Java Threads

Object Oriented Programming

http://softeng.polito.it/courses/09CBI

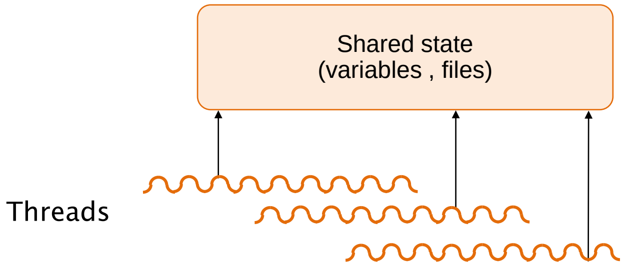
Version 2.2.5 - June 2019

1

Licensing Note

2

What Are Threads?



- General-purpose solution for managing concurrency
- Multiple independent execution streams
- Shared state

3

What Are Threads Used For?

- Operating systems
  - one kernel thread for each user process.
- Scientific applications
  - one thread per CPU (solve problems faster).
- Distributed systems
  - process requests concurrently (overlap I/Os).
- GUIs
  - Threads correspond to user actions; they can help display during long-running computations.
  - Multimedia, animations.

4

Process

- From an OS viewpoint, a Process is an instance of a running application
- Has it own
  - virtual address space
  - code,
  - data,
  - other OS resources (e.g. files)
- A process also contains one or more threads that run in the context of the process.

5

Thread

- A thread is the basic entity to which the operating system allocates CPU time.
- A thread can execute any part of the process code
  - Including a part currently being executed by another thread.
- All threads of a process share the same virtual address space, global variables, and operating system resources.

6

Multitasking

- User: capability to have several applications open and working at the same time.
  - A user can edit a file with one application while another application is printing or recalculating a spreadsheet.
- Developer: capability to create processes that use more than one thread of execution, e.g.
  - One handles interactions with the user
  - Another performs background work

7

Multitasking

- A multitasking OS assigns CPU time (slices) to threads
- A preemptive OS executes a thread until
  - Its assigned time slice is over,
  - It ends its own execution,
  - It blocks (synchronization with other threads)
  - A thread with higher priority becomes available
- Using small time-slices (e.g. 20 ms) the thread execution is apparently parallel
  - Actually parallel in multiprocessor systems

8

Multitasking Problems

- O.S. consumes memory for the structures required by both processes and threads.
  - Keeping track of a large number of threads also consumes CPU time.
- Multiple threads accessing the same resources should be synchronized to avoid conflicts (deadlocks or race conditions)
  - System resources (communications ports, disk drives),
  - Handles to resources shared by multiple processes (files)
  - Resources of a process (variables used by multiple threads)

SoftEng

9

JVM and Operating System

- Do not interpret the behavior on one machine as "the way usually threads work"
- Design a program so that it will work regardless of the underlying JVM.
- Thread programming motto:

When it comes to threads,
very little is guaranteed

SoftEng

10

JVM Scheduler

- The Scheduler is the JVM part that decides
  - Which thread should run at any given time,
  - Takes threads out of the running state.
  - Some JVMs use O.S. scheduler (**native threads**)
- Assuming a single processor machine:
  - Only one thread can actually run at a time.
- The order in which runnable threads are chosen to be THE ONE running is **NOT** guaranteed.

SoftEng

11

Create a thread

- Threads can be created by extending **`Thread`** and overriding the **`run()`** method.
- **`Thread`** objects can also be created by calling the Thread constructor that takes a **`Runnable`** argument (the target of the thread)
  - The same Runnable object can be the *target* of different **`Thread`** objects

SoftEng

12

Create a Thread

1. Extends **Thread** class

```
class X extends Thread {
   public void run() {  //code here  }
}
Thread t = new X();
t.start(); // Create and start
```

2. Implementing **Runnable** interface (better)

```
class Y implements Runnable {
   public void run() {  //code here  }
}
Thread r = new Thread (new Y());
r.start(); //invoke run() & create new call-stack
```

SoftEng

13

Start a Thread

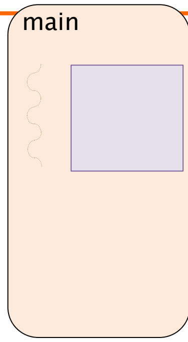
- When a **Thread** object is created, it does not become a thread of execution until its **start()** method is invoked.
- When a **Thread** object exists but hasn't been started, it is in the *New* state and it is not considered alive.
- Method **start()** can be called on a **Thread** object only once.
  - If it is called more than once on same object, it will throw a **RuntimeException**

