

# Chapter 3

## Multi threaded Programming

# 1. Introduction

- Java provides built-in support for multithreaded programming.
- A multithreaded program contains two or more parts that can run concurrently.
  - ✓ Each part of such a program is called a thread.
  - ✓ Each thread defines a separate path of execution.
- Multithreading is a specialized form of multitasking.
- There are two types of multitasking: process-based and thread-based.
- Process-based multitasking is the feature that allows your computer to run two or more programs concurrently.

# 1. Introduction

- ✓ Two tasks are operating concurrently means they're both making progress at once.
- ✓ Two tasks are operating in parallel means they're executing simultaneously.
- Processes are heavyweight tasks that require their own separate address spaces.
- In process-based multitasking,
  - ✓ Interprocess communication is expensive and limited.
  - ✓ Context switching from one process to another is costly
  - ✓ Process-based multitasking is not under Java's direct control

# 1. Introduction

- In a thread-based multitasking a single program can perform two or more tasks simultaneously.
- Multitasking threads require less overhead than multitasking processes.
- In a thread-based multitasking environment,
  - ✓ Threads are lighter weight.
  - ✓ Threads share the same address space and cooperatively share the same heavyweight process.
  - ✓ Inter-thread communication is inexpensive.
  - ✓ Context switching from one thread to the next is lower in cost.
  - ✓ It is under java's direct control.

## 2. Creating Threads

- All processes have at least one thread of execution, which is usually called the main Thread
- From the main thread, you can create other threads.
- You create a thread by instantiating an object of type Thread.
- The Thread class encapsulates an object that is runnable.
- Java defines two ways in which you can create a runnable object:
  - ✓ You can implement the Runnable interface.
  - ✓ You can extend the Thread class.
- To create a new thread, either extend Thread class or implement the Runnable interface.

## 2. Creating Threads

### The Thread class

- The Thread class defines several methods that help to manage Threads. Some of its methods are:
  - ✓ `final String getName()`
  - ✓ `final int getPriority()`
  - ✓ `final boolean isAlive()`
  - ✓ `void run()` :- Entry point for the thread
  - ✓ `void start()` :- starts a thread by calling its `run()` method
  - ✓ `static void sleep(long milliseconds)`:- suspends the thread
  - ✓ `final void join()` :- waits a thread to terminate.

## 2. Creating Threads

```
1 class MyThread extends Thread{
2     public MyThread(String name){ super(name); }
3     public void run(){
4         System.out.println(getName() + " starting");
5         try{
6             for(int i = 0; i<5 ; i++){
7                 Thread.sleep(1000);
8                 System.out.println(getName() + ", count: " + i);
9             } } catch(InterruptedException e){}
10    }
```

## 2. Creating Threads

```
1 public class create {  
2     public static void main(String args[]) {  
3         System.out.println(Thread.currentThread.getName() + " start");  
4         MyThread thread1 = new MyThread("Thread 1");  
5         thread1.start();  
6         MyThread thread2 = new MyThread("Thread 2");  
7         thread2.start();  
8         System.out.println(Thread.currentThread.getName() + " end");  
9     }  
10 }
```



## 2. Creating Threads

### The Runnable interface

- You can construct a thread on any object that implements the Runnable interface.
- Runnable defines only one method called run( ):  
**public void** run()
- Inside run( ), you will define the code that constitutes the new thread.
- run( ) establishes the entry point for another concurrent thread of execution within your program.
- This thread will end when run( ) returns.

## 2. Creating Threads

```
1  class MyThread2 implements Runnable{
2      public void run(){
3          System.out.println(Thread.currentThread().getName() + " starting");
4          try{
5              for(int i = 0; i<5 ; i++){
6                  System.out.println(Thread.currentThread().getName() + ", count: " +i);
7                  Thread.sleep(1000);
8              }
9          }catch(InterruptedException e){}
10     } }
```

