CS205 Object Oriented Programming in Java

Module 4 - **Advanced features of Java**

(Part 9)

Prepared by

Renetha J.B.

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

Topics

- **Multithreaded Programming :**
  - The Java Thread Model
  - The Main Thread
  - Creating Thread

Thread

- Using **thread** we can do multiple activities within a single process.
- E.g In a web browser
  - we can scroll the page while it's downloading an applet or image, play animation and sound concurrently,
  - print a page in the background while you download a new page
- A thread in Java, is the path followed when executing a program.
- All Java **programs** have at least one thread, known as the main thread,
  - which is created by the Java Virtual Machine (JVM) at the program's start, when the main() method is invoked with the main thread.

Multithreaded Programming

- Java provides built-in support for *multithreaded programming*.
- A **multithreaded program** contains **two or more parts** that can run **concurrently(simultaneously)**.
  - Each part of such a program is called a ***thread***,
    - separate memory area is not allocated to threads.
  - Each thread defines a separate path of execution.
- Threads are **lightweight processes** within a process.
- Multithreading is a specialized form of ***multitasking***.
- Multithreading **maximizes the utilization of CPU**

Multitasking

- Multitasking-more than one task run concurrently.
- Two distinct types of multitasking:
  - **process based**
    - *A process-based multitasking* is the feature that allows your computer to **run two or more programs concurrently**
    - E,g. We can execute browser, paint software , calculator , notepad etc at the same time.
  - **thread-based.**
    - The thread is the smallest unit of dispatchable code.
    - A **single program can perform two or more tasks simultaneously.**
    - E.g. text editor can format text at the same time that it is printing, if two actions are being *performed by two separate threads.*

Multithreading

- **ADVANTAGE** Multithreading enables to
  - write **very efficient programs**
  - that make **maximum the use of the CPU**
    - because idle time can be kept to a minimum.
    - This is especially important for the interactive, networked environment in which Java operates, because idle time is common.
  - Threads are independent.
- Applications; Games, animation etc..

Single threaded vs Multithreaded

- In a *single-threaded environment*, one program has to wait for other program to finish —even though the CPU is sitting idle most of the time.
- Multithreading helps to effectively make use of this idle time.

The Java Thread Model

- The Java run-time system **depends on threads** for many things
- All the class libraries are designed with multithreading.
- Java uses threads to enable the entire environment to be **asynchronous**.
  - Asynchronous threading means, a thread once start executing a task, can hold it in mid, save the current state and start executing another task.
- This helps reduce inefficiency by preventing the waste of CPU cycles.

The Java Thread Model(contd.)

- **Single-threaded systems** use an approach called an *event loop with polling*.
  - In this model, *a* single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.
  - In a singled-threaded environment, when a thread *blocks* (*that is, suspends execution)* because it is waiting for some resource, the entire program stops running.
- The benefit of Java's **multithreading** is that the **main loop/polling mechanism is eliminated.**
  - One thread can pause **without stopping** other parts of your program.
  - When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

The Java Thread Model(contd.)

Thread –

Threads exist in several states.

