The threat sleep method instructs the operating system not to schedule the current thread until that time passes. **During that time, this thread is not consuming any CPU.**

Thread class - Encapsulates all thread-related functionality

Two ways to run code on a new thread
A- Implement a Runnable interface, and pass it to a new Thread object
B- Extend the Thread class, and create an object of that class

Thread Termination - Why and When?

- **Threads consume resources**
  - **Memory and kernel resources**
  - **CPU cycles and cache memory**

- **If a thread finished its work, but to app is still running, we want to clean up the thread's resources**
- **If a thread misbehaving we want to stop it**
- **By default, the app will not stop at least one thread is still running**

Daemon Threads:

Background threads do not prevent the application from exiting if the main thread terminates.

Thread.join() ->

- ☐ **More control over independent threads**
- ☐ **Safely collect and aggregate results**
- ☐ **Gracefully handle runaway threads using Thread.join(timeout)**

```java
BigInteger pow(BigInteger base, BigInteger exponent) {
  BigInteger result = BigInteger.ONE;
  while (exponent.signum() > 0) {
    if (exponent.testBit(0)) result = result.multiply(base);
    base = base.multiply(base);
    exponent = exponent.shiftRight(1);
  }
  return result;
}
```

signum

```
public int signum()
```

Returns the signum function of this BigInteger.

Returns:

-1, 0 or 1 as the value of this BigInteger is negative, zero or positive.

Latency -> The time to completion of a task. Measured in time units.

How can we reduce latency with multithreading programming?

Latency = T/N

Theoretical reduction of latency by N = Performance improvement by a factor of N.

Can we break any task into subtasks? no , not every tasks into parallel

Hyperthreading = A single physical core can run two threads at a time which is achieved by having some hardware units in a physical core duplicated. (Threading Technology is a hardware innovation that allows more than one thread to run on each core. More threads mean more work can be done in parallel.)

Throughput -> The amount of tasks completed in a given time period. Measured in tasks/time unit.

Pixels and Color Space Background - Start

In digital imaging, a Pixel represents the smallest element of a picture displayed on the screen.

An image is nothing more than a 2-dimensional collection of Pixels.

The color of a pixel can be encoded in different ways.

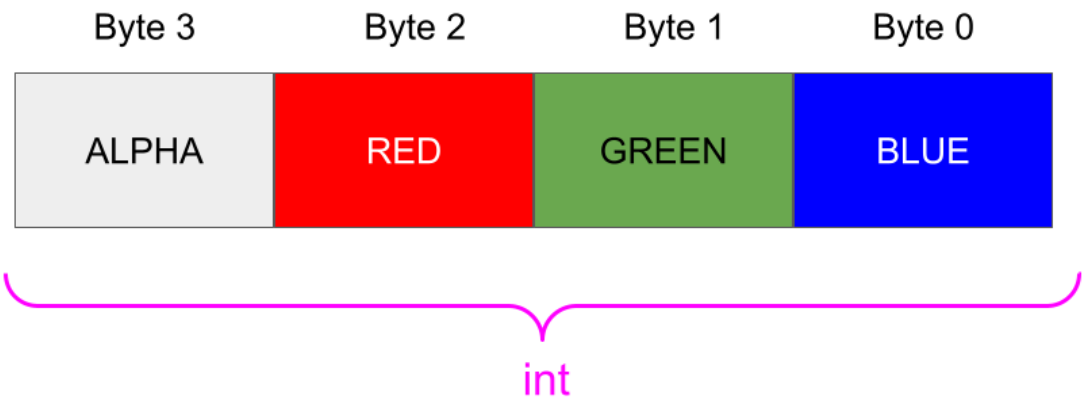
A few frequently used groups of pixel color encoding are:

- **Y'UV - Luma (brightness), and 2 chroma (color) components**
- **RGB - Red, Green, Blue**
- **HSL and HSV - Hue, Saturation, Lightness/Brightness**
- **CIE XYZ - Device independent Red, Green and Blue**

ARGB Memory Representation

The format used in our Image Processing example is a version of the RGB family called ARGB, where A stands for alpha (transparency)

The representation of this color in memory is as follows:



As we can see, each component is represented by 1 byte (8 bits) so the value of each component is in the range of 0 (0x in hexadecimal) and 255 (0xFF in hexadecimal)

Since we have 4 bytes, we can store the entire color of a pixel in a variable of type `int`.

