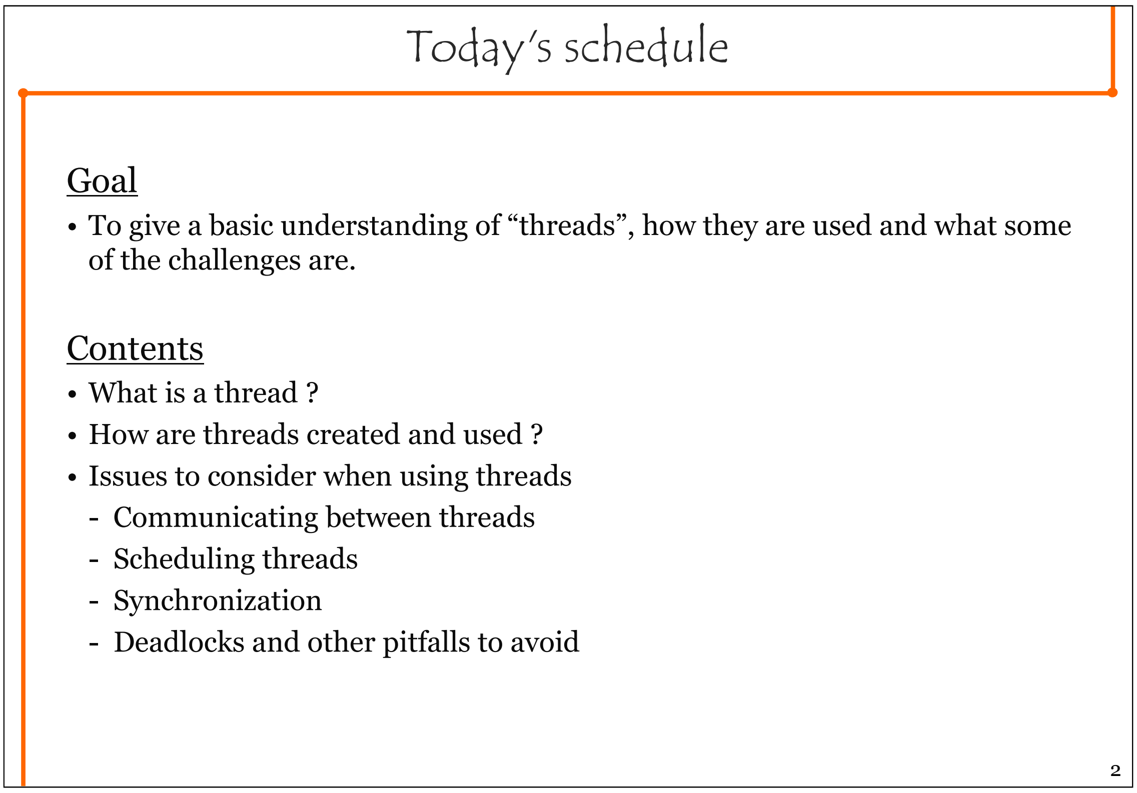
OOP Lecture 4
Thread Programming

Mette Jaquet
IT University Copenhagen



Today's schedule

Goal

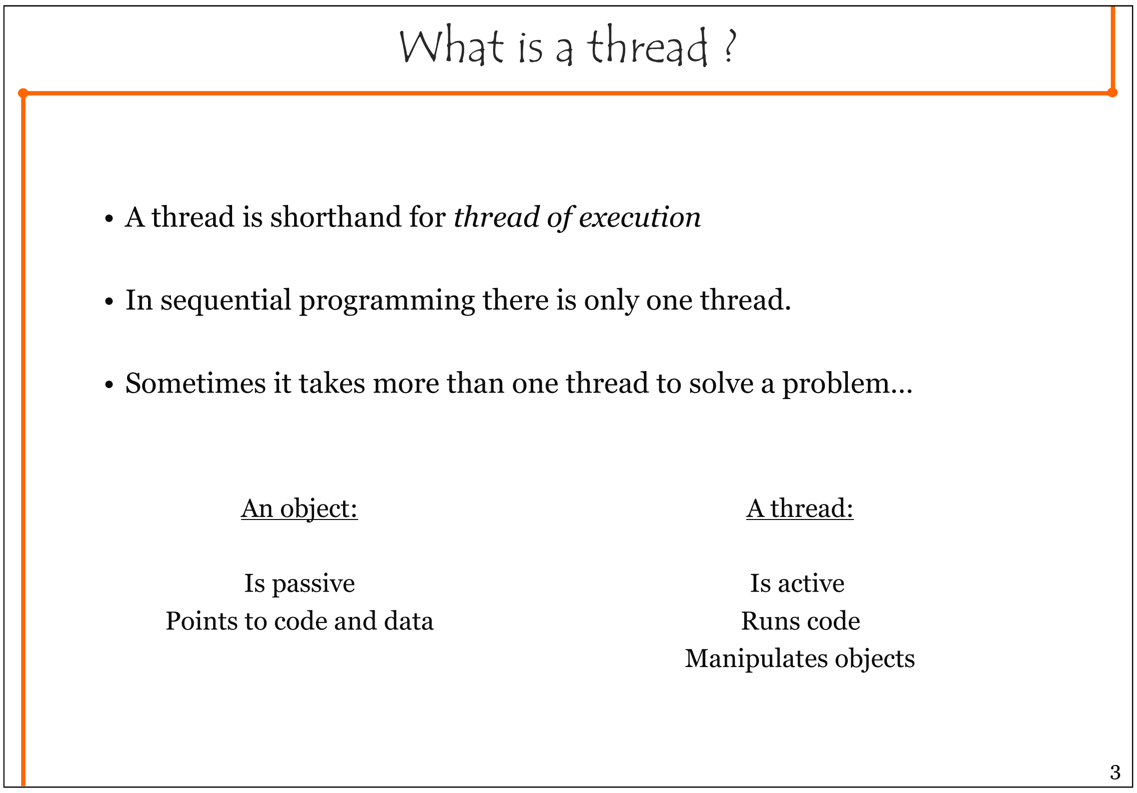
- To give a basic understanding of "threads", how they are used and what some of the challenges are.

Contents

- What is a thread ?
- How are threads created and used ?
- Issues to consider when using threads
  - Communicating between threads
  - Scheduling threads
  - Synchronization
  - Deadlocks and other pitfalls to avoid

2

Thread programming is a big topic and obtaining a thorough understanding of the challenges and possible approaches is not within the scope of this OOP course.



What is a thread ?

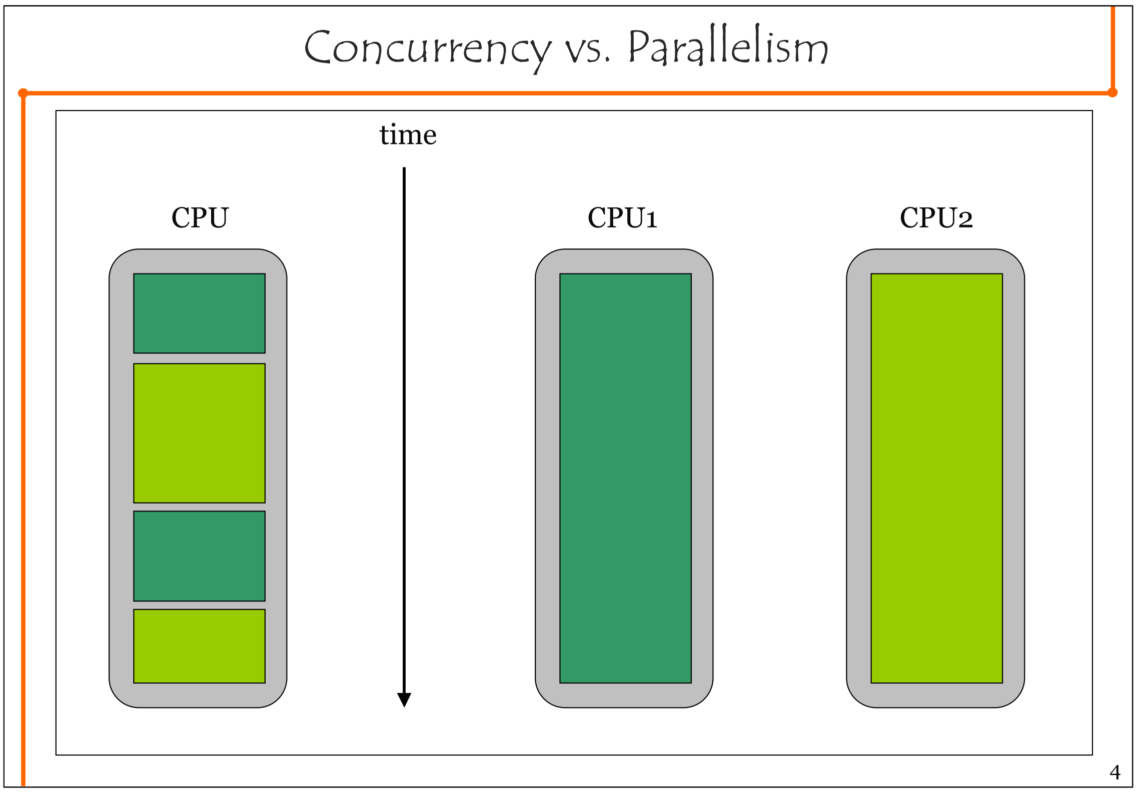
- A thread is shorthand for *thread of execution*
- In sequential programming there is only one thread.
- Sometimes it takes more than one thread to solve a problem…

An object:	A thread:
Is passive	Is active
Points to code and data	Runs code
	Manipulates objects

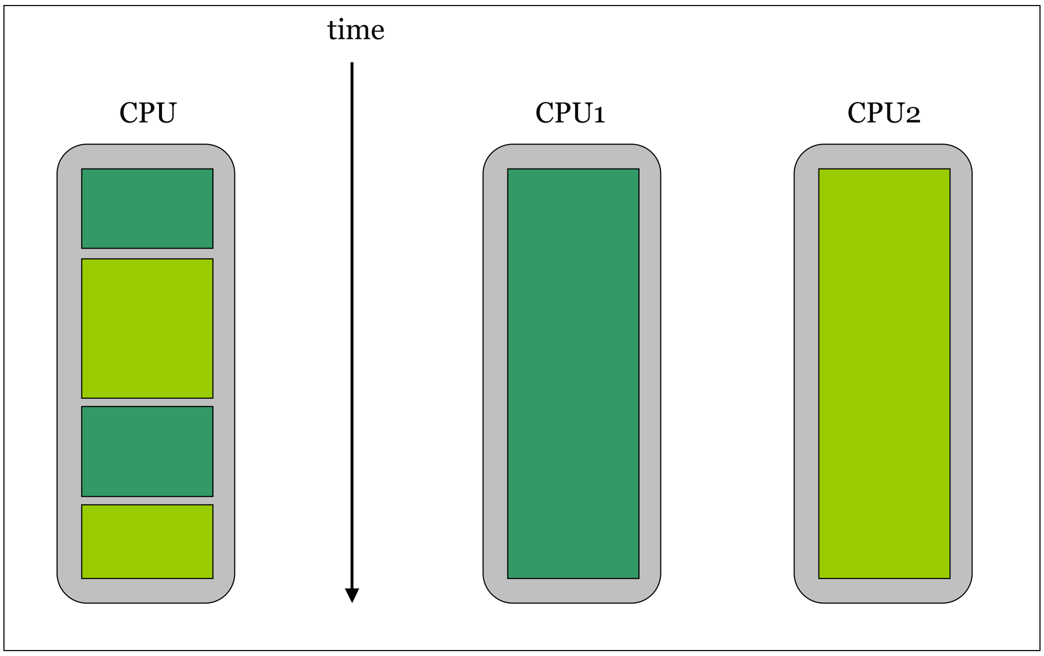
3

When running a Java program the JVM will create a thread that calls main() and executes it sequentially. When the end of main() is reached the thread will die and the program will exit if no other threads exist.

