M.S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of Computer Science and Engineering

Course Name: Programming in Java

Course Code: CSOE07-06

Credits: 3:0:0

UNIT 4

Term: February 2021-May 2021

Faculty:
Hanumatharaju R
Assistant Professor
Dept of CSE, MSRIT

UNIT 4

Exception Handling: Exception-Handling Fundamentals, Exception Classes, Exception Types, Uncaught Exceptions, Using try and catch, Multiple catch clauses, Nested try Statements, throw, throws, finally.

Multithreaded Programming: Java Thread Classes, The Java Thread Model, The Main Thread, Creating a Thread, Creating Multiple Threads, Using isAlive() and 22 join(), Thread Priorities, Synchronization, Suspending, Resuming and Stopping Threads.

RAMAIAH
Institute of Technology

Exception Handling

- Exceptions are abnormal conditions which change the normal flow of execution of a program.
- Exceptions are used for signaling erroneous (exceptional) conditions which occur during the run time processing.
- In Java, **Exception is an object that describes an exceptional condition that has occurred in a piece of code.**

Exception Handling

- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.

- Exceptions can be generated by the Java run- time system, or they can be manually generated by your code.

Advantage of Using Exception Handling

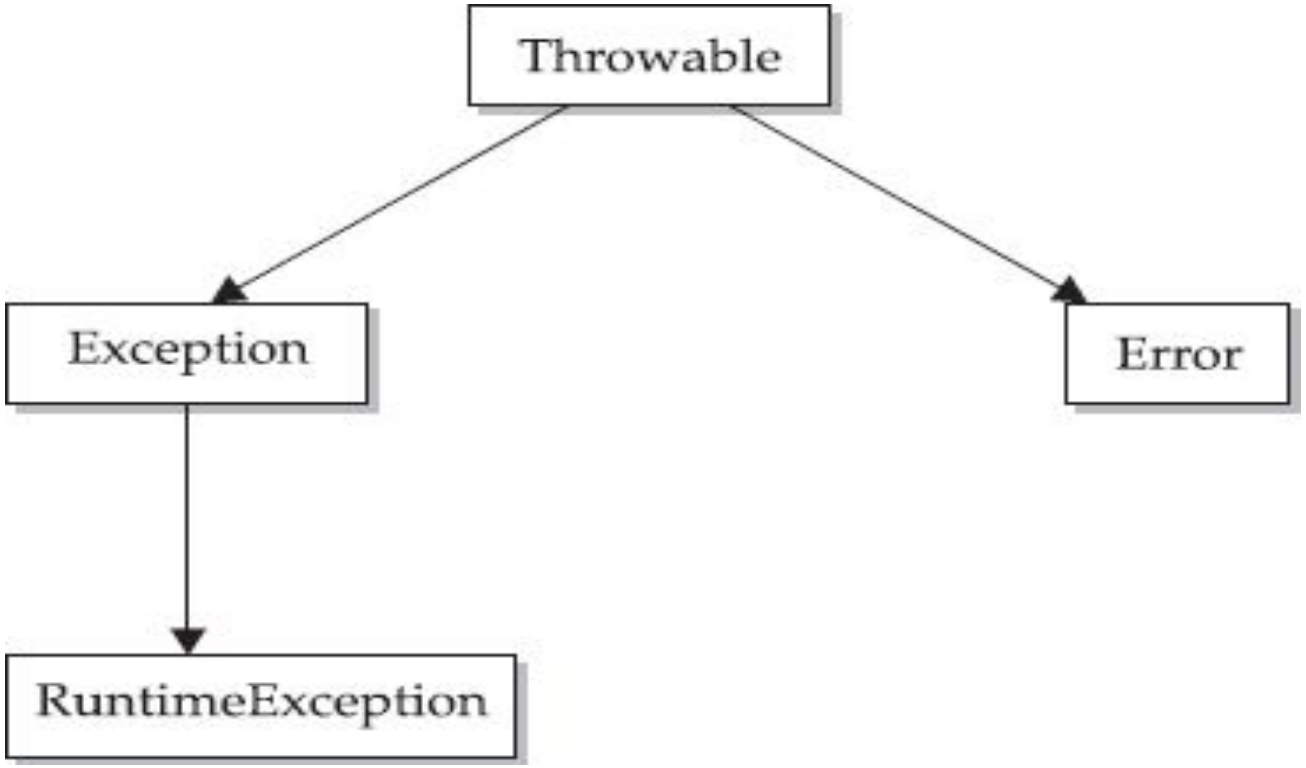
Advantage of Exception Handling

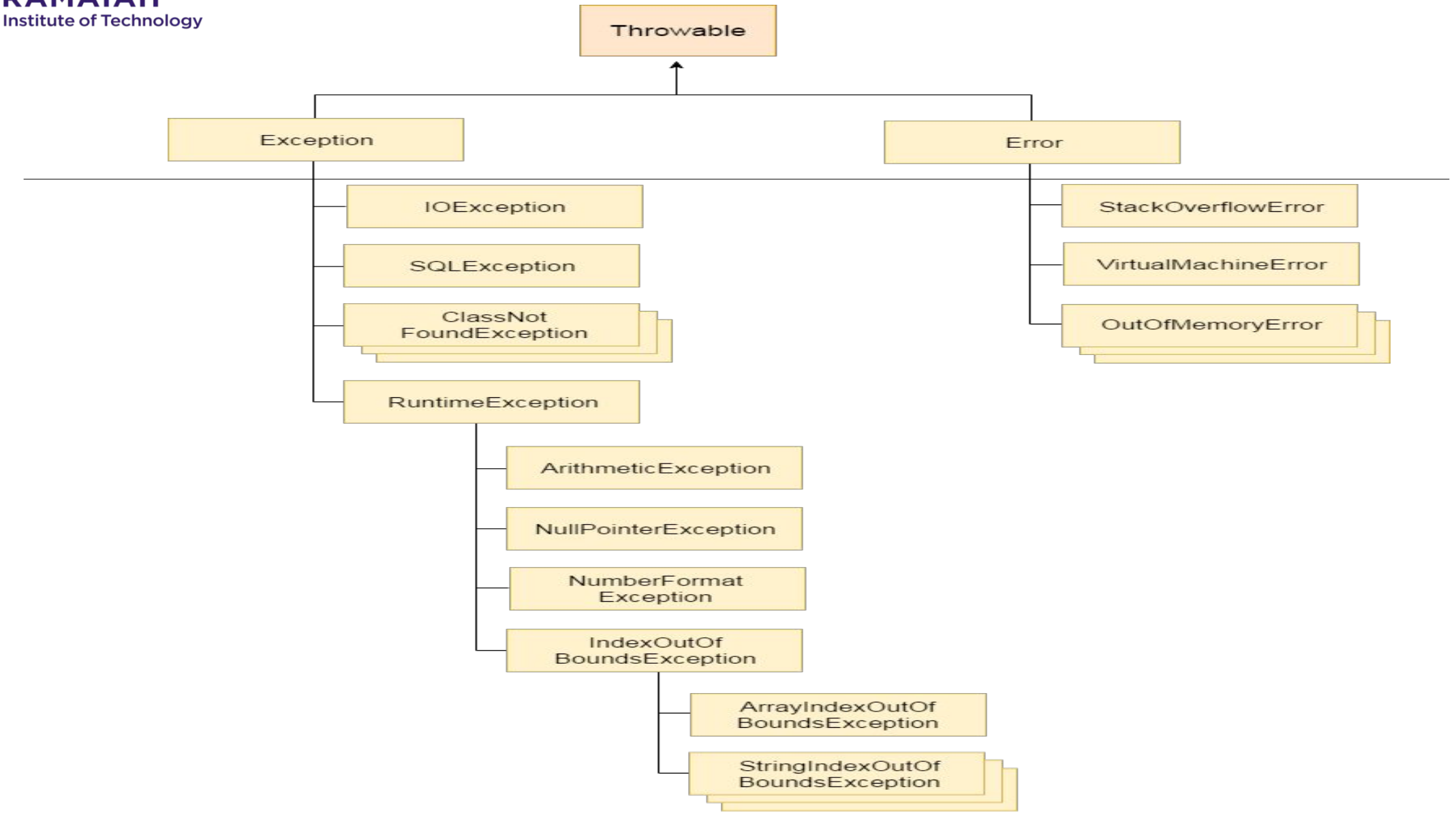
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

```
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 your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

Throwable




Throwable
Exception
Error
IOException
SQLException
ClassNot FoundException
RuntimeException
StackOverflowError
VirtualMachineError
OutOfMemoryError
ArithmeticException
NullPointerException
NumberFormat Exception
IndexOutOf BoundsException
ArrayIndexOutOf BoundsException
StringIndexOutOf BoundsException

Throwable

- All exception types are subclasses of the built- in class **Throwable**.
- **Exception** is a subclass of Throwable.
- **Exception class** is used for catching exceptional conditions arise in the user programs.
- This is also the class that you will subclass to **create your own custom exception types.**
- There is an important subclass of **Exception**, called **RuntimeException**.

Throwable

- **Error** is another subclass of Throwable.
- These are not exceptions at all, but problems that arise beyond the control of the user or the programmer.
- Errors are typically ignored in your code because you can rarely do anything about an error.
- For example, if a stack overflow occurs, an error will arise.

Types of Exception

- Two Types

1. Checked Exception
2. Unchecked Exception

- **Checked Exception**: A checked exception is an exception that occurs at the compile time.
- *Example :* File that need to be opened is not found. These type of exceptions must be checked at compile time.

Types of Exception

- **Unchecked Exception**: An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**.
- These include programming bugs, such as logic errors or improper use of an API.
- Runtime exceptions are ignored at the time of compilation.

Exception Handling Mechanism

