---

1. What is a race condition (”kapplöpning”)? Give an example.
svar:

Kapplöpning innebär att flera trådar samtidigt arbetar med samma data, därmed resultatet kan bli felaktigt på grund av att trådarna inte är överens om datats aktuella värde.

---

2. What does it mean to say that a class is thread-safe?
Svar:

En klass är tråd säkert när den fortsätter bete sig korrekt när den nås från flera trådar (utan utvecklaren behöver synkronisera eller koordinera sitt kod).

---

3. Can a class always be made thread-safe simply by declaring all methods synchronized? (We assume all attributes to be private and/or final.) Motivate your answer. Can you come up with an example to make your point?

Svar: nej, example

```
List<String> list = new Vector<String>();

Thread t1 = new Thread(() -> {
  // ...
  int n = list.indexOf("cowabunga");
  list.remove(n);  // remove "cowabunga" from list
  // ...
});

Thread t2 = new Thread(() -> {
  // ...
  list.add(0, "turtles");
  // ...
});
```

Trots vektor har alla metoder synkroniserad kan de ändå bli fel om trådbyte sker exakt efter , “int n = list.indexOf(“cowabunga”);” Om en annan tråd lägger till eller tar bort en element i början av listan kommer index av “cowabunga” att ändras. Därmed kommer “list.remove(n)” ta bort fel element.
Lecture 5 7/24.

---

4. Intrinsic locks (i.e., the synchronized locks) are reentrant. As the name suggests, ReentrantLockobjects are too.

a. What does ”reentrant” mean in this context?

b. Why are reentrant locks useful?
Svar:
a)Det menar att vi kan vara säkra på att programmet kommer att bete sig korrekt när den nås via fler tråd.
In computing, a computer program or subroutine is called reentrant if multiple invocations can safely run concurrently.

b) För skapa mutex (mutual exclusion)/ atomic action så att en resurs enbart nås av en tråd åt gången.
“We sometimes call a lock a "mutex", short for mutual exclusion. We can use a mutex (or lock) to ensure a particular resource is accessed by one thread at a time.”
Lecture 2 8/26.

---

5. We can think of many ways of implementing a BankAccount class. All of the following implementations work correctly when used by a single thread. For each of the following implementations, determine whether it is thread-safe. Motivate your answer. (Assume that a BankAccount object can be accessed by two or more threads concurrently.)

a.

```
public class BankAccount {
  private long balance = 0;

  public void deposit(long n)  { balance += n; }
  public void withdraw(long n) { balance -= n; }
  public long getBalance()     { return balance; }
}
```

Svar:
Nej,
+=,-= är egentligen 3 instruktioner, read-update-write. Om tråd 1 enbart med hinner read-update men inte write och en kontextbyte sker exakt efter update kommer de medföra att tråd 2 läser in fel värde av balance.

b.

```
public class BankAccount {
  private long balance = 0;

  public synchronized void deposit(long n)  { balance += n; }
  public synchronized void withdraw(long n) { balance -= n; }
  public synchronized long getBalance()     { return balance; }
}
```

Svar:
Ja, Eftersom klassen är en monitor. Nyckelordet synchronized medför att alla metoder blir mutex och balance är privat, således blir trådsäker.

c.

```
public class BankAccount {
  private volatile long balance = 0;

  public void deposit(long n)  { balance += n; }
  public void withdraw(long n) { balance -= n; }
  public long getBalance()     { return balance; }
}
```

Svar:
Nej, volatile variable innebär enbart att:
The value of this variable will never be cached thread-locally: all reads and writes will go straight to "main memory";
Till Skillnad synchronization kan volatile inte användas för mutual exclusion:
Because accessing a volatile variable **never holds a lock**, it is **not suitable** for cases where we want to ***read-update-write*** as an atomic operation (unless we're prepared to "miss an update");
Därmed blir klassen fortfarande inte trådsäker.
https://www.javamex.com/tutorials/synchronization_volatile.shtml
lecture 5 19/24.

