ComTrade
IT Solutions and Services

Threads

Željko Vasiljević

ComTrade Global Map

HERMES SoftLab Austria

ComTrade HR Croatia

CT Computers Serbia

HERMES SoftLab Germany

ComTrade IT Solutions and Services Belgrade, Serbia

HERMES SoftLab Ireland

CT Retail Belgrade, Serbia

ComTrade IT Solutions and Services Kragujevac, Serbia

HERMES SoftLab USA

CT Computers Montenegro

ComTrade IT Solutions and Services USA

HERMES SoftLab France

CT Computers FYR Macedonia

HERMES SoftLab Nova Gorica, Slovenia

CT Computers Albania

HERMES SoftLab Ljubljana, Slovenia

CT Retail Sarajevo, BIH

ComTrade IT Solutions and Services Montenegro

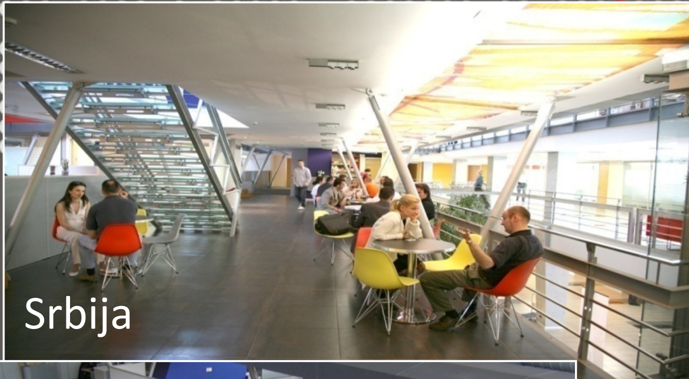
HERMES SoftLab Maribor, Slovenia

CT Computers BIH

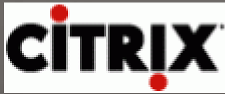
HERMES SoftLab Banja Luka, BIH

HERMES SoftLab Sarajevo, BIH

ComTrade Centri in pictures

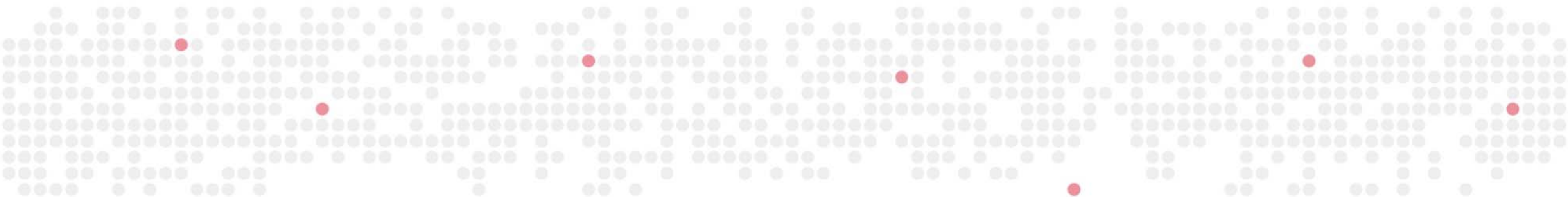


ComTrade IT Solutions and Services Partners



Overview

- Defining, Instantiating, and Starting Threads
- Transitioning Between Thread States
- Sleep, Yield, and Join
- Concurrent Access Problems and Synchronized Threads
- Deadlocked Threads
- Communicating with Objects by Waiting and Notifying
- Monitor – Signal and Continue



ComTrade
IT Solutions and Services

Introduction

In Java, "thread" means two different things:

An instance of class java.lang.Thread – object, lives and dies on the heap.

A thread of execution - "lightweight" process

One call stack per thread - main() is the first method on the stack of main thread

Concurrently, in parallel?

