JAVA FULL STACK DEVELOPMENT PROGRAM

Session 6: Thread

OUTLINE

- Thread & Process
- Thread in Java
    - Thread
    - Runnable
- Daemon Threads
- Synchronization
- Thread problem

PROCESS

- A process has a self-contained execution environment.
    - A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

- Processes are often seen as synonymous with programs or applications
    - However, what the user sees as a single application may in fact be a set of cooperating processes
    - eg. To facilitate communication between processes, most operating systems support *Inter Process Communication* (IPC) resources, such as pipes and sockets

THREAD

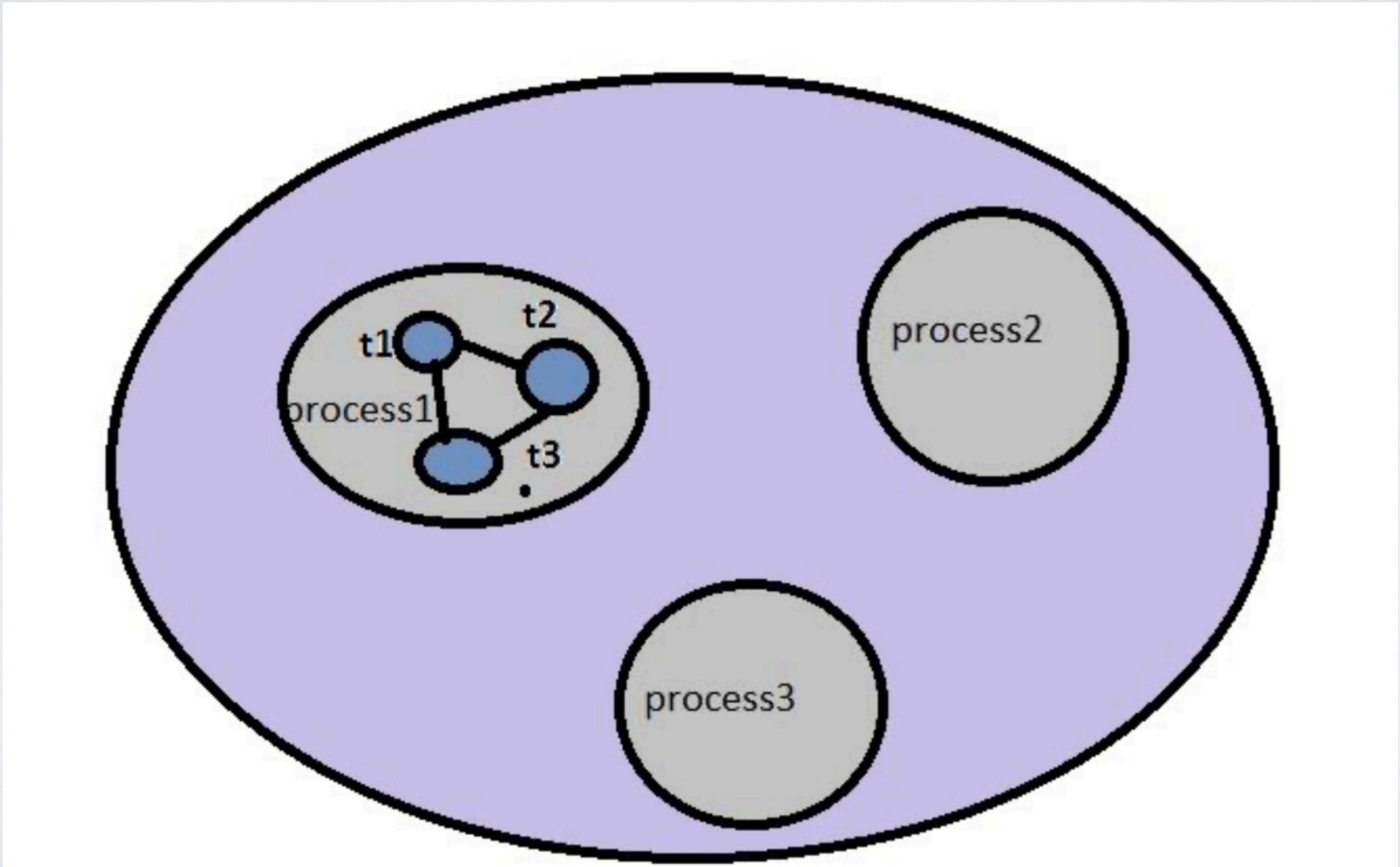
- A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution

- Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

THREAD VS PROCESS

S.NO	PROCESS	THREAD
1.	Process means any program is in execution.	Thread means segment of a process.
2.	Process takes more time to terminate.	Thread takes less time to terminate.
3.	It takes more time for creation.	It takes less time for creation.
4.	It also takes more time for context switching.	It takes less time for context switching.
5.	Process is less efficient in term of communication.	Thread is more efficient in term of communication.
6.	Process consume more resources.	Thread consume less resources.
7.	Process is isolated.	Threads share memory.