Secretary.java is an example of a problem that is hard to solve elegantly with one thread. It can either halt, “busy-wait” or be quite inflexible



Concurrency vs. Parallelism



4

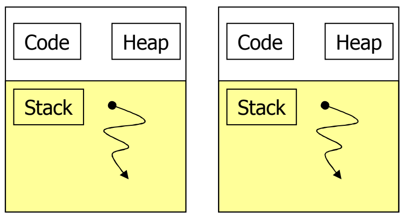
Two programs cannot execute simultaneously using one CPU. But we can create the illusion of parallelism by using multiple threads and assigning them slices of CPU time to run in.

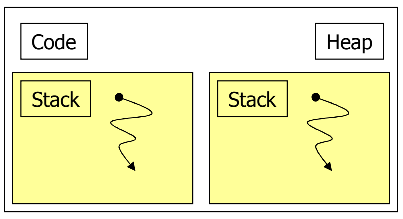
The management of time and switching of threads takes time as well, so having many threads is expensive CPU wise. If it is done excessively it may lead to **thread thrashing** where not much time is left for executing the code.

Multitasking vs. Multithreading

- *Multitasking* is a computers ability to do multiple program executions concurrently
- Each program has its own code, stack and heap

- *Multithreading* is having multiple threads in a program
- All threads access the same code and share the same heap
- Each thread has its own stack for local variables and method arguments





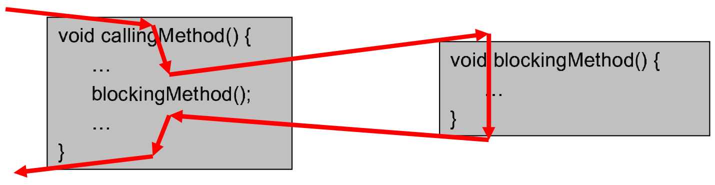
5

Note that because objects and their fields are located on the heap they are shared among all threads.

The left figure shows two programs or processes that each have a thread. The figure to the right shows one process with two threads.

Blocking vs. non-blocking method calls

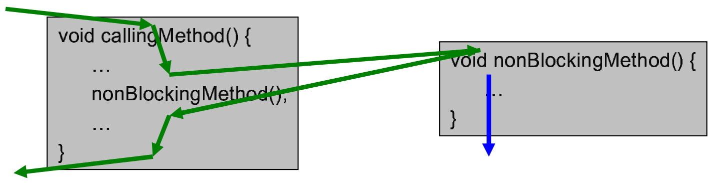
A blocking (synchronous) method call:



```
void callingMethod() {
    …
    blockingMethod();
    …
}
```

```
void blockingMethod() {
    …
}
```

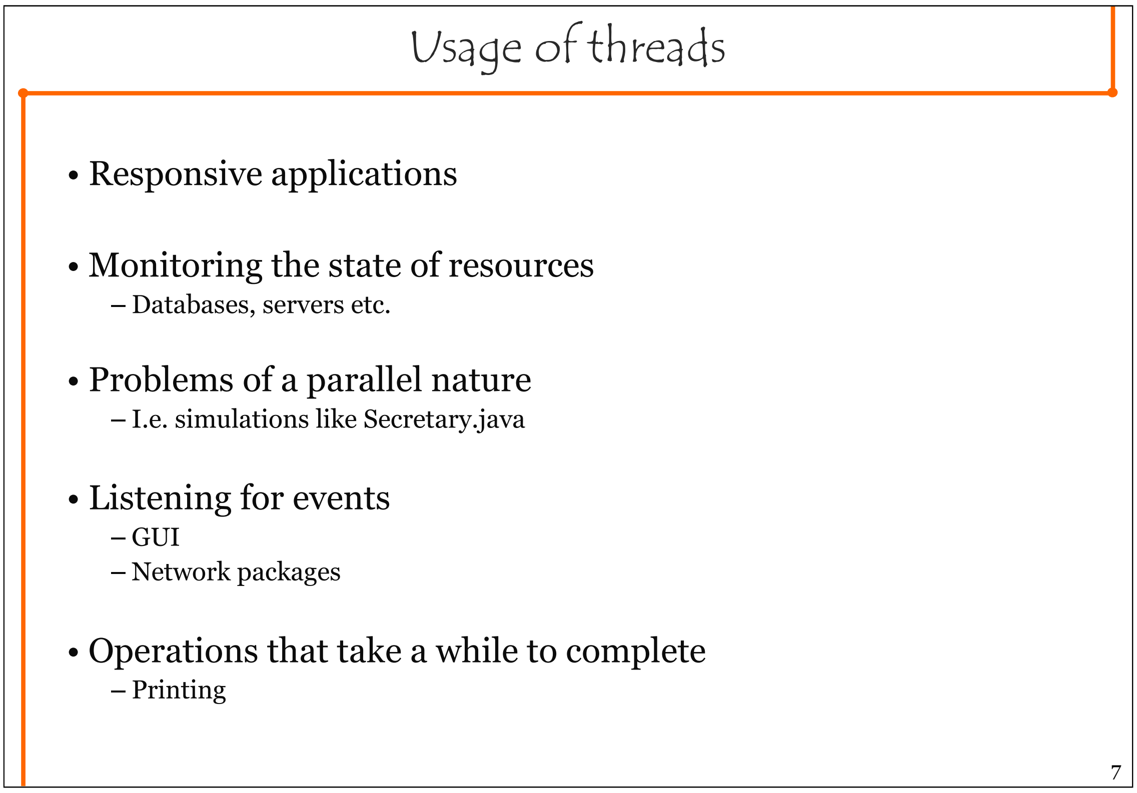
A non-blocking (asynchronous) method call:



```
void callingMethod() {
    …
    nonBlockingMethod();
    …
}
```

```
void nonBlockingMethod() {
    …
}
```

A new thread executes the method

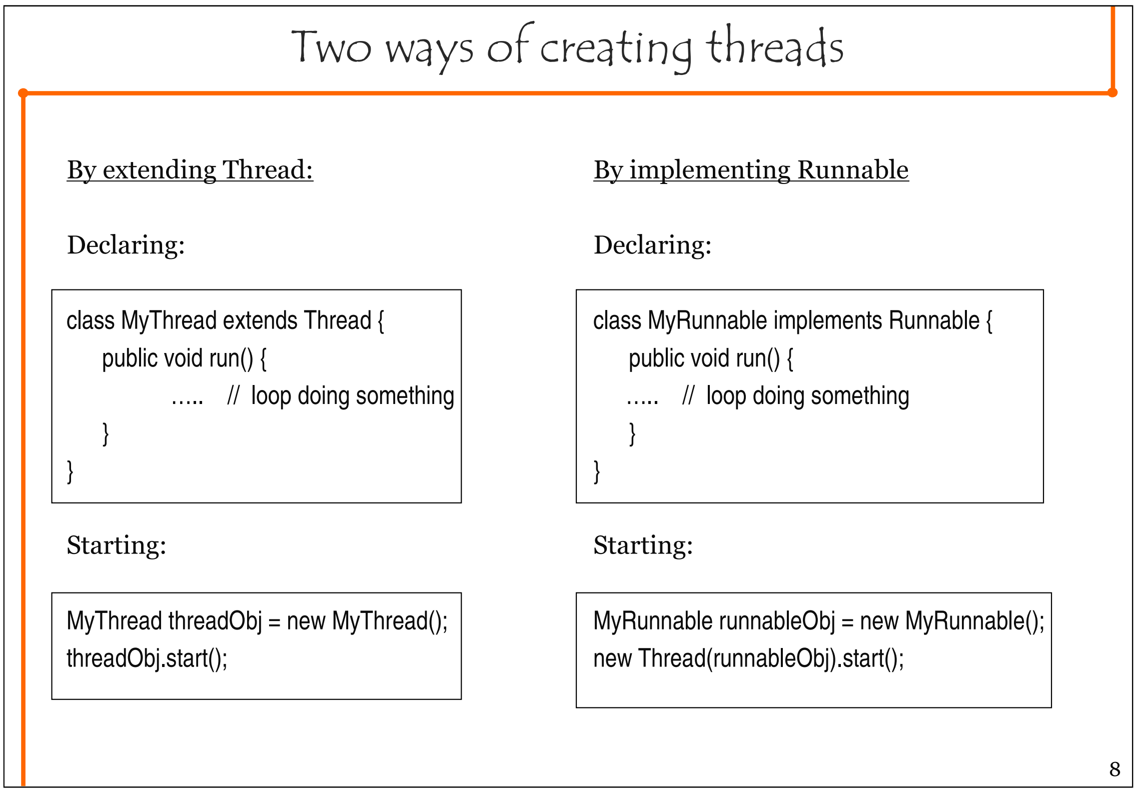


Usage of threads

- Responsive applications
- Monitoring the state of resources
  - Databases, servers etc.
- Problems of a parallel nature
  - I.e. simulations like Secretary.java
- Listening for events
  - GUI
  - Network packages
- Operations that take a while to complete
  - Printing

7

Concurrent programs are typically written for one of two reasons: to improve performance, or to satisfy an inherently concurrent specification.



