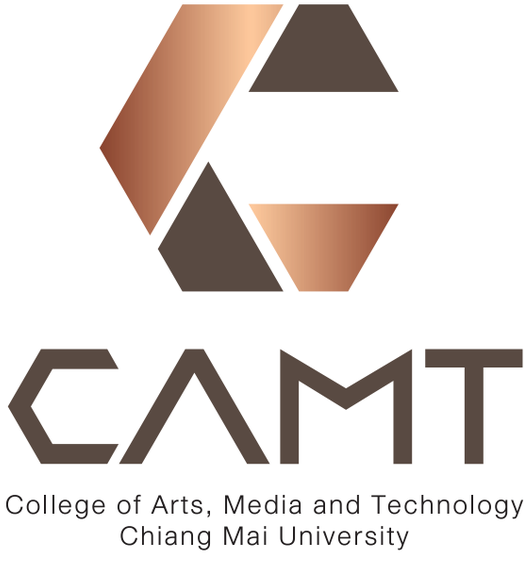
SE 233 Advanced Programming

Chapter III Multithreaded programming



Lect Passakorn Phannachitta, D.Eng.

passakorn.p@cmu.ac.th

College of Arts, Media and Technology

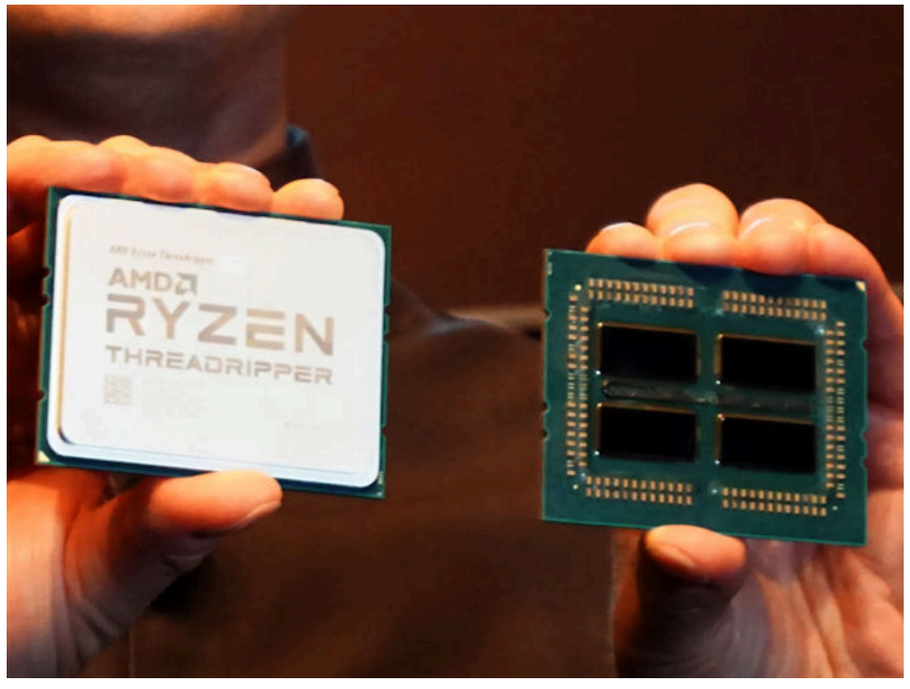
Chiang Mai University, Chiangmai, Thailand