Kort svar:
Nej, volatile variable innebär enbart att ändringar till variabeln syns i alla trådar men garanterar inte mutex (mutual exclusion). Därmed finns fortfarande risk att flera trådar läser och skriver till variabeln samtidigt.

d.

```
public class BankAccount {
  private AtomicLong balance = new AtomicLong(0);

  public void deposit(long n)  { balance.addAndGet(n); }
  public void withdraw(long n) { balance.addAndGet(-n); }
  public long getBalance()     { return balance.get(); }
}
```

Svar
Ja, den är trådsäker eftersom alla metoder i bankAccount använder AtomicLong (på korrekt sätt) och AtomicLong är trådsäker.

6. Consider the following four techniques for thread safety:

a. thread confinement
b. instance confinement,
c. stack confinement, and
d. Immutability.

Describe each of these in one or two sentences. For each technique, explain how it guarantees thread safety.
Svar:
a) One way to ensure thread safety is to not share data between threads and to ensure data is only accessed from a single thread. This technique is called thread confinement,
Sida 28, 3.3.

Svar på svenska:
Ett sätt att säkerställa tråd säkerhet är att inte dela data mellan trådar och att se till att data endast nås från en enda tråd.

b)
Encapsulating data with in an object confines access to the data to the object's methods,making it easier to ensure that the data is always accessed with the appropriate lock held.
Sida 39, 4.2.

Svar på svenska:
Instance confinement går ut på att kapsla in data så att alla anrop till datan sker med hjälp från en objekt som har trådsäkra metoder.

c)
Stack confinement is a special case of thread confinement in which an object can only be reached through local variables.
Sida 29, 3.3.2

Svar på svenska:
Är ett speciellt fall av thread-confinement där ett objekt endast kan nås via lokala variabler.

d) Nearly all the atomicity and visibility hazards, have to do multiple threads trying to access the same mutable state at the same time. Immutable objects are stateless*, thus thread-safe.

* An immutable object is one whose state cannot be changed after construction.
Sida 31, 3.4.

Svar på svenska:
Oföränderlig object är automatisk trådsäker eftersom den är tillståndslös.

7. Consider the terms in above (task 6) again.
a. Which of the terms 6a–d corresponds most closely to a monitor?
b. Which of the terms 6a–d is used with the Swing EDT?

Svar:
a) instance confinement
b) thread confinement

8. The BlockingQueue interface is implemented by ArrayBlockingQueue and LinkedBlockingQueue. The former has a bounded size, whereas the latter does not. Consider the call

```
q.put(m);
```

a. Can the put() call block if q is an ArrayBlockingQueue? (Feel free to consult the documentation.)

b. Can the put() call block if q is a LinkedBlockingQueue?

c. Suggest one possible advantage, and one possible disadvantage, of a blocking put().

Svar:
Def put()= Inserts the specified element into this queue, waiting if necessary for space to become available.
I LinkedBlockingQueue är max kapacitet samma som `Integer.MAX_VALUE`.
a) ja
b) ja.
c) fördel : det blir trådsäkert om vi har en tråd som lägger in task och en annan som utför de.
Nackdel: vi kan behöva vänta lång tid ibland.

9. What is InterruptedException? In which situations is it thrown?
Svar:
InterruptedException är en exception som (oftast) signalerar till tråden att avsluta exekveringen så snabbt som möjligt (och att rensa upp efter sig om behövs). InterruptedException kastas av blockerande metoder som t.ex. wait(),sleep(),put() etc* då interrupted status är sant.

* Thread.interrupt() sätter enbart “interrupted status till sant”. Inom blockerande metoder används förmodligen “if(`Thread.interrupted()`) throw new InterruptedException() “

Calling interrupt() instructs a thread to
 > stop whatever it’s doing, "as soon as possible"
> perform any final clean-up work, if necessary
> exit (typically return from run())
Lecture 7

https://www.yegor256.com/2015/10/20/interrupted-exception.html

10. Suppose one thread interrupts another as follows:

```
1    t1.interrupt();
2    System.out.println("t1 has now been interrupted");
```

The thread t1 has been implemented to terminate when interrupted. However, when we run the program, the printout (line 2 above) often appears before t1 actually terminates.
a. Why is interrupt()not immediate? What must happen for t1 to terminate?

b. Add one line above to ensure that line 2 is not executed until t1 has actually terminated before the printout is made.