Two ways of creating threads

By extending Thread:

Declaring:

```
class MyThread extends Thread {
     public void run() {
               .....   // loop doing something
     }
}
```

Starting:

```
MyThread threadObj = new MyThread();
threadObj.start();
```

By implementing Runnable

Declaring:

```
class MyRunnable implements Runnable {
     public void run() {
     .....   // loop doing something
     }
}
```

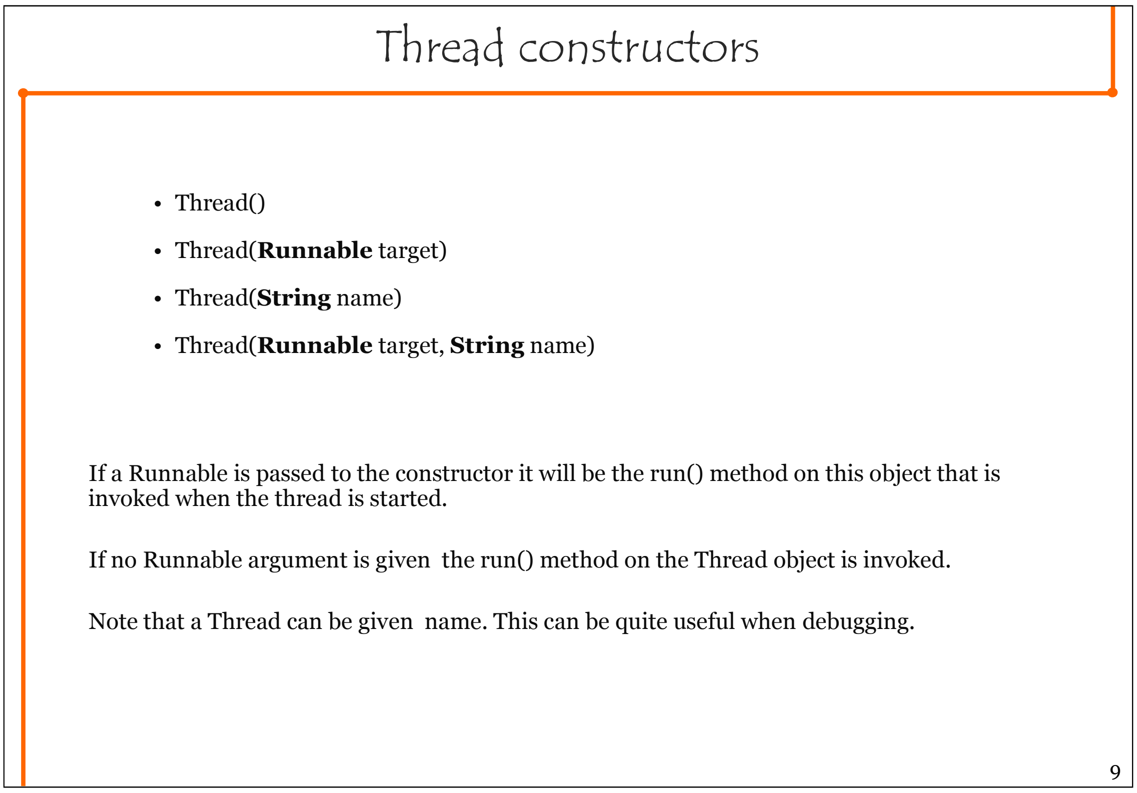
Starting:

```
MyRunnable runnableObj = new MyRunnable();
new Thread(runnableObj).start();
```

8

If you implement Runnable and write a run() method here, you have to pass the Runnable object to the constructor of a Thread object, and this run-method will be executed when the thread is started. If you want your class to inherit some other class than Thread, implementing the Runnable interface is the solution.

Also if run() is the only method on Thread that is overridden and no modifications or enhancements to the fundamental behavior of the class are made, it is recommended to implement Runnable instead of extending Thread.



Thread constructors

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

If a Runnable is passed to the constructor it will be the run() method on this object that is invoked when the thread is started.

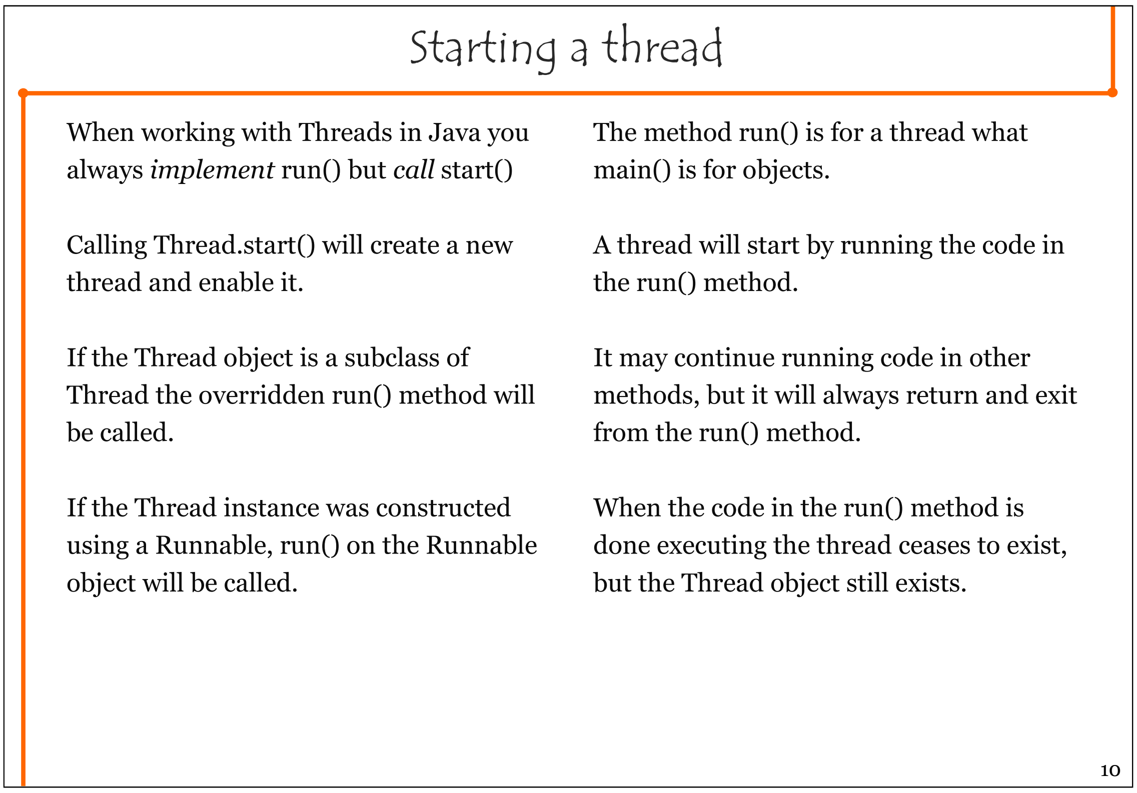
If no Runnable argument is given the run() method on the Thread object is invoked.

Note that a Thread can be given name. This can be quite useful when debugging.

9

Note that the number of threads doesn’t necessarily correspond to the number of Thread objects !

We don’t cover ThreadGroups in this lecture, but besides from the above constructors there are 3 more that are almost the same as last 3 above. The only difference is that they also take a ThreadGroup as an argument and adds the Thread object created to this group.



Starting a thread

When working with Threads in Java you always *implement* run() but *call* start()

Calling Thread.start() will create a new thread and enable it.

If the Thread object is a subclass of Thread the overridden run() method will be called.

If the Thread instance was constructed using a Runnable, run() on the Runnable object will be called.

The method run() is for a thread what main() is for objects.

A thread will start by running the code in the run() method.

It may continue running code in other methods, but it will always return and exit from the run() method.

When the code in the run() method is done executing the thread ceases to exist, but the Thread object still exists.

10

Enabling a thread means scheduling the run() method to be executed when CPU time is available.

Attempting to call start() more than once on a Thread before the running thread is done will give an IllegalThreadStateException

If you try calling run() directly on a Thread you will see that it is executed like it was any other synchronous method and doesn't get a thread of its own. The start() method is the key to executing the code in run() in a separate thread of execution.

A ThreadPrinterExample

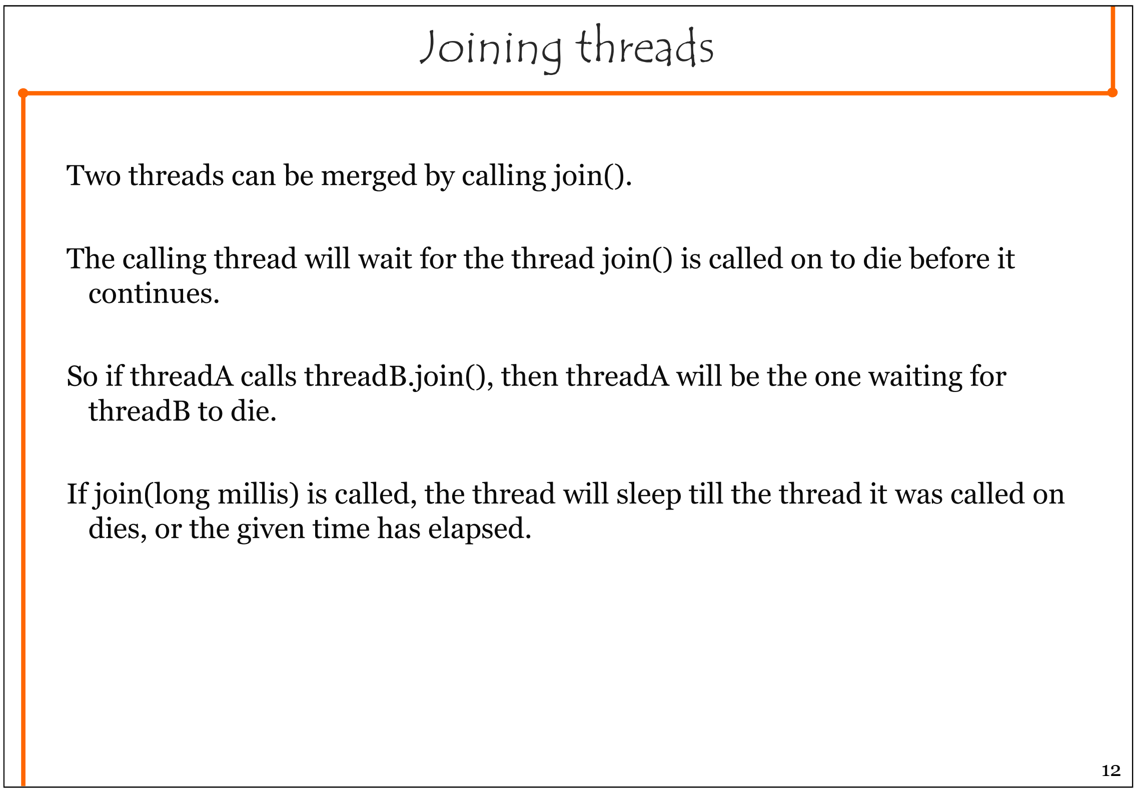
The two different ways of starting threads are shown to the right:

What will happen when the main() method below is executed?

```
class Print implements Runnable {
   public void run() {
            for(int i = 0; i < 50; i++)
            System.out.print("R");
   }

   public void start() {
            System.out.println("Hi there!");
   }
}
```

```
class ThreadPrinterExample {
   public static void main(String[] args) {
            Print p0 = new Print();
            Printer p1 = new Printer('|');
            Printer p2 = new Printer('-');
            p1.start();
            p2.start();
            new Thread(p0).start();
   }
}
```

```
class Printer extends Thread {
   char character;

   Printer(char character) { this.character = character; }

   public void run () {
            for(int i = 0; i < 50; i++)
            System.out.print(character);
   }
}
```



Joining threads

Two threads can be merged by calling join().