## 2. Creating Threads

```
1 public class creat1 {  
2     public static void main(String args[]) {  
3         System.out.println(Thread.currentThread().getName() + " start" );  
4         MyThread2 thrd1 = new MyThread2();  
5         Thread mt1 = new Thread(thrd1);  
6         mt1.start();  
7         MyThread2 thrd2 = new MyThread2();  
8         Thread mt2 = new Thread(thrd2, "Adama");  
9         mt2.start();  
10        System.out.println(Thread.currentThread().getName() +" end");  
11    } }
```

## 2. Creating Threads

- We can add a Thread member variable and a constructor to MyThread class as:

```
1 Thread thrd; //a member variable of type Thread
2 public MyThread(String name){ //a constructor
3     thrd = new Thread(this, name);
4 }
```

## 2. Creating Threads

➤ Then it's run() method will be:

```
1 public void run(){
2     System.out.println(thrd.getName() + " starting");
3     try{
4         for(int i = 0; i<5 ; i++){
5             System.out.println(thrd.getName() + ", count: " +i);
6             Thread.sleep(1000);
7         }
8     }
9     catch(InterruptedException e){}
```

## 2. Creating Threads

➤ And the main() method will be:

```
1 public static void main(String args[]) {  
2     System.out.println(Thread.currentThread().getName() + " start" );  
3     MyThread thread1 = new MyThread("Adama");  
4     thread1.thrd.start();  
5     MyThread thread2 = new MyThread("Hawassa");  
6     thread2.thrd.start();  
7     System.out.println(Thread.currentThread().getName() +" end");  
8 }
```

## 2. Creating Threads

- We can also add a factory method to MyThread class as:

```
1 public static MyThread createThrd(String name){  
2     MyThread myThrd = new MyThread(name);  
3     myThrd.thrd.start();  
4     return myThrd;  
5 }
```

