Multithreading

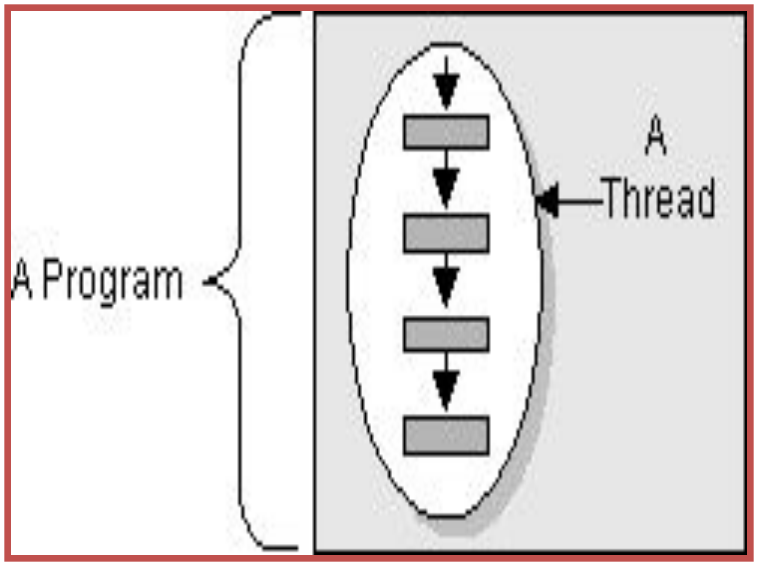
Objective

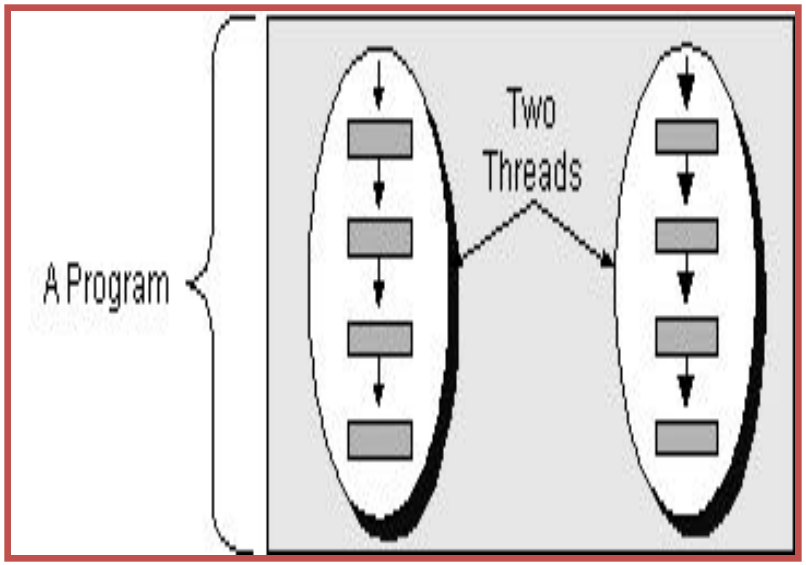
In this session, you will be able to:

- Define a thread
- Create threads in java using two approaches
- Describe the life cycle of a thread
- Synchronize threads

Thread

- A thread is a single sequential flow of control within a program.
- lightweight process
- execution context





Threads in Java

- There are two techniques for creating a thread:
  - Subclassing *Thread* and Overriding run
  - Implementing the *Runnable* Interface

The *Runnable* Interface

- The *Runnable* interface should be implemented by any class whose instances are intended to be executed by a thread.
- The class must define a method of no arguments called run.
- public void **run()**
  - When an object implementing interface *Runnable* is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.

The Thread Class

- **Thread()**
  - Allocates a new Thread object.
- **Thread(Runnable target)**
  - Allocates a new Thread object.
- **Thread(Runnable target,String name)**
  - Allocates a new Thread object.
- **Thread(String name)**
  - Allocates a new Thread object.