```
class Exc0 {
        public static void main(String args[]) {
                int d = 0;
                int a = 42 / d;
        }
}

Output:
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```


Main.java

Exception Handling Mechanism

```
class Exc1 {
        static void subroutine() {
                int d = 0;
                int a = 10 / d;
        }
        public static void main(String args[]) {
                Exc1.subroutine();
        }
}

Output:
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

Main1.java

Exception Handling Mechanism

- To make use of Exception Handling we have to use mainly 5 keywords.
  1. **try**
  2. **catch**
  3. **throw**
  4. **throws**
  5. **finally**

Exception Handling Mechanism

- **try:**
  - It is a block of executable statements which may or may not throw an expression.

- **catch:**
  - It is a method which is used to catch the Exception arises in the try block and which is thrown by Java Runtime Environment(JRE).

Exception Handling Mechanism

- **throw:**
  - To manually throw an exception, use the keyword **throw**.

  throws:
  - Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- **finally:**
  - Any code that absolutely must be executed after a try block completes is put in a finally block.

Exception Handling Mechanism

```
class Exc2 {
        public static void main(String args[]) {
                int d, a;
                try { // monitor a block of code.
                        d = 0;
                        a = 42 / d;
                        System.out.println("This will not be printed.");
                }
                catch (ArithmeticException e) { // catch divide-by-zero error
                        System.out.println("Division by zero.");
                }
                System.out.println("After catch statement.");
        }
}

Output:
Division by zero.
After catch statement.
```

Main2.java

Exception Handling Mechanism

- **Note:**

1. try must be followed by catch
2. try & catch must be perfect pairs.
3. A try can be followed by multiple catches.

Generating random numbers in Java

Java provides three ways to generate random numbers using some built-in methods and classes as listed below:


Main3.java

java.util.Random class

Math.random method : Can Generate Random Numbers of double type.

ThreadLocalRandom class

1) java.util.Random For using this class to generate random numbers, we have to first create an instance of this class and then invoke methods such as nextInt(), nextDouble(), nextLong() etc using that instance.

We can generate random numbers of types integers, float, double, long, booleans using this class.

We can pass arguments to the methods for placing an upper bound on the range of the numbers to be generated. For example, nextInt(6) will generate numbers in the range 0 to 5 both inclusive.

RAMAIAH
Institute of Technology

Exception Handling Mechanism

```
try {
    // block of code to monitor for errors
}

catch (ExceptionType1 exOb) {
    // exception handler for ExceptionType1
}

catch (ExceptionType2 exOb) {
    // exception handler for ExceptionType2
}
// ...
finally {
    // block of code to be executed after try block ends
}
```

Multiple Catch Statements

- When you use multiple catch statements, exception subclasses must come before any of their super classes.
- This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses.
- Thus, a subclass would never be reached if it came after its superclass.
- In Java, unreachable code is an error.

Multiple Catch Statements

```
// This program contains an error.

class SuperSubCatch {
        public static void main(String args[]) {
                try {
                        int a = 0;
                        int b = 42 / a;
                }
                catch(Exception e) {
                        System.out.println("Generic Exception catch.");
                }
                // This catch is never reached because
                //ArithmeticException is a subclass of Exception.

                catch(ArithmeticException e) { // ERROR - unreachable
                        System.out.println("This is never reached.");
                }
        }
}
```

Multiple Catch Statements

```
// Demonstrate multiple catch statements.
class MultipleCatches {
        public static void main(String args[]) {
                try {
                        int a = args.length;
                        System.out.println("a = " + a);
                        int b = 42 / a;
                        int c[] = { 1 };
                        c[42] = 99;
                }
                catch(ArithmeticException e) {
                        System.out.println("Divide by 0: " + e);
                }
                catch(ArrayIndexOutOfBoundsException e) {
                        System.out.println("Array index oob: " + e);
                }
                System.out.println("After try/catch blocks.");
        }
}
```

Main4.java

Nested try Statements

- The try statement can be nested.
- That is, a try statement can be inside the block of another try.
- Each time a try statement is entered , the context of that exception is pushed on the stack.
- If an inner try statement does not have catch handler for a particular exception the stack is unwound and the next try statement's catch handler's are inspected for a match.
- This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.
- If no catch statement matches, then Java Run-time system will handle the exception.

RAMAIAH
Institute of Technology

Nested try Statements

```
class NestTry {
        public static void main(String args[]) {
                try {
                        int a = args.length;
                        int b = 42 / a;
                        System.out.println("a = " + a);
                        try {           // nested try block
                                if(a==1)
                                        a = a/(a-a); // division by zero
                                if(a==2) {
                                        int c[] = { 1 };
                                        c[42] = 99;
                                        // generate an out-of-bounds exception
                                }
                        }
                        catch(ArrayIndexOutOfBoundsException e) {
                        System.out.println("Array index out-of-bounds: " + e);
                        }
                }
                catch(ArithmeticException e) {
                        System.out.println("Divide by 0: " + e);
        }
}
```

Nested try Statements

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
  java.lang.ArrayIndexOutOfBoundsException:42
```

Throw

- Normally JRE will identify the Exception and throw the Exception but as a developer if you want to throw manually you have to use:throw" or "throws".
- "throw" is to throw Exception manually in the "body level".
- **General form:**

throw ThrowableInstance;

Throw

```
// Demonstrate throw.
class ThrowDemo {
        static void demoproc() {
                try {
                        throw new NullPointerException("demo");
                }
                catch(NullPointerException e) {
                        System.out.println("Caught inside demoproc.");
                        throw e; // rethrow the exception
                }
        }
        public static void main(String args[]) {
                try {
                        demoproc();
                }
                catch(NullPointerException e) {
                        System.out.println("Recaught: " + e);
                }
        }
}
```

Main6.java

Throws

- 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.
- Including a **throws** clause in the method's declaration.
- A throws clause lists the types of exceptions that a method might throw.
- This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.

General Form of Throws

```
type method-name(parameter-list) throw exception-list
{
        // body of method
}
```

Throws

```
class ThrowsDemo
{
        static void throwOne() throws IllegalAccessException
        {
                System.out.println("Inside throwOne.");
                throw new IllegalAccessException("demo");
        }
        public static void main(String args[])
        {
                try {
                        throwOne();
                }
                catch (IllegalAccessException e) {
                        System.out.println("Caught " + e);
                }
        }
}
```

Main7.java

finally

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method.
- This could be a problem in some methods.
- For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism.

finally

- The **finally** keyword is designed to address this contingency.
- **finally** creates a block of code that will be executed after a **try /catch** block has completed and before the code following the **try/catch** block.
- The **finally** block will execute whether or not an exception is thrown.

finally

- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.
- Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed.

finally

```
// Demonstrate finally.
class FinallyDemo {
        // Throw an exception out of the method.
        static void procA() {
                try {
                        System.out.println("inside procA");
                        throw new RuntimeException("demo");
                }
                finally {
                        System.out.println("procA's finally");
                }
        }
        // Return from within a try block.
        static void procB() {
                try {
                        System.out.println("inside procB");
                        return;
                }
                finally {
                        System.out.println("procB's finally");
                }
        }
```

Main8.java

RAMAIAH Institute of Technology

finally

```
// Execute a try block normally.
static void procC() {
        try {
                System.out.println("inside procC");
        }
        finally {
                System.out.println("procC's finally");
        }
}

public static void main(String args[]) {
        try {
                procA();
        }
        catch (Exception e) {
                System.out.println("Exception caught");
        }
        procB();
        procC();
}
}
```

Java's Built-in Exceptions

- Java defines several exception classes inside the standard package **java.lang**.
- Unchecked exception or RuntimeException need not be included in the method's **throws** list.
- Compiler does not check to see if a method handles or throws these exceptions.
- Checked exception must be included in the method's **throws** list.

