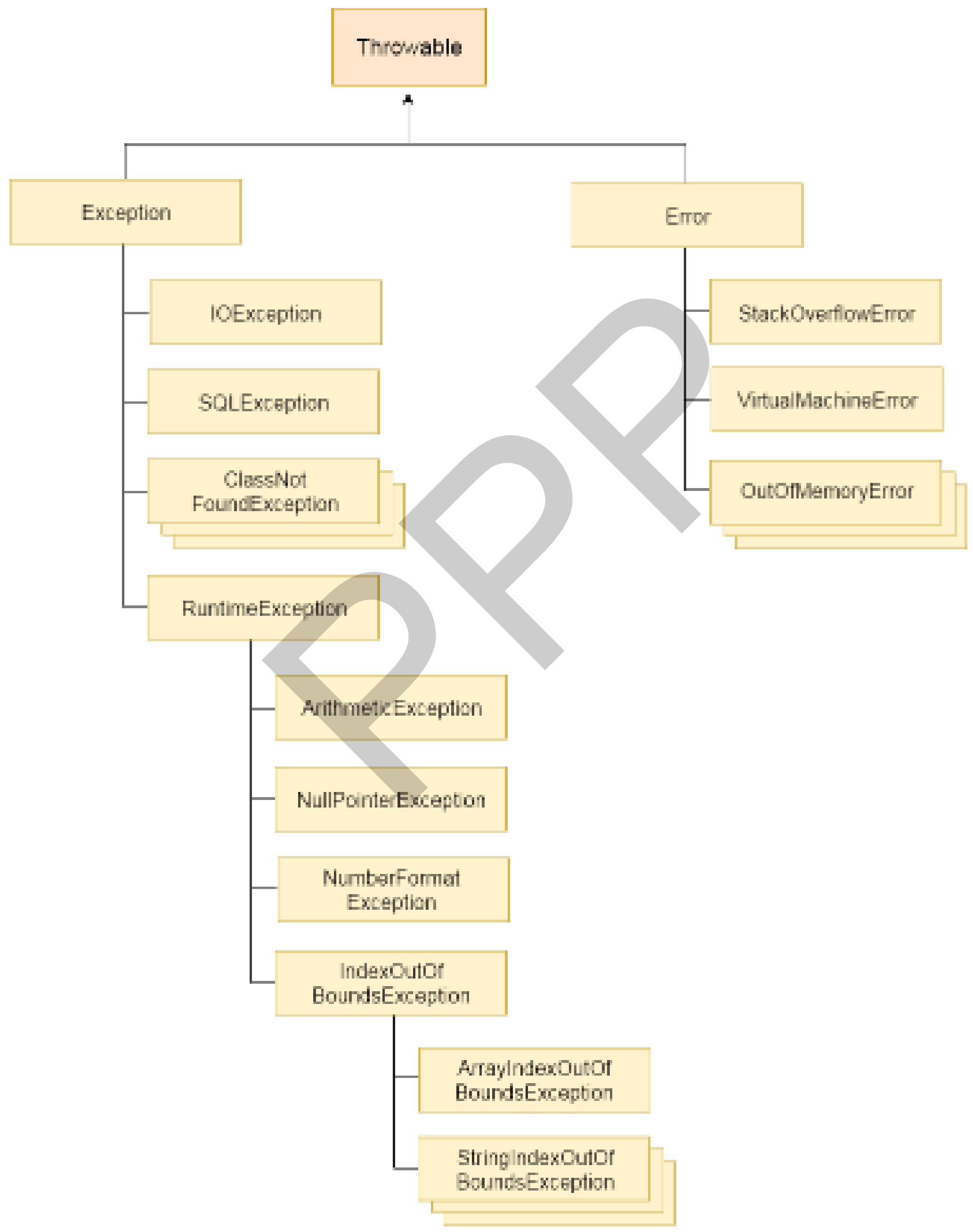
Unit – III More Features of the Java Platform

- Exception handling
- Input-output and file handling
- The collections framework and handling classes in it
- Introduction to the java.util package
- Multithreading
- Introduction to network programming
- Introduction to lambda expressions and serialization

Exception

- Exception handling is a mechanism to handle the runtime errors so that normal flow of the application can be maintained.
- The core advantage of exception handling is **to maintain the normal flow of the application**.
- In other words exception is a **run-time error.**
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.
- That method may choose to handle the exception itself, or pass it on.
- Either way, at some point, the exception is caught and processed.
- Exception can be generated by
  - Java run-time system (**In built exception**) or
  - Manually generated by code.(**User Defined Exception**)

Hierarchy of Exception Classes



Hierarchy of Exception Classes

- The **java.lang.Throwable** class is the **root class** of Java Exception hierarchy

- **It is inherited by two subclasses:**
  Exception and
  Error.

Hierarchy of Exception Classes

Types of Java Exceptions

Two types of exceptions: Checked & Unchecked Exception

Difference between Checked and Unchecked Exceptions

1) Checked Exception

- Checked exceptions are **checked at compile-time.**
- The classes which inherit Throwable class except RuntimeException and Error are known as checked exceptions
- **e.g.** IOException
  SQLException
  ClassNotFoundException **etc.**

2) Unchecked Exception

- Unchecked exceptions are **checked at runtime.**
- The classes which inherit RuntimeException or inherit Error are known as unchecked exceptions
- **e.g.** ArithmeticException
  NullPointerException
  ArrayIndexOutOfBoundsException
  StackoverflowError
  VirtualMachineError **etc**.

Common Scenarios of Java Exceptions

1. If we divide any number by zero, there occurs an ArithmeticException.

```
int a = 50 / 0; // ArithmeticException
```

2. If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s = null;
System.out.println( s.length() );  // NullPointerException
```

Common Scenarios of Java Exceptions

3.The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

```
String s = "abc“ ;
int I = Integer.parseInt(s); // NumberFormatException
```

4. If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

