Advanced Programming Methods
Lecture 11

- Concurrency in Java and C#

Concurrent Programing

- 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

3

Processes

- processes are often seen as synonymous with programs or applications

- what the user sees as a single application may in fact be a set of cooperating processes.

- in C# for each *.exe loaded into memory, the OS creates a separate and isolated process for use during its lifetime

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

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.

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

- By default, a process contains only a single thread, which executes from the beginning of the program to the end

- A thread can spawn other threads, so that at any time, a process might have multiple threads in various states, executing different parts of the program.

- If there are multiple threads in a process, they all share the process's resources.

Threads

- Threads are the units that are scheduled by the system for executing on the processor not processes

- 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.

Threads vs. Proceses

- processes are independent while thread is within a process.

- processes have separate address spaces while threads share their address spaces.

- processes communicate each other through inter-process communication.

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.

13

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

Concurrency in JAVA

Contents

1. What is a thread ?
2. Define and launch a thread
3. The life-cycle of a thread
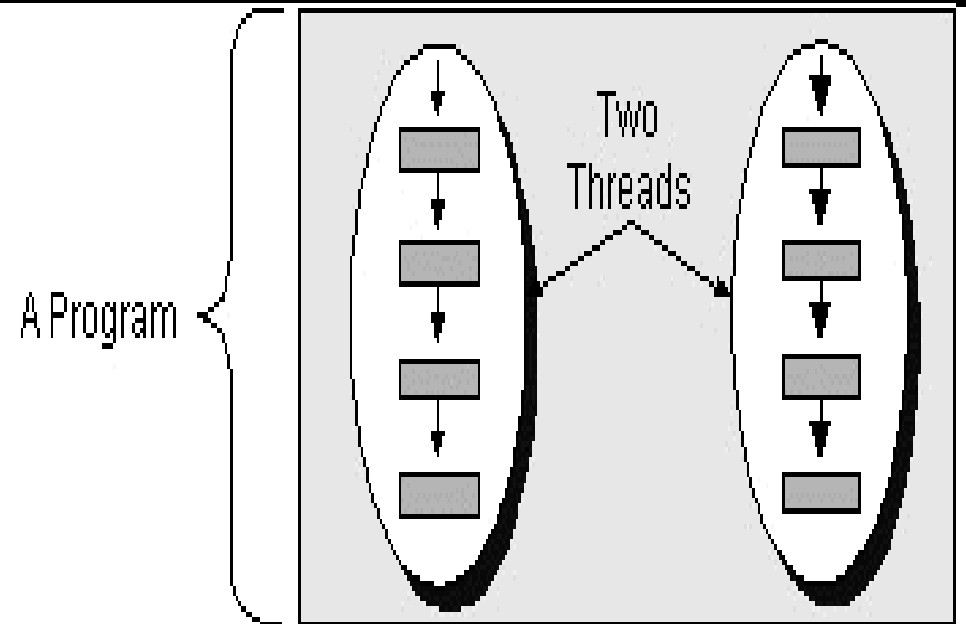4. interrupt a thread
5. thread synchronization
6. other issues