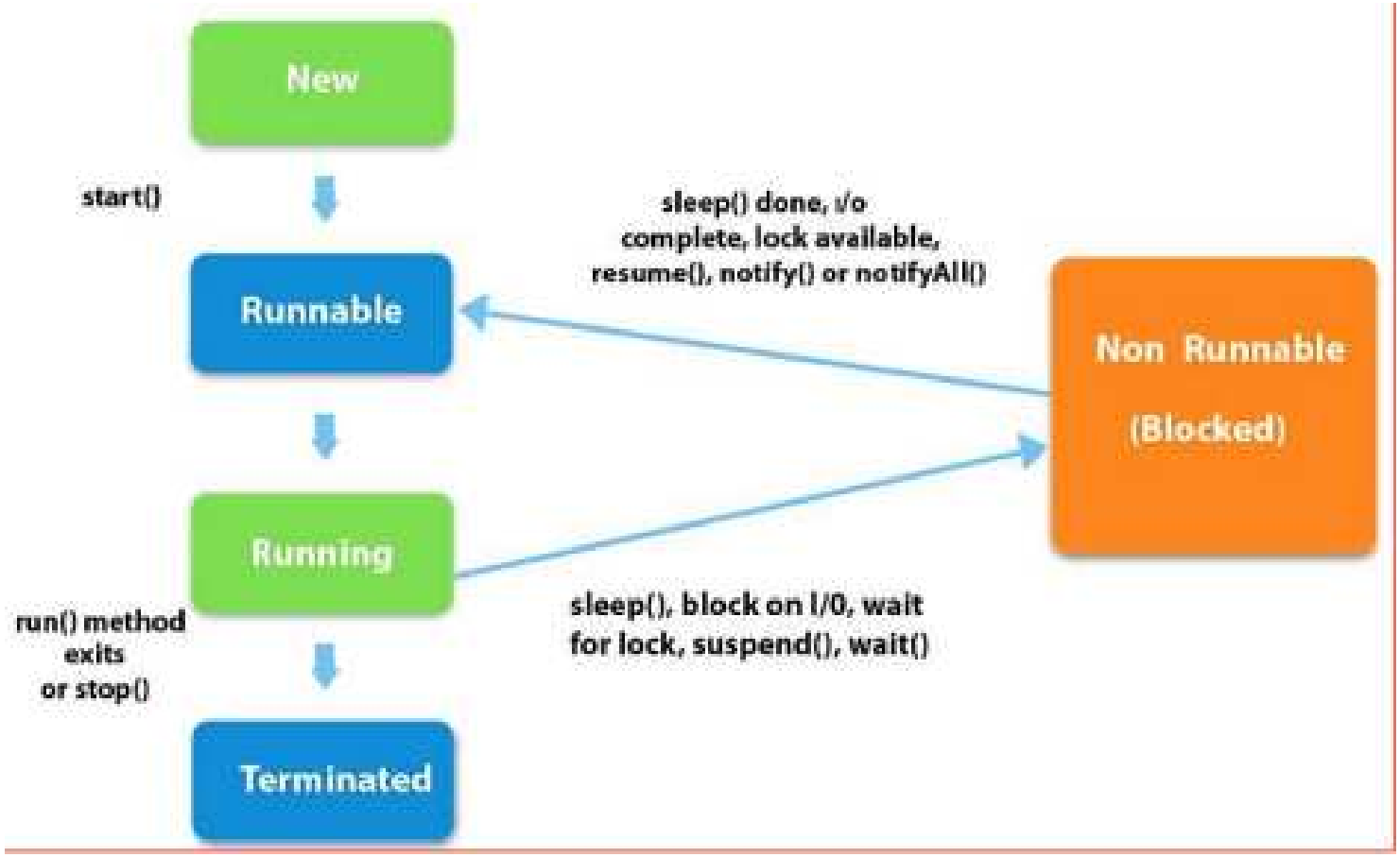
- A thread can be **running**.
- It can be **ready to run(runnable)** as soon as it gets CPU time.
- A running thread can be **suspended(blocked)**, which temporarily suspends its activity.
- A suspended thread can then be **resumed(runnable)**, allowing it to pick up where it left off.
- A thread can be **blocked** when waiting for a resource.
- At any time, a thread can be **terminated**, which halts its execution immediately.
  - Once terminated, a thread cannot be resumed.

The Java Thread Model(contd.)