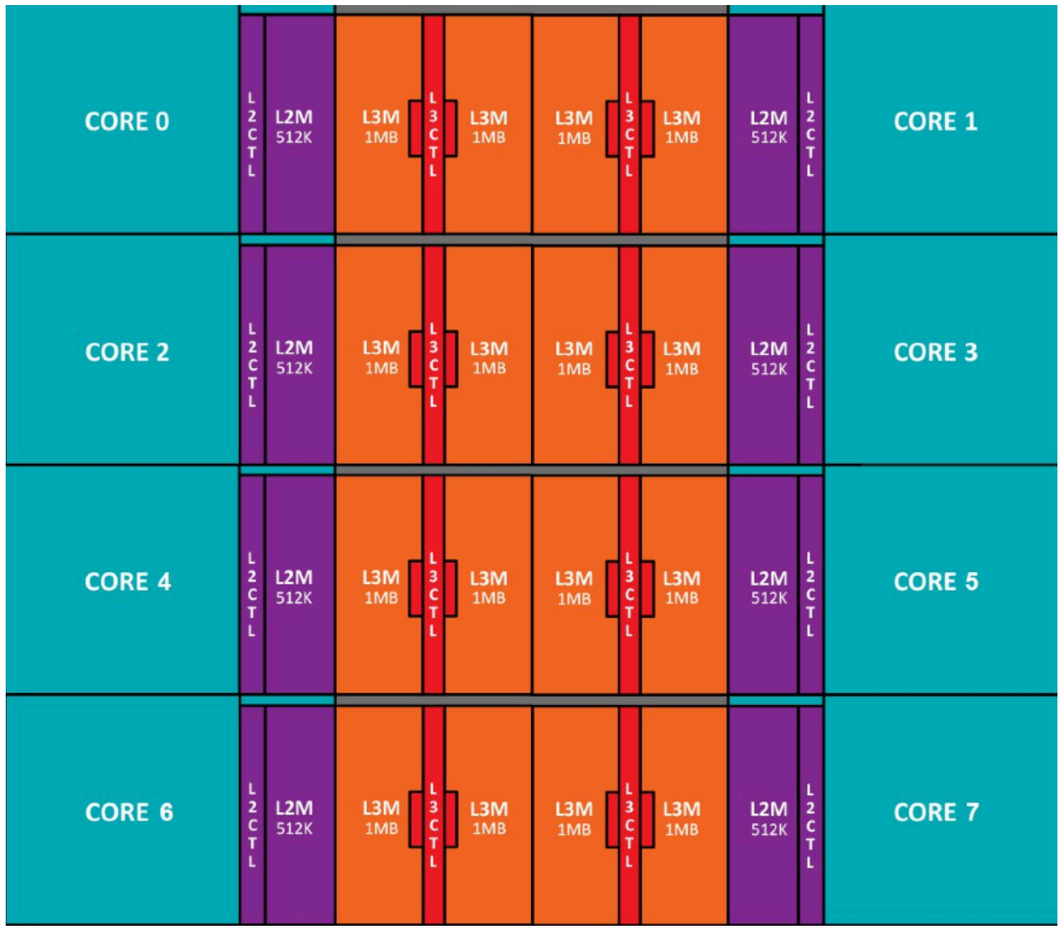
Agenda

- Multithread and multiprocess
- Multithreaded and GUI programming
- Multithreaded and accelerated computing
- TheJavaFX concurrent framework
- Real-world example project

