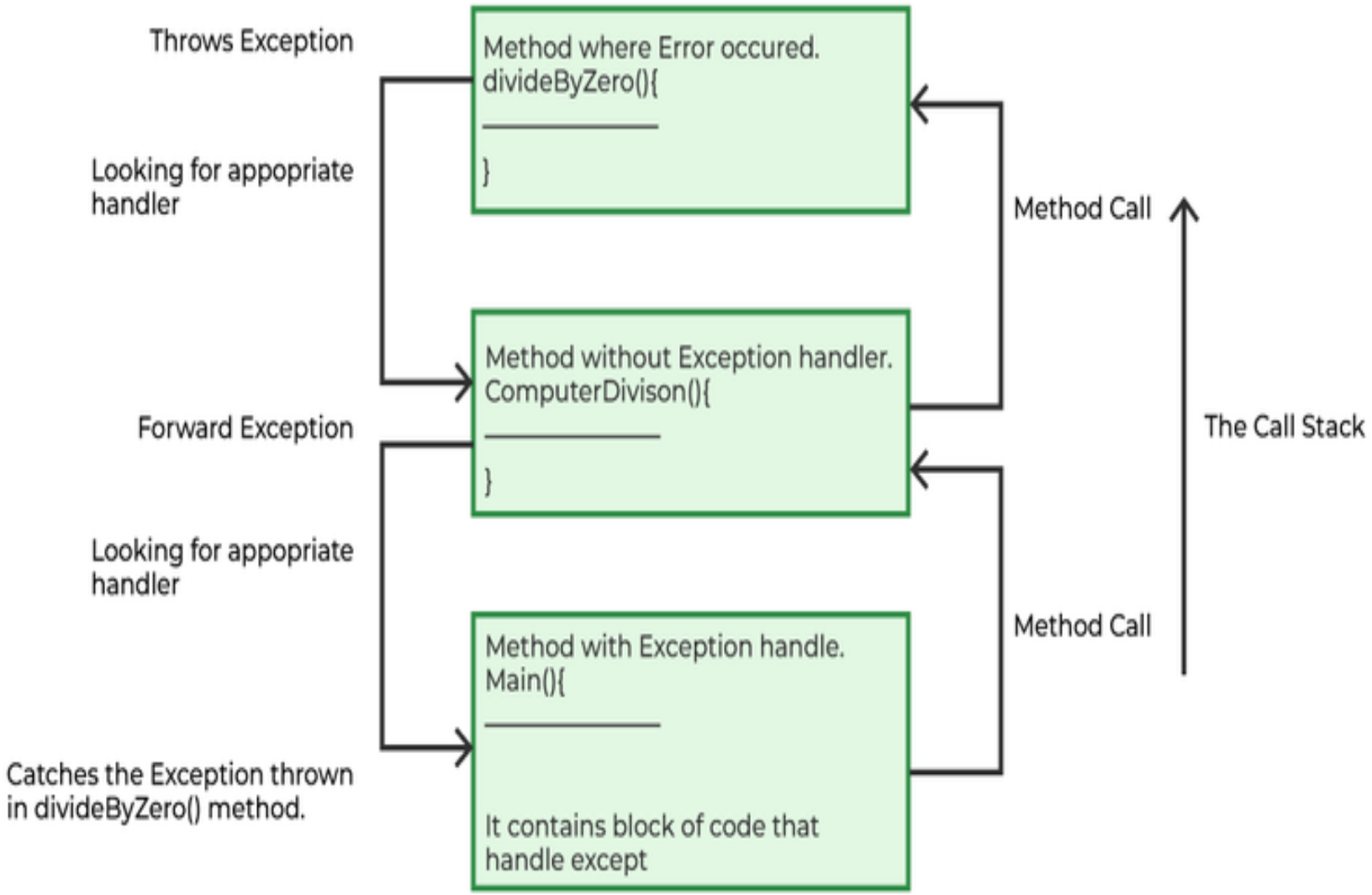
Exception not raised:

If an exception does not occur in the try block then the control flow will be finally block followed by the rest of the program

Default Exception Handling

- Whenever inside a method, if an exception has occurred, the method creates an Object known as an Exception Object and hands it off to the run-time system(JVM).
- The exception object contains the name and description of the exception and the current state of the program where the exception has occurred.
- Creating the Exception Object and handling it in the run-time system is called throwing an Exception.

There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of methods is called Call Stack.



The Call Stack and searching the call stack for exception handler.

- The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an Exception handler.
- The run-time system starts searching from the method in which the exception occurred and proceeds through the call stack in the reverse order in which methods were called.

.

- If it finds an appropriate handler, then it passes the occurred exception to it. An appropriate handler means the type of exception object thrown matches the type of exception object it can handle.

.

- If the run-time system searches all the methods on the call stack and couldn't have found the appropriate handler, then the run-time system handover the Exception Object to the **default exception handler**, which is part of the run-time system.
- This handler prints the exception information in the following format and terminates the program **abnormally**

Exception in thread "xxx" Name of Exception : Description

... . // Call Stack

Java Exception Keywords

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.

```
class example {
   public static void main (String args[]) {
      int num1 = 15, num2 = 0, result = 0;
      try{
          result = num1/num2;
          System.out.println("The result is" +result);
      }
      catch (ArithmeticException e) {
        System.out.println ("Can't be divided by Zero " + e);
      }
   }
}
```

Java Multi-catch block

A try block can be followed by one or more catch blocks.

Each catch block must contain a different exception handler.