SoftEng

14

Starting a thread

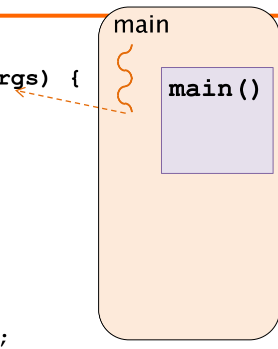
```
public class Starting {
  public static void main(String[] args) {
    m();
  }
  static void m(){
    Thread t = new MyThread();
    t.start();
    sayHello("main");
  }
  static void sayHello(String a){
    System.out.println(a+": Hello!");
  }
  static class MyThread extends Thread{
    public void run(){
       sayHello("t");
    }
  }
}
```



15

Starting a thread

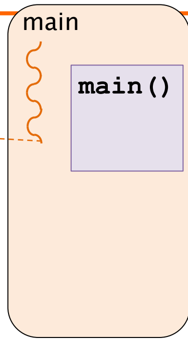
```
public class Starting {
  public static void main(String[] args) {
    m();
  }
  static void m(){
    Thread t = new MyThread();
    t.start();
    sayHello("main");
  }
  static void sayHello(String a){
    System.out.println(a+": Hello!");
  }
  static class MyThread extends Thread{
    public void run(){
       sayHello("t");
    }
  }
}
```



16

Starting a thread

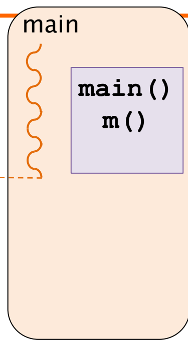
```
public class Starting {
  public static void main(String[] args) {
    m();
  }
  static void m(){
    Thread t = new MyThread();
    t.start();
    sayHello("main");
  }
  static void sayHello(String a){
    System.out.println(a+": Hello!");
  }
  static class MyThread extends Thread{
    public void run(){
       sayHello("t");
    }
  }
}
```



17

Starting a thread

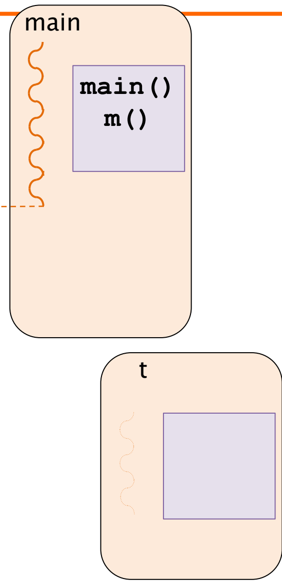
```
public class Starting {
  public static void main(String[] args) {
    m();
  }
  static void m(){
    Thread t = new MyThread();
    t.start();
    sayHello("main");
  }
  static void sayHello(String a){
    System.out.println(a+": Hello!");
  }
  static class MyThread extends Thread{
    public void run(){
       sayHello("t");
    }
  }
}
```



18

Starting a thread

```
public class Starting {
  public static void main(String[] args) {
    m();
  }
  static void m(){
    Thread t = new MyThread();
    t.start();
    sayHello("main");
  }
  static void sayHello(String a){
    System.out.println(a+": Hello!");
  }
  static class MyThread extends Thread{
    public void run(){
       sayHello("t");
    }
  }
}
```

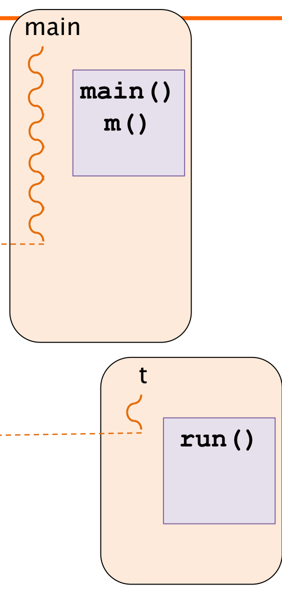


SoftEng

19

Starting a thread

```
public class Starting {
  public static void main(String[] args) {
    m();
  }
  static void m(){
    Thread t = new MyThread();
    t.start();
    sayHello("main");
  }
  static void sayHello(String a){
    System.out.println(a+": Hello!");
  }
  static class MyThread extends Thread{
    public void run(){
       sayHello("t");
    }
  }
}
```