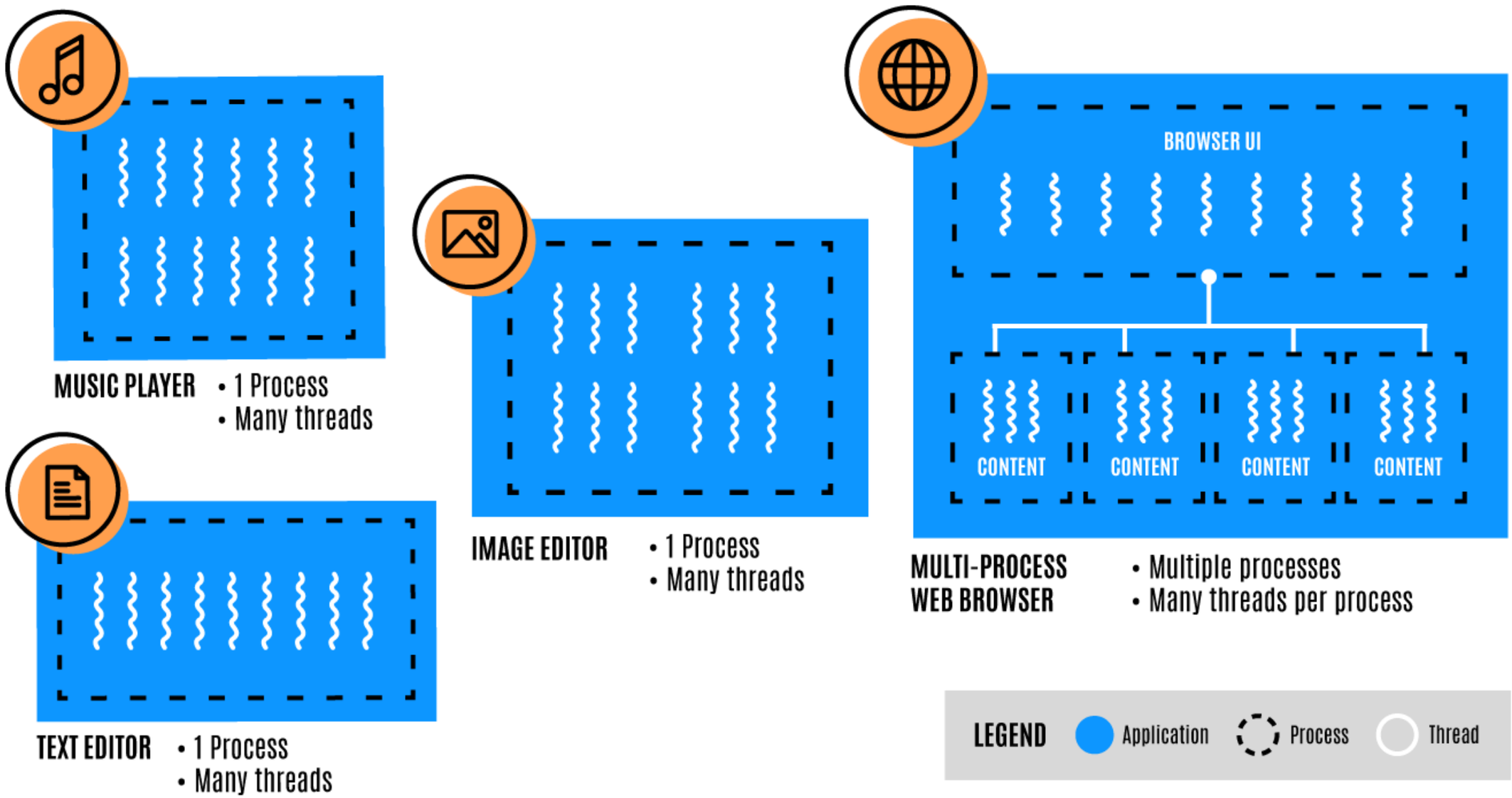
Modern commodity computer

- Multicore CPUs





Can execute multiple tasks concurrently

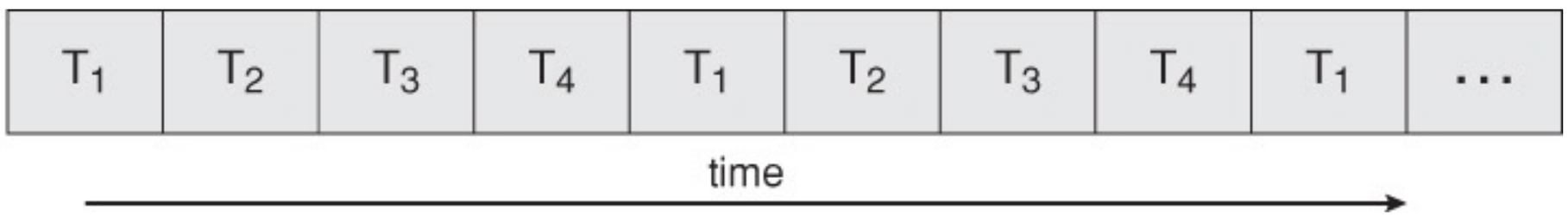


ref: https://www.extremetech.com/wp-content/uploads/2017/06/Process-and-Threads.png (May 2020)

Multithread and multiprocess

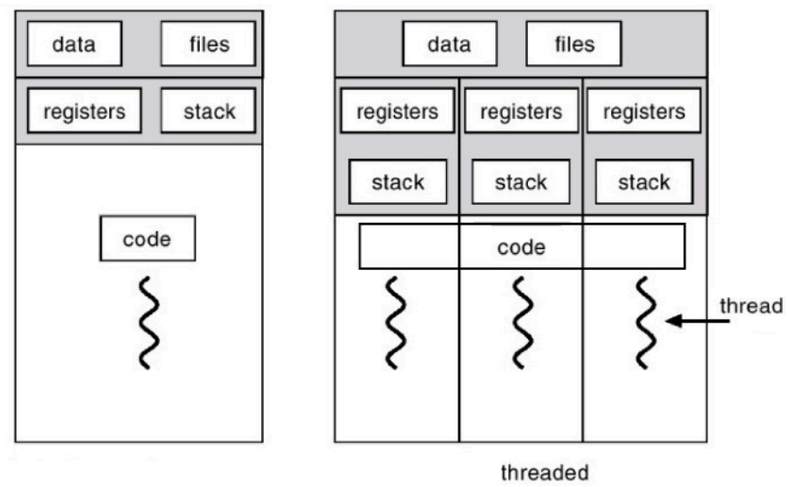
- Single-Threaded Programming
  - All code runs in a single line of execution
  - Input -> Sequence of processes -> output

- Consider a kind of wasted resource utilization



Multithread and multiprocess

- Basic units of execution
  - process and thread.
- A running program can be called a process
- Threads exist within a process



Multiprocessing

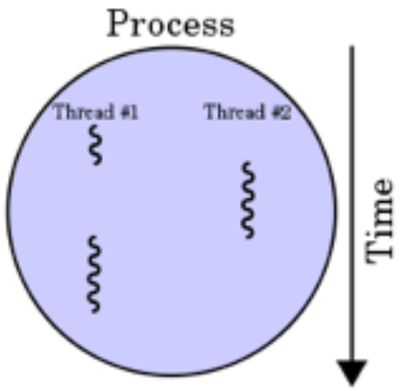
- Executing multiple processes (programs) at the same time
- Each process has its registers, program counter, stack pointer, and memory space.
- Thus, multiple processes are executed as if they do not know each other.

Main advantage

— More works can be done in a shorter period of time

Multithreading, a.k.a. lightweight process

- A model that allows a single process to have multiple code segments running concurrently within the “context” of that process
- Shared the most resources as possible
- Threads share the same address space
  - Changes made by one thread may be read by others.
  - So, multithreaded programming is also known as shared-memory multiprocessing



Main Differences

Multiprocessing	Multithreading
The system allows executing **multiple programs and tasks** at the same time	The system executes **multiple threads of the same or different processes** at the same time.
Increases the **computing speed** of the system	Increase the **responsiveness** of the system
CPU has to switch between **multiple programs** so that it appears that **multiple programs are running simultaneously**	CPU has to switch between **multiple threads** to make it appear that **all threads are running simultaneously**
Allocates separate memory and resources for each process/program	Threads belonging to the same process shares the same memory and resources as that of the process

Main Differences (Cont)

Multiprocessing	Multithreading
Forking a process to work in **non-blocking mode**, allocates **separate code and data space** for each child process	A multitasking within a process, where **child threads share the same code and data space** with parent thread
Heavyweight tasks that **require their own address space**.	Lightweight process and can **share same address space**
Interprocess **communication** is **very expensive** and context switching from one process to another process is **costly**	Inter-thread **communication** is less expensive than interprocess

Multithreaded and GUI programming

- Without multithreading, it would be impossible to build a GUI application since all the control and visual elements will be unresponsive

Unresponsive GUI

- Problem with single threaded GUI: Unresponsive GUI

Consider another GUI with two buttons: when you click on "start", it activates it counts up once per second, and when you click on "close" the application terminates.

```
public void countup() {
    draw();
    for (int i = 1; i <= 10000; i++) {
        increment();
        try {
            Thread.sleep(5);
        }
        catch(InterruptedException e) {

        }
    }
}
```

Unresponsive GUI

```
public void countup() {
    draw();
    for (int i = 1; i <= 10000; i++) {
        increment();
        try {
            Thread.sleep(5);
        }
        catch(InterruptedException e) {
        }
    }
}
```

The single thread of control is executing the for loop (event loop), so it is unable check the button click until the loop has terminated.