```
int a[ ] = new int [ 5 ];
a[ 10 ] = 50;   // ArrayIndexOutOfBoundsException
```

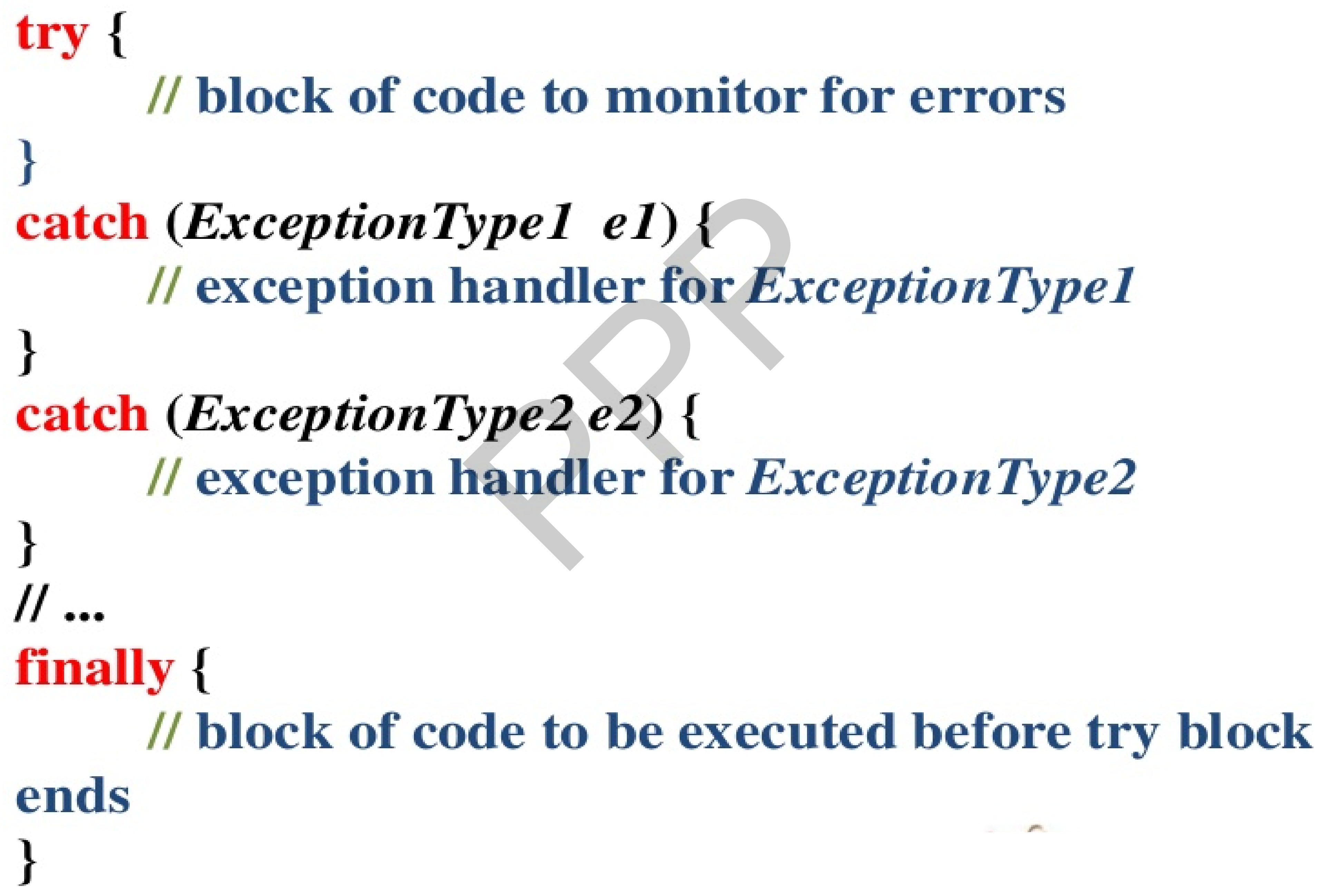
Exception

- **Some causes for an exception are**
  - Division-by-zero
  - Array index negative or out-of-bounds
  - Unexpected end-of-file condition
  - Incorrect number format
  - Malformed URL
- **Classes for Exception Handling**
  - ❖ EOFException
  - ❖ FileNotFoundException
  - ❖ ArithmeticException
  - ❖ NumberFormatException
  - ❖ ArrayIndexOutOfBoundsException
  - ❖ NegativeArraySizeException
  - ❖ NoSuchMethodException
  - ❖ NoSuchFieldException
  - ❖ ClassNotFoundException **etc.**

Exception Handling

- **Try Block**
  - It contains a block of statements that you want to monitor for exception.
- **Catch Block**
  - Immediately following the try block is a sequence of catch blocks.
  - An argument is passed to each catch block to catch the exception and handle it.
  - Each catch block argument is type of exception.
- **Finally Block**
  - Contains a code that must be executed before a method returns.
  - It must be executed either exception is generated or not.
- **Throw Statement**
  - Manually throws an exception.
- **Throws Clause**
  - To specify any exception that is thrown out of a method.

Syntax of Exception Handling



```
try {
        // block of code to monitor for errors
}
catch (ExceptionType1  e1) {
        // exception handler for ExceptionType1
}
catch (ExceptionType2 e2) {
        // exception handler for ExceptionType2
}
// ...
finally {
        // block of code to be executed before try block
ends
}
```

Exception Handling

- If a problem occurs during execution of the try block, the JVM immediately stops executing the **try** block and looks for a **catch** block that can be process that type of exception. Any remaining statements in the **try** block are not executed.

- The search beings at the **first catch block**. It the type of the exception object matches the type of the catch block parameter, those statements are executed. Otherwise, the remaining catch clauses are examined in sequence for a type match.

- When a catch block execution complete, control passes to the statements in the **finally** block.
- **Finally block** must be execute either try block generate error or not.

- **Finally block is optional. It can be used for relinquish resources. E.g.** you may wish to close files or databases at this point.

Example of Exception

```
public class Test {
  public static void main(String args[]) {
    try{

      System.out.println("Before Division");

      int i = Integer.parseInt(args[0]);
      int j = Integer.parseInt(args[1]);
      System.out.println(i/j);
      System.out.println("After Division");
    }
   catch(Exception e) {
      System.out.println("main: " + e);
    }
  }
}
```

Example of Exception

```
public class Test {
  public static void main(String args[]) {
    try{

      System.out.println("Before Division");

      int i = Integer.parseInt(args[0]);
      int j = Integer.parseInt(args[1]);
      System.out.println(i/j);
      System.out.println("After Division");
    }
    catch(ArithmeticException e) {
     System.out.println("Arithmetic Exception");    }

}
}
```

Example of Exception

```
public class Test{
 public static void main(String args[]) {
  try {
     int a = 50;
     a = a / 0;                              // ArithmeticException

     String s=null;
     System.out.println( s.length() );  // NullPointerException

     String str;
     str="Hello";
     int i = Integer.parseInt(str);    // NumberFormatException

     int array1[];
     array1=new int[5];
     array1 [10]=50;          // ArrayIndexOutOfBoundsException
  }
```

Example of Exception