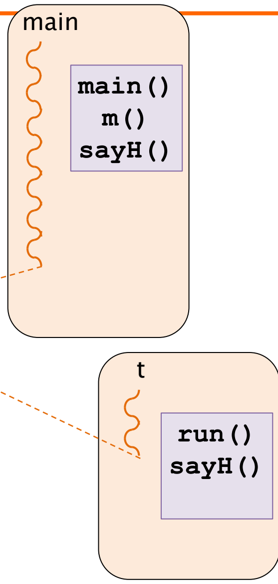


SoftEng

20

Starting a thread

```
public class Starting {
  public static void main(String[] args) {
    m();
  }
  static void m(){
    Thread t = new MyThread();
    t.start();
    sayHello("main");
  }
  static void sayHello(String a){
    System.out.println(a+": Hello!");
  }
  static class MyThread extends Thread{
    public void run(){
       sayHello("t");
    }
  }
}
```



Example: **extends Thread**

- Two threads, each counting up to N

```
class Counter extends Thread {
    private int num; String name;
    public Counter(String nn, int n) {
          name= nn; num = n;     }
    public void run(){
      for( int i=0; i<num; ++i)
        System.out.print(name+": "+i+" ");
    }
}
```

```
public static void main(String args[]){
Counter t1 = new Counter("Kevin",10);
   Counter t2 = new Counter("Bob",5);
   t1.start(); t2.start();
}
```

Ex. implements Runnable

```
class CounterR implements Runnable {
  private int num; private String lab;
  public CounterR(String l, int n) {
     num = n; lab = l; }
  public void run(){
    for(int i=0; i<num; ++i)
     System.out.print(lab+": "+i+" ");
} }
```

```
public static void main(String args[]){
   Thread t1,t2;
   t1 = new Thread(new CounterR("Kevin",10));
   t2 = new Thread(new CounterR("Bob",5));
   t1.start();  t2.start();
}
```

SoftEng

23

Ex. Runnable lambda

```
public static void main(String args[]){
  Thread t1,t2;
  final int num1 = 10;
  t1 = new Thread(() -> IntStream.range(0,num1)
                    .mapToObj(i->"Kevin: ",i+" ")
                    .forEach(System.out::print);
                    );
  final int num2 = 5;
  t2 = new Thread(() -> IntStream.range(0,num2)
                    .mapToObj(i->"Bob: ",i+" ")
                    .forEach(System.out::print);
                    );
  t1.start();
  t2.start();
}
```

SoftEng

24

Ex. **Runnable** factory w/λ

```
static Runnable counting(String l, int num){
  return () -> IntStream.range(0,num)
              .mapToObj(i->l+": "+i+" ")
              .forEach(System.out::print);
}
```

```
public static void main(String args[]){
    Thread t1,t2;
    t1 = new Thread(counting("Kevin",10));
    t2 = new Thread(counting("Bob",5));
    t1.start();
    t2.start();
}
```

25

EXECUTORS

26

Executors

- When multiple tasks have to be performed a few issues exist:
  - Thread creation and starting
  - Controlling number of threads
  - Queuing tasks
  - Stop all the running tasks
- Executor services can be used to simplify such operations
  - **`java.util.concurrent`**

SoftEng

27

ExecutorService

- **`submit()`**
  - Submits a new task to the service
- **`shutdown()`**
  - Awaits for task to terminate and then stops the service
- **`shutdownNow()`**
  - Terminates tasks and the service
- **`awaitTermination()`**
  - Awaits shutdown to terminate service

SoftEng

28

Create executor services

- Using class **Executors** static methods
  - **newCachedThreadPool()**
    - Creates as many threads as needed and reuse
  - **newFixedThreadPool()**
    - Creates fixed size thread pool
  - **newSingleThreadExecutor()**
    - Creates a single thread
  - **newWorkStealingPool()**
    - Creates as many threads to match the available number of processors

29

Tasks

- **Runnable**
  - Method void **run()**
- **Callable<T>**
  - Method **T call()**
  - Submit returns a **Future<T>**
    - **isDone()** checks if the computation is completed and the value available
    - **get()** blocks until a value is returned (or a timeout expires)