Multithreaded and GUI programming

- A GUI application should consist of at least two threads:
  - an event dispatch thread, which handles all GUI events, e.g., mouse events, keyboard events, and timer events, and
  - a thread for event listeners, which are method objects that will be invoked to respond to the GUI events.

General rules for GUI programming

1. All GUI activity is on event dispatch thread
2. No other time-consuming activity on this thread
    - E.g., Wait for file-writing, fetching data from an API

- Since multithreading may refer to 10s or 100s or 1000s threads:
    - Fail to achieve (1), the program may not deliver the collect result
    - Fail to achieve (2), the program may run super slowly or not it does not terminate

Multithreaded and accelerated computing

- Even if the multiprocessing technique appears to be more suitable for a computational power-demanding task, we can also use the multithreading technique for the task.
- Planning is analogous to how we do a group project.
- The degree of synchronization

Synchornization

- Two concurrent threads must be able to get executed correctly without any interleaving of their instructions
- There will be entirely no problem **if** any two threads are operating on **completely independent data** or the **shared data/resources are read-only**,
- Otherwise, synchornization is required to safely coordinate the access of the shared resources

Possible problem


```
int withdraw (int account, int amount) {
    double balance = readBalance(account);
    double balance = balance – amount;
    updateBalance(account, balance);
    return balance;
}
```

```
int deposit (int account, int amount) {
    double balance = readBalance(account);
    double balance = balance + amount;
    updateBalance(account, balance);
    return balance;
}
```

- Suppose that the balance starts at $500 and then two concurrent processes withdraw $100 at the same time

Possible problem

- $500 - $100 - $100 could be 400!

```
balance = readBalance(account);
balance = balance - amount;
updateBalance(account, balance);
```

$500

```
balance = readBalance(account);
balance = balance - amount;
updateBalance(account, balance);
```

$500

```
balance = readBalance(account);
balance = balance - amount;
updateBalance(account, balance);
```

$400

```
balance = readBalance(account);
balance = balance - amount;
updateBalance(account, balance);
```

Synchornization

- “Synchronized” code ensures that
  - an object (the data) is only observed when it is in a consistent state
  - when one thread starts running code in a synchronized method, it sees all the modifications on the same object (data) made by other threads

Synchronization is tricky

- Too little => your program may deliver incorrect results
  - Changes are not guaranteed to propagate thread to thread
  - Program can observe inconsistencies
- Too much => your program may run slowly or not at all
  - E.g., Deadlock
- Therefore **Synchronized** is the opposite of **Concurrent**

Too much synchronization

Parallel Algorithm

$PFibonacci(n)$

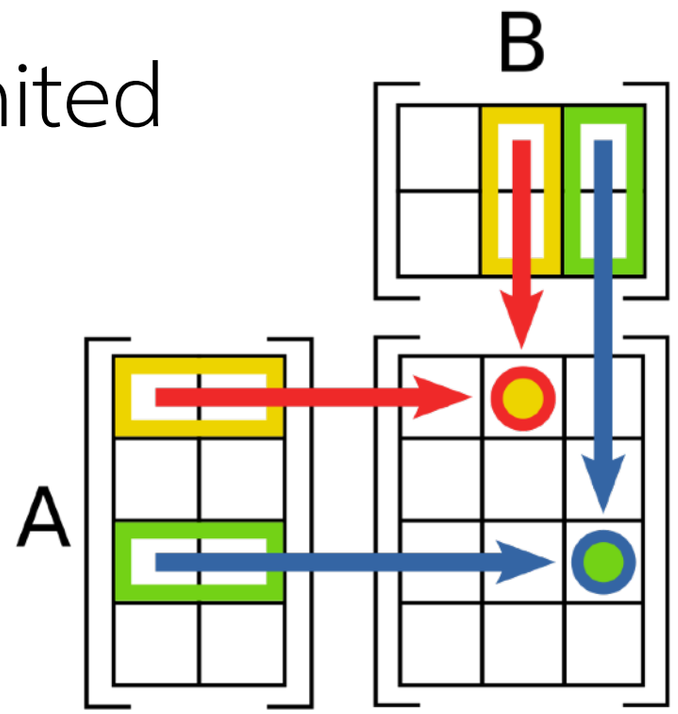
1. if $n \leq 1$ then
2. return n
3. else $x =$ **spawn** $Fibonacci(n-1)$
4. $y = Fibonacci(n-2)$
5. **sync**
6. return $x + y$

ref: https://www.slideshare.net/AndresMendezVazquez/24-multithreaded-algorithms (Access May 2020)

Embarrassingly parallelizable computation

- Each thread is entirely independent of the others
  - They examine different part of data
  - Write result to different array elements

