Threads

A single worker is called a thread. A thread is a sequence of instructions that can be executed independently of other parts of a program. It gives the illusion of doing multiple things at once, as it switches between the tasks really fast. This is called concurrent execution.

Threads share the same memory space within a process, allowing them to access and modify the same data directly. This shared memory simplifies communication and data sharing between threads.

In java, every program has at least one thread, the main thread. The main thread is created automatically when the program starts and is responsible for executing the main() method. You can also create additional threads to perform other tasks concurrently. Each thread is spawned from an existing thread.

When would you use threads?

An example of when you would use multithreading is when you have a program that performs time-consuming tasks or operations, such as downloading files from the internet, processing large amounts of data, or performing complex calculations.

By using multithreading, you can divide these tasks into smaller subtasks that can be executed concurrently, taking advantage of multiple CPU cores and potentially improving the overall performance and responsiveness of your application.

This can be used if you application is a cloud storage application, you can use multithreading to download multiple files at the same time.

Threads states

- New: The thread has been created but has not yet started.
- Runnable: The thread is eligible to run, but it may or may not be currently executing.
- Blocked: The thread is waiting for a resource or lock to become available.
- Waiting: The thread is waiting indefinitely for another thread to perform a particular action.
- Timed Waiting: The thread is waiting for a specified period of time.
- Terminated: The thread has finished executing and has terminated.

Thread synchronization

In Java, you can use synchronization mechanisms, such as locks, to coordinate the execution of multiple threads. Synchronization helps avoid race conditions and ensures thread safety when accessing shared resources.

A race condition occurs when two or more threads access a shared resource and try to modify it at the same time. This can lead to unpredictable results and data corruption.

Thread priorities

Java threads can have different priorities, ranging from 1 (lowest) to 10 (highest). The thread scheduler uses these priorities to allocate CPU time to threads. However, the exact behavior of thread priorities may vary across different JVM implementations.

Threads Object

You can make a new thread by making a new object of the thread class. The thread class takes a runnable object, and a name for the thread. The runnable object is the code that you want to run. You can pass the code as a lambda expression. The code will not be executed right now but you are passing it as a object that you can run. (Saves the object for later). To start the thread you have to call the start method. The start method will call the run method. The run method is the code that you want to run.

e.g.,

```
public static void main(String[] args){
    //Eg of a single threaded program
    // To get the current thread name you can call the method:
    System.out.println(Thread.currentThread().getName()); // Prints "main"
    System.out.println("Hello World");
}
```

```
public static void main(String[] args){
    //Eg of a mulitthreaded program
    System.out.println("Hello World");

    // Thread constructor takes a runnable object, and a name for the
thread
    Thread t = new Thread(() -> { //This code will not be executed right
now but you are passing it as a object that you can run. (Saves the object
for later)
        System.out.println("Hello Thread");
    } , "ThreadName"); // Create a new thread
    t.start(); // Start the thread

    System.out.println("YO");
}
```

The order of the output from the code above is not guaranteed. All we know is that Hello World would be printed first. But we can't predict if Hello Thread or YO would be printed after. It depends on which task is completed first. However, because`System.out.print` has synchronization, the output will never be mixed up, but it just locks it until the output is complete.

Threats as a class

To start a thread in a class you can either extend the thread class or implement the runnable interface.

```
public class MyThread extends Thread{
    public void run(){
        //code to run
    }
}
```

```
public class MyThread implements Runnable{
    public void run(){
        //code to run
    }
}
```

```
public static void main(String[] args){
    Thread t = new Thread(new MyThread()); // You dont have to do this if
you extend the thread class
    // Thread t = new MyThread(); // This is if you extend the thread
class
    t.start();
}
```