Subclassing Thread

```
class SimpleThread extends Thread {
        public SimpleThread(String str) {
            super(str);
        }
        public void run() {
            for (int i = 0; i < 10; i++) {
                System.out.println(i + " " + getName());
                try {
                    sleep((long)(Math.random() * 1000));
                } catch (InterruptedException e) {}
            }
            System.out.println("DONE! " + getName());
        }
}
```

Implementing the Runnable Interface

```
class SimpleThread implements Runnable {
        public void run() {
                Thread t = Thread.currentThread();
                for (int i = 0; i < 10; i++) {
                        System.out.println(i + " " + t.getName() );
                        try {
                                Thread.sleep((long)(Math.random() * 1000));
                        } catch (InterruptedException e) {}
                }
                System.out.println("DONE! " + t.getName());
        }
}
```

Implementing the Runnable Interface

```
import java.util.*;
import java.text.DateFormat;
import java.applet.Applet;
public class Clock extends Applet implements Runnable      {
        private Thread clockThread = null;
        public void start() {
                if (clockThread == null)          {
                clockThread = new Thread(this, "Clock");
                        clockThread.start();
                }
        }//start
        public void run()         {
                Thread myThread = Thread.currentThread();
                while (clockThread == myThread)      {
                        repaint();
                try       {
                                Thread.sleep(1000);
                        } catch (InterruptedException e){}
                }//while
 }//run
```

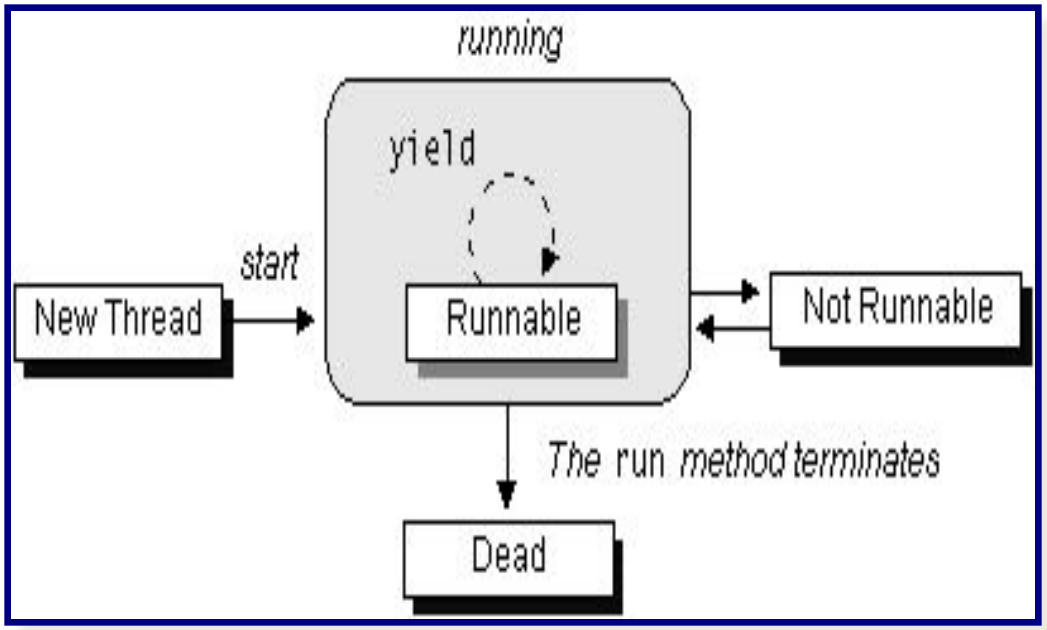
Implementing the Runnable Interface

```
    public void paint(Graphics g)  {
     // get the time and convert it to a date
          Calendar cal = Calendar.getInstance();
          Date date = cal.getTime();
          // format it and display it
          DateFormat dateFormatter = DateFormat.getTimeInstance();
          g.drawString(dateFormatter.format(date), 50, 50);
    }//paint

    // overrides Applet's stop method
    public void stop()    {
          clockThread = null;
    }//stop
}// Clock
```