30

Executor Service w/λ factory

```
static Runnable counting(String l, int num){
  return () -> IntStream.range(0,num)
                .mapToObj(i->l+": "+i+" ")
                .forEach(System.out::print);
}
```

```
public static void main(String args[]){
    ExecututorService es =
            Executors.newFixedThreadPool(2);
    es.submit(counting("Kevin",10));
    es.submit(counting("Bob",5));
}
```

SoftEng

31

Executor with Future

```
ExecutorService deepThought =
          Executors.newCachedThreadPool();
Callable<Long> lifeUniverseEverything = () -> {
               Thread.sleep(SEVEN_HALF_MY);
               return 42;        };
Future<Long> answer = deepThought
          .submit(lifeEverything);
try {
  Integer theAnswer = answer.get();  // blocks
  System.out.println(”Answer: " + theAnswer);
} catch (Exception e) {
  e.printStackTrace();  }
```

SoftEng

32

JAVA THREAD STATES

33

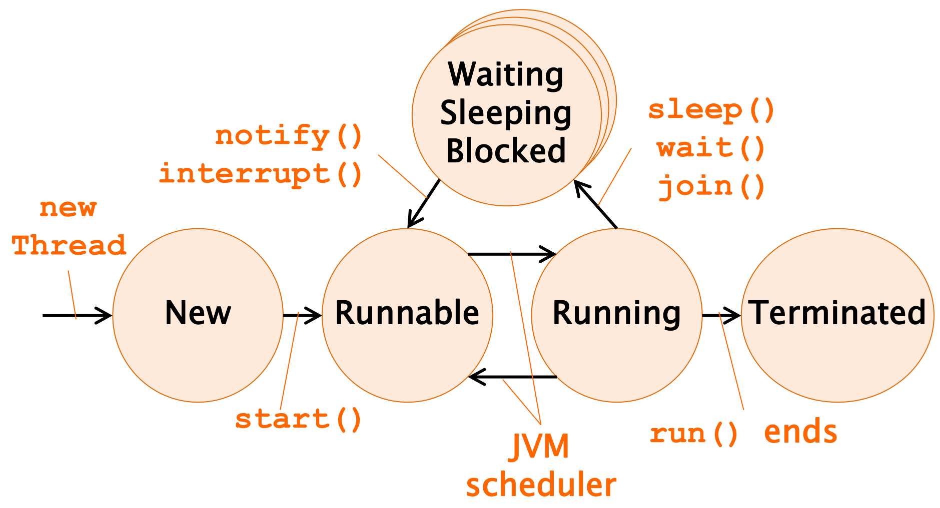
Running Multiple Threads

- There is no guarantee that:
  - threads will begin execution in the order they were started
  - a thread keeps executing until it's done
  - a loop completes before another thread begins
- Nothing is guaranteed except:

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

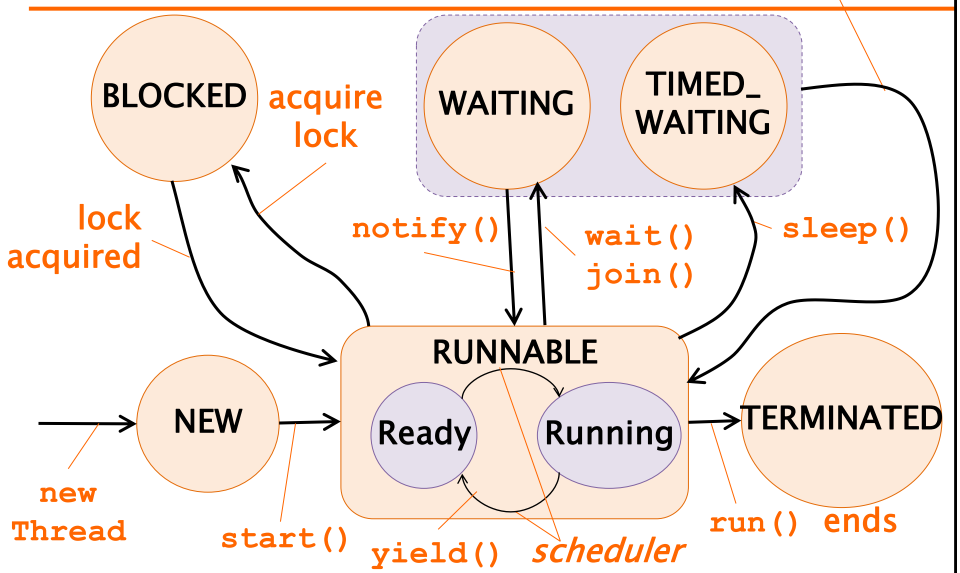
34

Java Thread States (simplified)