```
    catch (ArithmeticException e) {
        System.out.println("No. is Divide by Zero ");
    }
    catch (NullPointerException e) {
        System.out.println("Null value in a variable");
    }
    catch (NumberFormatException e) {
        System.out.println("Type coversion is illegal");
    }
    catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array Index Out Of Bound Exception");
    }
    catch(Exception e) {   // Other exceptions handling
           System.out.println("Inbuilt exception is generated: " + e);
    }
}
}
```

Catch block searches

- Exception handler begins at the **first catch** block immediately following the try block in which the exception occurred.
- The search continues until the type of the exception object matches the type of the catch block parameter.
- If none of these catch blocks are match than **default exception handler** is invoked to display the exception and terminate the program.

Exception Handling : Throw Statement

- Explicitly exception is generated with the help of **throw statement.**
- Inside a catch block, you may throw the same exception object that was provided as an argument.

- **Syntax**

```
catch (Exception Type param) {
….
throw param;
…..
}
```

- **Alternative Syntax**

```
throw new ExceptionType (args);
```

Exception Handling : Throw Statement

```
class ThrowDemo {

 public static void main(String args[]) {
  try {
   System.out.println("Before a");
   a();  //method defined in next slide.
   System.out.println("After a");
  }
  catch(ArithmeticException e) {
   System.out.println("main: " + e);
  }
  finally {
   System.out.println("main: finally");
  }
 }
```

Exception Handling : Throw Statement

```
public static void a() {
   try {
     int i = 1;
     int j = 0;
     System.out.println("Before division");
     System.out.println(i/j);
     System.out.println("After division");
   }
   catch(ArithmeticException e) {
     System.out.println("d: " + e);
// search for the exception handler for object e in calling environment.
     throw e;
}
   finally {
     System.out.println("a: finally");
   } }  }
```

Exception Handling: Throw Clause

E.g. Created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the Arithmetic Exception otherwise print a message welcome to vote.

```
public class Test {
static void validate(int age){
    if(age <18)
    throw new ArithmeticException("not a valid age for voting ");
    else
    System.out.println("welcome to vote");
}
public static void main(String args[]){
    validate(19);
    System.out.println("rest of the code...");
}}
```

Exception Handling: Throws Clause

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You can do this by including a **throws clause** in the **method's declaration**.

- A **throws clause** lists the types of exceptions that a method might throw.

- The java compiler checks each constructor and method to determine the types of exceptions it can generate.

- **We can use throws keyword to delegate the responsibility of exception handling to the caller** (It may be a method or JVM) then caller method is responsible to handle that exception.

Exception Handling: Throws Clause

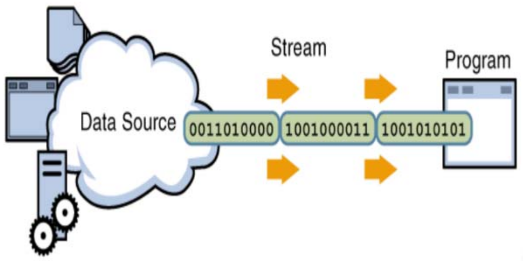
```
class Test {
public static void main(String args[]) {
          a();    }
public static void a() {
   try {       b();    }
   catch(ArithmeticException e) {
                    System.out.println(" Exception handler in Method a ::");
          }
}
public static void b()    throws ArithmeticException {
          c();  }
public static void c() throws ArithmeticException {
           int no=10;
          no=no/0;   }    }
```

Difference between throw and throws

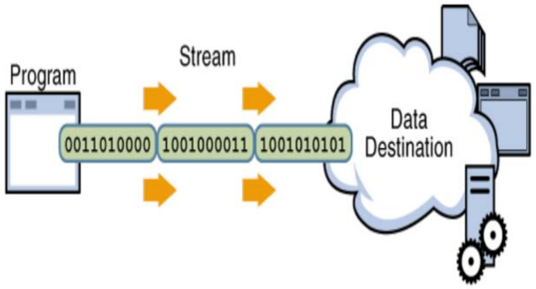
No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Streams

- The java.io package contains all the classes required for input and output operations.
- ***Stream***: an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)
  - it acts as a buffer between the data source and destination
- "Standard" I/O streams:
  - System.in – defaults to the keyboard
  - System.out – defaults to the monitor
  - System.err – defaults to the monitor.



- A stream connects a program to an I/O object
  - System.out connects a program to the screen
  - System.in connects a program to the keyboard



- ***Input stream***: a stream that provides input to a program. System.in is an input stream
- ***Output stream***: a stream that accepts output from a program. System.out is an output stream

Streams

Let's see the code to print **output and an error** message **to the console.**

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

Let's see the code to get **input from console.**

```
int i=System.in.read(); //returns ASCII code of 1st character
System.out.println((char)i); //will print the character
```

java.io Classes Hierarchy

✓The Streams can be divided into

- ✓ **Character Streams** and
- ✓ **Byte Streams**

Character Streams:

These streams are typically used to read and write text data.

Reader and **Writer** are classes for character streams in **java.io.**

Reader that read characters & **Writer** that write characters.

Byte Streams:

These streams are typically used to read and write binary data such as images and sounds.

InputStream that read bytes and

OutputStream that write bytes.

How to do I/O

```
import java.io.*;
```

- *Open* the stream
- *Use* the stream (read, write, or both)
- *Close* the stream

FileWriter

- **Java FileWriter class is used to write character-oriented data to a file.**
- **Constructor**
  - FileWriter(String filepath)
  - FileWriter(String filepath , boolean append)
  - FileWriter(File fileobj)
- **Methods**
  - **It is used to write the string into FileWriter.**
    void write (String text)
  - **It is used to write some portion of string into FileWriter.**
    void write (String s , int index, int size)
  - **It is used to write the char into FileWriter.**
    void write(char c)
  - **It is used to write char array into FileWriter.**
    void write(char[] c)
  - void flush ( ) : It is used to flushes the data of FileWriter.
  - void close ( ) : It is used to close the FileWriter.

FileReader

- **Java FileReader class is used to read data from the file.**
- **Constructor**
  - FileReader(String filepath)
  - FileReader(File fileobj)

- **Reader Class Methods**
  - **It is used to return a character in ASCII form. It returns -1 at the end of file.**
    int read( )
  - void close ( ) **: It is used to close the FileReader class.**

FileReader & FileWriter

```
import java.io.*;
public class Test{
    public static void main(String args[]){
try {
        FileWriter fw=new FileWriter("D:\\testfile.txt");
        fw.write(" Hello ");   fw.write(" How ");  fw.write(" Are You ? ");
        fw.close();

