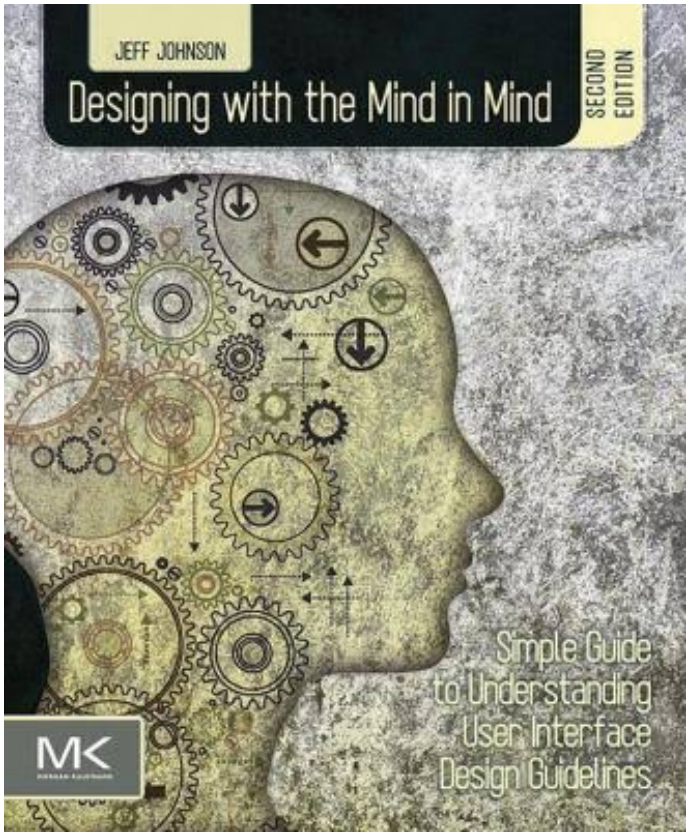
Responsiveness

- Human perception
- Responsiveness in Java



Chapter 14

Responsiveness

Responsive Applications

User experience is affected by how responsive an application *appears* to be to the user.

Previously, we talked about adapting user interfaces to particular devices (e.g. responsive layout when adjusting screen size).

But… it also means delivering data and feedback to users in a timely manner. i.e. making applications *feel* responsive.

We can achieve responsiveness in two ways, by

1. designing the UI to meet human deadline requirements
2. loading data efficiently so that it's available quickly

Responsiveness

Perceiving objects and events takes time. But how much time?

Knowing the duration of perceptual and cognitive processes can help us design interactive system that seems responsive.

Think about interactive systems as having **deadlines**, where these deadlines are imposed by the user's real time needs.

Responsiveness

What factors affect user's perception of responsiveness?

User expectations

- how quickly a system "should" react, or complete some task.

The system's ability to

- keep up with the user.
- keep them informed about its status.
- not make them wait unexpectedly.

Researchers have found that responsiveness is the most important factor in determining user satisfaction, more so than ease of learning or ease of use.

Responsiveness is not Performance

Responsiveness ≠ Performance!

- This is a human-factors deadline, which *may or may not* be helped by improving application performance.

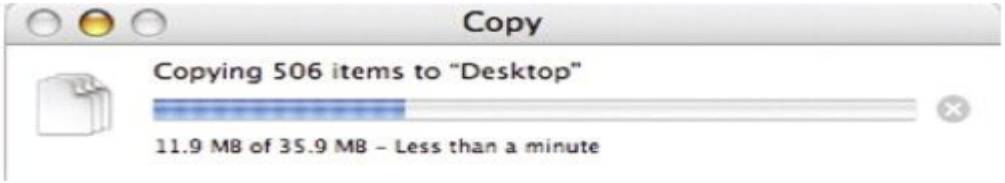
Performance = computations per unit of time

Responsiveness = compliance with human time requirements

Slow Performance, but Responsive

Interactive systems can be responsive to a user's needs, despite low performance, by

- providing feedback about what the user has done (e.g., let them know that their input was received)
- keeping users informed if they cannot fulfill the requests immediately, or free them up to do other things while waiting.
- providing feedback about what is happening (e.g., indication of how long the operations will take).
- allow users to perform other tasks while waiting



Fast Performance, but not Responsive

Fast systems can have poor responsiveness when they do not meet human time deadlines.