Component Extraction Code Explanation

In the Image Processing example we have the following methods that extract individual components of a pixel:

```
public static int getRed(int rgb) {
   return (rgb & 0x00FF0000) >> 16;
}

public static int getGreen(int rgb) {
   return (rgb & 0x0000FF00) >> 8;
}

public static int getBlue(int rgb) {
   return rgb & 0x000000FF;
}
```

Let's explain each method, in particular the math that happens to get each color component.

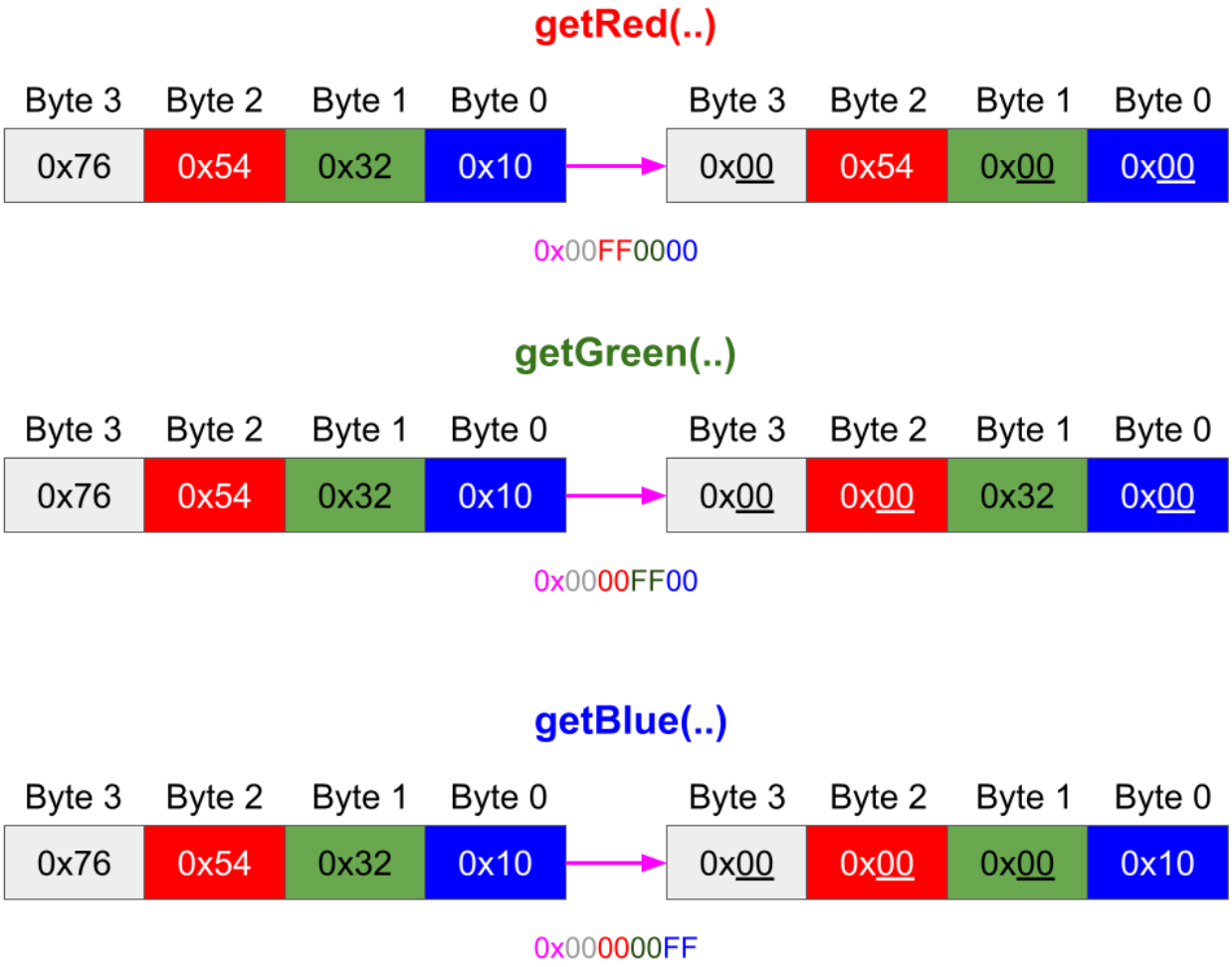
In order to get a particular component (red, green, or blue), we need to first get rid of all the other color components in the pixel, while keeping the desired component.