The calling thread will wait for the thread join() is called on to die before it continues.

So if threadA calls threadB.join(), then threadA will be the one waiting for threadB to die.

If join(long millis) is called, the thread will sleep till the thread it was called on dies, or the given time has elapsed.

12

If a timeout occurs an InterruptedException is thrown.

Daemon threads

Normally a program will continue to run as long as there are any live threads.

An exception to this is daemon threads.

A daemon thread is made by setting Thread.setDaemon(true)

If a new thread is created by a daemon thread it will itself be a daemon thread.

Daemon threads are usually used for processes that run in the background for an extended period of time. I.e. monitoring state of resources or collecting garbage.

13

Scheduling

Thread scheduling determines how threads are allocated CPU time.
Some approaches are:

Preemptive scheduling – the scheduler pauses (preempts) the running thread to allow others to execute.

Non-preemptive scheduling – a running thread is never interrupted. The running thread must yield control to the CPU.

Both types can cause *livelocks* where threads are assigned CPU time but never progress.

Non-preemptive scheduling may also cause thread *starvation* where low-priority threads never are assigned CPU time.

Priorities

In Java scheduling is preemptive and based on priorities of threads.

When the scheduler assigns control to a thread it generally favours the one with the highest priority.

The methods used to influence thread-scheduling with priorities are:

```
setPriority(int newPriority)
getPriority()
Thread.yield()
```

The yield() method will voluntarily give up the control and let the scheduler activate another thread.

Inappropriate use of priorities might lead to *thread-starvation*

15

When running Java under Windows selfish thread behavior is fought with a strategy known as **time slicing**. Time slicing comes into play when there are multiple “runnable" threads of equal priority and those threads are the highest priority threads competing for the CPU.

If all threads have the same priority and no one yields/ waits its would be like non-preemptive scheduling if it wasn’t for Windows.

In Java priorities are between 1 and 10 and the default priority for a thread is 5.

When a new Thread object is created its will by default get the same priority as the current thread.

The challenge of sharing data

Local variables and method arguments are placed on the stack and are not available to other threads.

But objects and their fields are located in the heap and can be accessed by multiple threads.

When multiple threads read, modify and update data simultaneously a dependency on timing is introduced.

A situation may occur where the values that are updated depend on the timing. When different threads *race* to store their values it is called a *race condition*.

16

An example of a race condition

Imagine a bank account with an initial balance of 500.

Two transactions A and B are requested.

Transaction A deposits 200 units.

Transaction B withdraws 100 units.

What is the balance after the two transactions are done ?

The answer should be 600, but it will depend on the timing...

Thread A: // +200

- Get balance (500)
- Calculate new balance (700)
- Save new balance (700)

Thread B: // -100

- Get balance (500)
- Calculate new balance (400)
- Save new balance (400)

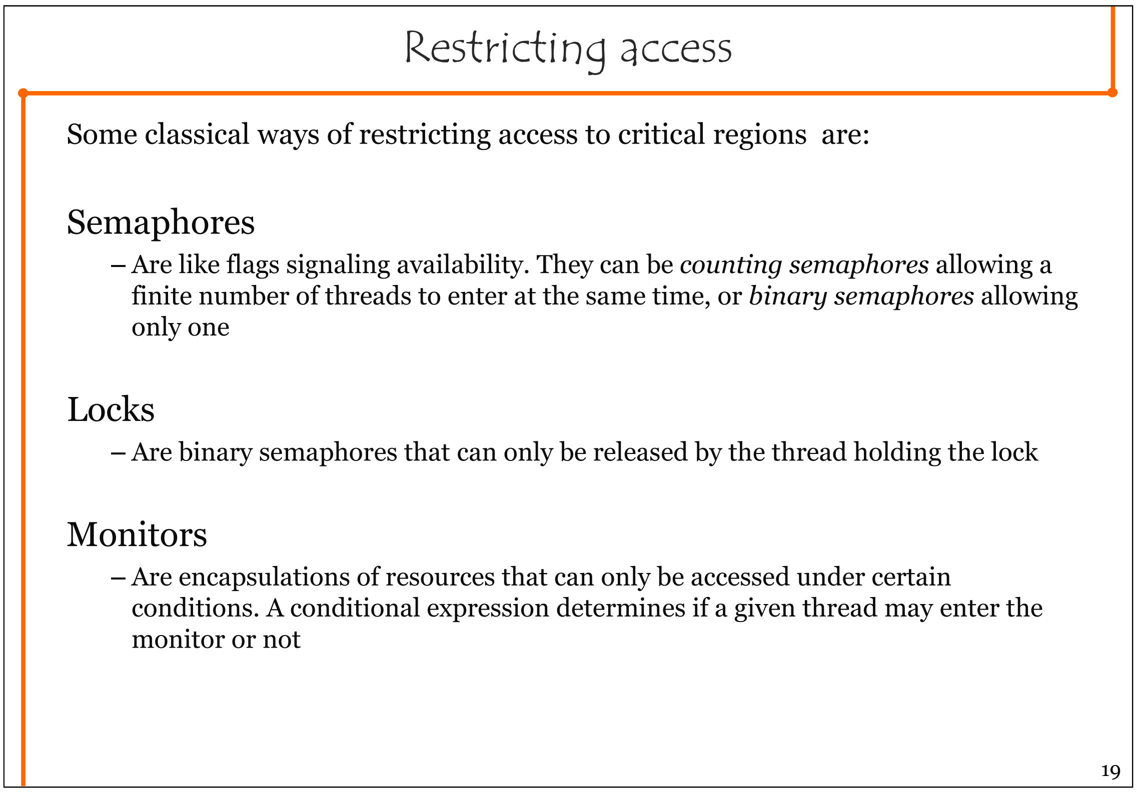
17

So what we need is..

A semaphore !

18

A semaphore is like a flag or signal restricting access to a critical area. If it is a *binary semaphore* it has two positions – one meaning “go” and one meaning “stop”.



Restricting access

Some classical ways of restricting access to critical regions are:

Semaphores

- Are like flags signaling availability. They can be *counting semaphores* allowing a finite number of threads to enter at the same time, or *binary semaphores* allowing only one

Locks

- Are binary semaphores that can only be released by the thread holding the lock

Monitors

- Are encapsulations of resources that can only be accessed under certain conditions. A conditional expression determines if a given thread may enter the monitor or not

19

Semaphores have a non-negative integer value and counter. If it is a binary semaphore the value is 1 and the counter is always 0 or 1.

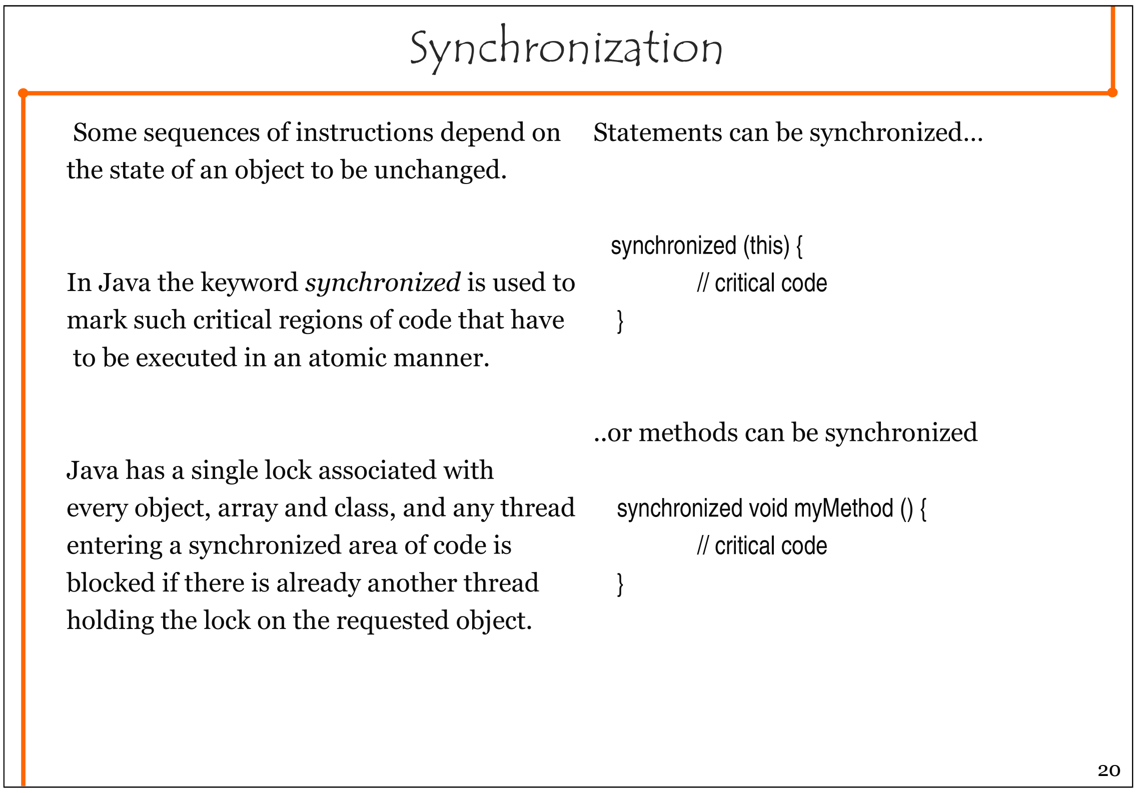
A pseudo code example of a the use of a semaphore for a boat rental booth with 6 boats available could be:

semaphore boat = 6; … acquire(boat); useBoat(); release(boat);

When a boat is acquired the semaphores counter is decremented, when it is released it is incremented. If someone calls acquire(boat) and the counter is 0 they will have to wait till it is incremented by someone else calling release(boat).

