



Multi-Threading



Chapter 2

Multithreading

Computer Science and Engineering Department@SoEEC
Advanced Programming(CSE 3312)



Key Objectives:

At the end of this chapter, you will be able to:

- Understand what multi-threaded programs are
- Know how to write multi-threaded programs
- Describe how to interrupt, suspend and resume threads
- Describe inter-thread communication
- Understand what synchronized methods and



1. Introduction

- We can walk, talk, breathe, see, hear, smell... all at the same time
 - Computers can do this as well - download a file, print a file, receive email, run the clock
 - Only computers that have multiple processors can truly execute multiple instructions concurrently
 - Operating systems on single-processor computers create the illusion of concurrent execution by rapidly switching between activities, but on such computers only a single instruction can execute at once.
 - Java makes concurrency available to you through the language and APIs.
- 

2. What are Threads

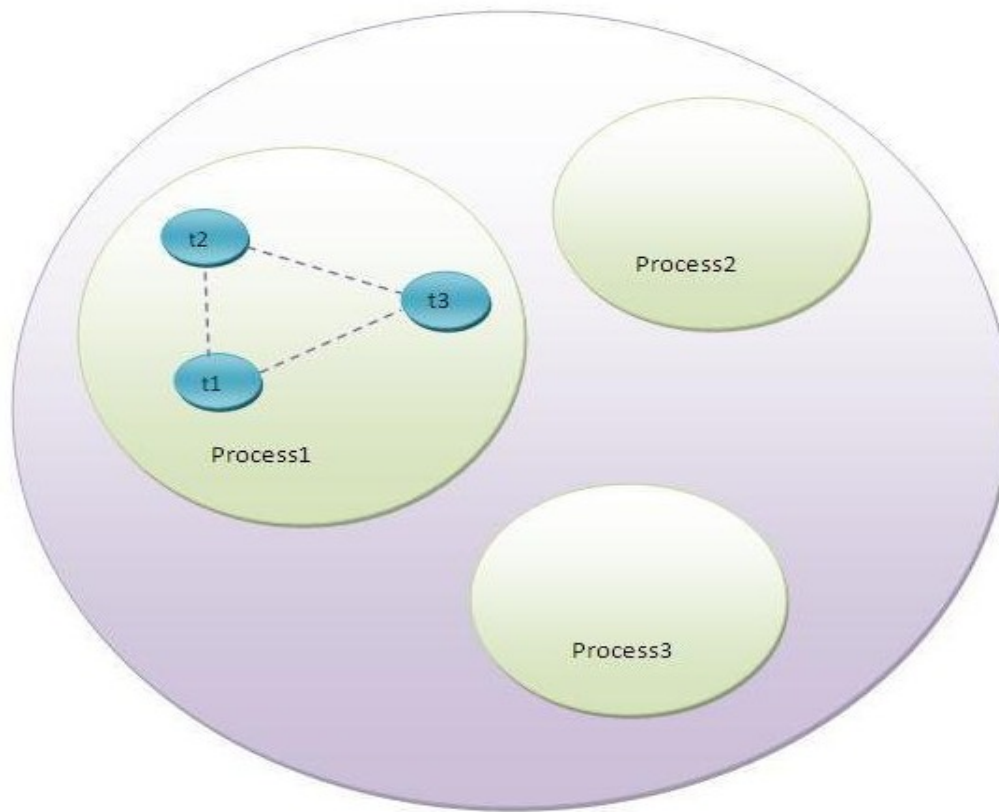
- A thread is a control/flow/path of execution that exists within a process.
- Thread is portion of a program that can execute concurrently with other threads.
- Each thread is a statically ordered sequence of instructions.
 - Sequential flow of control within a program
- Threads are being extensively used to express concurrency on both single and multiprocessors machines.
 - Sharing a single CPU between multiple tasks (or "threads") in a way designed to minimize the time required to switch tasks.
 - accomplished by sharing as much as possible of the program execution environment between the different tasks.
- Programming a task having multiple threads of control is called **Multithreading** or **Multithreaded Programming**.
- A multithreaded application contains separate threads of execution, where each thread

• Has its own method-call stack and program counter

.....What are Threads

- execute concurrently with other threads.
- shares application-wide resources such as memory with other threads.
- In single-threaded applications lengthy activities must complete before others can begin which leads to poor responsiveness.
- In a multithreaded application, threads can be distributed across multiple processors (if available) so that multiple tasks execute concurrently and the application can operate more efficiently.
- Multithreading can also increase performance on single-processor systems that simulate concurrency—when one thread cannot proceed (because, for example, it is waiting for the result of an I/O operation), another can use the processor.
- The Java Virtual Machine (JVM) creates threads to run a program, the JVM also may create threads for performing housekeeping

.....What are Threads



OS

As shown in the above figure, thread is executed inside the process.