To achieve this we apply a bitmask.

A bitmask defines which bits we want to keep, and which bits we want to clear.

We apply a bitwise AND with 0x00 (0000 0000 in binary) to get rid of a component since X AND 0 = 0, for any X.

We apply a bitwise AND with 0xFF (1111 1111 in binary) to keep the value of a component since X AND 1 = X, for any X.



However, after applying a bitmask we are not done. We still need to shift the byte representing our component to the lowest byte.

For example in the `getRed(..)` method, after we apply the bitmask on 0x76543210 we end up with 0x00540000, but what we need is 0x00000054

So we need to shift all the bits in the result of the bitmask to the right., using the >> operator.

- **For the blue color extraction, we don't need to perform any shifting since it's already the right-most byte.**
- **For the green color extraction, we need to move all the bits 1 byte (8 bits) to the right.**
- **For the red color extraction, we need to move all the bits 2 bytes (16 bits) to the right.**

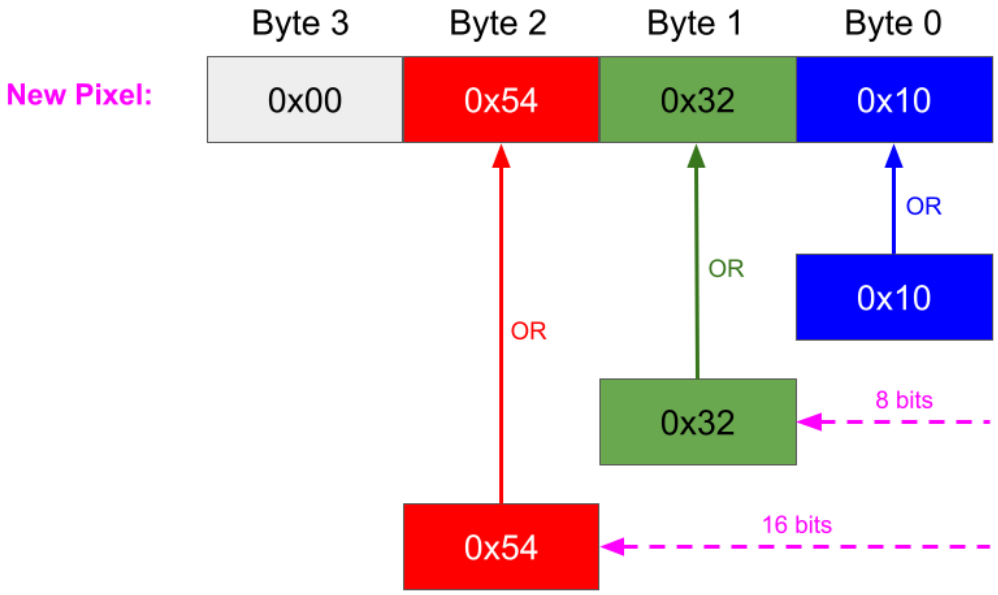
Combining Color Components into a Pixel

When building a pixel's color from individual red, green and blue components we had the following method:

```
public static int createRGBFromColors(int red, int green,
int blue) {
    int rgb = 0;

    rgb |= blue;
    rgb |= green << 8;
    rgb |= red << 16;

    rgb |= 0xFF000000;

    return rgb;
}
```