RAMAIAH
Institute of Technology

Java's Unchecked RuntimeException Subclasses in java.lang

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.

Java's Checked Exceptions in java.lang

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Creating Your Own Exception Subclasses

- Java's built-in exceptions handle most common errors.
- You need to create your own exception types to handle situations specific to the applications.
- Define a subclass of **Exception.**
- Your subclasses don't need to actually implement anything.
- The **Exception** class does not define any methods of its own.
- It inherits those methods from **Throwable.**

Creating Your Own Exception Subclasses

- Exception defines four public constructors.
  - **Exception( )**: This creates an exception that has no description.
  - **Exception(String *msg)***:* This form lets you specify a description of the exception.
  - **Exception(Throwable cause)**: Constructs a new exception with the specified cause and a detailed message.
  - **Exception(String message, Throwable cause)**: Constructs a new exception with the specified detail message and cause.

Creating Your Own Exception Subclasses

```
// This program creates a custom exception type.
class MyException extends Exception {
  private int detail;

  MyException(int a) {
    detail = a;
  }

  public String toString() {
    return "MyException[" + detail + "]";
  }
}
```

Creating Your Own Exception Subclasses

```
class ExceptionDemo {
   static void compute(int a) throws MyException {
      System.out.println("Called compute(" + a + ")");
      if(a > 10)
         throw new MyException(a);
      System.out.println("Normal exit");
   }

   public static void main(String args[]) {
      try {
         compute(1);
         compute(20);
      } catch (MyException e) {
         System.out.println("Caught " + e);
      }
   }
}
```

Custom Exception Example

```
class LowCGPA extends Exception {
        double CGPA;
        public LowCGPA(double CGPA) {
                this.CGPA = CGPA;
        }
        public String toString() {
                return "Low CGPA : " + CGPA + "\n Not Eligible for Placement" ;
        }
}
```

Custom Exception Example

```
public class TestException{
        static void compute(double cgpa) throws LowCGPA {
                if(cgpa < 6)
                        throw new LowCGPA(cgpa);
                else
                        System.out.println("Eligible for Placement");
        }
        public static void main(String[] args) {
                try {
                        compute(5);
                }
                catch (LowCGPA e) {
                        System.out.println("Caught " + e);
                }
        }
}
```

Custom Exception Example 2

```
class LowBalanceException extends Exception{
        public String toString(){
                return "Your Account has Insufficient Funds : ";
        }
}

class InCorrectAmountException extends Exception{
        public String toString(){
                return "Entered Amount should be in multiples of 100 : ";
        }
}
```

Custom Exception Example 2

```
class Account{
        double balance;
        Account(){
                balance = 1000;
        }
        Account(double bal){
                balance = bal;
        }
        public void withdrawl(double x) throws LowBalanceException{
                if((balance - x) < 500){
                        throw new LowBalanceException();
                }
                else{
                        balance = balance - x;
                        System.out.println("Amount Deducted ");
                        System.out.println("Updated balance : " + balance);
                }
        }
        public void deposit(double x){
                balance = balance + x;
                System.out.println("Amount Deposited ");
                System.out.println("Updated balance : " + balance);
        }
}
```

Custom Exception Example 2

```
public class TestAccountException{
        public static void main(String[] args){
                Account obj = new Account(1000);
                Account obj1 = new Account(10000);
                int x = 600;
                try{
                        if(x % 100 != 0 ){
                                throw new InCorrectAmountException();
                        }
                        else{
                                obj.withdrawl(x);
                                obj1.deposit(x);
                        }
                }
                catch(LowBalanceException e){
                        System.out.println(e);
                }
                catch(InCorrectAmountException e){
                        System.out.println(e);
                }
        }
}
```

Introduction to Java Threads

- Thread is a lightweight components and it is a flow of control.
- It is a part of process or a Single path of execution.
- Threads share the same address space and therefore can share both data and code.
- Threads allow different tasks to be performed concurrently.

Chained Exception in Java

- Chained Exception was added to Java in JDK 1.4.
- This feature allows you to relate one exception with another exception, i.e one exception describes cause of another exception.
- For example, consider a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero.
- The method will throw only **ArithmeticException** to the caller. So the caller would not come to know about the actual cause of exception.
- Chained Exception is used in such type of situations.

Chained Exception in Java

Two new constructors and two new methods were added to **Throwable** class to support chained exception.

Throwable(Throwable cause)

Throwable(String str, Throwable cause)

In the first constructor, the paramter **cause** specifies the actual cause of exception. In the second form, it allows us to add an exception description in string form with the actual cause of exception.

getCause() and **initCause()** are the two methods added to **Throwable** class.

Chained Exception in Java

getCause() method returns the actual cause associated with current exception.

initCause() set an underlying cause(exception) with invoking exception.

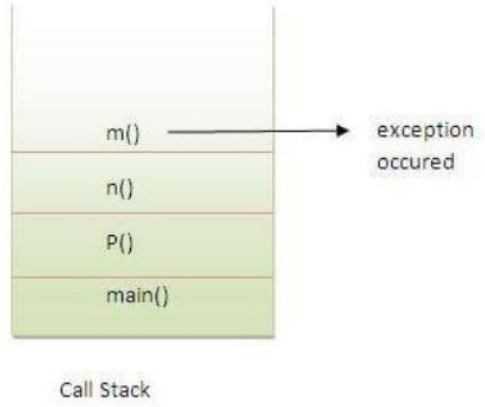
ChainedException.java

ChainedDemo1.java

Exception Propagation

Demo.java TestExceptionPropagation1.java TestExceptionPropagation2.java

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method,If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.This is called exception propagation.



In the example exception occurs in m() method where it is not handled,so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.
Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

Introduction to Java Threads

- Multithreading in java is a process of executing multiple threads simultaneously.
- The aim of multithreading is to achieve the concurrent execution.
- But we use multithreading than multiprocessing because threads share a common memory area.
- They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- Java Multithreading is mostly used in games, animation etc.

Introduction to Multithreading

- Multitasking means when multiple processes share common processing resources such as a CPU.
  - Process-based Multitasking(Multiprocessing)
  - Thread-based Multitasking(Multithreading)

Introduction to Multithreading

Process-based Multitasking (Multiprocessing)

- Each process has its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

Thread-based Multitasking (Multithreading)

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

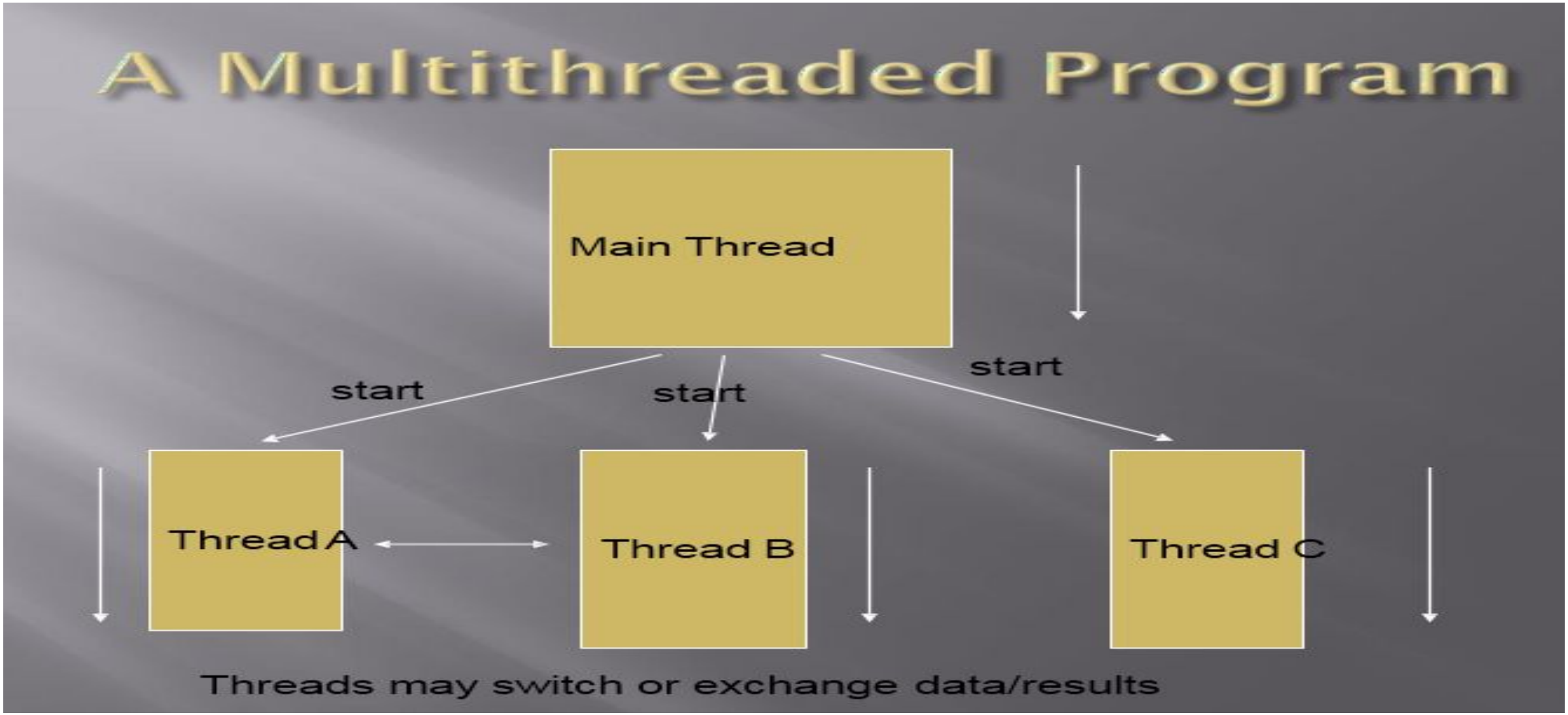
A single threaded program

```
Class ABC
{
….
public void main(..)
{
…
..
}
}
```