THREAD VS PROCESS



WHY THREAD

- To enhance parallel processing
    - Massively parallel server monitoring infrastructure, for example u are required to monitor many servers/routers every 5 seconds
- To reduce response time to the user
    - HTTP/JEE servers/frameworks like tomcat, jetty uses multithreads to achieve high throughput
- To utilize the idle time of the CPU
    - jUnit uses threads to run test cases in parallel
- Prioritize your work depending on priority
    - Computer games is a good example of multi-threaded processing (Loading maps when you are working on other things)

THREAD IN JAVA

- Two way to create a new thread
  - Extends the Thread class
  - Implements the Runnable interface

THREAD CLASS

- In Java, there is a class named as Thread class, which belongs to java.lang package, declared as
  - *public class Thread extends Object implements Runnable*
- This class encapsulates any thread of execution. Threads are created as the instance of this class, which contains run() methods in it. The functionality of the thread can only be achieved by overriding this run() method
  - *public void run( ) {*
    // statement for implementing thread
    }

THREAD CLASS

```
class MyThread extends Thread{
    @Override
    public void run() {
        System.out.println("MyThread -----> "+Thread.currentThread().getName());
    }
}
```

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

RUNNABLE INTERFACE

- Runnable interface is implemented by Thread class in the package java.lang. This interface is declared as

  - *public interface Runnable*

- The interface needs to be implemented by any class whose instance is to be executed by a thread. The implementing class must also override a void method named as run(), defined as a lone method in the Runnable interface as

  - *public void run( ) {*
    // statement for implementing thread
    }

RUNNABLE INTERFACE

```
Thread myThread2 = new Thread(()->{
    System.out.println("Implement Runnable Interface "+Thread.currentThread().getName());});
myThread2.start();
```

START VS RUN

- When a program calls the *start()* method, a new thread is created and then the *run()* method is executed

- When a program directly call the *run()* method then no new thread will be created and *run()* method will be executed as a normal method call on the current calling thread itself and no multi-threading will take place

START VS RUN

- Can we start a thread twice?
  - eg. call start method on same thread twice
- What if I invoke run() method instead of start() method
  - eg. call run method on same thread twice

START VS RUN

START()	RUN()
Creates a new thread and the run() method is executed on the newly created thread.	No new thread is created and the run() method is executed on the calling thread itself.
Can’t be invoked more than one time otherwise throws *java.lang.IllegalStateException*	Multiple invocation is possible
Defined in *java.lang.Thread* class.	Defined in *java.lang.Runnable* interface and must be overriden in the implementing class.

RUNNABLE OR THREAD

- Simply put, we generally encourage the use of *Runnable* over *Thread*:
    - When extending the *Thread* class, we're not overriding any of its methods. Instead, we override the method of *Runnable (*which *Thread* happens to implement). This is a clear violation of IS-A *Thread* principle
    - Creating an implementation of *Runnable* and passing it to the *Thread* class utilizes aggregation/composition and not inheritance – which is more flexible
    - After extending the *Thread* class, we can't extend any other class
    - From Java 8 onwards, *Runnable* can be represented as lambda expressions

