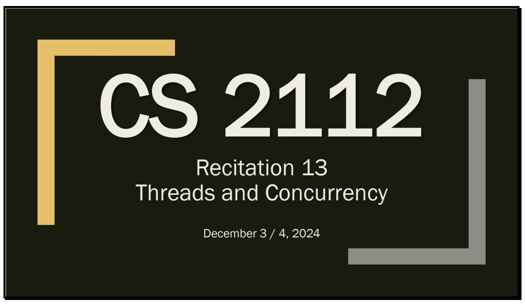
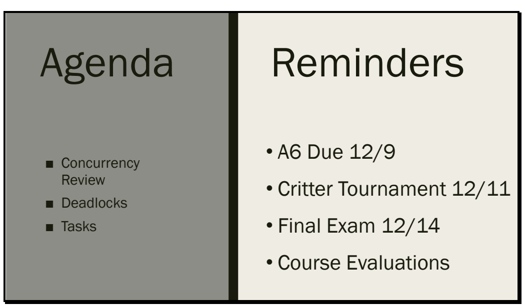
Slide 1

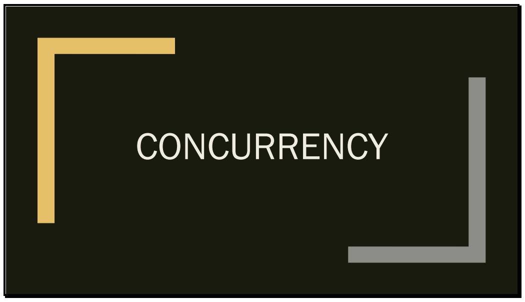
Slide 2



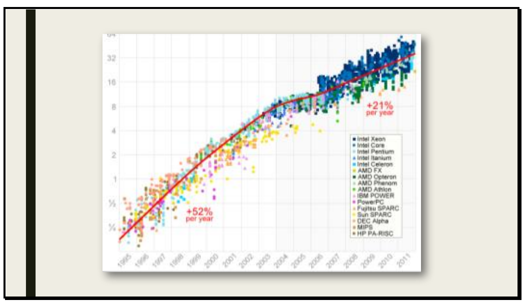
Slide 3



Slide 4

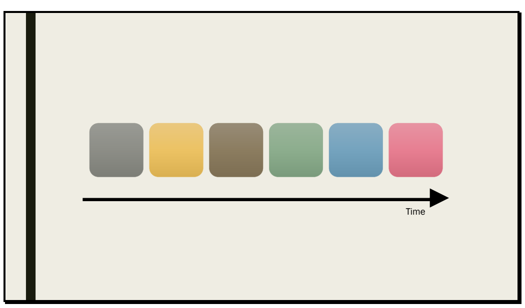


Slide 5



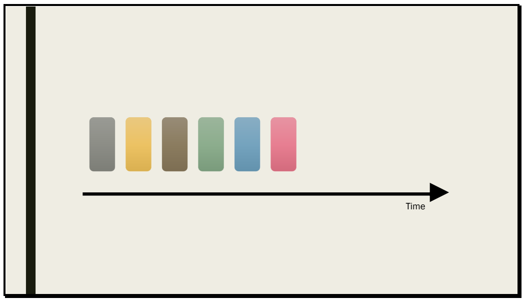
For the longest time, computers got twice as fast basically every two years. But unfortunately, around the 2000's, this stopped happening.

