

# OS Support for Building Distributed Applications: Multithreaded Programming using Java Threads



Object-Oriented  
Programming  
with **JAVA**  
Essentials and Applications

Rajkumar Buyya | S Thamarai Selvi | Xingchen Chu  
Copyrighted Material

*Dr. Rajkumar Buyya*

**Cloud** Computing and **D**istributed **S**ystems (CLOUDS) Laboratory  
School of Computing and Information Systems  
The University of Melbourne, Australia

<http://www.buyya.com>

# Agenda

- Introduction
- Thread Applications
- Defining Threads
- Java Threads and States
- Examples

# Introduction

- In a networked world, it is common practice to share resources among multiple users.
- Even on desktop computers, users typically run multiple applications and carry out some operations in the background (e.g., printing) and some in the foreground (e.g., editing) simultaneously.
- Modern programming languages and operating systems are designed to support the development of applications containing multiple activities that can be executed concurrently
- Preemptive vs. cooperative multitasking?
- Multitasking vs. multithreading?

# Threaded Applications

## ■ Modern Systems

- Multiple applications run concurrently!
- This means that... there are multiple processes on your computer



# A single threaded program

```
class ABC
```

```
{
```

```
....
```

```
    public void main(..)
```

```
    {
```

```
        ...
```

```
        ..
```

```
    }
```

```
}
```

begin

body

end



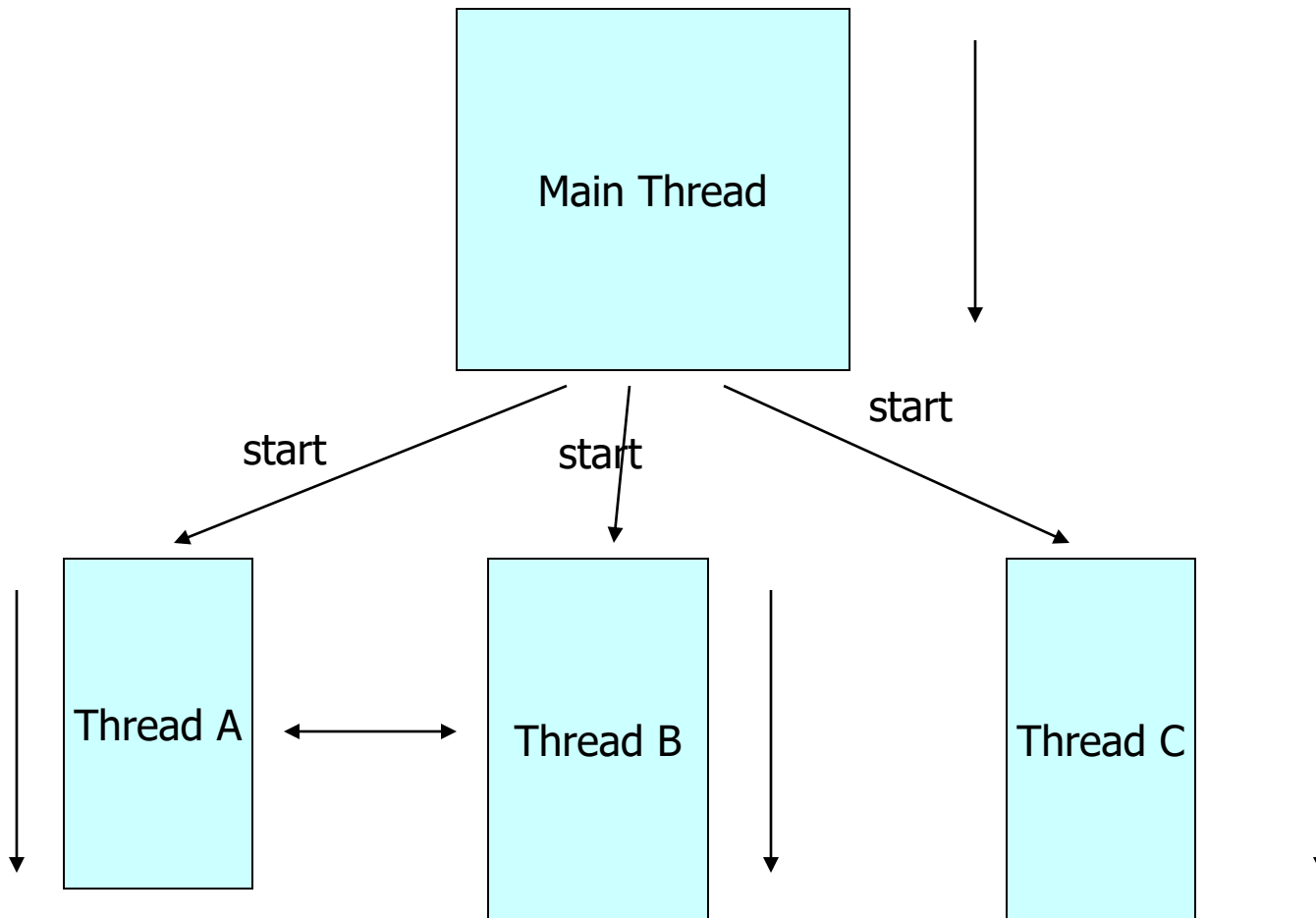
# Threaded Applications

## ■ Modern Systems

- Applications perform many tasks at once!
- This means that... there are multiple threads within a single process.



