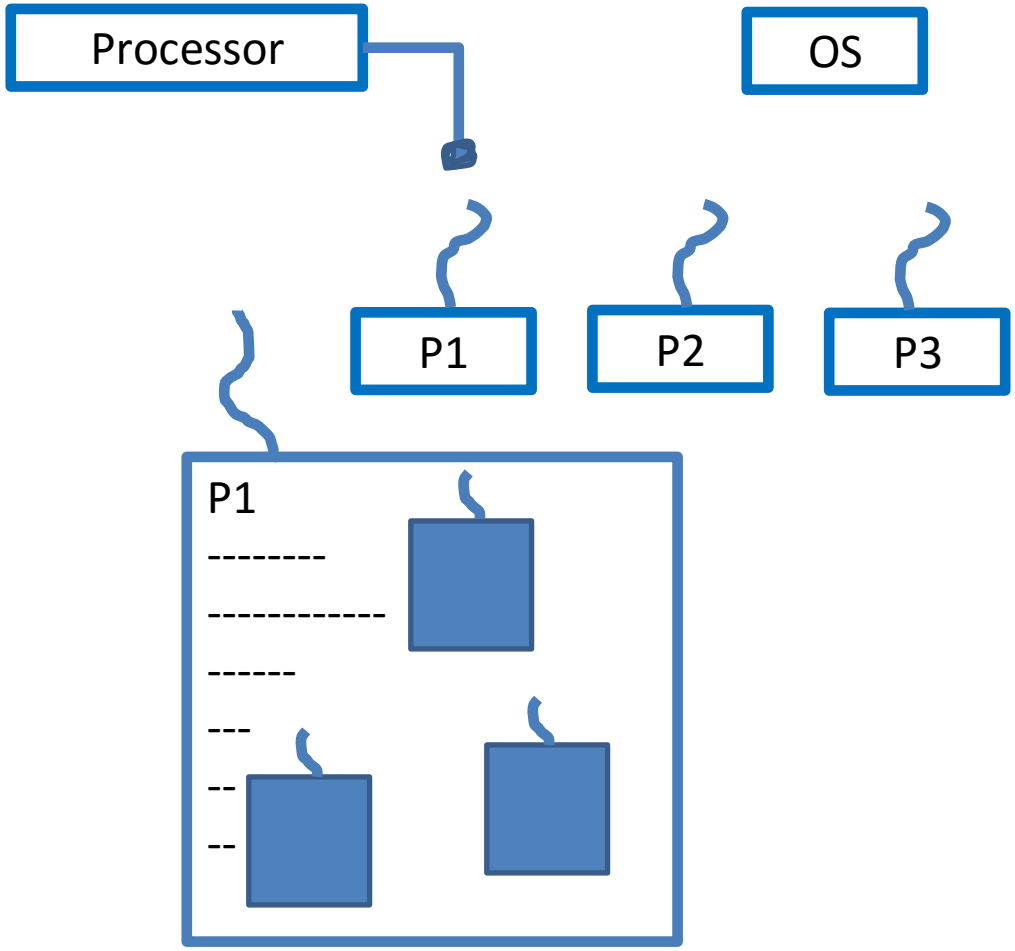
Multithreading Vs. Multitasking



Multithreading

- Ability to execute several programs simultaneously: **multitasking** or **multiprogramming.**
- A program (process) is divided into two or more subprograms (processes) implemented at the same time (**multithreading**).
- Single processor switches between processes very fast so that it appears to be running simultaneously.
- A thread is similar to a program that has a single flow of control.
- However, Java supports multithreading, i.e. multiple flows of control in a single program are possible.

- Individual flow of controls are called as threads.
- The ability of a language to support multithreading is called as **concurrency**.
- Threads are called as *lightweight threads* or *lightweight processes*.
- Threads are extensively used in java enabled browsers where many things are happening on a web page simultaneously.

What is thread?

- A thread is an independent path of execution within a program.
- Many threads can run concurrently within a program
- Multithreading refers to two or more tasks executing concurrently within a single program.

