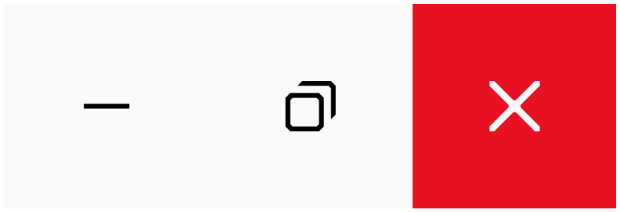


Responsiveness

Human Perception and Expectations

Performing CPU-intensive Tasks

Jul 19

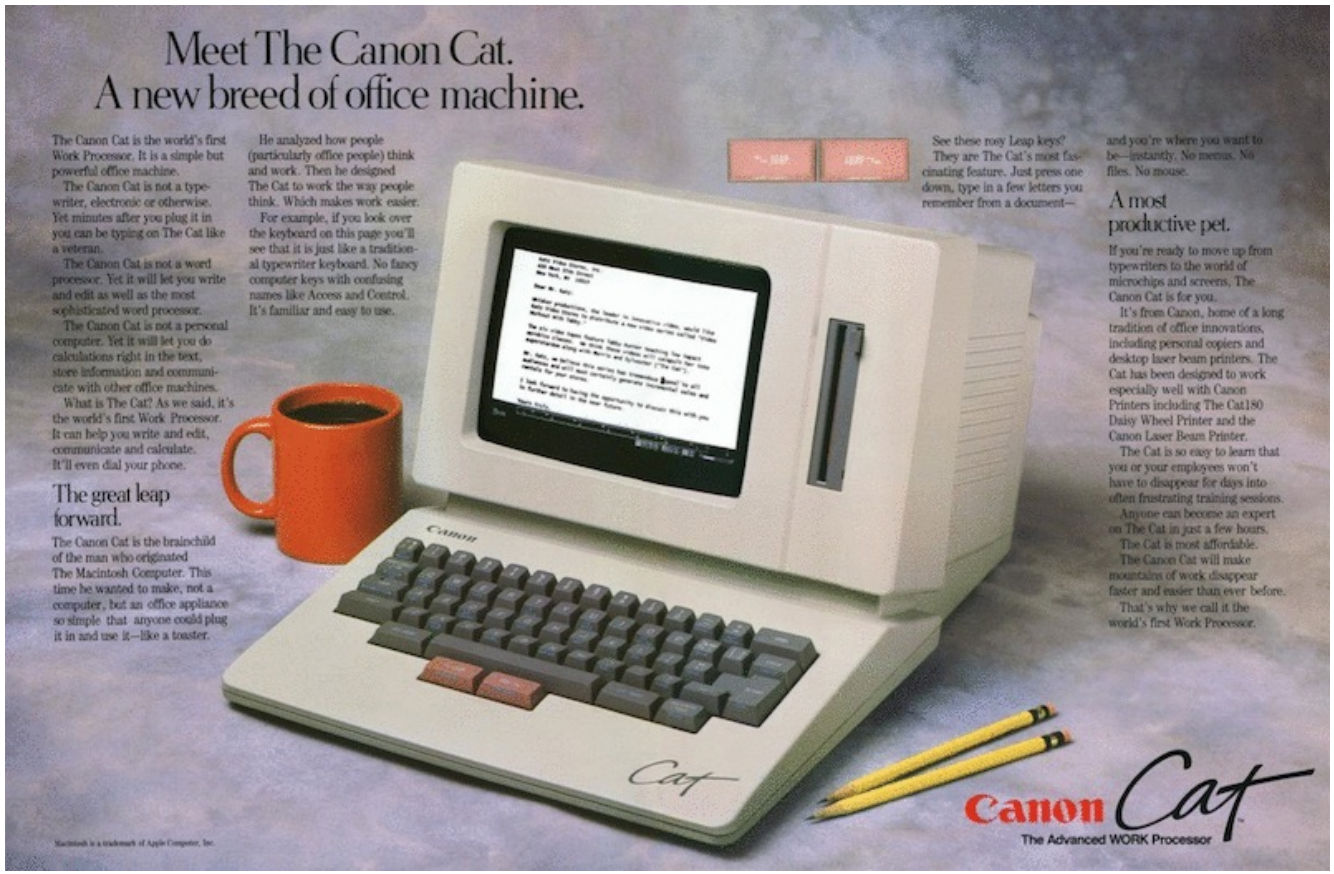
Human Perception and Expectations

Steve Souders' , "The Illusion of Speed" (Velocity 2013)

https://www.youtube.com/watch?v=bGYgFYG2Ccw

Design Implications

Time to prepare for conscious cognition task: Display a fake version of an application interface, or image of document on last save, while the real one loads in less than 10s



Responsive User Interfaces

A responsive UI delivers feedback to the user in a timely manner

- It does not make the user wait any longer than necessary
- It communicates state of long tasks to the user

We can make a UI responsive in two ways:

- Designing to meet human expectations and perceptions
- Loading data efficiently while maintaining interactivity

What factors affect responsiveness?