Things that make applications appear unresponsive:

- delayed feedback for button press, scroll movement, etc.
- ignore user input completely (e.g. wait-cursor)
- providing no clues on how long the operation will take
- jerky, hard to follow animation
- ignoring user input while performing “housekeeping” tasks and other time-consuming operations.

How long does the brain take to … ?

shortest gap of silence we can detect in a sound	0.001s
shortest time a visual stimulus can be shown and still affect us	0.01s
duration of saccade during which vision is suppressed	0.1s
maximum interval between cause-effect events	0.14s
time for a skilled reader's brain to comprehend a printed word	0.15s
visual-motor reaction time to inspected events	1s
duration of unbroken attention to a single task	6-30s

Deadline Implications

0.001 second

- **minimum detectable silent-audio gap**

Implications:

- people can only tolerate ~0.001 second of delay in audio feedback.

Deadline Implications

0.01 second

- shortest noticeable pen-ink lag
- **preconscious (unconscious) perception**

Implications:

- the lag time for electronic ink transmission must be under 0.01s
- we can induce unconscious familiarity of images / symbols by making them disappear within 0.01s of appearing.

Deadline Implications

0.14 second

- perceiving the number of 1-4 items
- involuntary eye movement (saccade)
- audiovisual “lock” threshold
- **perception of cause and effect**
- **perceptual-motor feedback**
- object identification
- perceptual moment

Implications:

- if software waits longer than 0.1 second to show a response to user’s action, the perception of cause and effects is broken. The software’s reaction will not seem to be a result of user’s action.
- Busy indicator should be used if an operation takes longer than the perceptual moment.

Deadline Implications

1 second

- max conversation gaps
- **visual-motor reaction time for unexpected events**
- attentional blink

Implications:

- It is necessary to display progress indicators for a long operation.
- If information suddenly appears on the screen, it will take users at least 1 second to react to it. The system can make use of this lag time to its advantage (e.g., present a “fake” inactive version of the object first).

Designing for Responsiveness

- Leveraging human cognition
- Making an application “feel” responsive

Busy Indicators

- Use for any function that blocks user actions, even if the function normally executes quickly.

animated

static

Progress Indicators

- *Better* than busy indicators because they let users see how much time remains.
- Should be displayed for any operation that will take longer than a few seconds

"Responsiveness" UI Tricks

Progress Indicators best practices (McInerney and Li, 2002)

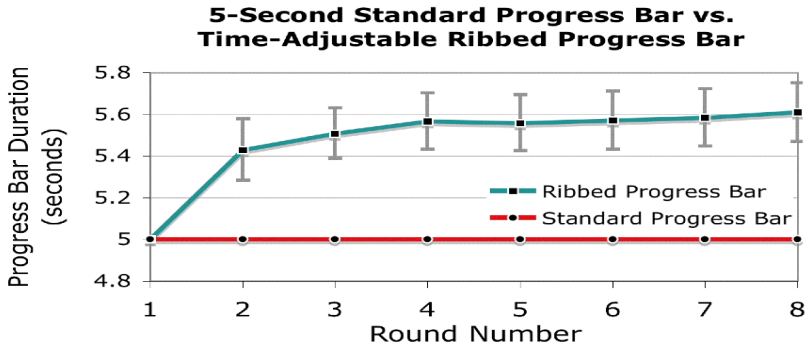
- Show work remaining, not work completed.
- Show total progress, not progress on current step.
- Show percentage of an operation that is complete, start at 1% and not 0%.
- Display 100% only very briefly at the end of an operation.
- Show smooth, linear progress, not erratic bursts of progress.
- Use human-scale precision, not computer precision (Bad: 240 seconds, Good: about 4 minutes)