- Anyway, we can achieve this in limited situations



JavaFX concurrent framework

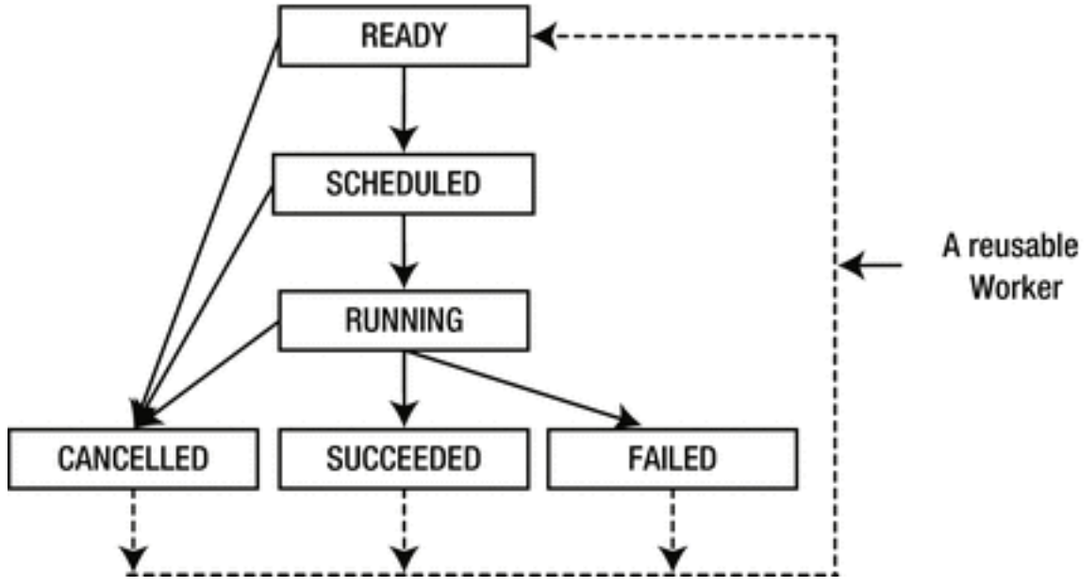
- **javafx.concurrent Package**
  - Improve the java.util.concurrent package by considering many constraints faced by GUI developers.
  - Consist of the **Worker** interface and four classes: **Task**, **Service**, **ScheduledService**, and **WorkerState**:
    - **Task**, **Service**, **ScheduledService:** choices to run code on background
    - **WorkerState:** an enum represents different states of **Worker**

The Worker interface

- A Worker<V> interface provides the specification for any tasks,
  - the generic parameter V is the data type of the result generated from the task implemented in the class implementing Worker.
  - The state of a Task is observable and is published on the Java FX application thread.
  - This makes it possible for a Task to communicate with the Java FX Scene graph

The Worker interface

- A Worker transitions through different states defined by the Worker.State Enum.



The Task class

- An instance of the Task<V> class is the where we implement our programming logic to be executed in a background thread.
  - By overriding the **call** method, put all the codes in it, and can execute it.
  - A Task instance can also be monitored throughout the execution

The Service class

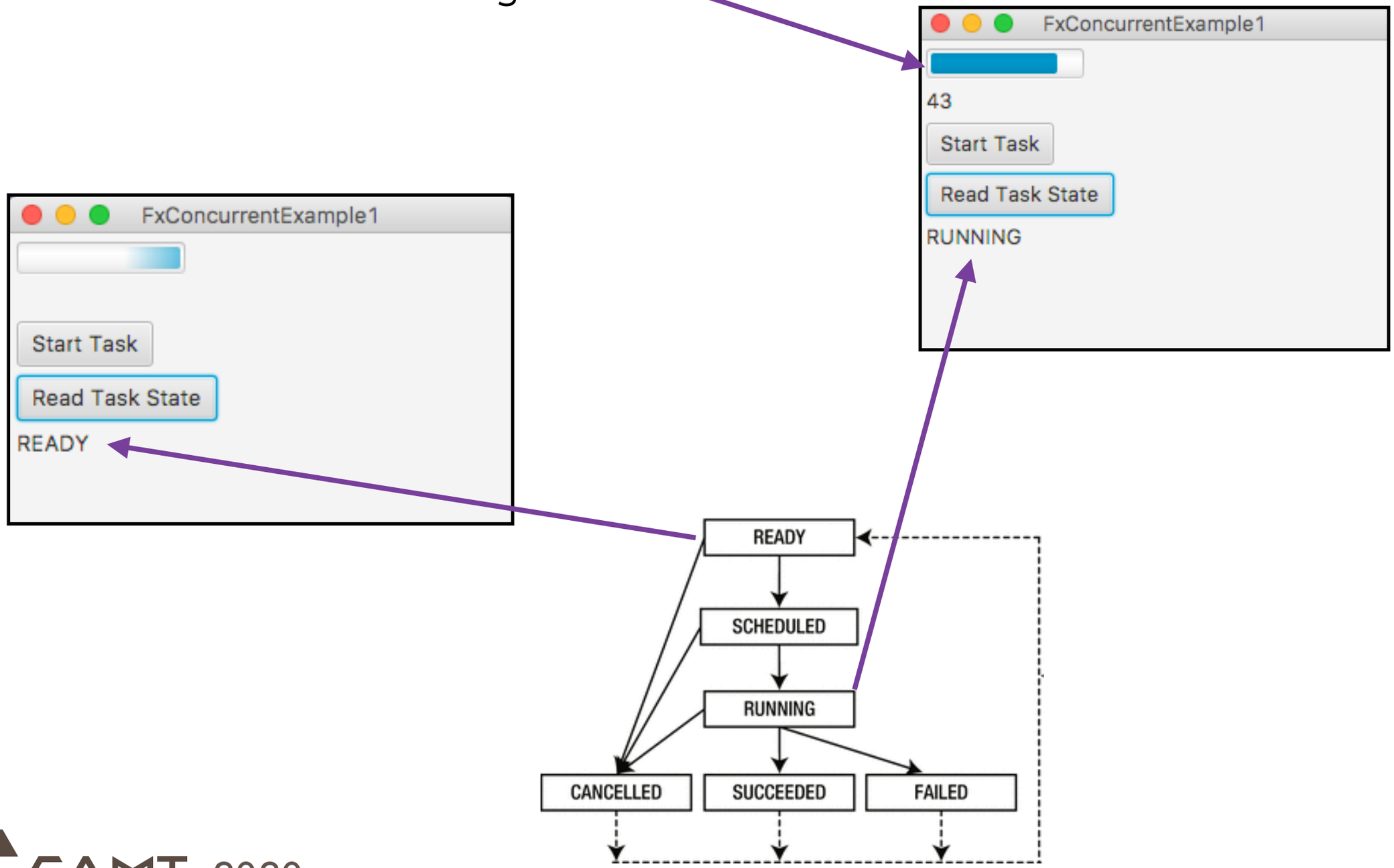
- A Service<V> instance encapsulates the Task<V> class and makes it reusable.
  - We can reset a Service instance but not a Task instance

