Introduction to Concurrency and Threads

Process is a unit of execution, that has its own memory space.

The terms process and application are often used interchangeably, and I've done that myself, and will again in this section.

Process memory space = Heap

Each application has its own memory space, also known as **the heap**.

The heap isn't shared between two applications or two processes, they each have their own.

A Thread

A thread is a single unit of execution, within a process.

Each process can have multiple threads.

Every application has at least one thread, and that's **the main thread**.

Our code will run on the main thread.

We can also have our code run in other threads, which we can explicitly create and start.

Threads Share Process Memory

Creating a thread doesn't require as many resources as creating a process does.

Every thread created by a process, shares that process's memory space, the heap.

This can cause big problems with your applications.

Threads also have Stack Memory

Each thread's got what's called a thread stack.

This is memory, that only a single thread, will have access to.

Every Java application runs as a single process, and each process can then have multiple threads within it.

Every process has a heap, and every thread has a thread stack.

Why use multiple threads?

One of the most common reasons, is to offload long running tasks.

Instead of tying up the main thread, we can create additional threads, to execute tasks that might take a long time.

This frees up the main thread so that it can continue working, and executing, and being responsive to the user.

You also might use multiple threads to process large amounts of data, which can improve performance, of data intensive operations.

A web server, is another use case for many threads, allowing multiple connections and requests to be handled, simultaneously.

Concurrency

Concurrency, refers to an application doing more than one thing at a time.

Concurrency allows **different parts of a program to make progress independently**, often leading to better resource utilization, and improved performance.

One task doesn't have to complete, before another one can start, and **multiple threads can make incremental progress**.