User Expectations

- how quickly a system “should” react to complete some task
- expectations for technology (e.g., web app vs. native desktop)

Application and Interface Design

- the interface keeps up with user actions
- the interface informs the user about application status
- the interface doesn’t make users wait unexpectedly

Responsiveness is the most important factor in determining user satisfaction, more so than ease of learning, or ease of use.

Responsiveness is not just system performance!

Responsive User Interfaces

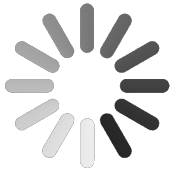
Slow Performance can still be responsive:

- Provide feedback to confirm user actions
- Provide feedback about progress
- Allow users to perform other tasks while waiting
- Perform low-priority system tasks in the background

Responsive User Interfaces

Provide feedback to confirm user actions, e.g.:

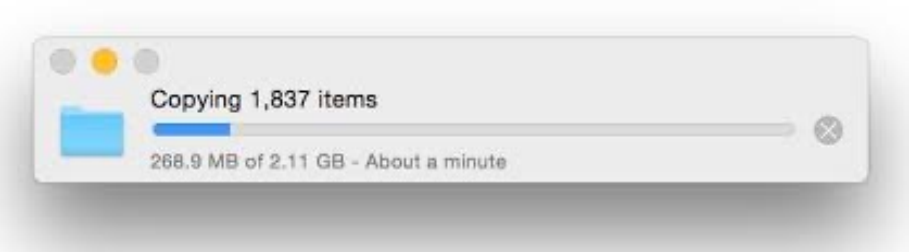
- Busy indicators

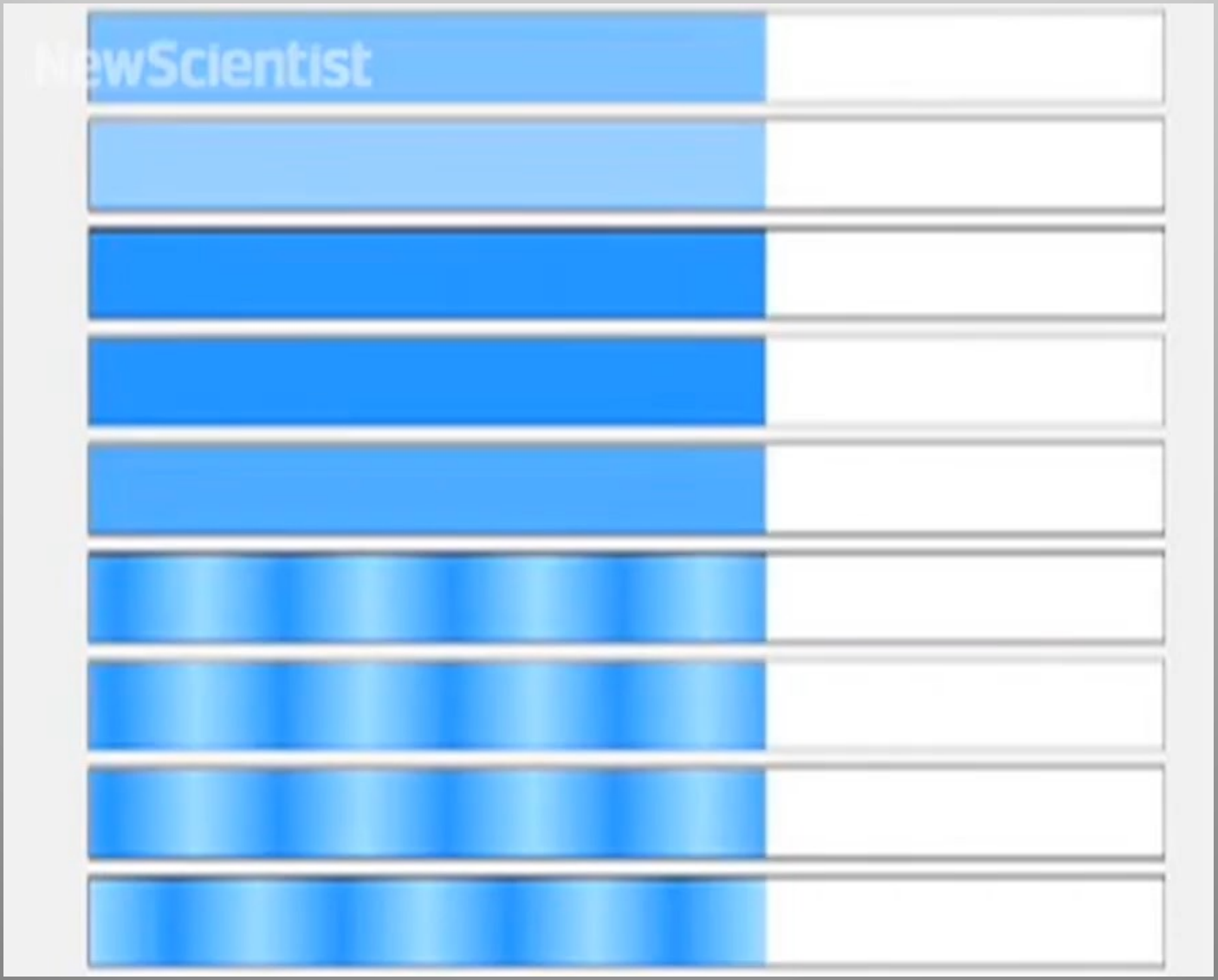


Responsive User Interfaces

Provide feedback about progress, e.g., progress bars:

- Show work remaining, not work completed
- Show total progress when multiple steps, not only step progress
- Display finished state (e.g., 100%) very briefly at the end
- Show smooth progress, not erratic bursts
- Use human precision, not computer precision: “74.5 seconds remaining” (bad), “About a minute” (good)



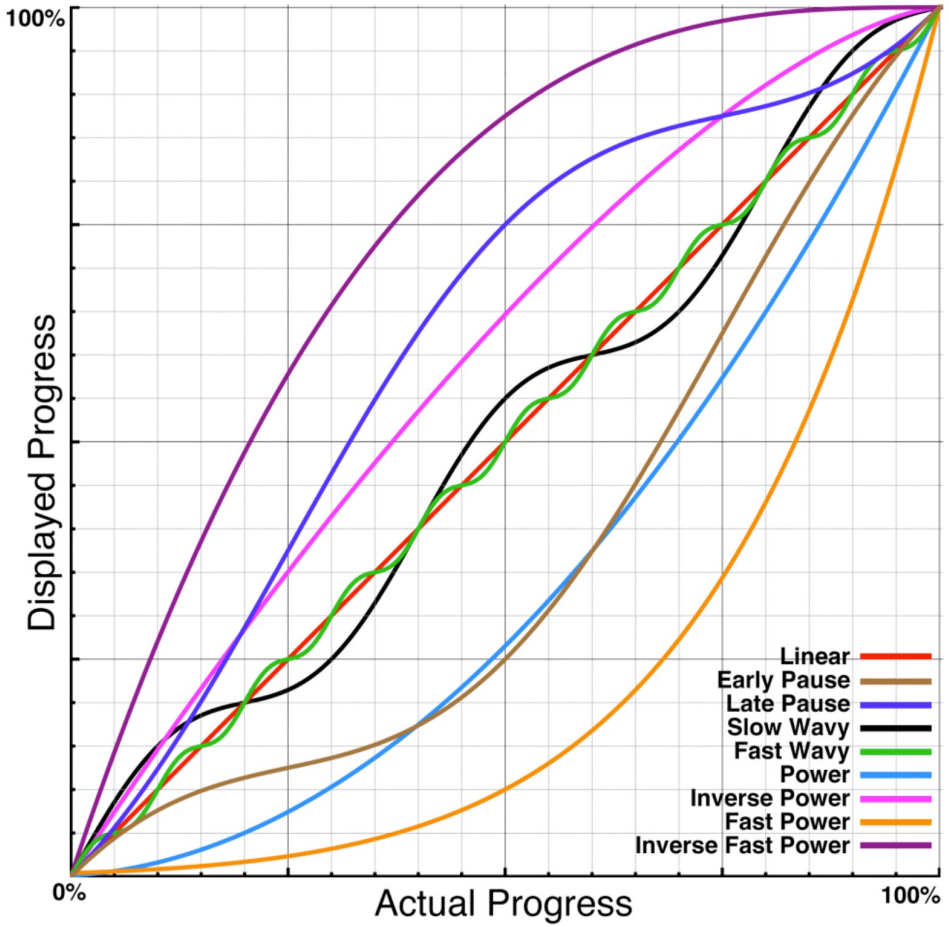


Harrison et al. Faster Progress Bars (2010)

https://www.newscientist.com/article/dn18754-visual-tricks-can-make-downloads-seem-quicker/