- Thread States
  - **New**
  - **Runnable –*ready to run***
  - **Running**
  - **Non-Runnable (Blocked)**
  - **Terminated**

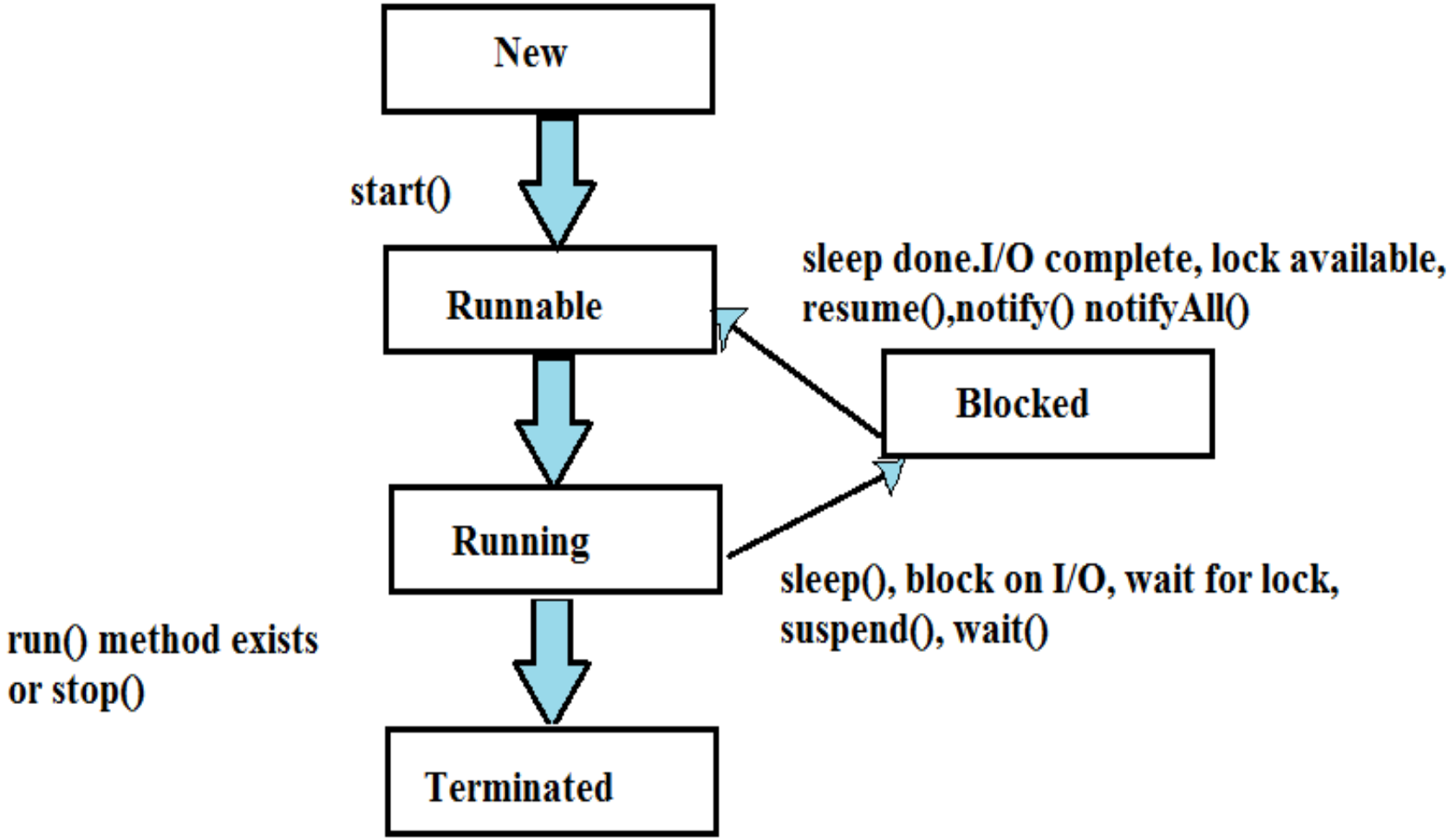
The Java Thread Model(contd.)

- **Thread life cycle**



The Java Thread Model(contd.)

- **Thread life cycle**



The Java Thread Model(contd.)

Thread Priorities

- Java assigns to each thread a priority
  - priority determines how that thread should be treated with respect to the others.
- Thread priorities are **integers**
  - that specify the relative priority of one thread to another
- A higher-priority thread **doesn't run any faster** than a lower-priority thread if it is the only thread running.
- A thread's priority is **used to decide when to switch from one running thread to the next**.
  - Switching from one thread to another is called a **context switch**

The Java Thread Model-Thread priorities(contd.)

