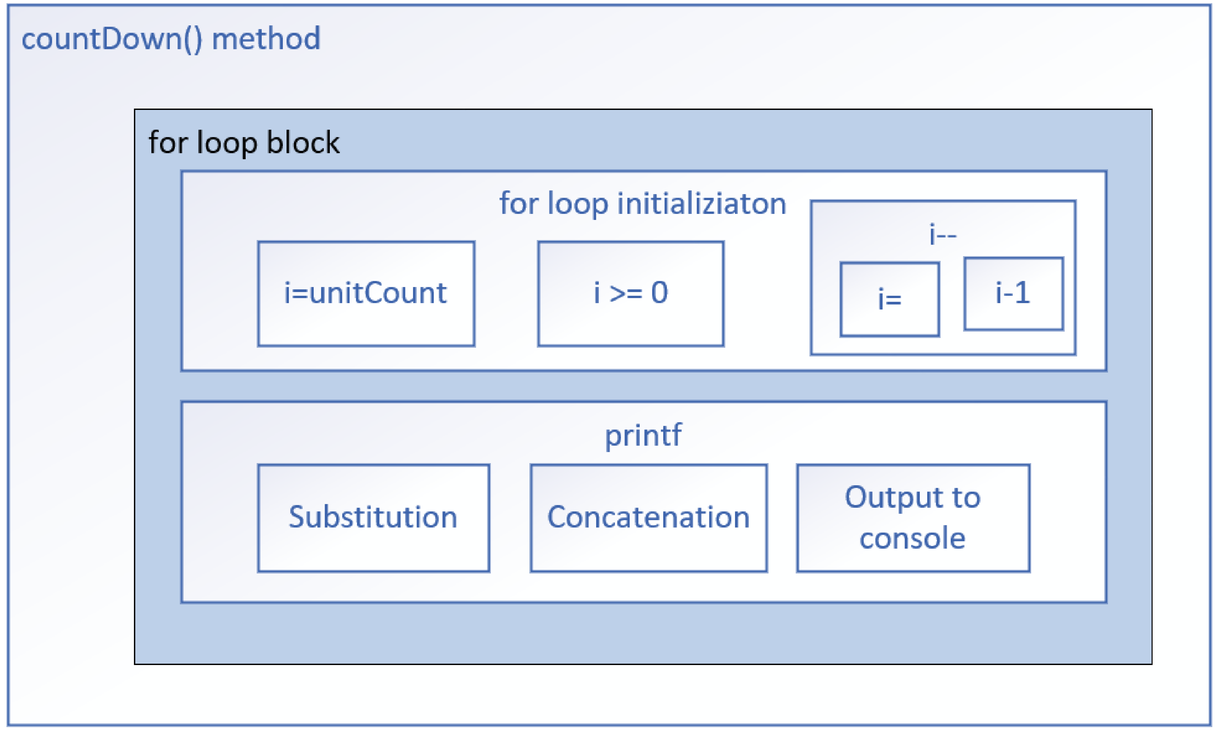
Interference

On this slide, I'm showing you a conceptual picture of the countDown method.

Each box shown in this diagram is a unit of work.

Only the smallest blocks might be atomic.