        FileReader fr=new FileReader("D:\\testfile.txt");
          int i;
          while( (i = fr.read() ) != -1)
                          System.out.print( (char) i );
          fr.close();
  }
  catch(Exception e)
   {           System.out.println("Exception Message: "+e);          }   } }
```

BufferedReader & BufferedWriter

- It is used to reduce the number of reads and writes to the physical device.
- It improves the performance

BufferedWriter Class

- **Constructor**
  - BufferedWriter(Writer w) :Default bufsize is 8192 chars.
  - BufferedWriter(Writer w , int bufsize)
- **Methods**
  - void close() **Closes the stream, flushing it first.**
  - void flush() **Flushes the stream.**
  - void newLine() **Writes a line separator.**
  - void write(int c) **Writes a single character.**

  Writes a portion of an array of characters.
  - void write(char[] cbuf, int off, int len)

  Writes a portion of a String.
  - void write(String s, int off, int len)

BufferedReader Class

- **Constructor**
  - BufferedReader(Reader r)
  - BufferedReader(Reader r , int bufsize)

- **Methods**
  - int read() **: Reads a single character.**
  - String readLine() **: Reads a line of text.**

  Reads characters into a portion of an array.
  - int read(char[] cbuf, int off, int len) :

  Closes the stream and resources associated with it.
  - void close() :

BufferedReader Class

```
import java.io.*;
class Test {
  public static void main(String args[]) {
    try {
     // Create a file writer
      //FileWriter fw = new FileWriter(args[0]);
      FileWriter fw = new FileWriter("D:\\a.txt");
      // Create a buffered writer
      BufferedWriter bw = new BufferedWriter(fw,10);
      // Write strings to the file
      for(int i = 1; i <= 12; i++) {
        bw.write("Line " + i + "\n");      }
      // Close buffered writer
      bw.close();
```

BufferedReader Class

```
 // Create a file reader
 //FileReader fr = new FileReader(args[0]);
 FileReader fr = new FileReader("D:\\a.txt");

 // Create a buffered reader
 BufferedReader br = new BufferedReader(fr);

 // Read and display lines from file
 String s;
 while((s = br.readLine()) != null)
   System.out.println(s);
 // Close file reader
 fr.close();    }
catch(Exception e) {
 System.out.println("Exception: " + e);    }   }  }
```

<u>InputStreamReader Class</u>

- It converts stream of bytes into stream of characters.
- **<u>Constructor</u>**
  - InputStreamReader (InputStream is)
  - InputStreamReader (InputStream is ,
    String encoding)
- **<u>Method</u>**
  - String getEncoding( )
    - Returns the name of the character encoding .
- To read from a keyboard and convert into character stream.
  - InputStreamReader isr = new InputStreamReader (System.in)

BufferedReader & BufferedWriter

```
import java.io.*;      // Program - 1
class Test {
        public static void main ( String args[] )  {
        try {
        //FileWriter fw = new FileWriter(args[0]);
        //FileReader fr = new FileReader(args[0]);
        FileWriter fw = new FileWriter("D:\\a.txt");
        FileReader fr = new FileReader("D:\\a.txt");
        int i;
InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        String sno, sname ;    // store user input
        String record = new String();  // store record data
```

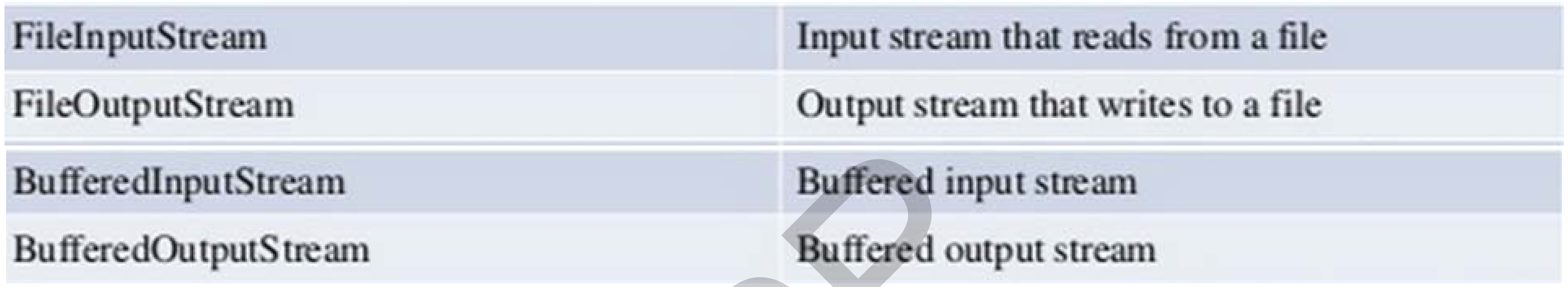
BufferedReader & BufferedWriter

```
for ( i = 0 ; i < 2 ; i++)   {
        System.out.print("Enter the no::");
        sno = br.readLine();

        record = record.concat(sno);

        System.out.print("Enter the name::");
        sname = br.readLine();

        record = record.concat(" ").concat(sname+"\n");

        fw.write(record);
        record = "";     }
```

BufferedReader & BufferedWriter

```
     fw.close();
     while ( ( i = fr.read() ) != -1 )        {
        System.out.print( (char) i );
        }
        fr.close();
        }
        catch(Exception e)  {
                System.out.println("e"+e);
        }
    }
}
```

Byte stream classes



FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream

BufferedInputStream & BufferedOutputStream

```
import java.io.*;
class Test {
public static void main(String args[]) {
try {
    FileOutputStream fos = new FileOutputStream(args[0]);
    //FileOutputStream fos = new FileOutputStream("D:\\a");
    BufferedOutputStream bos =   new BufferedOutputStream(fos);

     for(int i = 1; i <= 12; i++) {
       bos.write(i);
     }
     bos.close();
```

BufferedInputStream & BufferedOutputStream

```
// Create a file input stream
    FileInputStream fis = new FileInputStream(args[0]);
    //FileInputStream fis = new FileInputStream("D:\\a);
    // Create a buffered input stream
    BufferedInputStream bis = new BufferedInputStream(fis);
    // Read and display data
    int i;
    while((i = bis.read()) != -1) {     System.out.println(i);      }
    // Close file input stream
    bis.close();
}
catch(Exception e) {
    System.out.println("Exception: " + e);    }  }  }
```

Collections Framework

What is a Framework?

A framework is a set of classes and interfaces which provide a ready-made architecture.

- Any group objects represented as a single unit is known as the collection of the objects.

- In Java, a separate framework named the “Collection Framework” has been defined in JDK 1.2 which holds all the collection classes and interface in it.

- **Two main “root” interfaces of Java collection classes are**
  - java.util.Collection and
  - java.util.Map

Collections Framework

ArrayList:

- ArrayList provides us with **dynamic arrays** in Java.

- The size of an ArrayList is increased automatically if the collection grows or shrinks if the objects are removed from the collection.

- Java ArrayList allows us to **randomly access the list**.

- ArrayList can not be used for primitive types like int, char, etc. We will **need a wrapper class for such cases.**

Collections Framework