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 Program

A
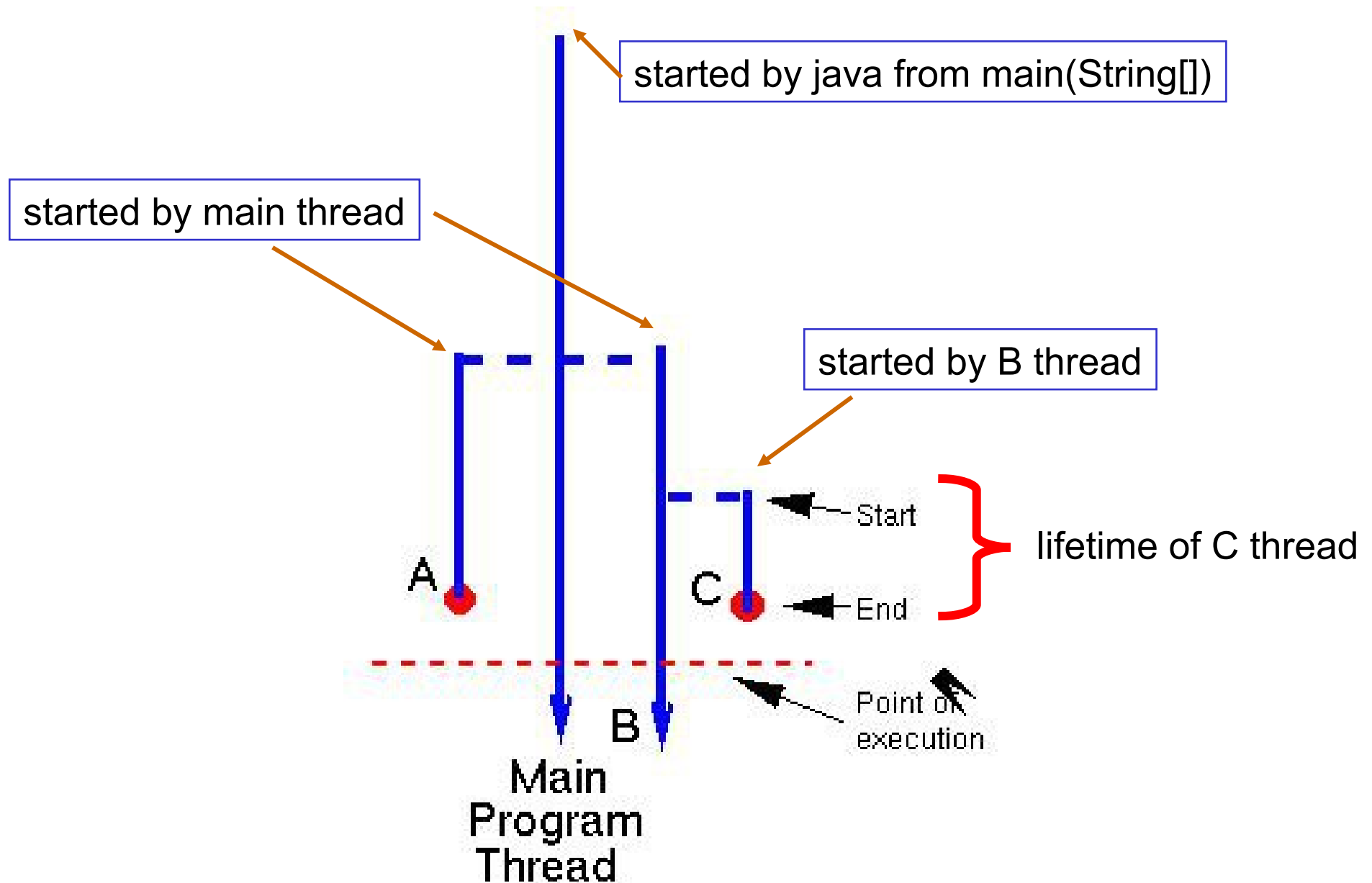Thread

A Program

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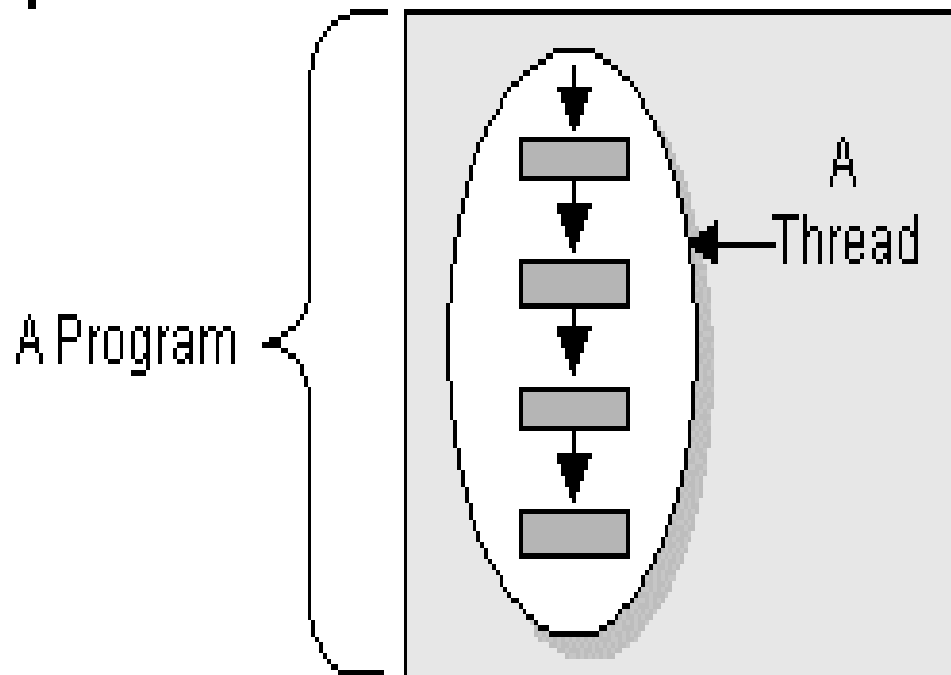
```
{ A(); A1();  A2();   A3();
  B1();  B2();  }
```

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

Thread ecology in a java program

started by java from main(String[])

started by main thread

started by B thread

lifetime of C thread

Start

A

C

End

Point of execution

B

Main
Program
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:**

  1. **Subclass the Thread class;**

  2. **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 …
    }
```

- **Impement 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] of Thread, say **thread**.

    1. Thread thread = new Print2Console();
    2. 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();
- …

// 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 thred 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.

- **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.