There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

3. Multi-threading

- **Multi-threading in java** is a process of executing multiple threads simultaneously.
- Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
- 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(see figure above).
- Java Multithreading is mostly used in games, animation etc.

3.1 Why Multithreading ?

- To maintain responsiveness of an application during a long running task.
- To enable cancellation of separable tasks.
- Some problems are intrinsically parallel.
- To monitor status of some resource (DB).
- To take advantage of multiple processors.
- To perform multiple operations at the same time.



States)

- A thread occupies one of several thread states.
 - **Born (New) state**
 - Thread just created
 - A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.
 - **Runnable(Ready state)**
 - When the program starts the call thread (start method) it enters the *ready* state.
 - After invocation of start() method on new thread, the thread becomes runnable.
 - Highest-priority ready thread enters running state
 - **Running state**
 - A thread is in running state if the thread scheduler has selected it.
 - System assigns processor to thread (thread begins executing)
 - When the thread's quantum expires, the thread returns to the *Ready* state and the operating system dispatches another thread
 - Transitions between the ready and running states are handled solely by the operating system

....Thread States

- Blocked state

- Entered from running state when
 - waiting on I/O request
 - Wants to enter a synchronized statement
- cannot use processor, even if available

- Sleeping (Timed waiting) state

- Entered from running state when
 - sleep method called or
 - wait is called with a time interval.
- Cannot use processor , even if available
- transitions back to the *Ready* state when that time interval expires or when it is notified by another thread

- Waiting state

- Entered from running state when calls wait, the thread enters a waiting state for the particular object on which wait was called.
- One waiting thread becomes ready when object calls notify
- notifyAll - all waiting threads become ready
- cannot use a processor, even if one is available.
- In this stage the thread is still alive

