Threads

Avancerad Javaprogrammering

Utbildare: Mikael Olsson

Innehåll

- Vad är en tråd / thread?
- Definiera och starta en tråd
- Trådens livscykel
- Avbryta en tråd
- Synkronisera trådar

Enskilda samtalen

- Andelen som har sagt att de kan kontakta mig om det är något man vill förändra i utbildningen: 100%
- Ett par studenter har framfört åsikter till Fia om utbildningens kvalité.
- Vi saknar dock någon konkret punkt på vad som är fel.
    - För mig ger det intrycket att man inte hänger med och att det då måste vara utbildningens fel. Jag är dock öppen för andra tolkningar.
- Vi har låtit oberoende lärare granska allt material i utbildningen och fått omdömet att materialet är pedagogiskt och välstrukturerat, exemplen är väl uppbyggda och övningsuppgifterna är väl strukturerade och ligger på en nivå man kan förvänta sig i den här kursen.

Vad kan vi göra?

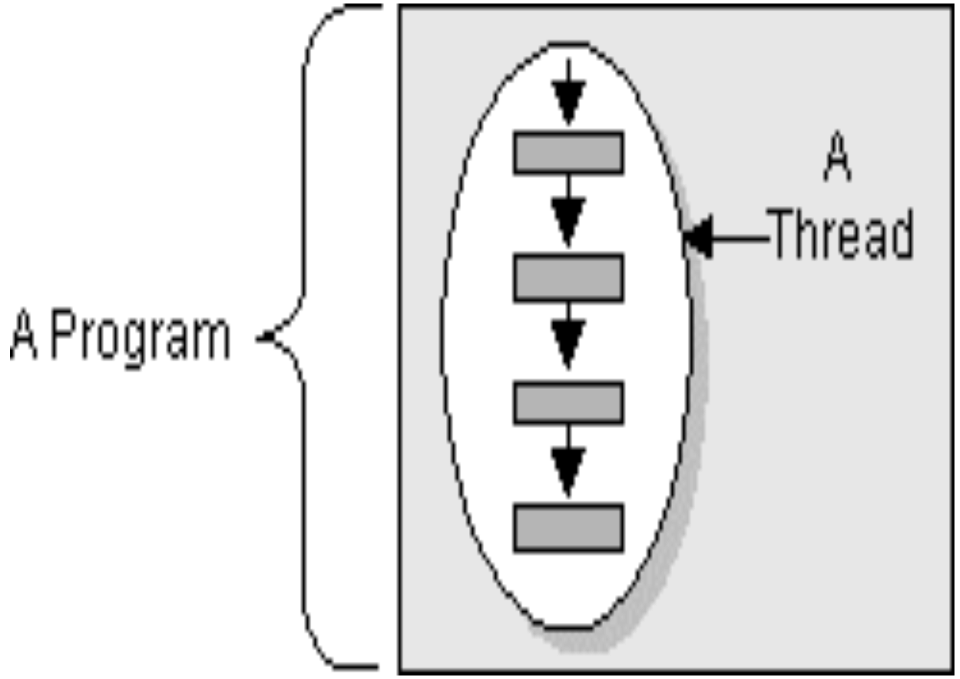
- Ni har ett stort ansvar i att
  - Följa kursen och se till att ni förstår innehållet.
  - Göra uppgifter.
  - Fråga på föreläsningarna.
  - Prata med mig om vad ni **konkret** vill förändra.
- Jag förändrar gärna arbetssätt om det kan hjälpa fler. Mitt intryck är dock inte att det är arbetssättet (som bygger på väl beprövad pedagogik) som är fel utan att ämnet är svårt och kräver mycket egna studier.
- Nu har vi inte så mycket tid kvar, men vi kan prata om vad vi kan göra med den tiden som är kvar.
- Vårt mål är att alla ska förstå och klara kursen, men det är ni som måste göra jobbet.

Vad är en tråd?