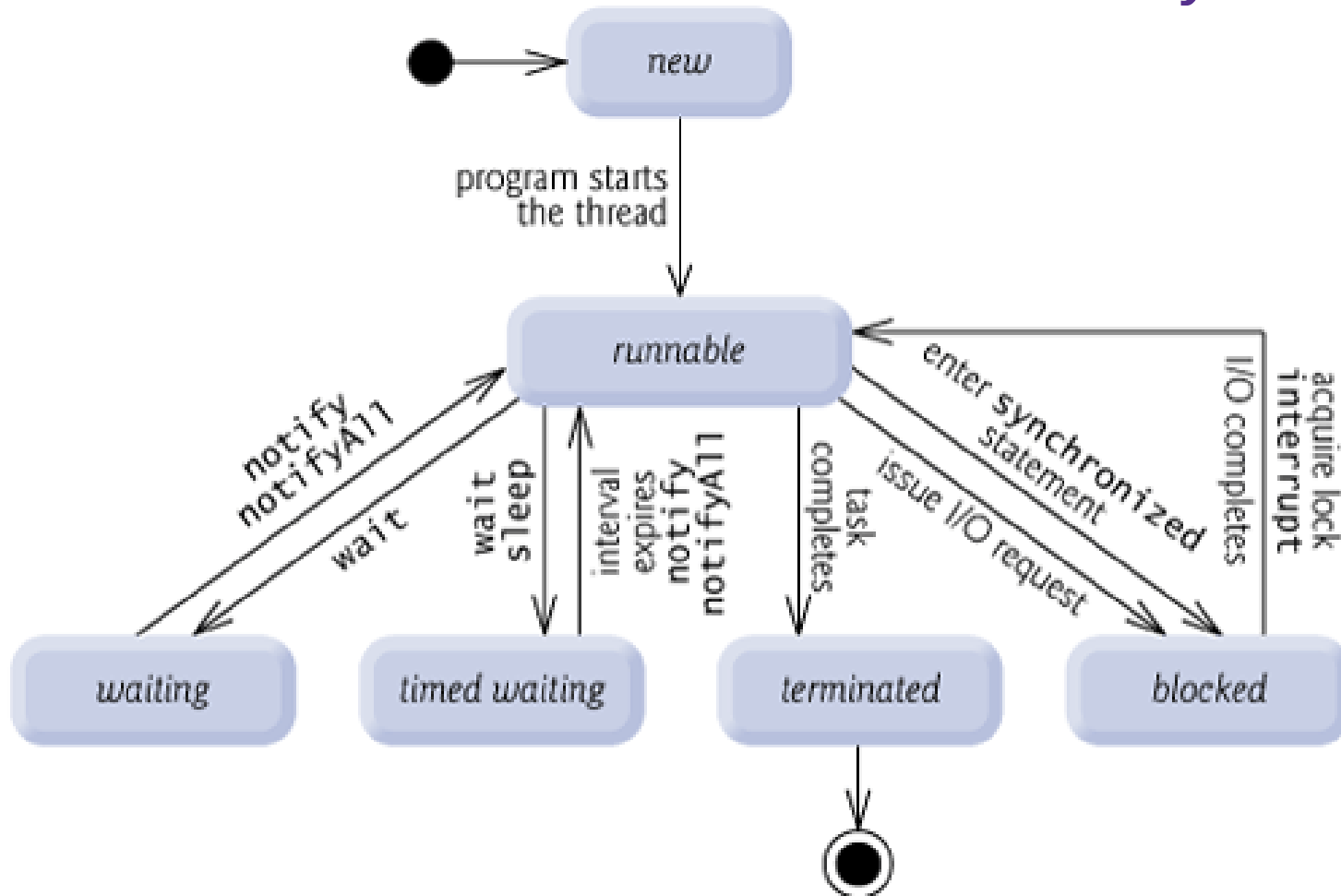
## 2. Creating Threads

➤ So that the main() method will be:

```
1 public static void main(String args[]) {  
2     System.out.println(Thread.currentThread().getName() + " start" );  
3     MyThread.createThrd("Adama");  
4     MyThread.createThrd("Hawassa");  
5     System.out.println(Thread.currentThread().getName() +" end");  
6 }
```