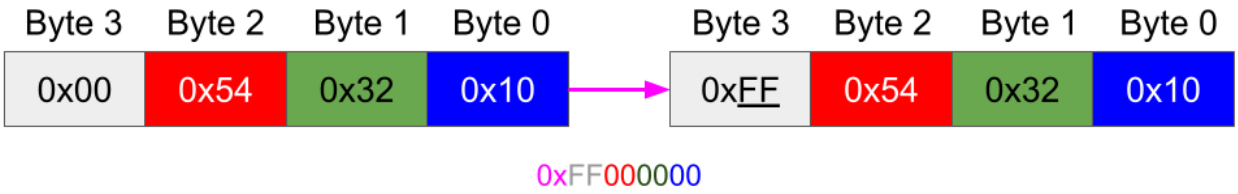
In the above code, we perform the opposite of color component extraction. We take each component and shift it to the right place in the ARGB pixel representation.

- **Blue is placed at the lowest byte so we simply bitwise OR the pixel color representation with the blue component**
- **Green needs to be placed at the second byte so it is first shifted 1 byte (8 bits) to the left, and then is bitwise ORed with the pixel color**
- **Similarly, red needs to be placed at the third byte so its component is shifted 2 bytes (16 bits) to the left, and then it is bitwise ORed with the pixel color**



The final step is to set the transparency level to the highest, making the color completely opaque (0 levels mean fully transparent, 255 means fully opaque).

That is achieved by setting the left-most byte, representing the alpha component to 0xFF which is 1111 1111 in binary.



Pixels and Color Space Background - END

Thread Pooling

Create threads and use them later. Fixed Thread Pool

Apache Jmeter -> Java Performance Tool

Response code 200 -> OK (HTTP)

Good job!
That's correct! Since the threads are not in the "running" state, all the time while serving the incoming requests, we may have all of the threads blocked on IO (waiting for a response from the database), but the CPU is not actually executing any tasks). So if we create more threads to handle the incoming requests, we will get better throughput. There is no way of knowing the best number of threads ahead of time since more threads means more requests can be handled, but also more overhead and context switching. So we need to perform a load test

Question 2:

We are running an HTTP server on a single machine.

Handling of the HTTP requests is delegated to a fixed-size pool of threads.

Each request is handled by a single thread from the pool by performing a **blocking** call to an external database which may take a **variable duration**, depending on many factors.

After the response comes from the database, the server thread sends an HTTP response to the user.

Assuming we have a 64 core machine.

What would be the optimal thread pool size to serve the HTTP request?

- ○ 16
- ○ 64
- ○ 128
- ◉ **There is no way of knowing, since the tasks include a blocking call. But definitely more than 64**

Memory Management Fundementals

Stack Memory Region:

- ☐ Methods are called
- ☐ Arguments are passed
- ☐ Local variables are stored

Stack + Insturcution Pointer = State of each thread's execution

Stack Frame -> A stack frame is comprised of:

- **Local variables, Local references**
- **Saved copies of registers modified by subprograms that could need restoration**
- **Argument parameters**
- **Return address**

246

A stack frame is a frame of data that gets pushed onto the stack. In the case of a call stack, a stack frame would represent a function call and its argument data.

If I remember correctly, the function return address is pushed onto the stack first, then the arguments and space for local variables. Together, they make the "frame," although this is likely architecture-dependent. The processor knows how many bytes are in each frame and moves the stack pointer accordingly as frames are pushed and popped off the stack.

EDIT:

There is a big difference between higher-level call stacks and the processor's call stack.

When we talk about a processor's call stack, we are talking about working with addresses and values at the *byte/word level* in assembly or machine code. There are "call stacks" when talking about higher-level languages, but they are a debugging/runtime tool managed by the runtime environment so that you can log what went wrong with your program (at a high level). At this level, things like line numbers and method and class names are often known. By the time the processor gets the code, it has absolutely no concept of these things.

