Advanced Programming Methods

Lecture 5-6-7 - Concurrency in Java

Overview

- Introduction
- Java threads
- Java.util.concurrent

References

NOTE: The slides are based on the following free tutorials. You may want to consult them too.

1.http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/

2.http://tutorials.jenkov.com/java-util-concurrent/index.html

3. http://www.javacodegeeks.com/2015/09/java-concurrency-essentials.html

4. Oracle tutorials

Concurrent Programming

- there are two basic units of execution: processes and threads
- a computer system normally has many active processes and threads –even for only one execution core
- processing time for a single core is shared among processes and threads through an OS feature called time slicing.
- more and more common for computer systems to have multiple processors or processors with multiple execution cores

Processes

- a process has a self-contained execution environment.

- in general it has a complete, private set of basic run-time resources; for example, each process has its own memory space.

- most implementations of the Java virtual machine run as a single process

Processes

- processes are fully isolated from each other

-to facilitate communication between processes, most operating systems support Inter Process Communication (IPC) resources, such as pipes and sockets. (for communication between processes either on the same system or on different systems)

Threads

- are called lightweight processes

- creating a new thread requires fewer resources than creating a new process.

-threads exist within a process — every process has at least one.

-threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Threads

- Threads are the units that are scheduled by the system for executing on the processor

- On a single processor, each thread has its turn by multiplexing based on time

- On a multiple processor, each thread is running at the same time with each processor/core running a particular thread.

Threads

- a thread is a particular execution path of a process.
- one allows multiple threads to read and write the same memory (no process can directly access the memory of another process).
- when one thread modifies a process resource, the change is immediately visible to sibling threads.

Advantages of Multi-threading

- faster on a multi-CPU system

- even in a single CPU system, application can remain responsive by using worker thread runs concurrently with the main thread

Cost of Multi-threading

- program overhead and additional complexity

- there are time and resource costs in both creating and destroying threads

- the time required for scheduling threads, loading them onto the process, and storing their states after each time slice is pure overhead.

Cost of Multi-threading

- biggest cost: Since the threads in a process all share the same resources and heap, it adds additional programming complexity to ensure that they are not ruining each other's work.

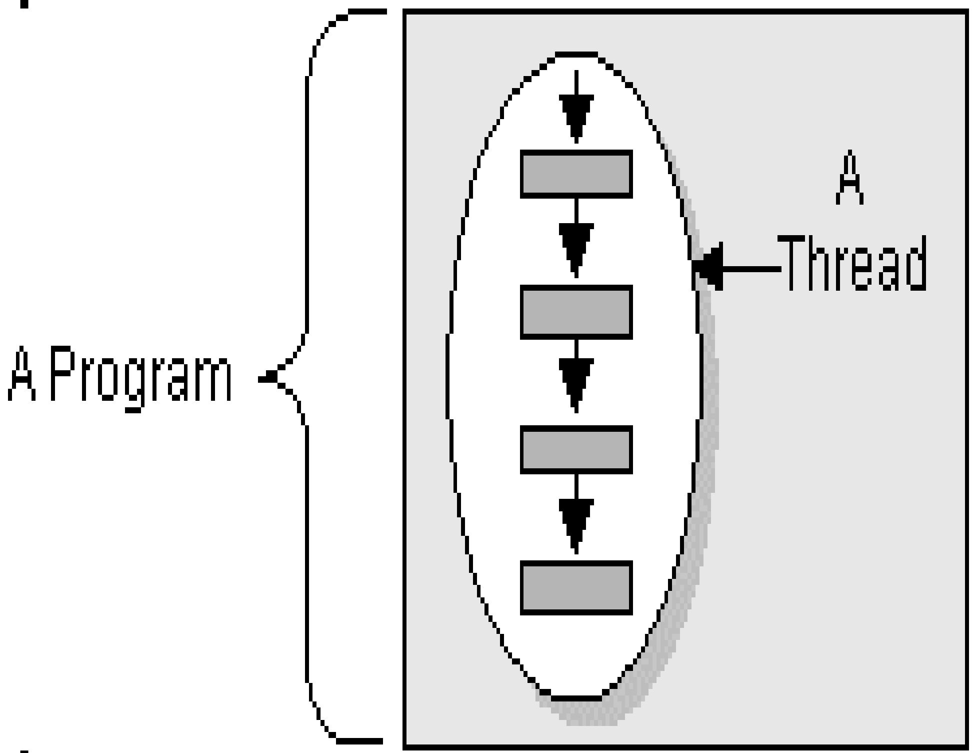
- debugging multithreaded programs can be quite difficult: the timing on each run of the program can be different; reproducing the same scheduling results is difficult

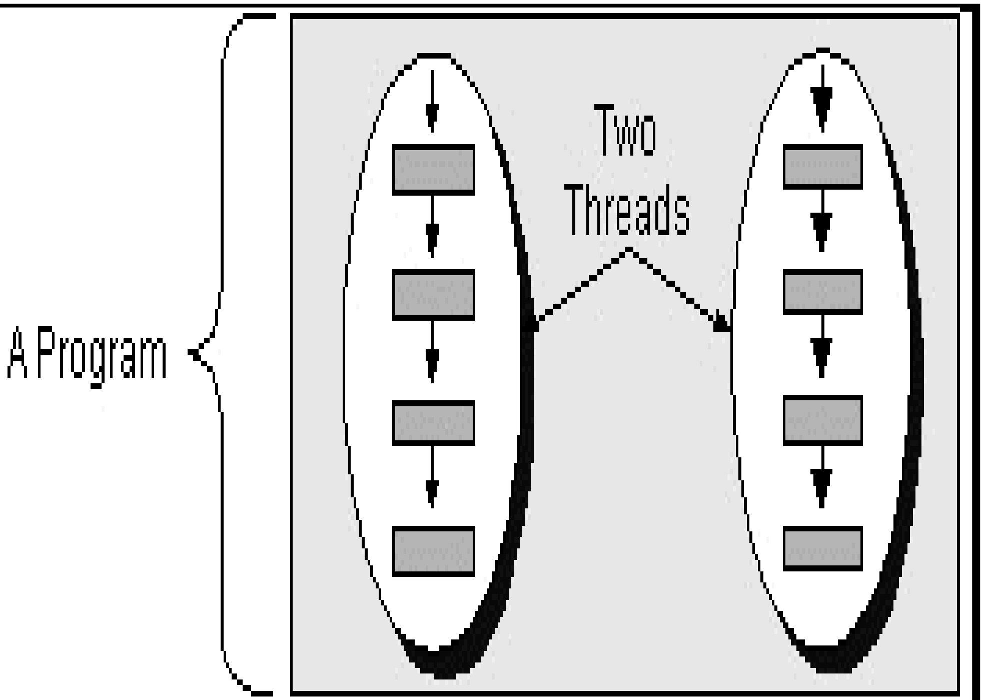
Contents

What is a thread ?