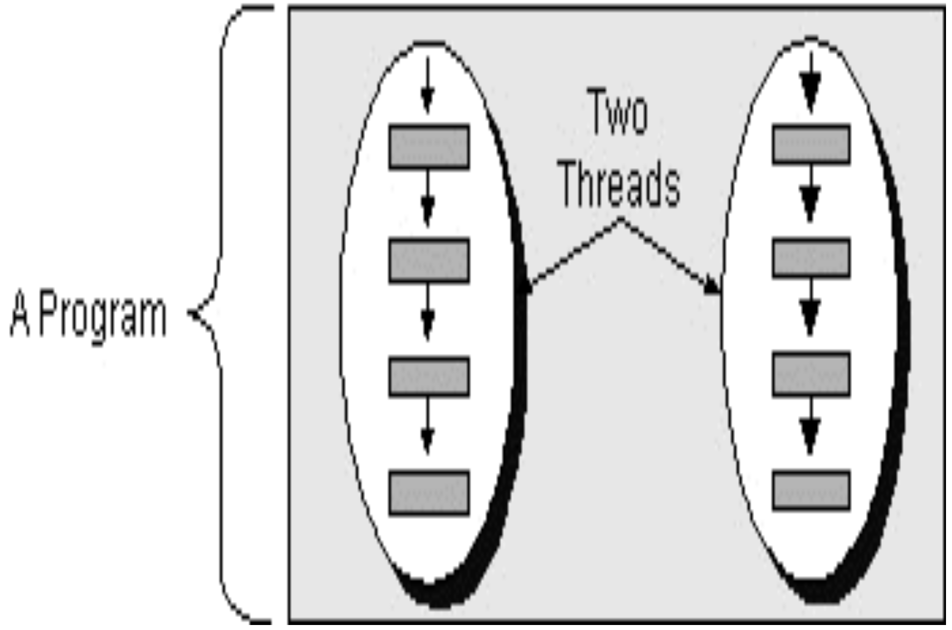
- Ett sekventiellt (singeltrådat) program har ett enda flöde.
  - Vid en godtycklig tidpunkt kan som mest en instruktion utföras i programmet.
- Ett multitrådat program kan ha flera flöden parallellt.
  - Vid en godtycklig tidpunkt kan som mest en instruktion utföras i programmet.
- En tråd är ett kontrollflöde

Singeltrådad vs multitrådad

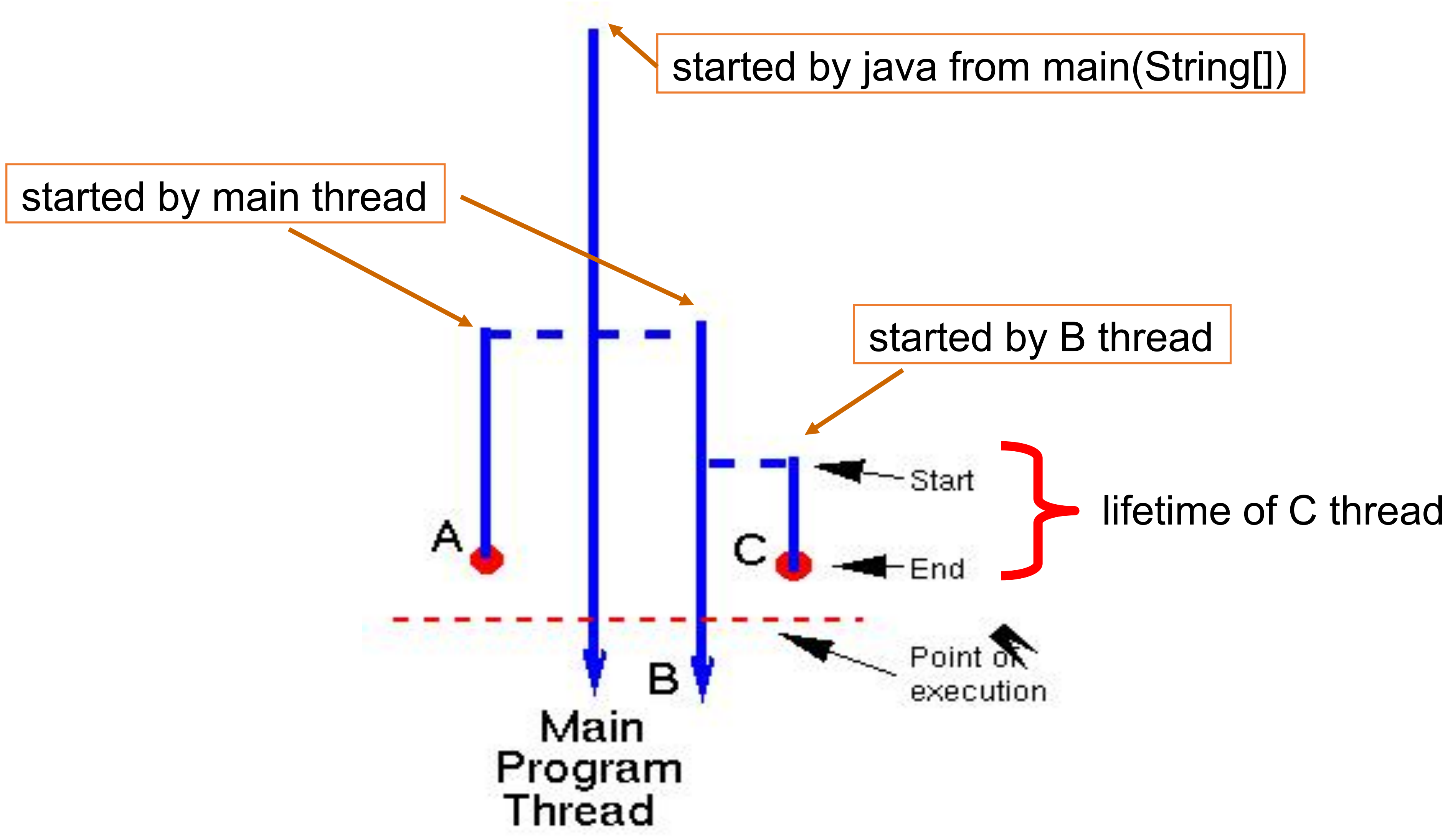
- En tråd kan använda en kärna.