Slide 6



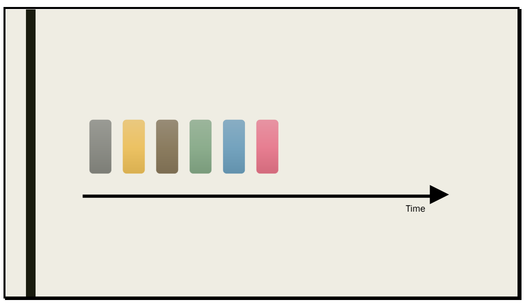
This is unfortunate. It used to be that if you had code that was slow…

Slide 7



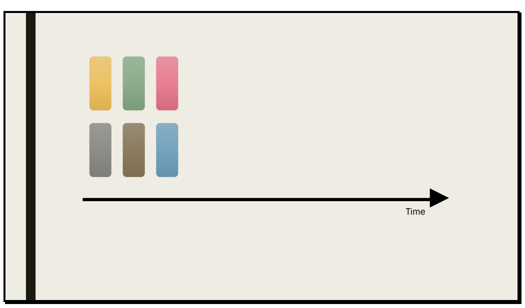
... Just wait two years and it would be twice as fast.

Slide 8



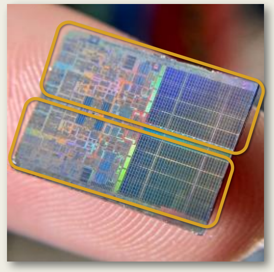
But as single core performance started plateauing, it became increasingly difficult to squeeze more performance out of our computers.

Slide 9



As such, another idea to improve the speed of computation is to run multiple parts of it at the same time.

Slide 10



Indeed, that's exactly what used to happen. CPUs in the 2000s, like the Core 2 Duo, just straight up had two whole cores on the chip.

Slide 11

```
task1();
task2();
task3();
task4();
task5();
task6();
```

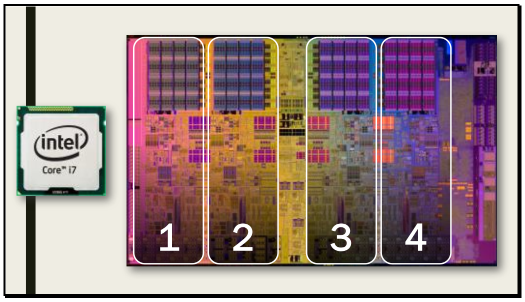
So if we imagine writing code that can take advantage of this fact, let's write some pseudocode.

Slide 12

```
run(core1) {
    task1();
    task2();
    task3();
}
run(core2) {
    task4();
    task5();
    task6();
}
```

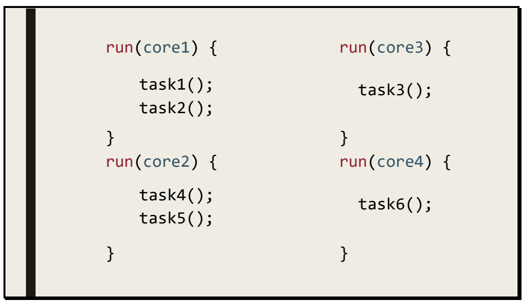
One proposal might be for the programmer to just write out what tasks go on which core.

Slide 13



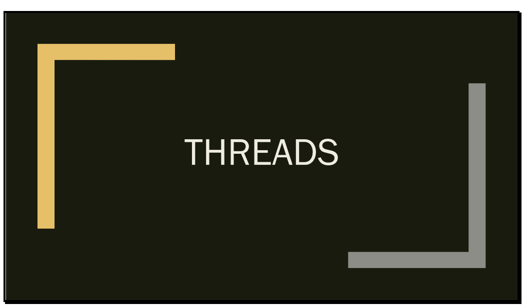
But then along comes the 2010's and new chips now have 4 or more cores!

Slide 14



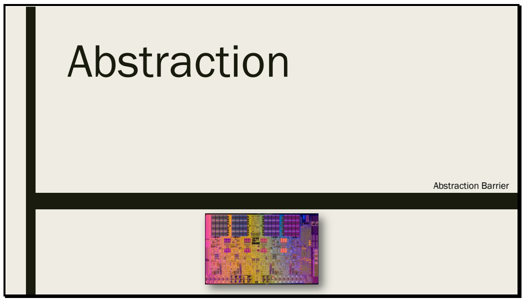
So now what? We could imagine having to write new code that now takes advantage of all four cores. But this isn't very sustainable as CPUs keep gaining more cores. What's worse, now our code no longer runs on an older CPU that doesn't have 4 cores.