### 3. Thread States and Life Cycle



### 3. Thread States and Life Cycle

- At any time, a thread is said to be in one of several thread states

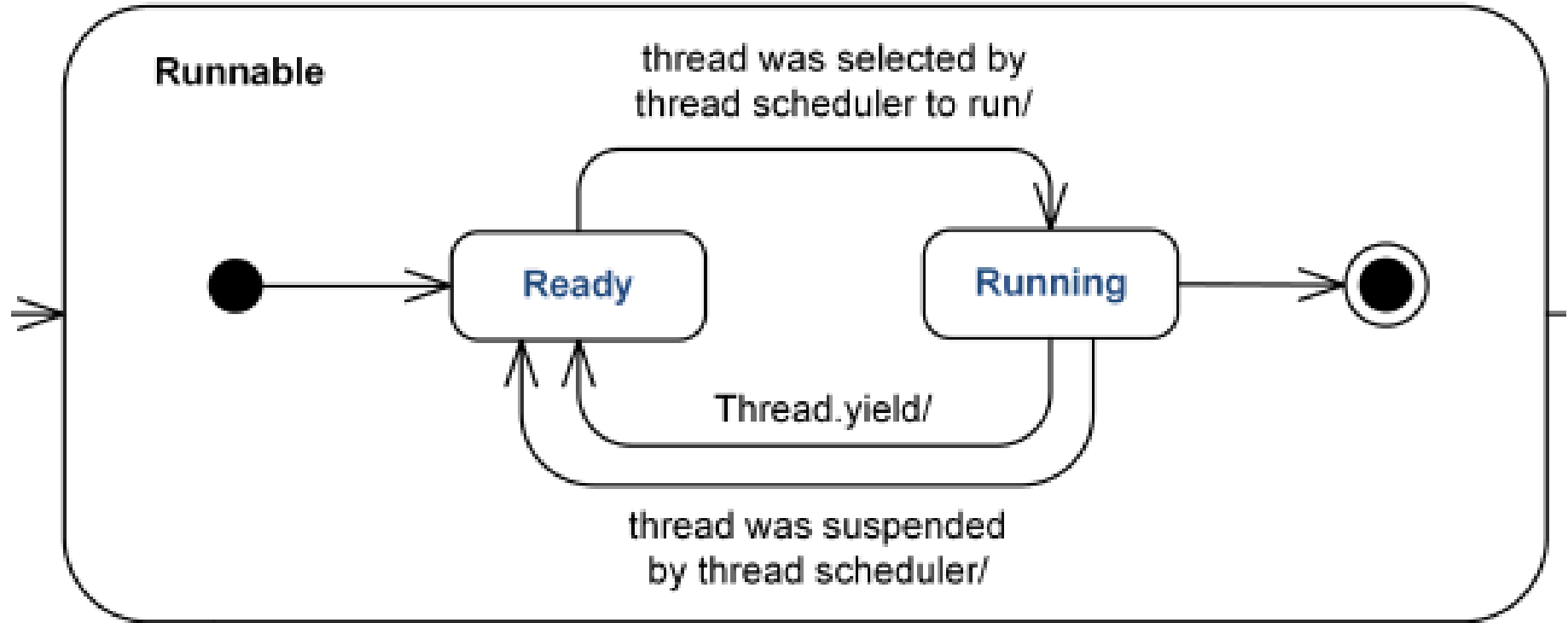
#### **New State:**

- When a thread has just been created, it is in the "New" state.
  - ✓ the Thread is not yet scheduled for execution and has not started running.

#### **“Runnable” State:**

- A thread enters the "Runnable" state after invoking the start() method.
- A thread in the runnable state is either
  - ✓ Executing its task or.
  - ✓ Is ready to run and waiting to be selected by thread scheduler to run.

### 3. Thread States and Life Cycle



# 3. Thread States and Life Cycle

## Waiting State

- A runnable thread can transition to the waiting state while it waits for another thread to perform a task.
- A waiting thread transitions back to the runnable state only when another thread notifies it to continue executing.

## Timed Waiting State

- A runnable thread can transition to the timed waiting state if it provides an optional wait interval when it's waiting for another thread to perform a task.
- It returns to the runnable state when it's notified by another thread or when the timed interval expires.

### 3. Thread States and Life Cycle

- Another way to place a thread in the timed waiting state is to put a runnable thread to sleep
  - ✓ Returns to the runnable state when the sleep interval expires.

#### **Blocked state**

- A runnable thread transitions to the blocked state when it attempts to perform a task that cannot be completed immediately.
- For example, when a thread issues an input/output request.
  - ✓ OS blocks the thread until that I/O request completes

#### **Terminated State**

- A runnable thread enters the terminated or dead state when it successfully completes its task or terminates due to an error.

## 4. Creating Threads with The Executor

- Runnable specify a task that execute concurrently with other tasks.
- An Executor object executes Runnables by Creating and managing a group of threads called a thread pool.

### Executor interface

- has method execute, which accepts a Runnable as an argument.
- Assigns every Runnable passed to its execute method to one of the available threads in the thread pool.
  - ✓ If there are no available threads, it creates a new thread or waits for a thread to become available.
- To execute a Runnable, it calls its run method.

## 4. Creating Threads with The Executor

### The `ExecutorService` interface

- extends `Executor` interface and declares various methods for managing the life cycle of an `Executor`.
- Its object can be obtained by calling one of the static methods declared in class `Executors`
  - ✓ `Executors` class provides several static methods that can be used to create and manage a thread pool.
- Advantages of using `Executor` over creating threads yourself.
  - ✓ `Executors` can reuse existing threads to eliminate the overhead of creating a new thread for each task
  - ✓ `Executor` improve performance by optimizing the number of threads

## 4. Creating Threads with The Executor

```
1 class MyThread3 implements Runnable{
2     String thrdName;
3     public MyThread3(String name){
4         thrdName = name;
5     }
6     public void run(){
7         System.out.println(thrdName + " starting");
8         try{
9             for(int i = 0; i<5 ; i++){
10                 System.out.println(thrdName + ", count: " +i);
11                 Thread.sleep(1000);
12             }
13         }catch(InterruptedException e){}
14     } }
```