JVM, operates like a mini-OS and schedules *its* own threads regardless of the underlying operating system

When it comes to threads, very little is guaranteed

Once all user threads are complete, the JVM will shut down

Defining a Thread

The action happens in the run() method. Two ways to define and instantiate a thread

- Implement the **Runnable interface**
- Extend the **java.lang.Thread** class

```
class MyThread extends Thread {                         //extend Thread
    public void run() {                                 //override run()
       System.out.println("Important job running in MyThread");
    }
    public void run(String s) {
       System.out.println("String in run is " + s);
    }
}
```

Overload run() - normal method call

```
class MyRunnable implements Runnable {                  //implement Runnable
    public void run() {                                 //implement run()
       System.out.println("Important job running in MyRunnable");
    }
}
```

Instantiating a Thread

Every thread of execution begins as an instance of class Thread

Thread is the "worker," and the Runnable is the "job" to be done

```
Thread MyThread tr = new MyThread();

MyRunnable r = new MyRunnable();

Thread t = new Thread(r); // Pass your Runnable to the Thread
```

Thread constructors:

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

When a thread has been instantiated, the thread is said to be in **new state**

Starting a Thread

How to start the Thread

```
t.start();
```

What happens after we call start?

- A new thread of execution starts (with a new call stack).
- The thread moves from the ***new* state** to the ***runnable* state**.
- When the thread gets a chance to execute, its target run() method will run.

We call **start()** on a Thread instance, not on a Runnable instance

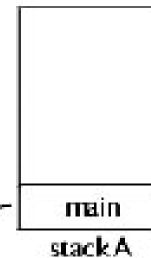
The following code does not start a new thread of execution, it will just invoke method:

```
Runnable r = new Runnable();
r.run(); // Legal, but does not start a separate thread
```

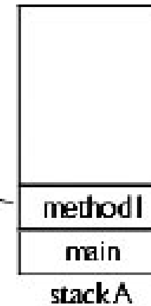
ComTrade
IT Solutions and Services

Process of starting a thread

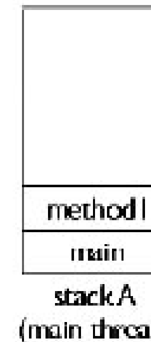
```
public static void main(String [] args) {
  //  running
  //  some code
  //  in main()
  method1();
  //  running
  //  more code
}
void method1() {
  Runnable r = new MyRunnable();
  Thread t = new Thread(r);
  t.start();
  //  do more stuff
}
```



1) main() begins



2) main() invokes method1()



3) method1() starts a new thread

ComTrade
IT Solutions and Services

Starting and Running Multiple Threads

```
class NameRunnable implements Runnable {
public void run()
{ for (int x = 1; x <= 3; x++) { System.out.println("Run by " + Thread.currentThread().getName() + ", x is " + x}}
}
```

Target Runnable instance doesn't *have* a reference to the Thread instance.

That is why we use static method Thread.currentThread() to get reference of currently executing thread.

```
public class ManyNames {
    public static void main(String [] args) {
      // Make one Runnable
      NameRunnable nr = new NameRunnable();
      Thread one = new Thread(nr);    Thread two = new Thread(nr);    Thread three = new Thread(nr);
      one.setName("Fred"); two.setName("Lucy"); three.setName("Ricky");
      one.start(); two.start(); three.start();
    }
}
```

Starting and Running Multiple Threads

Behavior you see below is not guaranteed!

% java ManyNames

Run by Fred, x is 1

Run by Fred, x is 2

Run by Fred, x is 3

Run by Lucy, x is 1

Run by Lucy, x is 2

Run by Lucy, x is 3

Run by Ricky, x is 1

Run by Ricky, x is 2

Run by Ricky, x is 3

ComTrade
IT Solutions and Services

400 iterations segment example:

Run by Fred, x is 345

Run by Lucy, x is 337

Run by Ricky, x is 310

Run by Lucy, x is 338

Run by Ricky, x is 311

Run by Lucy, x is 339

Run by Ricky, x is 312

Run by Lucy, x is 340

Run by Ricky, x is 313

Run by Lucy, x is 341

Run by Ricky, x is 314

Run by Lucy, x is 342

Run by Ricky, x is 315

Run by Fred, x is 346

Each thread will start, and each thread will run to completion.