- Två trådar kan använda varsin kärna och utnyttja processorn mer effektivt.



Trådar i ett java-program



Definiera och starta en tråd

- Varje tråd finns inkapslad i en `java.lang.Thread`-instans.
- Det finns två sätt att definiera en tråd:
    1. Extenda Thread-klassen
        1. Ärva från Thread-klassen
        2. Overrida Thread-metoden `run()`.
            - `run()` är entrypoint i en tråd, precis som `main(String[])` är det i ett program.

    2. Implementera interfacet Runnable

```
package java.lang.*;
public interface Runnable { public void run(); }
```

Definiera en tråd - arv

```
public class Print2Console extends Thread {
  public void run() {
    for (int b = 0; b < 10; b++) {
      System.out.println(b);
    }

    // additional methods, fields …
  }
}
```

Definiera en tråd - interface

```
public class Print2GUI implement Runnable {
  public void run() {
    for (int b = 0; b < 10; b++) {
      System.out.println(b);
    }
  }
}
```

Hur man startar en tråd

1. Skapa en instans av [en subklass till] Thread.

```
Thread thread = new Print2Console();
Thread thread = new Thread( new Print2GUI( ... ) );
```

2. Kalla på dess `start()`-metod (inte `run()`)

```
Printer2Console t1 = new Print2Console(); // t1 is a thread instance
t1.start() ; // this will start a new thread -> t1.run()
... // main thread continues, doesn't wait for child thread to complete
Print2GUI jtext = new Print2GUI();
Thread t2 = new Thread( jtext);
t2.start();
```

java.lang.Thread-konstruktorer

- // 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.

Några access-metoder för trådar

- `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.`

Vad är en daemon thread?

- A daemon thread is a thread that does not prevent the JVM from exiting when the program finishes but the thread is still running. An example for a daemon thread is the garbage collection.
- You can use the `setDaemon(boolean)` method to change the Thread daemon properties before the thread starts.
- https://stackoverflow.com/questions/2213340/what-is-a-daemon-thread-in-java#answer-2213348
- https://www.baeldung.com/java-daemon-thread

State-metoder för nuvarande thread-åtkomster

- `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.

