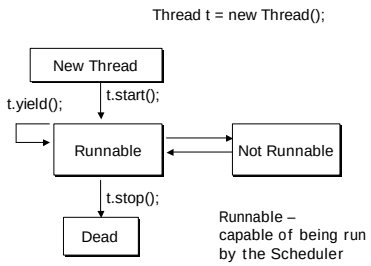
Chapter 4
Threads

Threads
Thread Groups

Life Cycle of a Thread

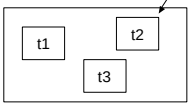


Thread Priority

- t1.start();
- t2.start();
- t3.start();

t2 runs until:

1. a higher priority thread becomes runnable
2. it yields, or its run() method terminates
3. in multi-tasking systems, its slice expires



The "Runnable" pool

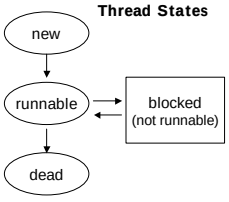
Thread States and Thread Messages

Messages in class Thread

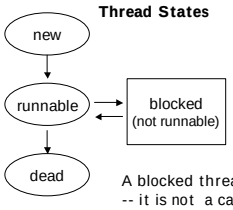
- t.start()
- t.yield()
- t.stop()
- t.resume()
- t.sleep()
- t.interrupt()
- t.destroy()

Other Messages:

- wait()
- notifyAll()

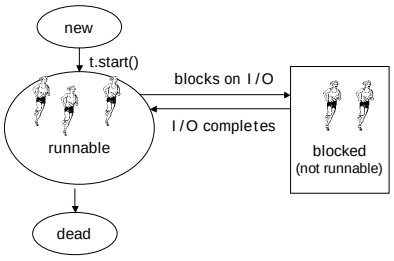


Blocked Threads

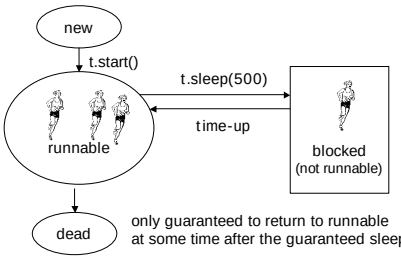


A blocked thread is not in the Runnable pool
-- it is not a candidate for selection by the scheduler

Threads Block on I/O

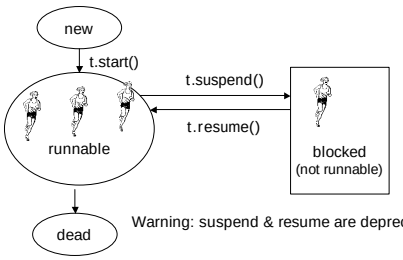


Threads Block with sleep()



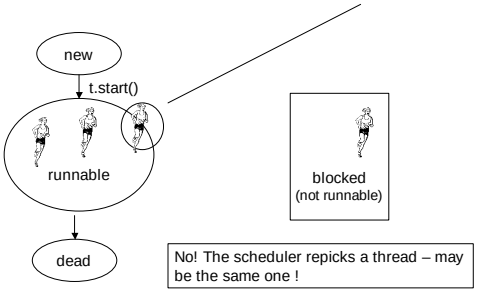
only guaranteed to return to runnable
at some time after the guaranteed sleep time

Threads Block with suspend()



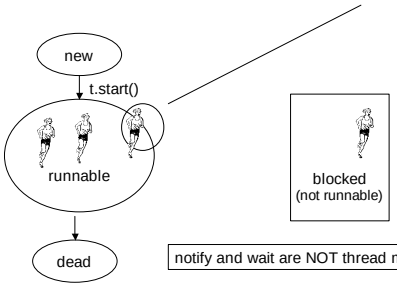
Warning: suspend & resume are deprecated !

Do Threads Block with yield() ?



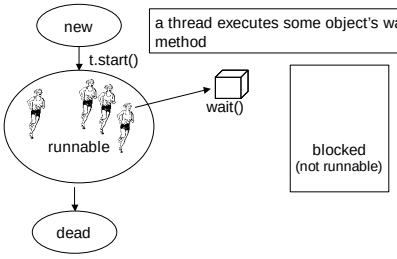
No! The scheduler repicks a thread – may be the same one !