Thread, scheduler

A thread is done being a thread when its target run() method completes

Once a thread has been started, it can never be started again

The thread scheduler is the part of the JVM that decides which thread should run at any given moment

*The order in which **runnable** threads are chosen to run is not guaranteed*

You must be able to look at thread code and determine whether the output is and is not guaranteed

ComTrade
IT Solutions and Services

Thread & thread manipulating methods

Some of the methods that can help us influence thread scheduling are as follows:

From the java.lang.Thread Class

```
public static void sleep(long millis) throws InterruptedException

public static void yield()

public final void join() throws InterruptedException

public final void setPriority(int newPriority)
```

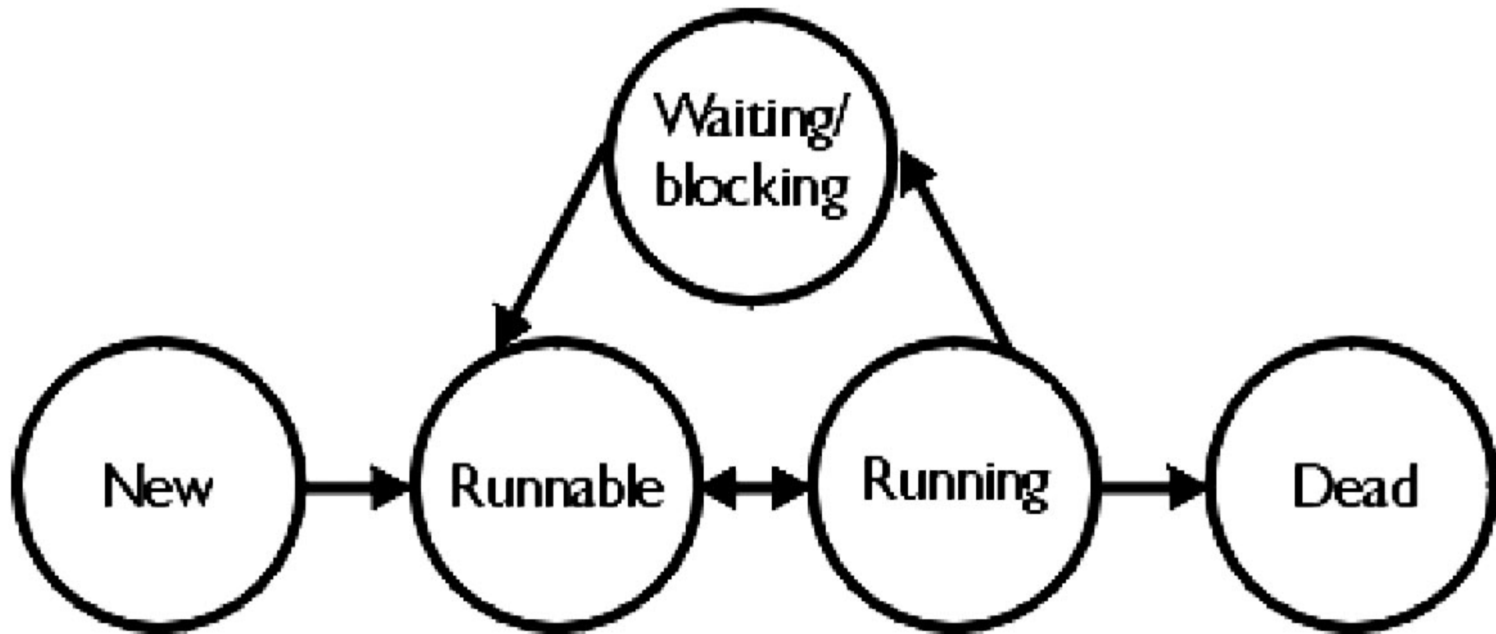
From the java.lang.Object

```
public final void wait() throws InterruptedException

public final void notify()

public final void notifyAll()
```

Thread & thread manipulating methods



ComTrade
IT Solutions and Services

Thread States description

New

This is the state the thread is in after the Thread instance has been created - ***not alive***

Runnable

This is the state a thread is in when it's eligible to run, but the scheduler has not selected it to be the running thread - ***alive***.

Running

This is the state a thread is in when the thread scheduler selects it (from the runnable pool) to be the currently executing process

Waiting/blocked/sleeping

The thread is still alive, but is currently not eligible to run. In other words, it is not *runnable*, but it might *return* to a runnable state later if a particular event occurs and the scheduler chooses a thread from the runnable pool
One thread does not *tell another thread to block (at least not in a safe way) suspend(), resume(), stop()* - **deprecated**

Dead

A thread is considered dead when its run() method completes. It may still be a viable Thread object, but it is no longer a separate thread of execution