Exempel

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

Huvudprogram

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

- Möjlig utskrift:

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

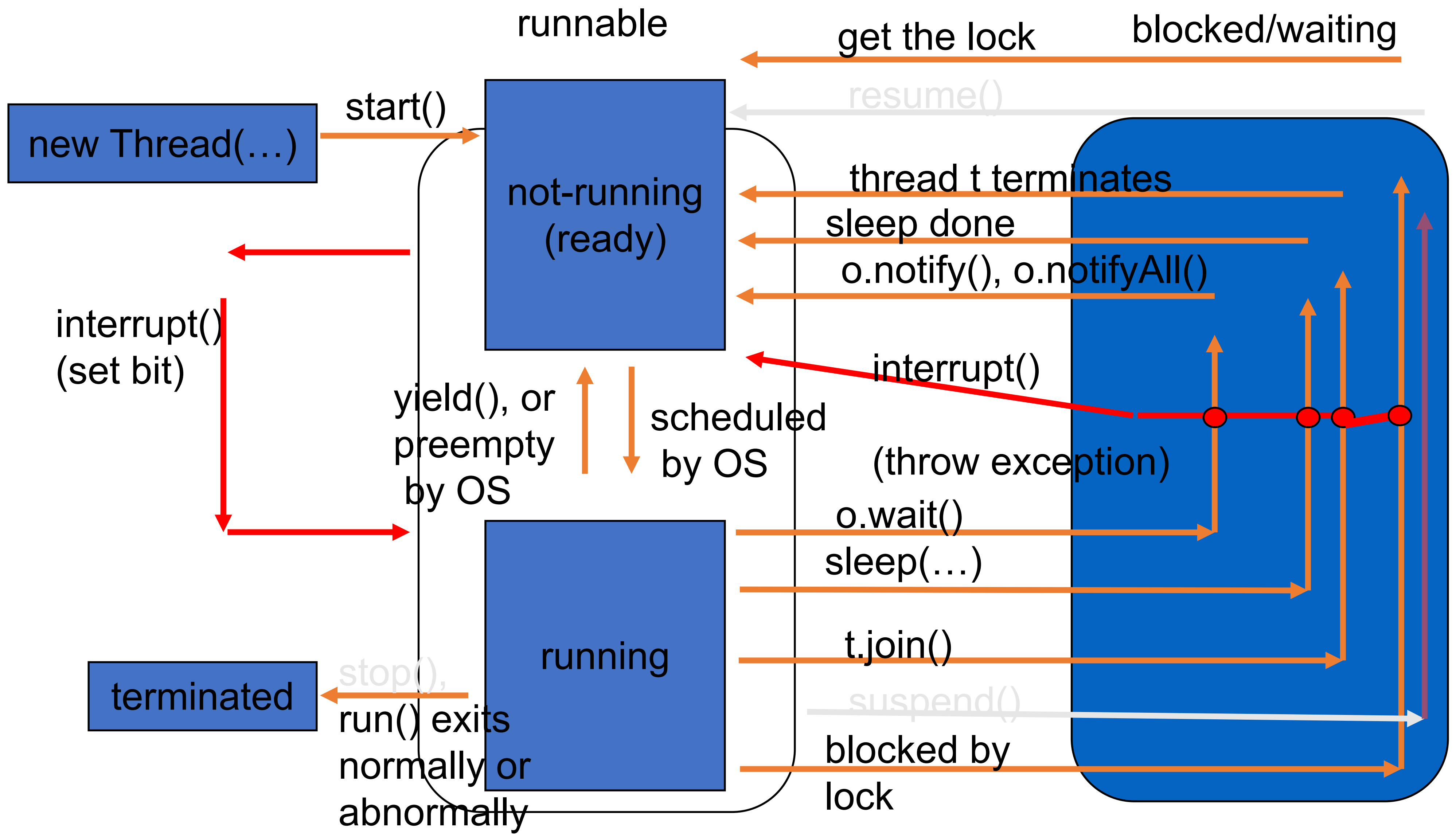
Trådens livscykel

- 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.