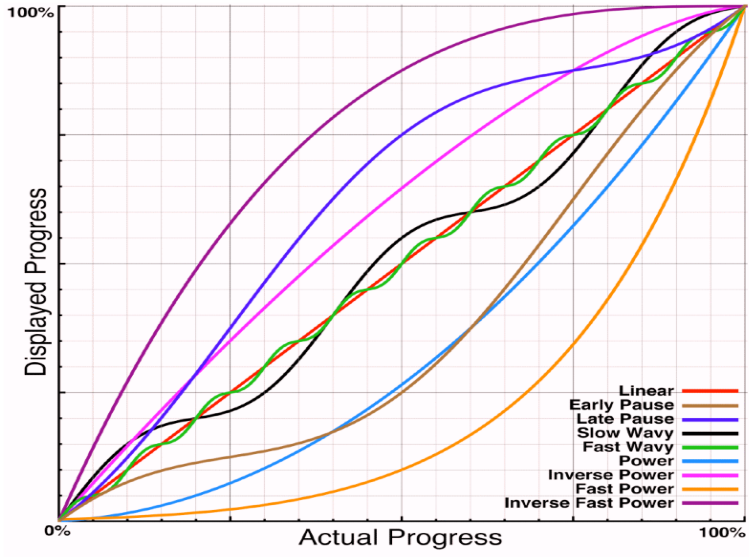
"Responsiveness" UI Tricks

- "Faster Progress Bar: Manipulating Perceived Duration with Visual Augmentations" and "Rethinking the Progress Bar" (Chris Harrison CMU)
  http://www.chrisharrison.net/index.php/Research/ProgressBars2

Figure 3. Progress bar used in Mac OSX.

Figure 4. Ribbed progress bar used in the study.





"Responsiveness" UI Tricks

Render / Display important information first

- Provide the user with data to work with while loading subsequent data
- Examples
  - document editing software that shows the first page as soon as users open a document, rather than waiting until it has loaded the entire document.
  - web or database search function that displays items as soon as it finds them, while continuing to search for more.
  - webpages that display images at low resolution first, then re-render at higher resolution.

"Responsiveness" UI Tricks

Working ahead

- Use periods of low load to pre-compute responses to high probability requests. Speeds up subsequent responses
- Examples
  - text search function that looks for the next occurrence of the target word while you look at the current one.
  - document viewer that renders the next page while you view the current page.
  - web browser which pre-downloads linked pages.
  - using cursor movement to predict and pre-compute button clicks in interfaces.

"Responsiveness" UI Tricks

Fake heavyweight computations during hand-eye coordination tasks (e.g., scrolling, moving a game character, resizing a window, dragging)

- Users will perceive the application as unresponsive if feedback lags behind user actions by more than 0.1 seconds.
- Example
  - graphics editor that fakes feedback by providing rubber-band outlines of objects that a user is trying to move or resize.

"Responsiveness" UI Tricks

"Batch" processing to avoid interrupting users

- Avoid doing processing during user interaction
- Example
  - validate after users hit "enter", not on a character by character basis.
  - e.g. navigation system prompts the user to enter the City, then Street, then Address, and validates every keystroke.

There's a 1-2 second delay for every key pressed, since it does validation at every step. Compare this to entering an address into Google Maps, which does validation once.

Responsiveness in Java

- Handling long-running tasks

Goal for Long Tasks

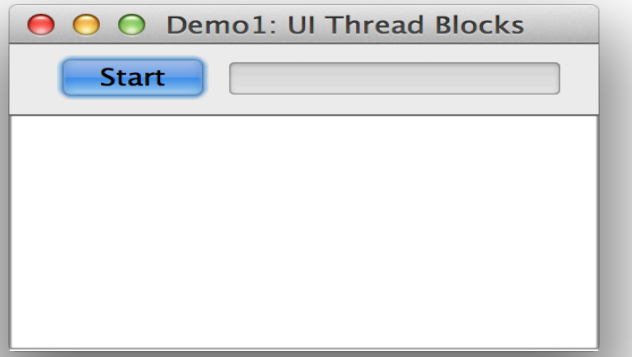
Long-running or complex tasks may take longer than these human requirements allow