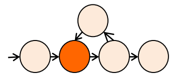
35

Java Thread States



36

Thread state: Runnable



- A thread is either
  - queued & eligible to run, but waiting for the CPU time
  - Running on the CPU
- A thread first enters the *Runnable* state when the **`start()`** method is invoked
- A thread can also return to the *Runnable* state coming back from a blocked, waiting, or sleeping state

SoftEng

37

Thread Priorities

- A thread always runs with a priority number
- The scheduler in most JVMs uses preemptive, priority-based round-robin scheduling
- Usually **time-slicing** is used:
  - Each thread is allocated a fair amount of time
  - After that a thread is sent back to the ready queue to give another thread a chance
  - JVM specification does not require a VM to implement a time-slicing scheduler!!!

SoftEng

38

JVM Scheduling Policy

- Non-Preemptive: current thread is executed until the end, unless thread explicitly releases CPU to let another thread take its turn
  - used in real-time apps (interruption can cause problems)
- Preemptive time-slicing: thread is executed until its time-slice is over, then the JVM suspends it and starts another runnable thread
  - Simpler development, as all resources handled by JVM
  - Apps do not require to use yield() to release resources
- High priority threads:
  - Are executed more often, or have longer time-slice
  - Stop execution of lower-priority threads before their time-slice is over

39

Setting a Thread's Priority

- By default, a thread gets the priority of the thread of execution that creates it.
- Priority values are defined between 1 and 10

```
Thread.MIN_PRIORITY    (1)
Thread.NORM_PRIORITY  (5)
Thread.MAX_PRIORITY    (10)
```

- Priority can be directly set

```
FooRunnable r = new FooRunnable();
Thread t = new Thread(r);
t.setPriority(8);     t.start();
```

40

yield

- The method **yield()** sets the currently running thread back to Runnable state
  - It allows other threads of the same priority to get their turn
  - **yield()** might have no effect at all
    - There's no guarantee the yielding thread won't just be chosen again over all the others

SoftEng

41

Thread state: Timed waiting

- A thread may be sleeping because the thread's **run()** code tells it to sleep for some period of time,
- It gets back to Runnable state when it wakes up because its sleep time has expired

```
try {
  Thread.sleep(5*60*1000);
  // Sleeps for 5 min
} catch (InterruptedException ex) { }
```

SoftEng

42

Example sleep

```
class NameRunnable implements Runnable {
public void run() {
   for (int x = 1; x < 4; x++) {
      System.out.println("Run by "+
              Thread.currentThread().getName());
      try {
         Thread.sleep(1000);
      } catch (InterruptedException ex) { }
} }  }
```

```
public class ManyNames {
public static void main (String [] args) {
NameRunnable nr = new NameRunnable();
Thread one = new Thread(nr,"Kevin");
Thread two = new Thread(nr,"Stuart");
Thread three = new Thread(nr,"Bob");
one.start(); two.start(); three.start();
} }
```

SoftEng

43

Thread state: Waiting

- The thread asked to wait for a signal from another thread
- It comes back to *Runnable* state when another thread
  - Terminates and the current tread asked to join (**`join()`**)
  - Sends a notification (**`notify()`**) that this thread waiting for (**`wait()`**)
- Used for thread coordination

SoftEng

44

join

- The `join()` method lets a thread "*join onto the end*" of another thread

```
Thread t = new Thread();
t.start();
t.join();
```

- The current thread moves to the *Waiting* state and it will be *Runnable* when thread *t* terminates
- An optional timeout can be set

```
t.join(5000);
```

SoftEng

45

Thread state: Blocked

- A thread is waiting for acquiring a mutually exclusive access to a resource that is currently owned by another thread
- The thread returns to *Runnable* state when the lock on the resource is released by the other thread

SoftEng

46

Interrupting a thread

- A thread cannot be forced to stop!
  - The **`stop()`** method is deprecated
- Method **`interrupt()`** can be used to "suggest" a thread to stop execution
- When a thread is in Sleep/Wait state and its **`interrupt()`** method is invoked the method throws an **`InterruptedException`**