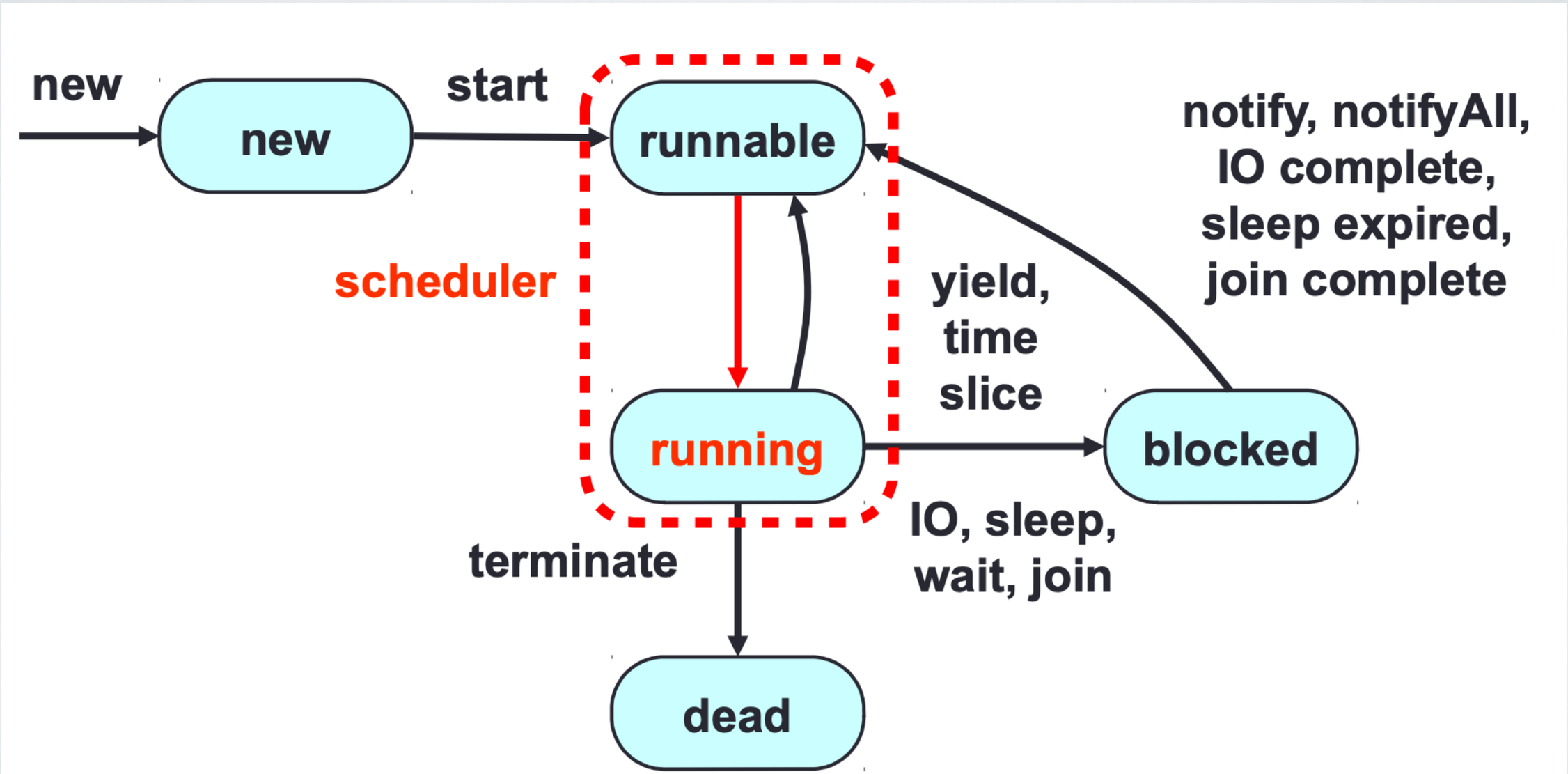
MAIN THREAD

- Every Java program has a default thread, main thread.
- When the execution of Java program starts, the JVM creates the main thread and calls the program's main() method within that thread.
- Apart from this JVM also creates some invisible threads, which are important for JVM housekeeping tasks
- Programmers can always take control of the main thread.

THREAD LIFE CYCLE

- Five states of thread
  - new — just created but not started
  - runnable — created, started, and able to run
  - running — thread is currently running
  - blocked — created and started but unable to run because it is waiting for some event to occur
  - dead — thread has finished or been stopped

THREAD LIFE CYCLE



BLOCK THREAD

- Sleep — `Thread.sleep` causes the current thread to suspend execution for a specified period.
- Join — The `join` method allows one thread to wait for the completion of another
    - If `t` is a `Thread` object whose thread is currently executing,

      t.join();

- causes the current thread to pause execution until `t`'s thread terminates. Overloads of `join` allow the programmer to specify a waiting period.
- However, as with `sleep`, `join` is dependent on the OS for timing, so you should not assume that `join` will wait exactly as long as you specify.
- Search later synchronization: what's the difference between sleep() and wait()

THREAD EXAMPLE

- SimpleThreads

- Several things to notice
  - Thread.currentThread().getName() is returning the thread name in the context of thread
  - sleep() and join() both throw the InterruptedException which needs to be handled properly
  - what happens if we removed t.join(1000)?
  - what happens if we removed t.join()?

SLEEP WAIT

- wait(): an instance method on Object that's used for thread synchronization. Can only be called from a synchronized block. It release the lock on the object so that another thread can jump in and acquire a lock.

- Thread.sleep(): static method and can be called from any context. Thread.sleep() pauses the current thread and does not release any locks.