- Goal: maintain a highly interactive application (even if this means taking a bit longer to complete the task)
  - keep UI responsive
  - provide progress feedback
  - allow long task to be paused or canceled

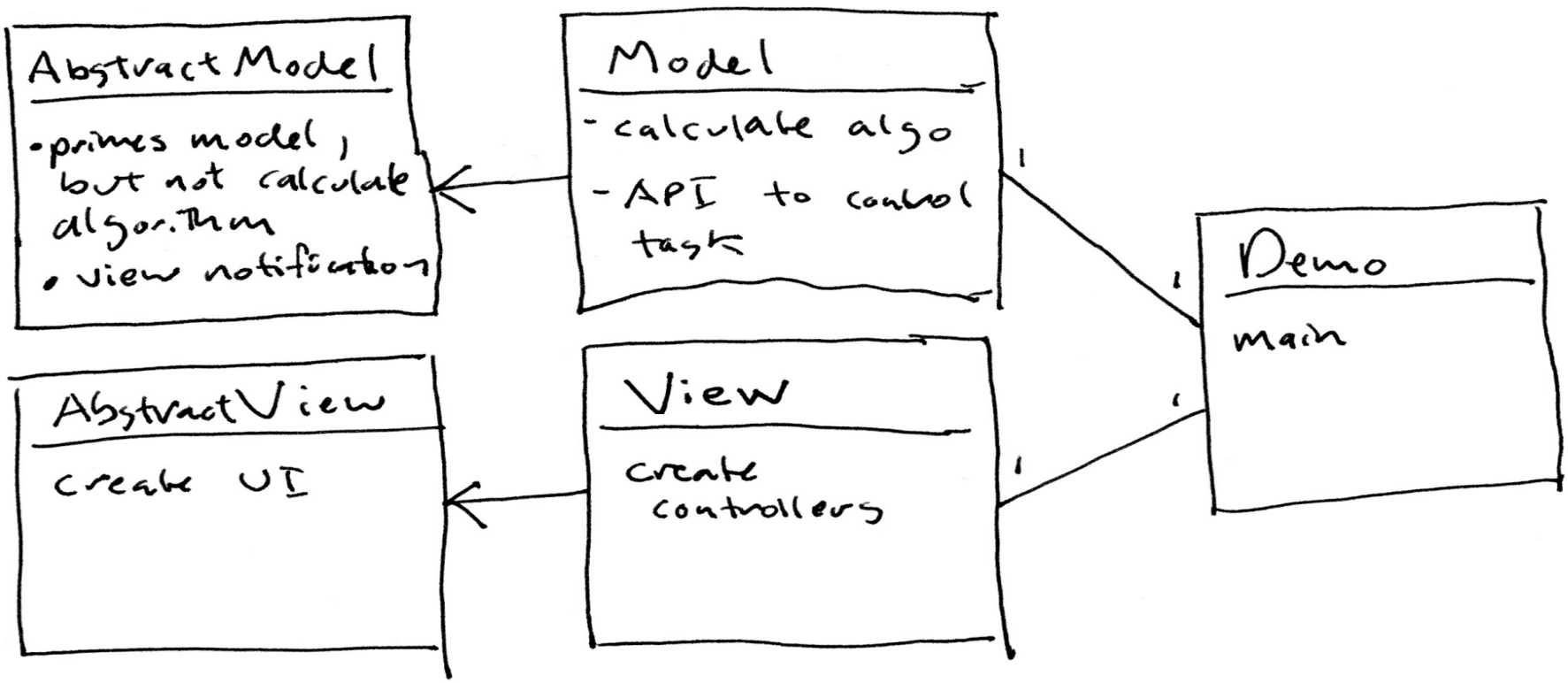
Long Tasks

- What should you do when a task will take significant time?
- Examples
  - Fetching a large image or long list over a (slow) internet connection
  - Searching a directory structure
  - Performing an image processing operation
  - Factoring a large number
  - Reading a large file
  - etc.