SoftEng

47

Handling an interruption

```
Thread t = new Thread(new ()->{
while(true){
  try {
    System.out.print(".");
     Thread.sleep(1000);
  }catch(InterruptedException e){
     System.out.println("|STOP|");
     return;
  }
}
});
```

Perform the usual task

On interruption clean up and terminate thread

SoftEng

48

A word of advice

- Some methods may look like they tell another thread to block, but they don't.
- If *t* is a thread object reference, you can write something like this:

  `t.sleep()` or **`t.yield()`**
- They are static methods of the Thread class:
  - they don't affect the instance t !!!
  - instead they affect the thread in execution
  - That's why it's a bad idea to use an instance variable to access a static methods

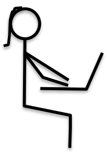
49

SYNCHRONIZATION

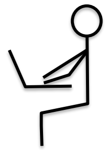
50

Example scenario

- What happens when two different threads are accessing the same data ?
- Imagine two people each having ATM cards, both linked to the same account.



```
class Account {
private int balance = 50;
public int getBalance() {
     return balance;
}
public void withdraw(int amount){
      balance = balance - amount;
}  }
```



51

Example scenario (II)

- Two steps are required for performing a withdrawal:
  1. Check the balance
  2. If there is enough money in the account, execute the withdrawal
- What happens if some times passes between step 1 and step 2?
  - …while another card holder is attempting the same task?

52

Example scenario (III)

- Stuart checks the balance and there is enough (10)
- Before he withdraws money, Kevin checks the balance and also sees that there's enough for his withdrawal.
- He is seeing the account balance before Stuart actually debits the account…
- Both Stuart and Kevin believe there's enough to make their withdrawals !
- If Stuart makes his withdrawal…
- …there isn't enough in the account for Homer's withdrawal
- … but he thinks there is since when he checked, there was enough!

SoftEng

53

Example: code

```
class DangerousWithdraw implements Runnable {
private Account account = new Account();
public static void main (String [] args) {
   DangerousWithdraw r = new DangerousWithdraw();
   Thread one = new Thread(r, "Kevin");
   Thread two = new Thread(r, "Bob");
   one.start();  two.start();
}
public void run() {
  for (int x = 0; x < 5; x++) {
    makeWithdrawal(10);
    if (account.getBalance() < 0)
      System.out.println("account is overdrawn!");
}  }
```

54

Example: code

```
private void makeWithdrawal(int amount) {
if (account.getBalance() >= amount)  {
    System.out.println(Thread.currentThread().getName() +
                   " is going to withdraw");
   try {
      Thread.sleep(500);
    } catch(InterruptedException ex) { }
    account.withdraw(amount);
    System.out.println(Thread.currentThread().getName()+
                              " completes the withdrawal");
} else {
    System.out.println("Not enough in account for "+
  Thread.currentThread() .getName()+ "to withdraw "+
  account.getBalance());
}  }  }
```

SoftEng

55

Race Condition

- A problem happening whenever:
  - Many threads can access the same resource (typically an object's instance variable)
  - This can produce corrupted data if one thread "*races in*" too quickly before another thread has completed its operation.

SoftEng

56

Preventing Race Conditions

- The individual steps that constitute the operation should be never split apart.
- It must be an atomic operation:
  - It is completed before any other thread can operate on the same resource
  - …regardless of the number or duration of individual steps

SoftEng

57

Preventing Race Conditions

- You can't guarantee that a single thread will stay running during the atomic operation.
- But even if the thread running the atomic operation moves in and out of the running state, no other running thread will be able to act on the same data.
- How to protect the data:
  - Mark the variables as **`private`**

  AND
  - Synchronize the code accessing the variables

SoftEng

58

Synchronization in Java

- The modifier **synchronized**
  - can be applied to a method or a code block
  - locks a code block: only one thread at a time can access it at a given time

```
void synchronized m1(){
  // synchronized context
}
```

```
void m2(){
  // normal (un-synchronized) context
  synchronized(anObject){
    // synchronized context
  }}
```