- A sequential (or single-threaded) program is one that, when executed, has only one single flow of control.
  - i.e., at any time instant, there is at most only one instruction (or statement or execution point) that is being executed in the program.
- A multi-threaded program is one that can have multiple flows of control when executed.
  - At some time instance, there may exist multiple instructions (or execution points) that are being executed in the program
  - Ex: in a Web browser we may do the following tasks at the same time:
  - 1. scroll a page,
  - 2. download an applet or image,
  - 3. play sound,
  - 4 print a page.
- A thread is a single sequential flow of control within a program.

single-threaded vs multithreaded programs

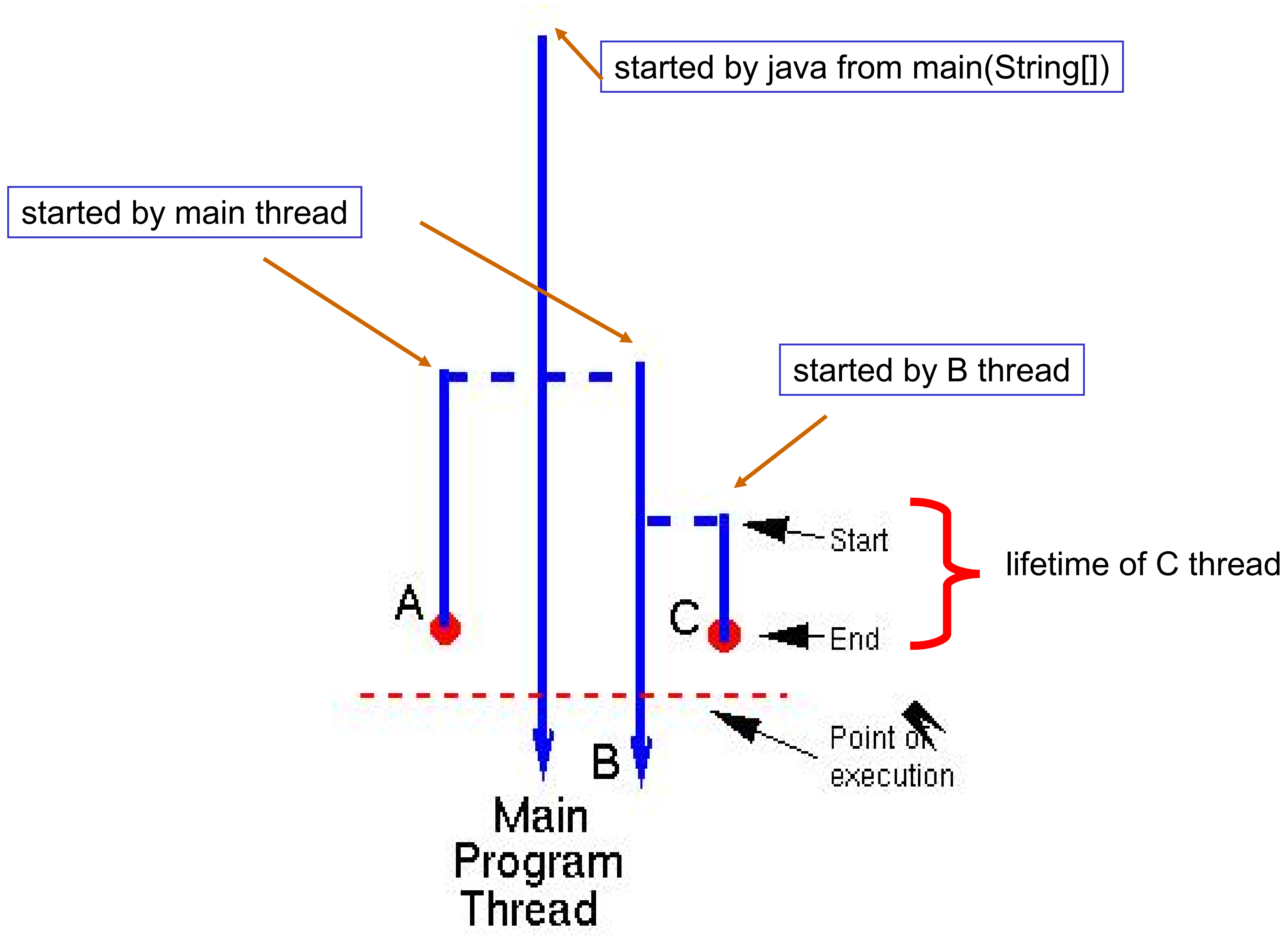




```
{ A(); A1();  A2();   A3();
  B1();  B2();  }
```

```
{  A();
   newThreads {
      { A1(); A2(); A3() };
      {B1(); B2() }
   }
}
```

Thread ecology in a java program



2. Define and launch a java thread

- Each Java Run time thread is encapsulated in a java.lang.Thread instance.
- Two ways to define a thread:
  1. Extend the Thread class
  2. Implement the Runnable interface :

     ```
     package java.lang;
     public interface Runnable {  public void run() ; }
     ```
- Steps for extending the Thread class:
  - Subclass the Thread class;
  - Override the default Thread method run(), which is the entry point of the thread, like the main(String[]) method in a java program.

Define a thread

```
// Example:
public class Print2Console extends Thread {
        public void run() {  // run() is to a thread what main() is to a java program
          for (int b = -128; b < 128; b++)  out.println(b);  }
        … // additional methods, fields …
         }
```

Implement the Runnable interface if you need a parent class:

```
// by extending JTextArea we can reuse all existing code of JTextArea
public class Print2GUI extend JTextArea implement Runnable {
        public void run() {
          for (int b = -128; b < 128; b++)  append( Integer.toString(b) + “\n” );  }
         }
```

How to launch a thread

1. create an instance of [a subclass of] Thread, say thread.
   - Thread thread = new Print2Console();
   - Thread thread = new Thread(new Print2GUI(..));

2. call its start() method, thread.start();. // note: not call run() !!

Ex:

- Printer2Console t1 = new Print2Console(); // t1 is a thread instance !
- t1.start() ; // this will start a new thread, which begins its execution by calling t1.run()
- … // parent thread continue immediately here without waiting for the child thread to complete its execution. cf: t1.run();
- Print2GUI jtext = new Print2GUI();
- Thread t2 = new Thread(jtext);
- t2.start();
- …

The java.lang.Thread constructors

```
// Public Constructors
Thread([ ThreadGroup group,] [ Runnable target, ]
                [ String name ] );
        Instances :
        Thread();
        Thread(Runnable target);
        Thread(Runnable target, String name);
        Thread(String name);
        Thread(ThreadGroup group, Runnable target);
        Thread(ThreadGroup group, Runnable target, String name);
        Thread(ThreadGroup group, String name);
//  name is a string used to identify the thread instance
//  group is the thread group to which this thread belongs.
```

Some thread property access methods

- int getID() // every thread has a unique ID, since jdk1.5
- String getName(); setName(String)
  - // get/set the name of the thread
- ThreadGroup getThreadGroup();
- int getPriority() ; setPriority(int) // thread has priority in [0, 31]
- Thread.State getState() // return current state of this thread

- boolean isAlive()
  - Tests if this thread has been started and has not yet died. .
- boolean isDaemon()
  - Tests if this thread is a daemon thread.
- boolean isInterrupted()
  - Tests whether this thread has been interrupted.

State methods for current thread accesses

- static Thread currentThread()
  - Returns a reference to the currently executing thread object.
- static boolean holdsLock(Object obj)
  - Returns true if and only if the current thread holds the monitor lock on the specified object.
- static boolean interrupted()
  - Tests whether the current thread has been interrupted.
- static void sleep([long millis [, int nanos]])
  - Causes the currently executing thread to sleep (cease execution) for the specified time.
- static void yield()
  - Causes the currently executing thread object to temporarily pause and allow other threads to execute.

An example

```
public 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 { // at this point, current thread is ‘this’.
                    Thread.sleep((long)(Math.random() * 1000));
                } catch (InterruptedException e) {}
            }
            System.out.println("DONE! " + getName());
        }
}
```

main program

```
public class TwoThreadsTest {
        public static void main (String[] args) {
          new SimpleThread(“Thread1").start();
          new SimpleThread(“Thread2").start();        } }
```

possible output:

```
0 Thread1      5 Thread1      DONE! Thread2
0 Thread2      5 Thread2      9 Thread1
1 Thread2      6 Thread2       DONE! Thread1
1 Thread1      6 Thread1
2 Thread1      7 Thread1
2 Thread2      7 Thread2
3 Thread2      8 Thread2
3 Thread1      9 Thread2
4 Thread1      8 Thread1
4 Thread2
```

3. The Life Cycle of a Java Thread

New → (Runnable → blocked/waiting) * → Runnable → dead(terminated)

sleep(long ms [,int ns])

// sleep (ms + ns x 10^{-3}) milliseconds and then continue

[IO] blocked by synchronized method/block

```
synchronized( obj ) { …  }      // synchronized statement
synchronized m(… ) { …  }     // synchronized method
```

// return to runnable if IO complete

obj.wait()