begin

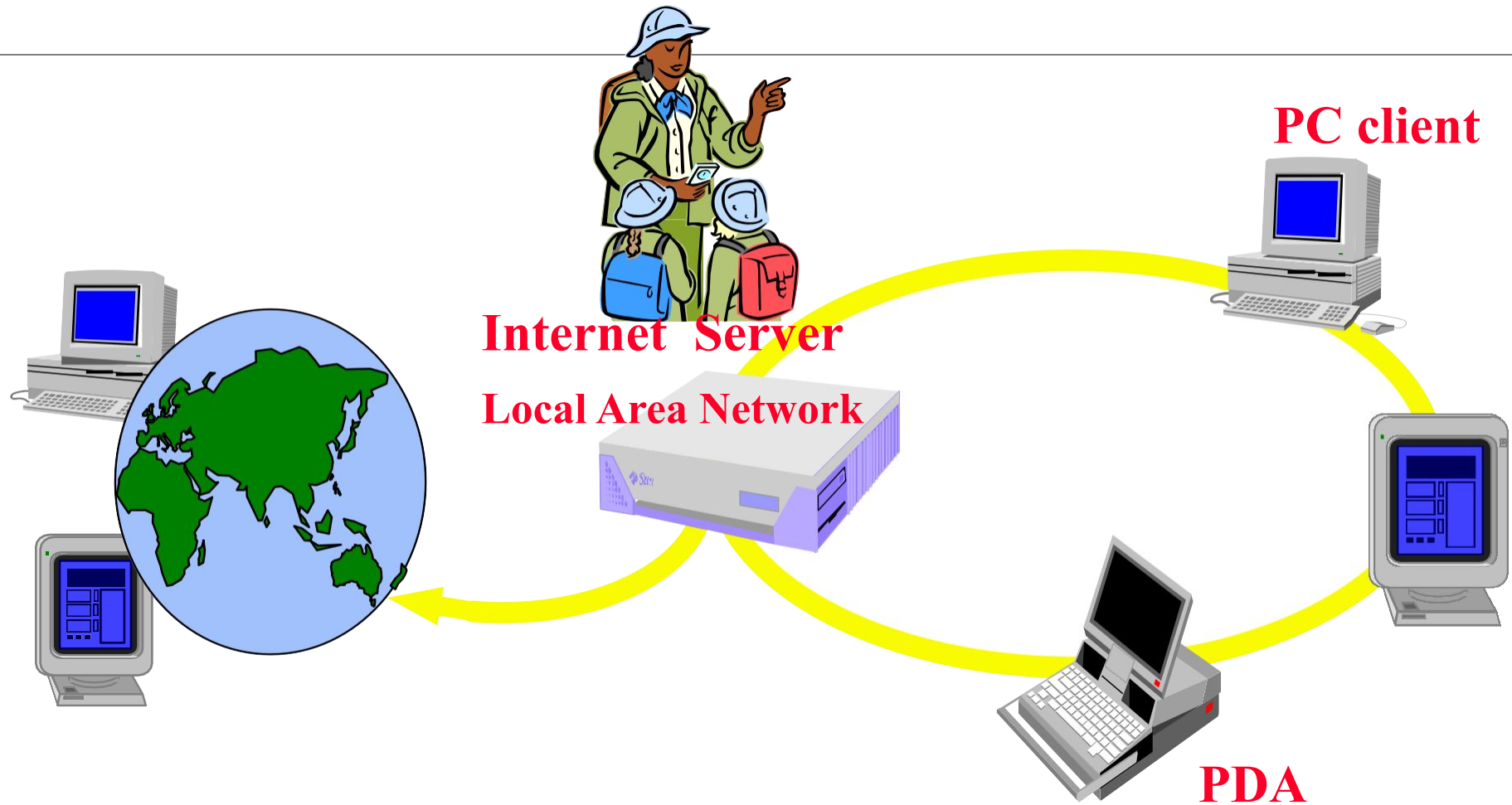
body

end


A Multithreaded Program
Main Thread
start
start
start
Thread A
Thread B
Thread C
Threads may switch or exchange data/results