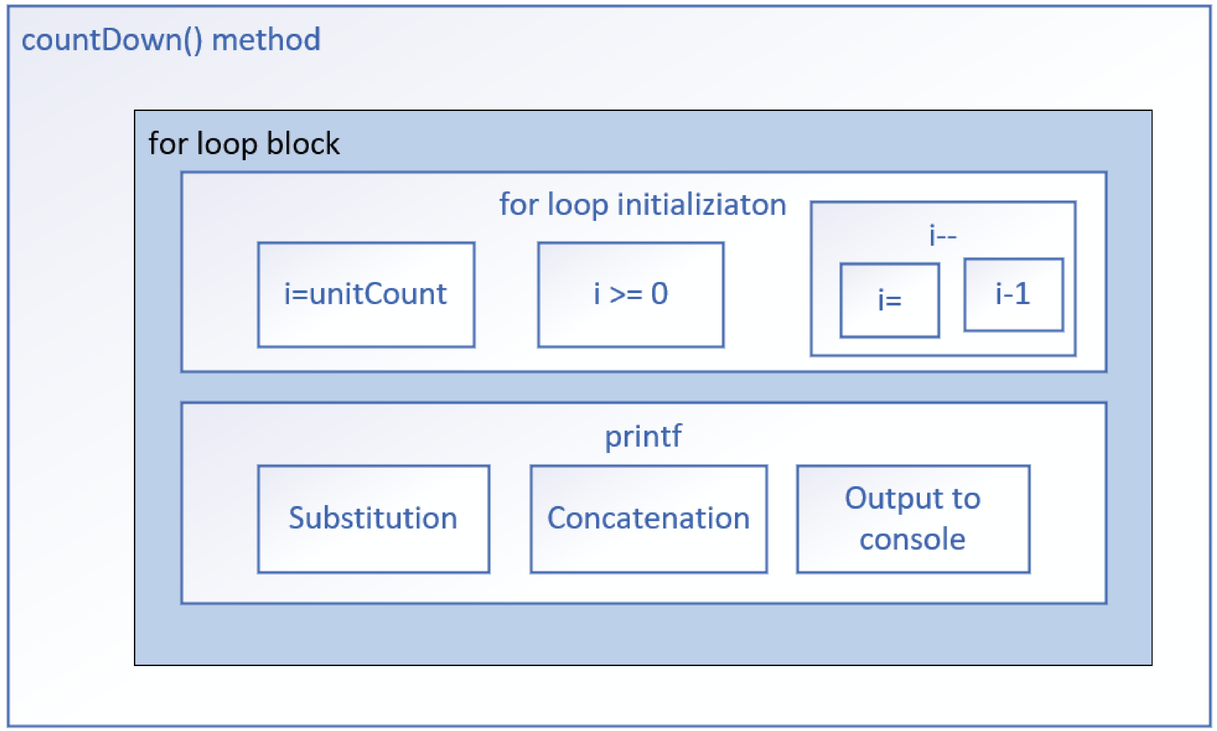
A thread can be halfway through the work in any one of these blocks, when it's time slice expires, and it then has to pause or suspend execution, to allow other threads to wake up and execute.



Interference

This means another active thread has an open door, to that same unit of work, where the paused thread is only partially done.

When threads start and pause, in the same blocks as other threads, this is called interleaving.



Interleaving

When multiple threads run concurrently, their instructions can overlap or interleave in time.

The execution of multiple threads happens in an arbitrary order.

The order in which the threads execute can't be guaranteed.

Atomic actions

In programming, an *atomic* action is one, that effectively happens **all at once**.

An atomic action either happens completely, or it doesn't happen at all.

Side effects of an atomic action **are never visible until the action completes**.

Even the simplest operations may not be atomic

Even decrements and increments, aren't atomic, nor are all primitive assignments.

For example, long and double assignments may not be atomic in all JVMs.

This slide shows three examples of operations that may not be atomic.

Increment Operand	Decrement Operand	Assignment of a long value
i++	--i	i = 100_000_000_000L

Even simple statements can translate to multiple non-atomic steps by the virtual machine.

Thread-Safe

An object or a block of code is thread safe, if it isn't compromised, by the execution of concurrent threads.

This means, the correctness and consistency of the program's output or its visible state, is unaffected by other threads.

Atomic operations and immutable objects are examples of thread-safe code.

Memory Consistency Errors

The operating system may read from heap variables, and **make a copy** of the value, in each thread's own storage cache.

Each thread has its own small and fast memory storage, that holds its own copy of a shared resource's value.

One thread can modify a shared variable, but this **change might not be immediately reflected or visible**.

Instead, it's first updated in the thread's local cache.

The operating system may not flush the first thread's changes to the heap, until the thread has finished executing.

volatile

The `volatile` keyword is used as a modifier for class variables.

It's an **indicator** that this variable's value may be changed by multiple threads.

This modifier **ensures** that the variable is always read from, and written to the main memory, rather than from any thread-specific caches.

This provides memory consistency for this variable's value across threads.

Volatile has limited usage though.

When to use volatile

There are specific scenarios when you'll want to use volatile.

- When a variable is used to track the state of a shared resource, such as a counter or a flag.
- When a variable is used to communicate between threads.

When NOT to use volatile

- When a variable is only used by a single thread.
- When a variable is used to store a large amount of data.