So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

```
public class MultipleCatchBlock1 {
  public static void main(String[] args) {
          try{
            int a[]=new int[5];
            a[5]=30/0;
           }
           catch(ArithmeticException e)
             {
              System.out.println("Arithmetic Exception occurs");
             }
           catch(ArrayIndexOutOfBoundsException e)
             {
              System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
           catch(Exception e)
             {
              System.out.println("Parent Exception occurs");
             }
           System.out.println("rest of the code");
    }
}
```

```
public class MyMain {
public static void main(String[] args) {
int a[]=new int[4];
try
{
a[0]=12/0;
System.out.println(a[6]);
}
catch(ArithmeticException e)
{
System.out.println("Aritmetic Exception"+e.getMessage());
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Index out of bound"+e.getMessage());
}
catch(Exception e)
{
System.out.println(e.getMessage());
}
finally
{
System.out.println("Finally block executed");
}
System.out.println("outside Finally block executed");
}
}
```

Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 2

Lecture 15

Lecture 15

- Throw and throws in Exception Handling

throw keyword

- The Java throw keyword is used to throw an exception explicitly.
- We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.
- We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

- We can also define our own set of conditions and throw an exception explicitly using throw keyword.
- For example, we can throw ArithmeticException if we divide a number by another number.
- Here, we just need to set the condition and throw exception using throw keyword.

syntax of the Java throw keyword

```
public class Main {
    public static void main(String[] args) {
        int dividend = 10;
        int divisor = 0;

        if (divisor == 0) {
            throw new ArithmeticException("Cannot divide by zero");
        }

        int result = dividend / divisor;
        System.out.println("Result: " + result);
    }
}
```

syntax of the Java throw keyword

throw new exception_class("error message");

Example of throw IOException.

throw new IOException("sorry device error");

Throwing Unchecked Exception

```
public class TestThrow1 {
   public static void validate(int age) {
      if(age<18) {
         //throw Arithmetic exception if not eligible to vote
    throw new ArithmeticException("Person is not eligible to vote");

      }
      else {
         System.out.println("Person is eligible to vote!!");
      }
   }
```

```
 public static void main(String args[]){
      //calling the function
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

Java throws keyword

- The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

- In a program, if there is a chance of raising an exception then the compiler always warns us about it and compulsorily we should handle that checked exception, Otherwise, we will get compile time error saying unreported exception xyz must be caught or declared to be thrown.
- To prevent this compile time error we can handle the exception in two ways:

➢ By using try catch

➢ By using the throws keyword

Syntax of java throws

return_type method_name() **throws** exception_class_name

{

//method code

}

Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

There are two cases:

- **Case1:**You caught the exception i.e. handle the exception using try/catch.
- **Case2:**You declare the exception i.e. specifying throws with the method.

```
// Java program to illustrate error in case
// of unhandled exception
class MyMain {
    public static void main(String[] args)
    {
        Thread.sleep(10000);
        System.out.println("Hello Geeks");
    }
}
```

Output

```
error: unreported exception InterruptedException; must be caught or declared to be
thrown
```

```
// Java program to illustrate throws
class  {
    public static void main(String[] args)
        throws InterruptedException
    {
        Thread.sleep(10000);
        System.out.println("Hello Geeks");
    }
}
```

Important Points to Remember about throws Keyword

- throws keyword is required only for checked exceptions and usage of the throws keyword for unchecked exceptions is meaningless.
- throws keyword is required only to convince the compiler and usage of the throws keyword does not prevent abnormal termination of the program.
- With the help of the throws keyword, we can provide information to the caller of the method about the exception.

Difference Between throw and throws

throw	throws
The throw keyword is used inside a function. It is used when it is required to throw an Exception logically.	The throws keyword is used in the function signature. It is used when the function has some statements that can lead to exceptions.
The throw keyword is used to throw an exception explicitly. It can throw only one exception at a time.	The throws keyword can be used to declare multiple exceptions, separated by a comma. Whichever exception occurs, if matched with the declared ones, is thrown automatically then.
throw keyword cannot propagate checked exceptions. It is only used to propagate the unchecked Exceptions that are not checked using the throws keyword.	throws keyword is used to propagate the checked Exceptions only.

Object Oriented Programming with Java (Subject Code: BCS-403)

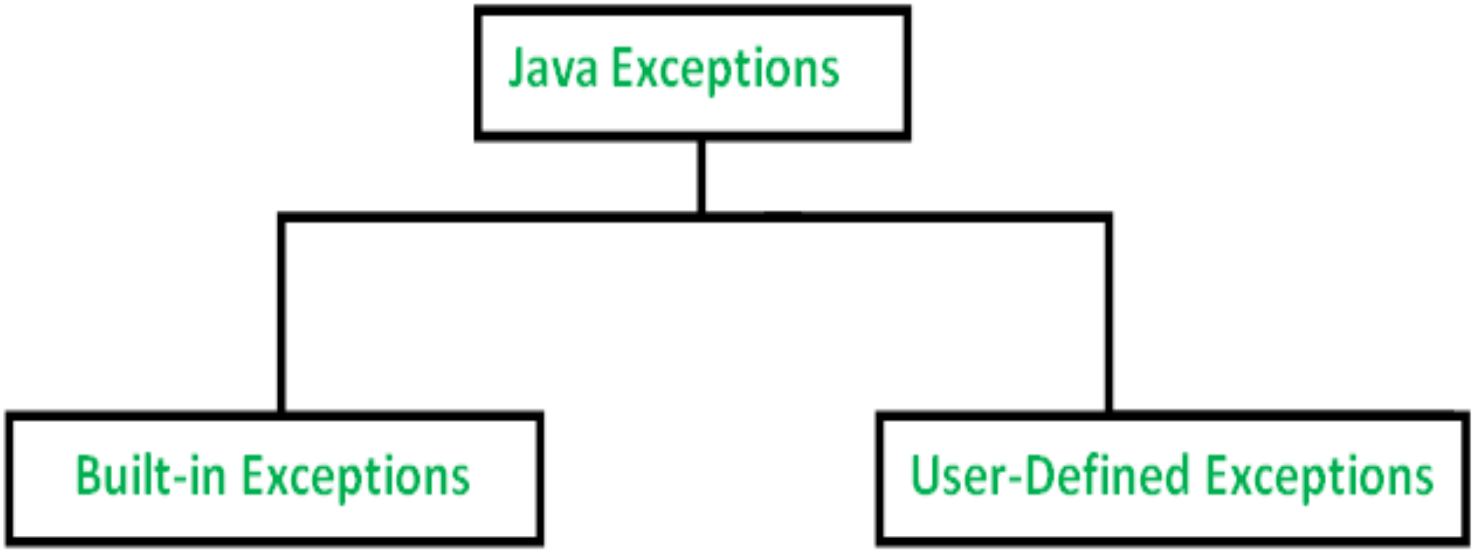
Unit 2

Lecture 16

Lecture 16

- In-built and User Defined Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **ArithmeticException:** It is thrown when an exceptional condition has occurred in an arithmetic operation.
- **ArrayIndexOutOfBoundsException:** It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
- **ClassNotFoundException:** This Exception is raised when we try to access a class whose definition is not found
- **FileNotFoundException:** This Exception is raised when a file is not accessible or does not open.

IOException: It is thrown when an input-output operation failed or interrupted

- **InterruptedException:** It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
- **NoSuchFieldException:** It is thrown when a class does not contain the field (or variable) specified
- **NoSuchMethodException:** It is thrown when accessing a method that is not found.
- **NullPointerException:** This exception is raised when referring to the members of a null object. Null represents nothing

NumberFormatException: This exception is raised when a method could not convert a string into a numeric format.

- **RuntimeException:** This represents an exception that occurs during runtime.
- **StringIndexOutOfBoundsException:** It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string
- **IllegalArgumentException :** This exception will throw the error or error statement when the method receives an argument which is not accurately fit to the given relation or condition. It comes under the unchecked exception.