Blocking with wait() and notify()



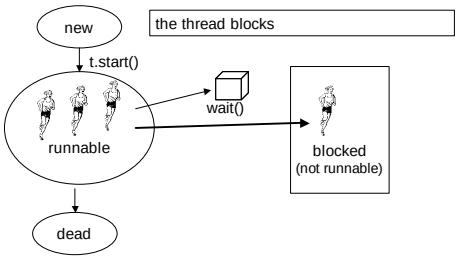
notify and wait are NOT thread methods!

Blocking - wait() and notify()

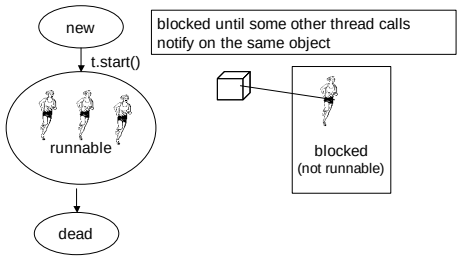


a thread executes some object's wait method

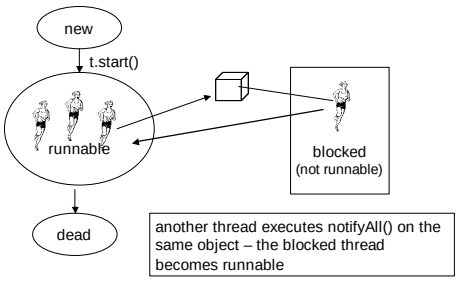
Blocking - wait() and notify()



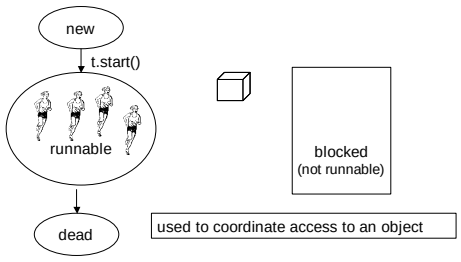
Blocking - wait() and notify()



Blocking - wait() and notify()

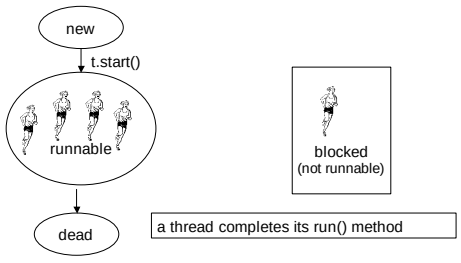


Blocking - wait() and notify()

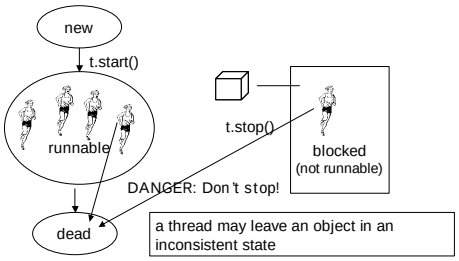


Thread Death

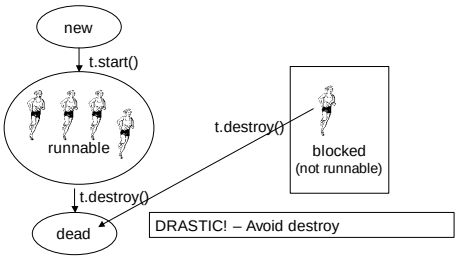
Natural Death



Death with stop()



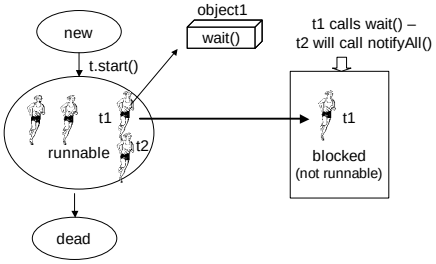
Death with destroy()



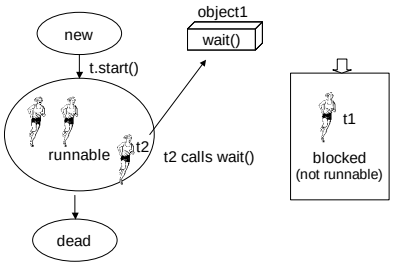
Deadlock I

(waiting for each other)