Changing Perception of Progress Bars

Change how actual progress maps to displayed progress



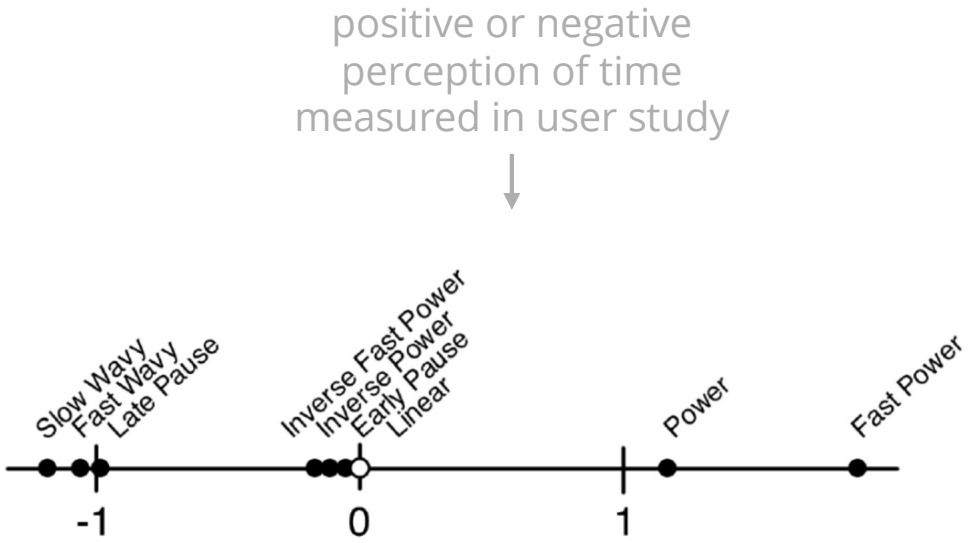


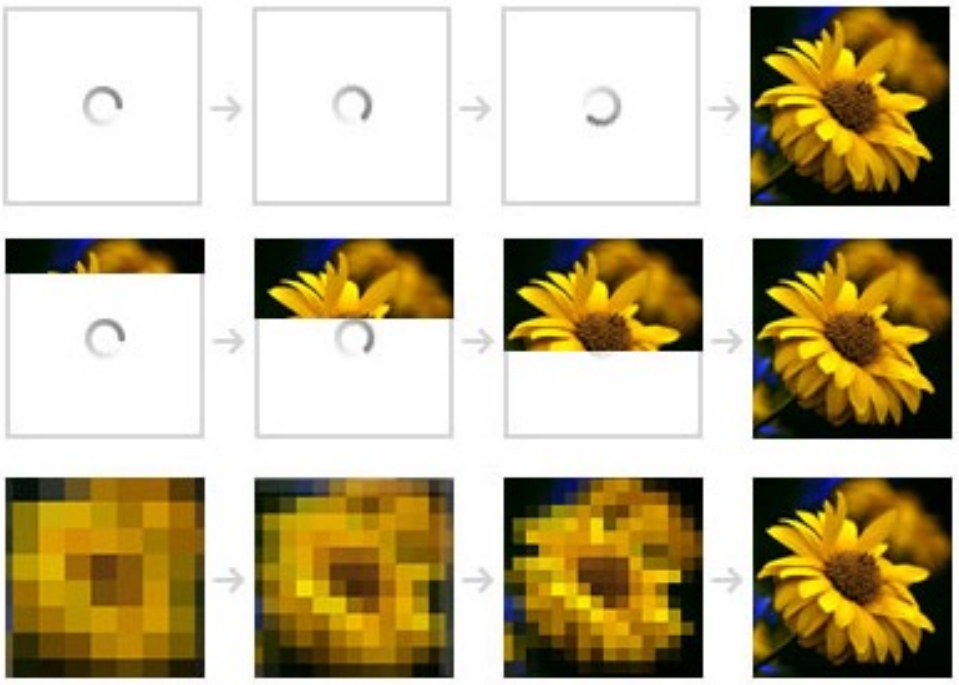
Figure 4: Number line showing relative distances from linear, which is centered at 0. Values generated from logistic regression model.

Harrison et al. Rethinking the progress bar. (2007)
http://www.chrisharrison.net/index.php/Research/ProgressBars

Responsive User Interfaces

Allow users to perform other tasks while waiting, e.g.:

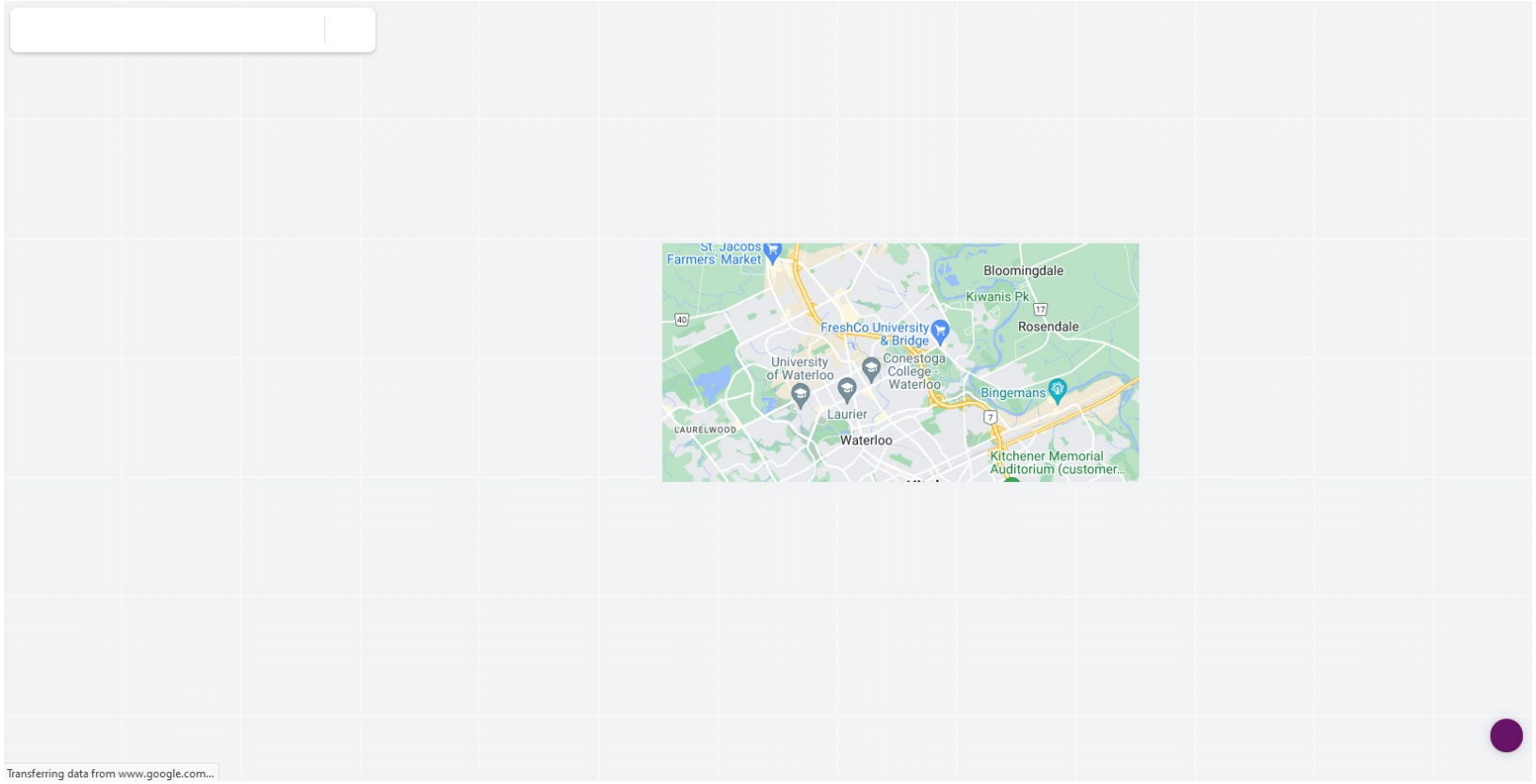
- Skeleton content



Skeleton Content

Provide user with a minimal version of an interface or data while the rest is being rendered or loaded. This can be generic layout or minimal version of actual content.

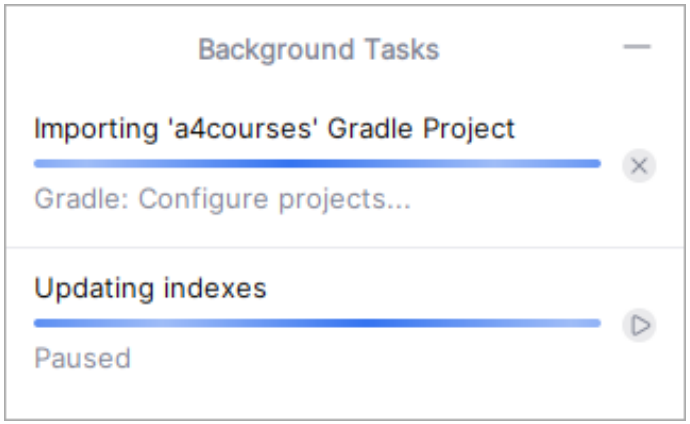
- Users adjust to a layout they will eventually see
- Users can act upon the limited amount of data they are already seeing
- Loading process seems faster because there is an initial result



Responsive User Interfaces

Perform low-priority system tasks in the background, e.g.:

- Background Tasks



Performing CPU-intensive Tasks

Handling “Long Running Tasks” in a User Interface

How do we handle tasks that take a significant amount of time to execute?

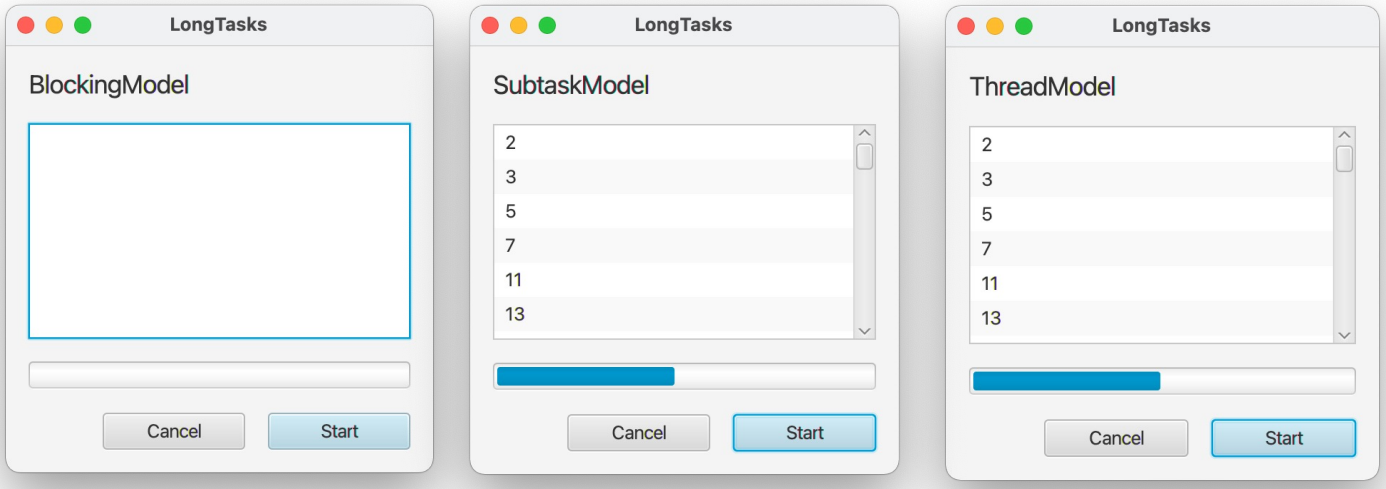
Goals

- keep UI responsive
- provide progress feedback
- allow long task to be paused or canceled