SoftEng

59

Synchronization and Monitor

- Every object in Java has a built-in *monitor*
- Before a thread can enter a synchronized context it must first acquire the lock of the object's monitor.
- Once a thread acquires a lock, it owns the lock until the thread itself release the lock
- Only one thread at a time can own a lock
  - If the lock is owned any thread attempting to acquire the lock is blocked until the lock has been released
  - Once a thread owns a lock on an object, no other thread can enter any of the synchronized methods in that object.
- When a thread exits a synchronized context it releases the lock

SoftEng

60

Synchronization and Monitor

- Not all methods in a class need to be synchronized.
  - Multiple threads can still access the class's non-synchronized methods
  - Methods that don't access the critical data, don't need to be synchronized
- A thread going to sleep, doesn't release locks
- A thread can acquire more than one lock, e.g.
  - A thread can enter a synchronized method
  - Then invoke a synchronized method on another object

SoftEng

61

Synchronize a code block

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

Is equivalent to this:

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

SoftEng

62

Synchronize a static method

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

Is equivalent to this:

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

`MyClass.class` represents asingle lock on the class which is different from the objects' locks

SoftEng

63

When to Synchronize?

- Two threads executing the same method at the same time may:
  - use different copies of local vars => no problem
  - access fields that contain shared data
- To make a thread-safe class:
  - methods that access changeable fields need to be synchronized.
  - Access to static fields should be done from static synchronized methods.
  - Access to non-static fields should be done from non-static synchronized methods

SoftEng

64

Example

> Returns a `List` whose methods are all synchronized and "thread-safe"

```
public class NameList {
private List names =
  Collections.synchronizedList(
                              new LinkedList());
public void add(String name) {
    names.add(name);
}
public String removeFirst() {
  if (names.size() > 0)
       return (String) names.remove(0);
  else return null;
} }
```

65

Example (II)

> As soon as the second thread tries to remove the first, it raises an IndexOutOfBoundsException

```
class NameDropper extends Th[illegible] {
 public void run() {
    String name = nl.removeFirst();
    System.out.println(name);
} }
public static void main(String[] args) {
  final NameList nl = new NameList();
  nl.add("Jacob");
  Thread t1 = new NameDropper(); t1.start();
  Thread t2 = new NameDropper(); t2.start();
}
```

66

Example (III)

- In a "thread-safe" class each individual method is synchronized.
  - Nothing prevents another thread from doing something else to the list *in between*
- Solution: **synchronize the code yourself !**

```
public class NameList {
private List names = new LinkedList();
public synchronized void add(String name) {
    names.add(name);
}
public synchronized String removeFirst() {
  if (names.size() > 0)
    return (String) names.remove(0);
  else  return null;
}  }
```

67

Deadlock

- Deadlock occurs when two threads are blocked, with each waiting for the other's lock.

⇒Neither can run until the other gives up its lock, so they wait forever

- Poor design can lead to deadlock
- It is hard to debug code to avoid deadlock

68

Thread Deadlock

```
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;
} } }
```

SoftEng http://softeng.polito.it

69

THREAD INTERACTIONS

SoftEng http://softeng.polito.it

70

Synchronization in `Object`

- **`void wait()`**
  - Causes current thread to wait until another thread invokes the **`notify()`** method or the **`notifyAll()`** method for this object.
- **`void notify()`**
  - Wakes up a single thread that is waiting on this object's lock.
- **`void notifyAll()`**
  - Wakes up all threads that are waiting on this object's lock.

SoftEng

71

Wait and Notify

- A thread can invoke a **`wait()`** on an object monitor
  - Provided it owns a lock on the object monitor
- A a result, the thread
  - Releases the lock
  - Is placed in a waiting pool
- When the thread is signaled
  - It wakes up
  - Tries to acquires back the lock
    - It is possibly blocked while any other owns the lock
  - Return from the wait method

SoftEng

72

Notify & NotifyAll

- The **notify()** method sends a signal to one of the threads that are waiting in the same object's waiting pool.
  - The **notify()** method CANNOT specify which waiting thread to notify.
- The method **notifyAll()** is similar but it sends the signal to all of the threads waiting on the object.