// return to runnable by obj.notify() or obj.notifyAll()

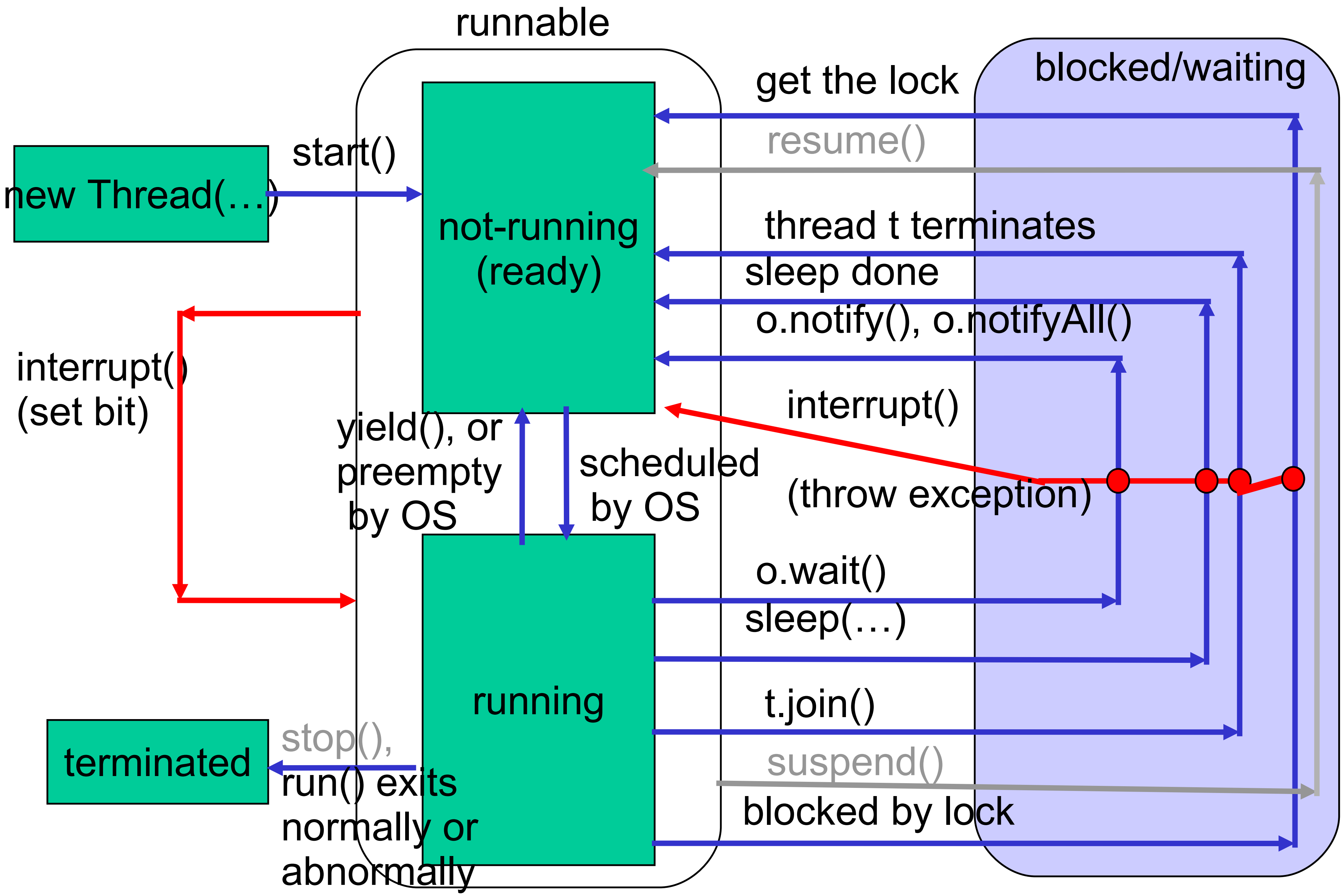
join(long ms [,int ns])

// Waits at most ms milliseconds plus ns nanoseconds for this thread to die.

3. The states(life cyccle) of a thread (java 1.5)

```
public class Thread { .. // enum in 1.5  is a special class for finite type.
 public static enum State { //use Thread.State for referring to this nested class
  NEW,   //   after new Thread(), but before start().
  RUNNABLE, // after start(), when running or ready
  BLOCKED,   //  blocked by monitor lock
                          // blocked by a synchronized method/block
  WAITING,     // waiting for to be notified; no time out set
                       // wait(), join()
 TIMED_WAITING, // waiting for to be notified; time out set
                            // sleep(time), wait(time), join(time)
TERMINATED  // complete execution or after stop()
} …
}
```

The life cycle of a Java thread



State transition methods for Thread

- public synchronized native void start() {
  - start a thread by calling its run() method …
  - It is illegal to start a thread more than once }
- public final void join([long ms [, int ns]]);
  - Let current thread wait for receiver thread to die for at most ms+ns time
- static void yield() // callable by current thread only
  - Causes the currently executing thread object to temporarily pause and allow other threads to execute.
- public final void resume(); // deprecated
- public final void suspend();// deprecated→may lead to deadlock
- public final void stop(); // deprecated → lead to inconsistency

// state checking

- public boolean isAlive() ; // true if runnable or blocked

Note: When we call t.join(), we in fact use current thread's time to execute code of t thread

4. interrupting threads

- A blocking/waiting call (sleep(),wait() or join()) to a thread t can be terminated by an InterruptedException thrown by invoking t.interrupt().
  - this provides an alternative way to leave the blocked state.
  - however, the control flow is different from the normal case.

```
Ex: public void run() {
try { …  while (more work to do) {      // Normal sleep() exit continue here

                 do some work;
                 sleep( … ); // give another thread a chance to work
               }
     }
 catch (InterruptedException e) { // if waked-up by interrupt() then continue here
     …  // thread interrupted during sleep or wait      }
}
```

- Note: the interrupt() method will not throw an InterruptedException if the thread is not blocked/waiting. In such case the thread needs to call the static interrupted() method to find out if it was recently interrupted. So we should rewrite the while loop by

```
while ( ! interrupted()  && moreWorkToDo() ) { … }
```

interrupt-related methods

- void interrupt()
  - send an Interrupt request to a thread.
  - the “interrupted” status of the thread is set to true.
  - if the thread is blocked by sleep(), wait() or join(), the The *interrupted status* of the thread is cleared and an InterruptedException is thrown.
  - conclusion: runnable ==> “interrupted” bit set but no Exception thrown.
  - not runnable ==> Exception thrown but “interrupted” bit not set
- static boolean interrupted() // destructive query
  - Tests whether the current thread (self) has been interrupted.
  - reset the “interrupted” status to false.
- boolean isInterrupted() // non-destructive query
  - Tests whether this thread has been interrupted without changing the “interrupted” status.
  - may be used to query current executing thread or another non-executing thread. e.g. if(t1.isInterrupted() | Thread.currentThread()...)