Creating Thread

- There are two ways to create thread in java;
  - Implement the Runnable interface (java.lang.Runnable)
  - By Extending the Thread class (java.lang.Thread)

- Threads are implemented in the form of objects that contain the method called **'run()'** .
- The run() method is heart and soul of any thread, through which thread's behaviour can be implemented.
- Syntax:

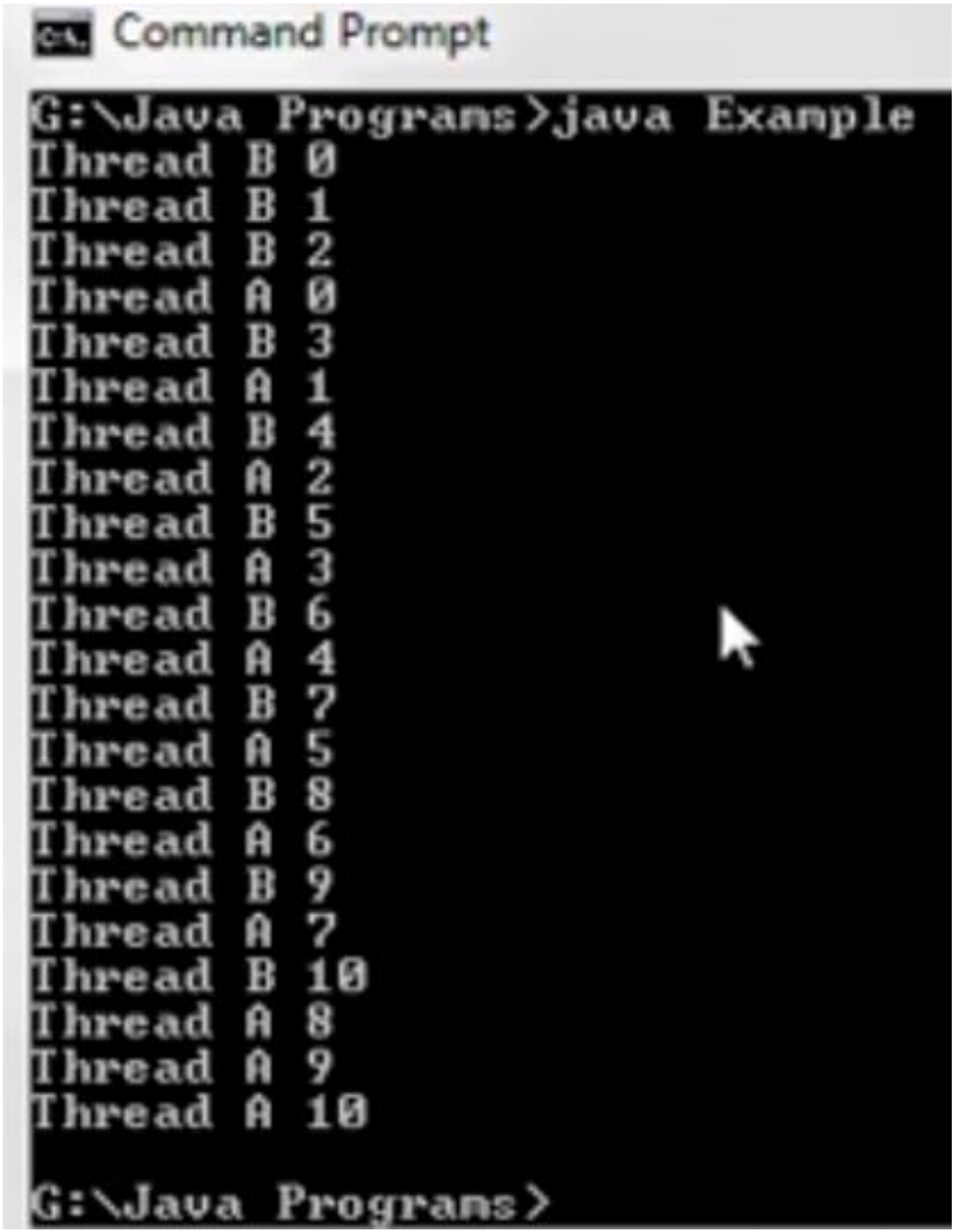
```
public void run()
{
.....
.....
}
```