Slide 15



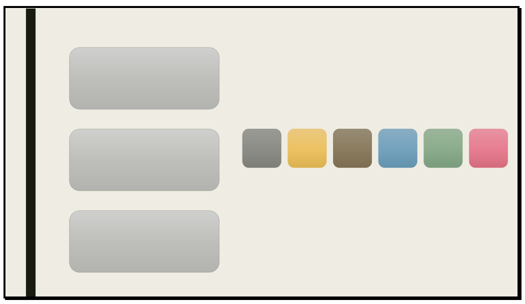
This brings us to the solution, which we call threads.

Slide 16



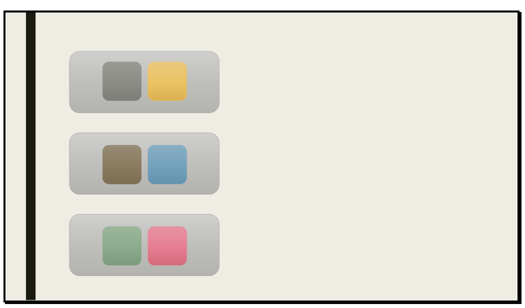
Instead of directly writing code for each CPU core, we abstract the cores away, behind the abstraction barrier.

Slide 17



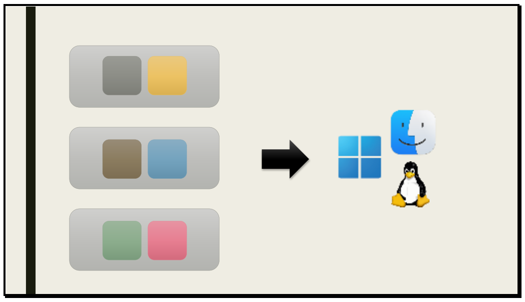
Instead, given some tasks, we can pretend we have as many cores as we naturally would need.

Slide 18



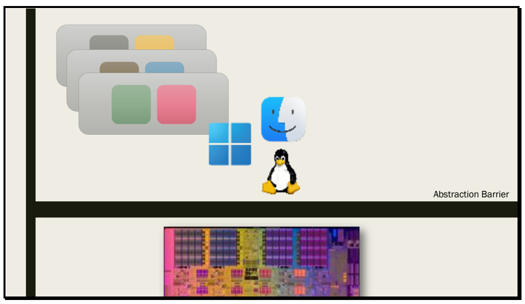
We can then assign our tasks to these imaginary cores based on what the natural split between tasks is, and not the actual hardware of the computer.

Slide 19



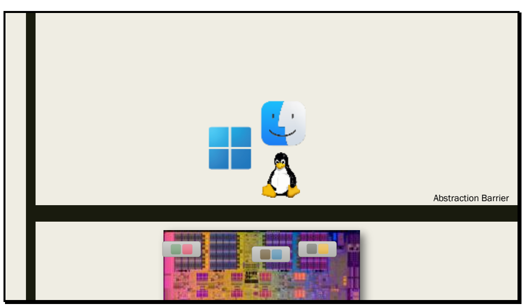
Then, we hand off these imaginary cores to the operating system.

Slide 20



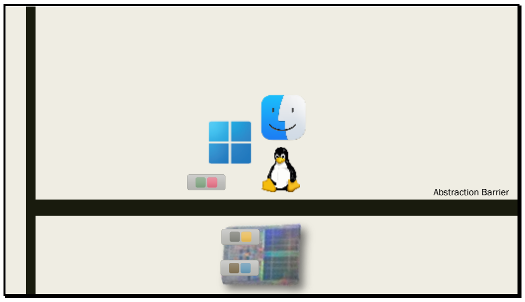
The OS can then look beneath the abstraction barrier and see how many actual cores exist.