```java
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
1 Thread2        6 Thread2        9 Thread1
1 Thread1        6 Thread1         DONE! Thread1
2 Thread1        7 Thread1
2 Thread2        7 Thread2
3 Thread2        8 Thread2
3 Thread1        9 Thread2
4 Thread1        8 Thread1
4 Thread2
```
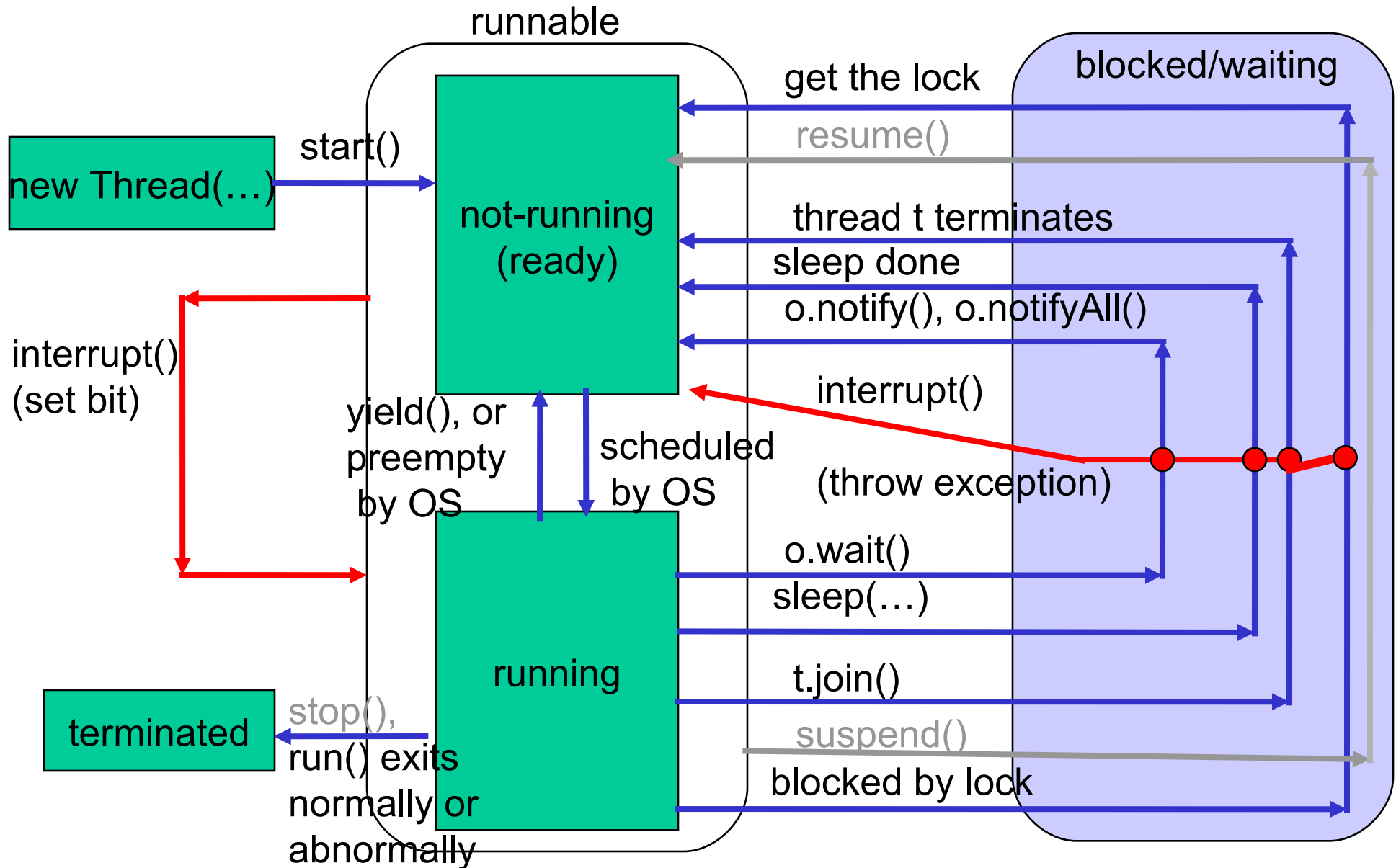
New → (Runnable → blocked/waiting) * → Runnable → dead(terminated)

- **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()**

- // retrun 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

runnable

new Thread(…)

start()

not-running
(ready)

running

terminated

stop(),
run() exits
normally or
abnormally

interrupt()
(set bit)

yield(), or
preempty
by OS

scheduled
by OS

blocked/waiting

get the lock

resume()

thread t terminates

sleep done

o.notify(), o.notifyAll()

interrupt()

(throw exception)

o.wait()

sleep(…)

t.join()

suspend()

blocked by lock

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 }**

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

- **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**

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()) { … }

- **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