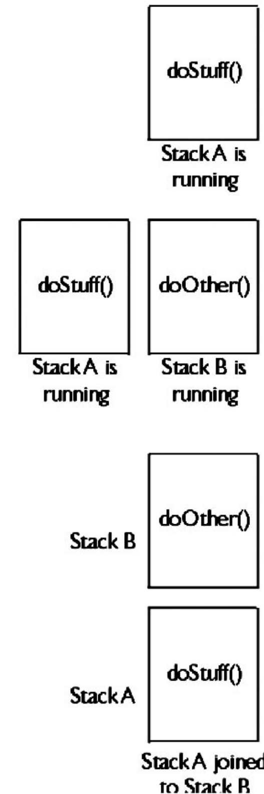
ComTrade
IT Solutions and Services

The join() Method

The non-static join() method of class Thread lets one thread "join onto the end" of another thread.

When one thread calls the join() method of another thread, the currently running thread will wait until the thread it joins with has completed

```
A is running
A is running
A is running
A is running
A is running                Thread b = new Thread(aRunnable);
A is running ——————         b.start();
A is running
B is running                // Threads bounce back and forth
B is running
A is running
B is running
A is running
A is running
B is running
B is running
A is running
B is running
A is running ——————         b.join();    // A joins to the end
B is running                             // of B
B is running
B is running
B is running
B is running
B is running
B is running
B is running ——————         //Thread B completes !!
A is running ——————         // Thread A starts again !
A is running
A is running
A is running
A is running
A is running
```



Race condition

10000$

Client#1 Client#2

Read account state->10000$

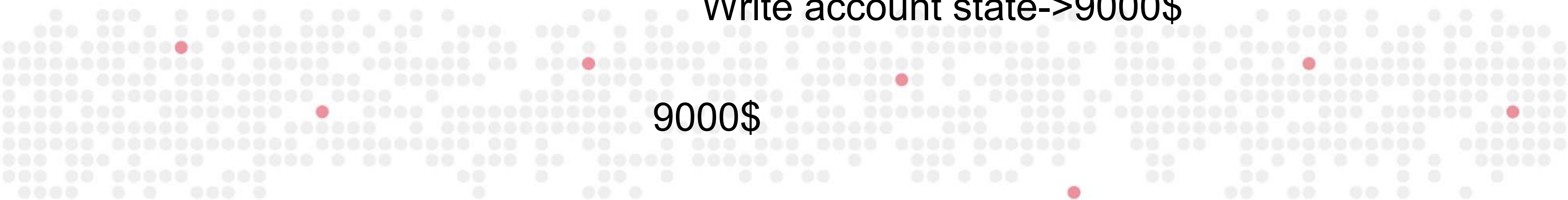
Read account state->10000$

Withdraw 9000$

Withdraw 1000$

Write account state->1000$

Write account state->9000$



9000$

Synchronizing Code

How does synchronization work? With locks. Every object in Java has a built-in lock that only comes into play when the object has synchronized method code

Key points about locking and synchronization:

- Only methods (or blocks) can be synchronized, not variables or classes
- Each object has just one lock
- Not all methods in a class need to be synchronized. A class can have both Synchronized and non-synchronized methods
- If two threads are about to execute a synchronized method in a class, and both threads are using the same instance of the class to invoke the method, only one thread at a time will be able to execute the method

ComTrade
IT Solutions and Services

Synchronizing Code

- If a class has both synchronized and non-synchronized methods, multiple threads can still access the class's non-synchronized methods!
- If a thread goes to **sleep, it holds any locks** it has—it doesn't release them
- A thread can acquire more than one lock. For example, a thread can enter a synchronized method, thus acquiring a lock, and then immediately invoke a synchronized method on a different object, thus acquiring that lock as well
- Once thread acquires the lock, it can invoce any other synchronized method on object it locked.