Demo1.java: Calculate Primes
aka **what not to do!**



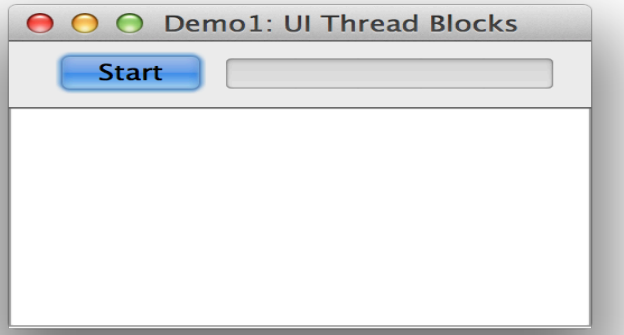
Demo1, Demo2, and Demo3 MVC Architecture



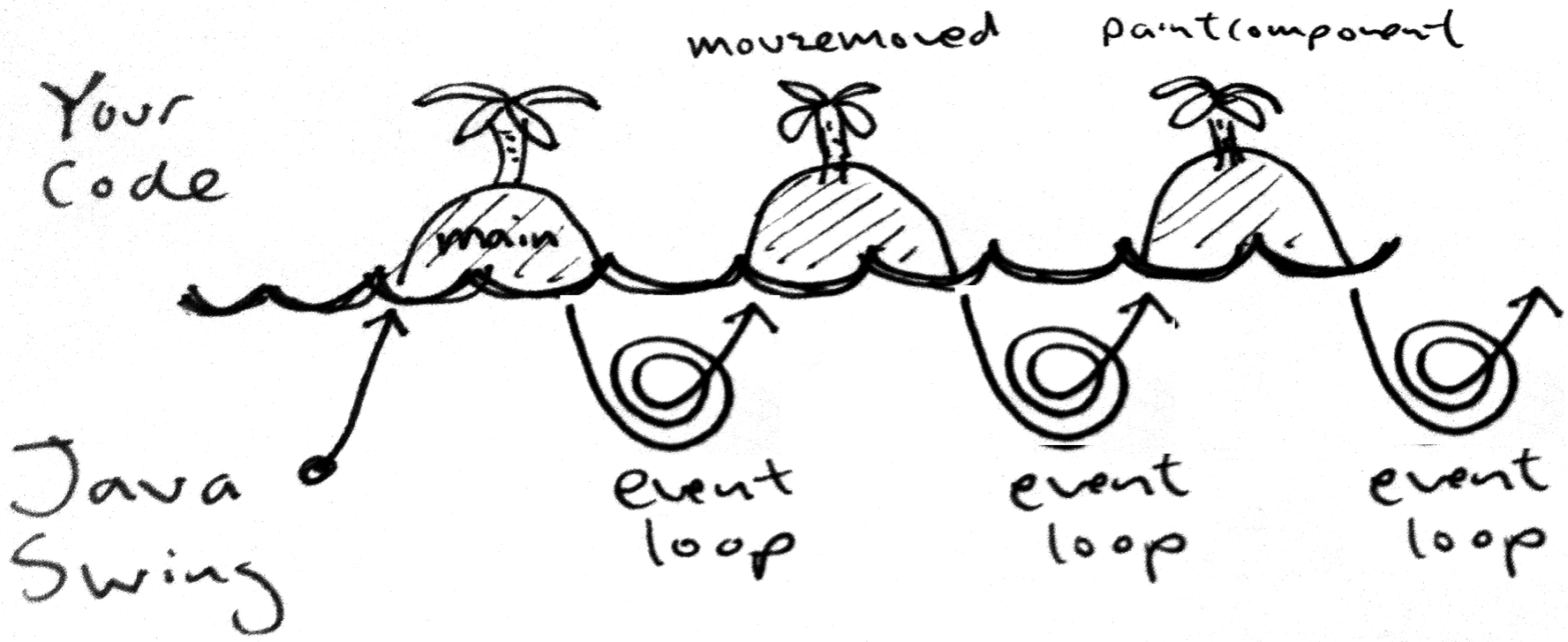
Demo1.java (what not to do)

```
protected void registerControllers() {
   // Handle presses of the start button
   this.startStopButton.addActionListener(new
   ActionListener() {
          public void actionPerformed(ActionEvent e) {
              model.calculatePrimes();
          }
       });
}
```