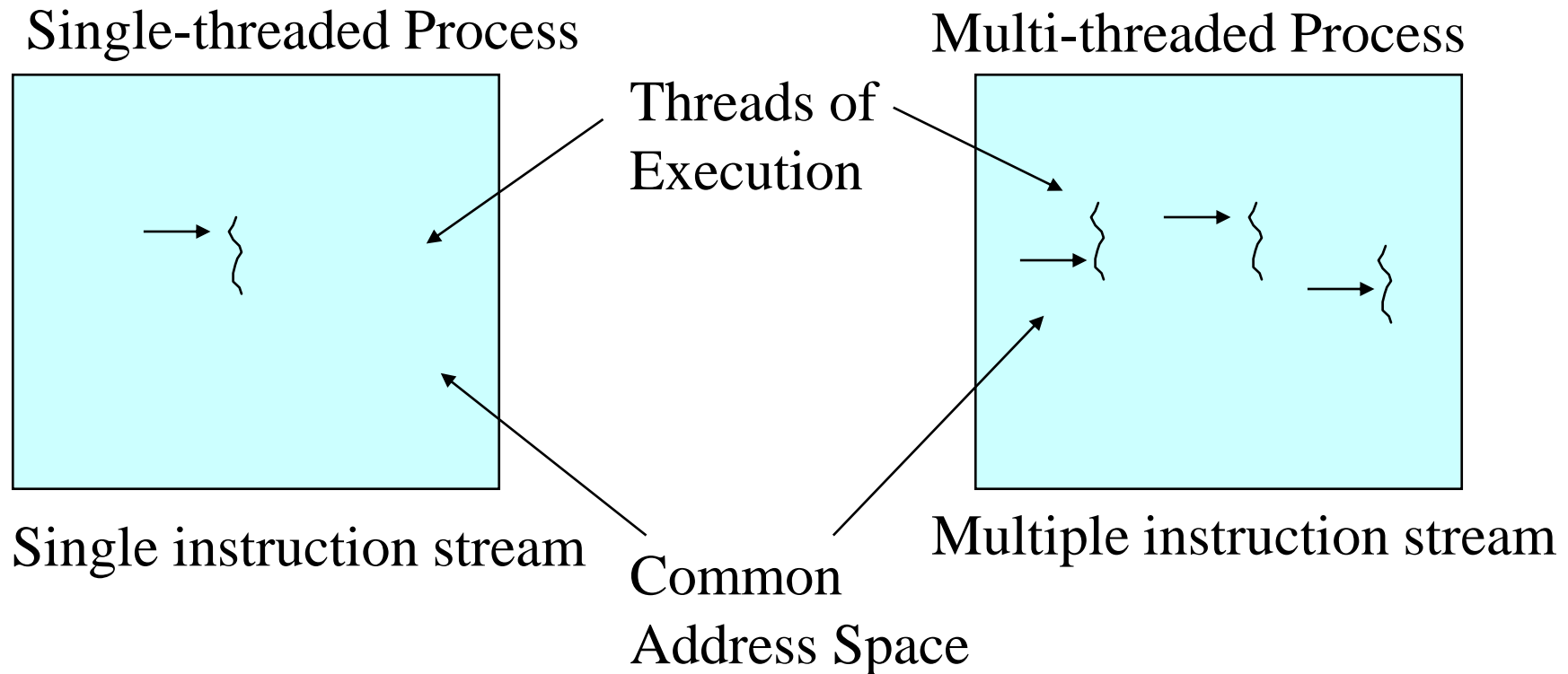
# A Multithreaded Program



Threads may switch or exchange data/results

# Single and Multithreaded Processes

threads are light-weight processes within a process



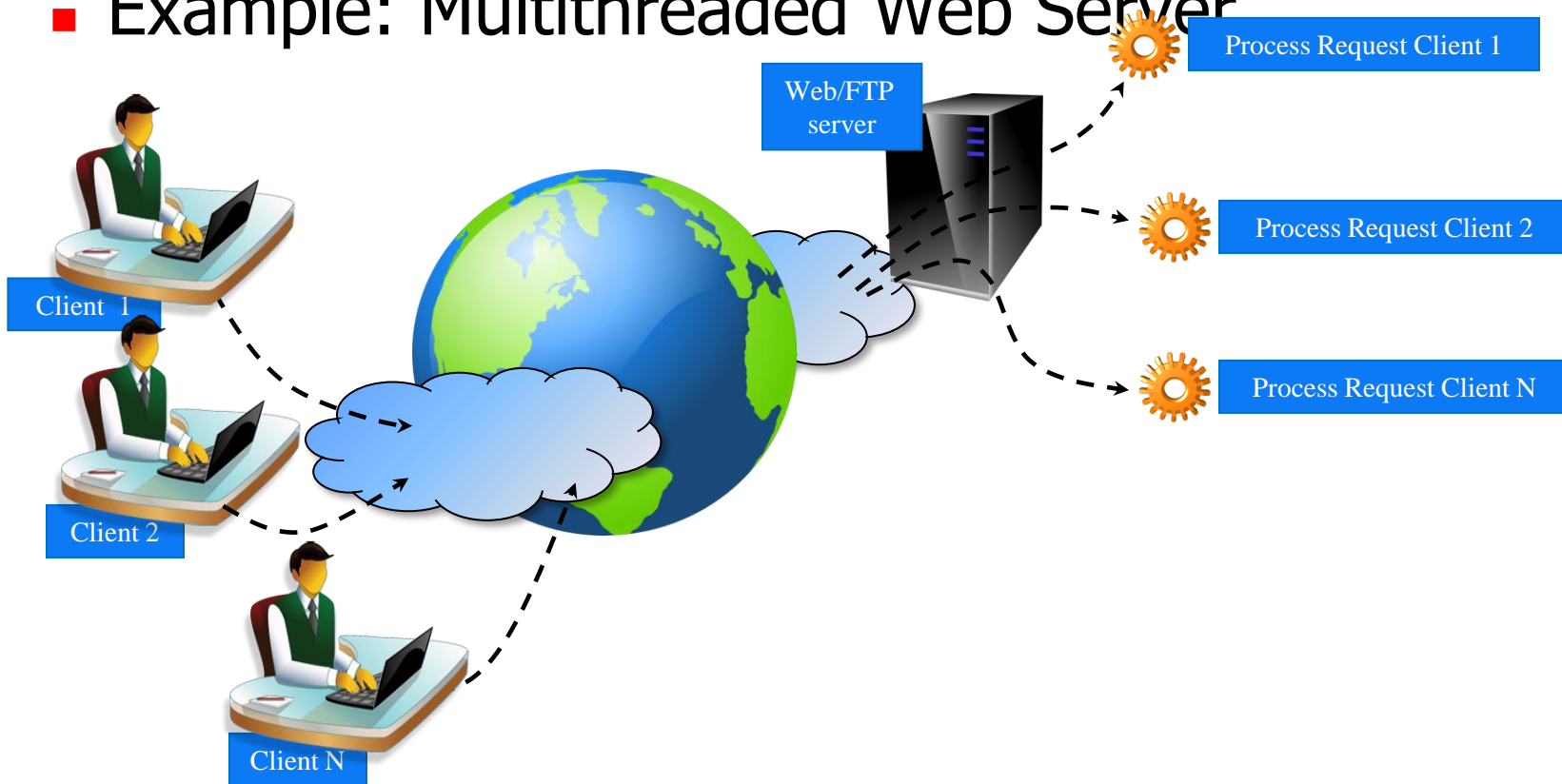
- A process?
- A thread?
- Process-based Multi-tasking vs. thread-based multitasking?



# Multithreaded Server: For Serving Multiple Clients Concurrently

- Modern Applications

- Example: Multithreaded Web Server



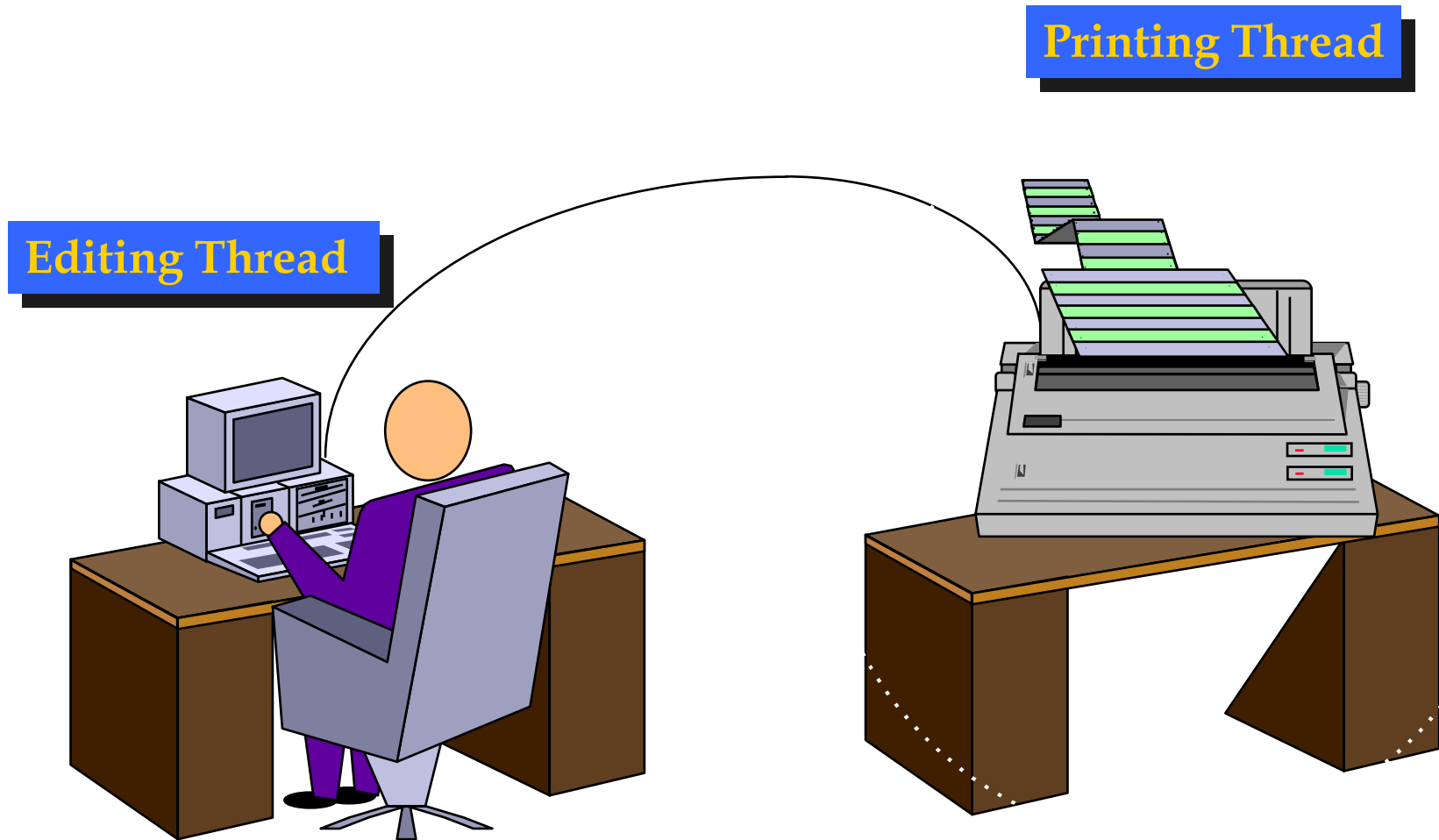
# Threaded Applications

- Modern Applications
  - Example: Internet Browser + Youtube

The image shows a screenshot of the YouTube website interface. Two red rounded rectangles with arrows point to specific features:

- Video Streaming:** A red rounded rectangle highlights the video player area, which shows a video titled "APC's 'V' Unfoot Trailer - with Elizabeth Mitchell - 2nd November 2009". The video is playing at 1:10 / 2:50. An arrow points from this rectangle to a red box labeled "Video Streaming".
- Favorites, Share, Comments Posting:** A red rounded rectangle highlights the bottom section of the page, which includes the "Statistics & Data" section, "Video Responses (20)", and a list of comments. An arrow points from this rectangle to a red box labeled "Favorites, Share, Comments Posting".