## 4. Creating Threads with The Executor

```
1 public static void main(String args[]) {  
2     System.out.println(" Main start");  
3     ExecutorService es = Executors.newCachedThreadPool();  
4     MyThread3 task1 = new MyThread3("Task #1");  
5     MyThread3 task2 = new MyThread3("Task #2");  
6     MyThread3 task3 = new MyThread3("Task #3");  
7     es.execute(task1);  
8     es.execute(task2);  
9     es.execute(task3);  
10    System.out.println(" Main start");  
11 }
```

## 5. Creating Threads Using Lambda Expressions

- Lambda expressions provide a concise way to represent anonymous functions.
- Lambda expressions are used to implement methods defined by functional interfaces.
- A functional interface is an interface with a single abstract method
- Runnable is a functional interface, which has a single abstract method `void run()`
- Its general syntax:
  - ✓ (parameters) -> expression or
  - ✓ (parameters) -> { statements; }

## 5. Creating Threads Using Lambda Expressions

### ➤ Parameters:

- ✓ Can be explicitly typed or inferred by the compiler.
- ✓ Parentheses are optional if there's only one parameter and its type is inferred.
- ✓ Empty parentheses () are used for methods with no parameters.

### ➤ Lambda Body:

- ✓ Can be a single expression or a block of statements.
- ✓ If it's a single expression, the result is implicitly returned.
- ✓ If it's a block of statements, you must use curly braces {} and explicitly use the return keyword to return a value (if needed).

## 5. Creating Threads Using Lambda Expressions

```
public class LambdaExamples {  
    public static void main(String [] args){  
        Arthmtc add = (x, y) -> x + y;  
        Arthmtc mult = (x, y) -> x * y;  
        Comparator<String> comp = (a, b) -> a.length() - b.length();  
        System.out.println("sum :" + add.oprt(12, 6) + " and prod :" + mult.oprt(12, 6));  
        List<String> sl = Arrays.asList("hello", "hel", "hi");  
        Collections.sort(sl, comp);  
        System.out.println("the list is :" + sl);  
    }  
}  
  
interface Arthmtc { double oprt(double a, double b); }
```

## 5. Creating Threads Using Lambda Expressions

```
public class NewClass {  
    public static void main(String [] args){  
        Thread t1 = new Thread()->{  
            for(int i = 1; i <= 5; i++)  
                System.out.println(Thread.currentThread().getName() + " : " + i);  
        };  
        t1.start();  
        new Thread()->System.out.println(" Lambda expression")).start();  
    }  
}
```

## 5. Creating Threads Using Lambda Expressions

```
public class ThreadLE {  
    public static void main(String [] args){  
        ExecutorService es = Executors.newCachedThreadPool();  
        es.execute(()->{for (int i = 1; i <= 5; i++)  
            System.out.println("Hello : " + i);  
        });  
        es.execute(()->{for (int i = 1; i <= 5; i++)  
            System.out.println("Lambda Expression : " + i);  
        });  
    }  
}
```

# 6. Thread Properties

## Thread Priority

- Every Java thread has a thread priority that helps determine the order in which threads are scheduled.
- Each new thread inherits the priority of the thread that created it.
- You can set the priority of a thread by using `setPriority(int n)` method of `Thread` class.
- Higher-priority threads gets processor time before lower-priority threads.
  - ✓ Nevertheless, thread priorities cannot guarantee the order in which threads execute.

## 6. Thread Properties

### Starvation

- Is the indefinite postponement of the execution of threads by higher-priority threads.
- To prevent starvation, operating systems gradually increase the lower priority thread's priority so that it will eventually run.

### Deadlock

- Occurs when two threads wait for each other simultaneously (directly or indirectly) to proceed.



## 6. Thread Properties

### Daemon Threads

- Is a thread that has no other role in life than to serve others.
- Runs in the background and does not prevent the JVM from exiting when all non-daemon threads in Java have been completed.
- To turn a thread to a daemon thread, use `setDaemon()` method as:  
`t.setDaemon(true)`

### Thread Name

- Threads have default names
- You can also set any name to a thread with Thread's `setName` method.