The ScheduledService class

- A ScheduledService<V> instance also encapsulates the Task<V> class, makes it reusable, and **allows** an automatic restarting.
- We can configure it to restart a ScheduledService instance whether after it finishes successfully or after it fails.

Example

Run in background



Example

```
+ 1  //Imports are omitted
+ 2  public class FxConcurrentExample extends Application {
+ 3     @Override
+ 4     public void start(Stage primaryStage) {
+ 5        final Task task = new Task<Void>() {
+ 6           @Override
+ 7           protected Void call() throws Exception {
+ 8              int max = 50;
+ 9              for (int i = 1; i <= max; i++) {
+ 10                updateProgress(i, max);
+ 11                updateMessage(String.valueOf(i));
+ 12                Thread.sleep(1000);
+ 13             }
+ 14             return null;
+ 15          }
+ 16       };
+ 17       ProgressBar pBar = new ProgressBar();
+ 18       pBar.setProgress(0);
+ 19       Label lblCount = new Label();
```

In brief

- Extend the Task class or use it, e.g.,
  - `task = new Task<Void>()`
  - `public class PrimeFinderTask extends Task<ObservableList<Long>>`
- Override the call method, e.g.,
  - `@Override protected Void call() throws Exception {`
  - `@Override protected Integer call() throws Exception {`
- Start the thread, e.g.,

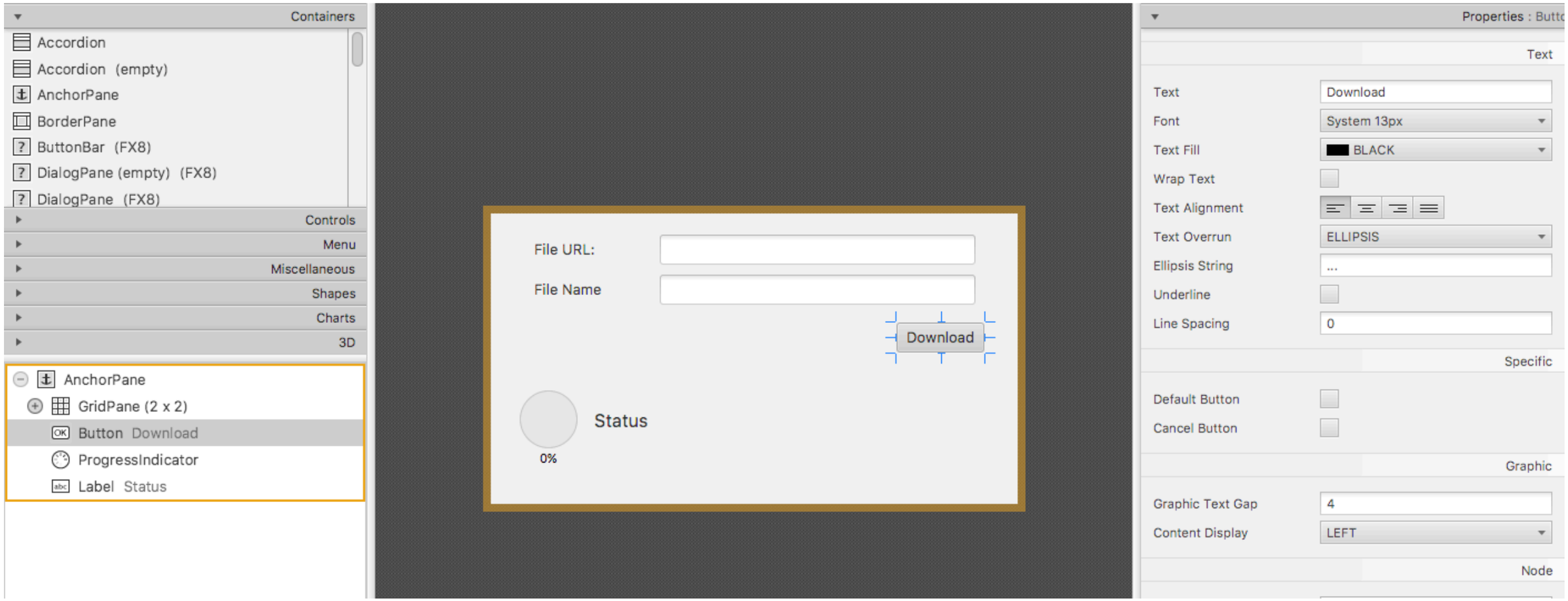
```
btnStart.setOnAction(new EventHandler<ActionEvent>() {

    @Override public void handle(ActionEvent t) {
        new Thread(task).start();
    }
});
```