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



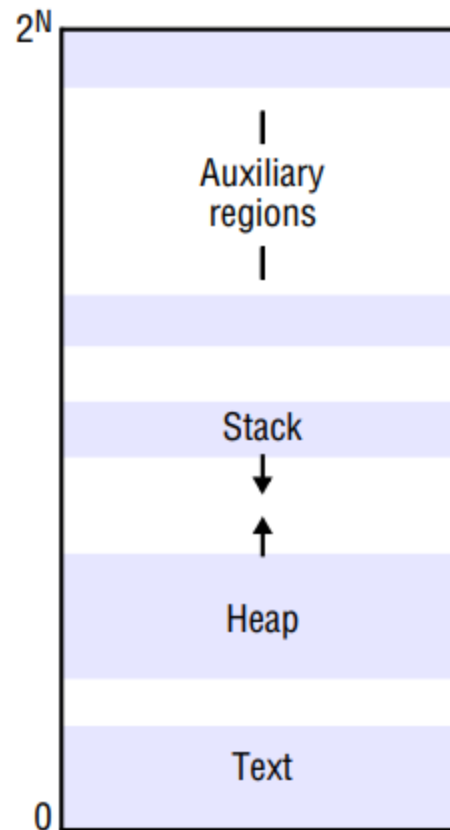
# A Process

- A process consists of an execution environment together with one or more threads
- An execution environment includes:
  - An address space.
  - Thread synchronization and communication resources (e.g., semaphores, sockets).
  - Higher-level resources such as open files and windows.

# Address Space?

- An address space is a unit of management of a process's virtual memory.
- It consists of one or more regions, separated by inaccessible areas of virtual memory.
- A region is an area of contiguous virtual memory that is accessible by the threads of the owning process.
- An execution environment provides protection from threads outside it, so that the data and other resources contained in it are by default inaccessible to threads residing in other execution environment

# Address Space



# Process vs. Thread

- Communication
- Context switching
- Security

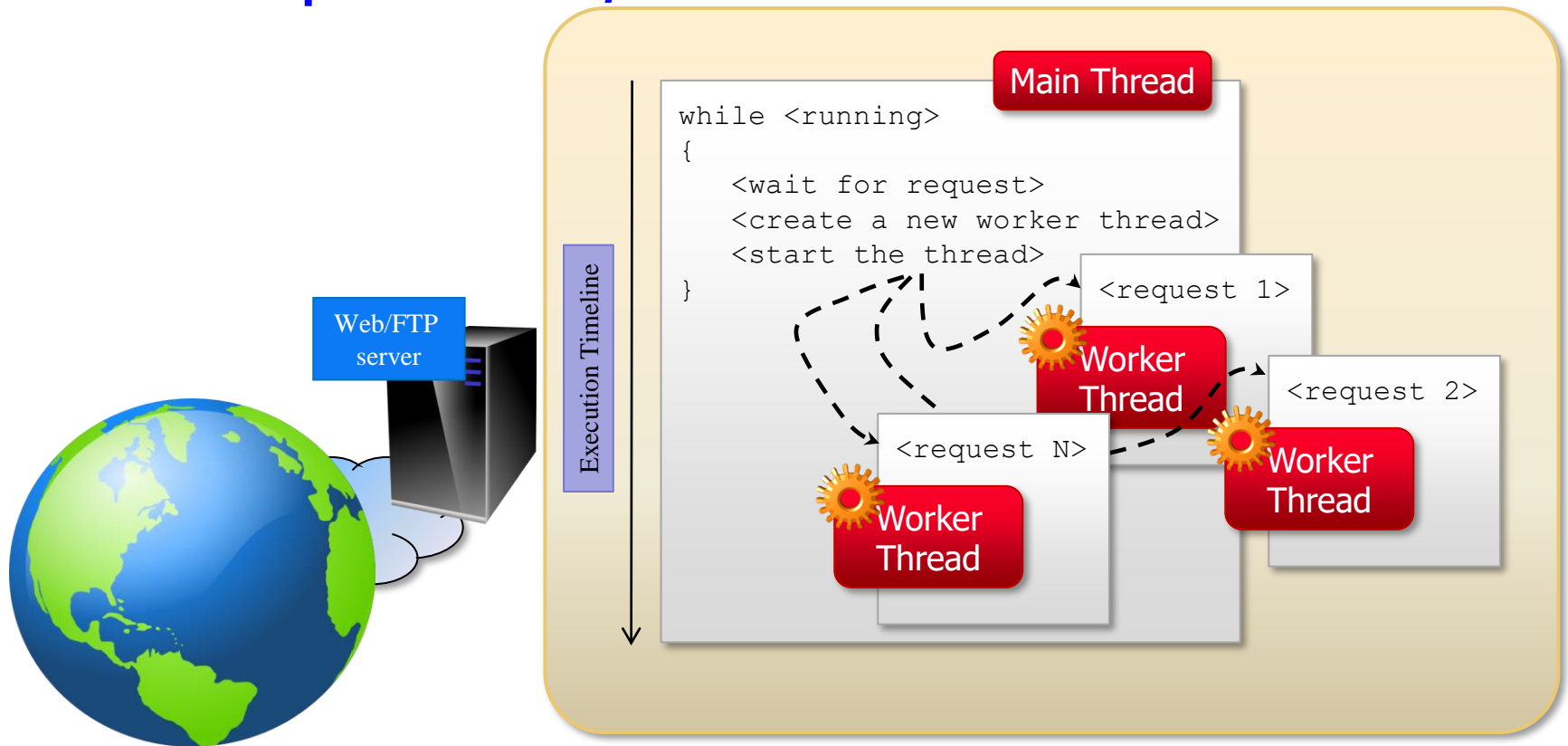
# Defining Threads

- Applications – Threads are used to perform:
  - Parallelism and concurrent execution of independent tasks / operations.
  - Non blocking I/O operations.
  - Asynchronous behavior.



# Defining Threads

- Example: Web/FTP Server



# Defining Threads

## ■ Summing Up

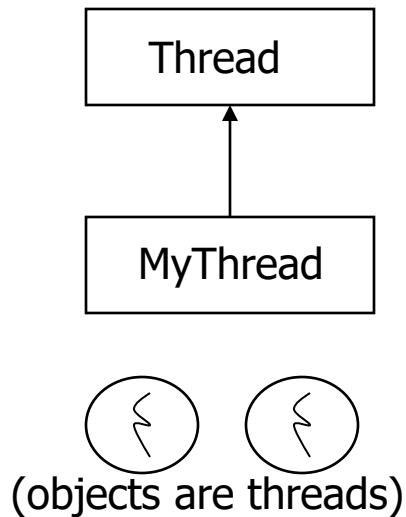
- A Thread is a piece of code that runs in concurrent with other threads.
- Each thread is a statically ordered sequence of instructions.
- Threads are used to express concurrency on both single and multiprocessors machines.
- Programming a task having multiple threads of control – Multithreading or Multithreaded Programming.

# Java Threads

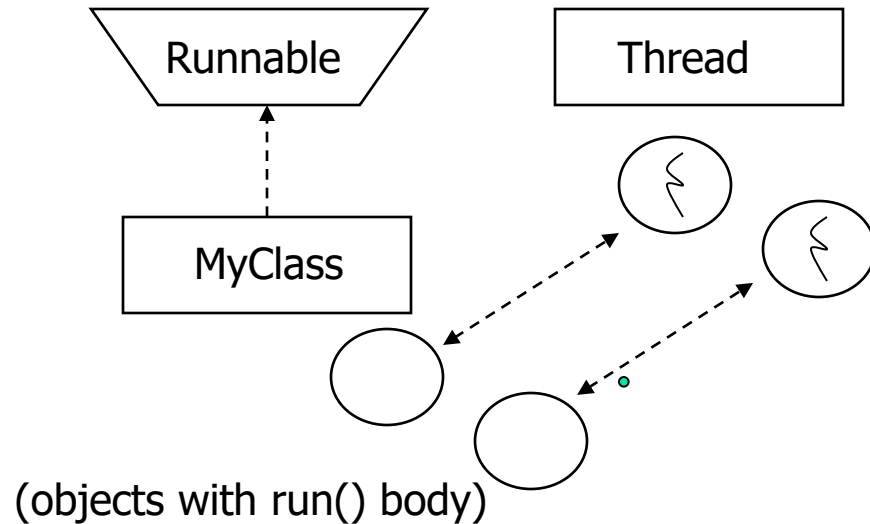
- Java has built in support for Multithreading
- Synchronization
- Thread Scheduling
- Inter-Thread Communication:
  - |               |       |             |
|---------------|-------|-------------|
| currentThread | start | setPriority |
| yield         | run   | getPriority |
| sleep         | stop  | suspend     |
| resume        |       |             |
- Java Garbage Collector is a low-priority thread.

# Threading Mechanisms...

- Create a class that extends the Thread class
- Create a class that implements the Runnable interface



[a]



[b]

Which is better?