73

Example: await notification

```
Worker b = new Worker();
synchronized(b) {
  b.start();
  try {
    System.out.println("Waiting for b to complete");
    b.wait();
  } catch (InterruptedException e) {}
  System.out.println("Total is: ″ + b.total);
}
```

```
class Worker extends Thread {
  int total;
  public synchronized void run() {
    for(int i=0;i<100;i++)
          total += i;
     notify();
} }
```

74

Example: future notification

```
ExecutorService exec = Executors.newCachedThreadPool();

Future<Integer> out = exec.submit(task);
System.out.println("Waiting for b to complete");
try {
  System.out.println("Total is: " + out.get());
}catch(ExecutionException | InterruptedException ie) {}
```

```
Callable<Integer> task = () -> {
  int total = 0;

  for (int i = 0; i < 100; i++)
      total += i;
  return total;
};
```

SoftEng

75

Example: Java FIFO

```
import java.util.ArrayList;

public class FIFO<T>{
  private ArrayList<T> v;

  FIFO() {
    v = new ArrayList<T>(3);
  }
  public synchronized void
  insert(T e) {
    v.add(e);
    notify();
  }
```

```
public synchronized
T extract()
   throws Exception{
     T temp;
     if(v.size()==0)
       wait();
     temp=v.get(0);
     v.remove(0);
     return temp;
   }
}
```

SoftEng

76

Spontaneous Wakeup

- A thread may wake up even though no code has called **`notify()`** or **`notifyAll()`**
  - Sometimes the JVM may call **`notify()`** for reasons of its own,
  - Other class calls it for reasons you just don't know.
- When your thread wakes up from a **`wait()`**, you don't know for sure why it was awakened!
- Solution: putting the **`wait()`** method in a while loop and re-checking the condition:
  - We ensure that *whatever the reason we woke up, we will* re-enter the **`wait()`** only if the thing we were waiting for has not happened yet.

SoftEng

77

Example: Java FIFO

```
import java.util.ArrayList;

public class FIFO<T>{
  private ArrayList<T> v;

  FIFO() {
    v = new ArrayList<T>(3);
  }
  public synchronized void
  insert(T e) {
    v.add(e);
    notify();
  }

public synchronized
T extract()
  throws Exception{
    T temp;
    while(v.size()==0)
      wait();
    temp=v.get(0);
    v.remove(0);
    return temp;
  }
}
```

SoftEng

78

Livelock

- A livelock happens when threads are actually running, but no work gets done
  - what is done by a thread is undone by another
- Ex: each thread already holds one object and needs another that is held by the other thread.
- What if each thread unlocks the object it owns and picks up the object unlocked by the other thread ?
  - These two threads can run forever in lock-step!

SoftEng

79

Thread Starvation

- Wait/notify primitives of the Java language do not guarantee ***liveness*** (=> starvation)
- When **`wait()`** method is called
  - thread releases the object lock prior to commencing to wait
  - and it must be reacquired before returning from the method, post notification

SoftEng

80

Thread Starvation

- Once a thread releases the lock on an object (following the call to wait), it is placed in a object's wait-set
  - Implemented as a queue by most JVMs
  - When a notification happens, a new thread will be placed at the back of the queue
- By the time the notified thread actually gets the monitor, the condition for which it was notified may no longer be true …
  - It will have to wait again
  - This can continue indefinitely => Starvation

SoftEng

81

Synchronization objects

- **Semaphore**
  - Methods: **acquire()** and **release()**
- **CountDownLatch**
  - Methods: **await()** and **countDown()**
- **CyclicBarrier**
  - Methods: **await()**
  - Constructor accepts number of parties

    – All classes are in package **java.util.concurrent**

SoftEng

82

Summary

- Threads are concurrent execution contexts
  - Concurrency may be physical (e.g. multicore) or virtual (OS preemption)
- Threads are supported through the class **`Thread`** that can be
  - Extended with an overridden **`run`** method
  - Initialized with a **`Runnable`** object
- Once created, threads must be started

83

Summary

- Threads are assigned time slices
- A thread can hand over execution time by
  - **`sleep()`** that pauses the thread
  - **`yield()`** that gives another thread the opportunity to run
- A thread can be interrupted with the **`interrupt()`** method that makes the thread return from a waiting method with an **`InterruptedException`**

84

Summary

- Concurrent access to shared variables must be controlled
- Mutual exclusion is achieved by means of **`synchronized`** methods and code blocks
  - Using the monitor associated with any Java object

SoftEng

85

Summary

- Coordination between threads can be performed by
  - **`wait()`** that suspends the execution
    - This is an alternative to a busy form of waiting
  - **`notify()`** that wakes up a waiting thread

SoftEng

86