Monitors are a synchronization mechanism based in some sense on data abstraction. A monitor encapsulates the representation of a shared resource and provides operations that are the only way of manipulating it. In other words, a monitor contains variables that represent the state of the resource and procedures that implement operations on the resource; a process or thread can access the monitor's variables only by calling one of its procedures. Mutual exclusion among the procedures of a monitor is guaranteed; execution of different procedures, or two calls to the same procedure, cannot overlap. Conditional synchronization is provided by condition variables

Not that the definitions above relate to concurrency issues in general and do not correspond directly to the mechanisms available in Java ! They will be described in following slides.



Synchronization

Some sequences of instructions depend on the state of an object to be unchanged.

In Java the keyword *synchronized* is used to mark such critical regions of code that have to be executed in an atomic manner.

Java has a single lock associated with every object, array and class, and any thread entering a synchronized area of code is blocked if there is already another thread holding the lock on the requested object.

Statements can be synchronized…

```
synchronized (this) {
        // critical code
}
```

..or methods can be synchronized

```
synchronized void myMethod () {
        // critical code
}
```

20

A critical section is a block or a method identified with the **synchronized** keyword. Java associates a lock with every object that has synchronized code.

Note that if someone writes a subclass of your class and overrides a synchronized method they don’t have to make it synchronized as it is just “syntactic sugar”

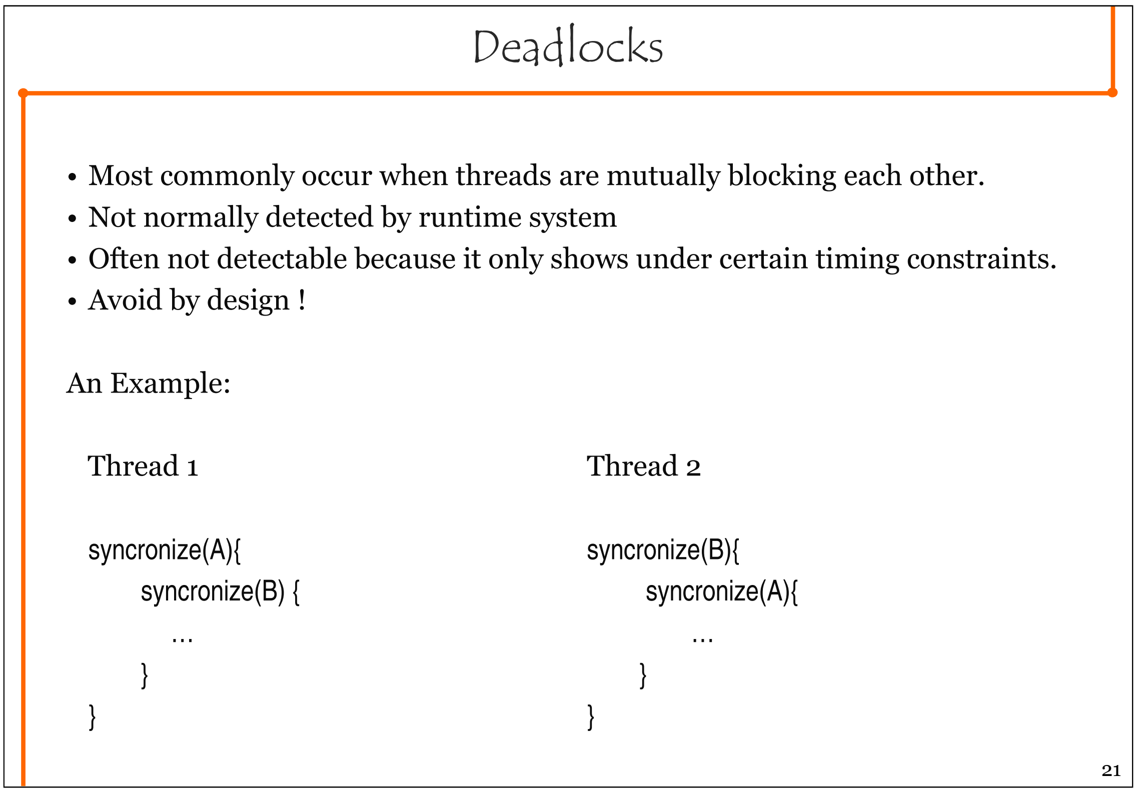
```
synchronized void myMethod () {
  // critical code
}
```

Is the same as writing:

```
void myMethod () {
  synchronized (this) {
    // critical code
  }
}
```

Constructors and initializers cannot be synchronized.

Fields in an object cannot be synchronized, but an array can be passed as the object for a synchronized (array) { } clause.



Deadlocks

- Most commonly occur when threads are mutually blocking each other.
- Not normally detected by runtime system
- Often not detectable because it only shows under certain timing constraints.
- Avoid by design !

An Example:

Thread 1

```
syncronize(A){
     syncronize(B) {
         …
     }
}
```

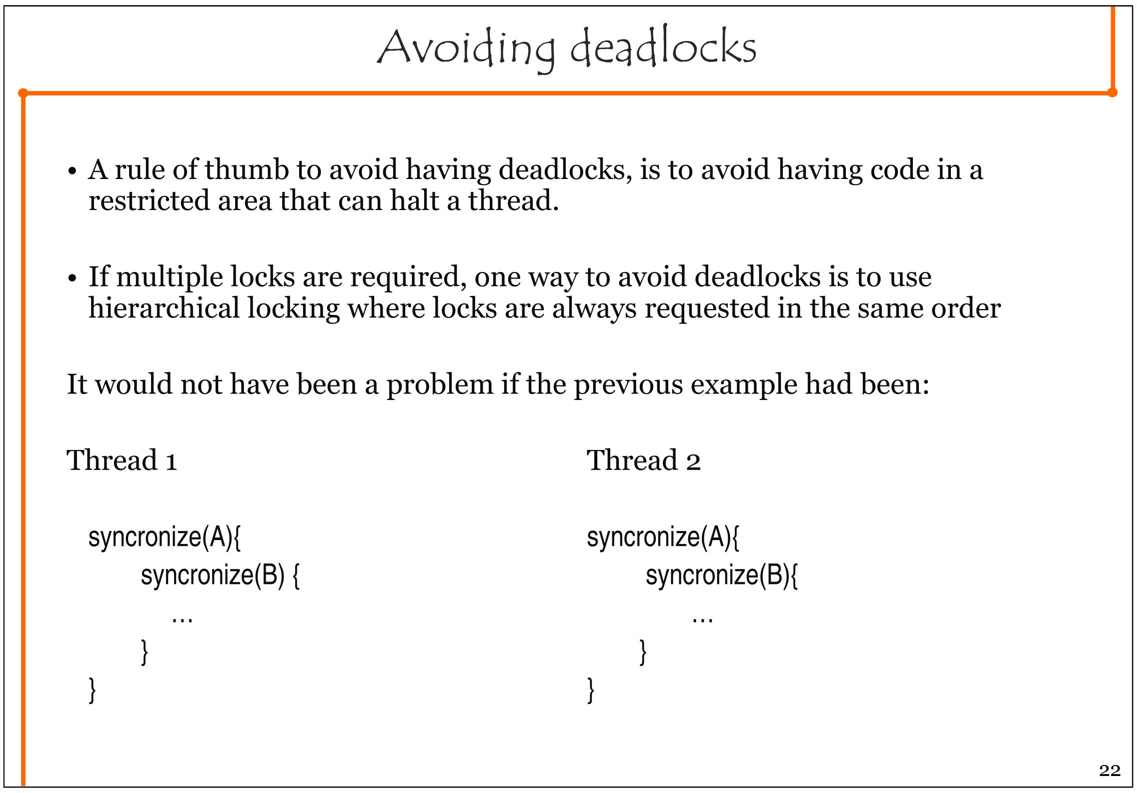
Thread 2

```
syncronize(B){
     syncronize(A){
         …
     }
}
```

21

Note that not just symmetric blocking like the above synchronization pattern of A waiting for B and B waiting for A is a problem. A deadlock can also be caused by a cyclic synchronization pattern like A waiting for B, B waiting for C and C waiting for A.

Another type of deadlocks is when two threads are waiting to join each other, i.e A.join(B) and B.join(A) effective at the same time.



Avoiding deadlocks

- A rule of thumb to avoid having deadlocks, is to avoid having code in a restricted area that can halt a thread.

- If multiple locks are required, one way to avoid deadlocks is to use hierarchical locking where locks are always requested in the same order

It would not have been a problem if the previous example had been:

Thread 1

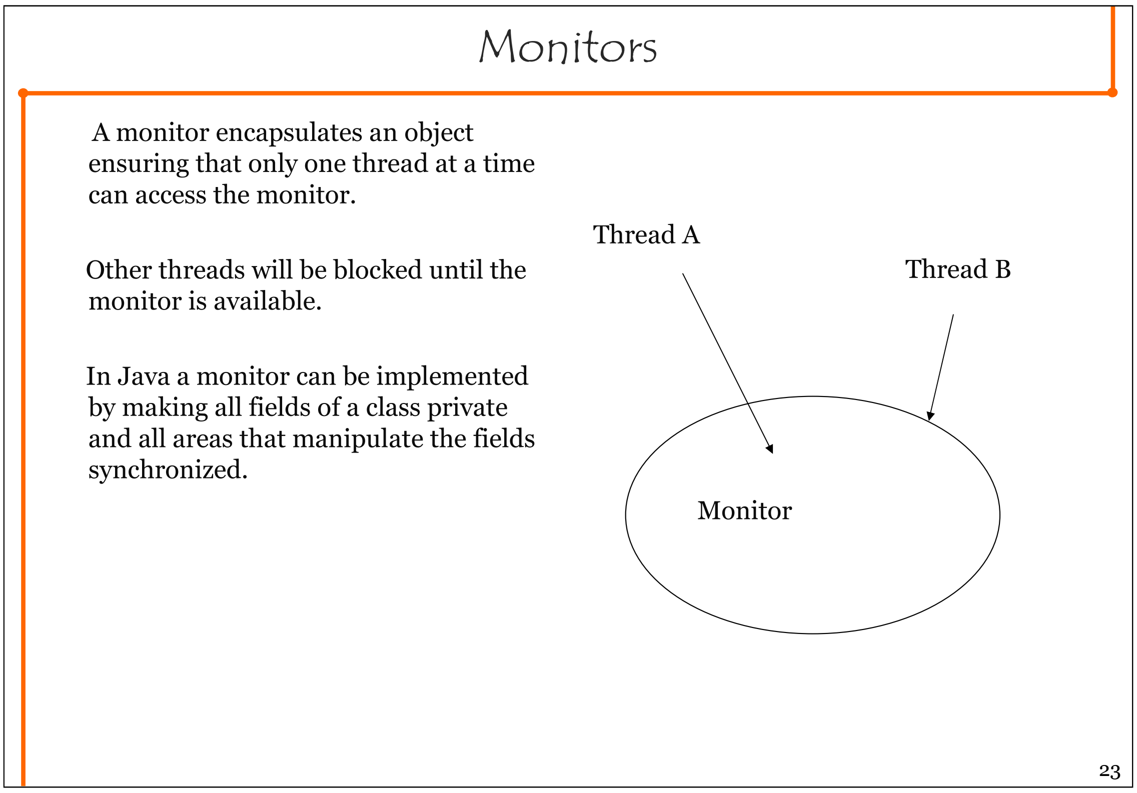
```
syncronize(A){
        syncronize(B) {
              …
        }
}
```

Thread 2

```
syncronize(A){
         syncronize(B){
                …
        }
}
```

22

Halting a thread can be caused in many different ways. It can be calling wait() or join() or it can be waiting for a database connection or network package.