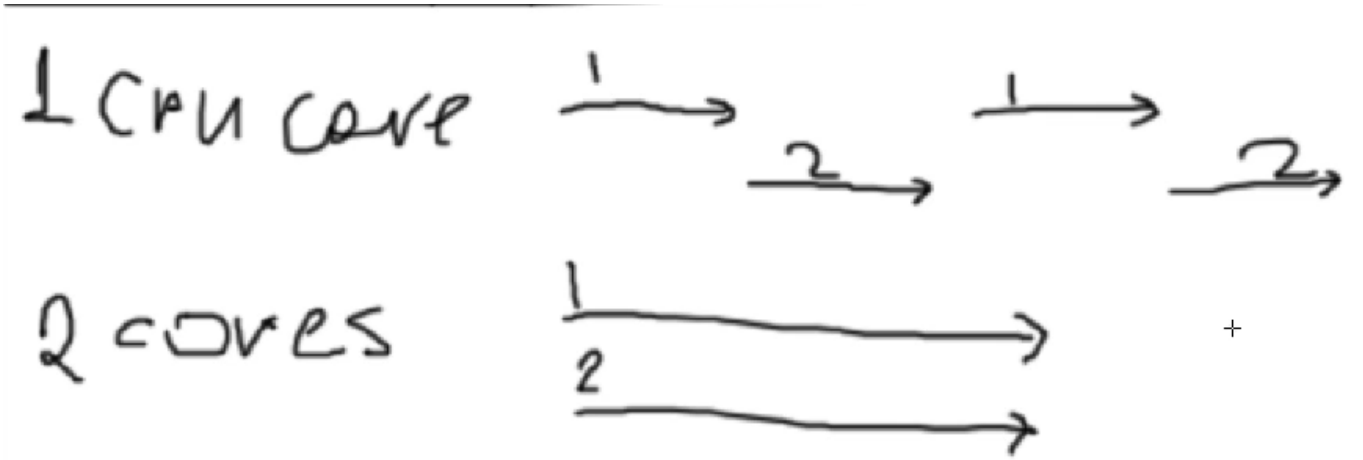
It is better to implement the runnable interface because you can only extend one class but you can implement multiple interfaces.

If one thread runs into an exception, the other threads will still run. The thread that ran into the exception will die.

Concurrent vs Parallel

say you have 1 cpu core, you can't execute the code in parallel, because you have one processor. So you have to execute the code one after the other. This is called sequential execution. This could be done really fast, to give the illusion that the code is running in parallel, by flipping between the tasks really fast.This is called concurrent execution. This is what multithreading does.

If you have 2 cores, you can execute the code in parallel. This is called parallel execution.



Sending data between threads

We want the main thread to pass data to another thread, this thread does something and passes it back to the main thread.

To make sure we get the data when the task has finished executing we can use blocking queues. A blocking queue represents a queue first in first out data structure.

An `ArrayBlockingQueue` is a specific implementation of the `BlockingQueue` interface in Java. It is designed to provide a blocking behavior where multiple threads are involved in producing and consuming elements from a shared queue.