## 6. Thread Properties

### Thread Join

- `join()` method can be used to pause the current thread execution until unless the specified thread is dead.
  - ✓ It puts the current thread on wait until the thread on which it's called is dead.
- `join(long millis)`: is used to wait for the thread on which it's called to be dead or wait for specified milliseconds.

### Yield() method

- causes the currently executing thread object to temporarily pause and allow other threads to execute.

## 6. Thread Properties

```
1  public static void main(String args[]) {  
2      System.out.println("Main start" );  
3      Thread mt1 = new Thread(new MyThread2());  
4      mt1.start();  
5      Thread mt2 = new Thread(new MyThread2(), "Adama");  
6      mt2.start();  
7      try{  
8          mt1.join();//main waits until thread mt1 is dead  
9          mt2.join(5000);//main waits until mt2 is dead or up to 5 seconds  
10     }catch(InterruptedException e){}  
11     System.out.println("Main end");  
12 }
```

## 7. Synchronization

- Synchronization is a mechanism for managing access to shared resources in a multithreaded environment.
- It ensures that only one thread can access a critical section of code at a time and prevents data inconsistency and race conditions.
  - ✓ **Race Conditions:** The final outcome of an operation depends on the unpredictable order in which the threads execute.
- There are two mechanisms for synchronization, synchronized keyword and ReentrantLock class.

### Synchronized keyword

- Every object in Java has an intrinsic lock
- A method can be declared with the synchronized keyword

## 7. Synchronization

```
synchronized r_type name(parameters) {  
    statements  
}
```

- ✓ To call a synchronized method, a thread must acquire the intrinsic object lock.
- ✓ The object's lock is held by a maximum of only one thread at any time.
- ✓ If the lock is currently held by another thread, the entering thread is blocked (put into a waiting state) until the lock becomes available.
- ✓ Once the lock is acquired, the thread can execute the synchronized method.

## 7. Synchronization

- We can declare static methods as synchronized.
  - If a static synchronized method is called, no other thread can call this or any other synchronized static method of the same class.
- We can also have a synchronized block declared as:  

```
synchronized (object) {  
    statements  
}
```
- Synchronized blocks allow you to synchronize specific sections of code within a method.
- All synchronized blocks that are supposed to protect the same data, must use the same lock object.
- If different objects are used, then the synchronization will not work as intended.

## 7. Synchronization

```
1 class MyArray{
2     private int[] myArr;
3     private int position = 0;
4     private SecureRandom sr = new SecureRandom();
5     public MyArray(int size){ myArr = new int[size]; }
6     public void add(int value){
7         int index = position;
8         try{ Thread.sleep(sr.nextInt(500));}
9         catch(InterruptedException e){}
10        myArr[index] = value;
11        System.out.println(Thread.currentThread().getName()+
12                             " wrote " + value + " at index " + index);
13        position++;
14    }
```

## 7. Synchronization

```
1 class MyThread implements Runnable{
2     MyArray myArr;
3     int startValue;
4     public MyThread(int value, MyArray arr){
5         myArr = arr;
6         startValue = value;
7     }
8     public void run(){
9         for(int i = startValue; i < startValue+3; i++)
10             myArr.add(i);
11     }
12 }
```



## 7. Synchronization

```
1 public class sync {  
2     public static void main(String args[]) {  
3         MyArray myArr = new MyArray(6);  
4         Thread mt1 = new Thread(new MyThread4(1, myArr));  
5         Thread mt2 = new Thread(new MyThread4(17, myArr));  
6         mt1.start();  
7         mt2.start();  
8     }  
9 }
```

## 7. Synchronization

- In the above program an object of MyArray class is shared among two threads
- MyArray enable these threads to put an int value to array.
- The threads have unsynchronized access to the array.
- i.e the two threads may access the array concurrently.
- A value which was written to the array by a thread may be overwritten later by the the other thread.
- There is unpredictability of thread scheduling and to increase the likelihood of producing erroneous output.

## 7. Synchronization

- The problem lies in method `add`, which stores **value** at index **position**, then increments position.
- If one thread obtains the position value, another thread may come along and increment position before the first thread to place a value in the array.
- `MyArray` allows any number of threads to read and modify shared data concurrently
- To ensure that no two threads can access its shared data at the same time. Make the access to array atomic operation.
- Atomicity can be achieved using the `synchronized` keyword.
- Make `add` method synchronized.

