PProg: Important things to know

Creating a Thread

```
public class Useless extends Thread {
    int i;
    Useless (int i) {
        this.i = i;
    }
    public void run() {
        System.out.println("Thread says hi" + i);
        System.out.println("Thread says bye" + i);
    }
}


public class M {
    public static void main(String[] args) {
        for (int i = 0; i < 20; i++) {
            Thread t = new Useless(i+ 1);
            t.start(); //Important: you cannot use t.run() --> t.run doesn't
create a new Thread
        }
    }
}
```

Important: you can only create a Thread, when you use start(). Thread, run method has no return value and no argument

Joining threads:

```
public class Useless extends Thread {
    int i;
    Useless (int i) {
        this.i = i;
    }
    public void run() {
        System.out.println("Thread says hi" + i);
        System.out.println("Thread says bye" + i);
    }
}


public class M {
    public static void main(String[] args) {
        Thread[] threads = new Thread[20];
```

```
        for (int i = 0; i < 20; i++) {
            Thread t = new Useless(i+ 1);
            t.start();
            threads[i] = t;
        }
        for (int i = 0; i < 20; i++) {
            try { //need catchblock around join
                threads[i].join();
            } catch (InterruptedException e) {
               //Some catch block
            }
        }
        System.out.println("All done.");
    }
}
```

With "join", you wait until every thread is done

Thread states

If we want to be able to talk about the effects of different thread operations, we need some notion of thread states. In short, a Java thread typically goes through the following states:

- **Non-Existing:** Before the thread is created, this is where it is. We don't know too much about this place, as it's not actually on our plane of reality, but it's somewhere out there.
- **New:** Once the Thread object is created, the thread enters the new state.
- **Runnable:** Once we call start() on the new thread object, it becomes eligible for execution and the system can start scheduling the thread as it wishes.
- **Blocked:** When the thread attempts to acquire a lock, it goes into a blocked state until it has obtained the lock, upon which it returns to a runnable state. In addition, calling the join() method will also transfer a thread into a blocked state.
- **Waiting:** The thread can call wait() to go into a waiting state. It'll return to a runnable state once another thread calls notify() or notifyAll() and the thread is removed from the waiting queue.
- **Terminated:** At any point during execution we can use interrupt() to signal the thread to stop its execution. It will then transfer to a terminated state. Note that when the thread is in a runnable state, it needs to check whether its interrupted flag is set itself, it won't transfer to the terminated state automatically. Of course, exiting the run method is equivalent to entering a terminated state. Once the garbage collector realizes that the thread has been terminated and is no longer reachable, it will garbage collect the thread and return it to a non-existing state, completing the cycle

Data Races:

A data race is a specific kind of race condition that is better described as a **simultaneous access error**, although nobody uses that term. There are two kinds of data races:

- When one thread might read an object field at the same moment that another thread writes the same field.
- When one thread might write an object field at the same moment that another thread also writes the same field.

```
class C {
    private int x = 0;
    private int y = 0;

    void f() {
        x = 1; //line A
        y = 1; //line B
    }
    void g() {
        int a = y; //line C
        int b = x; //line D
        assert (b >= a);
    }
}
```

Code has data races, but it doesn't occur --> proof by contradiction

Important to remember:

Speedup:

$S_p := \frac{T_1}{T_P}$

Where T_1 is the sequentail time (Time with one processor) and T_P with the time with P processors.

Reasons why program is nevertheless slower: _ *Additional overheads caused by inter-thread dependencies, creating threads, communicating between them and memory-hierarchy issues can greatly limit the speedup we gain from adding more processors.*

Amdahl:

- Fixed workload and upper bound on the speedup achievable when increasing the number of processors at our disposal.

Let f denote the non-parallelizable, serial fraction of the total work done in a program and P the number of processors at our disposal. Then, the following inequality holds:

$S_P <= \frac{1}{f + \frac{1 - f}{P}}$

If P is infinity then:

$S_\infty <= \frac{1}{f}$

Gustafson

- We increase the problem size as we improve the resources at our disposal. We consider the time interval to be fixed and look at the problem size.

Let f denote the non-parallelizable, serial fraction of the total work done in the program and P the number of processors at our disposal. Then, we get:

$S_P = f + P(1 -f) = P - f(P - 1)$

Divide and Conquer with ExecutorService

```
class MaxTask implements Callable  {
    //Runnable would work as well, but then you don't have a return value,
callable does have one
    int l;
    int h;
    int[] arr;
    ExecutorService ex;

    üublioc MaxTask(ExecutorService ex, int lo, int hi, int[] arr) {
       this...
    }
    public Integer call() throws Exception {
        //Check base case
        int size = h - l;
        if (size == 1) {
            return arr[l];
        } //split work
        int mid = size / 2;
        MaxTask m1 = new MaxTask(ex, l, l + mid, arr);
        MaxTask m2 = new MaxTask(ex, l + mid, h, arr);
        //Start subtasks
        Future<Integer> f1 = ex.submit(m1);
        Future<Integer> f2 = ex.submit(m2);
        //Combine results
        try {
            return Math.max(f1.get(), f2.get());
        } catch(Exception e) {
            return 0;
        }
    }
}

public static void main(String[] args)
int[] arr = new int[] {15, 7, 9, 8, 4, 22, 42, 13};
ExecutorService ex = Executors.newFixedThreadPool(8); //Attention, has to be
minimum number of thread which are needed --> otherwise you have an endless loop
MaxTask top = new MaxTask(ex, 0, arr.length, arr);
Future<Integer> max = ex.submit(top);
try {
    System.out.println(max.get());
} catch(Exception e){
    //somethinbg
}
ex.shutdown();
```

To avoid "knowing" how many threads you need, you can use Recursive Task:

```
class MaxForkJoin extends RecursiveTask<Integer>  {
    int l;
    int h;
    int[] arr;

    public MaxForkJoin(int lo, int hi, int[] arr) {
       this...
    }
    public Integer compute() {
        //Check base case
        int size = h - l;
        if (size == 1) {
            return arr[l];
        } //split work
        int mid = size / 2;
        MaxForkJoin m1 = new MaxForkJoin(l, l + mid, arr);
        MaxForkJoin m2 = new MaxForkJoin(l + mid, h, arr);
        //Run subtasks
        m1.fork();
        int max2 = m2.compute();
        int max1 = m1.join();
        //Combine results
        return Math.max(f1.get(), f2.get());
    }
}

public static void main(String[] args)
int[] arr = new int[] {15, 7, 9, 8, 4, 22, 42, 13};
MaxForkJoin top = new MaxTask(0, arr.length, arr);
ForkJoinPool jfp = new ForkJoinPool();
int res = fjp.invoke(tp);
System.out.println(res);
```

Note the following similarities: We use the library as follows:

- Instead of extending Thread, we extend RecursiveTask (with return value) or RecursiveAction (without return value)
- Instead of overriding run, we override compute
- Instead of calling start, we call fork
- Instead of a topmost call to run, we create a ForkJoinPool and call invoke Also, note that in the case of RecursiveTask<T>, join now returns a result.

Throughput

$Throughput \approx \frac{1}{max(computationtime(stages))}$

Latency

Time to perform a single computation, including wait time resulting from resource dependencies.

A pipeline is **balanced** if the latency remains constant over time.