```
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        try {
            int a = 30, b = 0;
            int c = a/b;  // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by 0");
        }
    }
}
```

//Java program to demonstrate NullPointerException

```
class NullPointer_Demo
{
    public static void main(String args[])
    {
        try {
            String a = null; //null value
            System.out.println(a.charAt(0));
        } catch(NullPointerException e) {
            System.out.println("NullPointerException..");
        }
    }
}
```

// Java program to demonstrate NumberFormatException

```
class  NumberFormat_Demo
{
    public static void main(String args[])
    {
        try {
            // "akki" is not a number
            int num = Integer.parseInt ("akki") ;

            System.out.println(num);
        } catch(NumberFormatException e) {
            System.out.println("Number format exception");
        }
    }
}
```

User-Defined Exceptions

- the built-in exceptions in Java are not able to describe a certain situation.
- In such cases, the user can also create exceptions which are called ‘user-defined Exceptions’.
- The user should create an exception class as a subclass of the Exception class.
- Since all the exceptions are subclasses of the Exception class, the user should also make his class a subclass of it.

Create user-defined Exception

The following steps are followed for the creation of a user-defined Exception.

class MyException extends Exception

We can write a default constructor in his own exception class.

MyException(){}

We can also create a parameterized constructor with a string as a parameter.

We can use this to store exception details. We can call the superclass(Exception) constructor from this and send the string there.

```
MyException(String str)
{
   super(str);
}
```

To raise an exception of a user-defined type, we need to create an object to his exception class and throw it using the throw clause, as:

```
MyException me = new MyException("Exception details");
throw me;
```

Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 2

Lecture 17

Lecture 17

- Byte Streams and Character Streams
- Reading and Writing File in Java

Byte Streams and Character Streams

In Java the streams are used for input and output operations by allowing data to be read from or written to a source or destination.

Java offers two types of streams:

- character streams
- byte streams.

Character Streams

- Character streams are designed to address character based records, which includes textual records inclusive of letters, digits, symbols, and other characters.
- These streams are represented by way of training that quit with the phrase "Reader" or "Writer" of their names, inclusive of FileReader, BufferedReader, FileWriter, and BufferedWriter.
- Character streams offer a convenient manner to read and write textual content-primarily based information due to the fact they mechanically manage character encoding and decoding.

Byte Streams

- Byte streams are designed to deal with raw binary data, which includes all kinds of data, including characters, pictues, audio, and video.
- These streams are represented through cclasses that cease with the word "InputStream" or "OutputStream" of their names,along with FileInputStream,BufferedInputStream, FileOutputStream and BufferedOutputStream.
- Byte streams offer a low-stage interface for studying and writing character bytes or blocks of bytes.

- They are normally used for coping with non-textual statistics, studying and writing files of their binary form, and running with network sockets.

Difference between Character Stream and Byte Stream in Java

Aspect	Character Streams	Byte Streams
Data Handling	Handle character-based data	Handle raw binary data
Representation	Classes end with "Reader" or "Writer"	Classes end with "InputStream" or "OutputStream"
Suitable for	Textual data, strings, human-readable info	Non-textual data, binary files, multimedia
Character Encoding	Automatic encoding and decoding	No encoding or decoding
Text vs non-Text data	Text-based data, strings	Binary data, images, audio, video

Performance	Additional conversion may impact performance	Efficient for handling large binary data
Handle Large Text Files	May impact performance due to encoding	Efficient, no encoding overhead
String Operations	Convenient methods for string operations	Not specifically designed for string operations
Convenience Methods	Higher-level abstractions for text data	Low-level interface for byte data
Reading Line by Line	Convenient methods for reading lines	Byte-oriented, no built-in line-reading methods
File Handling	Read/write text files	Read/write binary files

Java I/O

- **Java I/O** (Input and Output) is used to process the input and produce the output based on the input.
- Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

Stream

- A stream is a sequence of data. In Java a stream is composed of bytes. It's called a stream because it's like a stream of water that continues to flow.

In java, 3 streams are created for us automatically.

All these streams are attached with console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Code to print **output and error** message to the console.