## 7. Synchronization

```
1 class MyArray{
2     private int[] myArr;
3     private int position = 0;
4     private SecureRandom sr = new SecureRandom();
5     public MyArray(int size){ myArr = new int[size]; }
6     public synchronized void add(int value){
7         int index = position;
8         try{ Thread.sleep(sr.nextInt(500));}
9         catch(InterruptedException e){}
10        myArr[index] = value;
11        System.out.println(Thread.currentThread().getName()+
12            " wrote " + value + " at index " + index);
13        position++;
```

# 7. Synchronization

## ReentrantLock class

- ReentrantLock provides a more flexible and powerful alternative to the synchronized keyword.
- A thread that already holds a ReentrantLock can acquire it again without blocking.
- A try finally block is used when using a Reentrant lock to ensure that the lock is always released, even if an exception is thrown.

```
ReentrantLock lock = new ReentrantLock();  
lock.lock();  
try{  
    //critical section of code.  
} finally {  
    lock.unlock();  
}
```

## 8. Thread Communication

- Java supports inter-thread communication with the `wait( )`, `notify( )`, and `notifyAll( )` methods.
  - ✓ These methods are implemented by `Object` class – are part of all objects
  - ✓ These methods should be called only from within a synchronized context.
- When a thread calls `wait( )`.
  - ✓ The thread goes to sleep and releases the monitor for that object.
    - ◆ Allowing another thread to use the object.
  - ✓ At a later point, the sleeping thread is awakened when some other thread enters the same monitor and calls `notify( )`, or `notifyAll( )`.

## 8. Thread Communication

- A call to `notify( )` resumes one waiting thread.
- A call to `notifyAll( )` notifies all threads.
  - ✓ the scheduler determining which thread gains access to the object.
- waiting thread could be awakened due to a spurious wakeup.
  - ✓ calls to `wait( )` should take place within a loop that checks the condition on which the thread is waiting.

```
while(condition){  
    try{  
        wait();  
    }catch(InterruptedException e){}  
}
```

## 8. Thread Communication

```
class TickTock{  
    String state = "Tick"; //initial state is Tick  
    synchronized void tick(){ //prints Tick  
        while(state.compareTo("Tock")== 0){ //wait while state is Tock  
            try{  
                wait();  
            }catch(InterruptedException e){}  
        }  
        System.out.print("Tick "); //print Tick  
        state = "Tock"; //set state to Tock  
        Notify(); //notify the other thread  
    }  
}
```



## 8. Thread Communication

```
synchronized void tock(){ //prints Tock  
    while(state.compareTo("Tick") == 0){ //wait while state is Tick  
        try{  
            wait();  
        }catch(InterruptedException e){}  
    }  
    System.out.println("Tock "); //print Tock  
    state = "Tick"; //set state to Tick  
    Notify(); //notify the other thread  
}
```

## 8. Thread Communication

```
class ThreadTT implements Runnable{
    TickTock tt;
    ThreadTT(TickTock tt){
        this.tt = tt;
    }
    public void run(){
        if(Thread.currentThread().getName().compareTo("ticker") == 0)
            for(int i = 0; i < 5; i++)
                tt.tick();
        else if(Thread.currentThread().getName().compareTo("tocker")==0)
            for(int i = 0; i < 5; i++)
                tt.tock();
    }
}
```

## 8. Thread Communication

```
public class wait {  
    public static void main(String args[]) {  
        TickTock tt = new TickTock();  
        ThreadTT tick = new ThreadTT(tt);  
        ThreadTT tock = new ThreadTT(tt);  
        Thread ticker = new Thread(tick, "ticker");  
        Thread tocker = new Thread(tock, "tocker");  
        ticker.start();  
        tocker.start();  
    }  
}
```

## 9. Thread-safe Collections

- A thread-safe class is a class that always maintains valid state even when used concurrently by multiple threads.
- Thread-safe collections are designed to be thread-safe.
  - ✓ They can be safely used by multiple threads without the need for external synchronization.
- A thread-safe collection can be synchronized collection or a concurrent collection.
- Concurrent collections achieve thread-safety by partitioning the data into segments.
  - ✓ Threads can access these segments concurrently and obtain locks only on the segments that are used.