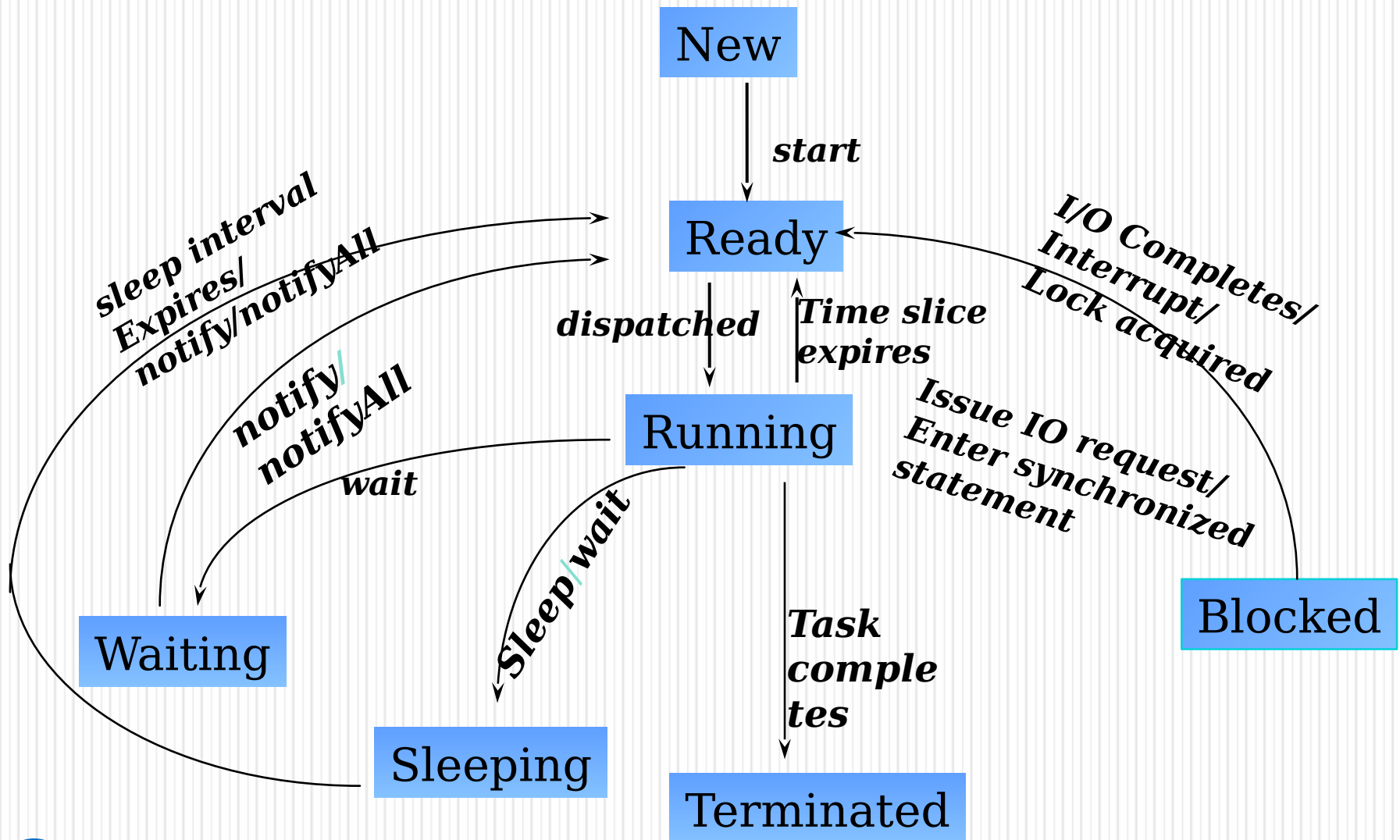


....Thread States

- Terminated(Dead) state
 - Thread marked to be removed by system
 - A thread enter the terminated when it complete its task.



....Thread States



5. Creating and Executing Threads

To implement multithreading, Java defines two ways by which a thread can be created.

1. Implementing the Runnable interface

- Runnable
 - represents a “task” that can be executed concurrently with other tasks
 - declares method `run` in which you place the code that defines the task to perform.
- Define a class that implements Runnable

```
class Task implements Runnable{  
    public void run(){//define the task here}  
}
```
- Instantiate the Thread class
 - invoke Thread constructor with an instance of this Runnable class
- Then, call `start` method of the Thread instance

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

5. Creating and Executing Threads

2. By Extending Thread class

- Define a subclass of `java.lang.Thread`

- Define a *run* method

```
class Task extends Thread{  
    public void run(){define the task here}  
}
```

- In another thread (e.g., the main), create an instance of the Thread subclass

- Then, call *start* method of that instance

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



Example

6. The Thread Class

- Constructors

 - Thread(threadName)***

 - Thread()***

 - Creates an auto numbered Thread of format Thread-1, Thread-2...

- Methods

 - ***void run()***

 - "Does work" of thread
 - Can be overridden in a subclass of Thread

 - ***start***

 - Launches thread, then returns to caller
 - Calls run
 - Error to call start twice for same thread (IllegalThreadStateException)

 - ***static sleep(milliseconds)***

 - Thread sleeps for number of milliseconds
 - Can give lower priority threads a chance to run



....Thread Class

- ***interrupt()***
 - Interrupts a thread
- ***static interrupted()***
 - Returns true if current thread interrupted
- ***isInterrupted()***
 - Determines if a thread is interrupted
- ***isAlive()***
 - Returns true if start has been called and not dead (run function has not completed)
- ***yield ()***
 - Cause the currently running thread to temporarily pause and allows other threads to execute.
- ***setName(threadName)***
 - Set name of the thread
- ***getName()***
 - Returns the name of the thread.



....Thread Class

- **toString()**

- Returns thread name, priority, and ThreadGroup

- **Join()**

- Wait for a thread to end
- No argument or 0 millisecond argument means thread will wait indefinitely
 - Can lead to deadlock



7. Thread Scheduler

- **Thread scheduler** in java is the part of the JVM that decides which thread should run.
- There is no guarantee that which runnable thread will be chosen to run by the thread scheduler. Only one thread at a time can run in a single process.
- The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.