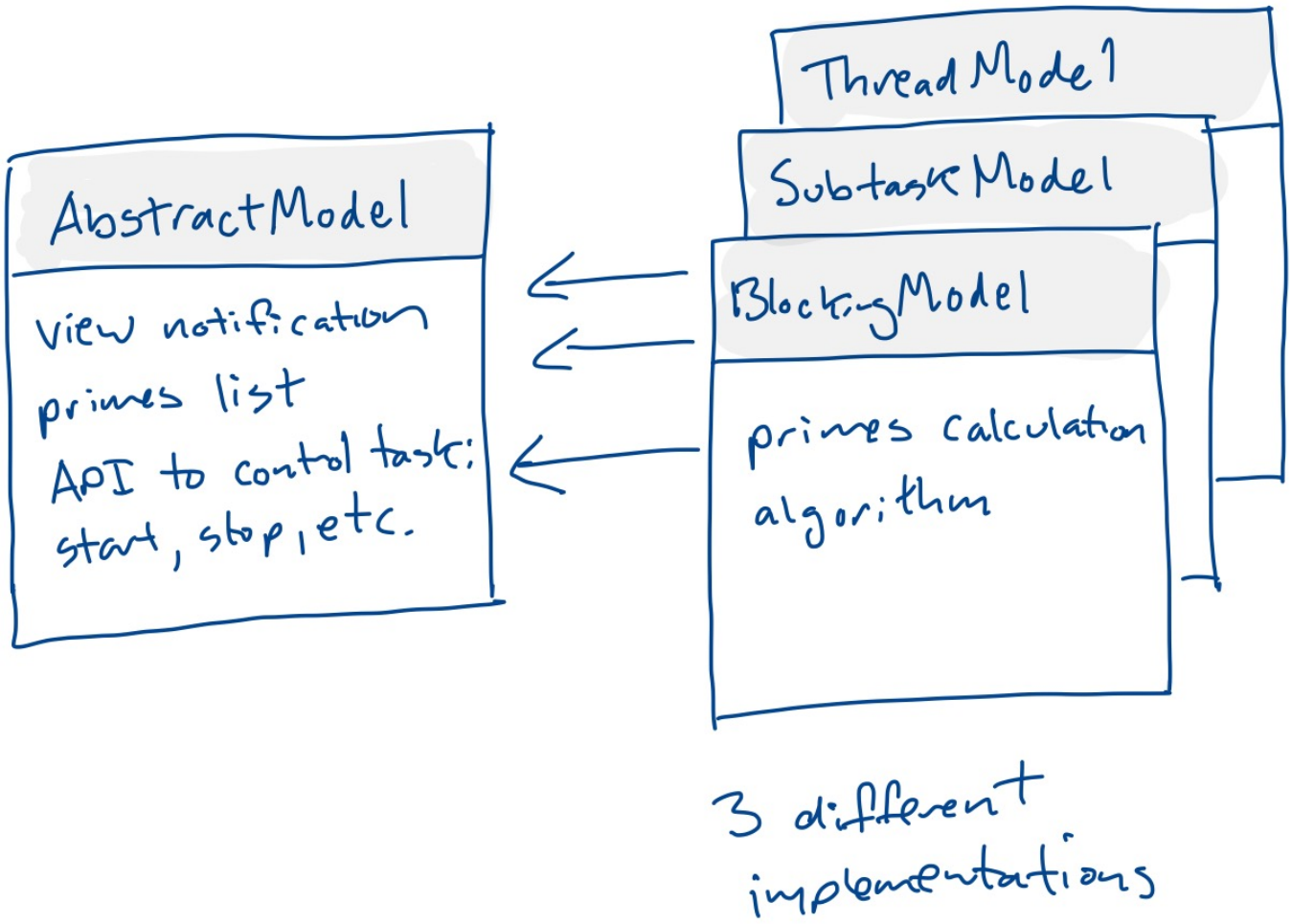
Do these even if it takes a bit longer to complete the long task!