## 9. Thread-safe Collections

- Synchronized collection locks the entire collection via intrinsic locking
  - ✓ Has low performance than Concurrent Collection
    - ◆ At a time only one thread is allowed to operate on an object so it increases the waiting time of the threads.
- Java 1.0 legacy classes like HashTable, Vector, and Stack are synchronized collections
- Java 1.2 collections are unsynchronized.
  - ✓ The Collections class provides static methods for wrapping collections as synchronized versions.
  - ✓ These are wrappers that returns a thread-safe collection backed up by the specified Collection.

## 9. Thread-safe Collections

```
List<Integer> syncList = Collections.synchronizedList(new ArrayList<>());
```

```
Map<Integer, String> syncMap = Collections.synchronizedMap(new  
HashMap<>());
```

```
Set<Integer> syncSet = Collections.synchronizedSet(new HashSet<>());
```

- There are concurrent collections in `java.util.concurrent` package.
  - ✓ Allow multiple threads to access and modify a collection concurrently, without the need for explicit synchronization.
  - ✓ Provide thread-safe implementations of the traditional collection interfaces like `List`, `Set`, and `Map`.

## 9. Thread-safe Collections

- Examples of concurrent collection classes include
  - ✓ `ArrayBlockingQueue`
    - ◆ Implements `BlockingQueue` interface
    - ◆ A fixed-size queue that supports the producer/consumer relationship
    - ◆ Has methods `put` and `take`
    - ◆ `put` places an element at the end of the `BlockingQueue`, waiting if the queue is full.
    - ◆ `take` removes an element from the head of the `BlockingQueue`, waiting if the queue is empty.
  - ✓ `ConcurrentHashMap`
    - ◆ A hash-based map (similar to the `HashMap`)

## 9. Thread-safe Collections

```
class MyBuffer{
    ArrayBlockingQueue<Integer> queue; //is thread-safe, no need of
    MyBuffer(){                          //synchronization
        queue = new ArrayBlockingQueue<>(1); //size 1
    }
    void myPut(int value) throws InterruptedException{
        queue.put(value); //put value to array blocking queue
        System.out.println(Thread.currentThread().getName()+" writes " +value);
    }
    Integer myTake() throws InterruptedException{
        int n = queue.take(); //get from array blocking queue
        System.out.println(Thread.currentThread().getName()+" reads " +n);
        return n;
    }
}
```



## 9. Thread-safe Collections

```
class Producer implements Runnable{  
    MyBuffer mb;  
    String thrdName;  
    Producer(MyBuffer mb, String name){  
        this.mb =mb;  
        thrdName = name;  
    }  
    public void run(){  
        Thread.currentThread().setName(thrdName);  
        for(int i=1; i<=5; i++)  
            try{  
                mb.myPut(i);  
            }catch(InterruptedException e){}  
    }  
}
```

## 9. Thread-safe Collections

```
class Consumer implements Runnable{  
    MyBuffer mb;  
    String thrdName;  
    Consumer(MyBuffer mb, String name){  
        this.mb =mb;  
        thrdName = name;  
    }  
    public void run(){  
        Thread.currentThread().setName(thrdName);  
        for(int i=1; i<=5; i++)  
            try{  
                mb.myTake();  
            }catch(InterruptedException e){}  
    }  
}
```

## 9. Thread-safe Collections

```
public class ConcurrentCollection {  
    public static void main(String args[]) {  
        MyBuffer mb = new MyBuffer();  
        ExecutorService es = Executors.newCachedThreadPool();  
        es.execute(new Consumer(mb, "Consumer 1"));  
        es.execute(new Producer(mb, "Producer 1"));  
        es.execute(new Consumer(mb, "Consumer 2"));  
        es.execute(new Producer(mb, "Producer 2"));  
        es.shutdown();  
    }  
}
```