Sleep method in java

- The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep() method in java

- The Thread class provides two overloaded methods for sleeping a thread:
 - `public static void sleep(long milliseconds) throws InterruptedException`
 - `public static void sleep(long milliseconds, int nanos) throws InterruptedException`



Sleep method in java

- Example

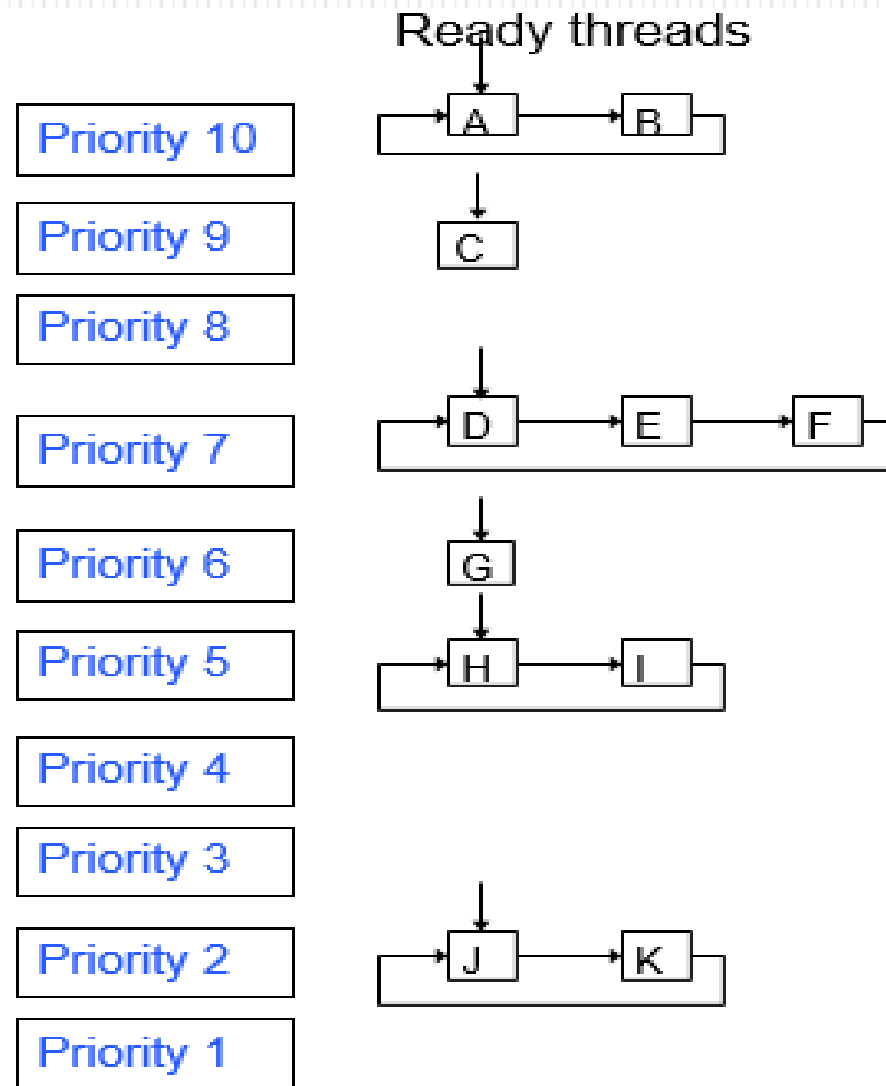
```
class TestSleepMethod1 extends Thread{
    public void run(){
        for(int i=1;i<5;i++){
            try
            {
                Thread.sleep(500);
            }
            catch(InterruptedException e)
            {
                System.out.println(e);
            }
            System.out.println(i);
        }
    }
    public static void main(String args[]){
        TestSleepMethod1 t1=new TestSleepMethod1();
        TestSleepMethod1 t2=new TestSleepMethod1();

        t1.start();
        t2.start();
    }
}
```

Thread Priorities and Thread Scheduling

- Every Java thread has a thread priority that helps the operating system determine the order in which threads are scheduled
- Priorities are represented by a number between `MIN_PRIORITY` (a constant of 1) and `MAX_PRIORITY` (a constant of 10).
- By default, every thread is given priority `NORM_PRIORITY` (a constant of 5)
- In most cases, thread scheduler schedules the threads according to their priority (known as **preemptive scheduling**). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

Thread Priorities and Thread Scheduling



Thread Priorities and Thread Scheduling

- Example:

```
class TestThreadPriority extends Thread{  
    public void run(){  
        System.out.println("running thread name is:"+Thread.currentThread().getName());  
        System.out.println("running thread priority is:"+Thread.currentThread().getPriority());  
    }  
  
    public static void main(String args[]){  
        TestThreadPriority m1=new TestThreadPriority();  
        TestThreadPriority m2=new TestThreadPriority();  
        m1.setPriority(Thread.MIN_PRIORITY);  
        m2.setPriority(Thread.MAX_PRIORITY);  
        m1.start();  
        m2.start();  
    }  
}
```