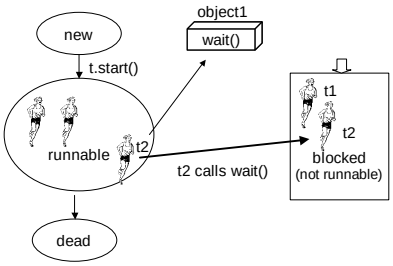
Deadlock



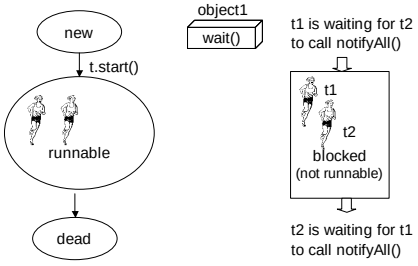
Deadlock



Deadlock



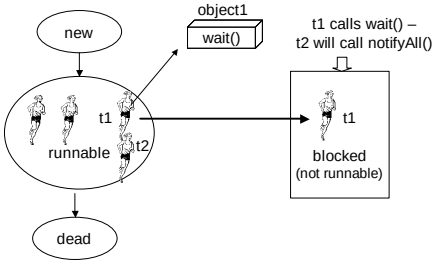
Deadlocked !



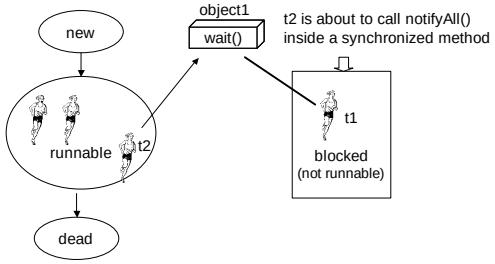
Deadlock I I

(Suspended before I could notify my friend)

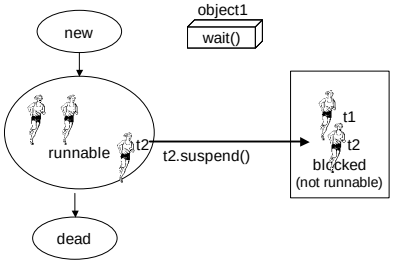
Deadlock



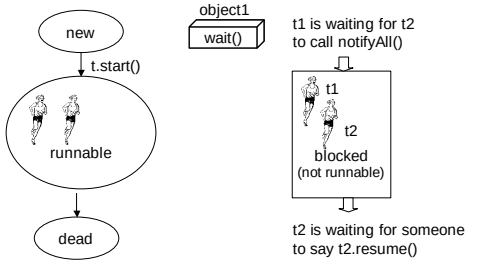
Deadlock

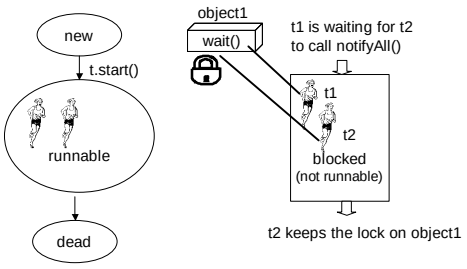


Deadlock



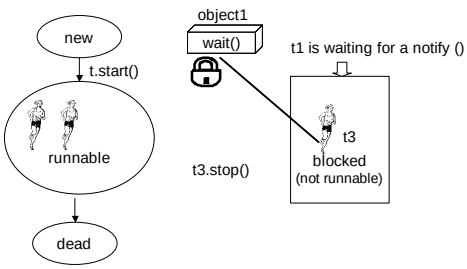
Deadlocked !

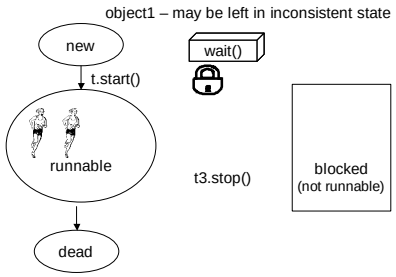




The problem with stop()

(Why it's deprecated)





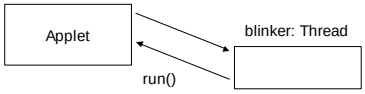
stop() - depracated

- stop() is inherently unsafe.
  - Stopping a thread with Thread.stop() causes it to unlock all monitors (objects) that it has locked
  - If any objects protected by these monitors were in an inconsistent state, the damaged objects are visible to other threads, potentially resulting in arbitrary behavior.

stop()...

- stop() should be replaced by code that modifies some variable to indicate that the target thread should stop running.
  - The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running.
- If the target thread waits for long periods (on a condition variable, for example), the interrupt method should be used to interrupt the wait.