```java
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);
```

```java
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);

  }
```

```java
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!!

- **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.

- **Java use the monitor concept to achieve <u>mutual exclusion</u> and <u>synchronization</u> 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();

```
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.
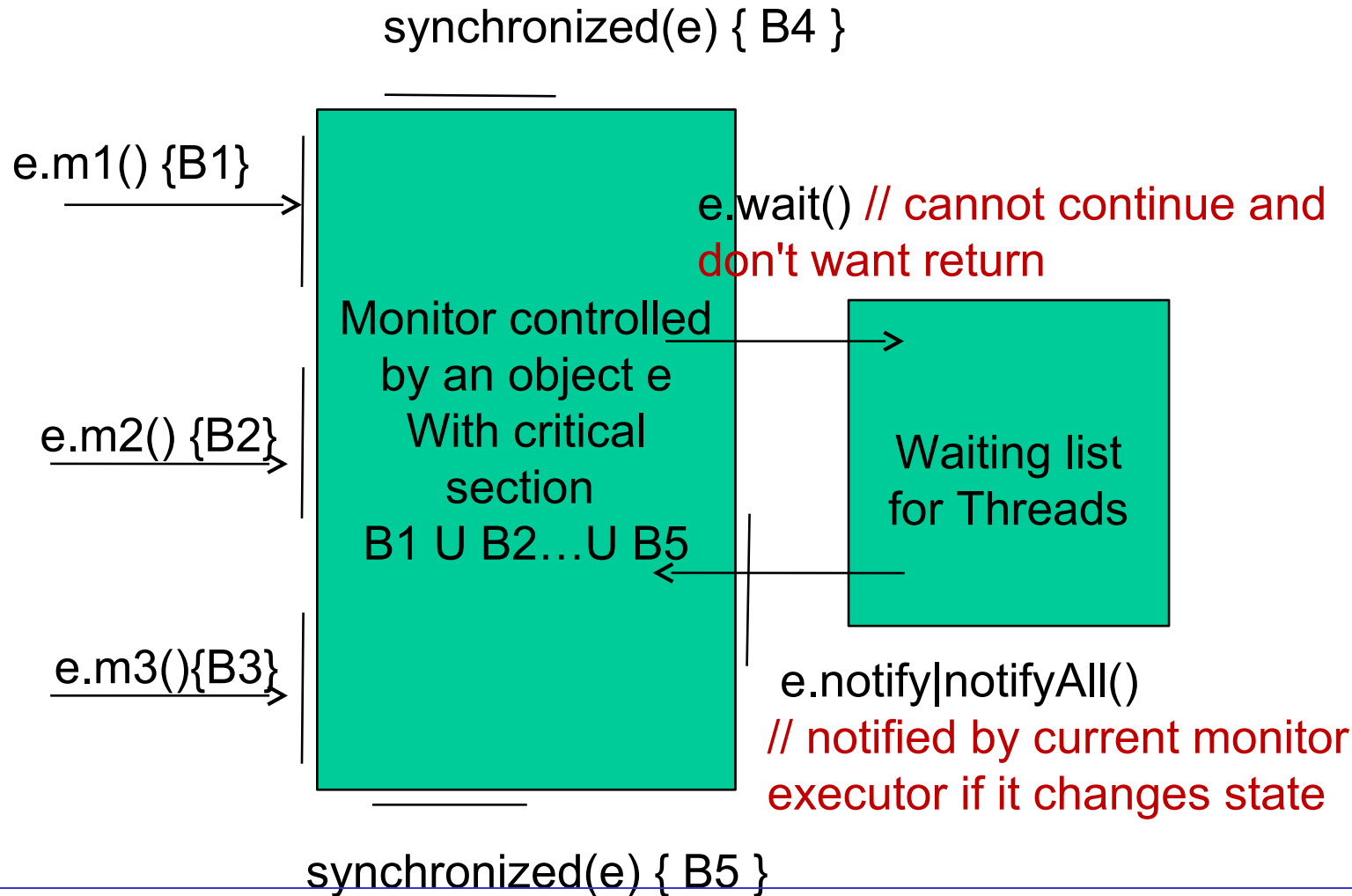
Java's Monitor Model

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

- 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

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

- **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.**

- **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

e.m2() {B2}

Waiting list
for Threads

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.**