Stack's properties:

- ☐ **All variables belong to the thread executing on that stack.**
- ☐ **Statically allocated when the thread is created**
- ☐ **The stack's size is fixed, and relatively small (platform-specific)**
- ☐ **If our calling hierarchy is too deep. We may get a StackOverflow Exception. (Risky with recursive calls)**

Heap Memory Region:

- ☐ Shared memory location belongs to the process
- ☐ Class members are allocated on the heap and therefore, are shared among threads

What is allocated on the Heap?

- ☐ Objects (anything created with the new operator)
  - ☐ String
  - ☐ Object
  - ☐ Collection
  - ☐ Etc…
- ☐ Members of classes
- ☐ Static variables

Heap Memory Management

- ☐ Governed and managed by Garbage Collector
- ☐ Objects - stay as long as we have a reference to them.
- ☐ Members of classes - exist as long as their parent objects exist (same life cycle as their parents)
- ☐ Static variables - stay forever (remain app lifetime)

Objects vs References

```
Object referenceVar1 = new Object();
Object referenceVar2 = referenceVar1;
```

References

- ☐ Can be allocated on the stack
- ☐ Can be allocated on the heap if they are members of a class

Objects

- ☐ Always allocated on the heap!!

Resource Sharing Between Threads:

- **Variables**
- **Data Structures**
- **File or connection handles, work queues**
- **Objects**

Atomic Operation

- **An operation or a set of operations is considered atomic if it appears to the rest of the system as if it occurred at once.**
- **Single step - `all or nothing**
- **No intermediate states**

Synchronized - Lock

- **A synchronized block is Reentrant**
- **A thread cannot prevent itself from entering a critical section.**

17.7. Non-atomic Treatment of double and long

For the purposes of the Java programming language memory model, a single write to a non-volatile long or double value is treated as two separate writes: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64-bit value from one write, and the second 32 bits from another write.

Writes and reads of volatile long and double values are always atomic.

Writes to and reads of references are always atomic, regardless of whether they are implemented as 32-bit or 64-bit values.

Some implementations may find it convenient to divide a single write action on a 64-bit long or double value into two write actions on adjacent 32-bit values. For efficiency's sake, this behavior is implementation-specific; an implementation of the Java Virtual Machine is free to perform writes to long and double values atomically or in two parts.

Implementations of the Java Virtual Machine are encouraged to avoid splitting 64-bit values where possible. Programmers are encouraged to declare shared 64-bit values as volatile or synchronize their programs correctly to avoid possible complications.

Atomic Operations - Primitive Types

- **All assignments to primitive types are safe except long and double.**

RACE CONDITION

- **When multiple threads are accessing a shared resource**
- **At least one thread is modifying the resource**
- **The timing of threads scheduling may cause incorrect results**
- **The core of the problem is non-atomic operations performed on the shared resource.**

DATA RACE

- **Data races are a common problem in multithreaded programming. Data races occur when multiple tasks or threads access a shared resource without sufficient protection, leading to undefined or unpredictable behavior.**
- **Compiler and CPU may execute the instructions OUT OF ORDER to OPTIMIZE PERFORMANCE and UTILIZATION.**
- **They will do so while maintaining the logical correctness of the code.**
- **OUT OF ORDER execution by the compiler and CPU are important features to speed up the code.**
- **The compiler re-arranges instructions for better**
    - **Branch prediction (optimized loops.. If statements etc.)**
    - **Vectorization - parallel instruction execution (SIMD)**
    - **Prefectching instructions - better cache performance**
- **CPU re-arranges instructions for better hardware units utilization**

Solutions

- **Synchronization of methods which modify shared variables**
- **Declaration of shared variables with the volatile keyword**

Every shared variable (modified by at least one thread) should be either

- **Guarded by a synchronized block (or any type of lock)**

> **Good job!**
> This is a very advanced question. Good job!

Question 1:

Note* : This is a more advanced question. Try to answer it yourself before looking at the options below.