Demo: "Responsiveness/LongTasks"

- Computes prime numbers (as example of long task)
- Uses MVC architecture
- 3 different approaches implemented in 3 different Models (switch Model to use in Main)
  - BlockingModel
  - SubtaskModel
  - ThreadModel



Demo Model Architecture



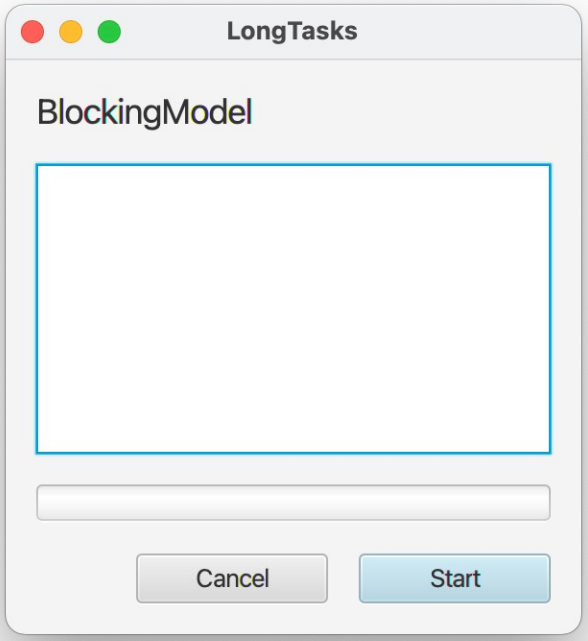
BlockingModel (what not to do)

- calculates primes in a loop on the UI thread
  - can take 5 – 10s for 250K range
  - "blocks" UI thread, appears to freeze

```
override fun calculatePrimes() {

    while (current <= max) {
        if (isPrime(current)) {
            addPrime(current)
        }
        current += 1
    }

    isRunning = false
    notifyObservers()
}
```



Implementation Approaches for Long Tasks

1. Subtasks (running on UI thread)

- Periodically execute subtasks between handling UI events

2. Run task in different thread (worker thread)

- Use thread-safe API to communicate between worker and UI

- Both strategies let UI control task and let task send feedback to UI

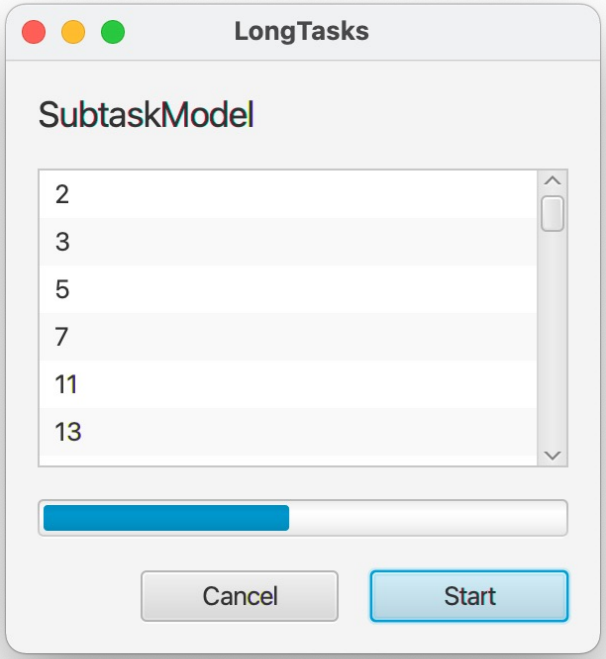
  - `fun run()`
  - `fun cancel()`

  - `running: Boolean`
  - `cancelled: Boolean`

  - `progress: Double`

SubtaskModel

- Task object keeps track of current task progress
- Subtasks periodically called on UI thread
- JavaFX: Platform.runLater()
- Every time object told to “run” for a bit, it checks current progress, executes subtask, updates progress, cancels if asked, …



SubtasksModel main method

- using recursion
- each call runs for a small number of milliseconds

```
override fun calculatePrimes() {
    Platform.runLater {
        calculateSomePrimes(100)
        if (!cancelled && current <= max) {
            calculatePrimes()
        }
    }
}
```

SubtasksModel main method

- runs algorithm as subtask for a time slice

```
private fun calculateSomePrimes(duration: Long) {
    val start = System.currentTimeMillis()
    while (true) {
        if (current > max) {
            isRunning = false
            notifyObservers()
            return
        } else if (System.currentTimeMillis() – start >= duration) {
            notifyObservers()
            return
        } else if (isPrime(current)) {
            addPrime(current)
        }
        current += 1
    }
}
```

Why use Subtasks?

- Advantages:
  - Can more naturally handle “pausing” (stopping/restarting) task because it maintains information on progress of overall task
  - Useful in single-threaded platforms (e.g. microcontroller)
- Disadvantages:
  - Maybe be difficult to predict length of time for subtasks
  - Not all tasks can easily break down into subtasks (e.g., Blocking I/O)