A complete example

- it consists of two threads.
- the first thread (main):
  - it is the main thread that every Java application has.
  - it creates a new thread from the Runnable object, MessageLoop, and waits for it to finish.
  - if the MessageLoop thread takes too long to finish, the main thread interrupts it.
- the second thread (MessageLoop):
  - it prints out a series of messages.
  - if interrupted before it has printed all its messages, it prints a message and exits.

```
public class SimpleThreads {
public static void main(String args[]) throws InterruptedException {
    // Delay, in milliseconds before  we interrupt MessageLoop thread
    long patience = 1000 * 60 * 60;
    threadMessage("Starting MessageLoop thread");
    long startTime = System.currentTimeMillis();
    Thread t = new Thread(new MessageLoop());
    t.start();
    threadMessage("Waiting for MessageLoop thread to finish");
    // loop until MessageLoop thread exits
    while (t.isAlive()) {
      threadMessage("Still waiting...");
      // Wait maximum of 1 second for MessageLoop thread to finish.
      t.join(1000);
if (((System.currentTimeMillis() - startTime) > patience) && t.isAlive()) {
            threadMessage("Tired of waiting!");
            t.interrupt();
            // Shouldn't be long now -- wait indefinitely
            t.join();
         }
      }
      threadMessage("Finally!");
   }
// Display a message, preceded by the name of the current thread
   static void threadMessage(String message) {
      String threadName = Thread.currentThread().getName();
      System.out.format("%s: %s%n", threadName, message);
   }
```

```
private static class MessageLoop implements Runnable {
    public void run() {
        String importantInfo[] = { "A","B","C","D"};
        try {
            for (int i = 0; i < importantInfo.length; i++) {
                // Pause for 4 seconds
                Thread.sleep(4000);
                // Print a message
                threadMessage(importantInfo[i]);
            }
        } catch (InterruptedException e) {
            threadMessage("I wasn't done!");
  }}}
```

5. Thread synchronization

- Problem with any multithreaded Java program :
  - Two or more Thread objects access the same pieces of data.
- too little or no synchronization ==> there is inconsistency, loss or corruption of data.
- too much synchronization ==> deadlock or system frozen.
- In between there is unfair processing where several threads can starve another one hogging all resources between themselves.

Multithreading may incur inconsistency : an Example

Two concurrent deposits of 50 into an account with 0 initial balance.:

```
void deposit(int amount) {
  int x = account.getBalance();
  x += amount;
  account.setBalance(x);  }
```

```
deposit(50) :  // deposit 1
 x = account.getBalance()  //1
 x += 50; //2
account.setBalance(x) //3
```

```
• deposit(50) : // deposit 2
  x = account.getBalance()  //4
  x += 50; //5
 account.setBalance(x) //6
```

The execution sequence:1,4,2,5,3,6 will result in unwanted result !!

Final balance is 50 instead of 100!!

Synchronized methods and statements

- multithreading can lead to racing hazards where different orders of interleaving produce different results of computation.
  - Order of interleaving is generally unpredictable and is not determined by the programmer.
- Java's synchronized method (as well as synchronized statement) can prevent its body from being interleaved by relevant methods.
  - synchronized(obj) { … } // synchronized statement with obj as lock
  - synchronized … m(…) {… } //synchronized method with this as lock
  - When one thread executes (the body of) a synchronized method/statement, all other threads are excluded from executing any synchronized method with the same object as lock.

Synchronizing threads

- Java use the monitor concept to achieve mutual exclusion and synchronization between threads.
- Synchronized methods /statements guarantee mutual exclusion.
  - Mutual exclusion may cause a thread to be unable to complete its task. So monitor allow a thread to wait until state change and then continue its work.
- wait(), notify() and notifyAll() control the synchronization of threads.
  - Allow one thread to wait for a condition (logical state) and another to set it and then notify waiting threads.
  - condition variables => instance boolean variables
  - wait => wait();
  - notifying => notify(); notifyAll();

Typical usage

```
synchronized void doWhenCondition() {
   while ( !condition )
     wait(); // wait until someone notifies us of changes in condition
   … // do what needs to be done when condition is true
}
synchronized void changeCondition {
   // change some values used in condition test
   notify(); // Let waiting threads know something changed
}
```

Note: A method may serve both roles; it may need some condition to occur to do something and its action my cause condition to change.

Java's Monitor Model

A monitor is a collection of code (called the critical section) associated with an object (called the lock)

At any time instant only one thread at most can has its execution point located in the critical section associated with the lock(mutual exclusion).

Java allows any object to be the lock of a monitor.

Java's Monitor Model(cont.)

The critical section of a monitor controlled by an object e [of class C] comprises the following sections of code:

The body of all synchronized methods m() callable by e, that is, all synchronized methods m(…) defined in C or super classes of C.

The body of all synchronized statements with e as target:

synchronized(e) { …   }. // critical section is determined by the lock object e

Java's Monitor Model(cont.)

A thread enters the critical section of a monitor by invoking e.m() or executing a synchronized statement.

Before it can run the method/statement, it must first own the lock e and will need to wait until the lock is free if it cannot get the lock.

A thread owing a lock will release the lock automatically once it exit the critical section.

Java's Monitor model (continued)

A thread executing in a monitor may encounter condition in which it cannot continue but still does not want to exit.

In such case, it can call the method e.wait() to enter the waiting list of the monitor.

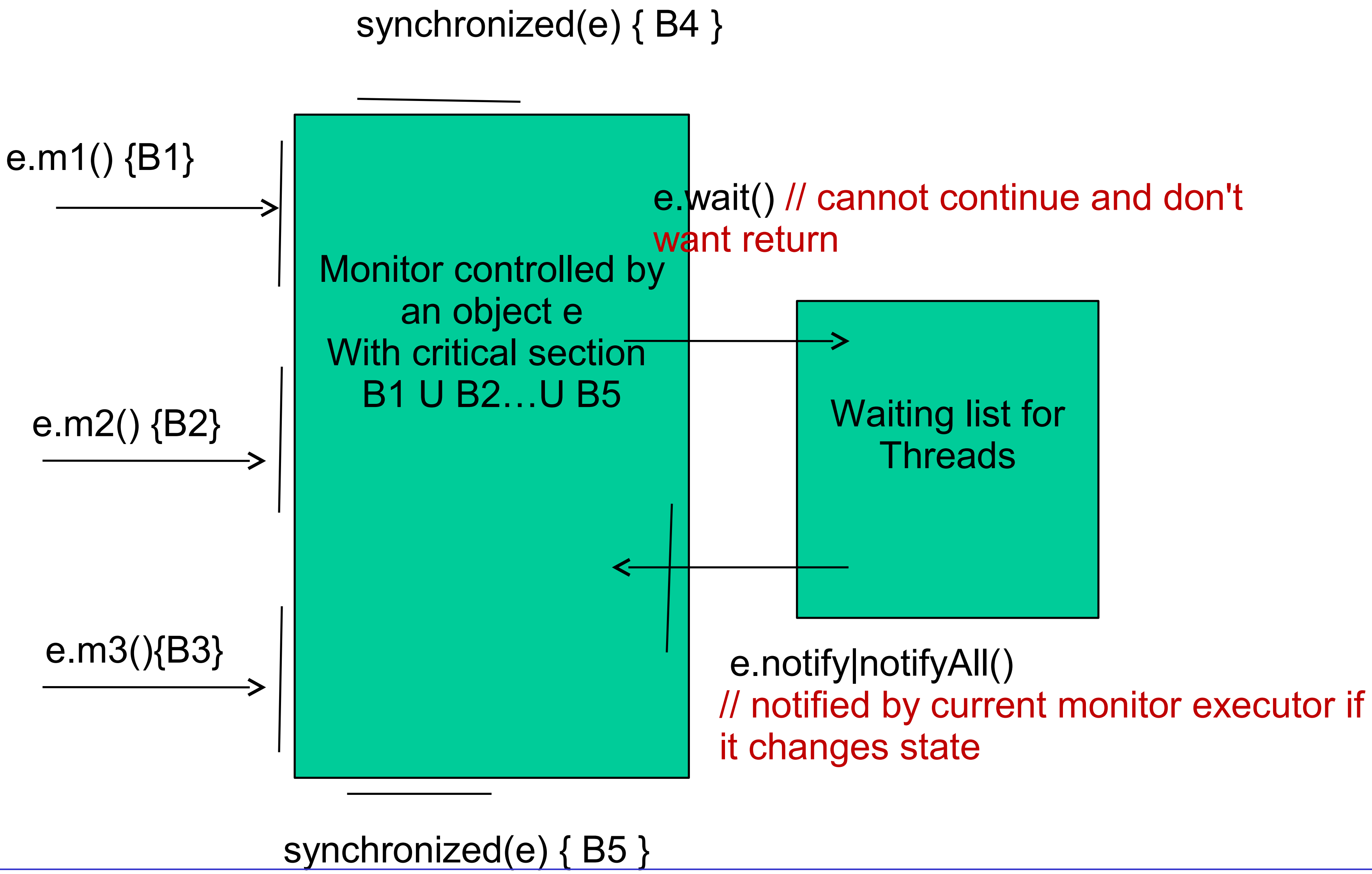
A thread entering waiting list will release the lock so that other outside threads have chance to get the lock.