- The rules that determine **when a context switch** takes place are:
  - **A thread can voluntarily relinquish control**. This is done by explicitly yielding, sleeping, or blocking on pending I/O. Here all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
  - **A thread can be preempted by a higher-priority thread.** Here a lower-priority thread is simply preempted(forcely suspended) by a higher-priority thread. i.e. As soon as a higher-priority thread wants to run, it can run. This is called **preemptive multitasking**.

The Java Thread Model-Thread priorities(contd.)

- For operating systems such as Windows,
  - threads of **equal priority** are time-sliced automatically in round-robin fashion.
- For other types of operating systems,
  - threads of **equal priority** must voluntarily yield control to their peers.
  - If they don't, the other threads will not run.

Synchronization

- Multithreading introduces an asynchronous behavior.
- But, when two or more threads need **access** to a **shared resource**, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization.*
- Key to synchronization is the concept of the ***monitor*** (also called a *semaphore).*
  - Only one thread can own a monitor at a given time.
- Synchronization in Java can be achieved by
  - **Synchronized Methods**
  - **The synchronized Statement**

Messaging

- Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have.
- Java's **messaging system**
  - allows a thread to enter a synchronized method on an object,
  - and then wait there until
    - some other thread explicitly notifies it to come out.

The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the **Thread** class, its **methods,** its companion interface-**Runnable.**
- Thread encapsulates a thread of execution
- To **create a new thread,** our program will either
  - extend **Thread** ***class*** or
  - implements **Runnable** ***interface*.**

Thread class methods	
getName	• Obtain a thread's name.
getPriority	• Obtain a thread's priority.
isAlive	• Determine if a thread is still running.
join	• Wait for a thread to terminate.
run	• Entry point for the thread.
sleep	• Suspend a thread for a period of time.
start	• Start a thread by calling its run method

The Main Thread

- When a Java program starts up, one thread begins running immediately.
  - This is usually called the **main thread** of our program, because it is executed when our program **begins**.
- The main thread is important for two reasons:
  1. It is the thread from which other "child" threads will be spawned.
  2. Often, it must be the **last thread to finish execution** because it performs various shutdown actions.

The Main Thread(contd.)

- The main thread is created automatically when our program is started.

- The Main thread can be controlled through a **Thread object.**

  – To do so, we must obtain a reference to the thread by calling the method **currentThread( ),** which is a public static member of Thread class.

  – Its general form is:

static Thread currentThread( )

  - This method returns a reference to the thread in which it is called.

  - Once we have a reference to the main thread, we can control it just like any other thread.

The Main Thread(contd.)

```
class CurrentThreadDemo
{
    public static void main(String args[]) {
    Thread t = Thread.currentThread();
    System.out.println("Current thread: " + t);
    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");
}   } }
```

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

Working of the program

- In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread( ),** and this reference is stored in the local variable t.
- Next, the program displays information about the thread.
- The program then calls **setName( )** to change the internal name of the thread. Information about the thread is then redisplayed.
- Next, a loop counts down from five, pausing **one second(1000ms)** between each line.
  - This pausing is accomplished by the **sleep( )** method.
  - The argument to sleep( ) specifies the delay period in milliseconds.

Main thread(contd)

In the output ,Thread[**main**,5,**main**]

- Denotes that by default, the **name of the main thread** is **main.**
- Its **priority** is **5,** which is the default value**,**
- **main** is also the **name of the group of threads** to which this thread belongs.

- A ***thread group*** is a data structure that controls the state of a *collection of threads* as a whole.

Main thread(contd)

- If a thread calls **sleep( )** method then execution of that thread is suspended for the specified period of milliseconds.
- Its general form:

```
static void sleep(long milliseconds) throws InterruptedException
```

- The number of milliseconds to suspend is specified in *milliseconds*. *This method may throw* an **InterruptedException.**
- The **sleep( )** method has a second form,

```
static void sleep(long milliseconds, int nanoseconds) throws
                                                InterruptedException
```

  - This form is useful only in environments that allow timing periods as short as nanoseconds.