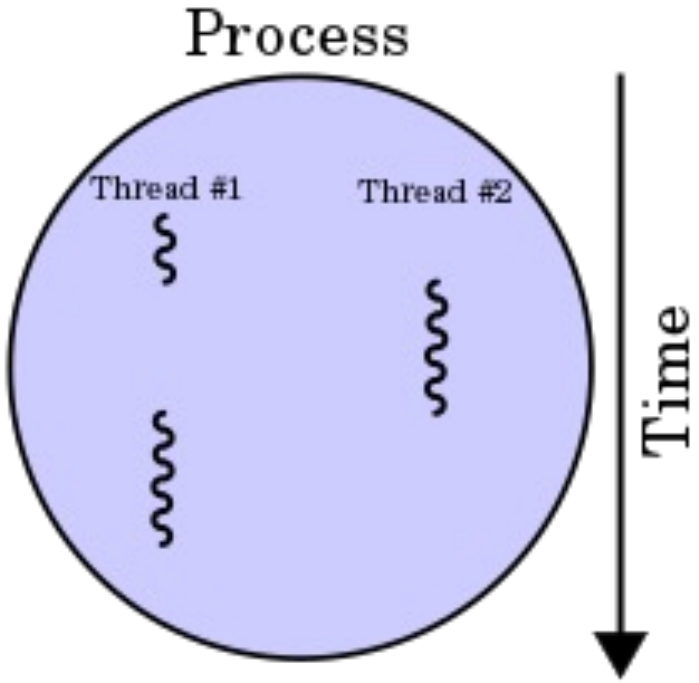
Sidebar: Definitions

Process: an instance of a computer program that is being executed

Thread: the smallest unit of execution on a CPU. A thread executes a series of instructions on behalf of a program

Threads are associated with a program, and may share some resources (like memory)

A process usually contains a single thread, but it's possible to structure a program to have multiple threads that can be processed concurrently (aka multi-threading)



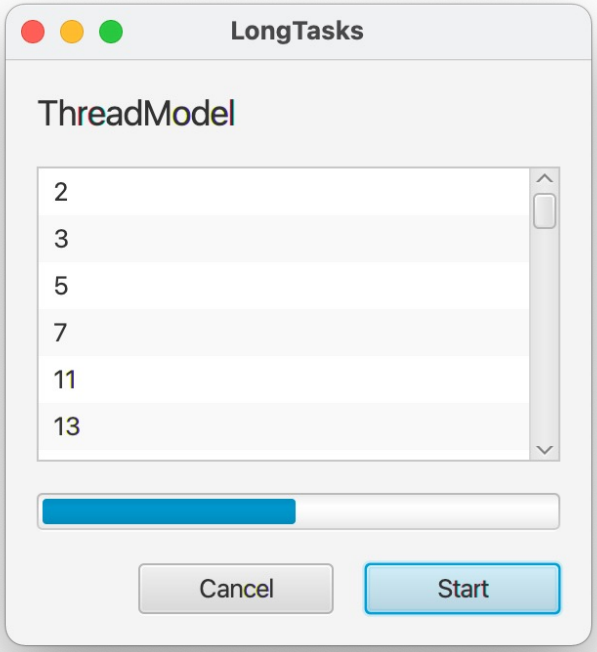
These are simplified definitions for threading concepts related to user interfaces.

Threads and Multi-Threading

- **Multi-threading:** manage multiple concurrent threads with shared resources, but executing different instructions
- Threads are a way to divide computation, reduce blocking.
- Concurrency has risks: what if two threads update a variable?
- Typically, three types of threads in a UI application:
  - one **main application** thread
  - one **UI Thread** (Java calls it Event Dispatch Thread - EDT)
  - 0 or more **worker threads** (also called “background threads”)

ThreadModel

- Long method runs in a separate thread
  - Thread class
  - Runnable object
- Method regularly checks if task should be cancelled and reports back to UI about progress (by updating views)



```
override fun calculatePrimes() {

    object : Thread() {
        override fun run() {
            isRunning = true
            var start = System.currentTimeMillis()
            while (true) {
                if (cancelled || current > max) {
                    isRunning = false
                    updateAllViewsInUiThread()
                    return
                } else if (isPrime(current)) {
                    addPrime(current)
                }
                current += 1
                if (System.currentTimeMillis() - start >= 100) {
                    updateAllViewsInUiThread()
                    start = System.currentTimeMillis()
                }
            }
        }

    }.start()
}
```

```
private fun updateAllViewsInUiThread() {
    Platform.runLater { notifyObservers() }
}
```

```
@Synchronized
protected fun notifyObservers() {
    for (view in views) {
        view.update()
    }
}
```

@Synchronized keyword

- Java uses the monitor abstraction for concurrency
  - Conceptually higher level than semaphores and mutexs
  - Goal is to enforce exclusive access to critical sections
  - All threads block until a condition is met
- synchronized methods can only be access by one thread a time
  - e.g. why synchronize methods to modify or return a Vector? (addPrime and getPrimes in AbstractModel)

Threading Summary

- Advantages:
  - Conceptually, easiest to implement
  - Takes advantage of multi-core architectures
- Disadvantages:
  - Need to be careful about inter-thread communication
  - All the usual threading caveats: race conditions, deadlocks, …

Thread-Safety

- Most UI toolkits (like JavaFX or Swing) are not **thread safe**
- Can't call toolkit methods or access widgets from other threads
- Invoke code to run on the UI thread:
  - e.g. `Platform.runLater`, `SwingUtilities.invokeLater`
- Handle concurrency by protecting critical sections of code
  - e.g. @Synchronized annotation

- Also see **coroutines** in Kotlin

https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.concurrent/thread.html
https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-synchronized/
https://kotlinlang.org/docs/coroutines-overview.html

End of the Chapter

Any further questions?