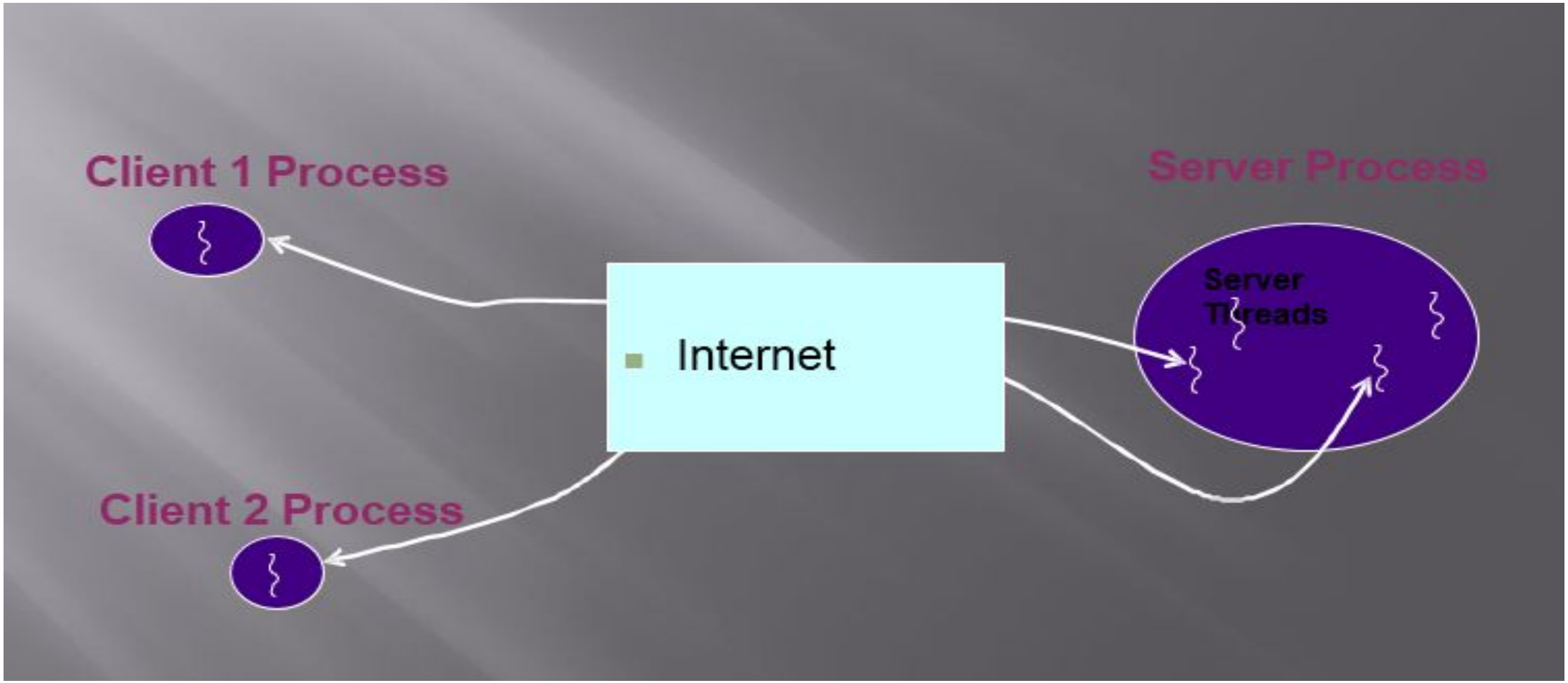
RAMAIAH Institute of Technology

Web/Internet Applications: Serving Many Users Simultaneously

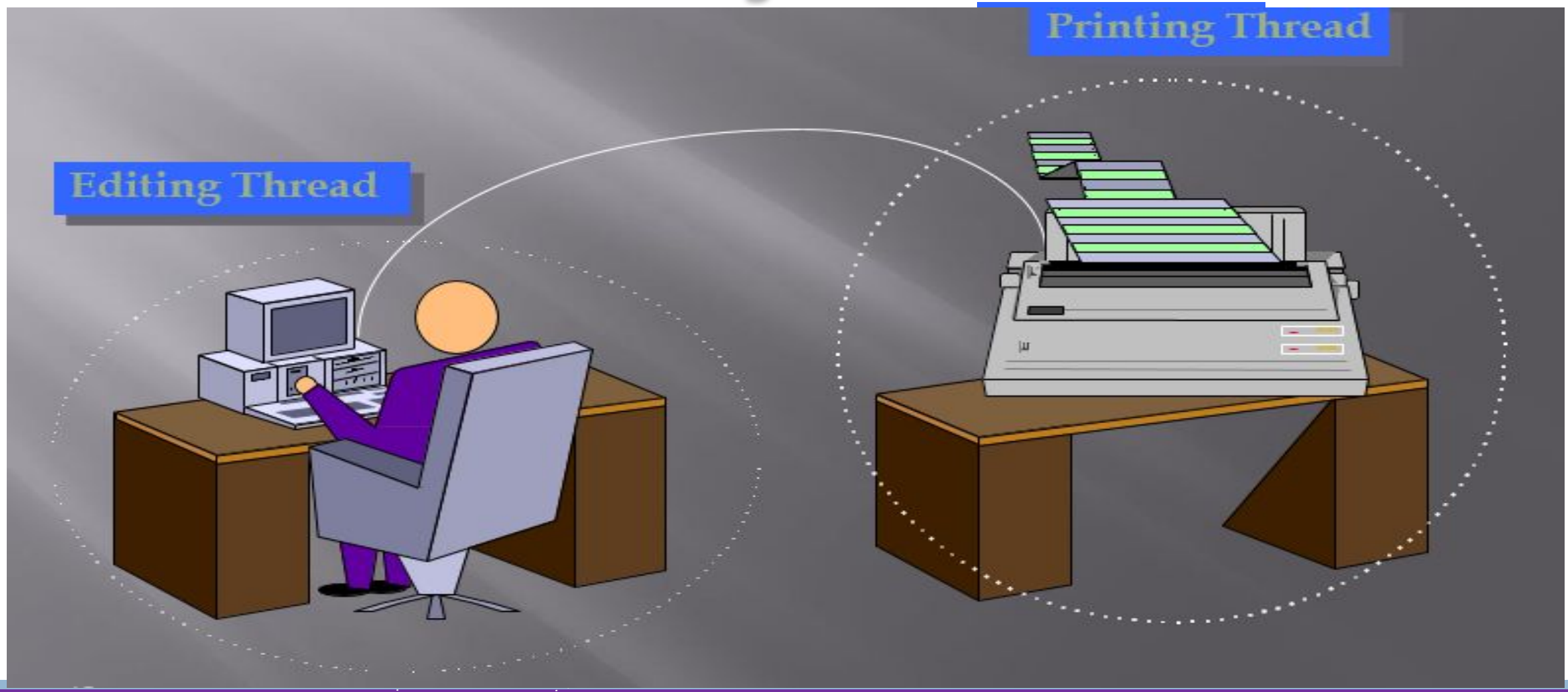


RAMAIAH Institute of Technology

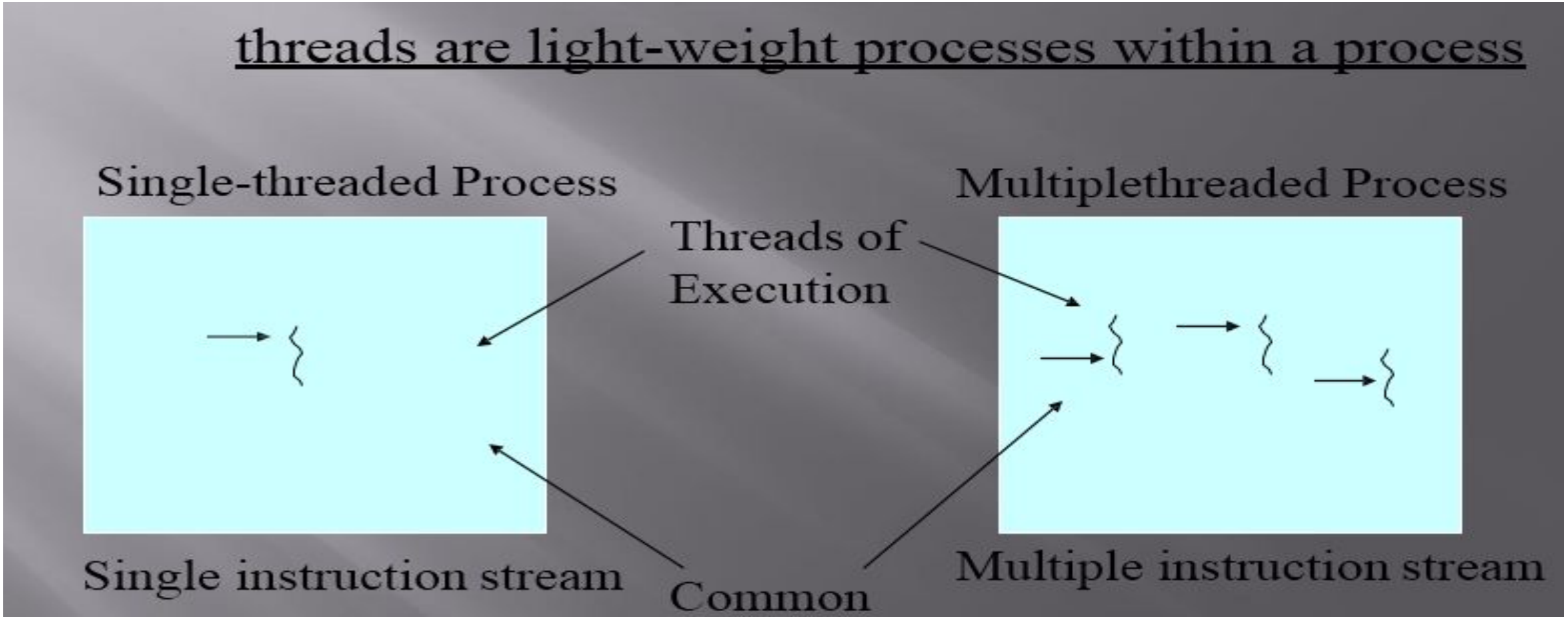
Multithreaded Server: For Serving Multiple Clients Concurrently



Modern Applications need Threads (ex1): Editing and Printing documents in background.