The Life Cycle of a Thread

- Creating a Thread
- Starting a Thread
- Making a Thread Not Runnable
- Stopping a Thread



The Life Cycle of a Thread

- Creating a Thread:

```
public void start() {
        if (clockThread == null) {
                clockThread = new Thread(this, "Clock");
                clockThread.start();
        }
}
```

- After the statement containing new is executed, clockThread is in the *New Thread* state; no system resources are allocated for it yet.

- When a thread is in this state, we can only call the start method on it.

The Life Cycle of a Thread

- Starting a Thread :

```
public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();}}
```

- The start method creates the system resources necessary to run the thread, schedules the thread to run, and calls the thread's run method.

- After the start method returns, the thread is said to be "running". In reality a thread that has been started is actually in the Runnable state.

- At any given time, a "running" thread actually may be waiting for its turn in the CPU.

The Life Cycle of a Thread

- The run( ) method :

```
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e){}
    }}
```

- Clock's run method loops while the condition
  clockThread = = myThread is true.
- Within the loop, the applet repaints (which further calls paint) itself and then tells the thread to sleep for one second.

The Life Cycle of a Thread

- Making a Thread Not Runnable:

- A thread becomes Not Runnable when one of these events occurs:
  - Its **sleep( )** method is invoked.
  - The thread calls the **wait( )** method to wait for a specific condition to be satisifed.
  - The thread is blocking on I/O.

The Life Cycle of a Thread

- The clockThread in the Clock applet becomes *Not Runnable* when the run method calls *sleep* on the current thread:

```
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e){}
    }
}
```

The Life Cycle of a Thread

- Stopping a Thread :
- A program doesn't stop a thread, rather, a thread arranges for its own death by having a run method that terminates naturally. For example, the while loop in this run method is a finite loop-- it will iterate 100 times and then exit:

```
public void run() {
    int i = 0;
        while (i < 100) {
           i++;
           System.out.println("i = " + i);
        }   }
```

- A thread with this run method dies naturally when the loop completes and the run method exits.

The Life Cycle of a Thread

- Stopping a Thread :
  - Stopping the Clock applet thread

```
public void run() {
        Thread myThread = Thread.currentThread();
        while (clockThread == myThread) {
                repaint();
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e){}
        }}
```

- The exit condition for this run method is the exit condition for the while loop because there is no code after the while loop:

```
while (clockThread = = my Thread)
```

The Life Cycle of a Thread

- Stopping a Thread :
  - Stopping the Clock applet thread

- When we leave the page, the application in which the applet is running calls the applet's stop method. This method then sets the clockThread to null, thereby telling the main loop in the run method to terminate:

```
public void stop() {// applets' stop method
clockThread = null; }
```

- If we revisit the page, the start method is called again and the clock starts up again with a new thread.

The Life Cycle of a Thread

- The **isAlive()** Method :
  - The **isAlive()** method returns true if the thread has been started and not stopped.
  - If the **isAlive()** method returns false, you know that the thread either is a New Thread or is dead.
  - If the **isAlive()** method returns true, you know that the thread is either Runnable or Not Runnable.

Thread Priority

- The Java runtime supports fixed priority preemptive scheduling.
- When a Java thread is created, it inherits its priority from the thread that created it. Which can be changed later using the *setPriority* method.
- Thread priorities are integers ranging between MIN_PRIORITY and MAX_PRIORITY.