Java's Monitor model (continued)

A thread changing the monitor state should call e.notify() or e.notifyAll() to have one or all threads in the waiting list to compete with other outside threads for getting the lock to continue execution.

Note: A static method m() in class C can also be synchronized. In such case it belongs to the monitor whose lock object is C.class.

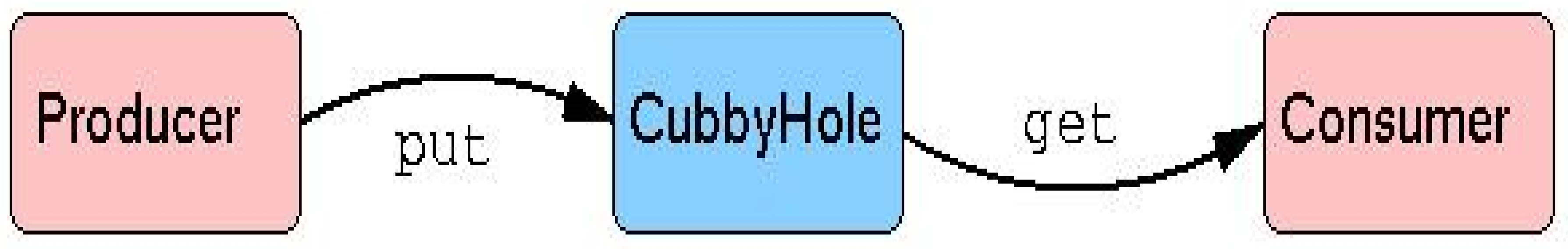
Java's monitor model (continued)


synchronized(e) { B4 }
e.m1() {B1}
e.wait() // cannot continue and don't want return
Monitor controlled by an object e With critical section B1 U B2…U B5
Waiting list for Threads
e.m2() {B2}
e.m3(){B3}
e.notify|notifyAll()
// notified by current monitor executor if it changes state
synchronized(e) { B5 }

- Note since a section of code may belong to multiple monitors, it is possible that two threads reside at the same code region belonging to two different monitors..

Producer/Consumer Problem

- Two threads: producer and consumer
- One monitor: CubbyHole
- The Producer :
  - generates a pair of integers between 0 and 9 (inclusive), stores it in a CubbyHole object, and prints the sum of each generated pair.
  - sleeps for a random amount of time between 0 and 100 milliseconds before repeating the number generating cycle.
- The Consumer:
  - consumes all pairs of integers from the CubbyHole as quickly as they become available.



Producer.java

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;          private int id;
    public Producer(CubbyHole c, int id) {
    cubbyhole = c;          this.id = id;          }
    public void run() {
        for (int i = 0; i < 10; i++)
          for(int j =0; j < 10; j++ ) {
             cubbyhole.put(i, j);
             System.out.println("Producer #" + this.id   + " put: ("+i +","+j + ").");
              try { sleep((int)(Math.random() * 100));  }
              catch (InterruptedException e) { }
           }
      }
  }
```

Consumer.java

```
public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int id;

    public Consumer(CubbyHole c, int id) {
        cubbyhole = c;          this.id = id;    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.id + " got: " + value);
        }
     }
}
```

CubbyHole without mutual exclusion

```
public class CubbyHole {    private int x,y;
    public synchronized int get() {  return x+y;    }
    public synchronized void put(int i, int j) {x= i; y = j }    }
```

Problem : data inconsistency for some possible execution sequence

Suppose after put(1,9) the data is correct , i.e., (x,y) = (1,9)

And then two method calls get() and put(2,0) try to access CubbyHole concurrently => possible inconsistent result:

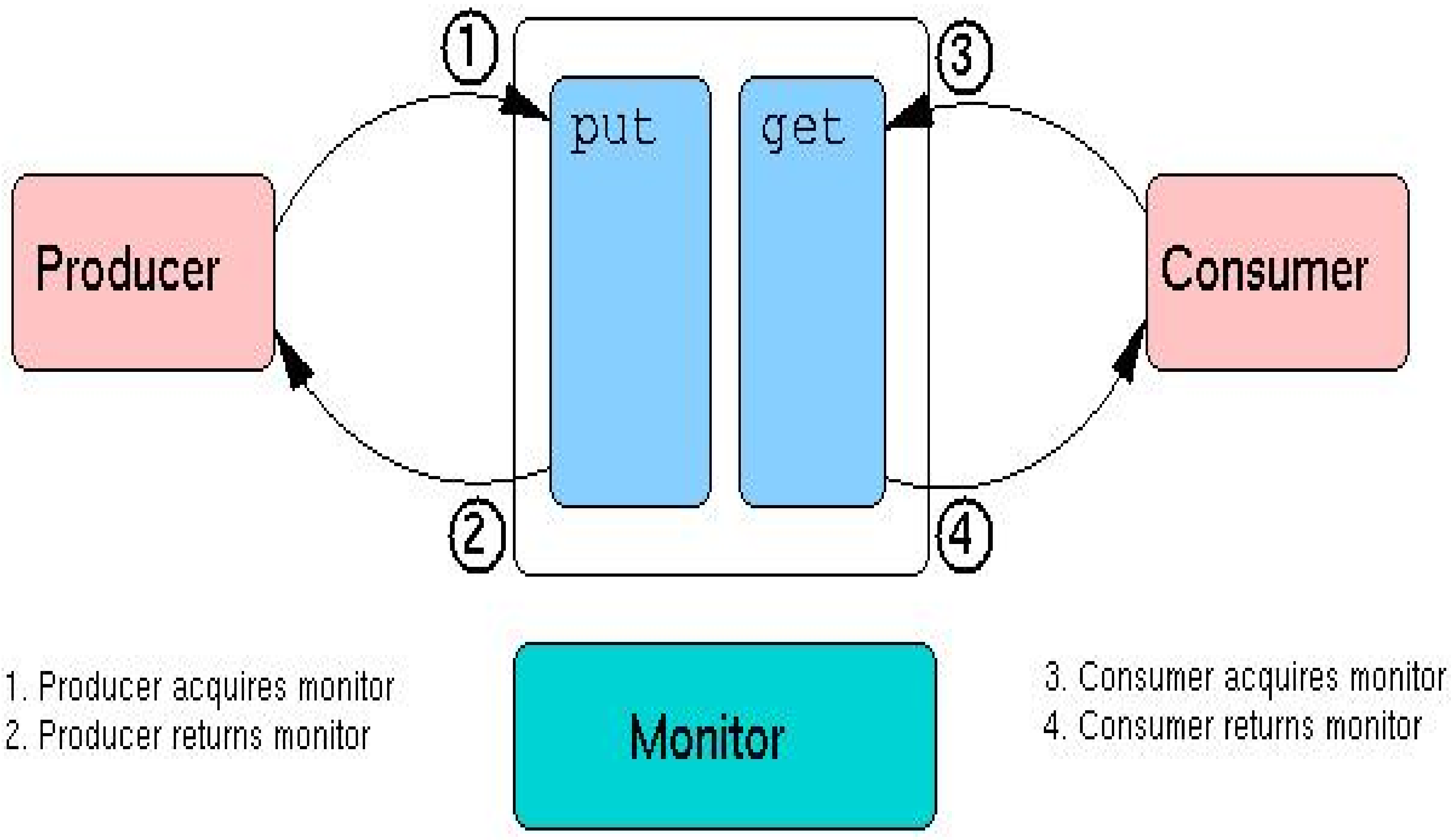
(1,9) → get() { return x + y ; } → { return 1 + y ; }

(1,9) → put(2,0) {x = 2; y = 0;} → (x,y) = (2,0)

(2,0) → get() { return 1 + y ;} → return 1 + 0 = return 1 (instead of 10!)

By marking get() and put() as synchronized method, the inconsistent result cannot occur since, by definition, when either method is in execution by one thread, no other thread can execute any synchronized method with this CubbyHole object as lock.

The CubbyHole



1. Producer acquires monitor
2. Producer returns monitor

3. Consumer acquires monitor
4. Consumer returns monitor

CubbyHole without synchronization

```
public class CubbyHole {
    private int x,y;
    public synchronized int get() {  return x+y;    }
    public synchronized void put(int i, int j) { x= i ; y  = j; }
}
```

Problems:

Consumer quicker than Producer : some data got more than once.

Producer quicker than Consumer: some put data not used by consumer.