# 1st method: Extending Thread class

- Create a class by extending Thread class and override run() method:

```
class MyThread extends Thread
{
    public void run()
    {
        // thread body of execution
    }
}
```

- Create a thread:

```
MyThread thr1 = new MyThread();
```

- Start Execution of threads:

```
thr1.start();
```

- Create and Execute:

```
new MyThread().start();
```

# An example

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println(" this thread is running ... ");  
    }  
}
```

```
class ThreadEx1 {  
    public static void main(String [] args ) {  
        MyThread t = new MyThread();  
        t.start();  
    }  
}
```

## 2nd method: Threads by implementing Runnable interface

- Create a class that implements the interface Runnable and override run() method:

```
class MyThread implements Runnable
{
    .....
    public void run()
    {
        // thread body of execution
    }
}
```

- Creating Object:

```
MyThread myObject = new MyThread();
```

- Creating Thread Object:

```
Thread thr1 = new Thread( myObject );
```

- Start Execution:

```
thr1.start();
```

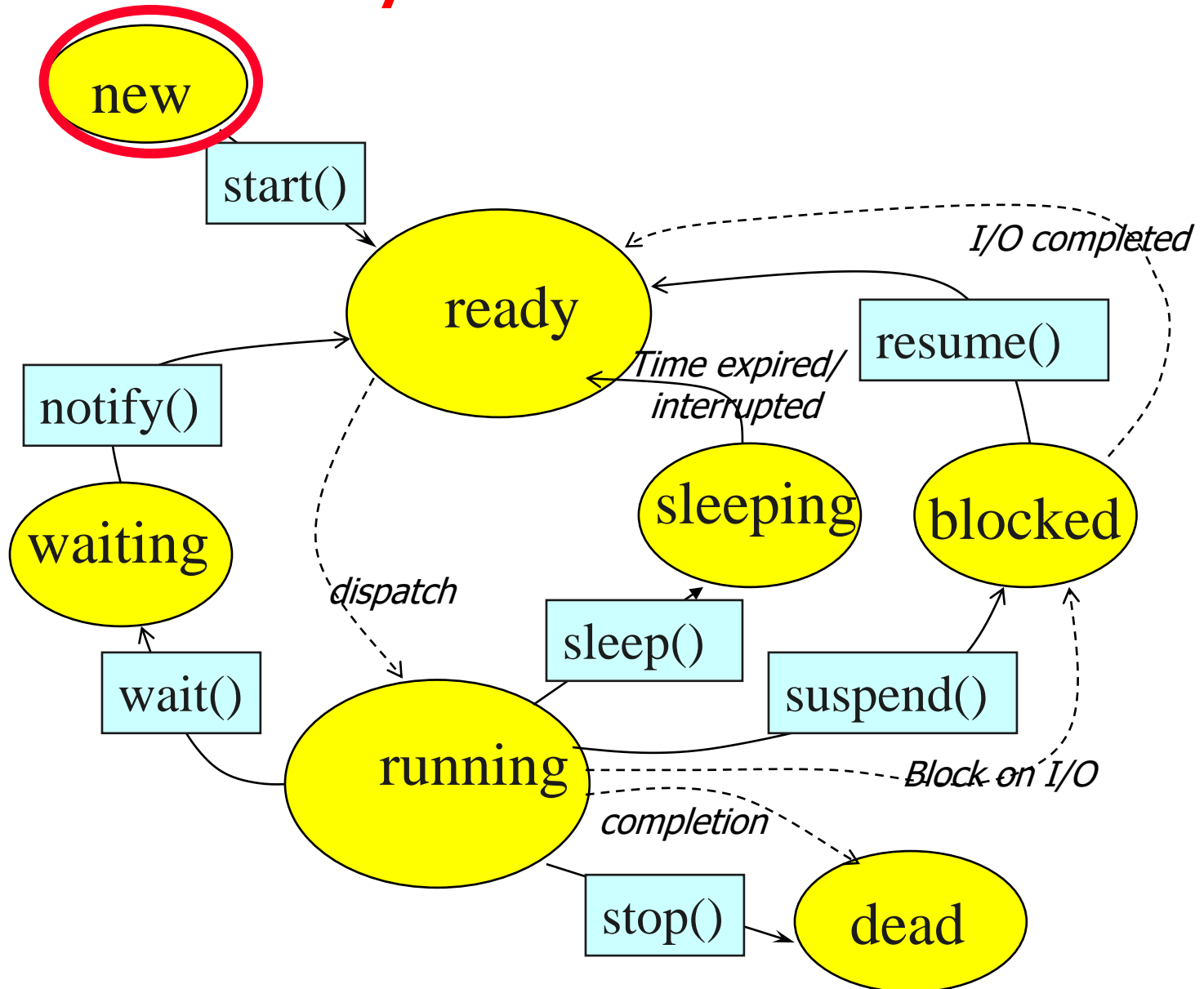
# An example

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

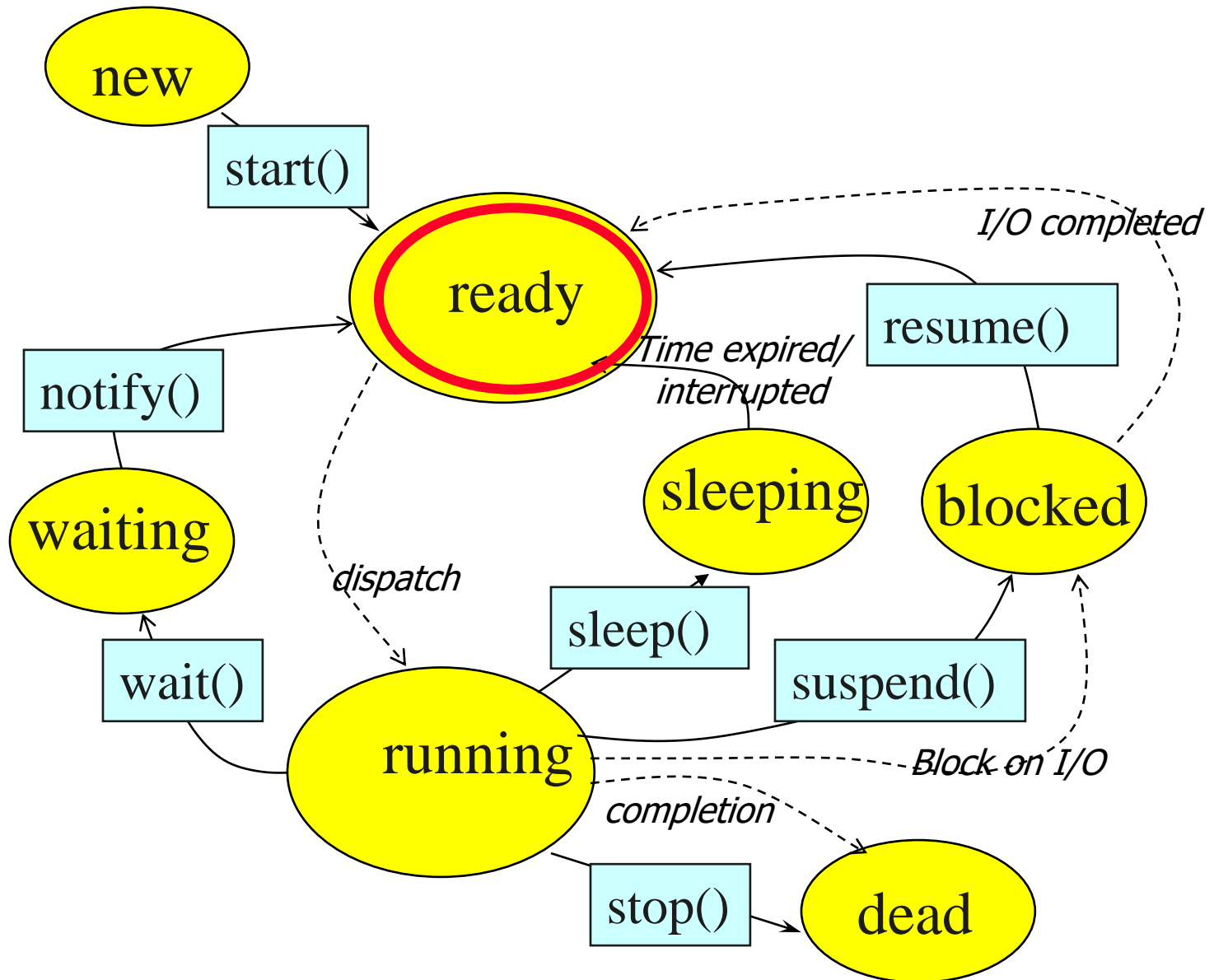
```
class ThreadEx2 {  
    public static void main(String [] args ) {  
        Thread t = new Thread(new MyThread());  
        t.start();  
    }  
}
```



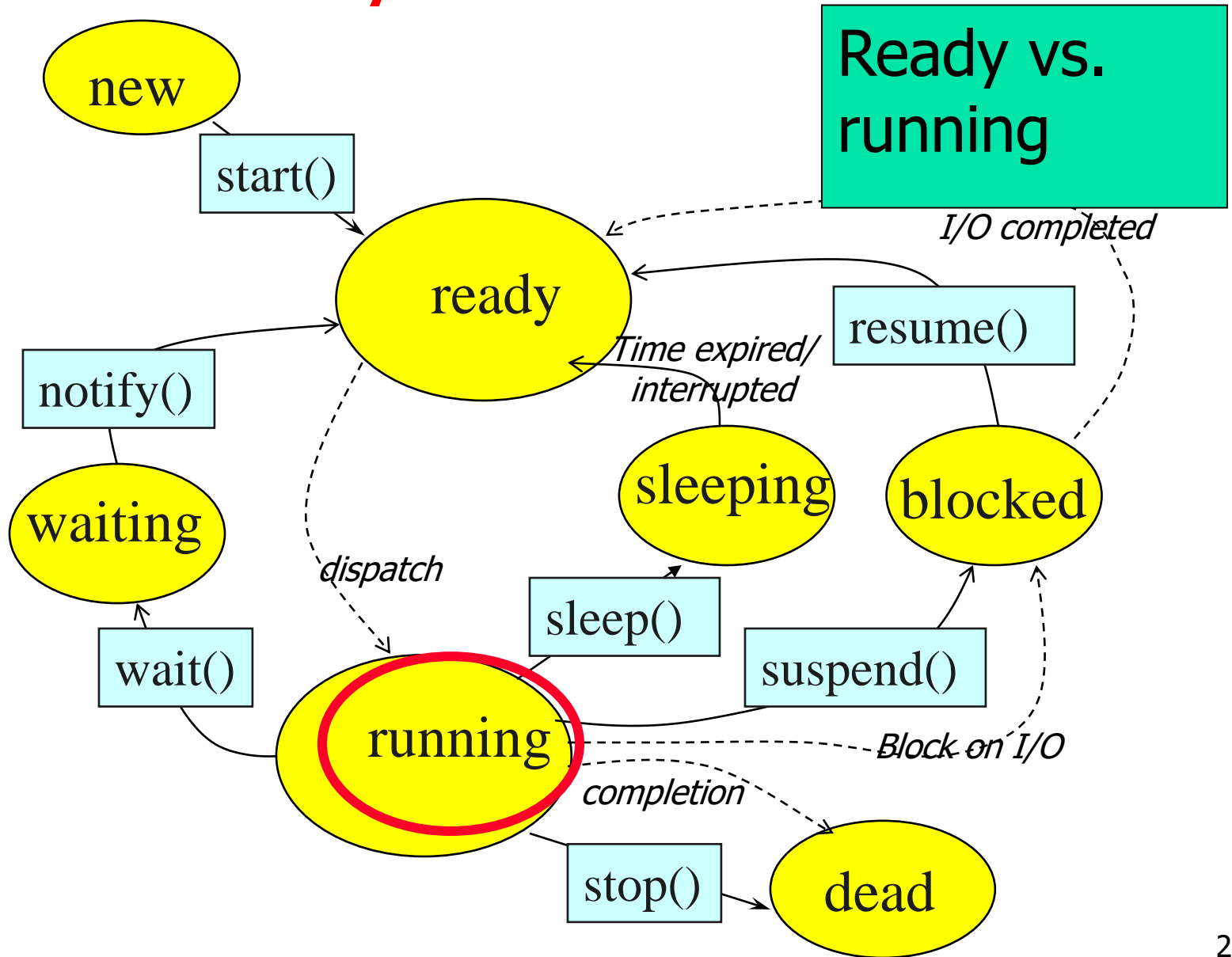
# Life Cycle of Thread



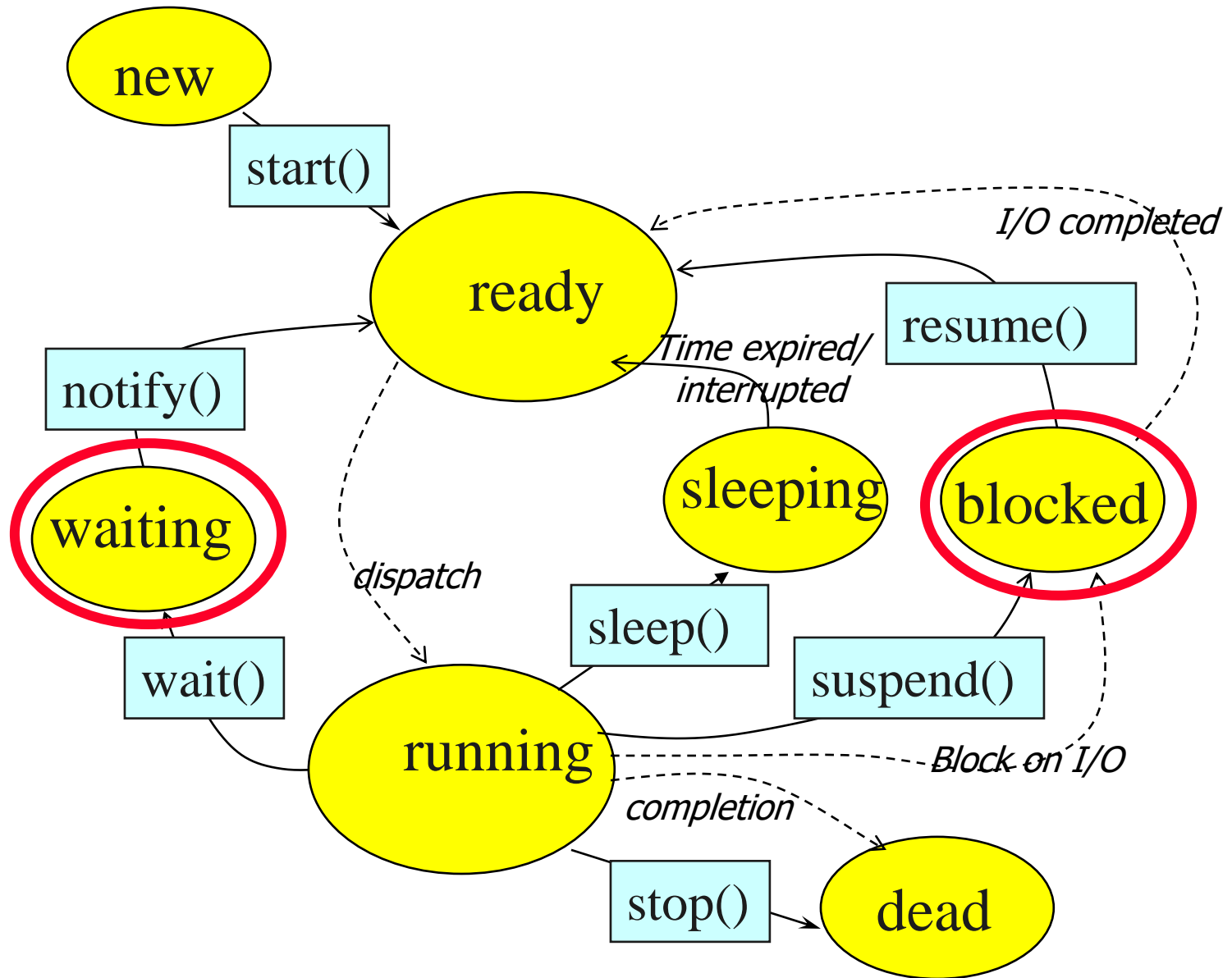
# Life Cycle of Thread



# Life Cycle of Thread



# Life Cycle of Thread



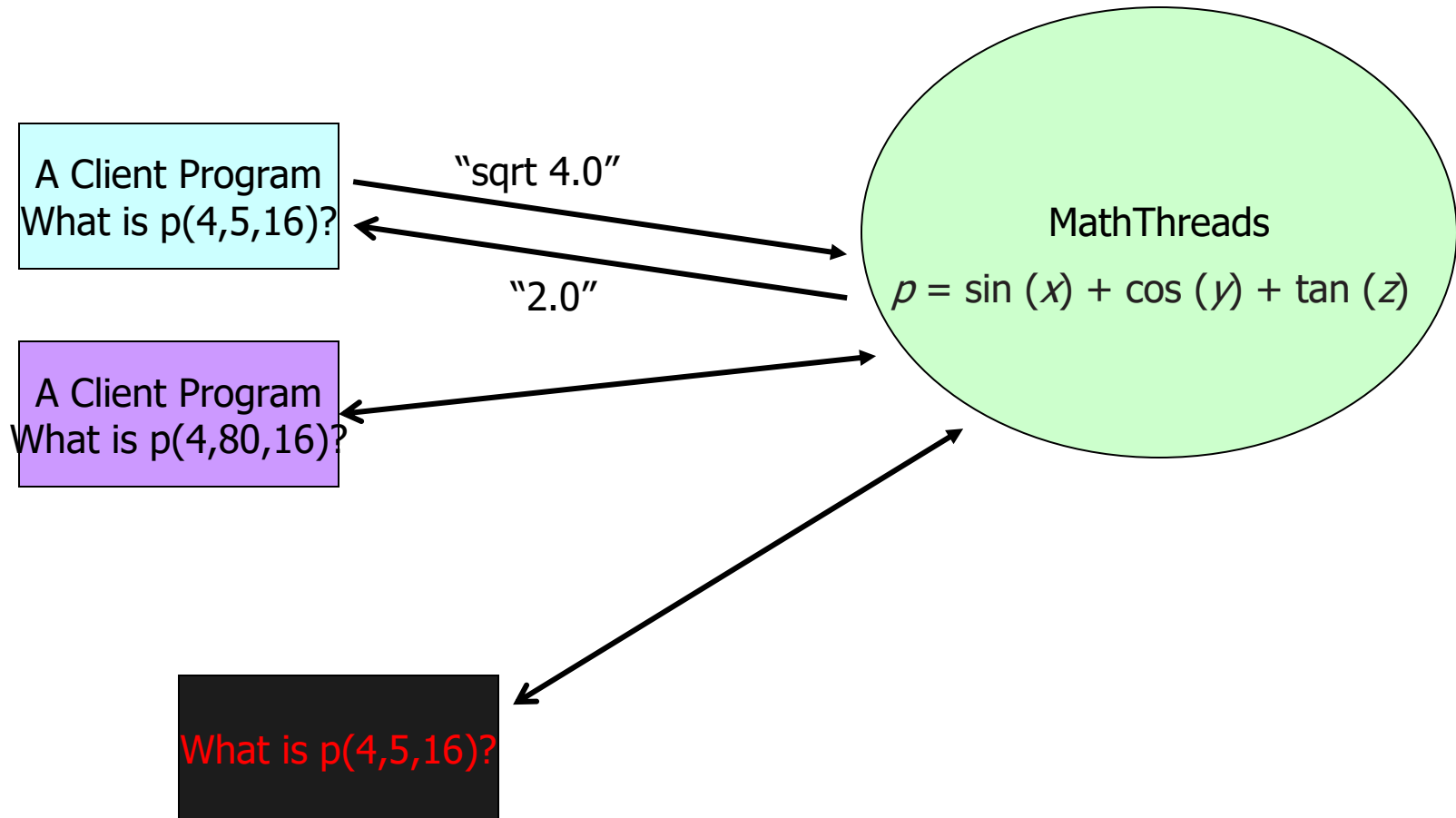
# Thread Operations

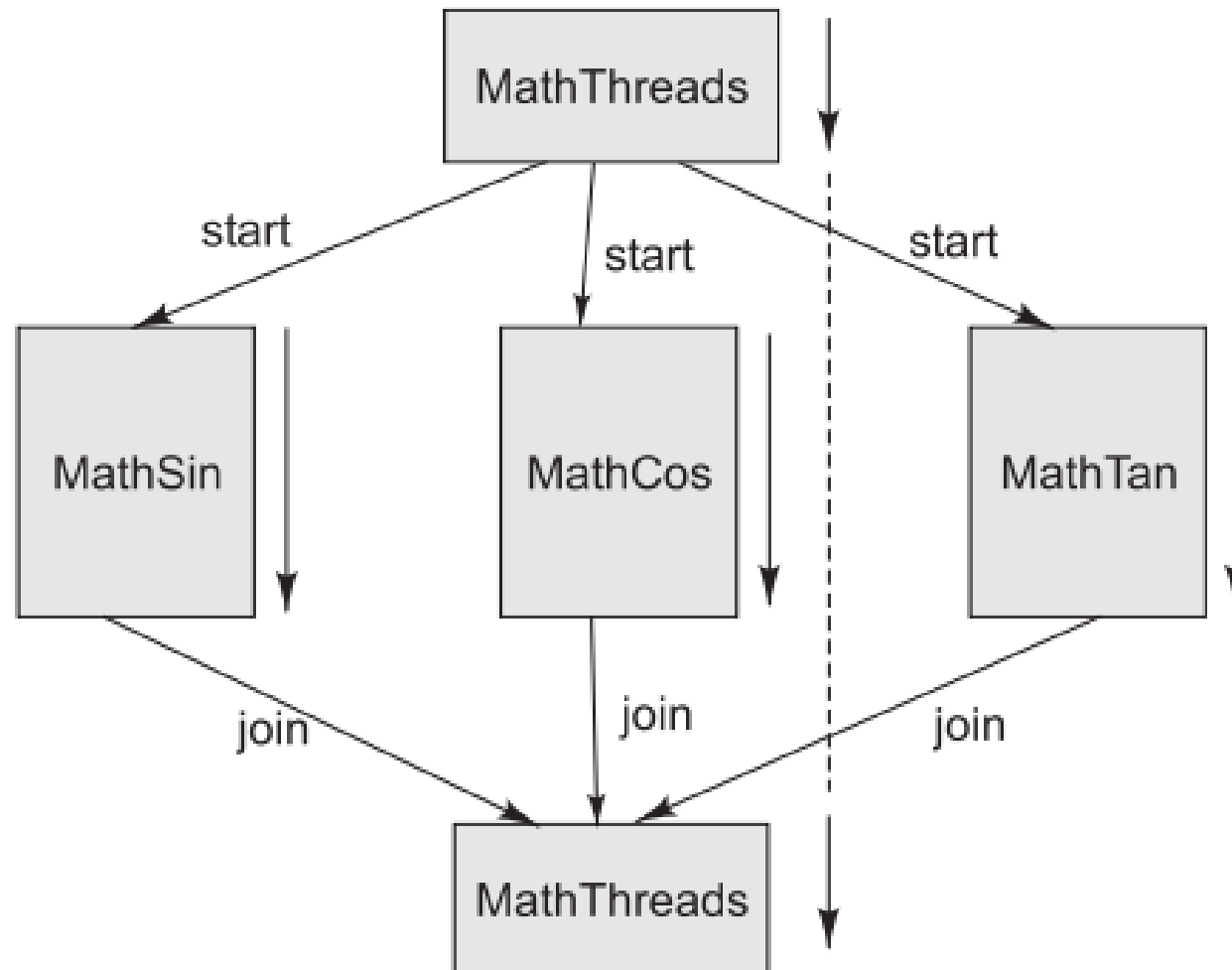
- `public static void sleep(long millis)`  
throws `InterruptedException`
- For example, the code below puts the thread in sleep state for 3 minutes
- ```
try {  
    Thread.sleep(3 * 60 * 1000); // thread  
    sleeps for 3 minutes  
} catch(InterruptedException ex){}
```

# Thread Operations

- `public static void yield()`
- `public final boolean isAlive()`
- `void join()`  
`void join(long millis)`  
`void join(long millis, int nanos)`

# Example 1: MathServer – Demonstrates the use of Threads