Slide 21



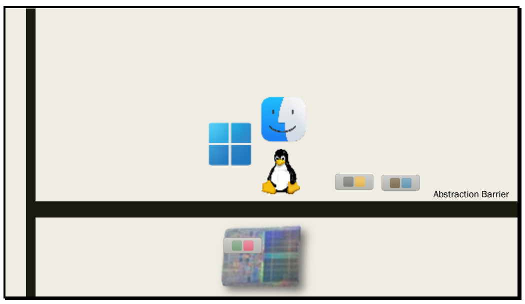
In this case, there's more cores then we need, so the CPU can just map each task to a core.

Slide 22



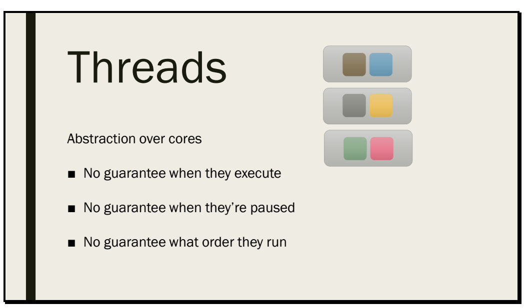
But if we have an older CPU with less cores, the OS may instead choose to first assign some of the tasks to the cores and have the other tasks wait…

Slide 23



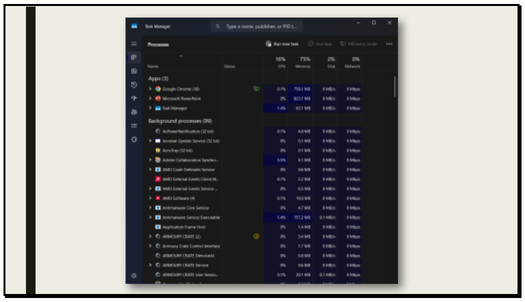
… and then run that other task later.

Slide 24



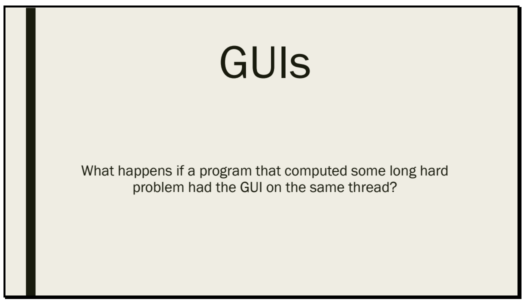
As you may have guessed, these imaginary cores are what we call threads. They are an abstraction we use to cover up the complexity of different hardware setups, and the semantics are such that we don't know when they run, when they're paused, or what order they're run in. It's annoying to give up this much control, but in return, we can allow the OS's scheduler to take care of figuring out which threads run on which core and when that should happen. What we lose in simplicity, we gain in flexibility.

Slide 25



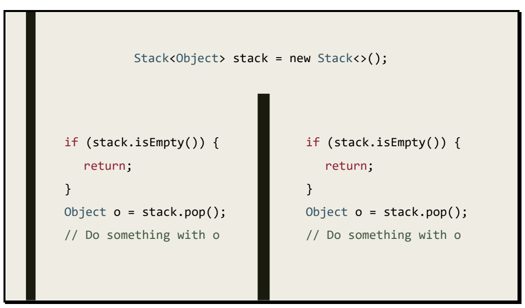
This is especially important when you remember that your program is not the only one running on a user's machine. With hundreds or thousands of programs and processes, each spawning as many threads as they need, no matter how many cores you have on your CPU, it's probably still not enough. What's more, the user may choose to do things like move their mouse or press a button on the keyboard, and these should be responded to immediately.

Slide 26



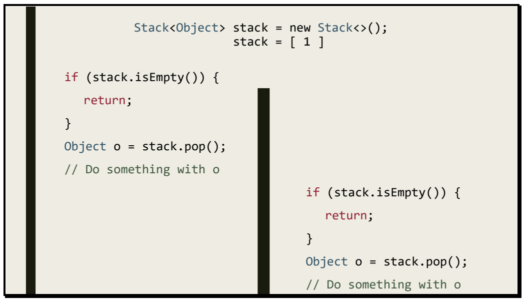
Rather than waste an entire core to wait for user input, the flexibility provided by threads (for example, the scheduler may pause a thread to make room for the code that responds to user input) allows a modern computer to maximize the usage of its CPU while performing as many tasks as fast as possible.