Trådens livscykel

```
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()
  }
}
```

Trådens livscykel



State transition-metoder för 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 -> leads to inconsistency`
- // state checking
- `public boolean isAlive() ; // true if runnable or blocked`

Avbrutna (interrupted) trådar

- 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 interrupt() then continue here
    // thread interrupted during sleep or wait      }
  }
}
```

Avbrutna trådar

- 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-relaterade metoder

- `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()...)` …

Trådsynkronisering

- 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.

Multitrådning kan förorsaka inkonsekvens

- 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);
}
```

- The execution sequence:
- 1,4,2,5,3,6 will result in unwanted result !!
- Final balance is 50 instead of 100!

```
• 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
```

Synkroniserade metoder och 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.

Synkronisera trådar

- 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();`

Typisk användning

- 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.

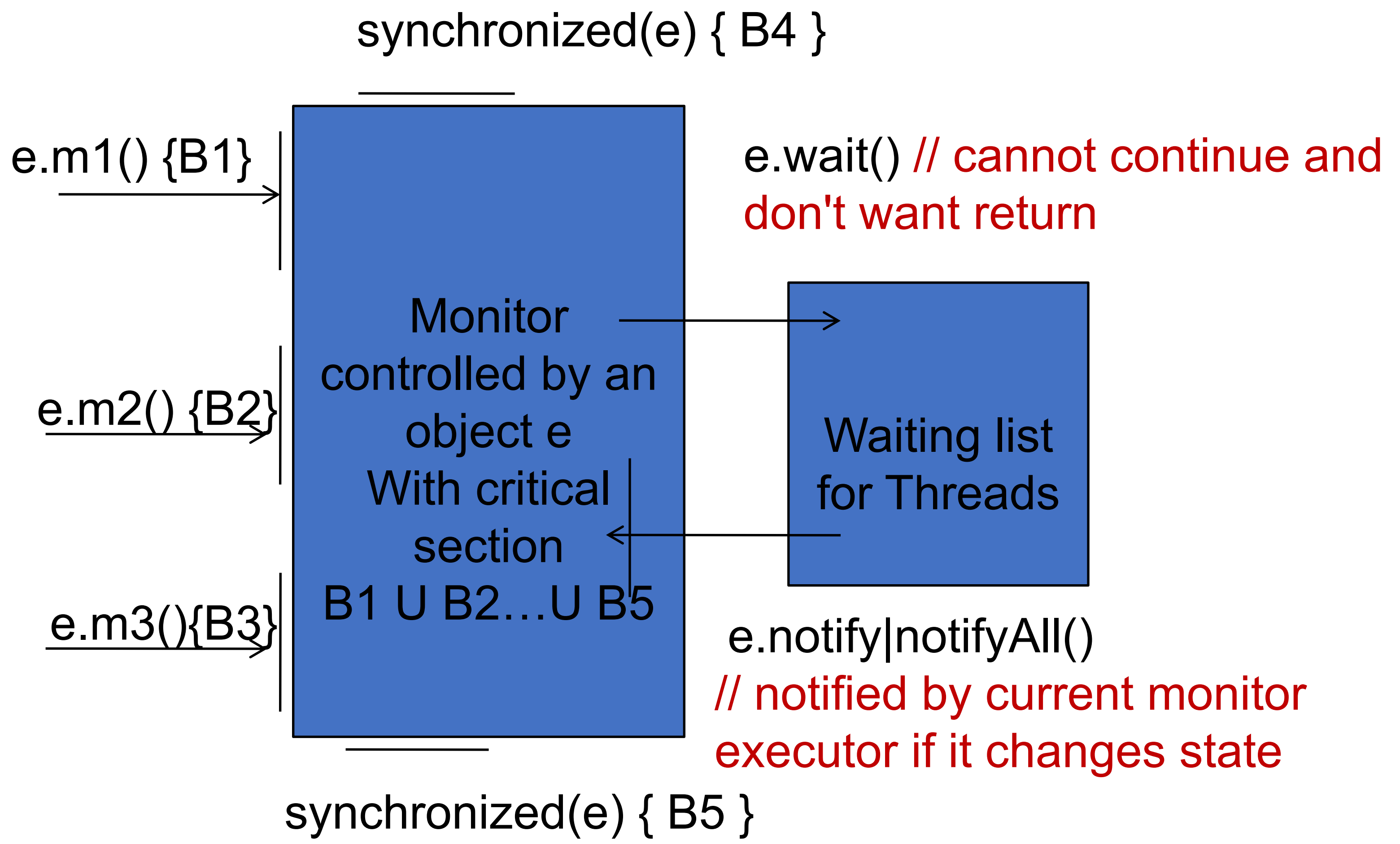
Javas monitor-modell

- 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 have 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 exits the critical section.