```
// Java program to demonstrate the working of ArrayList in Java
import java.io.*;   import java.util.*;
class Test {
    public static void main(String[] args)
    {    ArrayList<Integer> al = new ArrayList<Integer>();

        // Appending new elements at the end of the list
        for (int i = 1; i <= 5; i++)    al.add(i);

        // Printing elements
        System.out.println(al);

        // Remove element at index 3
        al.remove(3);

        // Displaying the ArrayList after deletion
        System.out.println(al);

        // Printing elements one by one
        for (int i = 0; i < al.size(); i++)  System.out.print(al.get(i) + " ");
    }    }
```

Collections Framework

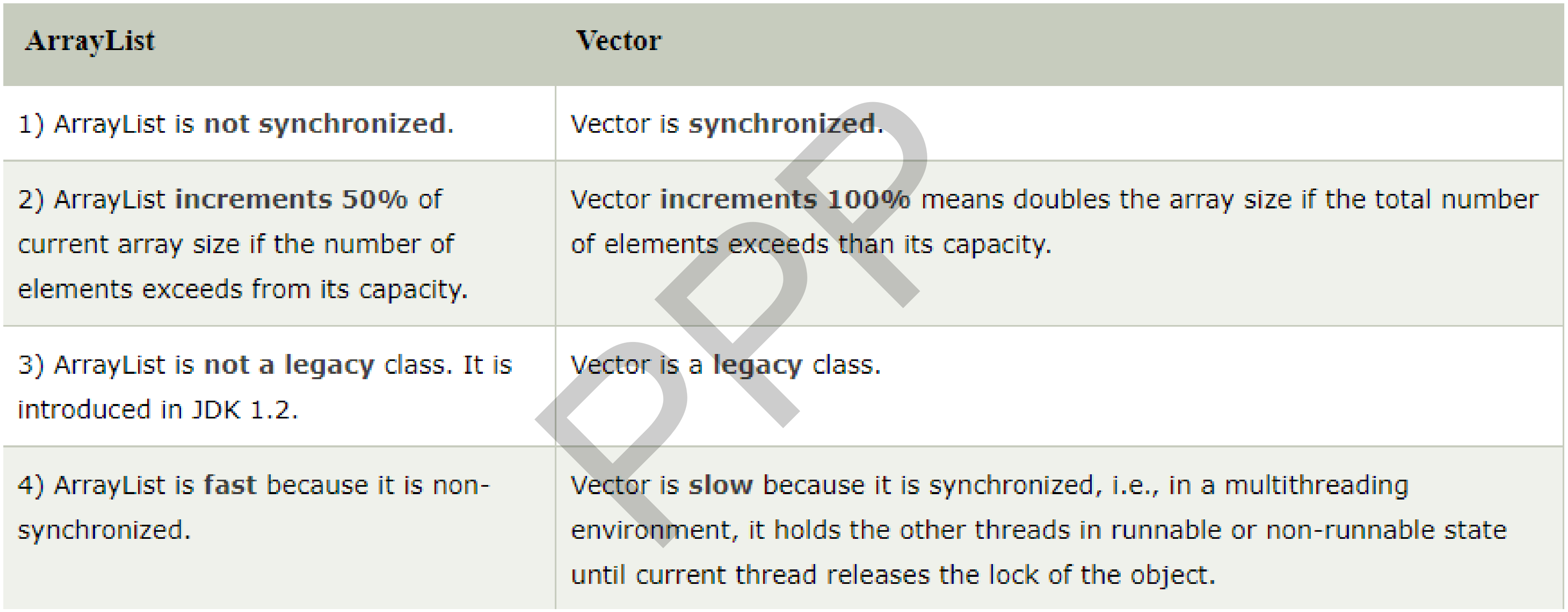
Vector:

- A vector provides us with dynamic arrays in Java.
- This is identical to ArrayList in terms of implementation.
- The primary difference between a Vector and an ArrayList is that a Vector is synchronized and an ArrayList is non-synchronized.
- **Synchronized** means in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object.

Collections Framework

```
// Java program to demonstrate the working of Vector in Java
import java.io.*;  import java.util.*;
class Test {
   public static void main(String[] args)
   {     Vector<Integer> v = new Vector<Integer>();

        // Appending new elements at the end of the list
        for (int i = 1; i <= 5; i++)  v.add(i);

        // Printing elements
        System.out.println(v);

        // Remove element at index 3
        v.remove(3);

        // Displaying the Vector after deletion
        System.out.println(v);

        // Printing elements one by one
        for (int i = 0; i < v.size(); i++)    System.out.print(v.get(i) + " ");
   }    }
```

Differences between ArrayList & Vector



ArrayList	Vector
1) ArrayList is **not synchronized**.	Vector is **synchronized**.
2) ArrayList **increments 50%** of current array size if the number of elements exceeds from its capacity.	Vector **increments 100%** means doubles the array size if the total number of elements exceeds than its capacity.
3) ArrayList is **not a legacy** class. It is introduced in JDK 1.2.	Vector is a **legacy** class.
4) ArrayList is **fast** because it is non-synchronized.	Vector is **slow** because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object.

Introduction to the java.util package

- It Contains the collections framework, legacy collection classes, date and time facilities, and miscellaneous **utility** classes (a string tokenizer, a random-number generator).
- **Random class** allows to generate random double , float, int , or long numbers.
- **Constructor**

  Random() :: It uses the current time as seed

  Random(long seed)

- **Instance Methods**

  nextInt()

  nextFloat()

  nextLong()

  nextDouble()

Introduction to the java.util package

```
import java.util.*;
class RandomInts
{
  public static void main(String args[])   {

    // Create random number generator
    Random generator = new Random();
    System.out.println(generator.nextInt());
    System.out.println(generator.nextFloat());
    System.out.println(generator.nextDouble());
    System.out.println(generator.nextLong());