```
System.out.println("simple message");
System.err.println("error message");
```

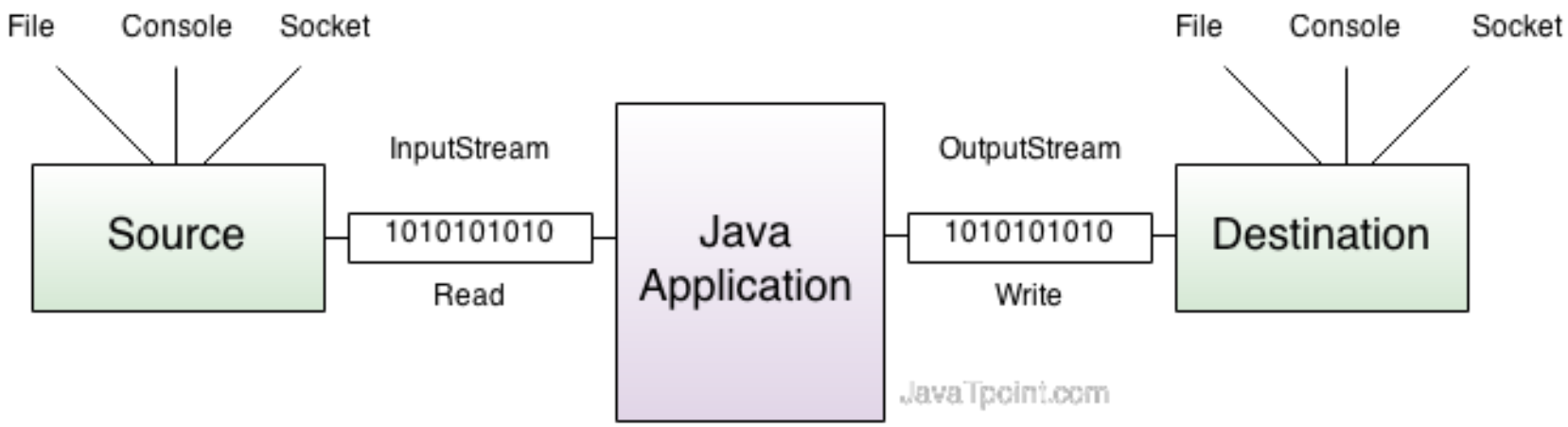
OutputStream

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

InputStream

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

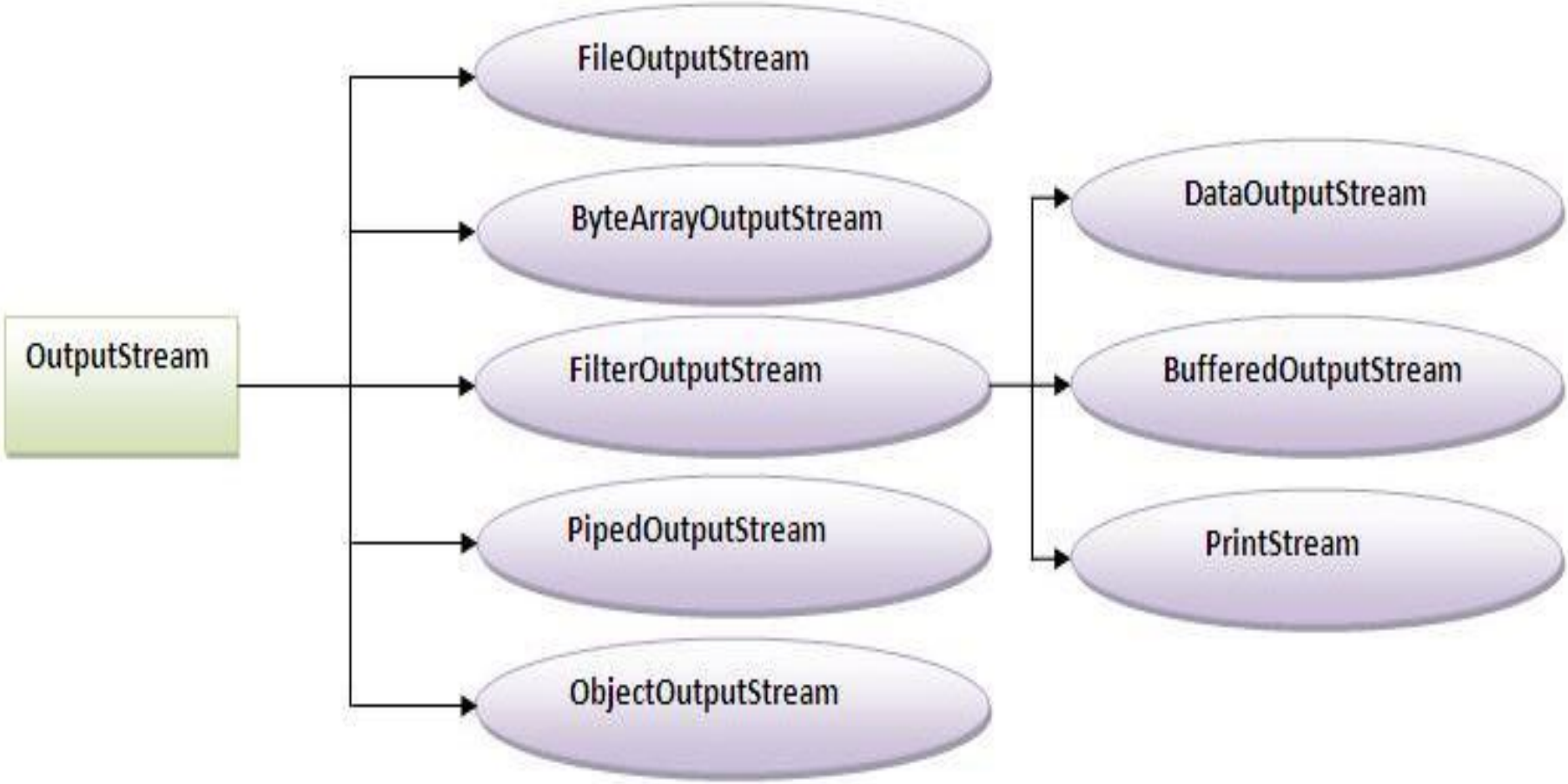
Working of Java OutputStream and InputStream



OutputStream class

- OutputStream class is an abstract class. It is the super class of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Method	Description
1) public void write(int)throws IOException:	Is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException:	Is used to write an array of byte to the current output stream.
3) public void flush()throws IOException:	Flushes the current output stream.
4) public void close()throws IOException:	Is used to close the current output stream.

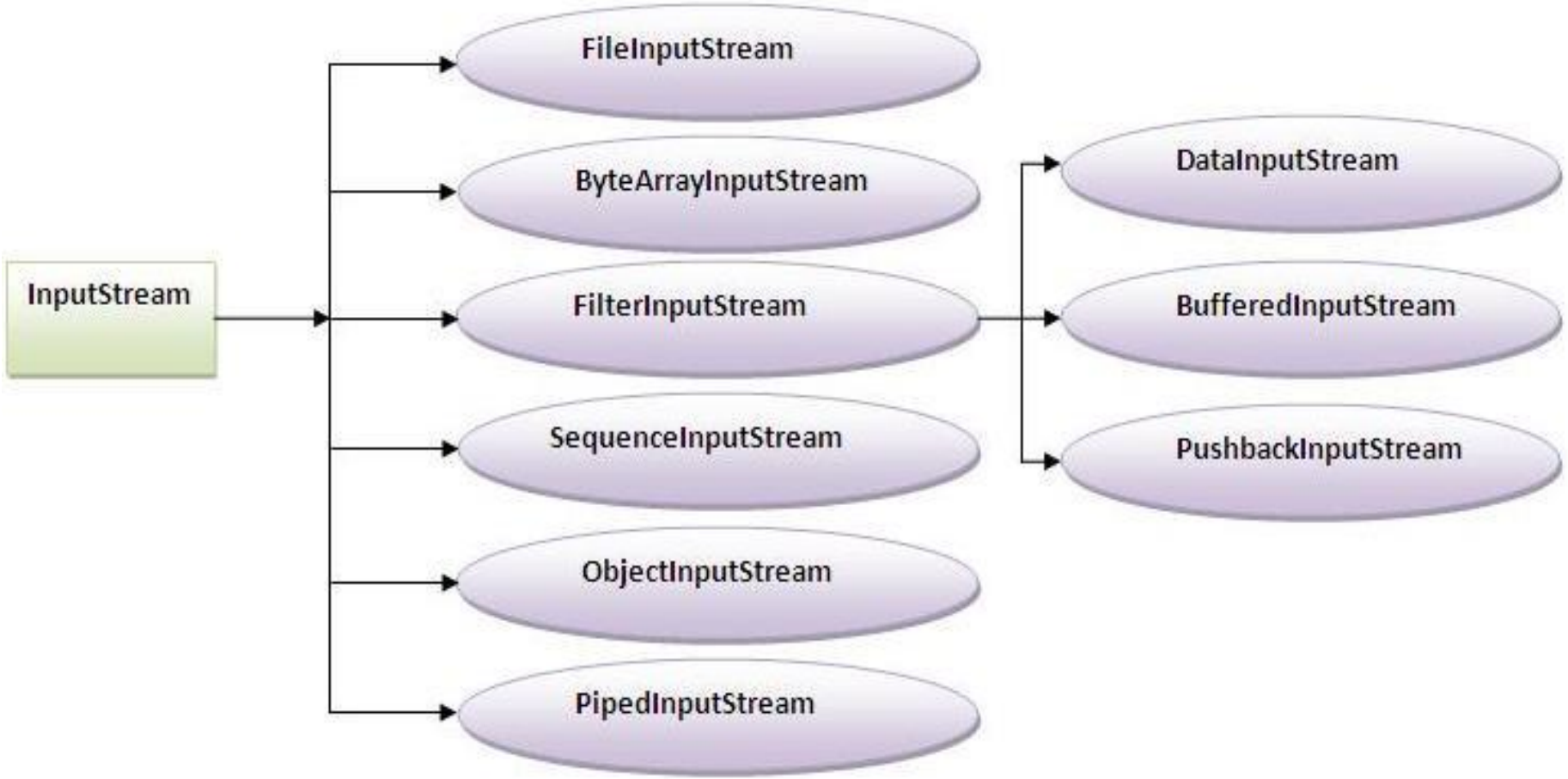

OutputStream
FileOutputStream
ByteArrayOutputStream
FilterOutputStream
PipedOutputStream
ObjectOutputStream
DataOutputStream
BufferedOutputStream
PrintStream

InputStream class

- InputStream class is an abstract class. It is the super class of all classes representing an input stream of bytes.