Main thread(contd)

- We can set the name of a thread by using **setName( ).**
- We can obtain the name of a thread by calling **getName( )**
- These methods are members of the **Thread class** and are declared as:

final void **setName**(String *threadName)*

final String **getName**( )

  - Here, *threadName specifies the name of the thread.*

Creating a Thread

- Java defines two ways for creating thread:
  - implement the **Runnable interface.**
  - extend the **Thread class.**

Implementing Runnable

- The easiest way to create a thread is to create a class that implements the **Runnable interface.**
- To implement **Runnable, a** class need only implement a single method called **run( ):**

```
public void run( )
```

- Inside **run( ),** we will define the code that constitutes the new thread.
- **run( )** establishes the entry point for another, concurrent thread of execution within ourprogram. This thread will end when **run( )** returns

Implementing Runnable(contd.)

1. Create a class that implements **Runnable.**
2. Instantiate an object of type **Thread** from within that class.
   - ✓ Thread defines several constructors.

Thread(Runnable *threadOb, String threadName)*

   - Here, *threadOb* is an instance of a class that implements the **Runnable interface.** This defines where execution of the thread will begin.
   - The name of the new thread is specified by *threadName*.

1. After the new thread is created, it will start running when we call its **start( ) method,** which is declared within **Thread.**

- **start( )** executes a call to **run( ).**
- The **start( )** method declaration is:

void **start( )**


Java

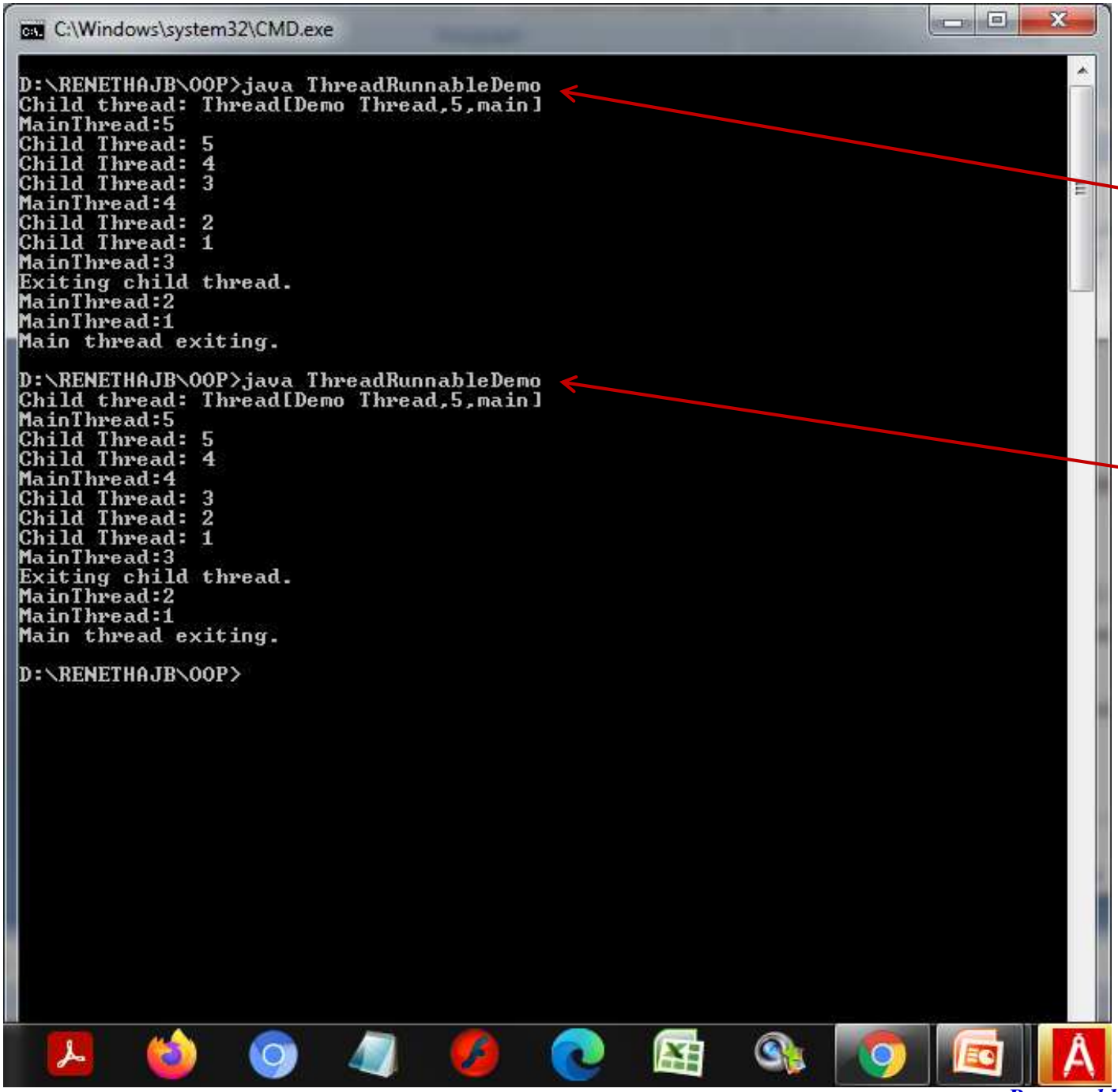
```
class NewThread implements Runnable
 {
      Thread t;
      NewThread()
      {
            t = new Thread(this, "Demo Thread");        //create new Thread
            System.out.println("Child thread: " + t);
            t.start();                                  //thread starts. Calls run()
      }
      public void run()
      {                                 // action to be done by thread
            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.");
      }
}
```