Find primes in [1, 250000]
Takes ~10 seconds to complete



What's wrong?


mouseMoved
paintComponent
Your Code
main
Java Swing
event loop
event loop
event loop

Solution

Java supports multi-threading

- A **thread** is the smallest "stream" of execution in a program.
- Some systems allow multiple threads to operate concurrently (i.e. shared resources/memory, but executing different instructions).
- Allows multiple users to utilize a program, or divide computation.
- Risk: concurrent updates/writes can cause issues.
  - e.g. what happens if two thread update the same variable?

Three types of threads in any Java application:

1. Initial Thread
2. Event Dispatch Thread (EDT) (or "UI Thread")
3. Worker Threads (or "Background Threads") - *optional*

Two Strategies for Long Tasks

Strategy A: Run in UI thread

- Break task into subtasks
- Periodically execute each subtask on the UI thread (between handling regular events)

Strategy B: Run work in separate thread

- Create a “worker thread” (with a thread-safe API)
- Communicate w/ UI thread

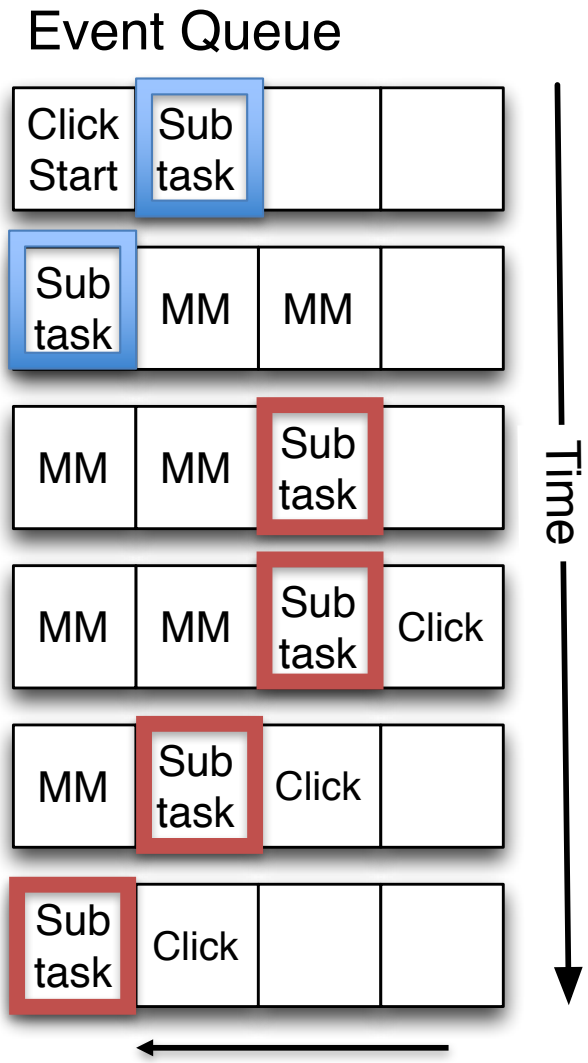
For both strategies, provide an API for the UI to control tasks and for tasks to provide feedback to the UI:

- void run()
- void cancel()
- int progress()
- boolean isRunning()
- boolean isDone()
- boolean wasCancelled()

Strategy A: Run in UI Thread (Demo2)

- Task object keeps track of current task progress
- Task object adds a short-running subtask to the Swing event queue. Subtask makes some progress on the overall task.
- At the end of the subtask, check if there's more work. If so, add another short-running subtask to the back of the event queue.
- Handle any events that accumulated while the subtask was running.