We have the following code snippet:

```
     public static void main(String[] args) {
         SharedClass sharedClass = new SharedClass();

         Thread thread1 = new Thread(() -> sharedClass.method1());
         Thread thread2 = new Thread(() -> sharedClass.method2());

         thread1.start();
         thread2.start();
     }

    private static class SharedClass {
         int a = 0;
         int b = 0;

         public void method1() {
             int local1 = a;
             this.b = 1;
         }

         public void method2() {
             int local2 = b;
             this.a = 2;
         }
    }
```

As we can see `method1()` and `method2()` are 2 methods that are executed concurrently by 2 threads.

```
Thread 1        Thread 2
1: local1 = a;  3: local2 = b;
2: b = 1;       4: a = 2;
```

Think about a possible scenario, in which after the execution of both `method1()` and `method2()` the values of the local variables are:

```
local1 = 2
local2 = 1
```

Consider everything we've learned so far.

- **It's possible since each of the methods executes 2 instructions that do not depend on each other. Therefore the compiler is free to re-arrange the order of the instructions.**

  If method1() is re-arrange to

```
public void method1() {
    this.b = 1;
    int local1 = a;
}
```

  Then the following order of execution is possible:

```
Thread 1: b = 1
Thread 2: local2 = b (1)
Thread 2: a = 2
Thread 1: local1 = a (2)
```

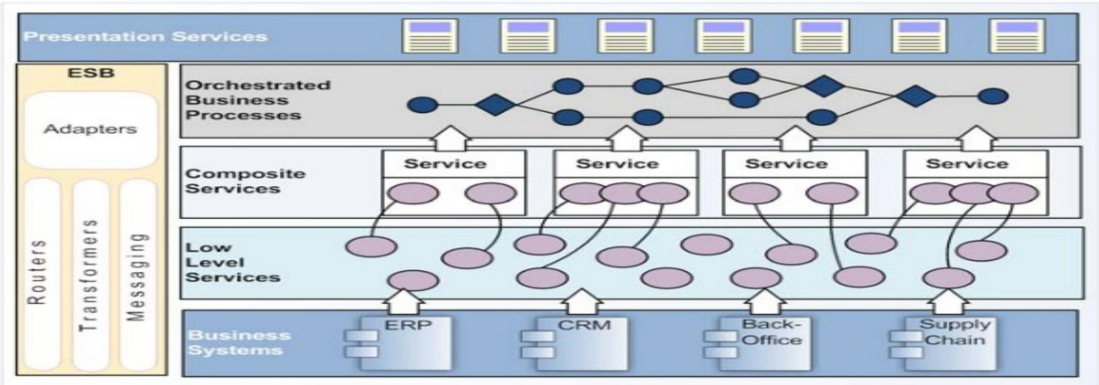
Locking Strategies

> Granularity is the extent to which a system is broken down into small parts, either the system itself or its description or observation. It is the extent to which a larger entity is subdivided. For example, a yard broken into inches has finer granularity than a yard broken into feet.
>
> Coarse-grained systems consist of fewer, larger components than fine-grained systems; a coarse-grained description of a system regards large subcomponents while a fine-grained description regards smaller components of which the larger ones are composed.

In simple terms

- **Coarse-grained - larger components than fine-grained, large subcomponents. Simply wraps one or more fine-grained services together into a more coarse-grained operation.**
- **Fine-grained - smaller components of which the larger ones are composed, lowerlevel service**

It is better to have more coarse-grained service operations, which are composed by fine-grained operations



Source: Open Source SOA

If you have one mutex protecting an entire program, then it is coarse-grained locking. If you have many mutex, say, one per integer in your program that you might want to read or write, then you have fine-grained locking.

Conditions for Deadlock

- **Mutual exclusion: Only one thread can have exclusive access to a resource**
- **Hold and Wait: At least one thread is holding a resource and is waiting for another resource**
- **Non-preemptive allocation - A resource is released only after the thread is done using it.**
- **Circular wait - A chain of at least two threads each one holding one resource and waiting for another resource**

Solution -> Avoid Circular wait - Enforce a strict order in lock acquisition
Deadlock detection - Watchdog
Read the article below
Code that debugs itself: Fixing a deadlock with a watchdog | by Evan Jones | Bluecore Engineering | Medium