Commonly used methods of InputStream class

Method	Description
1) public abstract int read()throws IOException:	Reads the next byte of data from the input stream. It returns -1 at the end of file.
2) public int available()throws IOException:	Returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException:	Is used to close the current input stream.


InputStream
FileInputStream
ByteArrayInputStream
FilterInputStream
SequenceInputStream
ObjectInputStream
PipedInputStream
DataInputStream
BufferedInputStream
PushbackInputStream

FileInputStream and FileOutputStream (File Handling)

In Java, FileInputStream and FileOutputStream classes are used to read and write data in file. In another words, they are used for file handling in java.

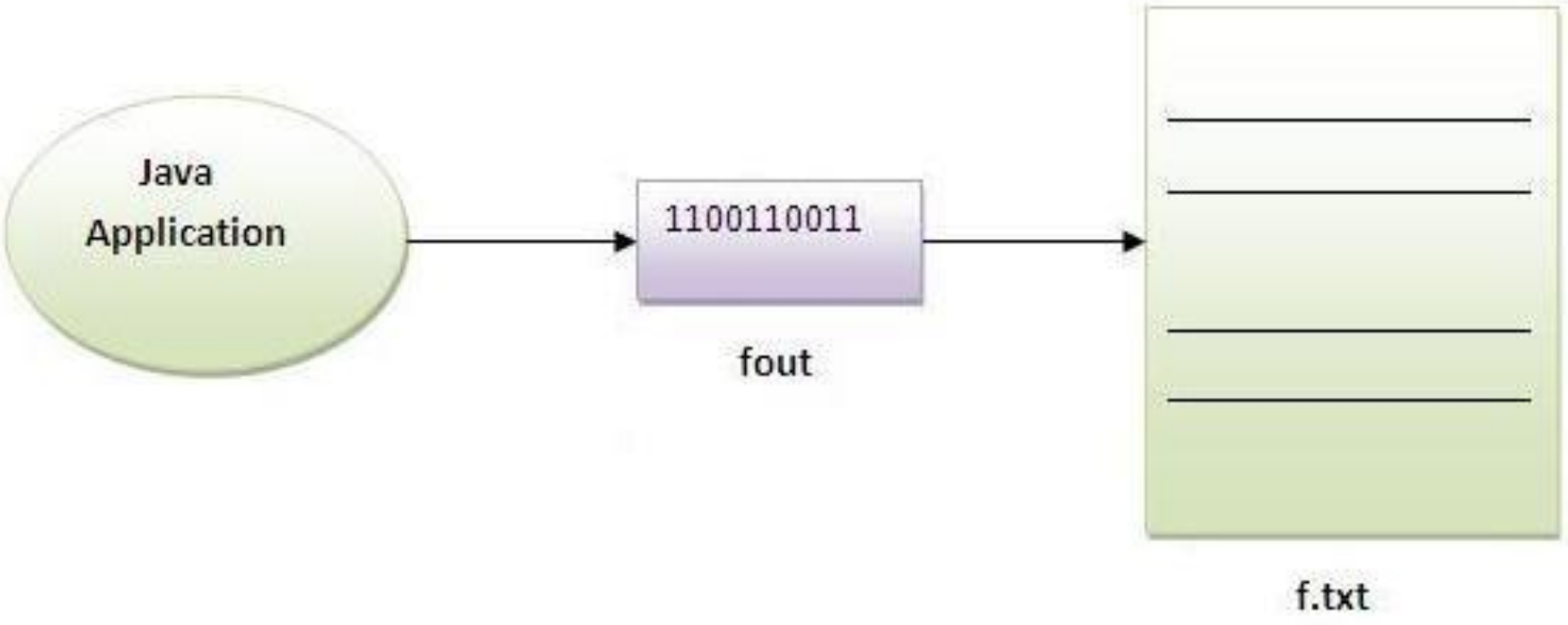
Java FileOutputStream class

Java FileOutputStream is an output stream for writing data to a file.

If you have to write primitive values then use FileOutputStream. Instead, for character-oriented data, prefer FileWriter.But you can write byte-oriented as well as character-oriented data.