```
ex:  Producer #1   put:  (0,4)
     Consumer #1 got: 4
     Consumer #1 got: 4
     Producer #1   put: (0,5)
```

```
Consumer #1  got: 3
Producer #1    put: (0,4)
Producer #1    put: (0,5)
Consumer #1  got: 5
```

Another CubbyHole implementation (still incorrect!)

```
pubic class CubbyHole {  int x,y;  boolean available = false;
   public synchronized int get() {    // won't work!
          if (available == true) {
               available = false;   return x+y;
          }  }   // compilation error!! must return a value in any case!!
   public synchronized void put(int a, int b) {    // won't work!
          if (available == false) {
               available = true;     x=a;y=b;
        } }} // but how about the case that available == true ?
put(..); get(); get();  // 2nd get()  must return something!
put(..);put(..); // 2nd put() has no effect!
```

CubbyHole.java

```
public class CubbyHole {
    private int x,y;      private boolean available = false; // condition var
    public synchronized int get() {
        while (available == false) {
            try {    this.wait();      } catch (InterruptedException e) { }       }
        available = false;  // enforce consumers to wait again.
        notifyAll(); // notify all producer/consumer to compete for execution!
                      // use notify() if just wanting to wakeup one waiting thread!
        return x+y;     }
    public synchronized void put(int a, int b) {
        while (available == true) {
            try {  wait();   } catch (InterruptedException e) { }       }
        x= a; y = b;
        available = true;  // wake up waiting consumer/producer to continue
        notifyAll(); // or notify();    }
}
```

The main class

```
public class ProducerConsumerTest {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);
        p1.start();
        c1.start();
    }       }
```

Other issues

- Thread priorities
  - public final int getPriority();
  - public final void setPriority();
  - get/set priority between MIN_PRIORITY and MAX_PRIORITY
  - default priority : NORMAL_PRIORITY
- Daemon threads:
  - isDaemon(), setDaemon(boolean)
  - A Daemon thread is one that exists for service of other threads.
  - The JVM exits if all threads in it are Daemon threads.
  - setDaemon(.) must be called before the thread is started.
- **public static boolean holdsLock(Object obj)**
  - check if this thread holds the lock on obj.
  - ex: synchronized(e) { Thread.holdLock(e) ? true:false // is true … }

Thread Groups

- Every Java thread is a member of a thread group.
- Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.
- When creating a thread,
  - let the runtime system put the new thread in some reasonable default group (the current thread group) or
  - explicitly set the new thread's group.
- you cannot move a thread to a new group after the thread has been created.
  - when launched, main program thread belongs to main thread group.

Creating a Thread Explicitly in a Group

```
public Thread(ThreadGroup group, Runnable runnable)
public Thread(ThreadGroup group, String name)
public Thread(ThreadGroup group, Runnable runnable, String
    name)
```

```
ThreadGroup myThreadGroup = new ThreadGroup(
                                        "My Group of Threads");
       Thread myThread = new Thread(myThreadGroup,
                                 "a thread for my group");
```

Getting a Thread's Group

```
    theGroup = myThread.getThreadGroup();
```

The ThreadGroup Class

Collection Management Methods:

```
public class EnumerateTest {
        public void listCurrentThreads() {
            ThreadGroup currentGroup =
                    Thread.currentThread().getThreadGroup();
            int numThreads = currentGroup.activeCount();
            Thread[] listOfThreads = new Thread[numThreads];

            currentGroup.enumerate(listOfThreads);
            for (int i = 0; i < numThreads; i++)
                System.out.println("Thread #" + i + " = " +
                                    listOfThreads[i].getName());
        }
    }
```

Methods that Operate on the ThreadGroup

- getMaxPriority(),   setMaxPriority(int)
- isDaemon(),   setDaemon(boolean)
  - A Daemon thread group is one that destroys itself when its last thread/group is destroyed.
- getName() // name of the thread
- getParent() and parentOf(ThreadGroup)   // boolean
- toString()
- activeCount(), activeGroupCount()
- // # of active descendent threads, and groups

- suspend(); //deprecated; suspend all threads in this group.
- resume();
- stop();

Content(java.util.concurrent)

1)ExecutorService

2)ForkJoinPool

3)Blocking Queue

4)Concurrent Collections

5)Semaphore

6)CountDownLatch

7)CyclicBarrier

8)Lock

9)Atomic Variables

Java.util.concurrency

- A a set of ready-to-use data structures and functionality for writing safe multithreaded applications

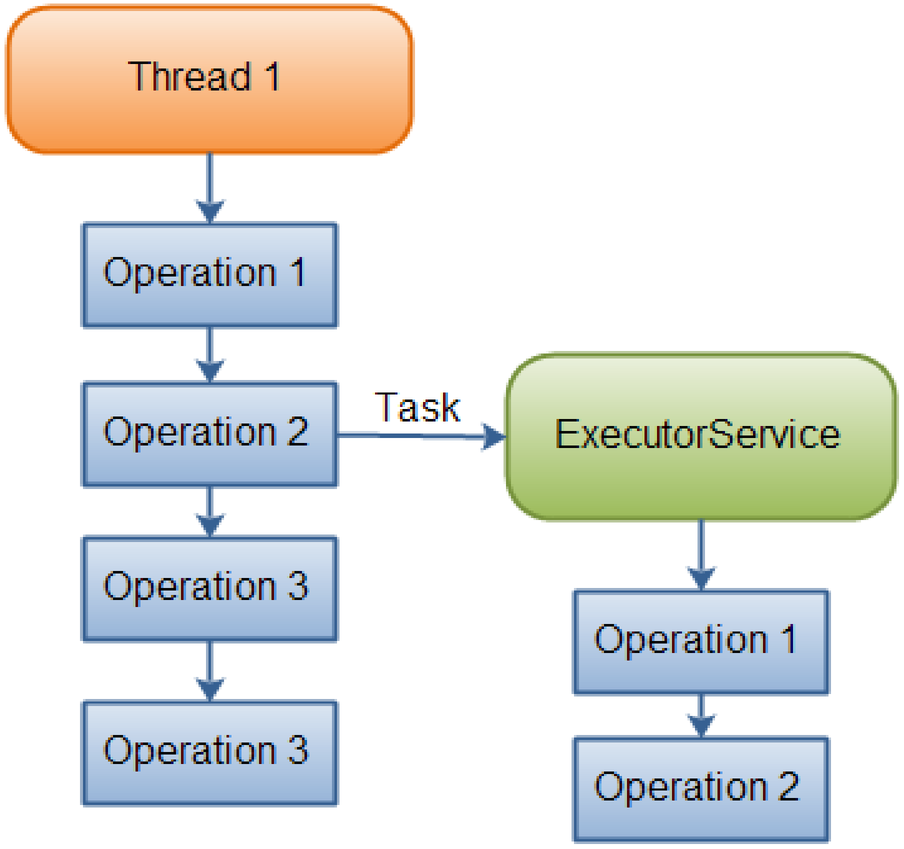
- it is not always trivial to write robust code that executes well in a multi-threaded environment

Executor Service

- The java.util.concurrent.ExecutorService interface represents an asynchronous execution mechanism which is capable of executing tasks in the background.
- It is very similar to a thread pool. In fact, the implementation of ExecutorService present in the java.util.concurrent package is a thread pool implementation.

ExecutorService