```
/* MathThreads.java: A program with multiple threads performing concurrent
operations. */
import java.lang.Math;
class MathSin extends Thread {
    public double deg;
    public double res;

    public MathSin(int degree) {
        deg = degree;
    }
    public void run() {
        System.out.println("Executing sin of "+deg);
        double Deg2Rad = Math.toRadians(deg);
        res = Math.sin(Deg2Rad);
        System.out.println("Exit from MathSin. Res = "+res);
    }
}
class MathCos extends Thread {
    public double deg;
    public double res;
```

```

    public MathCos(int degree) {
        deg = degree;
    }
    public void run() {
        System.out.println("Executing cos of "+deg);
        double Deg2Rad = Math.toRadians(deg);
        res = Math.cos(Deg2Rad);
        System.out.println("Exit from MathCos. Res = "+res);
    }
}
class MathTan extends Thread {
    public double deg;
    public double res;

    public MathTan(int degree) {
        deg = degree;
    }
    public void run() {
        System.out.println("Executing tan of "+deg);
        double Deg2Rad = Math.toRadians(deg);
        res = Math.tan(Deg2Rad);
        System.out.println("Exit from MathTan. Res = "+res);
    }
}

```

```

class MathThreads {
    public static void main(String args[]) {
        MathSin st = new MathSin(45);
        MathCos ct = new MathCos(60);
        MathTan tt = new MathTan(30);
        st.start();
        ct.start();
        tt.start();
        try {    // wait for completion of all thread and then sum
            st.join();
            ct.join(); //wait for completion of MathCos object
            tt.join();
            double z = st.res + ct.res + tt.res;
            System.out.println("Sum of sin, cos, tan = "+z);
        }
        catch(InterruptedException IntExp) {
        }
    }
}

```

**Run 1:**

```

[raj@mundroo] threads [1:111] java MathThreads
Executing sin of 45.0
Executing cos of 60.0
Executing tan of 30.0

```

# A Program with Three Java Threads

- Write a program that creates 3 threads

# Three threads example

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

■ class B extends Thread
■ {
■     public void run()
■     {
■         for(int j=1;j<=5;j++)
■         {
■             System.out.println("\t From ThreadB: j= "+j);
■         }
■         System.out.println("Exit from B");
■     }
■ }
```

# Three threads example

```
■ class C extends Thread
■ {
■     public void run()
■     {
■         for(int k=1;k<=5;k++)
■         {
■             System.out.println("\t From ThreadC: 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

- [raj@mundroo] 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

# Run 2

- [raj@mundroo] 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



# 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");
    }
}
```