```
import java.io.*;
class Test{
  public static void main(String args[]){
   try{
     FileOutputstream fout=new FileOutputStream("abc.txt");
     String s="Sachin Tendulkar is my favourite player";
     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);}
  }
}
```

Output:success...

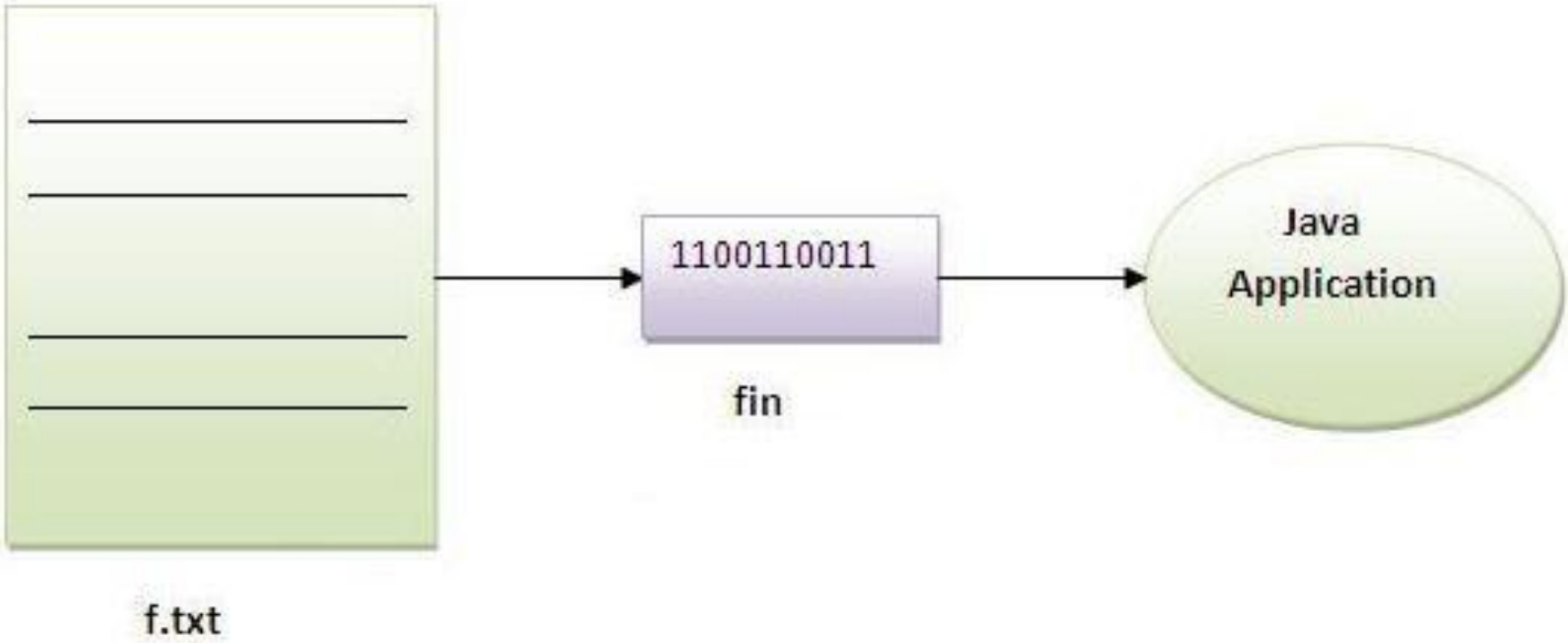

Java
Application
1100110011
fout
f.txt

Java FileInputStream class

Java FileInputStream class obtains input bytes from a file.It is used for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.It should be used to read byte-oriented data for example to read image, audio, video etc.