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

Implementing Runnable(contd.)

```
NewThread implements Runnable {
Thread t;
NewThread()
  {
 t = new Thread(this, "Demo Thread");
 System.out.println("Child thread: " + t);
 t.start();
  }
public void run() {
try {
       for(int i = 5; i > 0; i--) {
       System.out.println("Child Thread: " + i);
       Thread.sleep(500);       }
       } catch (InterruptedExce ption e)
   { System.out.println("Child interrupted.");
    }
System.out.println("Exiting child thread.");
} }
```

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



Execution

Execution

Implementing Runnable(contd.)

- Inside NewThread's constructor, a new Thread object is created by the following statement:

t = new **Thread**(**this**, "Demo Thread");

- Passing **this** as the first argument indicates that we want the new thread to call the run( ) method on this object.
- Next, **start( )** is called, which starts the thread of execution beginning at the **run( )** method.
- This causes the child thread's for loop to begin. After calling start( ), NewThread's constructor returns to main( ).
- When the main thread resumes, it enters its for loop. Both threads continue running, sharing the CPU, until their loops finish. The output produced by this program may vary based on processor speed and task load.

Extending Thread

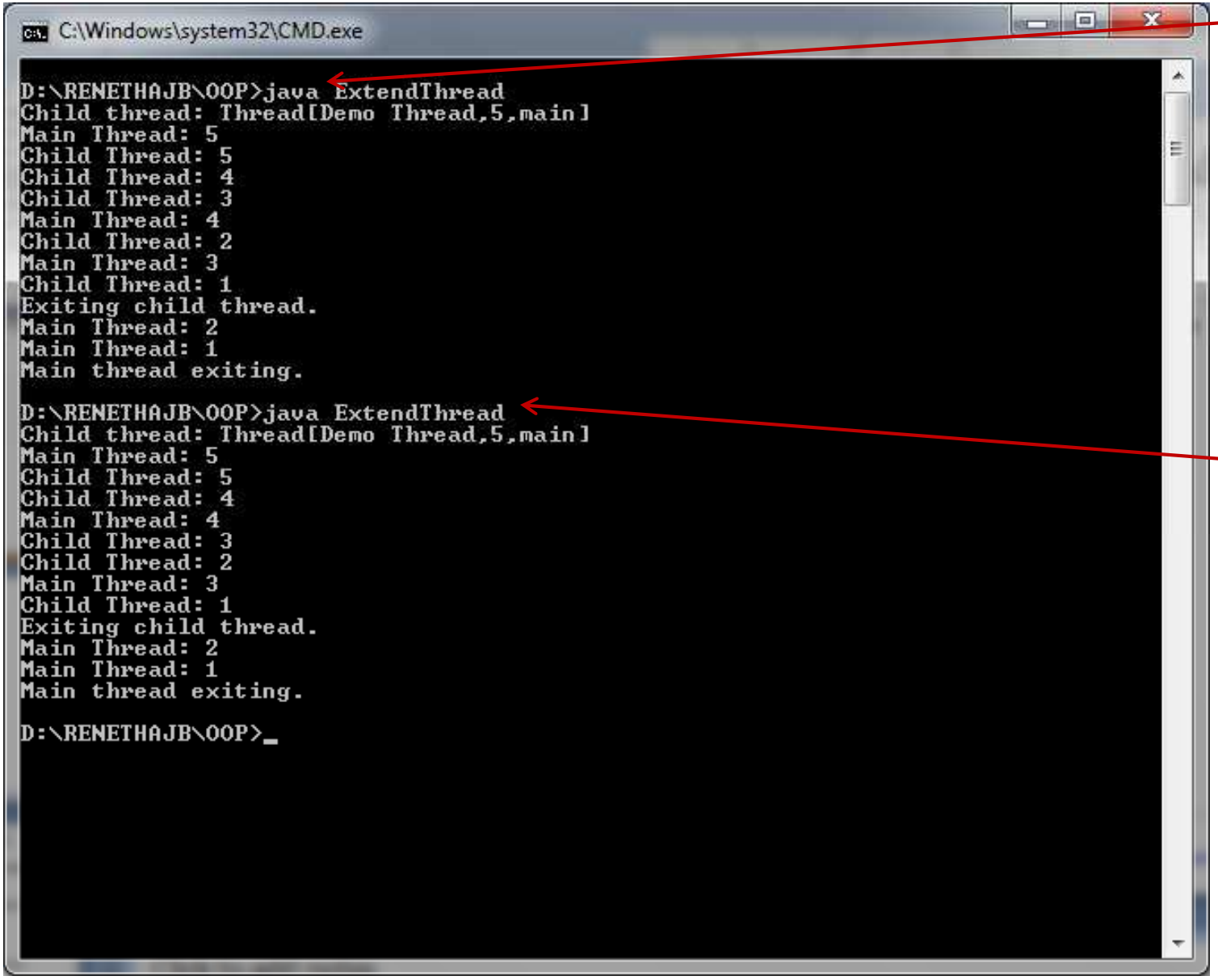
- Another way to create a thread is to
  - Create a **new class that extends Thread,**
    - The extending class must **override** the **run( )** method, which is the entry point for the new thread.
    - It must also call **start( )** to begin execution of the new thread
  - then create an **instance of that class.**

```
class NewThread extends Thread
{
    NewThread()
    {
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start();
    }
    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.");
}     }
```

```
class ExtendThread
{
    public static void main(String args[])
    {
        new NewThread();
        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.");
    }
}
```

```
class NewThread extends Thread
{
    NewThread()
    {
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start();
    }
    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.");
    }
 }
```

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



Execution

Execution

Extending Thread(contd.)

- The child thread is created by instantiating an object of class **NewThread**, which is derived from **Thread**.
- The call to **super( )** inside NewThread invokes the **Thread** constructor:

```
public Thread(String threadName)
```

  - Here, *threadName specifies the name of the thread*

Choosing an Approach

- If we **will not be overriding any of Thread's other methods,** it is probably best simply to implement **Runnable.**
- The classes should be **extended using Thread** only when they are being **enhanced or modified** in some way.

Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.