```
ExecutorService executorService
   =Executors.newFixedThreadPool(10);  //10 threads for executing
   tasks are created


executorService.execute(new Runnable() {
   public void run() {
      System.out.println("Asynchronous task");
   }
});


executorService.shutdown();
```



Thread 1 delegates a task to an Executor Service for asynchronous execution

Creating executor service

Using Executors factory class:

```
ExecutorService executorService1 =
   Executors.newSingleThreadExecutor();

ExecutorService executorService2 =
   Executors.newFixedThreadPool(10);

ExecutorService executorService3 =
   Executors.newScheduledThreadPool(10);
```

ExecutorService usage

There are a few different ways to delegate tasks for execution to an ExecutorService:

- **execute(Runnable)**
- **submit(Runnable)**
- **submit(Callable)**
- **invokeAny(...)**
- **invokeAll(...)**

execute(Runnable)

```
ExecutorService executorService =
    Executors.newSingleThreadExecutor();
executorService.execute(new Runnable() {
   public void run() {
       System.out.println("Asynchronous task");
   }
});
executorService.shutdown();
```

- There is no way of obtaining the result of the executed Runnable, if necessary. We have to use a Callable for that

submit(Callable)

```
Future future = executorService.submit(new Callable(){
    public Object call() throws Exception {
        System.out.println("Asynchronous Callable");
        return "Callable Result";
    }
});
System.out.println("future.get() = " + future.get());
```

- The Callable's result can be obtained via the Future object returned

invokeAll()

```
ExecutorService executorService =
    Executors.newSingleThreadExecutor();
Set<Callable<String>> callables = new HashSet<Callable<String>>();
callables.add(new Callable<String>() {
   public String call() throws Exception {
      return "Task 1";
   }
});
......
List<Future<String>> futures = executorService.invokeAll(callables);
for(Future<String> future : futures){
   System.out.println("future.get = " + future.get());
}
executorService.shutdown();
```

```
ExecutorService executor = Executors.newWorkStealingPool();
List<Callable<String>> callables = Arrays.asList(
        () -> "task1",
        () -> "task2",
        () -> "task3");
executor.invokeAll(callables)
    .stream()
    .map(future -> {
        try {
            return future.get();
        }
        catch (Exception e) {
            throw new IllegalStateException(e);
        }
    })
    .forEach(System.out::println);
```

Callables and Futures

- Callables are functional interfaces just like runnables but instead of being void they return a value.

```
Callable<Integer> task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
};
```

Callables and Futures

- Callables can be submitted to executor services just like runnables.
- the executor returns a special result of type Future which can be used to retrieve the actual result at a later point in time.
- After submitting the callable to the executor we can check if the future has already been finished execution via isDone()

```
ExecutorService executor = Executors.newFixedThreadPool(1);

Future<Integer> future = executor.submit(task);

System.out.println("future done? " + future.isDone());

Integer result = future.get();

System.out.println("future done? " + future.isDone());

System.out.print("result: " + result);


//future done? false

//future done? true

//result: 123
```

ExecutorService Shutdown

- To terminate the threads inside the ExecutorService you call its shutdown() method. It will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the ExecutorService shuts down

- to shut down the ExecutorService immediately, you can call the shutdownNow() method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks.

java.util.concurrent.ThreadPoolExecutor

- is an implementation of the ExecutorService interface.
- executes the given task (Callable or Runnable) using one of its internally pooled threads.

- The number of threads in the pool is determined by these variables:
  - corePoolSize
  - maximumPoolSize

ScheduledExecutorService

- It is an interface
- can schedule tasks to run after a delay, or to execute repeatedly with a fixed interval of time in between each execution.
- Tasks are executed asynchronously by a worker thread, and not by the thread handing the task to the ScheduledExecutorService.

ScheduledExecutorService

```
ScheduledExecutorService scheduledExecutorService =
        Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =
    scheduledExecutorService.schedule(new Callable() {
        public Object call() throws Exception {
            System.out.println("Executed!");
            return "Called!";
        }
    }, 5,TimeUnit.SECONDS);
```

- the Callable should be executed after 5 seconds

ScheduledExecutorService Usage

Once you have created a ScheduledExecutorService you use it by calling one of its methods:

- schedule (Callable task, long delay, TimeUnit timeunit)
- schedule (Runnable task, long delay, TimeUnit timeunit)
- scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)
- scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

ForkJoinPool

- is similar to the ExecutorService but with one difference
- implements the work-stealing strategy, i.e. every time a running thread has to wait for some result; the thread removes the current task from the work queue and executes some other task ready to run. This way the current thread is not blocked and can be used to execute other tasks. Once the result for the originally suspended task has been computed the task gets executed again and the join() method returns the result.

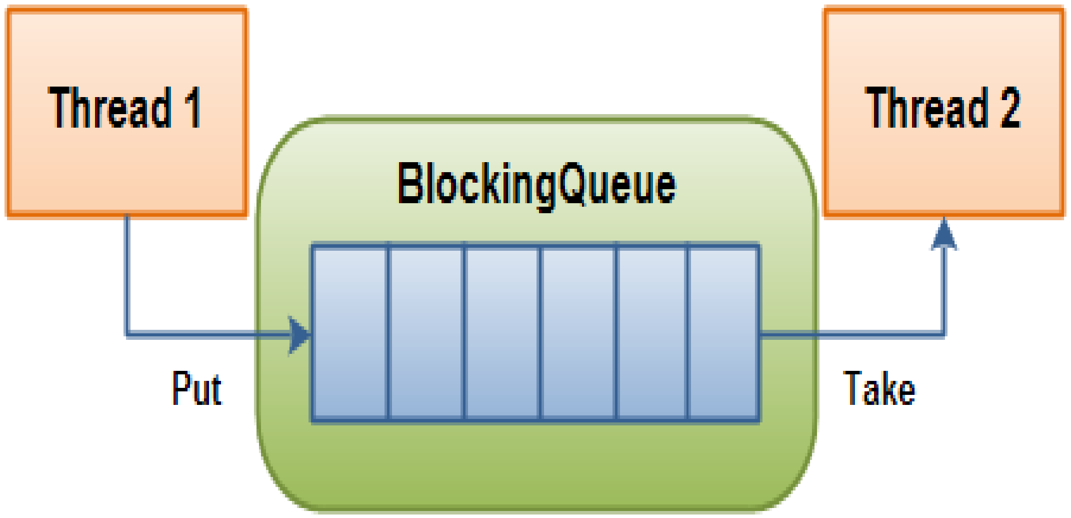
ForkJoinPool

- a call of fork() will start an asynchronous execution of the task,
- a call of join() will wait until the task has finished and retrieve its result.
- makes it easy for tasks to split their work up into smaller tasks(divide and conquer approach) which are then submitted to the ForkJoinPool too.

```
public class FindMin extends RecursiveTask<Integer> {
private static final long serialVersionUID = 1L;
private int[] numbers;
private int startIndex;
private int endIndex;

public FindMin(int[] numbers, int startIndex, int endIndex) {
this.numbers = numbers;
this.startIndex = startIndex;
this.endIndex = endIndex;
}

@Override
protected Integer compute() {
int sliceLength = (endIndex - startIndex) + 1;
if (sliceLength > 2) {
FindMin lowerFindMin = new FindMin(numbers, startIndex, startIndex + (sliceLength / 2)- 1);
lowerFindMin.fork();
```

```java
FindMin upperFindMin = new FindMin(numbers, startIndex + (sliceLength / 2), endIndex);
upperFindMin.fork();
return Math.min(lowerFindMin.join(), upperFindMin.join());
} else {
return Math.min(numbers[startIndex], numbers[endIndex]);
}
}

public static void main(String[] args) {
int[] numbers = new int[100];
Random random = new Random(System.currentTimeMillis());
for (int i = 0; i < numbers.length; i++) {
numbers[i] = random.nextInt(100);
}
ForkJoinPool pool = new ForkJoinPool(Runtime.getRuntime().availableProcessors());
Integer min = pool.invoke(new FindMin(numbers, 0, numbers.length - 1));
System.out.println(min);
}
}
```

Interface BlockingQueue



Interface BlockingQueue

- methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future:
  - one throws an exception: **add(e), remove(), element()**
  - the second returns a special value (either null or false, depending on the operation): **offer(e), poll(), peek()**
  - the third blocks the current thread indefinitely until the operation can succeed: **put(e), take()**
  - the fourth blocks for only a given maximum time limit before giving up: **offer(e, time, unit), poll(time, unit)**