Our objective

- Creating Thread
- Attach code to the thread
- Executing thread

```
class A implements Runnable
{
 public void run()
 {
  int i;
  for(i=0;i<=10;i++)
   System.out.println("Thread A "+i);
 }
}
class B implements Runnable
{
 public void run()
 {
  int i;
  for(i=0;i<=10;i++)
   System.out.println("Thread B "+i);
 }
}
```

```
public class Example
{
 public static void main(String[] args)
 {
  Thread t1=new Thread(new A());
  Thread t2=new Thread(new B());
  t1.start();
  t2.start();
 }
}
```


Command Prompt
G:\Java Programs>java Example
Thread B 0
Thread B 1
Thread B 2
Thread A 0
Thread B 3
Thread A 1
Thread B 4
Thread A 2
Thread B 5
Thread A 3
Thread B 6
Thread A 4
Thread B 7
Thread A 5
Thread B 8
Thread A 6
Thread B 9
Thread A 7
Thread B 10
Thread A 8
Thread A 9
Thread A 10
G:\Java Programs>

```
public interface Runnable {
    void run();
}
```

- One way to create a thread in java is to implement the Runnable Interface and then instantiate an object of the class
- We need to override the run() method into our class which is the only method that needs to be implemented

Steps

- An object of Thread class is created by passing a Runnable object as argument to the Thread constructor. The Thread object now has a Runnable object that implements the run() method.
- The start() method is invoked on the Thread object created in the previous step. The start() method returns immediately after a thread has been spawned

- The thread ends when the run() method ends, either by normal completion or by throwing an uncaught exception

Extending the thread class

- We can make our class runnable as thread by extending the class **java.lang.Thread**
- This gives us access to all the thread methods directly.
- Steps:
  - Declare the class as extending to Thread class
  - Implement the **run()** method.
  - Create a thread object and call **start()** method to initiate thread execution

Declaring the class

- The Thread class can be extended as follows.

```
Class MyThread extends Thread

{

…….

……

}
```

We have now a new type of thread **'MyThread'.**

Implementing the run() method

- The run method has now been inherited by the class MyThread.
- We have to now override this method to implement the code to be executed by our thread.

```
Public void run()
{
.....          //thread code here
}
```

- When we start the new thread java calls the thread's run() method, so it is actually run() where all the action takes place.

```
class A extends Thread
{
 public void run()
 {
  int i;
  for(i=1;i<=10;i++)
   System.out.println("i="+i+"Thread A");
 }
}
class B extends Thread
{
 public void run()
 {
  int i;
  for(i=1;i<=10;i++)
   System.out.println("i="+i+"Thread B");
 }
}
```

```
public class Example
{
 public static void main(String []args)
 {
  A o1=new A();
  B o2=new B();
  o1.start();
  o2.start();
 }
}
```


Command Prompt
G:\Java Programs>javac Example.java
G:\Java Programs>java Example
i=1Thread A
i=2Thread A
i=3Thread A
i=4Thread A
i=5Thread A
i=6Thread A
i=1Thread B
i=2Thread B
i=3Thread B
i=4Thread B
i=7Thread A
i=8Thread A
i=5Thread B
i=9Thread A
i=10Thread A
i=6Thread B
i=7Thread B
i=8Thread B
i=9Thread B
i=10Thread B
G:\Java Programs>_

Starting New Thread

- To actually create and run an instance of our thread class:

```
MyThread aThread = new MyThread();

aThread.start();
```

- The first line instantiates a new object of class MyThread.
- Here the thread is in **newborn** state.
- The second line calls the start() method causing the thread to go in **runnable** state.
- The java runtime will schedule the thread to run by invoking its run() method, here the thread will move into **running** state.