    // Generate and display 4 random integers

    for(int i = 0; i < 4; i++)
      System.out.println(generator.nextInt());
  }    }
```

Multithreading

- Thread is the smallest executable unit of a process.
- Threads allows a program to operate more efficiently by doing multiple things at the same time.
- Threads can be used to perform complicated tasks in the background without interrupting the main program.
- A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources.
- **Multitasking** is when multiple processes share common processing resources such as a CPU.
- **Multi-threading** extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads.
- Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multithreading

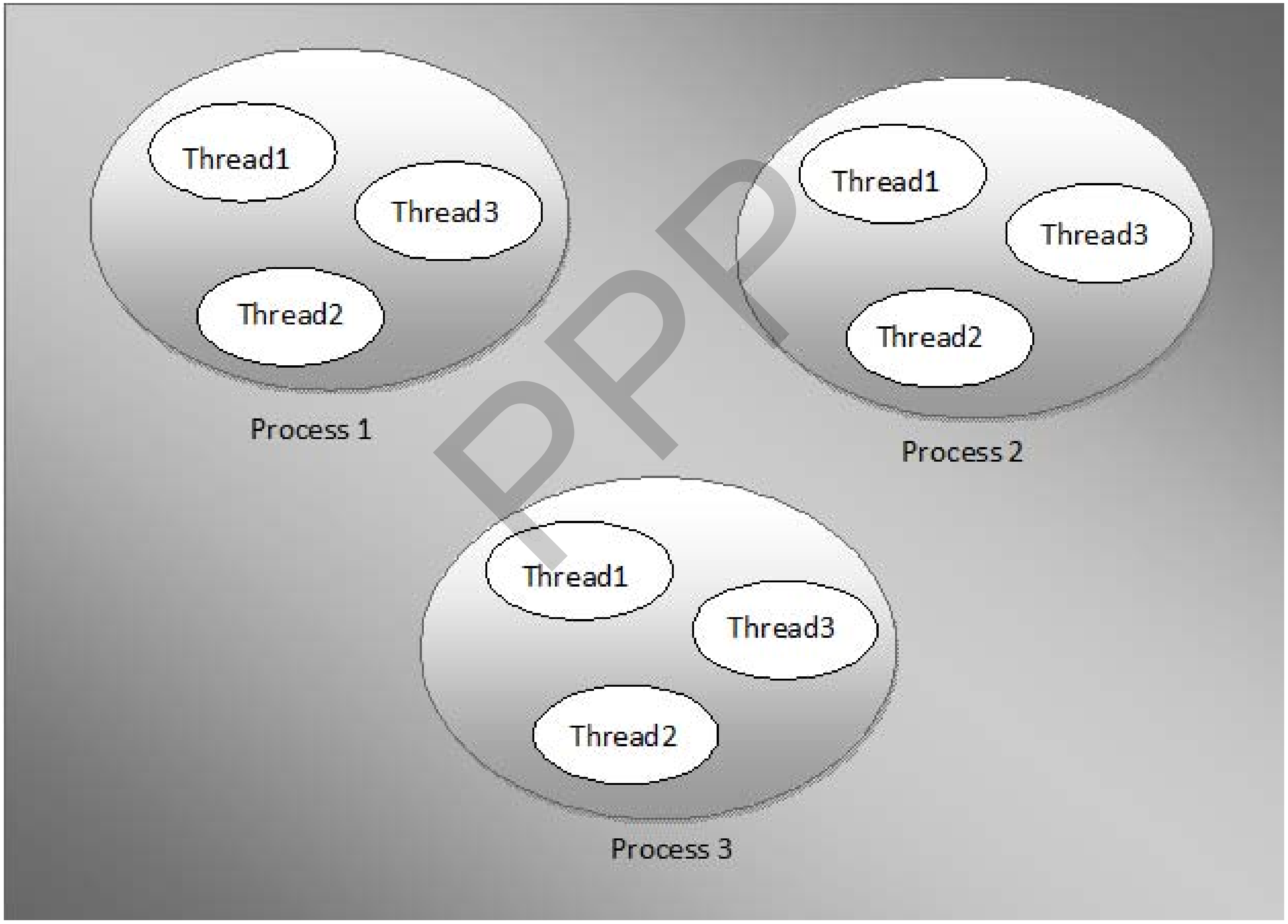
- Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.
- Each thread will have their own task and own path of execution in a process.
- All threads of the same process share memory of that process, so communication between the threads is fast.
- **Multithreaded Applications**
- **Text Editor :** Microsoft Word allows the user to hit print, save, and copy some portion, all at the same time. In this case three threads are running: one in charge of printing, another saving, and a third copying some portion.
- **Web Browsers** - A web browser can download any number of files and web pages (multiple tabs) at the same time and still lets you continue browsing. If a particular web page cannot be downloaded, that is not going to stop the web browser from downloading other web pages.

Multithreading

- **Web Servers -** A threaded web server handles each request with a new thread. There is a thread pool and every time a new request comes in, it is assigned to a thread from the thread pool.

- **Computer Games -** You have various objects like cars, humans, birds which are implemented as separate threads. Also playing the background music at the same time as playing the game is an example of multithreading.

Process & Thread



Multithreading

- **Process**
  - **It is an instance of a program.**
  - It contains program code and its current activity.
- **Thread**
  - A sequence of execution within a process.
  - **It is 'lightweight' process.**
    - Less time for Context switch.
    - Less space in memory and less resources.
  - **JVM manages threads and schedules them for execution.**
  - A process having multiple threads that can share resources.
- Processes have their own memory space, threads share memory

Multithreading

Thread class: It is in java.lang package

Constructors

Thread()

Thread(String name)

Thread(Runnable r)

Thread(Runnable r,String name)

Methods

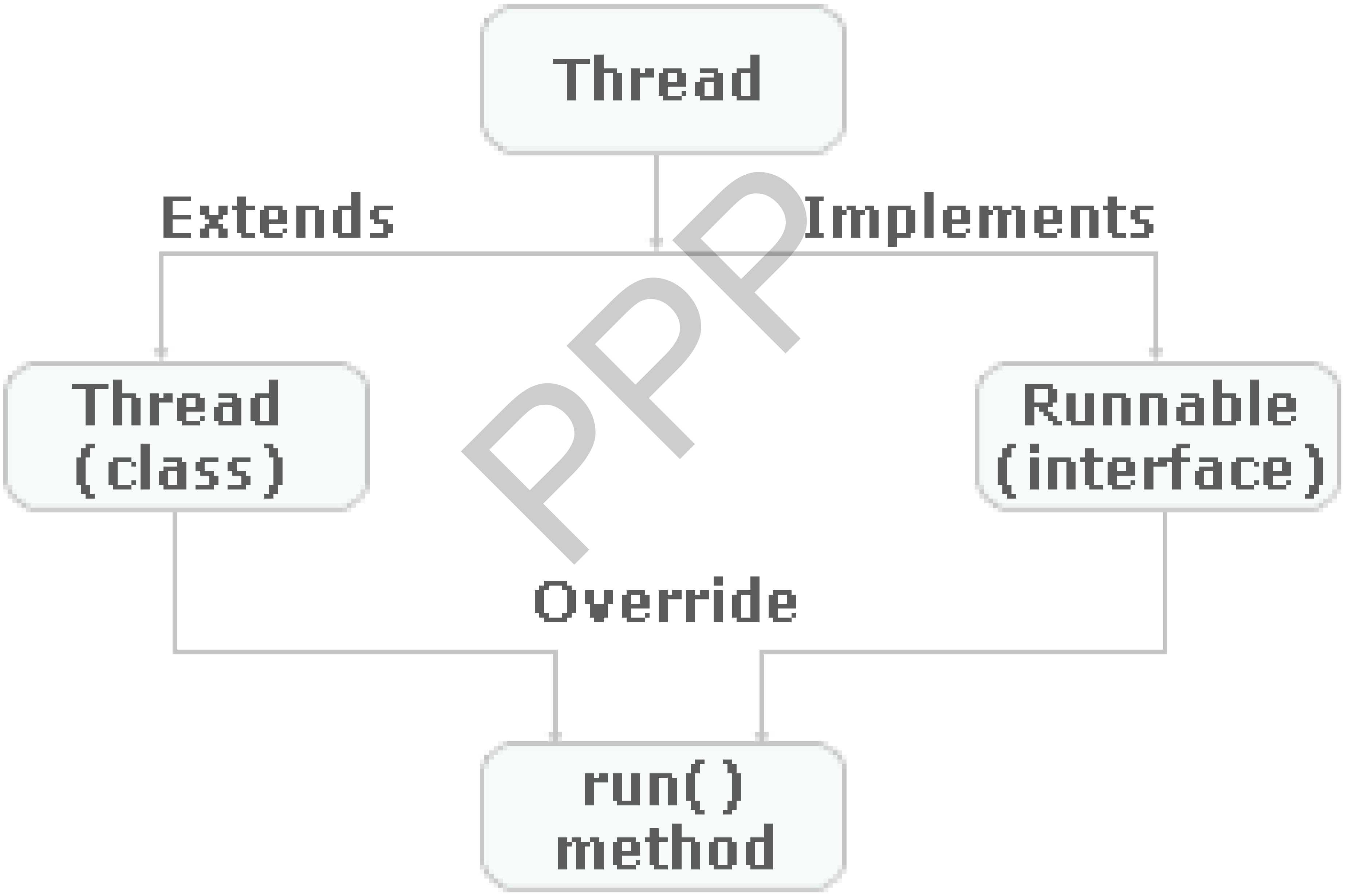
public void **start():** starts the execution of the thread.

JVM calls the run() method on the thread.

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

public void **sleep(long miliseconds):** Currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

public boolean **isAlive():** tests if the thread is alive.


Thread
Extends
Implements
Thread (class)
Runnable (interface)
Override
run() method

Multithreading : Program 1

[1] Extending the Thread class:

Running Threads:

If the class extends the Thread class, the thread can be run by **creating an instance of the class and call its start() method:**