Example – stop()



thread sleeps and when it wakes up calls repaint()

```
private Thread blinker;
public void start() {
   blinker = new Thread(this);
   blinker.start();
}

public void stop() {
   blinker.stop(); //UNSAFE
}

public void run() {
   Thread thisThread = Thread.currentThread();
   while (true) {
        try {
          thisThread.sleep(interval);
        } catch (InterruptedException e) { }
   }
   repaint();
}
```

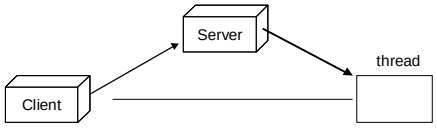
⇦ creates thread to execute applet's run method

```
private Thread blinker;

public void stop() {
   blinker = null;
}

public void run() {
   Thread thisThread = Thread.currentThread();
   while (blinker == thisThread) {
        try {
          thisThread.sleep(interval);
        } catch (InterruptedException e) { }
   }
   repaint();
}
```

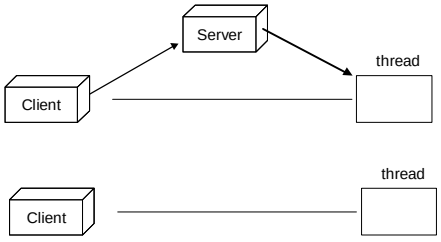
⇦ blinker no longer points to current thread

Thread Groups

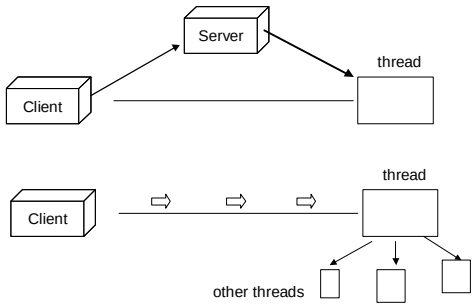
Multi-Threaded Server



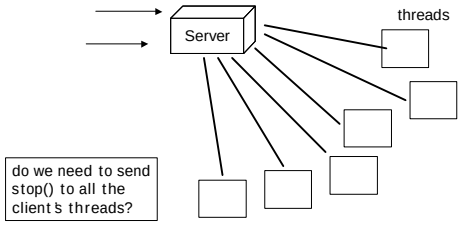
Multi-Threaded Server



Multi-Threaded Server



Server Thread Management



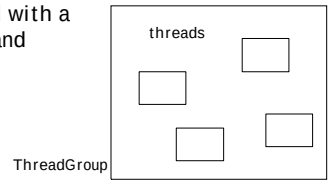
do we need to send stop() to all the client's threads?

ThreadGroup

- A set of threads
  - can include other thread groups
- Thread groups form a tree
  - every thread group except the initial one has a parent
- A thread may access information about its own thread group, but not about its thread group's parent or any other thread groups.

ThreadGroup

- Allows groups of threads to be maniputlated with a single command
  - suspend()
  - resume()
  - stop()

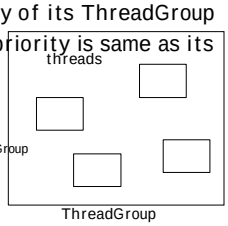


ThreadGroup Priority

- The priority of a thread cannot be set higher than the priority of its ThreadGroup
- Default ThreadGroup priority is same as its parent ThreadGroup

int getMaxPriority()
Returns the maximum priority of this ThreadGroup

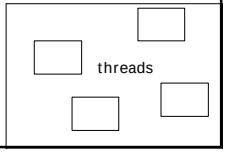
void setMaxPriority(int priority)
Sets max priority for the group



ThreadGroup Priority

- The priority of the:
  - System ThreadGroup is 10
  - Applet ThreadGroup is 6
- Attempt to set thread priority higher than ThreadGroup priority will silently fail

can never raise a ThreadGroup's priority



Default ThreadGroup

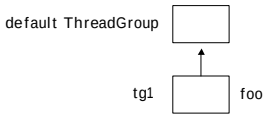
- Akk Java threads belong to some thread group
- If you don't specify, an arbitrary thread belongs to the "default" ThreadGroup