Thread States

- A Java thread is always in one of several states which could be running, sleeping, dead, etc.
- States:
  - New thread
  - Runnable
  - Not Runnable
  - Dead

New Thread

- A thread is in this state when the instantiation of a **Thread** object creates a new thread but does not start it running.
- A thread starts life in the Ready-to-run state.
- You can call only the **start()** or **stop()** methods when the thread is in this state.
- Calling any method besides **start()** or **stop()** causes an IllegalThreadStateException. (A descendant class of RuntimeException)

Runnable

- When the **start()** method is invoked on a New Thread() it gets to the runnable state or running state by calling the run() method.

- A Runnable thread may actually be running, or may be awaiting its turn to run.

Not Runnable

- A thread becomes Not Runnable when one of the following four events occurs:
- When **sleep()** method is invoked and it sleeps for a specified amount of time
- When **suspend()** method is invoked
- When the **wait()** method is invoked and the thread waits for notification of a free resource or waits for the completion of another thread or waits to acquire a lock of an object
- The thread is blocking on I/O and waits for its completion

Switching from not runnable to runnable

- If a thread has been put to sleep, then the specified number of milliseconds must elapse (or it must be interrupted).
- If a thread has been suspended, then its **resume()** method must be invoked
- If a thread is waiting on a condition variable, whatever object owns the variable must relinquish it by calling either **notify()** or **notifyAll()**.
- If a thread is blocked on I/O, then the I/O must complete.

Dead State

- A thread enters this state when the **run()** method has finished executing or when the **stop()** method is invoked. Once in this state, the thread cannot ever run again.

Lifecycle of a Thread

States:

1) Newborn
2) Runnable
3) Running
4) Blocked
5) Dead

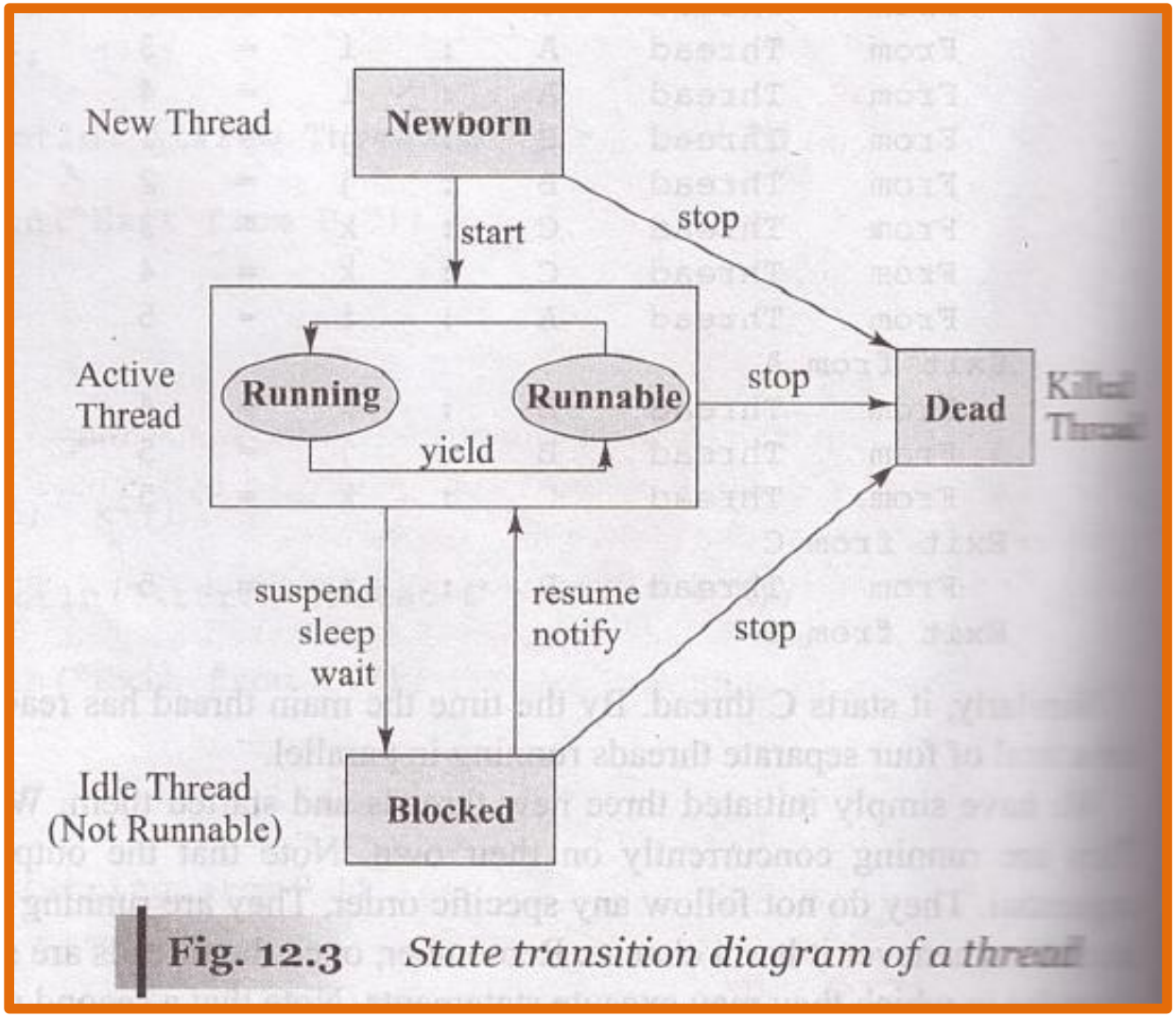


Fig. 12.3 *State transition diagram of a thread*

Stopping and Blocking a Thread

- A running thread at any stage can be stopped by calling its stop() method.

```
aThread.Stop();
```

- This causes a thread to move to a dead state.
- A thread will also move to a dead state automatically when it reaches to the end of its method.
- stop() may be used when a premature death of a thread is desired.

Blocking a Thread

- A running thread can be temporarily suspended or blocked from entering into runnable and subsequently running state by using either of the following thread methods.

  sleep() //blocked for a specified time

  suspend() //blocked until further orders

  wait() //blocked until certain condition occurs

- The thread will return back to runnable state when:
  - Specified time is elapsed in case of sleep()
  - **resume()** method is invoked in case of suspend()
  - **notify()** method is called in case of wait()

Thread Priority

- Priority affects the order in which the thread is scheduled for running.
- The threads of same priority are given equal treatment (first-come-first-serve).
- We can set the priority of a thread using **setPriority()** method:

```
ThreadName.setPriority(intNumber);
```

- The intNmber is an integer value to which the thread's priority is set.
- The Thread class defines several priority constants:
  - MIN_PRIORITY =1
  - NORM_PRIORITY =5
  - MAX_PRIORITY =10
- The higher priority thread always pre-empts the lower priority thread.

Thread Synchronization

- Threads use their own data and methods provided inside their run() methods.
- What if they try to use data and methods outside their run()?
- In such case they may compete for same resources at the same point of time and may lead to serious problems.
- Java enables us to overcome this problem using a technique known as **Synchronization.**
- The shared data and methods may be kept in a separate block labelled with the keyword *synchronized.*

```
Synchronized (method/object)
{
…… //code here is synchronized
}
```

Daemon Thread

- **Daemon thread** in java is a service provider thread that provides services to the user thread.
- Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.
- There are many java daemon threads running automatically e.g. gc, finalizer etc.
- It is a low priority thread.
- The java.lang.Thread class provides two methods for java daemon thread:

  1) public void setDaemon(boolean status) - used to mark the current thread as daemon thread or user thread.

  2) public boolean isDaemon() - is used to check that current thread is daemon?