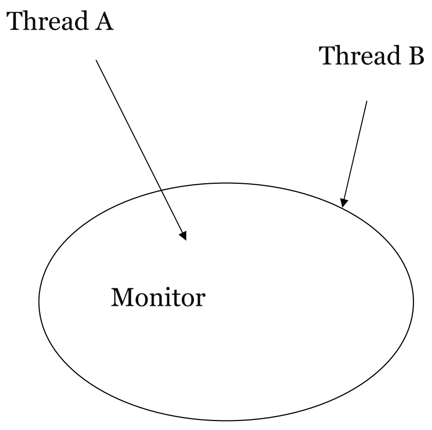


Monitors

A monitor encapsulates an object ensuring that only one thread at a time can access the monitor.

Other threads will be blocked until the monitor is available.

In Java a monitor can be implemented by making all fields of a class private and all areas that manipulate the fields synchronized.



23

The difference between classical monitors and Java's built in monitors, is that there is no condition variable associated with a monitor in Java, and that the threads waiting to access the monitor will be notified in random order, independent on priority and who has waited the longest.

Wait & Notify

The following methods on the Object class can be used for monitors:

void wait(): Causes the calling thread to wait until another thread calls notify() or notifyAll() on the object wait() was called on. A thread can only call wait() on an object o if it has the lock on o. This means wait() can only be called inside a synchronized block of code.

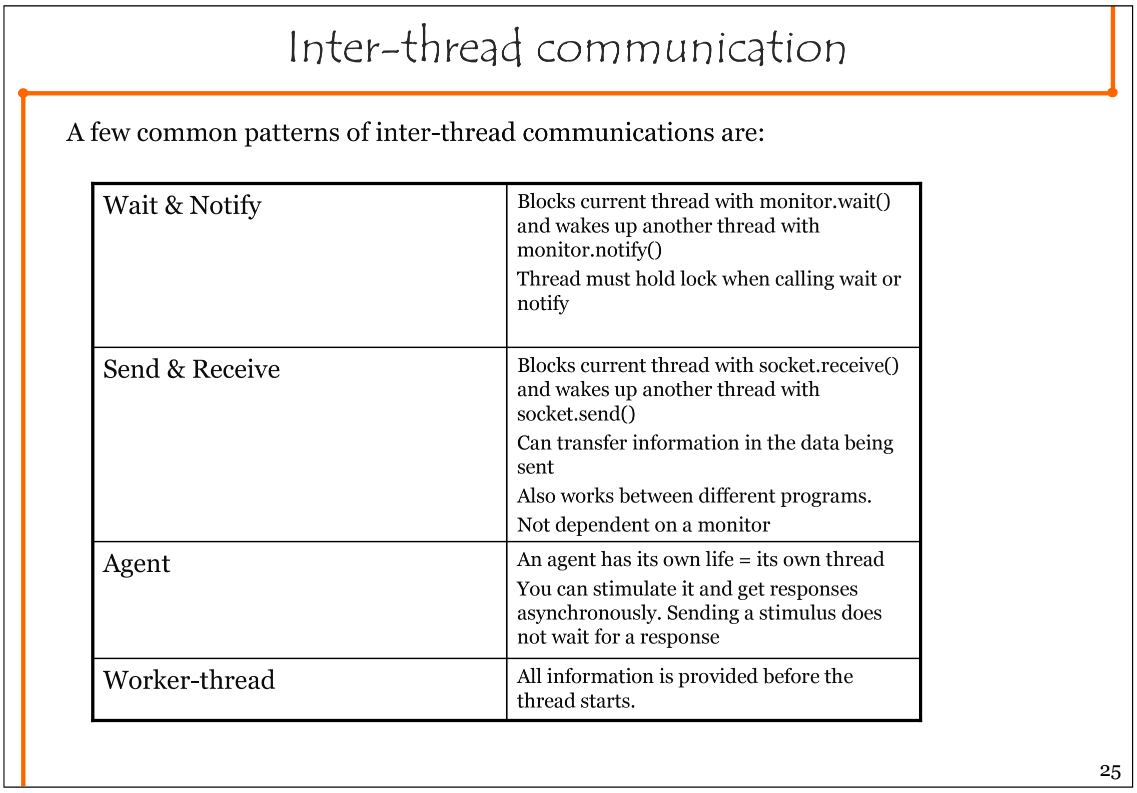
void wait(long timeout): As wait() but with a maximum limit of time to wait.

void notify(): Wakes a thread waiting for the object. If more than one thread is waiting only one is notified.

void notifyAll(): Wakes all threads waiting for an object.

24

Note that a higher priority thread is not favored over a lower priority one in the selection of which should receive a notification !



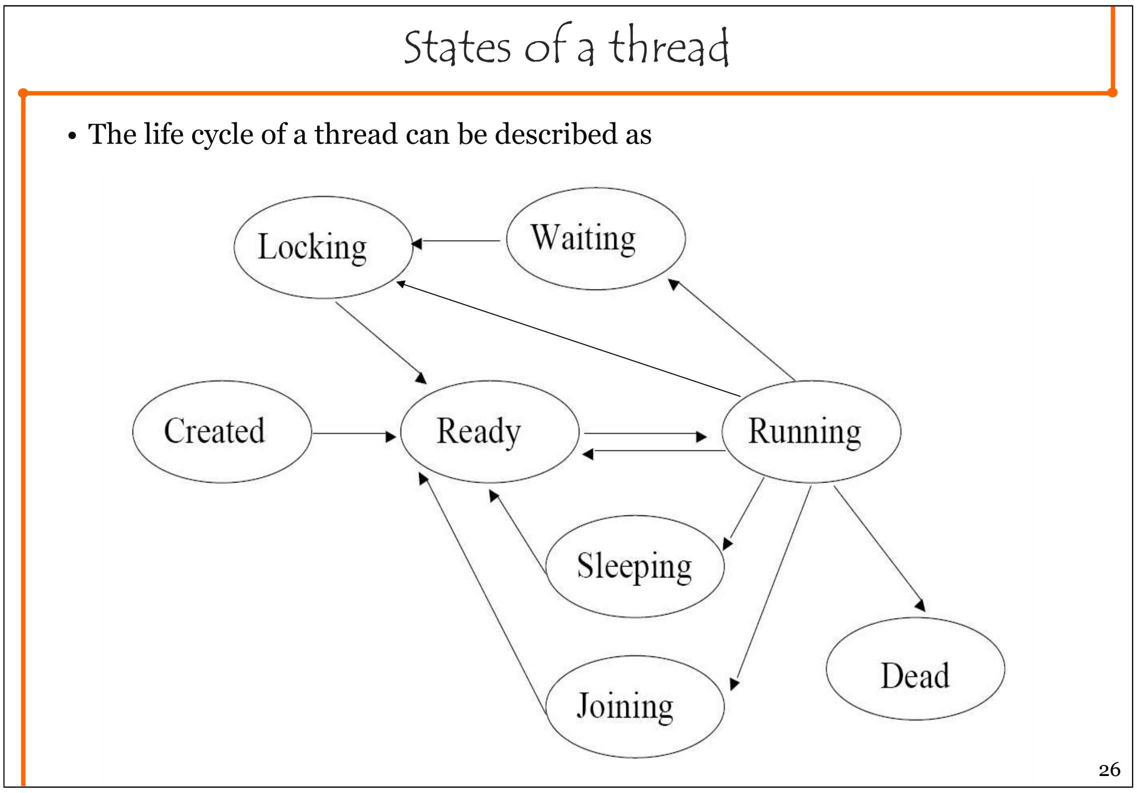
Inter-thread communication

A few common patterns of inter-thread communications are:

Wait & Notify	Blocks current thread with monitor.wait() and wakes up another thread with monitor.notify() Thread must hold lock when calling wait or notify
Send & Receive	Blocks current thread with socket.receive() and wakes up another thread with socket.send() Can transfer information in the data being sent Also works between different programs. Not dependent on a monitor
Agent	An agent has its own life = its own thread You can stimulate it and get responses asynchronously. Sending a stimulus does not wait for a response
Worker-thread	All information is provided before the thread starts.

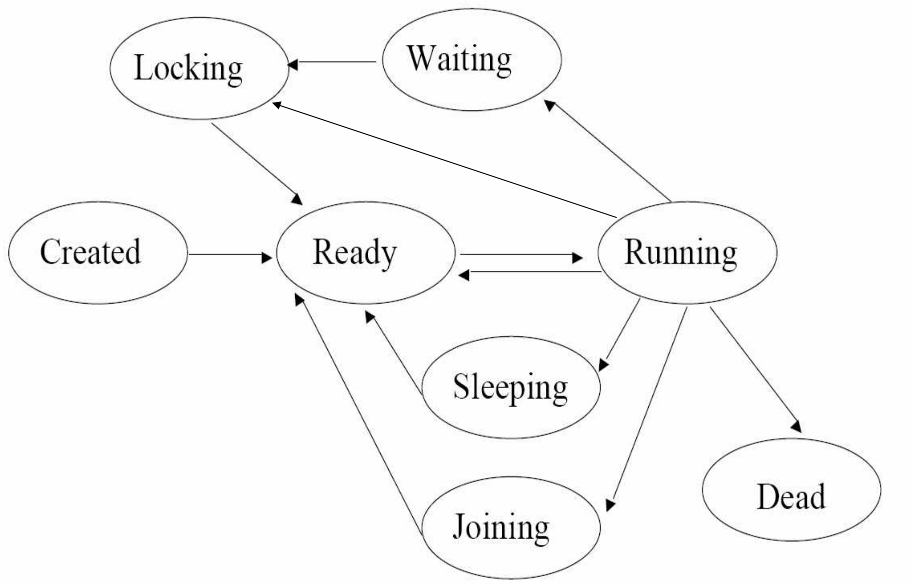
25

Wait & notify might be a producer/consumer scenario. In general it is used when a thread needs to wait for a condition to become true or a resource to be available.