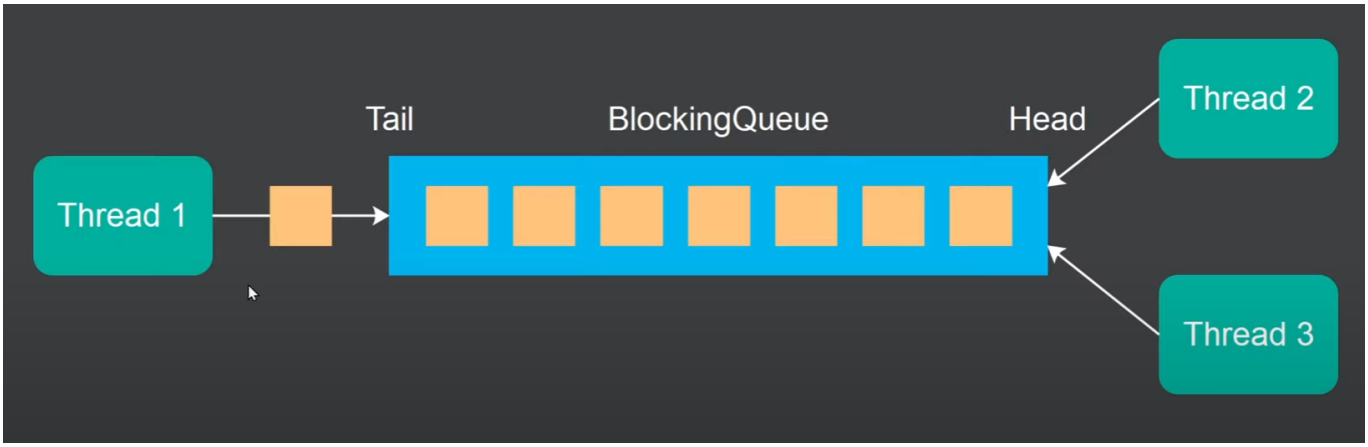
Here's how an `ArrayBlockingQueue` works:

1. Capacity: When creating an `ArrayBlockingQueue`, you specify its capacity, which represents the maximum number of elements that can be stored in the queue. Once the capacity is reached, any further attempts to add elements will block until space becomes available.
2. Array-based Structure: Internally, the `ArrayBlockingQueue` uses an array to store the elements. The array size is fixed and does not change after initialization.
3. Blocking Behavior: The `ArrayBlockingQueue` provides blocking operations for adding and removing elements. If a thread tries to add an element to a full queue, it will be blocked until space becomes available. Similarly, if a thread tries to remove an element from an empty queue, it will be blocked until an element is available.
4. FIFO Ordering: Elements are stored and retrieved in a first-in-first-out (FIFO) order. The element that has been in the queue for the longest time will be the first one to be removed.
5. Thread Safety: The `ArrayBlockingQueue` is thread-safe, meaning it can be safely accessed by multiple threads concurrently without external synchronization. It internally uses locks and conditions to ensure proper synchronization and blocking behavior.
6. Insertion and Removal Operations: The `ArrayBlockingQueue` provides several methods for adding and removing elements. Some commonly used methods include:
    - `put(element)`: Inserts an element into the queue, blocking if the queue is full.
    - `take()`: Removes and returns the first element from the queue, blocking if the queue is empty.

- `offer(element)`: Inserts an element into the queue if space is available, otherwise returns `false` immediately.
- `poll()`: Removes and returns the first element from the queue if available, otherwise returns `null` immediately.
- `offer(element, timeout, unit)`: Inserts an element into the queue if space is available within the specified timeout duration, blocking until either space becomes available or the timeout expires.
- `poll(timeout, unit)`: Removes and returns the first element from the queue if available within the specified timeout duration, blocking until either an element becomes available or the timeout expires.

By utilizing the blocking behavior and thread-safe operations of `ArrayBlockingQueue`, you can implement efficient coordination and communication between producer and consumer threads in concurrent applications.

The difference between a blocking queue and a normal queue is that a blocking queue will allow the thread that are inserting and removing elements to block. For example if the queue is full and the thread tries to add an element to the queue, the thread will block until there is space in the queue.



eg Find the length of a string

```
public static void main(String [] args){
    BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(1); //the ()
is the size of the queue

    int result = 0;
    String str = "Hello World";


    Thread t = new Thread(() -> {
        int length = str.length();
        queue.add(length);
    }, "calculate length");
    t.start();

    System.out.println("The length of the string is: " + queue.take());
}
```

What happens if the UI thread is blocked?

If the UI thread is blocked, the UI will freeze. If the UI thread is blocked, the OS will not update the application, in some OS such as IOS and Android the OS will kill the application.

The way to fix this is to use a background thread to do the heavy tasks such as downloading a file. Then when the file is downloaded, the UI thread will be notified and it will update the UI.