# Thread Priority Example

```
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

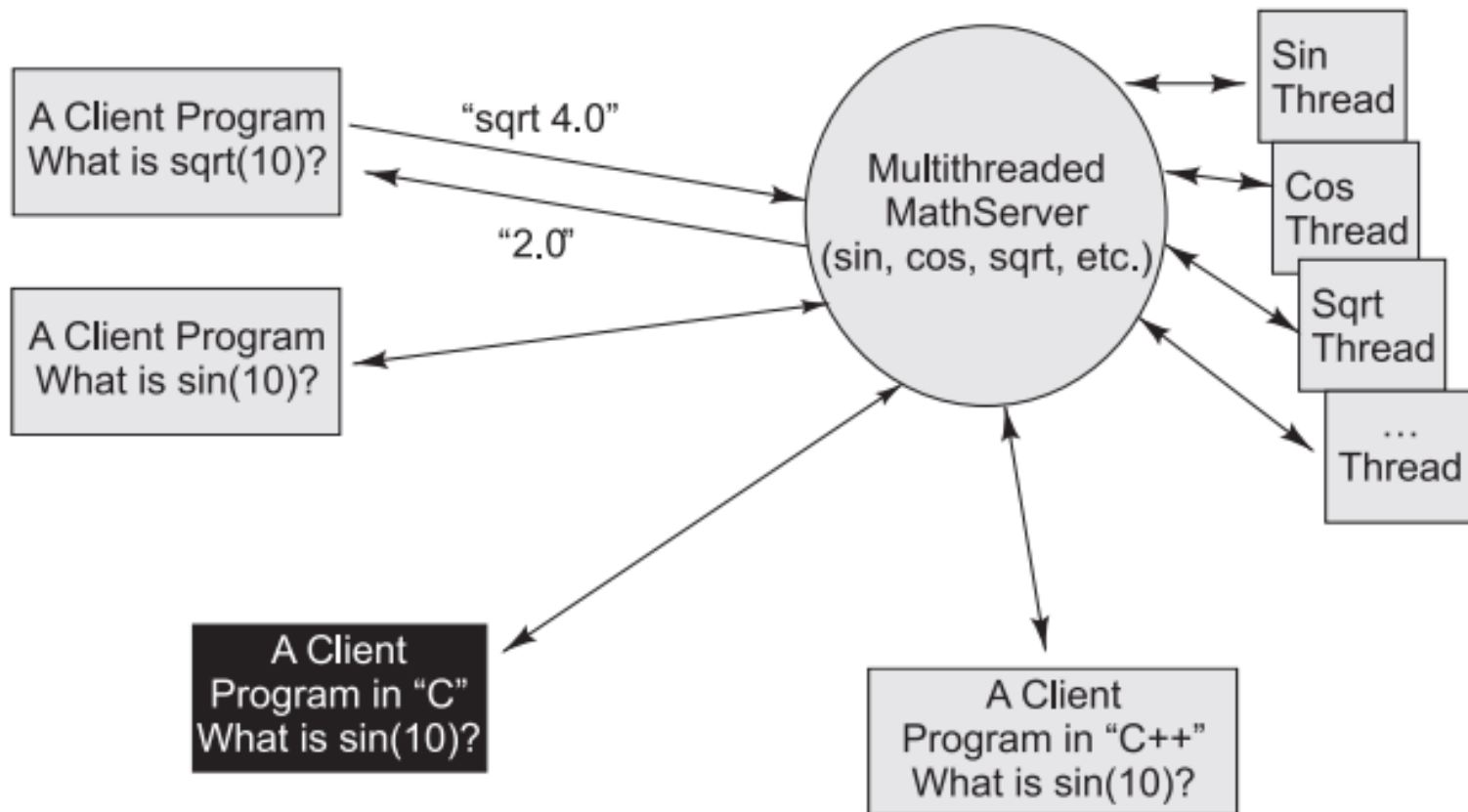
How does  
the order go?

```
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

- Started Thread A  
Started Thread B  
Started Thread C  
Thread B started  
End of main thread  
Thread C started  
From ThreadC: k= 1  
From ThreadC: k= 2  
From ThreadC: k= 3  
From ThreadC: k= 4  
Exit from C  
From ThreadB: j= 1  
From ThreadB: j= 2  
From ThreadB: j= 3  
From ThreadB: j= 4  
Exit from B  
Thread A started  
From ThreadA: i= 1  
From ThreadA: i= 2  
From ThreadA: i= 3  
From ThreadA: i= 4  
  
Exit from A

# Multi-threaded Math Server



# Multi-threaded Math Server

```
/* MultiThreadMathServer.java: A program extending MathServer which  
allows concurrent client requests and opens a new thread for each socket  
connection. */  
import java.net.ServerSocket;  
import java.net.Socket;  
public class MultiThreadMathServer  
    extends MathServer implements Runnable {  
    public void run() {
```



# Multi-threaded Math Server

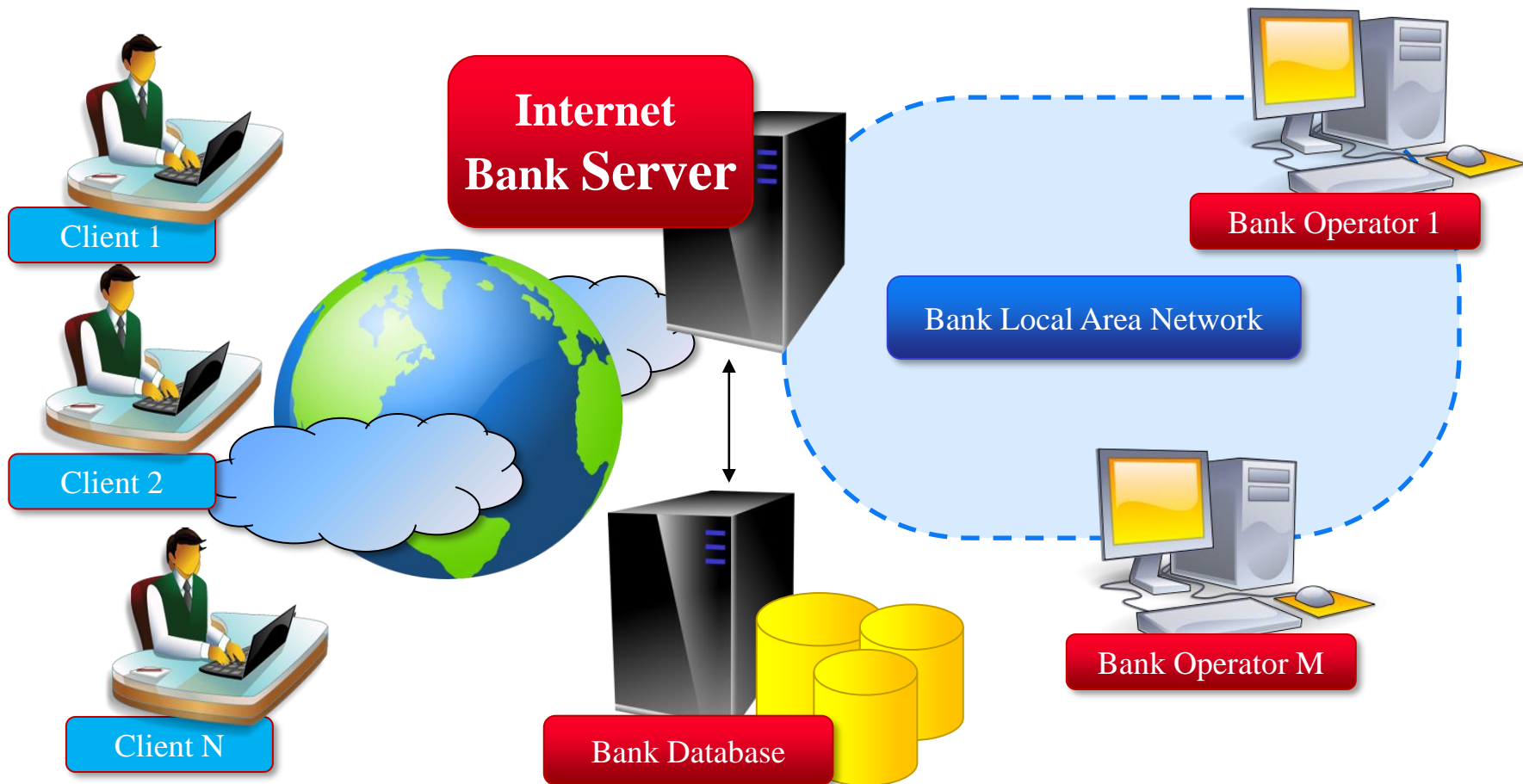
```
        execute();
    }
    public static void main(String [] args) throws Exception {
        int port = 10000;
        if (args.length == 1) {
            try {
                port = Integer.parseInt(args[0]);
            }
            catch(Exception e) {
            }
        }
        ServerSocket serverSocket = new ServerSocket(port);
        while(true){
            //waiting for client connection
            Socket socket = serverSocket.accept();
            socket.setSoTimeout(14000);
            MultiThreadMathServer server = new MultiThreadMathServer();
            server.setMathService(new PlainMathService());
            server.setSocket(socket);
            //start a new server thread...
            new Thread(server).start();
        }
    }
}
```

How about  
the client code?