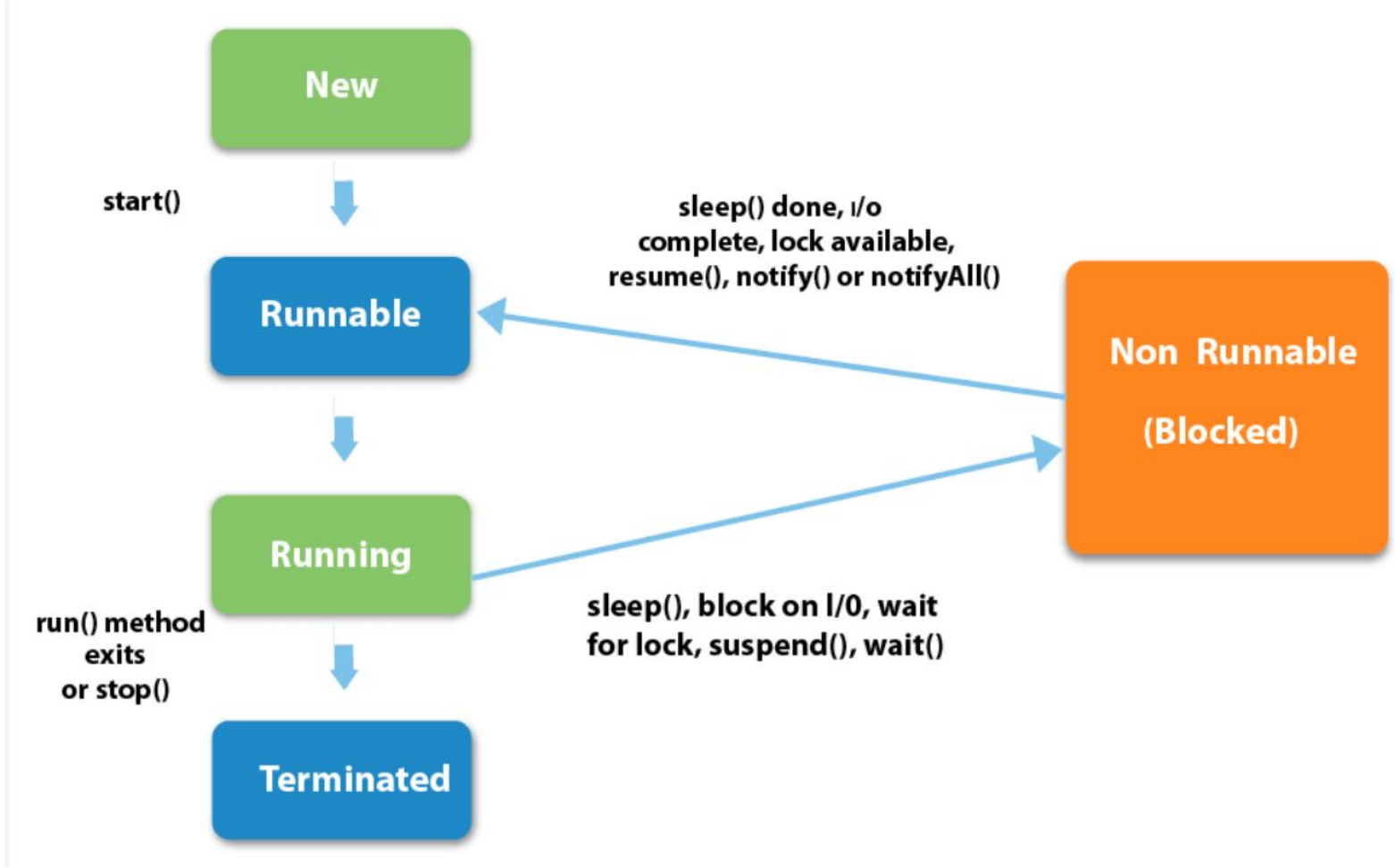
Single and Multithreaded Processes



The Java Thread Model

- The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind.
- Java uses threads to enable the entire environment to be asynchronous.
- This helps reduce inefficiency by preventing the waste of CPU cycles.

State or Life cycle of thread



State or Life cycle of thread

New State

- If any new thread class is created that represent new state of a thread.
- In new state, thread is created and about to enter into main memory.
- No memory is available if the thread is in new state.

Ready State

- In ready state thread will enter into main memory.
- Memory space is allocated and waiting for the CPU.

State or Life cycle of thread

Waiting:

- Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task.
- A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Running State

- Whenever the thread is executing.

Halted or dead State

- If the thread execution is stopped permanently than it comes under dead state, no memory is available for the thread if its comes to dead state

Multithreading in Java

- 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.
- The Java Virtual machine has its own runtime threads
- Injavalanguage multithreading can be achieved in two different ways.
- Using thread class
- Using Runnable interface

The Main Thread

- When a Java program starts up, one thread begins running immediately.
- This is usually the main thread of your program, because it is the one that is executed when your program begins.
- The main thread is important for two reasons:
  - It is the thread from which other "child" threads will be spawned.
  - Often, it must be the last thread to finish execution because it performs various shutdown actions.
- Main thread is created automatically and can be controlled through a Thread object.
- Obtain a reference to it by calling the method currentThread( ).

The Main Thread

```
// Controlling the main Thread.
class CurrentThreadDemo {
        public static void main(String args[]) {
                Thread t = Thread.currentThread();
                System.out.println("Current thread: " + t);
                // change the name of the thread
                t.setName("My Thread");
                System.out.println("After name change: " + t);
                try {
                        for(int n = 5; n > 0; n--) {
                                System.out.println(n);
                                Thread.sleep(1000);
                        }
                }
                catch (InterruptedException e) {
                        System.out.println("Main thread interrupted");
                }
        }
}
```

The Main Thread

- Output:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

- This displays, in order: the name of the thread, its priority, and the name of its group.
- By default, the name of the main thread is main.
- Its priority is 5.
- And main is also the name of the group of threads to which this thread belongs.

Creating a Thread

1. Java defines two ways in which this can be accomplished:
2. 1.You can implement the Runnable interface.
3. 2.You can extend the Thread class, itself.

Implementing Runnable

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

- Runnable abstracts a unit of executable code.
- You can construct a thread on any object that implements Runnable.
- To implement Runnable, a class need only implement a single method called run( ), which is declared like this:

■public void run( )

- Inside run( ), you will define the code that constitutes the new thread.

Implementing Runnable

- After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class.
- Thread(Runnable threadOb, String threadName)
- After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread.
- In essence, start( ) executes a call to run( ).

Implementing Runnable

```
// Create a second thread.
class NewThread implements Runnable {
        Thread t;
        NewThread() {
                // Create a new, second thread
                t = new Thread(this, "Demo Thread");
                System.out.println("Child thread: " + t);
                t.start(); // Start the thread
        }
        // This is the entry point for the second thread.
        public void run() {
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println("Child Thread: " + i);
                                Thread.sleep(500);
                        }
                }
                catch (InterruptedException e) {
                        System.out.println("Child interrupted.");
                }
                System.out.println("Exiting child thread.");
        }
}
```

Implementing Runnable

```
class ThreadDemo {
        public static void main(String args[ ] ) {
                new NewThread(); // create a new thread
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println("Main Thread: " + i);
                                Thread.sleep(1000);
                        }
                }
                catch (InterruptedException e) {
                        System.out.println("Main thread interrupted.");
                }
                System.out.println("Main thread exiting.");
        }
}
```

Implementing Runnable

Output:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.  Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Extending Thread

- The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.

- The extending class must override the run( ) method, which is the entry point for the new thread.

- It must also call start( ) to begin execution of the new thread.

Extending Thread

```
// Create a second thread by extending Thread
class NewThread extends Thread {
        NewThread() {
                // Create a new, second thread
                super("Demo Thread");
                System.out.println("Child thread: " + this);
                start(); // Start the thread
        }
        // This is the entry point for the second thread.
        public void run() {
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println("Child Thread: " + i);
                                Thread.sleep(500);
                        }
                }
                catch (InterruptedException e) {
                        System.out.println("Child interrupted.");
                }
                System.out.println("Exiting child thread.");
        }
}
```

Extending Thread

```
class ExtendThread {
        public static void main(String args[]) {
                new NewThread(); // create a new thread
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println("Main Thread: " + i);
                                Thread.sleep(1000);
                        }
                }
                catch (InterruptedException e) {
                        System.out.println("Main thread interrupted.");
                }
                System.out.println("Main thread exiting.");
        }
}
```

Choosing an Approach

- The Thread class defines several methods that can be overridden by a derived class.
- Of these methods, the only one that must be overridden is run( ).
- This is the same method required when you implement Runnable.
- Classes should be extended only when they are being enhanced or modified in some way.
- So, if you will not be overriding any of Thread's other methods, it is probably best simply to implement Runnable.
- By implementing Runnable, your thread class can inherit a different class.

RAMAIAH Institute of Technology

Thread Object Creation

- **If Runnable interface is implemented:**

```
class FirstThread implements Runnable{  public void run(){
System.out.println("Running FirstThread");
}
}
public TestThread{
public static void main(String args[]){
FirstThread obj1 = new FirstThread();
Thread threadA = new Thread(obj1);
threadA.start();
}
}
```

Thread Object Creation

- **If Thread class is extended:**

```
class SecondThread implements Runnable{  public void
        run(){
System.out.println("Running SecondThread ");
}
}
public TestThread{
public static void main(String args[]){
SecondThread threadB= new SecondThread();
threadB.start();
}
}
```

RAMAIAH Institute of Technology

Thread Examples

Java Program to create 2 threads. Print Even numbers from 1 to 50 using Thread1. Print Odd numbers from 1 to 50 using Thread2. After Every number, put the thread to sleep.

```
class MultithreadDemo1 extends Thread{
        public void run(){
                System.out.println("Thread 1");
                for(int i=0; i<=50; i++){
                        try{
                                if(i%2 == 0){
                                        System.out.println("Even : " + i);
                                        Thread.sleep(500);
                                }
                        }
                        catch(InterruptedException e){
                        }
                }
        }
}
```

Thread Examples

```
class MultithreadDemo2 extends Thread{
        public void run(){
                System.out.println("Thread 2");
                for(int i=0; i<=50; i++){
                        try{
                                if(i%2 == 1){
                                        System.out.println("Odd : " + i);
                                        Thread.sleep(500);
                                }
                        }
                        catch(InterruptedException e){
                        }
                }
        }
}


public class Main{
        public static void main(String args[]){
                MultithreadDemo1 obj1=new MultithreadDemo1();
                obj1.start();
                MultithreadDemo2 obj2=new MultithreadDemo2();
                obj2.start();
        }
}
```