Result return type

Walking through a real-world example

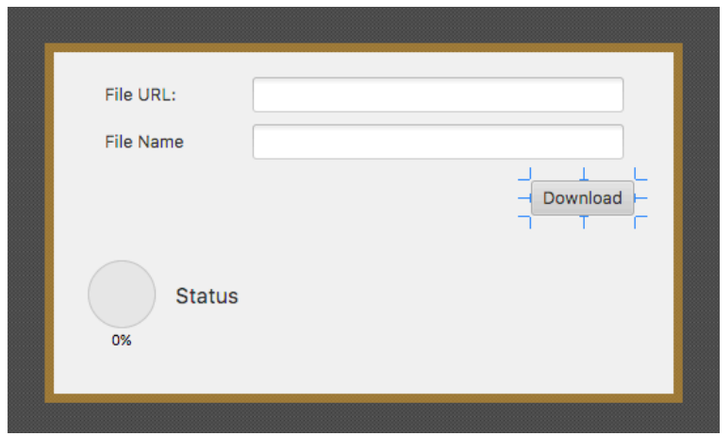
- https://github.com/mhrimaz/JavaFXWorkshop_06
- It is a simple download manager
- The key design concept is to dispatch the download task to a background thread after the download button is pressed. Then, the main process observes the download progress whether the download is complete or not.

fxml


Containers
Accordion
Accordion (empty)
AnchorPane
BorderPane
ButtonBar (FX8)
DialogPane (empty) (FX8)
DialogPane (FX8)
Controls
Menu
Miscellaneous
Shapes
Charts
3D
AnchorPane
GridPane (2 x 2)
Button Download
ProgressIndicator
Label Status
File URL:
File Name
Download
Status
0%
Properties : Butto
Text
Text
Download
Font
System 13px
Text Fill
BLACK
Wrap Text
Text Alignment
Text Overrun
ELLIPSIS
Ellipsis String
...
Underline
Line Spacing
0
Specific
Default Button
Cancel Button
Graphic
Graphic Text Gap
4
Content Display
LEFT
Node

JavaFX Scene Builder

```
+ 1  //Imports are omitted
+ 2  public class Launcher extends Application {
+ 3     @Override
+ 4     public void start(Stage stage) throws Exception {
+ 5        Parent root = FXMLLoader.load(getClass().getResource("../view/MainView.fxml"));
+ 6        Scene scene = new Scene(root);
+ 7        stage.setScene(scene);
+ 8        stage.show();
+ 9     }
+10     static void main(String[] args) { launch(args); }
+11  }
```



fxml — linkage example

```
1   //Imports are omitted
2   <AnchorPane id="AnchorPane" prefHeight="261.0" prefWidth="474.0" xmlns:fx="http://
        javafx.com/fxml/1" xmlns="http://javafx.com/javafx/8.0.65" fx:controller="
        FXMLDocumentController">
3   <AnchorPane id="AnchorPane" prefHeight="261.0" prefWidth="474.0" xmlns="http://
        javafx.com/javafx/8.0.121" xmlns:fx="http://javafx.com/fxml/1" fx:controller="
        controller.MainController">
      ...
19        </GridPane>
20        <Button layoutX="365.0" layoutY="98.0" mnemonicParsing="false" onAction="#
              handleDownloadAction" text="Download" />
21        <ProgressIndicator fx:id="progress" layoutX="25.0" layoutY="159.0" prefHeight=
              "72.0" prefWidth="53.0" progress="0.0" stylesheets="@style.css" />
22        <Label fx:id="status" layoutX="93.0" layoutY="176.0" prefHeight="19.0"
              prefWidth="345.0" text="Status" wrapText="true">
23           <font>
24              <Font name="Segoe UI Bold" size="16.0" />
25           </font>
26        </Label>
```

fxml — linkage example

```
+13    @FXML
+14    private void handleDownloadAction() {
+15       CountDownLatch countDownLatch = new CountDownLatch(2);
+16       try {
+17          executor = Executors.newFixedThreadPool(2);
+18
+19          URL url = new URL(urlField.getText());
+20          String filename = fileField.getText();
+21          URLConnection openConnection = url.openConnection();
+22
+23          totalSizeOfFile = openConnection.getContentLength();
+24          long baseLength = totalSizeOfFile/2;
+25
+26          Downloader downloader1 = new Downloader(url, filename+"-part1", 0,
                baseLength, countDownLatch);
+27          Downloader downloader2 = new Downloader(url, filename+"-part2", (baseLength
                +1),totalSizeOfFile, countDownLatch);
+28          Merger merger = new Merger(filename, 2, countDownLatch);
```