"Click start" is a mouse-click event.
MM is a mouse-move event.
"Click" is another mouse-click.

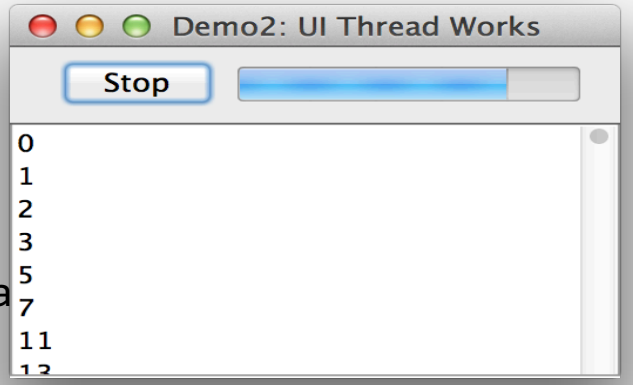


Strategy A: Run in the UI Thread

```
class Model2 extends AbstractModel {
    private boolean cancelled = false;
    private boolean running = false;
    private int current = 0;    // progress so far

    public Model2(int min, int max) {  super(min, ma

    public void calculatePrimes() {
        this.running = true;
        SwingUtilities.invokeLater(
            new Runnable() {
                public void run() {
                    // calculate primes for 100 ms
                    calculateSomePrimes(100);
                    if (!cancelled && current <= max) {
                        calculatePrimes();
                    }
                }
            });
    }
```



1 Housekeeping

3 Add object to event queue

2 Create an object

run() does not execute until something (the event loop!) calls it.

Strategy A: Run in the UI Thread

```
private void calculateSomePrimes(long duration) {
    long start = System.currentTimeMillis();
    while (true) {
        if (this.current > this.max) {
            this.running = false;
            updateAllViews();
            return;
        } else if (System.currentTimeMillis() - start >= duration) {
            updateAllViews();
            return;
        } else if (isPrime(this.current)) {
            this.addPrime(this.current);
        }
        this.current += 1;
    }
}
```

Strategy A: Run in the UI Thread

- Advantages:
  - More naturally handles "pausing" (stopping/restarting) task because it maintains information on progress of overall task
  - Can be run in Swing event thread or separate thread
  - Useful in single-threaded platforms (e.g., mobile)
- Disadvantages:
  - Tricky to predict length of time for subtasks
  - Not all tasks can easily break down into subtasks (e.g., Blocking I/O)

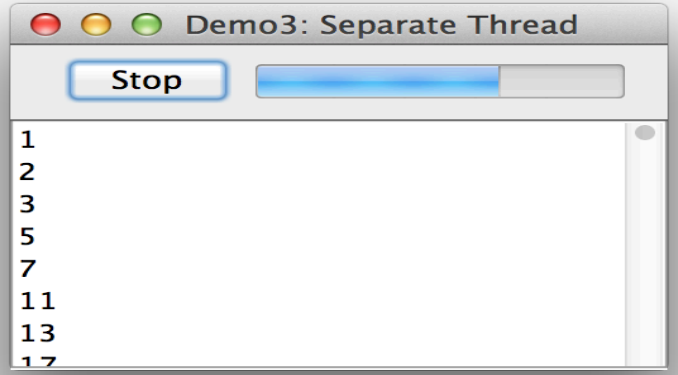
These are some big disadvantages!
It's better to use threads (Strategy B) when possible.

Strategy B: Another Thread (Demo3)

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

This is NOT as simple as just firing up another thread!