Svar:
a)
Interruption is asynchronous: the interrupt()ed thread doesn’t terminate immediately.
First, it has to be ***scheduled for execution***; then, the work outlined above may take some (preferably short) time.
Lecture 7 18/22.

Svar svenska:
Interruption sker asynkront. Utöver anropet till interrupt() måste tråden först schemaläggas för körning innan de kan avbrytas.

b)
1 t1.interrupt();
2 t1.join(); // wait for t1 to actually stop
3 System.out.println(“t1 has now been ..”);
Lecture 7 19/22

11. Suppose thread t2 runs the following code:

```
1  Queue q = new Queue();
2  // ...
3  public static void t2run() {
4    try {
5      while (true) {
6        String m = q.fetch();
7        System.out.println("received " + m);
8      }
9    } catch (InterruptedException e) {
10     System.out.println("terminated");
11   }
12 }
13
14 Thread t2 = new Thread(() -> t2run());
```

The monitor q is an instance of class Queue:

```
public class Queue {
  private List<String> list = new LinkedList<String>();

  public synchronized String fetch() {
    while (list.isEmpty()) {
      try {
        wait();
      } catch (InterruptedException e) {
        // not my problem
      }
    }
    return list.remove(0);
  }

  public synchronized void put(String m) {
    list.add(m);
    notifyAll();
  }
}
```

Another thread now attempts to interrupt t2, just like in the previous example. However,t2 doesn't respond to the interruption: the printout ”terminated” never appears.
a. Why doesn’t the InterruptedException reach t2 run()?

b. Modify Queue to rectify the problem, by removing a few lines, and changing another. (No changes to t2 or t2run()should be necessary.)When the following is executed

```
t2.interrupt();
```

Thread t2 should now execute its catch clause (line 10) and print ”terminated”.

Svar
a)
De pga interrupted fångas i Queue och kastas inte vidare till T2, därmed kommer rad 9 i t2run alldrig att köras.

b) ta bort try-catch blocket i fetch() och lägg till throws InterruptedException till metoden fetch().

12. A server application includes 1000 tasks, all implementing the Runnable interface. Each task requires significant computation (minutes of processor time), and is launched when the method launch() below is executed.

```
public class Server {
  /** Called whenever a new task arrives */
  public void launch(Runnable task) {
    // to be implemented
  }
}
```

For the implementation of launch(), consider two alternatives:
a. We could start a new thread for each task, like this:

```
public class Server {
  public void launch(Runnable task) {
    Thread t = new Thread(task);
    t.start();
  }
}
```

b. Or we could submit the task to a thread pool with a limited number of threads, like this:

```
public class Server {
  private ExecutorService pool = Executors.newFixedThreadPool(4);

  public void launch(Runnable task) {
    pool.submit(task);
  }
}
```

We assume the number of threads above (4) to match the computer's capacity for simultaneous threads (the number of cores or threads in the hardware).

In both solutions, the server turns out to be fully utilized – the computer is completely busy with task execution. Nevertheless, there is a performance difference between the two alternatives : when given the same (large) set of tasks, one of them always completes before the other.

Which solution is the more efficient one here, when we consider throughput (number of tasks completed per second) ? Motivate your answer.

Svar (asmail):
På grund task tar minuter att bli klara medför alternativ b) att högst 4 kan bli klara samtidigt dvs 4 per sekund medans i a finns de ingen gräns hur många kan bli klara samtidigt. Därför ger a) högst throughput.

svar(mohammad):
12 threadpool, ty mindre kontextbyten och scheduling overhead.
https://stackoverflow.com/questions/3733427/why-net-threadpool-is-used-only-for-short-time-span-tasks
https://stackoverflow.com/questions/10298641/is-it-true-that-for-long-running-processes-it-is-better-to-do-thread-manually-in

Sida 73, 6.1.3.
Lecture 6 5/22

13. In the previous assignment, tasks were long-running (minutes). Suppose we also introduce occasional short-running tasks (say, less than 10 milliseconds). The server still runs a great number of long-running tasks, and occasionally also a short-running one. Again, consider the two alternatives 12a and 12b. Which solution would give the shortest average response times for such occasional, short-running tasks? (Here, we can define the response time as the time from the call to launch()to the completion of the task.)