Slide 27



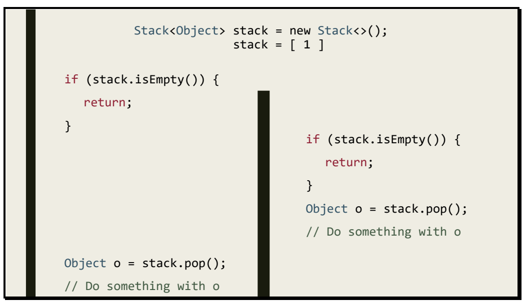
We've seen that the flexibility behind how threads run provides many benefits. But it means our job as programmer is harder. Imagine we have some stack like this, and two threads that want to pop an element if it's not empty.

Slide 28



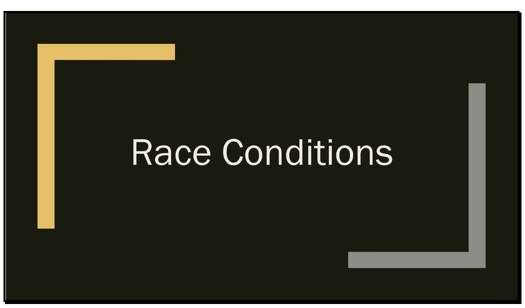
If there's only one element left in the stack and both threads run, you can imagine one situation where it's fine, because one thread runs first, and then the other sees it's empty.

Slide 29

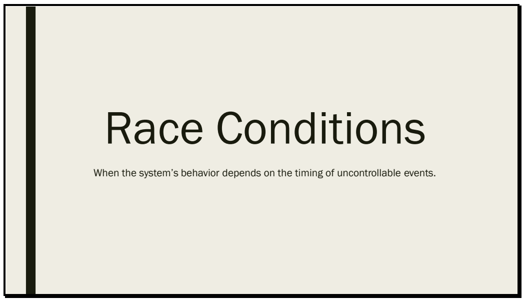


But if you get unlucky, you can imagine the second thread popping the last element after the first one already checked that the stack is non-empty. This is going to cause an exception, and is a classic **race condition**.

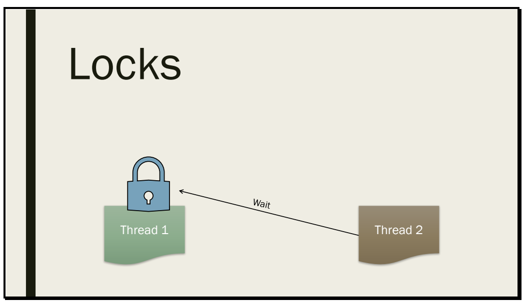
Slide 30



Slide 31

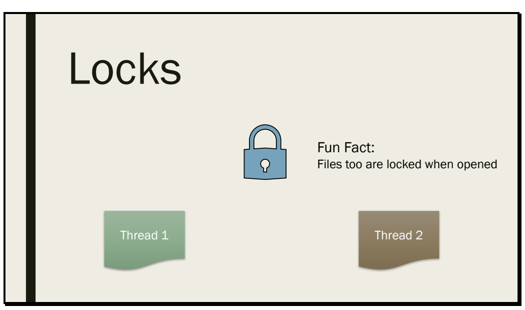


Slide 32

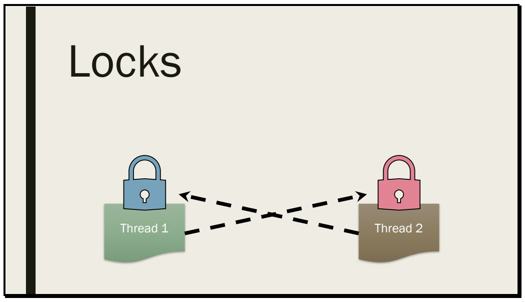


We can solve race conditions with locks, where one thread must wait for another thread to "release" the lock before it can "acquire" it and proceed.