(Adding threads is never simple)



Strategy B: Separate Thread

```
class Model3 extends AbstractModel {
   ...
   public void calculatePrimes() {
      new Thread() {

         public void run() {
            ...
         }

         private void updateUI() {
            ...
         }

      }.start();
```

Strategy B: Separate Thread (Demo3)

```
public void run() {
    running = true;
    long start = System.currentTimeMillis();
    while (true) {
        if (cancelled || current > max) {
            running = false;
            updateUI();
            return;
        } else if (isPrime(current)) {
            addPrime(current);
        }
        current += 1;
        if (System.currentTimeMillis() - start >= 100) {
            updateUI();
            start = System.currentTimeMillis();
    }}}
```

Strategy B: Separate Thread

```
private void updateUI() {
   // We're calling this from a worker thread, which is not safe!
   // i.e. don't call updateAllViews() from another thread!
   // Instead, add to the Swing event queue and let Swing manage it
   SwingUtilities.invokeLater(
      new Runnable() {
         public void run() {
            updateAllViews();
         }
      });
}
```

- Make an object.
- Add it to the event queue.
- run() is executed when the object gets to the front of the queue.

Strategy B: Separate Thread

- Advantages:
  - Conceptually, the easiest to implement
  - Takes advantage of multi-core architectures
- Disadvantages:
  - Extra code required to be able to pause/restart method
  - All the usual Thread baggage
    - Race conditions
    - Deadlocks
    - Etc.

Swing is NOT Threadsafe (Demo4)

Has a JList to display a list of numbers.

Two kinds of threads. Both

- Generate a random number
- Tell JList to add it (if not already there) or remove it (if already there)

GoodWorkerThreads use invokeLater to put a task in the event queue telling JList to add/remove the number on the UI thread

BadWorkerThreads call the JList method directly (with a potential conflict as two threads try and update the JList at the same time!)

Using a Separate Thread

Lessons:

- Don't call Swing methods or access Swing components from outside the Event Dispatch thread
- From task thread, use `SwingUtilities.invokeLater()` to schedule code to run in the Event Dispatch thread
- If multiple threads access the same resource, consider the use of `synchronized` keyword to protect critical sections (not shown on slides).

Why aren't GUI toolkits thread-safe?

- https://weblogs.java.net/blog/kgh/archive/2004/10/multithreaded_t.html (removed; find at web.archive.org)

Two sets of abstractions that flow in opposite directions:

  - User-initiated threads travel "down" to the hardware to run (e.g. start a thread to find a bunch of primes)
  - Events travel from hardware up to higher-level abstractions (e.g. button-click to cancel finding primes)
- Any locking protocol for these two kinds of abstractions will conflict
- There is a long history of very smart people trying to build thread-safe toolkits.

Threading in Swing

- Three types of threads:
  - Initial Thread
  - Event Dispatch Thread (EDT) (or “UI Thread”)
  - Worker Threads (or “Background Threads”)
- http://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html

```
public static void main(String[] args) {
   SwingUtilities.invokeLater(new Runnable() {
      public void run() {
         new createAndShowGUI();
      }
   });
}
```

Scheduling Code to Run on UI Thread

- invokeAndWait()
- invokeLater()
- Runnable interface (vs Thread class)

```
private void updateUI() {
   SwingUtilities.invokeLater(new Runnable() {
      public void run() {
         ... do something on UI thread ...
      }
   });
}
```

SwingWorker

- As of Java SE 6, there's a standard way to create worker threads: SwingWorker
- Good introductory tutorials:
  - http://www.javacreed.com/swing-worker-example/
  - http://www.javaadvent.com/2012/12/multi-threading-in-java-swing-with.html
- Android has something similar, AsyncTask
  - http://developer.android.com/reference/android/os/AsyncTask.html

Long Tasks and MVC

- Long tasks start to break clean separation of MVC
- Model's methods need to be designed to allow user to stop them, to maintain interactivity
  - Needed to service event queue
  - Needed to allow user to stop method
- May need methods to inquire about length of task completion
  - Not part of "model" – part of interaction
- Usability concerns are thus directly influencing design of model to accommodate user interaction

Summary

- Understand the difference between performance and responsiveness
- Design User-Centric applications for responsiveness
- Think about giving/allowing user to do something else while long tasks execute.
  - Create non-blocking UIs and opportunities to cancel.
  - Take advantage of perceptual limits of users to fake interaction when possible.
  - Threading is your friend. Sometimes.