Svar(asmail):

b) är snabbare än a) eftersom fler trådar leder till fler kontextbyte dvs mer tid spenderas på schemaläggning än köra koden. Därmed kommer påbörjat task ta relativ lång tid att bli klar och speciellt gäller de när har många korta task.

svar(mohammad):
13 1000 trådar, ty threadpoolen kan fastna på 4 long running tasks => icke responsivt.

14. A single-thread ”pool” (ExecutorService) can be created as.

```
ExecutorService pool = Executors.newSingleThreadExecutor();
```

How can such a single-threadExecutorServicebe useful? (Hint: think about thread safety.)

Svar:
Thread-confinement. Alla objekt som blir “submitted” till pool kommer ändras endast av en tråd åt gången.
15. Thread confinement is commonly applied in user-interface frameworks such as Swing. How?
svar:
Everything related to with Swing GUI must be submitted through EDT, thus ensuring (Swing related) data is only accessed from a single thread.

16. What does the method SwingUtilities.invokeLater do? What is it used for?
Svar:
SwingUtilities.invokeLater() submits a task to the EDT. The EDT runs the task as soon as possible, that is, when the preceding tasks are done . SwingUtilities.nvokeLater() is used for updating Swing GUI components.

17. What is the output of the following lines?

```
SwingUtilities.invokeLater(() -> {
  if (SwingUtilities.isEventDispatchThread()) {
    System.out.println("X");
  } else {
    System.out.println("Y");
  }
});
```

Svar:
“X”, since the code inside the block, ie inside invokeLater, will be evaluated by EventDispatchTherad we will get X. we use invokeLater explicitly run the code inside in Event Dispatch Thread

Svar på svenska:
Eftersom koden inuti blocket invokeLater, kommer att utvärderas av EventDispatchThread får vi “X”. vi använder invokeLater explicit för köra koden inuti Event Dispatch Thread.

---

18. Lock-free (or non-blocking) data structures attain thread safety without using blocking or locks.

a. State one advantage of lock-free data structures.
b. How do lock-free data structures work? What synchronization mechanism(s) do they rely on? Explain the idea in a few (two–three) sentences.

Svar
a)There are no deadlock.

b)The typical pattern for using CAS is first to read the value A from V, derive the new value B from A, and then use CAS to atomically change V from A to B so long as no other thread has changed V to another value in the meantime. CAS addresses the problem of implementing atomic read-modify-write sequences without locking, because it can detect Interference from other threads.
Sida 197. 15.2.1

In computer science, compare-and-swap (CAS) is an atomic instruction used in multithreading to achieve synchronization. It compares the contents of a memory location with a given value

and, only if they are the same, modifies the contents of that memory location to a new given value.
Från wiki

Svar på svenska:
Lock-free data struktur bygger på en atomisk compare-and-swap instruktion finns till processorn och anropas av JVM (då en atomisk klass används). Lock-free data går ut på att se om värdet hos en atomisk variabel har ändrats (av annan tråd än sin egen) sedan senaste gången variabel kopierades. Om Ingen ändring har gjorts till variabeln (sedan senaste gången variabel kopierades) kan den ersättas med en ny värde (med CAS instruktion). Om en ändring har gjorts kopieras senaste värden och processen repeteras.

---

https://coderanch.com/t/679820/java/Compare-Swap-Implementation

19. Consider Goetz' example 15.6 (page 331). Suppose two threads execute push()at the exact same time.

What happens? How can we be sure that the stack remains in a consistent state, and that both push()ed elements end up in the stack (as they should)? Explain!