```
public class ThreadDemo extends Thread {
    public void run() {
    System.out.println("This code is running in a thread");
          }
  public static void main(String[] args) {
    ThreadDemo thread = new ThreadDemo();
    thread.start();
    System.out.println("This code is outside of the thread");
  }
}
```

Multithreading : Program 2

```
class ThreadX extends Thread {
  public void run() {
    try {
      while(true) {
        Thread.sleep(2000);
        System.out.println("Hello");
      }
    }
    catch(InterruptedException ex) {
      ex.printStackTrace();    }  }    }
class ThreadDemo1 {
  public static void main(String args[]) {
          ThreadX tx = new ThreadX();
          tx.start();
  }   }
```

Multithreading : Program 3

[2] To implement the Runnable interface:

Running Threads:

If the class implements the Runnable interface, the thread can be run by **passing an instance of the class to a Thread object's constructor and then calling the thread's start() method:**

```
public class ThreadDemo implements Runnable {
   public void run() {
    System.out.println("This code is running in a thread");
  }
  public static void main(String[] args) {
    ThreadDemo obj = new ThreadDemo();
    Thread thread = new Thread(obj);
    thread.start();
    System.out.println("This code is outside of the thread");
  }   }
```

Multithreading

Main Differences between "extending" and "implementing" Threads

- When a class extends the Thread class, you cannot extend any other class.
- Implementing the Runnable interface, it is possible to extend from another class as well

Networking

- TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream-based connections between hosts on the Internet.
- A socket can be used to connect Java's I/O system to other programs that may reside either on the local machine or on any other machine on the Internet.
- There are two kinds of TCP sockets in Java.
- One is for servers, and the other is for clients.
- The ServerSocket class is designed to be a "listener," which waits for clients to connect before doing anything.
- The Socket class is designed to connect to server sockets and initiate protocol exchanges.

Networking

- The creation of a Socket object implicitly establishes a connection between the client and server.

Two constructors used to create client sockets

- **Socket(String hostName, int port)**

  Creates a socket connecting the local host to the named host and port

- **Socket(InetAddress ipAddress, int port)**

  Creates a socket using a preexisting InetAddress object and a port

Constructor for ServerSocket

- **ServerSocket(int port)**

  Creates a server socket, bound to the specified port.

Networking

- Once the Socket object has been created, it can also be examined to gain access to the input and output streams associated with it.

- **InputStream getInputStream():**

Returns the InputStream associated with the invoking socket.

- **OutputStream getOutputStream():**

Returns the OutputStream associated with the invoking socket

- The **DataInputStream** class read primitive **Java** data types from an underlying input stream in a machine-independent way.

- While the **DataOutputStream** class write primitive **Java** data types to an output stream in a portable way.

Networking :: ServerSocketDemo.java

```
//Server Socket listen on port number args[0]
import java.io.*;
import java.net.*;
import java.util.*;
class ServerSocketDemo {
  public static void main(String args[]) {
    try {
          // Get port
      int port = Integer.parseInt(args[0]);
      // Create server socket
      ServerSocket ss = new ServerSocket(port);
      // Create random number generator
      Random random = new Random();
```

Networking :: ServerSocketDemo.java