Creating Thread Groups

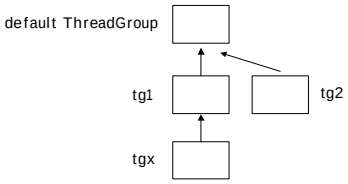
- ThreadGroup(String name)
  - creates a ThreadGroup with the given name
  - the thread group is automatically a child of the current ThreadGroup
- ThreadGroup(ThreadGroup parent, String name)
  - creates a thread group that descends from the parent

ThreadGroup Creation

ThreadGroup tg1 = new ThreadGroup("foo");



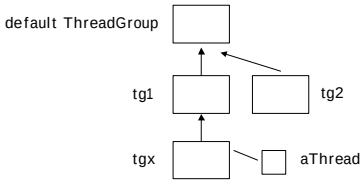
ThreadGroup Creation



Only applications can create ThreadGroups;

Threads created in an Applet are part of the Applet ThreadGroup

ThreadGroup Creation



Individual threads are not explicitly added to a ThreadGroup

A thread is placed in a ThreadGroup when the thread is created

Basic Thread Creation

- Thread()
- Thread(String name)
- Thread(Runnable target)
- Thread(Runnable target, String name)

The thread belongs to the ThreadGroup of the thread executing the statement!

Basic Thread Creation – with Group

- Thread(ThreadGroup g, String name)
- Thread(ThreadGroup g, Runnable target)
- Thread(ThreadGroup g, Runnable target, String name)

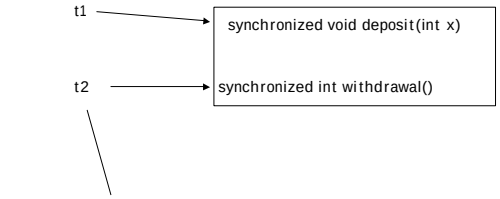
The thread belongs to the specified ThreadGroup g

There are NO methods to remove a thread from its ThreadGroup

Objects and Threads

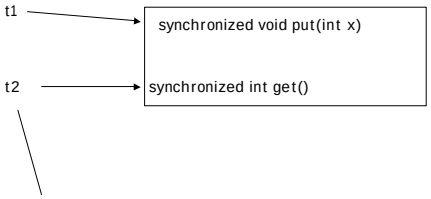
- Each object has a lock that can be obtained and released
  - Java uses an object's synchronized methods to control thread access to the object
- Each object also has a waiting area for threads that need something other threads can provide
  - wait and notify

Synchronization



The threads are essentially independent – no order enforced

What if ordering required? (put must preceed get)



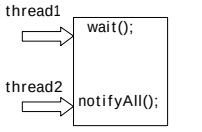
synchronized alone cannot guarantee on ordering

Coordination Solution #1

```
int value = -1;  //no value yet

synchronized void put(int x) {
   value = x;
}

synchronized int get() {
   while (value == -1) sleep(500);
   return value;
}
```

notify and wait

- wait()
  - wait for in an object's waiting area
    - hope that another thread will execute notify or notifyAll
- notifyAll()
  - wake up all threads waiting in the object's waiting area

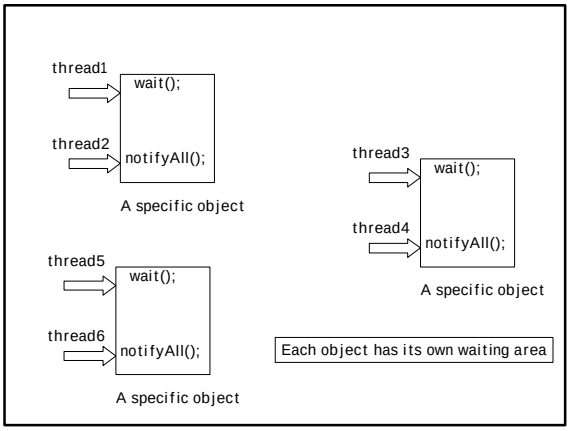


A specific object

notify and wait (methods in class Object)

- All objects understand nofity(), notifyAll() and wait()
- Use nofify & wait in your code if:
  - you have get and put methods
  - you want consumer threads to wait on producer threads
  - you want producer threads to notify customers

wait and notify is a communication mechanism



Coordination Solution #2

```
int value = -1;  //no value yet

synchronized void put(int x) {
   value = x; notifyAll();
}

synchronized int get() {
   while (value == -1) wait();
   return value;
}
```

guarantees that a put will aways occur before a get

⇦ No busy wait loop