States of a thread

- The life cycle of a thread can be described as



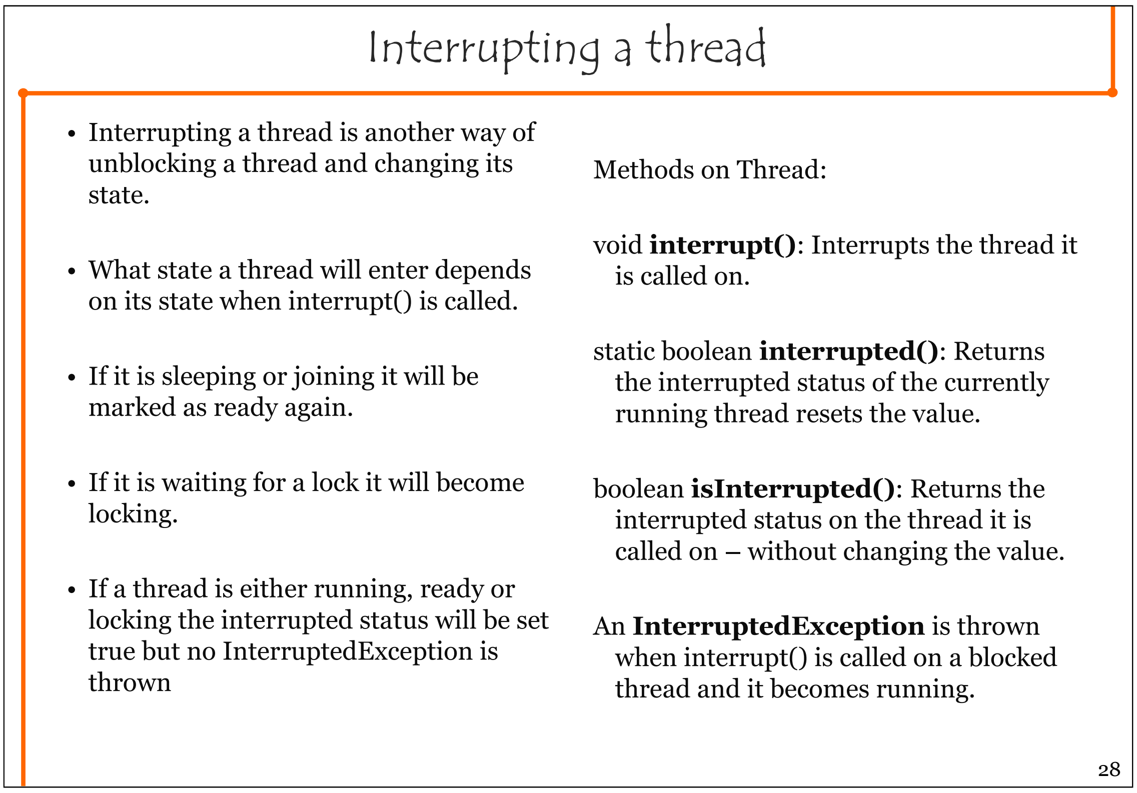
26

Ready:	Ready to run (Enabled)
Running:	The tread currently assigned CPU time
Sleeping:	Waiting for a timeout
Joining:	Waiting for another thread to die
Locking:	Waiting for a lock on an object, or for I/O.
Waiting:	Waiting for a notification

Blocking and unblocking

Thread state	Caused by	Changed by
Sleeping	Thread.sleep()	Time passed
Joining	otherThread.join()	otherThread dies
Waiting	object.wait()	object.notify()
Locking	synchronized(object) or waiting for i.e. I/O	Object gets unlocked or resource available

27



Interrupting a thread

- Interrupting a thread is another way of unblocking a thread and changing its state.
- What state a thread will enter depends on its state when interrupt() is called.
- If it is sleeping or joining it will be marked as ready again.
- If it is waiting for a lock it will become locking.
- If a thread is either running, ready or locking the interrupted status will be set true but no InterruptedException is thrown

Methods on Thread:

void **interrupt()**: Interrupts the thread it is called on.

static boolean **interrupted()**: Returns the interrupted status of the currently running thread resets the value.

boolean **isInterrupted()**: Returns the interrupted status on the thread it is called on – without changing the value.

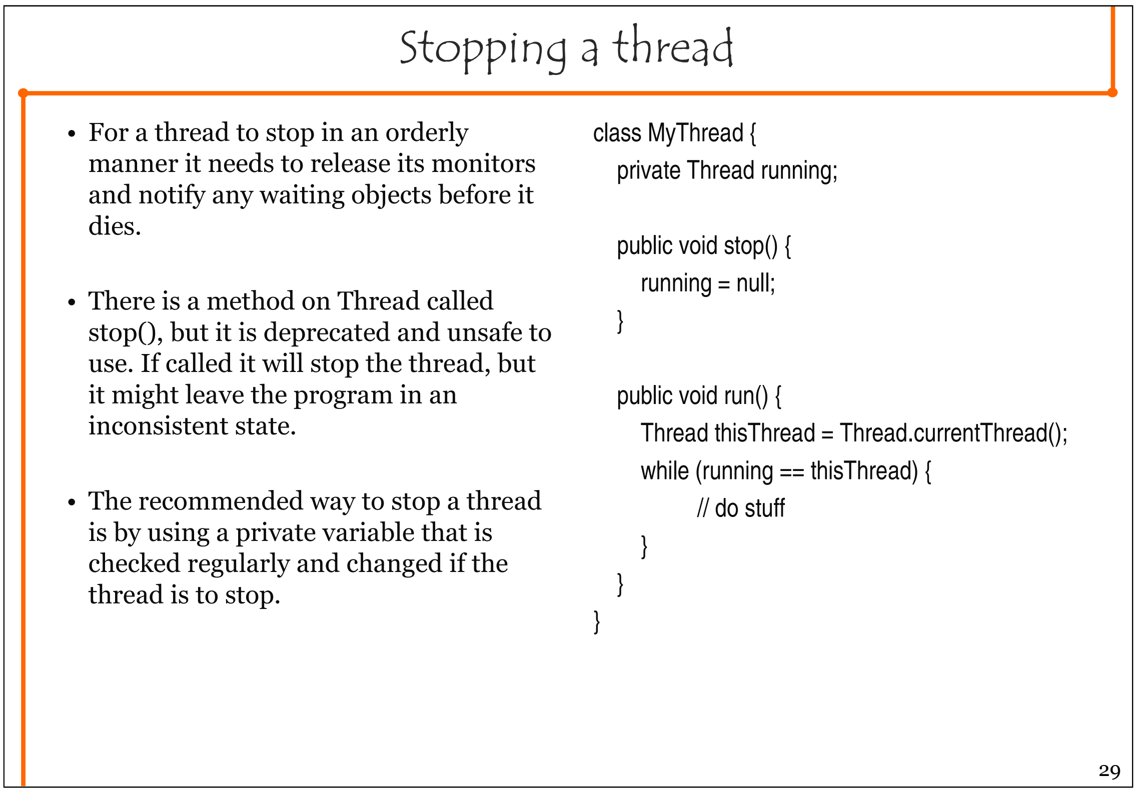
An **InterruptedException** is thrown when interrupt() is called on a blocked thread and it becomes running.

28

The *Thread.interrupt* method can seem a bit confusing. Despite what its name may imply, the method does not interrupt a running thread.

What the method actually does is to throw an interrupt if the thread is blocked, so that it exits the blocked state. More precisely, if the thread is blocked at one of the methods *Object.wait*, *Thread.join*, or *Thread.sleep*, it receives an *InterruptedException*, thus terminating the blocking method prematurely.

When a blocked thread is interrupted it becomes enabled and an InterruptedException is thrown when it becomes running (when it is allocated CPU time).



Stopping a thread

- For a thread to stop in an orderly manner it needs to release its monitors and notify any waiting objects before it dies.
- There is a method on Thread called stop(), but it is deprecated and unsafe to use. If called it will stop the thread, but it might leave the program in an inconsistent state.
- The recommended way to stop a thread is by using a private variable that is checked regularly and changed if the thread is to stop.

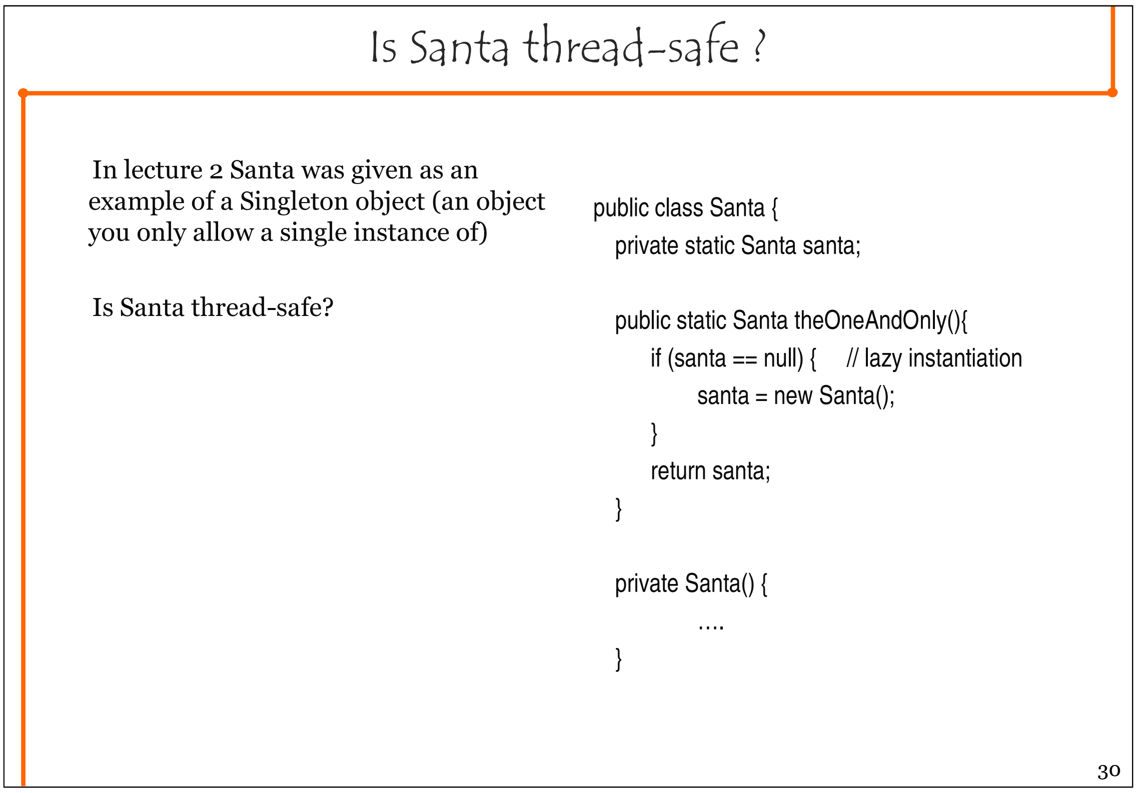
```
class MyThread {
  private Thread running;

  public void stop() {
    running = null;
  }

  public void run() {
    Thread thisThread = Thread.currentThread();
    while (running == thisThread) {
         // do stuff
    }
  }
}
```

29

Other deprecated and unsafe methods on Thread are suspend() and resume(). You can read more about them and why they should be avoided in the Java APIs documentation of the Tread class.