ReentrantLock

- **Works just like the synchronized keyword applied on an object**
- **Requires explicit locking and unlocking**
- **disadvantage: after we done with shared resources we have to unlock it!!**
    - **If any exception throws inside the method before the unlock, we'll never have a chance to unlock it!**
- **May reduce the throughput of the application**
- **Using tryLock()**
    - **We avoid blocking the real-time thread**
    - **Kept app responsive**
    - **Performed operations atomically**

ReadLock and WriteLock from ReentrantReadWrite Lock

SEMAPHORE

- Can be used to restrict the number of “users” to a particular resource or a group of resources
- Unlike the locks that allow only one “user” per resource
- The semaphore can restrict any given number of users to a resource
- Semaphore doesn't have a notion of owner thread
- Many threads can acquire a permit
- The same thread can acquire the semaphore multiple times
- The binary semaphore (initialized with 1) is not reentrant
- Semaphore can be released by any thread
- Even can be released by a thread that hasn’t actually acquired it.

Difference with Locks

Inter-thread-Condition.await() -> suspend the thread. Wait for a state change

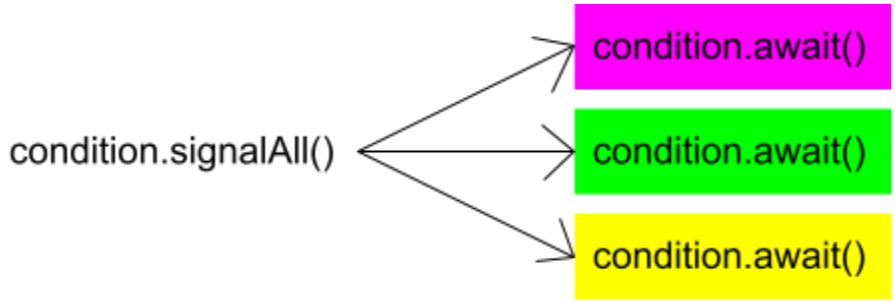
- void await - unlock lock, wait until signalled
- Long awaitNanos(long nanosTimeout) - wait no longer than nanosTimeout
- boolean await(long time, TimeUnit unit) - wait no longer than time, in given time units
- boolean awaitUntil(Date deadline) - wake up before the deadline date

Inter-thread-Condition.signal()

- void signal() - wakes up a single thread, waiting on the condition variable
- A thread that wakes up has to reacquire the lock associated with the condition variable
- If currently, no thread is waiting on the condition variable, the signal method doesn’t do anything

Inter-thread-Condition.signalAll()

- void signalAll() - Broadcast a signal to all threads currently waiting on the condition variable.
- Doesn’t need to know how many (if at all) threads are waiting on the condition variable.



wait(), notify(), and notifyAll()

- The Object Class contains the following methods:
  - public final void wait() throws InterruptedException
  - public final void notify()
  - public final void notifyAll()
- Every Java Class inherits from the Object Class
- We can use any object as a condition variable and a lock (using the synchronized keyword)
- wait() - Causes the current thread to wait until another thread wakes it up.
  - In the wait state, the thread is not consuming any CPU
- Two ways to wake up the waiting thread:
  - notify() - Wakes up a SINGLE thread waiting on that object.
  - notifyAll() = Wakes up ALL the threads waiting on that object
- To call() wait(), notify(), or notifyAll() we need to acquire the monitor of that object (use synchronized on that object)

PRIORITY INVERSION

- Two threads sharing a lock
  - Low priority thread (document saver)
  - High priority thread (UI)
- Low priority thread acquires the lock, and is preempted (scheduled out)
- High priority thread cannot progress because of the low priority thread is not scheduled to release the lock.

Thread Not Releasing a lock (Kill Tolerance)

- Thread dies gets interrupted or forgets to release a lock
- Leaves all thread hanging forever
- Unrecoverable, just like a deadlock
- To avoid developer need to write more complex code

AtomicX Classes -> Java 10

AtomicReference<T> - wraps a reference to an object, allows us to perform atomic operations on that reference, including the compareAndSet(..)