Javas monitor-modell

- 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`.

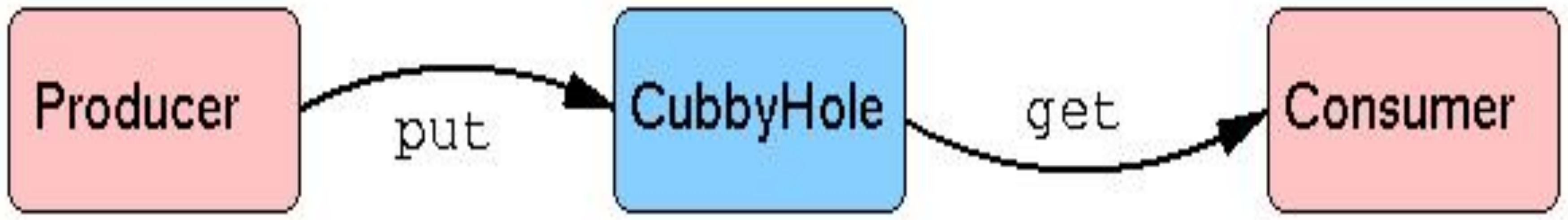
Javas monitor-modell


synchronized(e) { B4 }
e.m1() {B1}
e.m2() {B2}
e.m3(){B3}
Monitor controlled by an object e With critical section B1 U B2…U B5
e.wait() // cannot continue and don't want return
Waiting list for Threads
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) { }
      }
    }
  }
}
```

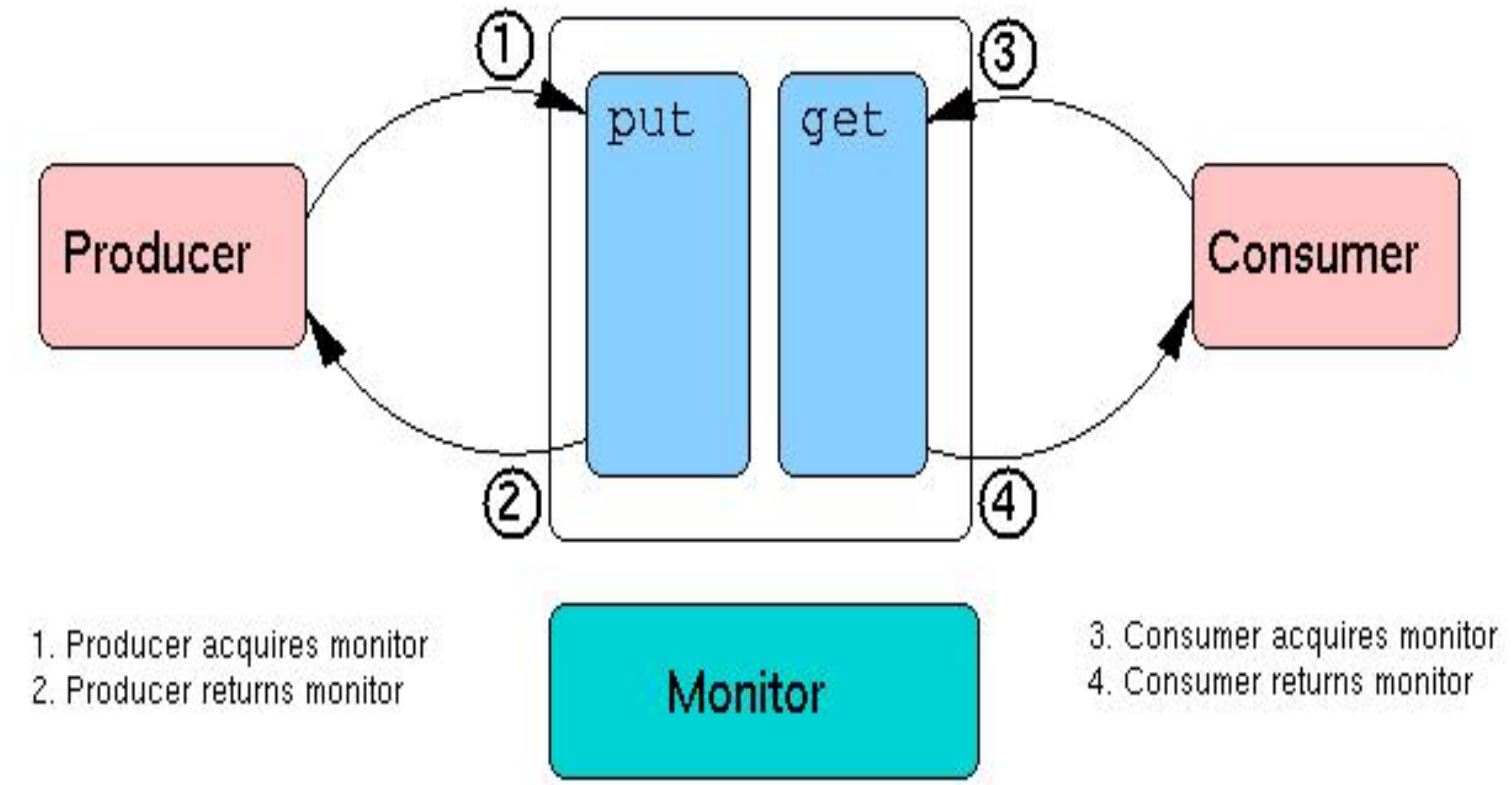
Consumer.java