Note that if a thread is blocked or waiting it may take a long time before it stops. One way to solve this is by sending an InterruptedException, if the thread is programmed to handle that:

```
public void stop() {
                Thread tmp = running;
                running = null;
                tmp.interrupt();
}
```



Is Santa thread-safe ?

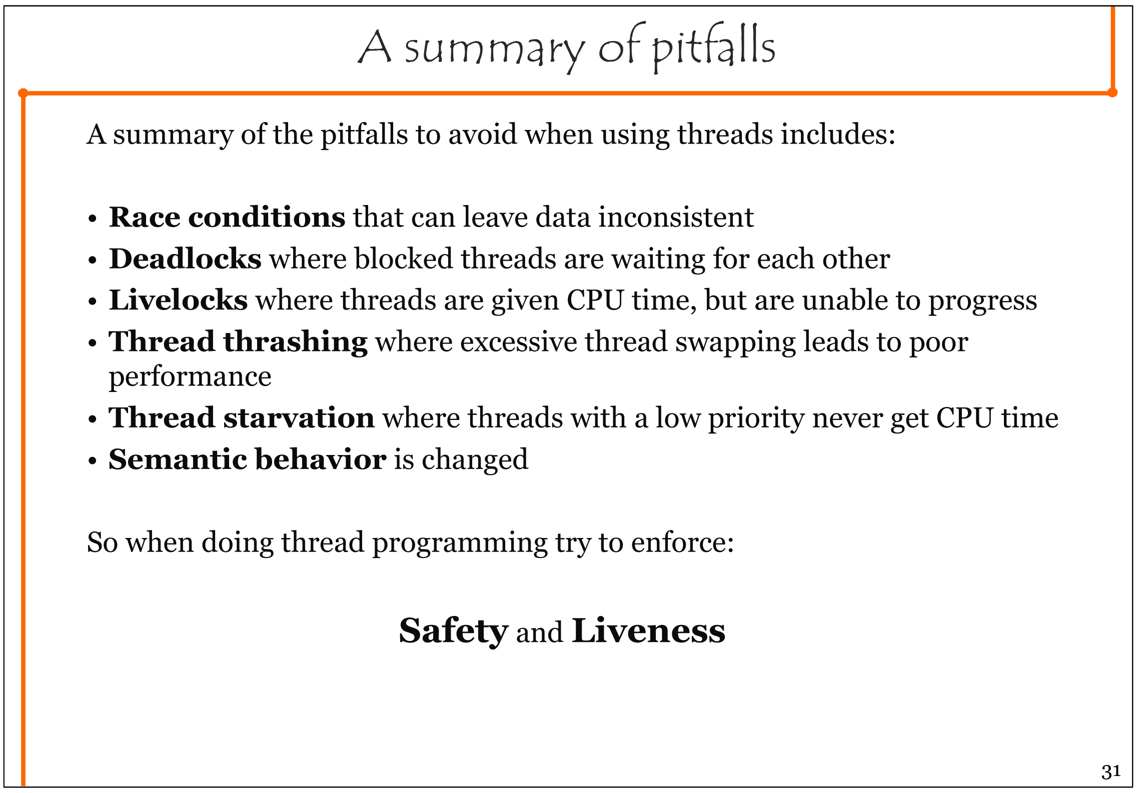
In lecture 2 Santa was given as an example of a Singleton object (an object you only allow a single instance of)

Is Santa thread-safe?

```
public class Santa {
   private static Santa santa;

   public static Santa theOneAndOnly(){
        if (santa == null) {     // lazy instantiation
              santa = new Santa();
        }
        return santa;
   }

   private Santa() {
              ….
   }
```

30

Note that constructors and initializers cannot be synchronized



A summary of pitfalls

A summary of the pitfalls to avoid when using threads includes:

- **Race conditions** that can leave data inconsistent
- **Deadlocks** where blocked threads are waiting for each other
- **Livelocks** where threads are given CPU time, but are unable to progress
- **Thread thrashing** where excessive thread swapping leads to poor performance
- **Thread starvation** where threads with a low priority never get CPU time
- **Semantic behavior** is changed

So when doing thread programming try to enforce:

Safety and **Liveness**

31

Deadlock occurs when some threads are blocked to acquire resources held by other blocked threads. A deadlock may arise due to a dependence between two or more threads that request resources and two or more threads that hold those resources. In Java, thread deadlock can occur:
1. When two threads call Thread.join() on one another.
2. When two threads use nested synchronized blocks to lock two objects and the blocks lock the same objects in different order.

Livelock occurs when all threads are blocked, or are otherwise unable to proceed due to unavailability of required resources, and there are no unblocked threads to make those resources available. In Java, thread livelock can occur:
1. When all the threads in a program execute Object.wait(0) on an object with zero parameter. The program is live-locked and cannot proceed until one or more threads call Object.notify() or Object.notifyAll() on the relevant objects. Because all the threads are blocked, neither call can be made.
2. When all the threads in a program are stuck in infinite loops.

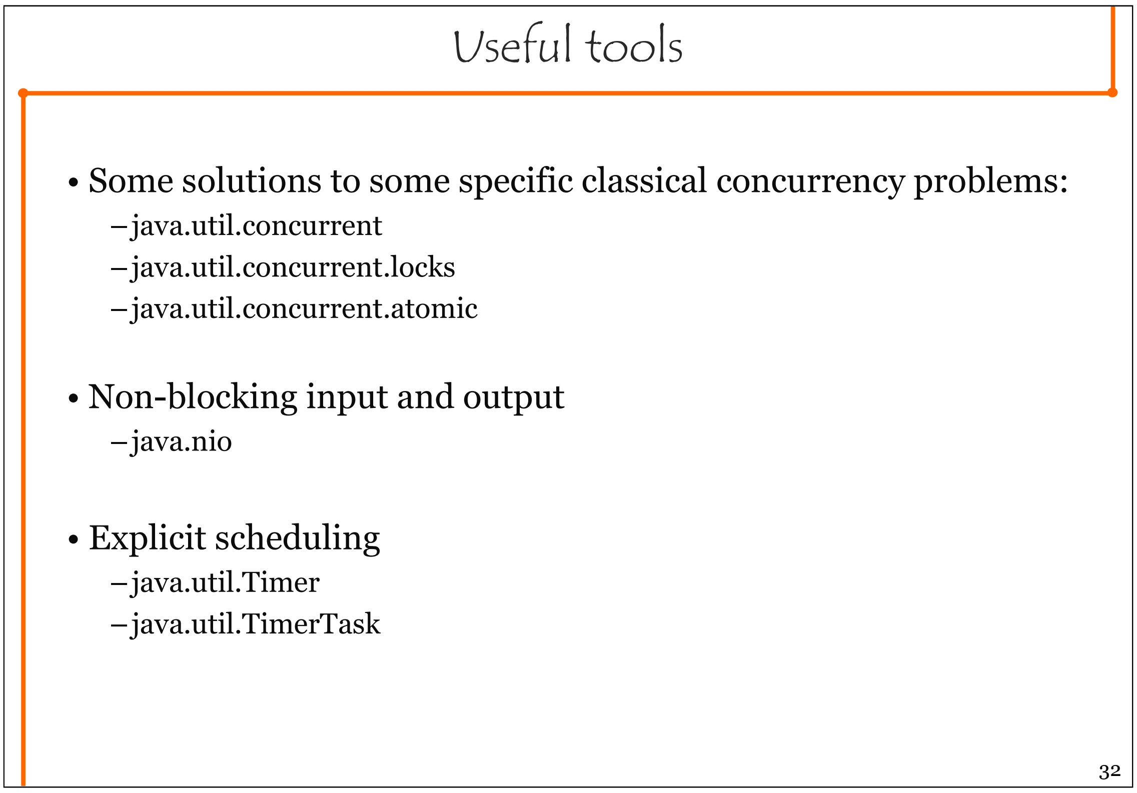
Thrashing occurs when a program makes little-to-no progress because threads perform excessive context switching. This may leave little or no time for the application (or applet) code to execute.

Starvation occurs when one thread cannot access the CPU because one or more other threads are monopolizing the CPU. In Java, thread starvation can be caused by setting thread priorities inappropriately. A lower-priority thread can be starved by higher-priority threads if the higher-priority threads do not yield control of the CPU from time to time.

Safety and **Liveness** are the two key principles of concurrent programming.

The **Safety** principle states "nothing bad can happen“, meaning data doesn’t get inconsistent

The **Liveness** principle states "eventually, something good happens“, meaning there is always at least one live thread that is capable of unblocking blocked threads.



Useful tools

- Some solutions to some specific classical concurrency problems:
  - java.util.concurrent
  - java.util.concurrent.locks
  - java.util.concurrent.atomic
- Non-blocking input and output
  - java.nio
- Explicit scheduling
  - java.util.Timer
  - java.util.TimerTask

32

If you plan on doing serious concurrent programming the these packages can be very useful, but if you are new to the concept of threads they might be more confusing than helpful. They are not required to solve any of the exercises given during this course.

A last word on performance

Using threads has an impact on performance.

But you can minimize the effects by:

- Limiting the number of threads
- Making as few areas as possible synchronized
- Using fine-grained locks in order to decrease the time threads wait for a lock
- Prioritizing threads to make selected ones wait less than others.
- Consider using shallow copies (clones) when running through i.e. Collections to avoid synchronizing large blocks of code.