Slide 33

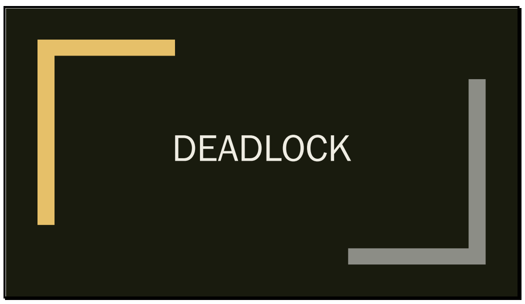


Slide 34

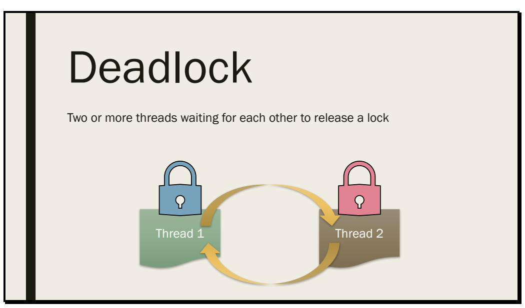


But now, if there's a scenario where each thread holds one lock and is waiting for the other, we end up with…

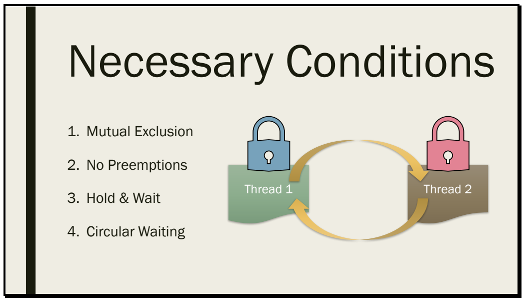
Slide 35



Slide 36



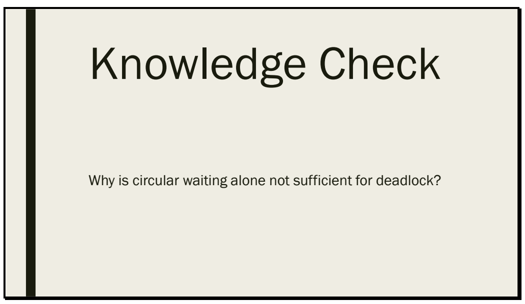
Slide 37



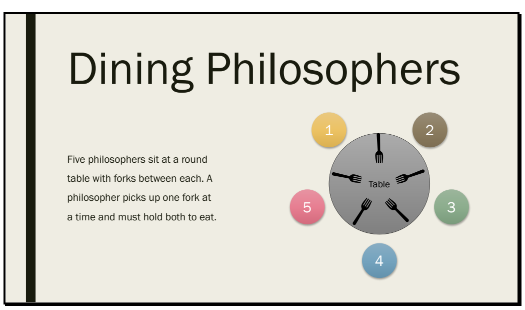
The 4 conditions that are necessary for a deadlock to occur:

- Mutual Exclusion: Each lock can be held by fewer threads than want it (in our case, a lock can only be held by 1 thread but 2 threads want it)
- No Preemptions: One thread cannot steal the lock held by another thread
- Hold & Wait: One thread does not release its lock before it acquires the other
- Circular Waiting: There is a cycle in the order of which thread acquire locks