```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 utan 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.

CubbyHole


1
put
get
3
Producer
Consumer
2
4
1. Producer acquires monitor
2. Producer returns monitor
Monitor
3. Consumer acquires monitor
4. Consumer returns monitor

CubbbyHole utan synkronisering

```
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)`

En annan implementation av CubbyHole

(fortfarande inte korrekt)

```
public 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
availeable == 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;
  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;     }
```

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

Main

```
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();
  }
}
```

Annat

- 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 … }

Trådgrupper (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.

Skapa en tråd explicit i en grupp

```
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();
```

ThreadGroup-klassen

- 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());
    }
  }
}
```

Metoder som opererar på ThreadGroups

```
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();
```

Övning

- Write a class Main.java with at least 3 classes
  - `public class  Main{…} , class  ThreadA, class ThreadB`
- ThreadA extends Thread, whose run() method will randomly and repeatedly println “You say hello” or “You say good morning” to the console, until interrupted by Main
- ThreadB extends Thread, whose run() method will println “I say good bye” or “I say good night” to the console depending on whether ThreadA said “…hello” or “…good moring”, until it is interrupted (or informed) by ThreadA.
- Main is the main class, whose main(…) method will
  - create an instance of ThreadA and an instance of ThreadB and start them.
  - read a char from the console
  - interrupt threadA, which will then interrupt (and terminate)threadB, and then terminate (by running to completion) itself. (notes: don’t use stop() method).
- Requirement: ThreadA and threadB must be executed in such a way that
  - The output is started with a “…hello” or “…good morning“ line said by A,
  - Every line said by A is followed by a corresponding line said by B, which, unless is the last line, is followed by a line said by A .
  - The output is ended with a line said by B.
  - To avoid too many messages shown in the console in a short time, you threads are advised to sleep() for a short time after printing a message.

En kandidat till Main-klass

```
package exercise;
public class Main {
 public static void main(Sting[] args) {
  Thread a = new ThreadA();
  Thread b = new ThreadB();
   a.setPartner(b); a.start();  b.start();
  try {
    System.out.println(“type any key to terminate:”);
    System.in.read();
  } catch ( Exception e) { }
  a.interrupt();
  try {
    a.join();
  } catch(Exception e){ }
  }
}
```

- Note: You may change the content of Main if it does not meet your need.

Uppgift - Interface

- exercises/09 - interfaces/description.txt