```
 // Create infinite loop
while(true) {
  // Accept incoming requests
  Socket s = ss.accept();

  // Write result to client
  OutputStream os = s.getOutputStream();
  DataOutputStream dos = new DataOutputStream(os);

  dos.writeUTF("Random Number is: ");
  dos.writeInt(random.nextInt());
  // Close socket
  s.close();     }    }
catch(Exception e) {
    System.out.println("Exception: " + e);    }  }    }
```

Networking : SocketDemo.java

```
import java.io.*;
import java.net.*;
class SocketDemo {
  public static void main(String args[]) {
   try {

      // Get server and port
      String server = args[0];
      int port = Integer.parseInt(args[1]);

      // Create socket
      Socket s = new Socket(server, port);

      // Read random number from server
      InputStream is = s.getInputStream();
      DataInputStream dis = new DataInputStream(is);
```

Networking : SocketDemo.java

```
      // Display result
      System.out.println(dis.readUTF());
      System.out.println(dis.readInt());

      // Close socket
      s.close();
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

Networking

- Store both the file ServerSocketDemo.java and SocketDemo.java file in the same directory.

- Open **Two Command Line Windows** using **cmd command** and open the directory.

- **To compile program type as below.**
  - `> javac ServerSocketDemo.java`
  - `> javac SocketDemo.java`

- **In One Command Line Window run server program as below.**
  - `> java ServerSocketDemo 5000`
- **In Second Command Line Window run client program as below.**
  - `> java SocketDemo 127.0.0.1  5000`

Lambda expressions

- Lambda Expressions were added in Java 8.
- A lambda expression is a short block of code which takes in parameters and returns a value.
- Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

Lambda expressions

Syntax

- The simplest lambda expression contains a single parameter and an expression:

  parameter -> expression

- To use more than one parameter, wrap them in parentheses:

  (parameter1, parameter2) -> expression

- Expressions are limited. They have to immediately return a value.
- Expressions can not contain variables, assignments or statements such as if or for.

- If the lambda expression needs to return a value, then the code block should have a return statement.

  (parameter1, parameter2) -> { code block }

Lambda expressions

Example

Use a lambda expression to print ArrayList:

```
import java.util.ArrayList;
public class Test {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);    numbers.add(9);
    numbers.add(8);    numbers.add(1);
System.out.println("Print Arraylist using for loop");
for(int i=0; i < numbers.size(); i++) {
System.out.println( numbers.get(i) );
}
System.out.println("Print Arraylist using Lambda
expression:");
    numbers.forEach( (n) -> { System.out.println(n); } );
  }   }
```

Lambda expressions

```
interface Sayable{
public String say(String name);
}
public class Test {
   public static void main(String[] args){
// Lambda expression with single parameter.
Sayable s1=(name)->{ return "Hello, "+name; };
System.out.println(s1.say("Shyam"));

// You can omit function parentheses
Sayable s2= name ->{          return "Hi, "+name;  };
System.out.println(s2.say("Kalpna"));
   }   }
```

lambda expressions : Multiple Parameters

```
interface Addable{
        int add(int a,int b);
}

public class Test{

public static void main(String[] args) {

// Multiple parameters in lambda expression
  Addable ad1=(a,b)->(a+b);
  System.out.println(ad1.add(10,20));

// Multiple parameters with data type in lambda expression
  Addable ad2=(int a,int b)->(a+b);
  System.out.println(ad2.add(100,200));
}
}
```

lambda expressions : Multiple Parameters

```
interface Arithop  {
        int operation(int a,int b);  }

public class Test{
public static void main(String[] args) {
// Multiple parameters in lambda expression
   Arithop ad1=(a,b)->(a+b);
   System.out.println(ad1.operation(10,20));
// Multiple parameters with data type in lambda expression
   Arithop ad2=(int a,int b)->(a-b);
   System.out.println(ad2.operation(20,10));

   Arithop ad3=(int a,int b)->(a/b);
   System.out.println(ad3.operation(20,10));

   Arithop ad4=(int a,int b)->(a*b);
   System.out.println(ad4.operation(20,10));  }   }
```

Serialization

- Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

- After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

- Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

- **Classes ObjectInputStream and ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

Serialization

- The **ObjectOutputStream class** contains many write methods for writing various data types, but one method in particular stands out −

public final void writeObject(Object x) throws IOException

- Method serializes an Object and sends it to the output stream.

- Similarly, the **ObjectInputStream class** contains the following method for deserializing an object −

public final Object readObject() throws IOException, ClassNotFoundException

- Method retrieves the next Object out of the stream and deserializes it.
- The return value is Object, so you will need to cast it to its appropriate data type.
- E.g. Employee class, which implements the Serializable interface −

Serialization

- **For a class to be serialized successfully, two conditions must be met –**
  - The class must **implement the java.io.Serializable interface.**
  - All of the fields in the class must be serializable. If a field is not serializable, it must be marked transient.
- **Example program** instantiates an Employee object and serializes it to a file.When the program is done executing, a file named **employee.ser is created**. The we deserialize the employee.ser file in employee object and display object data.
- The try/catch block tries to catch a **ClassNotFoundException**, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.
- **Return value of readObject() is cast to an Employee reference.**
- The value of the **id field** was 1001 when the object was serialized, but **because the field is transient**, this value was not sent to the output stream. The **id field of the deserialized Employee object is 0**.

Serialization

```
import java.io.*;

class Employee implements java.io.Serializable {
   //public transient int id;
   public int id;
   public String name;
   public String address;
}

public class Test {
   public static void main(String [] args) {
      Employee e = new Employee();
      e.id = 1001;
      e.name = "Aakash";
      e.address = "Ahmedabad, Gujarat, India";
```

Serialization

```
try {

FileOutputStream fileOut = new FileOutputStream("d://employee.ser");
ObjectOutputStream out = new ObjectOutputStream(fileOut);

out.writeObject(e);

out.close();
fileOut.close();

System.out.printf("Serialized data is saved in d:/employee.ser");

FileInputStream fileIn = new FileInputStream("d://employee.ser");
ObjectInputStream in = new ObjectInputStream(fileIn);
e = (Employee) in.readObject();
    System.out.println("Deserialized Employee...");
    System.out.println("id: " + e.id);
    System.out.println("Name: " + e.name);
    System.out.println("Address: " + e.address);
      in.close();
      fileIn.close();     }
```

Serialization

```
catch (IOException i) {
        i.printStackTrace();
        return;
     }

catch (ClassNotFoundException c) {
        System.out.println("Employee class not found");
        c.printStackTrace();
        return;
     }
```