RAMAIAH Institute of Technology

Thread Examples

Java Program to create 2 threads. Print Prime Numbers from 1 to 50 using Thread1. Print Prime Numbers from 100 to 150 using Thread2. After Every number, put the thread to sleep. Create Threads using Runnable Interface

```
class MultithreadDemo1 implements Runnable{
        public void run(){
                System.out.println("Thread 1");
                for(int i=0; i<=50; i++){
                        try{
                                int counter=0;
                                for(int num =i; num>=1; num--){
                                        if(i%num==0){
                                                counter = counter + 1;
                                        }
                                }
                                if (counter ==2){
                                        System.out.println("Prime Number(1 - 50): " + i);
                                        Thread.sleep(500);
                                }
                        }
                        catch(InterruptedException e){
                                System.out.println("Caught: " + e);
                        }
                }
        }
}
```

Thread Examples

```
class MultithreadDemo2 implements Runnable{
        public void run(){
                System.out.println("Thread 2");
                for(int i=100; i<=150; i++){
                        try{
                                int counter=0;
                                for(int num =i; num>=1; num--){
                                        if(i%num==0){
                                                counter = counter + 1;
                                        }
                                }
                                if (counter ==2){
                                        System.out.println("Prime Number(100 - 150): " + i);
                                        Thread.sleep(500);
                                }
                        }
                        catch(InterruptedException e){
                                System.out.println("Caught: " + e);
                        }
                }
        }
}
public class Main{
        public static void main(String args[]){
                MultithreadDemo1 obj1=new MultithreadDemo1();
                Thread threadobj1 = new Thread(obj1);
                threadobj1.start();
                MultithreadDemo2 obj2=new MultithreadDemo2();
                Thread threadobj2 = new Thread(obj2);
                threadobj2.start();
        }
}
```

Can we start a thread twice

- No. After starting a thread, it can never be started again.
- If you does so, an IllegalThreadStateException is thrown.
- In such case, thread will run once but for second time, it will throw excep

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

Output:
Exception in thread "main" java.lang.IllegalThreadStateException
        at java.base/java.lang.Thread.start(Thread.java:794)
        at Main.main(Main.java:16)
running...
```

Thread Priority

- In Java, each thread is assigned priority, which affects the order in which it is scheduled for running.
- The threads so far had same default priority (NORM_PRIORITY) and they are served using FCFS policy.
- Java allows users to change priority:
- ThreadName.setPriority(intNumber)
  - MIN_PRIORITY = 1
  - NORM_PRIORITY=5
  - MAX_PRIORITY=10

Thread Priority Example

```
class A extends Thread{
        public void run(){
                System.out.println("Thread A started");
                for(int i=1;i<=4;i++){
                        System.out.println("\t From ThreadA: i= "+i);
                }
                System.out.println("Exit from A");
        }
}

class B extends Thread{
        public void run(){
                System.out.println("Thread B started");
                for(int j=1;j<=4;j++){
                        System.out.println("\t From ThreadB: j= "+j);
                }
                System.out.println("Exit from B");
        }
}
```

Thread Priority Example

```
class C extends Thread{
        public void run(){
                System.out.println("Thread C started");
                for(int k=1;k<=4;k++){
                        System.out.println("\t From ThreadC: k= "+k);
                }
                System.out.println("Exit from C");
        }
}
```

Thread Priority Example

```
class ThreadPriority{
        public static void main(String args[]){
                A threadA=new A();
                B threadB=new B();
                C threadC=new C();

                threadC.setPriority(Thread.MAX_PRIORITY);
                threadB.setPriority(threadA.getPriority()+1);
                threadA.setPriority(Thread.MIN_PRIORITY);

                System.out.println("Started Thread A");
                threadA.start();
                System.out.println("Started Thread B");
                threadB.start();
                System.out.println("Started Thread C");
                threadC.start();

                System.out.println("End of main thread");
        }
}
```

Thread Priority Example

- OUTPUT:

Started Thread A Started
Thread B Thread A started
From ThreadA: i= 1 From
ThreadA: i= 2 From ThreadA:
i= 3 From ThreadA: i= 4 Exit
from A

Started Thread C
Thread B started
From ThreadB: j= 1
From ThreadB: j= 2
From ThreadB: j= 3
From ThreadB: j= 4
Exit from B

Thread Methods

- Extending Thread Class is required to '***override run()***' method. The run method contains the actual logic to be executed by thread.
- Creation of thread object never starts execution, we need to call '***start()***' method to run a thread.
- ***join():*** It makes to wait for this thread to die. You can wait for a thread to finish by calling its join() method.
- ***sleep():*** It makes current executing thread to sleep for a specified interval of time. Time is in milli seconds.

Thread Methods

- ***yield():*** It makes current executing thread object to pause temporarily and gives control to other thread to execute.
- ***notify():*** This method is inherited from Object class. This method wakes up a single thread that is waiting on this object's monitor to acquire lock.
- ***notifyAll():*** This method is inherited from Object class. This method wakes up all threads that are waiting on this object's monitor to acquire lock.
- ***wait():*** This method is inherited from Object class. This method makes current thread to wait until another thread invokes the notify() or the notifyAll() for this object

A Program with Three Threads

```
class A extends Thread
{
public void run()
{
for(int i=1;i<=5;i++)
{
      ThreadA: i= "+i);
      }
      System.out.println("Exit from A");
      }

class B extends Thread
{
public void run()
{

for(int j=1;j<=5;j++)
{
      ThreadB: j= "+j);
      }
      System.out.println("Exit from B");
      }
```

```
class C extends Thread
{
public void run()
{

for(int k=1;k<=5;k++)
{
      k= "+k);
      }

      System.out.println("Exit from
      C");
      } }

      class ThreadTest
      {
      public static void main(String
      args[])

      {
      new A().start();
      new B().start();
      new C().start();

      } }
```

Run 1

```
[cse@lab1] threads [1:76] java ThreadTest  From
      ThreadA: i= 1
From ThreadA: i= 2  From ThreadA: i= 3
From ThreadA: i= 4  From ThreadA: i= 5
Exit from A
From ThreadC: k= 1  From ThreadC: k= 2  From
      ThreadC: k= 3  From ThreadC: k= 4  From
      ThreadC: k= 5
Exit from C
From ThreadB: j= 1  From ThreadB: j= 2  From
      ThreadB: j= 3  From ThreadB: j= 4  From
      ThreadB: j= 5
Exit from B
```

Run2

```
 cse@lab1] threads [1:77] java ThreadTest
      From ThreadA: i= 1
From ThreadA: i= 2  From ThreadA: i= 3
      From ThreadA: i= 4  From
      ThreadA: i= 5  From ThreadC: k= 1
      From ThreadC: k= 2  From
      ThreadC: k= 3  From ThreadC: k= 4
      From ThreadC: k= 5
Exit from C
From ThreadB: j= 1  From ThreadB: j= 2
      From ThreadB: j= 3  From ThreadB:
      j= 4  From ThreadB: j= 5
Exit from B  Exit from A
```

isAlive() and join()

- How will a one thread know whether another thread has ended or not?
- **isAlive()** and **join()** are two different methods to

check whether a thread has finished its execution.

- **Call isAlive()**
  - **General form:** final Boolean isAlive()
  - Returns true if thread is still running or false otherwise.
- **join()**
- **General form:** final void join() throws InterruptedException
  - Method waits until the thread on which it is called

terminates.

  - Name tells you calling thread waits until specified thread joins it.

Example of `isAlive` method

```
public class MyThread extends Thread
{
public void run()
    {
        System.out.println("r1 ");
        try{
         Thread.sleep(500);
     }catch(InterruptedException ie){}

        System.out.println("r2 ");
   }
public static void main(String[] args)
{
MyThread t1=new MyThread();
MyThread t2=new MyThread();
t1.start();
t2.start();
System.out.println(t1.isAlive());
System.out.println(t2.isAlive());
}
}
```

```
Output :
r1
true
true
r1
r2
r2
```

Example of thread without `join()` method

```
public class MyThread extends Thread
{
public void run()
   {
       System.out.println("r1 ");
       try{
        Thread.sleep(500);
    }catch(InterruptedException ie){}

       System.out.println("r2 ");
  }
public static void main(String[] args)
{
MyThread t1=new MyThread();
MyThread t2=new MyThread();
t1.start();
t2.start();
}
}
```

```
Output :
r1
r1
r2
r2
```

Example of thread with `join()` method

```
public class MyThread extends Thread
{
public void run()
   {
       System.out.println("r1 ");
       try{
        Thread.sleep(500);
    }catch(InterruptedException ie){}