You can synchronize a block of code rather than a method.

```
public synchronized void doStuff() {
    System.out.println("synchronized");
}
```

is equivalent to this:

```
public void doStuff() {
  synchronized(this) {
    System.out.println("synchronized");
  }
}
```

ComTrade
IT Solutions and Services

Static methods

Static methods can be synchronized

We only need one lock per class to synchronize static methods—a lock for the whole class

java.lang.Class instance lock is used to protect the static methods of the class (if they're synchronized)

```
public static synchronized int getCount() {
   return count;
}
```

If the method is defined in a class called MyClass, the equivalent code is as follows (block):

Compiler tells JVM to find instance of Class that represents the class called MyClass

```
public static int getCount() {
synchronized(MyClass.class) { //MyClass.class - class literal
    return count;
  }
}
```

ComTrade
IT Solutions and Services

Blocking

If a thread tries to enter a synchronized method and the lock is already taken, the thread is said to be blocked on the object's lock

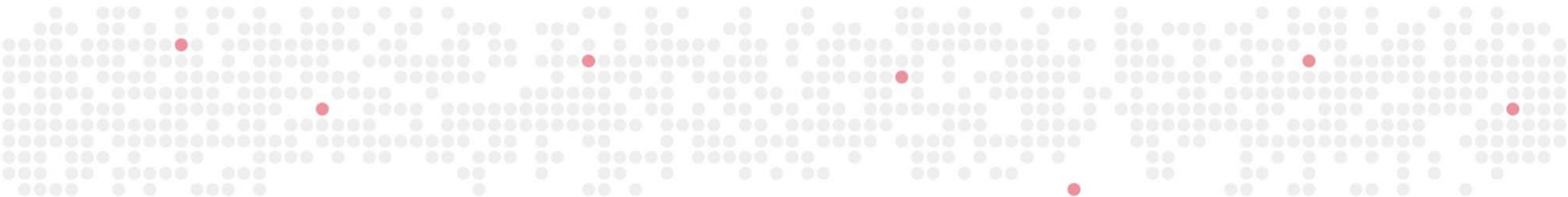
When thinking about blocking, it's important to pay attention to which objects are being used for locking

- Threads calling non-static synchronized methods in the same class will only block each other if they're invoked using the same instance
- Threads calling static synchronized methods in the same class will always block each other—they all lock on the same Class instance
- A static synchronized method and a non-static synchronized method will not block each other, ever
- For synchronized blocks, you have to look at exactly what object has been used for locking

ComTrade
IT Solutions and Services

Methods and Lock Status

Give Up Locks	Keep Locks	Class Defining the Method
wait ()	notify() (Although the thread will probably exit the synchronized code shortly after this call, and thus give up its locks.)	java.lang.Object
	join()	java.lang.Thread
	sleep()	java.lang.Thread
	yield()	java.lang.Thread



Thread Deadlock

Example : Two threads are waiting for each other's locks to be released; therefore, the locks will never be released!

The code is waiting for locks to be removed from objects

Deadlocking is baaad. Don’t do it.

```
public class DeadlockRisk {
	private static class Resource {
		public int value;
	}
	private Resource resourceA = new Resource();
	private Resource resourceB = new Resource();
	public int read() {
		synchronized(resourceA) {
			synchronized(resourceB) {return resourceB.value + resourceA.value;}
		}
	}
	public void write(int a, int b) {
		synchronized(resourceB) {
			synchronized(resourceA) {resourceA.value = a;resourceB.value = b;}
		}
	}
}
```

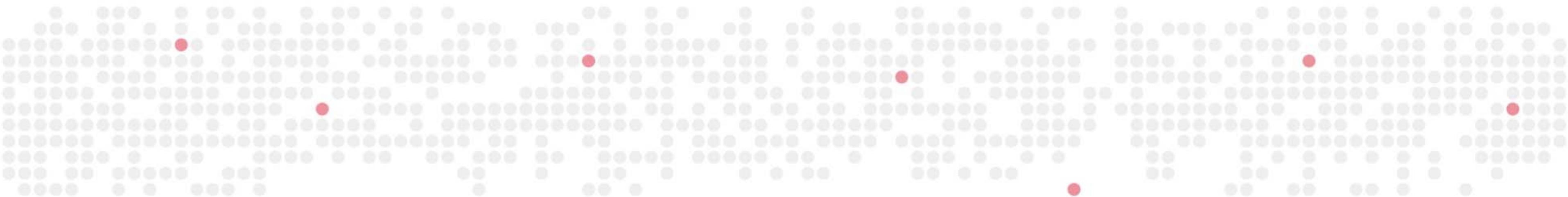
ComTrade
IT Solutions and Services

Thread Interaction

The Object class has three methods, **wait()**, **notify()**, and **notifyAll()** that help threads communicate about the status of an event that the threads care about