# Accessing Shared Resources

- Applications access to shared resources need to be coordinated.
  - Printer (two person jobs cannot be printed at the same time)
  - Simultaneous operations on your bank account.
  - Can the following operations be done at the same time on the same account?
    - Deposit()
    - Withdraw()
    - Enquire()

# Online Bank: Serving Many Customers and Operations



# Shared Resources

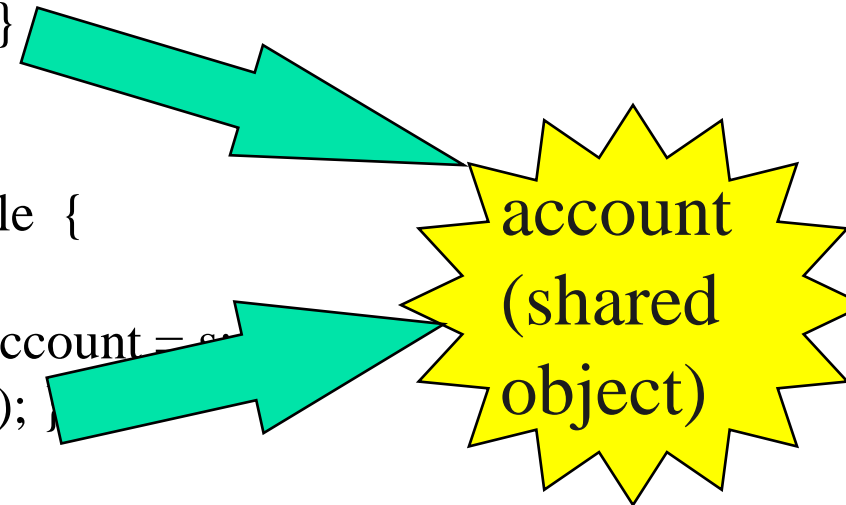


- If one thread tries to read the data and other thread tries to update the same data, it leads to inconsistent state.
- This can be prevented by synchronising access to the data.
- Use “synchronized” method:
  - `public synchronized void update()`
  - `{`
  - `...`
  - `}`

# Shared account object between 3 threads

```
class DepositThread implements Runnable {  
    Account account;  
    public DepositThread (Account s) { account = s;}  
    public void run() { account.deposit(); }  
} // end class MyThread
```

```
class WithdrawThread implements Runnable {  
    Account account;  
    public WithdrawThread (Account s) { account = s;}  
    public void run() { account.withdraw(); }  
} // end class YourThread
```



account  
(shared  
object)

The diagram illustrates the concept of a shared object. Two teal arrows originate from the `account` field in the `run()` methods of `DepositThread` and `WithdrawThread`. Both arrows point towards a yellow, star-shaped callout box on the right. This box contains the text "account (shared object)", indicating that both threads are interacting with the same instance of the `Account` class.

# The driver: 6 Threads sharing the same object

```
/* InternetBankingSystem.java: A simple program showing a typical invocation of  
banking operations via multiple threads. */  
public class InternetBankingSystem {  
    public static void main(String [] args ) {  
        Account accountObject = new Account(100);  
        new Thread(new DepositThread(accountObject,30)).start();  
        new Thread(new DepositThread(accountObject,20)).start();  
        new Thread(new DepositThread(accountObject,10)).start();  
        new Thread(new WithdrawThread(accountObject,30)).start();  
        new Thread(new WithdrawThread(accountObject,50)).start();  
        new Thread(new WithdrawThread(accountObject,20)).start();  
    } // end main()  
}
```

# The driver: 6 Threads sharing the same object

```
/* WithdrawThread.java A thread that withdraw a given amount from a given
account. */
package com.javabook.threading;
public class WithdrawThread implements Runnable {
    private Account account;
    private double amount;

    public WithdrawThread(Account account, double amount) {
        this.account = account;
        this.amount = amount;
    }

    public void run() {
        //make a withdraw
        account.withdraw(amount);
    }
} //end WithdrawThread class
/* DepositThread.java A thread that deposit a given amount to a given
account. */
```

# The driver: 6 Threads sharing the same object

```
package com.javabook.threading;

public class DepositThread implements Runnable {
    private Account account;
    private double amount;

    public DepositThread(Account account, double amount) {
        this.account = account;
        this.amount = amount;
    }

    public void run() {
        //make a deposit
        account.deposit(amount);
    }
} //end DepositThread class
```



## 6 Threads sharing the same object: Unpredictable Behavior

```
Withdraw 30.0 in thread 11, balance is 60.0  
Deposit 20.0 in thread 9, balance is 60.0  
Withdraw 50.0 in thread 12, balance is 60.0  
Deposit 10.0 in thread 10, balance is 60.0  
Withdraw 20.0 in thread 13, balance is 60.0  
Deposit 30.0 in thread 8, balance is 60.0
```

# Monitor (shared object access): serializes operation on shared objects

```
class Account { // the 'monitor'
    int balance;

    // if 'synchronized' is removed, the outcome is unpredictable
    public synchronized void deposit( ) {
        // METHOD BODY : balance += deposit_amount;
    }

    public synchronized void withdraw( ) {
        // METHOD BODY: balance -= deposit_amount;
    }
    public synchronized void enquire( ) {
        // METHOD BODY: display balance.
    }
}
```

## 6 Threads sharing the same object: Predictable Behavior

```
Deposit 30.0 in thread 8,new balance is 130.0  
Withdraw 50.0 in thread 12,new balance is 80.0  
Deposit 10.0 in thread 10,new balance is 90.0  
Withdraw 20.0 in thread 13,new balance is 70.0  
Withdraw 30.0 in thread 11,new balance is 40.0  
Deposit 20.0 in thread 9,new balance is 60.0
```

# Producer and Consumer Problem

- Two threads, the producer and the consumer, share a common fixed-length buffer
- Producers generate a piece of data and put it into the buffer.
- The consumer is consuming data from the same buffer simultaneously
- Problem?
- Solution?

```
/* MessageQueue.java: A message queue with synchronized methods for queuing
and consuming messages. */

package com.javabook.threading;
import java.util.ArrayList;
import java.util.List;

public class MessageQueue {
    //the size of the buffer
    private int bufferSize;

    //the buffer list of the message, assuming the string message format
    private List<String> buffer = new ArrayList<String>();

    //construct the message queue with given buffer size
    public MessageQueue(int bufferSize){
        if(bufferSize<=0)
            throw new IllegalArgumentException("Size is illegal.");
        this.bufferSize = bufferSize;
    }
    //check whether the buffer is full
    public synchronized boolean isFull() {
        return buffer.size() == bufferSize;
    }
}
```

```

}

//check whether the buffer is empty
public synchronized boolean isEmpty() {
    return buffer.isEmpty();
}

//put an income message into the queue, called by message producer
public synchronized void put(String message) {
    //wait until the queue is not full
    while (isFull()) {
        System.out.println("Queue is full.");
        try{
            //set the current thread to wait
            wait();
        }catch(InterruptedException ex){
            //someone wake me up.
        }
    }
    buffer.add(message);
    System.out.println("Queue receives message '"+message+"'");

    //wakeup all the waiting threads to proceed
    notifyAll();
}

```

```

//get a message from the queue, called by the message consumer
public synchronized String get(){
    String message = null;
    //wait until the queue is not empty
    while(isEmpty()){
        System.out.println("There is no message in queue.");
        try{
            //set the current thread to wait
            wait();
        }catch(InterruptedException ex){
            //someone wake me up.
        }
    }
    //consume the first message in the queue
    message = buffer.remove(0);

    //wakeup all the waiting thread to proceed
    notifyAll();
    return message;
}
} //end MessageQueue class

```

# The Producer

```
/* Producer.java: A producer that generates messages and put into a given message queue. */
```

```
package com.javabook.threading;
```

```
public class Producer extends Thread{
    private static int count = 0;
    private MessageQueue queue = null;

    public Producer(MessageQueue queue){
        this.queue = queue;
    }

    public void run(){
        for(int i=0;i<10;i++){
            queue.put(generateMessage());
        }
    }

    private synchronized String generateMessage(){
        String msg = "MSG#" + count;
        count ++;

        return msg;
    }
} //end Producer class
```



# The Consumer

```
/* Consumer.java: A consumer that consumes messages from the queue. */  
package com.javabook.threading;  
  
public class Consumer extends Thread {  
    private MessageQueue queue = null;  
  
    public Consumer(MessageQueue queue){  
        this.queue = queue;  
    }  
  
    public void run(){  
        for(int i=0;i<10;i++){  
            System.out.println("Consumer downloads "  
                +queue.get()+ " from the queue.");  
        }  
    }  
} //end Consumer class
```

# Main Program

```
/* MessageSystem.java: A message system that demonstrate the produce and
consumer problem with the message queue example. */

package com.javabook.threading;

public class MessageSystem {
    public static void main(String[] args) {
        MessageQueue queue = new MessageQueue(5);
        new Producer(queue).start();
        new Producer(queue).start();
        new Producer(queue).start();
        new Consumer(queue).start();
        new Consumer(queue).start();
        new Consumer(queue).start();
    }
} //end MessageSystem class
```

# Required Readings

- CDK Book (Text Book)
  - Chapter 7 – “Operating System Support”
- Chapter 14: Multithread Programming
  - R. Buyya, S. Selvi, X. Chu, **“Object Oriented Programming with Java: Essentials and Applications”**, McGraw Hill, New Delhi, India, 2009.