Svar:
Exempel hur de kan gå till

	T1	T2
1	Node<E> newHead = new Node<E>(item);	Node<E> newHead = new Node<E>(item);
2	Node<E> oldHead;	Node<E> oldHead;
3	oldHead = top.get();	
4	newHead.next = oldHead;	
5	! top.compareAndSet(oldHead, newHead) // ger true dvs	
6	// dvs while stopas	oldHead = top.get();
7		newHead.next = oldHead;
8		top.compareAndSet(oldHead, newHead) *1 ger false (while fortsätter)
9		oldHead = top.get();
		newHead.next = oldHead;
		! top.compareAndSet(oldHead, newHead) ger sant (while stopas)

Vad som kan hända är
1) oldHead == top.get() dvs ingen tråd har ändrat på tops element sedan senaste gången oldHead tilldelas värde. Då kommer while-loopen att stoppas och top uppdateras med newHead.

2) oldHead != top.get() dvs en tråd har ändrat på tops tillstånd sedan senaste gången oldHead tilldelas värde. Då kommer top inte uppdateras med newHead. Därmed fortsätter while-loopen och processen repeteras om tills oldHead == top.get() är sant.;

How can we be sure that the stack remains in a consistent state:Ingen ändring kan ske till huvodNodet om inte har identisk kopia av den.

Sida 203

20. In the same example (15.6), the nested class Node has two attributes item and next. Neither is volatile, and synchronized is not used.

a. How can we be sure that new item values are visible to other threads?

b. How can we be sure that updated next values are visible to other threads?

Svar:
a) “AtomicReference refers to an object reference. This reference is a volatile member variable in the AtomicReference instance as below.”
Eftersom den nya Node-objektet spares inuti AtomicReference som en volatile klassvariabel.

Fråga: finns skillnad de mellan att göra all attribute volatile i en klass jämfört med göra en instansen volatile?
https://dzone.com/articles/how-atomicreference-works-in-java
b) Eftersom next är en attribute sparad i Node

21. This task is about thread scheduling, and how it differs between an RTOS (real-time operating system, with strict priorities) and a desktop operating system such as Windows, Mac, or Ubuntu Linux. Consider the threads hi and lo below.

```
1  Semaphore sem = new Semaphore(0);
2
3  Thread hi = new Thread(() -> {
4    // ...
5    sem.acquire();
6    System.out.println("hi");
7    // ...
8  });
9
10 Thread lo = new Thread(() -> {
11   // ...
12   sem.release();
13   System.out.println("lo");
14   // ...
15 });
16
17 hi.setPriority(Thread.MAX_PRIORITY);
18 lo.setPriority(Thread.MIN_PRIORITY);
19 hi.start();
20 Thread.sleep(1000);   // ensure thread 'hi' actually reaches line 5 before we continue
21 lo.start();
```

a. If these threads run on a regular operating system (such as Windows), in which order will the printouts (lines 6 and 13) appear? Is the order guaranteed to be the same every time?

b. If these threads run on an RTOS (with strict priorities), in which order will the printouts appear? Is the order guaranteed to be the same every time?
Note:assume a single-core processor.

Svar:
In an RTOS, sem.release() will lead to a context switch
> Assume hi has higher priority than lo.
> Assume hi reaches sem.acquire() before lo starts executing.

Svar svenska:
I en RTOS kommer sem.release () att leda till en trådbyte och om följande är sant:
> Antag att hi har högre prioritet än lo.
> Antag att hi når sem.acquire () innan lo börjar köras.

a)
“Hi”
“Lo”
Eller
“Lo”
“Hi”

b)
“Hi”
“Lo”

22. Suppose we have a real-time system with four threads running in an RTOS, using strict priorities ,scheduled according to the RMS (Rate Monotonic Scheduling) strategy. For each thread i, we know the worst-case execution time C_i and the execution period T_i. We assume that the thread’s deadline equals the period – that is, the thread must complete its work before the next time it needs to execute. We now calculate U as

$$U = \sum_{i=1}^{4} \frac{C_i}{T_i}$$

For which values of U can we be sure that the system is schedulable – that is, that all threads will meet their deadlines? (Assume a single-core processor.)