DAEMON THREAD

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads

- Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

- It is a low priority thread

- eg. garbage collection thread

DAEMON THREAD

- How to create Daemon thread
  - public void setDaemon(boolean status) -- is used to mark the current thread as daemon thread or user thread
  - public boolean isDaemon() -- is used to check that current is daemon

DAEMON THREAD

- Why Daemon thread?
    - Create a daemon thread for functionalities that are not critical to system
        - logging thread or monitoring thread to capture the system resource details and their state
- Take another example:
    - Imagine you're writing a simple game where your main method loops until you decide to quit. And imagine that at the start of the game, you start a thread that will endlessly poll some website to trigger alerts. You would like the JVM to exit when you decide to end the game. You don't want the endless polling to prevent the game from ending. So you make this polling thread a daemon thread.

THREAD INTERFERENCE

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

temp = c;
temp++;
c = temp

- Virtual Machines like JVM will decompose the "c++" statement into following steps
  - Retrieve the current value of c.
  - Increment the retrieved value by 1.
  - Store the incremented value back in c.
- What if thread A is calling increment and thread B is calling decrement? (What will be the result?)
- This situation is also called Race Condition

RACE CONDITION

Thread A(c++)

temp = c

temp++

C = temp

Thread B(c- -)

temp = c

temp - -

C = temp

C = 0

value()

THREAD INTERFERENCE

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

- Consider a situation where two thread is operating on the same object at the same time.
- Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

MEMORY CONSISTENCY ERRORS

- *Memory consistency errors* occur when different threads have inconsistent views of what should be the same data

- Eg. Same Counter class as before — Thread A increases the counter by 1, and thread B tries to print the value of counter. Now the value can either be 1 or 0

THREAD

```
1  package com.beaconfire;
2
3  public class SynExample {
4      public static void main(String[] args) {
5          Booking booking = new Booking();
6          Thread fan = new Thread(()->{
7              for(int i=0;i<100;i++) {
8                  booking.buy();
9              }
0          },"Fan");
1          Thread landon = new Thread(()->{
2              for(int i=0;i<100;i++) {
3                  booking.buy();
4              }
5          },"Landon");
6          fan.start();
7          landon.start();
8      }
9  }
0  class Booking{
1      private int tickets;
2      public Booking() {
3          this.tickets = 50;
4      }
5      public void buy() {
6          if(tickets<=0)
7              return;
8          System.out.println(Thread.currentThread().getName()+" bought "+tickets--);
9      }
0
1  }
```

SYNCHRONIZED METHOD

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

- It is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a *synchronized method* for an **object**, all other threads that invoke synchronized methods for the **same object** block (suspend execution) until the first thread is done with the object.
- When a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.
- Think about synchronization in terms of locks

SYNCHRONIZATION

- Threads communicate primarily by sharing access to fields and the objects reference fields refer to
- This form of communication is extremely efficient, but makes two kinds of errors possible: *thread interference* and *memory consistency errors*.
- The tool needed to prevent these errors is *synchronization*.
- Synchronization can be achieved in two ways
  - By using synchronized keyword with method definition
  - By using synchronized keyword with any block of code

SYNCHRONIZATION

- method definition
- any block of code

SYNCHRONIZED BLOCK

- Synchronized block can be used to perform synchronization on any specific resource of the method
- Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block
- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method
- It is not required to understand the Synchronized Block, but it is required to know how to write Synchronized Block in code
- Synchronized method is just a special case of synchronized block

SYNCHRONIZED

- Object
- Class

THREAD PROBLEM

- Deadlock
- Starvation (Optional)
- Livelock (Optional)

JAVA.UTIL.CONCURRENT

- ConcurrentHashMap
- ReentrantLock
- AtomicInteger

DEADLOCK

- *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other

```
static class Friend {
    private final String name;
    public Friend(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
    public synchronized void bow(Friend bower) {
        System.out.format("%s: %s"
            + "  has bowed to me!%n",
            this.name, bower.getName());
        bower.bowBack(this);
    }
    public synchronized void bowBack(Friend bower) {
        System.out.format("%s: %s"
            + " has bowed back to me!%n",
            this.name, bower.getName());
    }
}
```

```
public static void main(String[] args) {
    final Friend alphonse =
        new Friend("Alphonse");
    final Friend gaston =
        new Friend("Gaston");
    new Thread(new Runnable() {
        public void run() { alphonse.bow(gaston); }
    }).start();
    new Thread(new Runnable() {
        public void run() { gaston.bow(alphonse); }
    }).start();
}
```

ANY QUESTIONS?