Slide 38

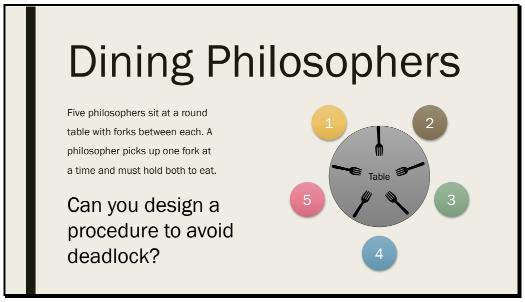


Slide 39

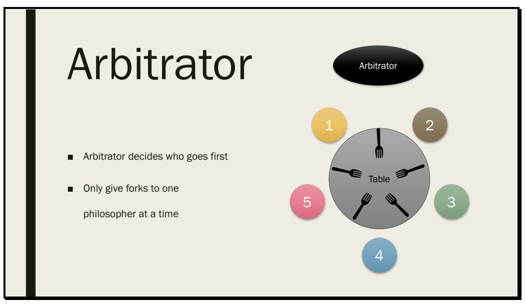


This is a classical philosophical problem. If each philosopher picks up one form, no one will ever eat.

Slide 40

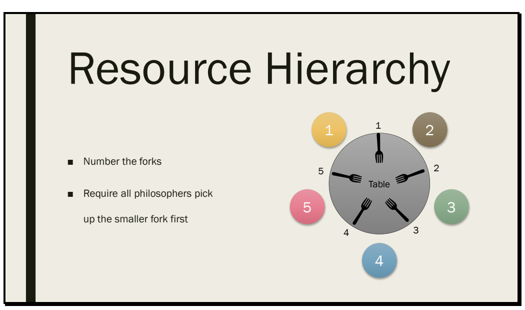


Slide 41



One option is to have a third party coordinate who gets which fork.

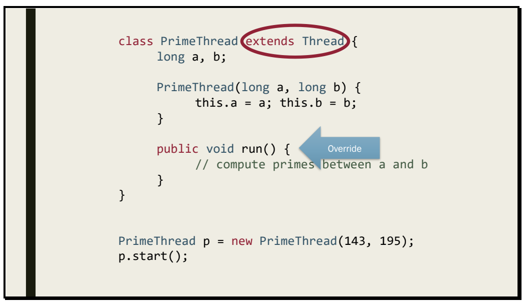
Slide 42



The other option is to enforce an ordering by numbering all the forks. If each philosopher is required to pick up the smaller fork before the larger one, then they will never have a circular wait.

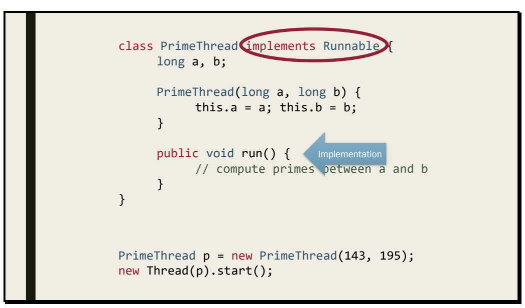
Slide 43

Slide 44



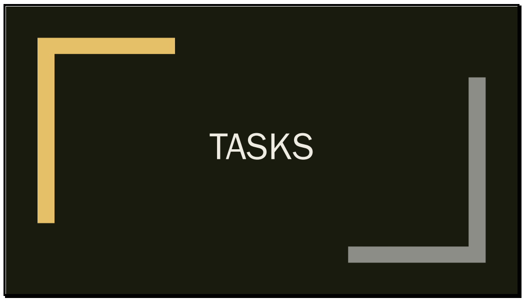
You could, in theory, create a thread by subclassing the Thread class and overriding the run method.

Slide 45



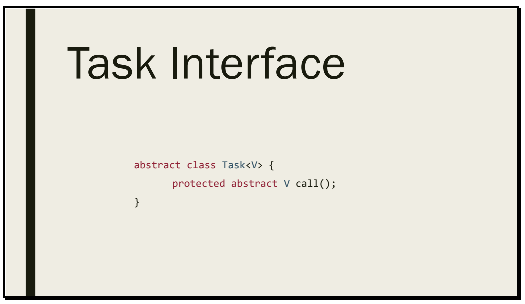
This is, however, less preferable to implementing the Runnable interface. Conceptually, your new object isn't using any of the parent state from the Thread class and doesn't really belong under it in the class hierarchy. It's more it's own independent class that provides some functionality. As such, it makes more sense for it to implement an interface.