Thread Priority

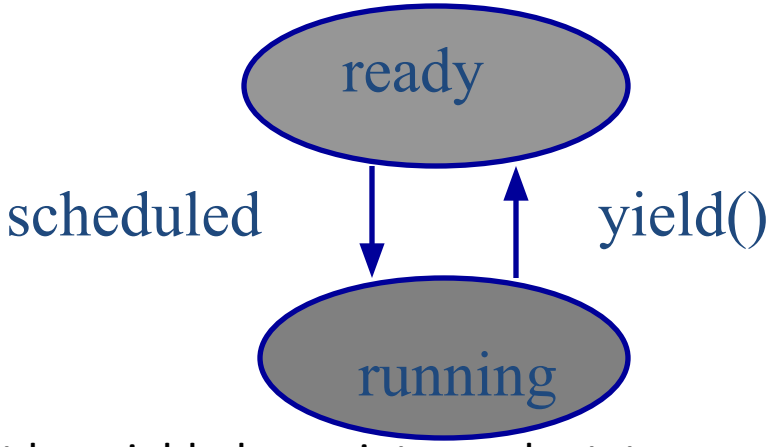
- The runtime system chooses the runnable thread with the highest priority for execution.
- If two threads of the same priority are waiting for the CPU, the scheduler chooses one of them to run in a round-robin fashion.
- The chosen thread will run until one of the following conditions is true:
  - A higher priority thread becomes runnable.
  - It yields, or its run method exits.
  - On systems that support time-slicing, its time allotment has expired.

Controlling Threads

- It’s an art of moving threads from one state to another state.
- It is achieved by triggering state transitions.
- The various pathways out of the running state can be :
  - Yielding
  - Suspending and then resuming
  - Sleeping and then waking up
  - Blocking and then continuing
  - Waiting and then being notified

Yielding

- A call to yield method causes the currently executing thread to move to the ready state, if the scheduler is willing to run any other thread in place of the yielding thread.



- A thread that has yielded goes into ready state.
- The yield method is a static method of the Thread class.

Suspending

- It’s a mechanism that allows any arbitrary thread to make another thread un-runnable for an infinite period of time.
- The suspended thread becomes runnable when some other thread resumes it
- Deadlock prone, because the control comes from outside the thread.
- The exact effect of suspend and resume is much better implemented using wait and notify.

Sleeping

- A sleeping thread passes time without doing anything and without using the CPU.
- A call to the sleep method requests the currently executing thread to cease it's execution for a specified period of time.
  - public static void sleep(long ms)
    throws InterruptedException
  - public static void sleep(long ms, int ns)
    throws InterruptedException
- Both sleep and yield work on currently executing thread.

Blocking

- Methods that perform input or output have to wait for some occurrence in the outside world before they can proceed, this behavior is known as blocking.

```
try   {
      Socket s = new Socket(“Devil”,5555);
      InputStream is = s.getInputStream();
      int b = is.read();
}
catch(IOException e)      {
// Handle the exception}
```

Blocking

- A thread can also become blocked if it fails to acquire the lock for a monitor if it issues a **wait( )** call.

- The **wait( )** method puts an executing thread into the waiting state, and the **notify( )** and **notifyAll( )** put them back to ready state.

Thread Synchronization

```
public class Consumer extends Thread  {
        private MailBox mailBoxObj;
        public Consumer(Mailbox box)      {
                this.mailBoxObj = box;
        }
        public void run()     {
                while(true)     {
                        if(mailBoxObj.request)   {
                        System.out.println(mailBoxObj.msg);
                                mailBoxObj.request = false;
                        }
                try   {
                        sleep(50);
                } catch(InterruptedException e){}
        }
}
```

Better Solution

```
class MailBox{
        private boolean request;
        private String msg;
        public synchronized void storeMessage( String s){
                request = true;
                msg = s;
        }
        public synchronized String retrieveMessage(){
                request = false;
                return msg;
        }
}
```

The Complete Solution

```
public synchronized String retrieveMessage(){
            while(request == false){ // No message to retrieve
                        try{
                        wait();
                  } catch (InterruptedException e) {}
            }
            request = false;
            notify();
            return str;
}
```

Summary

In this session, you learnt to:

- Define a thread
- Create threads using two approaches
- Describe the life cycle of a thread
- Synchronize threads