Notes

- start() is responsible for two things:
 - Instructing the JVM to create a new thread
 - Call your Thread object's run method in the new thread
- You might think of run as being similar to main
 - main is called by the JVM
 - Like main, the run method defines a starting point for the JVM
 - when the run method exits- The thread 'ends'
 - The thread's run() method must finish and return for the thread to stop
- The code in method main executes in the main thread, a thread created by the JVM

7. Thread Pools

- **Java Thread pool** represents a group of worker threads that are waiting for the job and reuse many times.
- In case of thread pool, a group of fixed size threads are created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, thread is contained in the thread pool again.
- **Executor interface** :- to manage the execution of Runnable objects
- An Executor object creates and manages a thread pool to execute Runnables
- Executor advantages over creating threads yourself
 - Reuse existing threads to eliminate new thread overhead
 - **Better performance:** It saves time because there is no need to create new thread.
- Executor method *execute()* accepts a Runnable as an argument
 - Assigns each Runnable it receives to one of the available threads in the thread pool
 - If none available, creates a new thread or waits for a thread to become

7.1 Thread Pools

- Interface *java.util.concurrent.ExecutorService*
 - extends Executor
 - declares methods for managing the life cycle of an Executor
 - Can be instantiated by using static method *newFixedThreadPool()* of class *java.util.concurrent.Executors*
- methods
 - *execute()* returns immediately from each invocation
 - *shutdown()* notifies the ExecutorService to stop accepting new tasks, but continues executing tasks that have already been submitted

.....

```
ExecutorService es = Executors.newFixedThreadPool();
```

```
es.execute(new Task());
```

```
es.execute(new Task());
```

[ThreadPoolDemo](#)



8. Thread Synchronization

- **Synchronization** in java is the capability *to control the access of multiple threads to any shared resource*.
- Java Synchronization is better option where we want to allow only one thread to access the shared resource.
- Other threads wait
- A shared resource may be corrupted if it is accessed simultaneously by multiple threads.



....Thread Synchronization

- Mutual exclusion
- Java provides built-in monitors to implement synchronization
- Every object has a monitor and a monitor lock.
 - Monitor ensures that its object's monitor lock is held by a maximum of only one thread at any time
 - Can be used to enforce mutual exclusion
- To enforce mutual exclusion
 - thread must acquire the lock before it can proceed with its operation
 - other threads attempting to perform an operation that requires the same lock will be blocked until the first thread releases the lock



.... Thread Synchronization

- synchronized statement
 - Enforces mutual exclusion on a block of code
- ```
synchronized (object){

statements

} // end synchronized statement
```
- where *object* is the object whose monitor lock will be acquired (normally [this](#))
- A synchronized method is equivalent to a synchronized statement that encloses the entire body of a method

# ....Thread Synchronization

## Why use Synchronization ?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

## Types of Synchronization

There are two types of synchronization

- i) Process Synchronization
- ii) Thread Synchronization

- Here, we will discuss only thread synchronization.



# ....Thread Synchronization

- There are two types of thread synchronization: **mutual exclusive** and **inter-thread communication**.

## 1) Mutual Exclusive

1. Synchronized method.
2. Synchronized block.

## 2) Cooperation (Inter-thread communication)



# Mutual Exclusive

- Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by two ways in java:
  - 1) by synchronized method
  - 2) by synchronized block

## Concept of Lock

- Synchronization is built around an internal entity known as the **lock** or **monitor**. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java package, [java.util.concurrent.locks](#) contains several lock implementations.