Slide 46



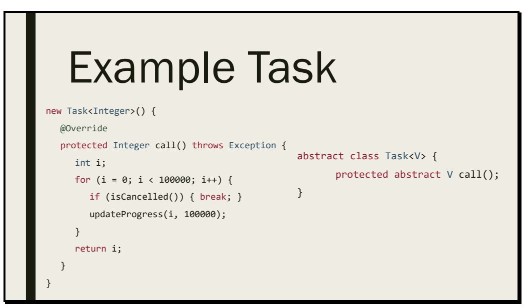
However, managing raw threads can be complicated, so JavaFX provides a further abstraction called a Task. These allow you to have some unit of computation run on a different thread that you can cancel or check the progress of.

Slide 47



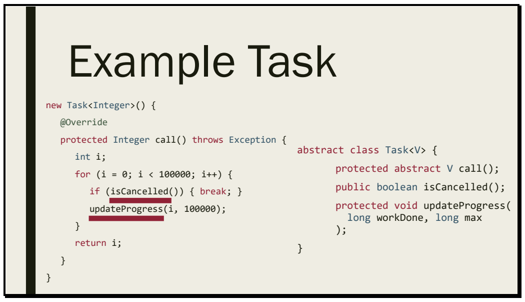
At its core is an abstract class called Task. Any class that extends this must implement the abstract method called call(). This is where you write the logic this task should compute (and note that it returns some generic type V you specify).

Slide 48



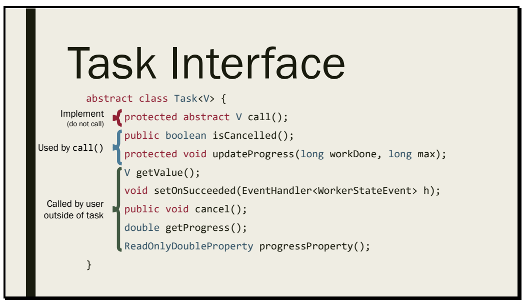
Here’s an example of a task that just counts to 100,000.

Slide 49



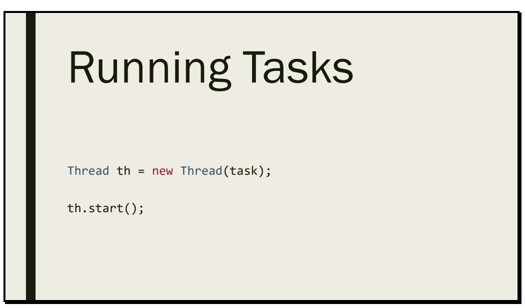
Notice that the task makes use of isCancelled() and updateProgress() to check if it has been cancelled and to report its progress. These are also part of the Task class. Note it’s your job to check periodically if the task has been cancelled; it’s not done for you.

Slide 50



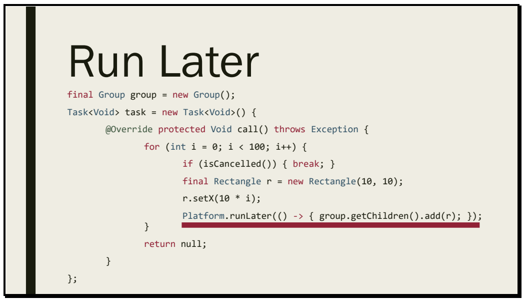
The methods we’ve seen so far are used by the implementer. The rest of the interface is for the users, and allows them to get the computed value, run a callback when the task is done, cancel the task, check its progress, or get an observable on the progress.

Slide 51



Tasks can be run by passing it to a thread.

Slide 52



Note that since the task is being run on a different thread, if you want to update your JavaFX GUI, you can't do that from a different thread. This is where Platform.runLater() becomes useful, as it schedules another unit of computation to be run back on the main JavaFX thread. This is particularly useful if you're using a push model where the background thread running your computation pushes data back to the GUI. (an alternative approach is a pull model where your JavaFX thread queries models for updated information).

Slide 53