```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

```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);
        }  }}
```

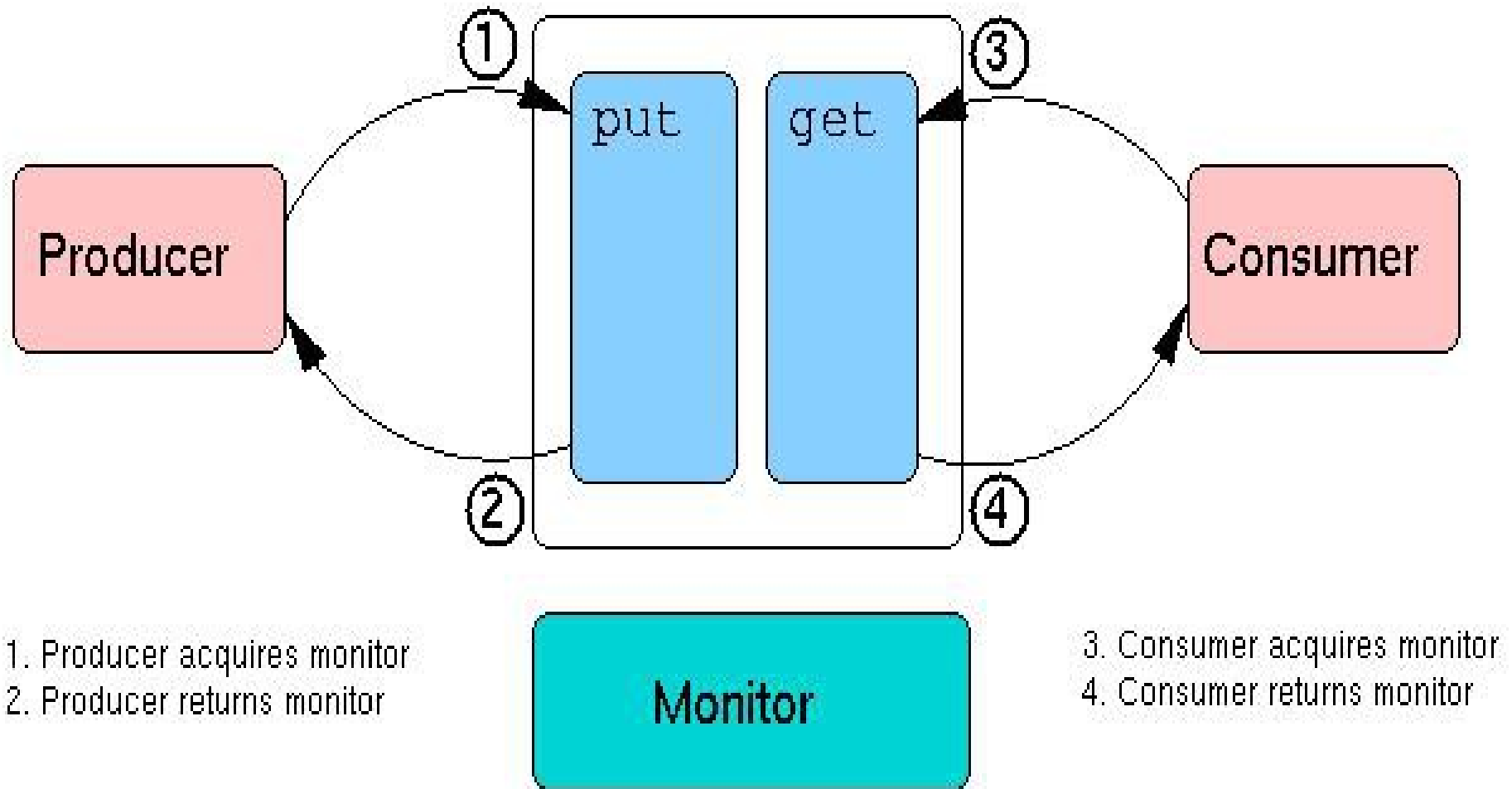## CubbyHole without mutual exclusion

```
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

```
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:

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

2.   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!)

```java
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

```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();    }}
```

```java
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();
    }
}
```

- **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 ... }**

- **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.

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();

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();**