       System.out.println("r2 ");
  }
public static void main(String[] args)
{
MyThread t1=new MyThread();
MyThread t2=new MyThread();
t1.start();

try{
  t1.join();            //Waiting for t1 to finish
}catch(InterruptedException ie){}

t2.start();
}
}
```

```
Output :
r1
r2
r1
r2
```

Using **join()** method, we tell our thread to wait until the specified thread completes its execution. There are overloaded versions of **join()** method, which allows us to specify time for which you want to wait for the specified thread to terminate.

Synchronization

When two or more threads need access to a shared resource, but only one thread uses it.

- Key to synchronization is the concept of the monitor.
- A monitor is an object that is used as a mutually exclusive lock.
- Only one thread can own a monitor at a given time.
- When a thread acquires a lock, all other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
- These other threads are said to be waiting for the monitor.
- A thread that owns a monitor can reenter the same monitor desires.

Using Synchronized Methods

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them
- To enter an object's monitor, call a method that has been modified with the synchronized keyword.
- While a thread is inside a synchronized method, all other threads that try to call it on the same instance have to wait.
- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

Without Synchronized Methods

```
// This program is not synchronized.
class Callme
{
        void call(String msg)
        {
                System.out.print("[" + msg);
                try {
                        Thread.sleep(1000);
                }
                catch(InterruptedException e) {
                        System.out.println("Interrupted");
                }
                System.out.println("]");
        }
}
```

Without Synchronized Methods

```
class Caller implements Runnable
{
        String msg;
        Callme target;
        Thread t;
        public Caller(Callme targ, String s)
        {
                target = targ;
                msg = s;
                t = new Thread(this);
                t.start();
        }

        public void run()
        {
                target.call(msg);
        }
}
```

Without Synchronized Methods

```
class Synch
{
        public static void main(String args[])
        {
                Callme target = new Callme();
                Caller ob1 = new Caller(target, "Hello");
                Caller ob2 = new Caller(target, "Synchronized");
                Caller ob3 = new Caller(target, "World");
                // wait for threads to end
                try {
                        ob1.t.join();
                        ob2.t.join();
                        ob3.t.join();
                }
                catch(InterruptedException e) {
                        System.out.println("Interrupted");
                }
        }
}
```

Without Synchronized Methods

```
Output:

[Hello[Synchronized[World]
]
]
```

```
Expected Output:

[Hello]
[Synchronized]
[World]
```

With Synchronized Methods

```
// This program is synchronized.
class Callme
{
        synchronized void call(String msg)
        {
                System.out.print("[" + msg);
                try {
                        Thread.sleep(1000);
                }
                catch(InterruptedException e) {
                        System.out.println("Interrupted");
                }
                System.out.println("]");
        }
}
```

With Synchronized Methods

```
class Caller implements Runnable
{
        String msg;
        Callme target;
        Thread t;
        public Caller(Callme targ, String s)
        {
                target = targ;
                msg = s;
                t = new Thread(this);
                t.start();
        }

        public void run()
        {
                target.call(msg);
        }
}
```

With Synchronized Methods

```
class Synch
{
        public static void main(String args[])
        {
                Callme target = new Callme();
                Caller ob1 = new Caller(target, "Hello");
                Caller ob2 = new Caller(target, "Synchronized");
                Caller ob3 = new Caller(target, "World");
                // wait for threads to end
                try {
                        ob1.t.join();
                        ob2.t.join();
                        ob3.t.join();
                }
                catch(InterruptedException e) {
                        System.out.println("Interrupted");
                }
        }
}
```

With Synchronized Methods

```
Output:

[Hello]
[Synchronized]
[World]
```

The synchronized Statement

- Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access.
- That is, the class does not use synchronized methods.
- Further, this class was not created by you, but by a third party, and you do not have access to the source code.
- Thus, you can't add synchronized to the appropriate methods within the class.
- You simply put calls to the methods defined by this class inside a **synchronized block.**

The synchronized Statement

```
// This program uses a synchronized block.
class Callme
{
        void call(String msg)
        {
                System.out.print("[" + msg);
                try {
                        Thread.sleep(1000);
                }
                catch (InterruptedException e) {
                        System.out.println("Interrupted");
                }
                System.out.println("]");
        }
}
```

The synchronized Statement

```
class Caller implements Runnable
{
        String msg;
        Callme target;
        Thread t;
        public Caller(Callme targ, String s)
        {
                target = targ;
                msg = s;
                t = new Thread(this);
                t.start();
        }
        // synchronize calls to call()
        public void run()
        {
                synchronized(target)
                {
                        target.call(msg);
                }
        }
}
```

The synchronized Statement

```
class Synch1
{
        public static void main(String args[])
        {
                Callme target = new Callme();
                Caller ob1 = new Caller(target, "Hello");
                Caller ob2 = new Caller(target, "Synchronized");
                Caller ob3 = new Caller(target, "World");
                // wait for threads to end
                try {
                        ob1.t.join();
                        ob2.t.join();
                        ob3.t.join();
                }
                catch(InterruptedException e) {
                        System.out.println("Interrupted");
                }
        }
}
```

Interthread Communication

- Synchronization unconditionally blocks other threads from asynchronous access to certain methods.
- But programmer can achieve a more subtle level of control through interprocess communication.
- Multithreading replacesevent loop programming by dividing your tasks into discrete, logical units.
- Threads also provide a secondary benefit: they do away with polling.
- Polling is usually implemented by a loop thatis used to check some condition.
- Once the condition is true, appropriate action is taken.
- This wastes CPU time.

RAMAIAH Institute of Technology

Interthread Communication

- To avoid polling, Java includes an elegant interprocess communication mechanism via following methods

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

  - These methodsare implemented as final methods in Object, so all classes have them.

RAMAIAH Institute of Technology

Interthread Communication

- **wait()** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify() or notifyAll( ).

- final void wait( ) throws InterruptedException

- notify() wakes up a thread that called wait() on the same object.

- final void notify( )

- notifyAll() wakes up all the threads that called wait() on the same object. C of the threads will be granted access.

- final void notify All( )

Producer – Consumer Example

```
// An incorrect implementation of a producer and consumer.
class Q {
  int n;

  synchronized int get() {
    System.out.println("Got: " + n);
    return n;
  }

  synchronized void put(int n) {
    this.n = n;
    System.out.println("Put: " + n);
  }
}
```

Producer – Consumer Example

```
class Producer implements Runnable {
   Q q;

   Producer(Q q) {
      this.q = q;
      new Thread(this, "Producer").start();
   }

   public void run() {
      int i = 0;

      while(true) {
         q.put(i++);
      }
   }
}
```

Producer – Consumer Example

```
class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }

  public void run() {
    while(true) {
      q.get();
    }
  }
}
```

Producer – Consumer Example

```
class PC {
  public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);

        System.out.println("Press Control-C to stop.");
  }
}
```

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

Producer – Consumer Example

```
// A correct implementation of a producer and consumer.
class Q {
  int n;
  boolean valueSet = false;

  synchronized int get() {
    while(!valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("InterruptedException caught");
      }

    System.out.println("Got: " + n);
    valueSet = false;
    notify();
    return n;
  }
```

Producer – Consumer Example

```
synchronized void put(int n) {
    while(valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("InterruptedException caught")
      }

      this.n = n;
      valueSet = true;
      System.out.println("Put: " + n);
      notify();
  }
}
```

Producer – Consumer Example

```
class Producer implements Runnable {
   Q q;

   Producer(Q q) {
      this.q = q;
      new Thread(this, "Producer").start();
   }

   public void run() {
      int i = 0;

      while(true) {
         q.put(i++);
      }
   }
}
```

Producer – Consumer Example

```
class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }

  public void run() {
    while(true) {
      q.get();
    }
  }
}
```

Producer – Consumer Example

```
class PCFixed {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);

    System.out.println("Press Control-C to stop.");
  }
}
```

```
Put: 1    Put: 3
Got: 1    Got: 3
Put: 2    Put: 4
Got: 2    Got: 4
```

RAMAIAH
Institute of Technology

Example Program

Write a Java Program to Print Multiplication table of 5 & 6 using Synchronized functions. After printing a number, put the thread to sleep.

Example Program

```
class computeTable{
        synchronized public void compute(int num){
                for(int i=1; i<=10; i++){
                        System.out.println(num + "*" + i + "=" + num*i );
                        try
                        {
                                Thread.sleep(1000);
                        }
                        catch(Exception ex)
                        {
                                System.out.println(ex);
                        }
                }
        }
}
```

Example Program

```
class MyThread extends Thread{
        public computeTable en;
        public int n;
        public MyThread(computeTable e,int num){
                en=e;
                n=num;
        }
        public void run(){
                en.compute(n);
        }
}
```

Example Program

```
public class MainThread{
        public static void main(String args[]){
                computeTable en = new computeTable();
                MyThread t1 = new MyThread(en , 5);
                MyThread t2 = new MyThread(en , 6);
                t1.start();
                t2.start();
        }
}
```

Thank you