```
import java.io.*;
class SimpleRead{
 public static void main(String args[]){
  try{
    FileInputStream fin=new FileInputStream("abc.txt");
    int i=0;
    while((i=fin.read())!=-1){
     System.out.println((char)i);
    }
    fin.close();
  }catch(Exception e){system.out.println(e);}
 } }
```

Output:Sachin is my favourite player.


1100110011
Java
Application
fin
f.txt

Reading the data of current java file and writing it into another file

```
import java.io.*;
class C{
public static void main(String args[])throws Exception{
FileInputStream fin=new FileInputStream("C.java");
FileOutputStream fout=new FileOutputStream("M.java");
int i=0;
while((i=fin.read())!=-1){
fout.write((byte)i);
}
fin.close();
}
}
```

We can read the data of any file using the FileInputStream class whether it is java file, image file, video file etc

Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 2

Lecture 18

Lecture 18

- **Thread**
- **Thread Life Cycle**

Multithreading in Java

- **Multithreading in java** is a process of executing multiple threads simultaneously.
- Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
- Java Multithreading is mostly used in games, animation etc.

Advantages of Java Multithreading

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

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

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

What is Thread in java

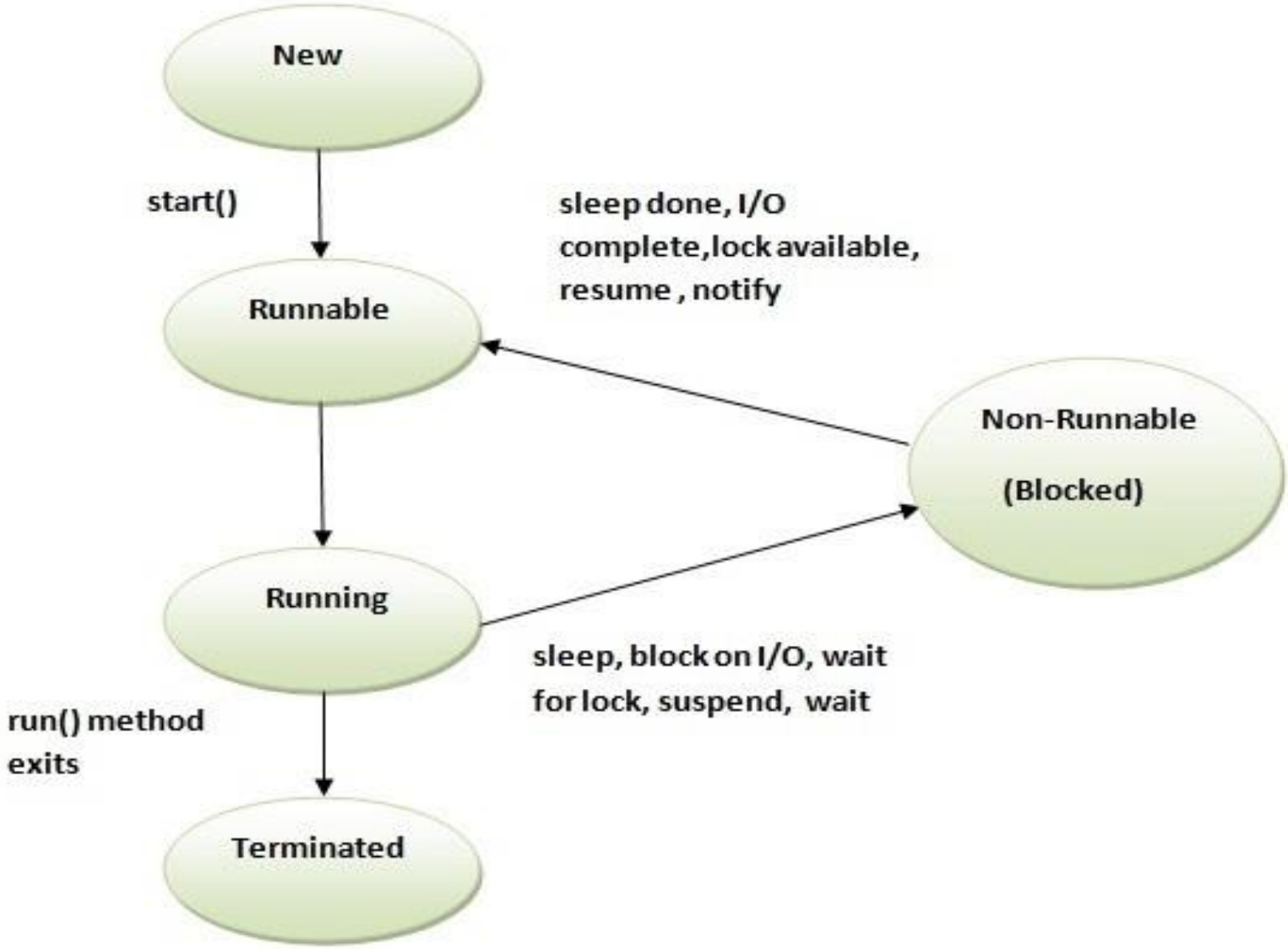
- A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.
- Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

Note: At a time one thread is executed only.

Life cycle of a Thread (Thread States)

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

- New
- Runnable
- Running
- Non-Runnable (Blocked)
- Terminated


New
start()
Runnable
sleep done, I/O
complete,lock available,
resume , notify
Non-Runnable
(Blocked)
Running
sleep, block on I/O, wait
for lock, suspend, wait
run() method
exits
Terminated

1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 2

Lecture 19

Lecture 19

- Creating Threads
- Thread Priorities

How to create thread

There are two ways to create a thread:

- By extending Thread class
- By implementing Runnable interface.

Thread class:

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

Commonly used 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.
- **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 void setName(String name):** changes the name of the thread.
- **public Thread currentThread():** returns the reference of currently executing 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.
- **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.

- **public void suspend():** is used to suspend the thread(depricated).
- **public void resume():** is used to resume the suspended thread(depricated).
- **public void stop():** is used to stop the thread(depricated).

Starting a thread

- **start() method** of Thread class is used to start a newly created thread. It performs following tasks:A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

Java Thread Example by extending Thread class

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

Java Thread Example by implementing Runnable interface

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

Sleep method in java

- The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep() method in java

```
public static void sleep(long miliseconds)throws InterruptedException
```

```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    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();
 }
}
```

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.

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.