# Mutual Exclusive

## i) Synchronized method

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Syntax: `synchronized void display (String msg)`

---

Example



# Mutual Exclusive

## ii) Synchronized block

- Synchronized block can be used to perform synchronization on any specific resource of the method. Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.
- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

**Syntax to use** `synchronized` (object reference expression) {

```
 //code block
}
```



**Example**

# Inter-thread communication

- **Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.
- Inter-thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.
- It is implemented by following methods of Object class:
  - 1) wait()
  - 2) notify()



# Inter-thread communication

## 1) wait() method

- Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
- The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise

| Method                                                                       | Description                             |
|------------------------------------------------------------------------------|-----------------------------------------|
| <code>public final void wait()throws InterruptedException</code>             | waits until object is notified.         |
| <code>public final void wait(long timeout)throws InterruptedException</code> | waits for the specified amount of time. |



# Inter-thread communication

## 2) notify() method

- Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened.
- **Syntax:** `public final void notify()`

## 3) notifyAll() method

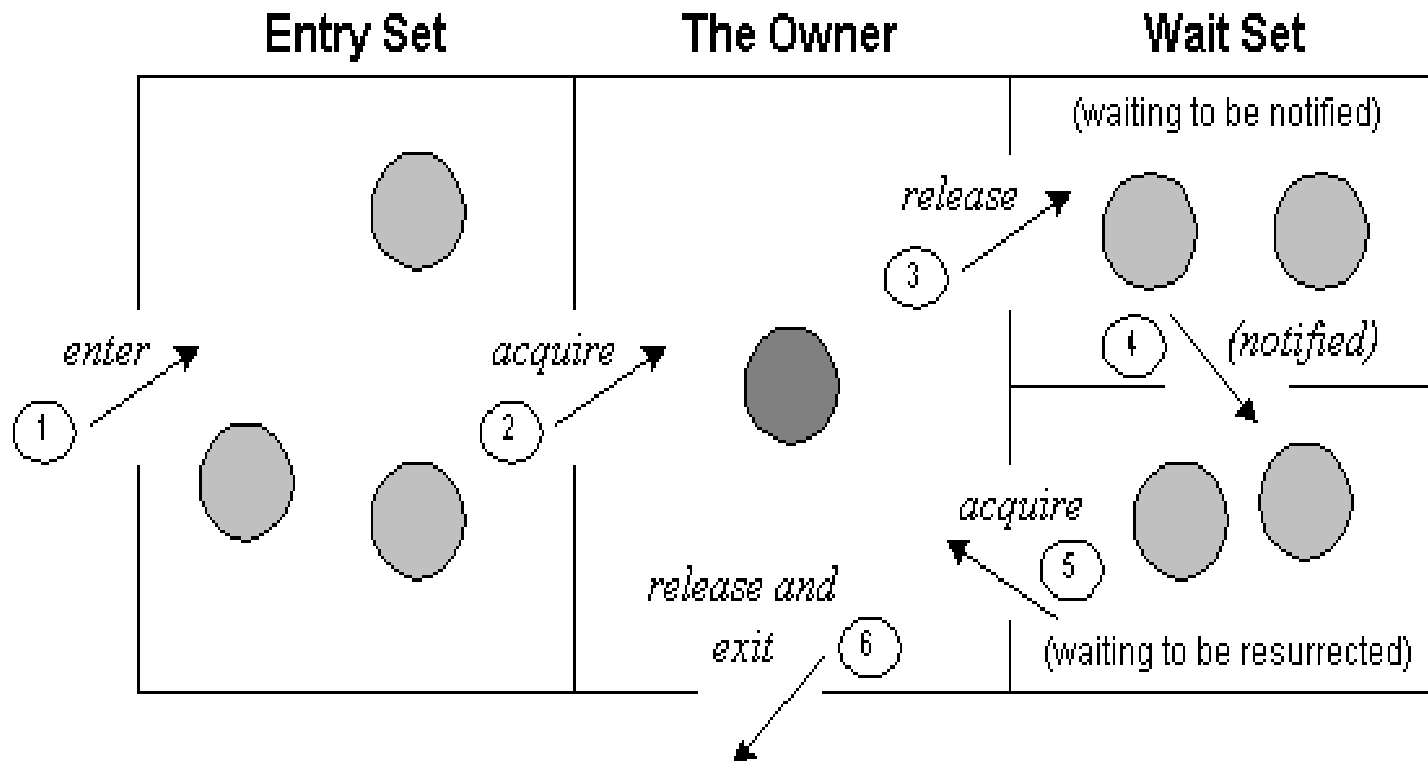
- Wakes up all threads that are waiting on this object's monitor.
- **Syntax:** `public final void notifyAll()`



**Example1 Example2**

# Inter-thread communication

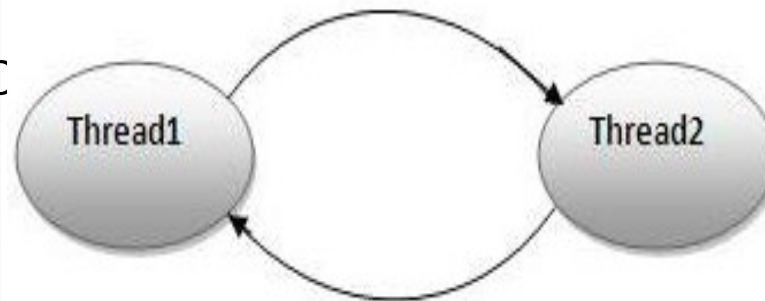
## The process of inter-thread communication



# Deadlock

- Deadlock in java is a part of multithreading.

Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for the lock, the condition is called deadlock.



# Producer/Consumer Relationship without Synchronization

- Multithreaded producer/consumer relationship
  - Producer thread generates data and places it in a shared object called a buffer
  - Consumer thread reads data from the buffer
- Operations on the buffer data shared by a producer and a consumer are state dependent
  - Should proceed only if the buffer is in the correct state
  - If in a not-full state, the producer may produce
  - If in a not-empty state, the consumer may consume
- Must synchronize access to ensure that data is written to the buffer or read from the buffer only if the buffer is in the proper state



# Producer/Consumer Relationship with Synchronization

- Can implement a shared using the synchronized keyword and Object methods wait, notify and notifyAll
  - can be used with conditions to make threads wait when they cannot perform their tasks
- A thread that cannot continue with its task until some condition is satisfied can call Object method wait
  - releases the monitor lock on the object
  - thread waits in the waiting state while the other threads try to enter the object's synchronized statement(s) or method(s)
- A thread that completes or satisfies the condition on which another thread may be waiting can call Object method notify
  - allows a waiting thread to transition to the *runnable* state
  - the thread that was transitioned can attempt to reacquire the monitor lock
- If a thread calls notifyAll, all the threads waiting for

# Producer/Consumer Relationship: Bounded Buffers

- Bounded buffer is used to minimize the amount of waiting time for threads that share resources and operate at the same average speeds
- If the buffer is full, the producer should wait until a consumer consumed a value to free an element in the buffer.
  - If the buffer is empty at any given time, a consumer thread must wait until the producer produces another value.
  - `ArrayBlockingQueue` is a bounded buffer that handles all of the synchronization details for you

## Example



# Producer/Consumer Relationship: The Lock and Condition Interfaces

- Give programmers more precise control over thread synchronization.
- Any object can contain a reference to an object that implements the `java.util.concurrent.locks .Lock` interface.
- A thread calls the Lock's `lock` method to acquire the lock.
- Once a lock has been obtained by one thread, the Lock object will not allow another thread to obtain the Lock until the first thread releases the Lock (by calling the Lock's `unlock` method).
- All other threads attempting to obtain that Lock on a locked object are placed in the *waiting* state
- Class `java.util.concurrent.locks.ReentrantLock` is a basic implementation of the Lock interface.
- `ReentrantLock` constructor takes a boolean argument that specifies whether the lock has a fairness policy
  - If true, the `ReentrantLock`'s fairness policy is "the longest-

# Producer/Consumer Relationship: The Lock and Condition Interfaces

- If false, there is no guarantee as to which waiting thread will acquire the lock.
- A thread that owns a Lock and determines that it cannot continue with its task until some condition is satisfied can wait on a condition object
- Lock objects allow you to explicitly declare the condition objects on which a thread may need to wait
- Condition objects
  - Associated with a specific Lock
  - Created by calling a Lock's `newCondition` method
- To wait on a Condition object, call the Condition's `await` method
  - immediately releases the associated Lock and places the thread in the *waiting* state for that Condition
- Another thread can call Condition method `signal` to allow a thread in that Condition's *waiting* state to return to the *runnable* state
  - Default implementation of Condition signals the longest-

# Producer/Consumer Relationship: The Lock and Condition Interfaces

- Condition method `signal` All threads waiting for that condition to the *runnable* state
- When finished with a shared object, thread must call `unlock` to release the Lock
- Lock and Condition may be preferable to using the `synchronized` keyword
  - Lock objects allow you to interrupt waiting threads or to specify a timeout for waiting to acquire a lock
  - Lock object is not constrained to be acquired and released in the same block of code
- Condition objects can be used to specify multiple conditions on which threads may wait
  - Possible to indicate to waiting threads that a specific condition object is now true



**Thank  
You!!**