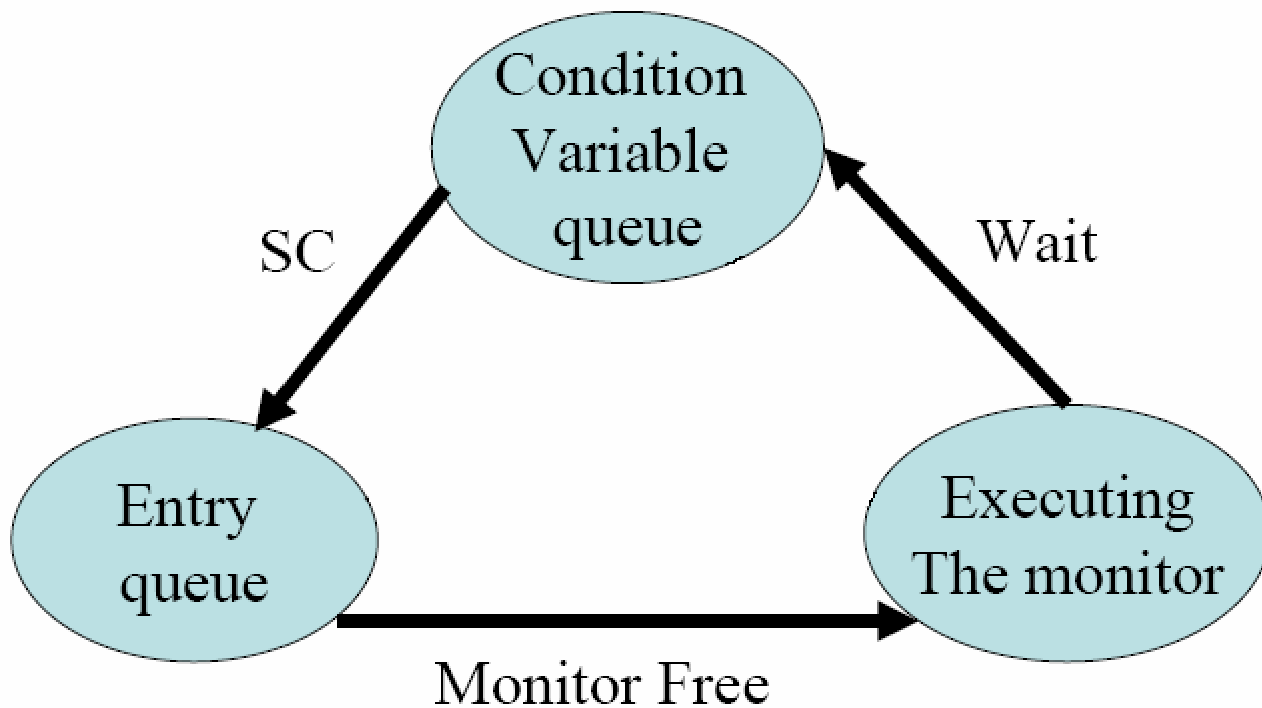
The methods wait(), notify(), notifyAll() are instance methods of **Object**

wait(), **notify()**, and **notifyAll()** must be called from within a **synchronized context**! A thread can't invoke a wait or notify method on an object unless it owns that object's lock



ComTrade
IT Solutions and Services

Monitor – Signal and Continue



Queue NOT guaranteed!

Operator-Machine

```
class Operator extends Thread {
	public void run(){
		while(true){
			// Get shape from user
			synchronized(this){
				// Calculate new machine steps from shape
				notify();
			}
		}
	}
}
class Machine extends Thread {
	Operator operator; // assume this gets initialized
	public void run(){
		while(true){
			synchronized(operator){
				try {
					operator.wait();
				} catch(InterruptedException ie) {}
				// Send machine steps to hardware
			}
		}
	}
}
```

ComTrade
IT Solutions and Services

MailBox

```
public class MailBox{
        private Letter todaysLetter;
        private boolean available = false;
        public synchronized Letter get() {
                while (available == false) {
                        try {
                                wait();
                } catch (InterruptedException e) { }
                }
                available = false;
                notifyAll();
                return todaysLetter;
        }
        public synchronized void put(Letter letter) {
                while (available == true) {
                        try {
                                wait();
                        } catch (InterruptedException e) { }
                }
                todaysLetter = letter;
                available = true;
                notifyAll();
        }
}
```

ComTrade
IT Solutions and Services

Hvala