```
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(“CSA Webtech");
  System.out.println("After changing name of t1:"+t1.getName());
 }
}
```

Priority of a Thread (Thread Priority)

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

Three constants defined in Thread class

- public static int MIN_PRIORITY
- public static int NORM_PRIORITY
- public static int MAX_PRIORITY

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

```
class TestMultiPriority1 extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentT
    hread().getName());
  System.out.println("running thread priority is:"+Thread.current
    Thread().getPriority());
    }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();
   }}
```

Object Oriented Programming with Java (Subject Code: BCS-403)

Unit 2

Lecture 20

Lecture 20

- Synchronizing Threads
- Inter-thread Communication

Synchronizing Threads

- Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.
- In order to overcome this problem, we have thread synchronization.

Synchronization means coordination between multiple processes/threads.

Why use Java Synchronization?

Java Synchronization is used to make sure by some synchronization method that only one thread can access the resource at a given point in time.

Java Synchronized Blocks

Java provides a way of creating threads and synchronizing their tasks using synchronized blocks.

A synchronized block in Java is synchronized on some object.

All synchronized blocks synchronize on the same object and can only have one thread executed inside them at a time.

All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

General Form of Synchronized Block

- Only one thread can execute at a time.
- sync_object is a reference to an object
- whose lock associates with the monitor.
- The code is said to be synchronized on
- the monitor object

```
synchronized(sync_object)
{
   // Access shared variables and other
   // shared resources
}
```

Types of synchronization

There are two types of synchronization that are as follows:

- Process synchronization
- Thread synchronization

1. Process Synchronization in Java

Process Synchronization is a technique used to coordinate the execution of multiple processes. It ensures that the shared resources are safe and in order.

2. Thread Synchronization in Java

Thread Synchronization is used to coordinate and ordering of the execution of the threads in a multi-threaded program.

There are two types of thread synchronization are mentioned below:

- Mutual Exclusive
- Cooperation (Inter-thread communication in Java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data.

Inter-thread Communication in Java

- Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.
- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

It is implemented by following methods of Object class:

- wait()
- notify()
- notifyAll()

1) wait() method

- The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
- The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

2) notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor.

If any threads are waiting on this object, one of them is chosen to be awakened.

The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

public final void notify()

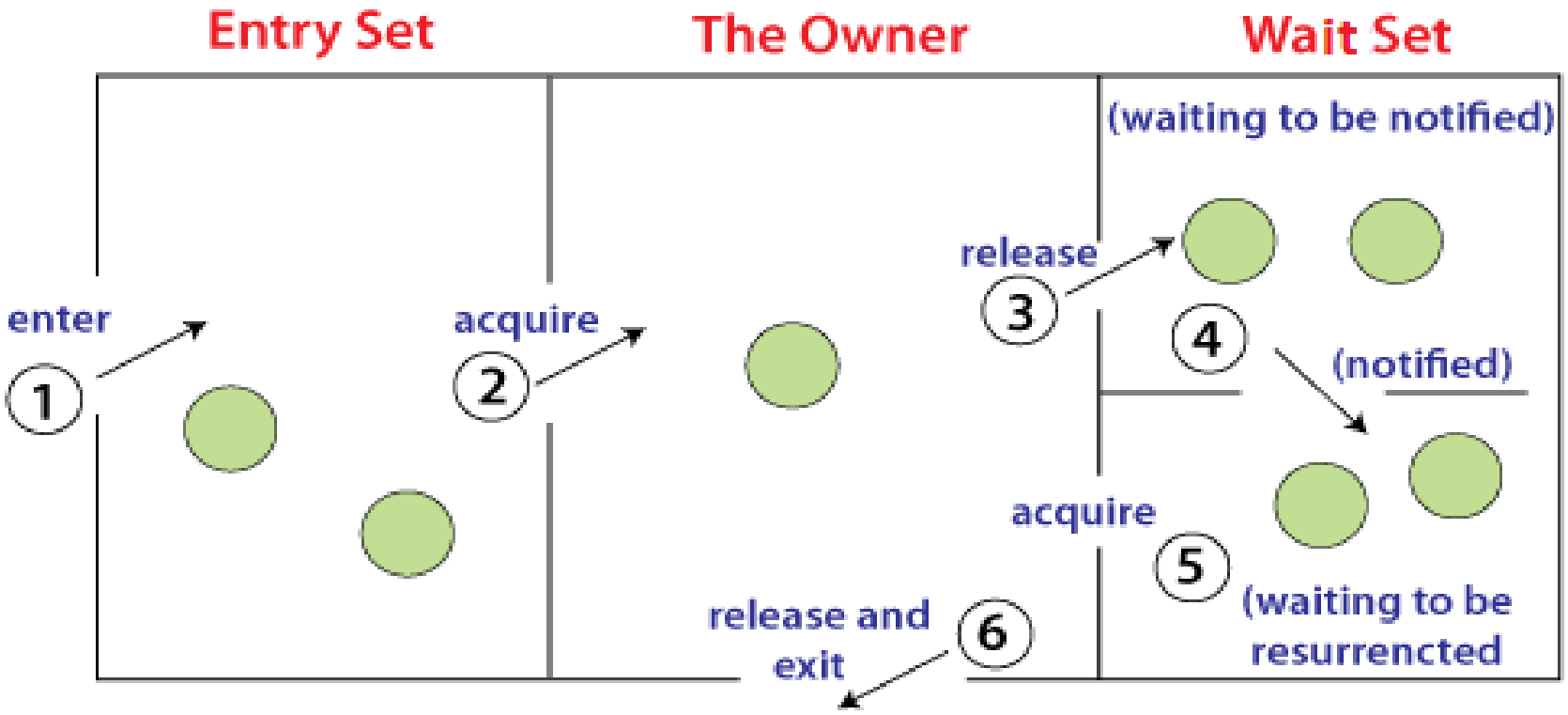
3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

Syntax:

public final void notifyAll()

Understanding the process of inter-thread communication



process of inter-thread communication

1. Threads enter to acquire lock.
2. Lock is acquired by one thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.