```
public class BlockingQueueExample {

    public static void main(String[] args) throws Exception {

        BlockingQueue queue = new ArrayBlockingQueue(1024);

        Producer producer = new Producer(queue);
        Consumer consumer = new Consumer(queue);

        new Thread(producer).start();
        new Thread(consumer).start();

        Thread.sleep(4000);
    }
}
```

```
public class Producer implements Runnable{
    protected BlockingQueue queue = null;
    public Producer(BlockingQueue queue) {
        this.queue = queue;}

    public void run() {
        try {
            queue.put("1");
            Thread.sleep(1000);
            queue.put("2");
            Thread.sleep(1000);
            queue.put("3");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }}}
```

```
public class Consumer implements Runnable{
    protected BlockingQueue queue = null;
    public Consumer(BlockingQueue queue) {
        this.queue = queue; }

    public void run() {
        try {
            System.out.println(queue.take());
            System.out.println(queue.take());
            System.out.println(queue.take());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }}}
```

ConcurrentHashMap

- is very similar to the java.util.HashTable class, except that ConcurrentHashMap offers better concurrency than HashTable does.

- does not lock the Map while you are reading from it.

- does not lock the entire Map when writing to it. It only locks the part of the Map that is being written to, internally.

Semaphore

- the java.util.concurrent.Semaphore class is a counting semaphore.
- The counting semaphore is initialized with a given number of "permits".
- For each call to acquire() a permit is taken by the calling thread.
- For each call to release() a permit is returned to the semaphore.
- Thus, at most N threads can pass the acquire() method without any release() calls, where N is the number of permits the semaphore was initialized with.

Semaphore

As semaphore typically has two uses:

- To guard a critical section against entry by more than N threads at a time.
- To send signals between two threads.

```
ExecutorService executor = Executors.newFixedThreadPool(10);

Semaphore semaphore = new Semaphore(5);

Runnable longRunningTask = () -> {

    boolean permit = false;

    try {

        permit = semaphore.tryAcquire(1, TimeUnit.SECONDS);

        if (permit) {

            System.out.println("Semaphore acquired");

            sleep(5);

        } else {System.out.println("Could not acquire semaphore");}

    } catch (InterruptedException e) {

        throw new IllegalStateException(e);

    } finally {

        if (permit) {semaphore.release();

        }}}


IntStream.range(0, 10).forEach(i -> executor.submit(longRunningTask));

stop(executor);
```

CountDownLatch

- is initialized with a given count.
- This count is decremented by calls to the countDown() method.
- Threads waiting for this count to reach zero can call one of the await() methods. Calling await() blocks the thread until the count reaches zero.

```
CountDownLatch latch = new CountDownLatch(3);

Waiter      waiter      = new Waiter(latch);
Decrementer decrementer = new Decrementer(latch);

new Thread(waiter)     .start();
new Thread(decrementer).start();

Thread.sleep(4000);
```

```
public class Waiter implements Runnable{
     CountDownLatch latch = null;
    public Waiter(CountDownLatch latch) {
        this.latch = latch;}
    public void run() {
        try {
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Waiter Released");
    }
}
```

```
public class Decrementer implements Runnable {
   CountDownLatch latch = null;
    public Decrementer(CountDownLatch latch) {
        this.latch = latch;}
    public void run() {
        try {
            Thread.sleep(1000);
            this.latch.countDown();
            Thread.sleep(1000);
            this.latch.countDown();
            Thread.sleep(1000);
            this.latch.countDown();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }}}
```

Cyclic Barrier

- is a synchronization mechanism that can synchronize threads progressing through some algorithm.
- it is a barrier that all threads must wait at, until all threads reach it, before any of the threads can continue.
- The threads wait for each other by calling the await() method on the CyclicBarrier.
- Once N threads are waiting at the CyclicBarrier, all threads are released and can continue running.

```
Runnable barrier1Action = new Runnable() {
   public void run() {
      System.out.println("BarrierAction 1 executed ");}};
Runnable barrier2Action = new Runnable() {
   public void run() {
      System.out.println("BarrierAction 2 executed ");}};

CyclicBarrier barrier1 = new CyclicBarrier(2, barrier1Action);
CyclicBarrier barrier2 = new CyclicBarrier(2, barrier2Action);
CyclicBarrierRunnable barrierRunnable1 =
      new CyclicBarrierRunnable(barrier1, barrier2);
CyclicBarrierRunnable barrierRunnable2 =
      new CyclicBarrierRunnable(barrier1, barrier2);
new Thread(barrierRunnable1).start();
new Thread(barrierRunnable2).start();
```

```java
public class CyclicBarrierRunnable implements Runnable{

    CyclicBarrier barrier1 = null;
    CyclicBarrier barrier2 = null;

    public CyclicBarrierRunnable(
            CyclicBarrier barrier1,
            CyclicBarrier barrier2) {

        this.barrier1 = barrier1;
        this.barrier2 = barrier2;
    }
public void run() {
      try {
         Thread.sleep(1000);
         System.out.println(Thread.currentThread().getName() +
                         " waiting at barrier 1");
         this.barrier1.await();
         Thread.sleep(1000);
         System.out.println(Thread.currentThread().getName() +
                         " waiting at barrier 2");
         this.barrier2.await();
         System.out.println(Thread.currentThread().getName() +  " done!");
      } catch (InterruptedException e) {
         e.printStackTrace();
      } catch (BrokenBarrierException e) {
         e.printStackTrace();
      }}}
```

Lock

- is a thread synchronization mechanism just like synchronized blocks. A Lock is, however, more flexible and more sophisticated than a synchronized block.

```
Lock lock = new ReentrantLock();
lock.lock();
//critical section
lock.unlock();
```

ReadWriteLock

- allows multiple threads to read a certain resource, but only one to write it, at a time.

- Read Lock: If no threads have locked the ReadWriteLock for writing, and no thread have requested a write lock. Thus, multiple threads can lock the lock for reading.

- Write Lock: If no threads are reading or writing. Thus, only one thread at a time can lock the lock for writing.

```
ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

readWriteLock.readLock().lock();
    // multiple readers can enter this section
    // if not locked for writing, and not writers waiting
    // to lock for writing.
readWriteLock.readLock().unlock();

readWriteLock.writeLock().lock();
    // only one writer can enter this section,
    // and only if no threads are currently reading.
readWriteLock.writeLock().unlock();
```

Atomic Variables

- the atomic classes make heavy use of compare-and-swap (CAS), an atomic instruction directly supported by most modern CPUs.
- Those instructions usually are much faster than synchronizing via locks.
- The advice is to prefer atomic classes over locks in case you just have to change a single mutable variable concurrently.
- Many atomic classes: AtomicBoolean, AtomicInteger, AtomicReference, AtomicIntegerArray, etc

AtomicBoolean

- provides a boolean variable which can be read and written atomically, and which also contains advanced atomic operations like compareAndSet()
- Example:

```
AtomicBoolean atomicBoolean = new AtomicBoolean(true);

boolean expectedValue = true;

boolean newValue      = false;

boolean wasNewValueSet =
    atomicBoolean.compareAndSet(expectedValue, newValue);
```

AtomicInteger

```
AtomicInteger atomicInt = new AtomicInteger(0);
ExecutorService executor = Executors.newFixedThreadPool(2);
IntStream.range(0, 1000)
    .forEach(i -> {
        Runnable task = () ->
            atomicInt.updateAndGet(n -> n + 2); //thread-safe without synchronization
        executor.submit(task);
    });

stop(executor);
System.out.println(atomicInt.get());    // => 2000
```