Svar:

We can guarantee schedulability if the sum C_1/T_1 + C_2/T_2 + … + C_n/T_n stays below the upper bound U:
Svenska
Alla trådar kommer att uppfylla sina deadline om följande är sant:
Summan av C_1/T_1 + C_2/T_2 + … + C_n/T_n blir mindre än övre gräns till U:

$$\sum \frac{C_i}{T_i} \quad < \quad n(2^{1/n} - 1) = U_n$$

“N = number of thread” U_n = C_n/T_n. Där C= execution time, T = period, U =C/T=Cpu utilization.

23. What is priority inversion? Explain in one or a few sentences.
svar:
In computer science, priority inversion is a problematic scenario in scheduling in which a high priority task is indirectly preempted by a lower priority task effectively "inverting" the relative priorities of the two tasks.

Wikipedia. Finns också på lecture 9.

Prioritetsinversion är ett fenomen som uppstår då en högprioriterad tråd blir blockerad av en lägre prioriterad tråd utan att de (för tillfället) delar på någon resurs.

Exempel: En lågprioriterad tråd exekverar och tar en resurs. En högprioriterad tråd börjar exekvera och försöker i sin tur ta resursen, vilket misslyckas eftersom den är låst. Den högprioriterade tråden blockerar i väntan på att resursen ska släppas av den lågprioriterade tråden. Under tiden blir en tredje, mellan prioriterad tråd körbar och tar över CPU:n i kraft av sin prioritet och blockerar den lågprioriterade tråden.Indirekt blockeras därmed även den högprioriterade som får vänta på den lägre prioriterade (mellan prioriterade) tråden.
Tenta 180823 uppgift 1

---

24. What is a distributed system? Explain in one or a few sentences.

Svar:

A *distributed system* is a system whose components are located on different networked computers, which communicate and coordinate their actions by message passing.

Det är ett system där deras komponent är inte direkt kopplade utan kopplade via nätverk, de kommunicerar med varandra via message passing.

---

25. When we design software, we often don’t need to consider hardware failures. For a distributed system, however, we must. Why?

svar:
Eftersom systemet komponent ligger på ett nätverk och inte direkt kopplade då måste vi ta hänsyn till att programmet kommer att köras även om att det inte har fått något information från andra enheter.

---

26. Why is message-passing useful in distributed systems?

svar:
It is easier to build parallel hardware using message passing model as it is quite tolerant of higher communication latencies.

Message passing model allows multiple processes to read and write data to the message queue without (***directly***) being connected to each other.

Svar på svenska
Det är lättare att bygga parallell hårdvara med hjälp av “message passing modell” eftersom det är tolerant för högre kommunikations fördröjning. “Message passing model” tillåter också flera processer att läsa och skriva data till kön utan (direkt) att vara anslutna till varandra.

https://www.tutorialspoint.com/message-passing-model-of-process-communication

---

---

27. Virtually all computers use cache memories. On a multi-core computer, different threads may observe different values for the same variable. How can this happen? (Remember that on a multi-core computer, different threads may run on different cores.)

svar:
Moderna dator har flera cacheminne. Därmed måste varje cacheminne ha en egen kopia variabelns värde. Därmed finns de risk att ha “gamla” variabel värde.

Påhittad svar 2:
Eftersom moderna dator använder flera cachmine finns flera kopierar av samma variabel på olika cacheminne. När en kärna ändrar variabelns värde i en cacheminne reflekteras ändringen inte direkt för alla andra cacheminne. Därför finns en risk att en annan kärna läser in en föråldrad värde.

---

28. Consider a situation where we use transactional memory instead of locks.

a. Deadlock is no longer possible. Why?

b. Instead, there is a greater potential for livelock. How?

Svar:
A)transactional memory: do a group of load and store instructions to execute in an atomic way, which means that the programmer is not responsible for explicitly identifying locks or the order in which they are acquired, programs that utilize transactional memory cannot produce a deadlock.

Svenska

Eftersom programmeraren inte är explicit ansvarig för identifiera lås eller i vilken ordning de blir “acquired”, kan program som använder transactional memory inte producera ett dödlås.

B)
For example, longer transactions may repeatedly revert in response to multiple smaller transactions, wasting both time and energy

---