/src/controller/MainController.java

progressProperty

```
+ 26        Downloader downloader1 = new Downloader(url, filename+"-part1", 0,
                baseLength, countDownLatch);
+ 27        Downloader downloader2 = new Downloader(url, filename+"-part2", (baseLength
                +1),totalSizeOfFile, countDownLatch);
+ 28        Merger merger = new Merger(filename, 2, countDownLatch);
+ 29
+ 30        thread1.progressProperty().bind(downloader1.progressProperty());
+ 31        thread2.progressProperty().bind(downloader2.progressProperty());
+ 32        merge_bar.progressProperty().bind(merger.progressProperty());
+ 33
+ 34        executor.submit(downloader1);
+ 35        executor.submit(downloader2);
+ 36        executor.submit(merger);
+ 37        executor.shutdown();
+ 38      } catch (Exception e) {
+ 39        e.printStackTrace();
+ 40      }
+ 41    }
+ 42  }
```

- It will work if what to observed extends the Task<V> class
- Then binding it allows you to observe the progress

progressProperty

`public final ReadOnlyDoubleProperty progressProperty()`

Description copied from interface: `Worker`
Gets the ReadOnlyDoubleProperty representing the progress.

Specified by:

`progressProperty` in interface `Worker<V>`

Returns:

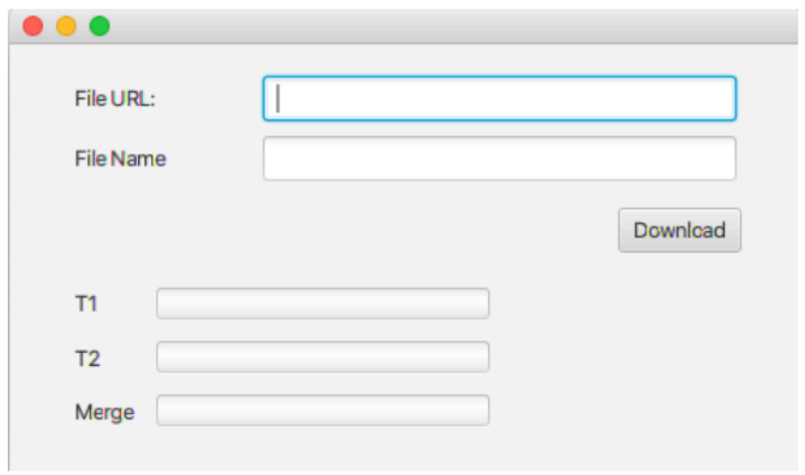
the property representing the progress

See Also:

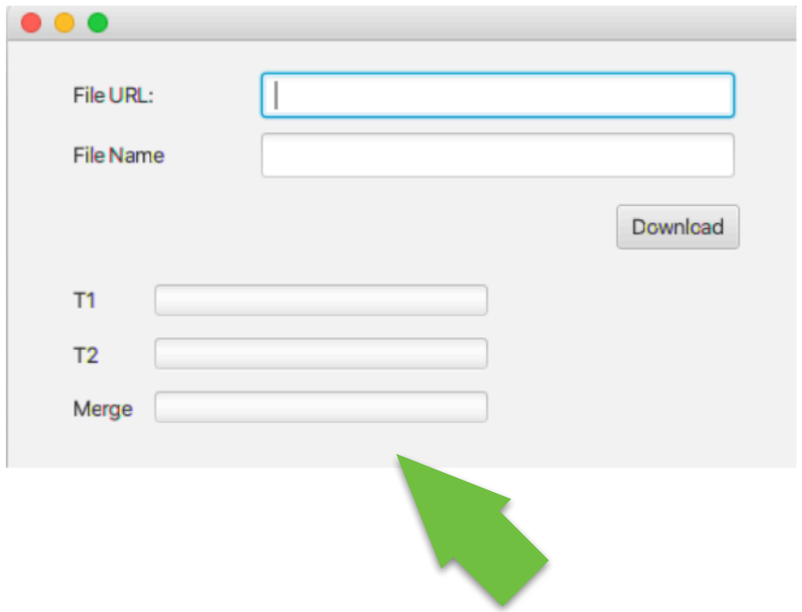
`getProgress()`

Enable multithreaded downloading

- To yield the least synchronization (least communication)
  - Get the file size;
  - Divide the size by the number of downloaders;
  - Create a fixed thread pool and execute all the downloaders;
  - Synchronize the downloader only once when all the downloaders complete their task;
  - Merge all the pieces of downloaded data.



A fixed #Worker in this example



Grid of labels and progress bars

Updating the fxml file

```
+ 38        <children
+ 39            <Label text=    GridPane.rowIndex="0" />
+ 40            <Label text=  2" GridPane.rowIndex="1" />
+ 41            <ProgressBar fx:id="thread1" prefWidth="200.0" progress="0.0" GridPane.
                    columnIndex="1" />
+ 42            <ProgressBar fx:id="thread2" pref idth="200.0" progress="0.0" GridPane.
                    columnIndex="1" GridPane.    ndex="1" />
+ 43            <Label text="Merge" GridPane.    wIndex="2" />
+ 44            <ProgressBar fx:id="merge_bar" prefWidth="200.0" progress="0.0" GridPane
                    .columnIndex="1" GridPane.rowIndex="2" />
+ 45        </children>
```

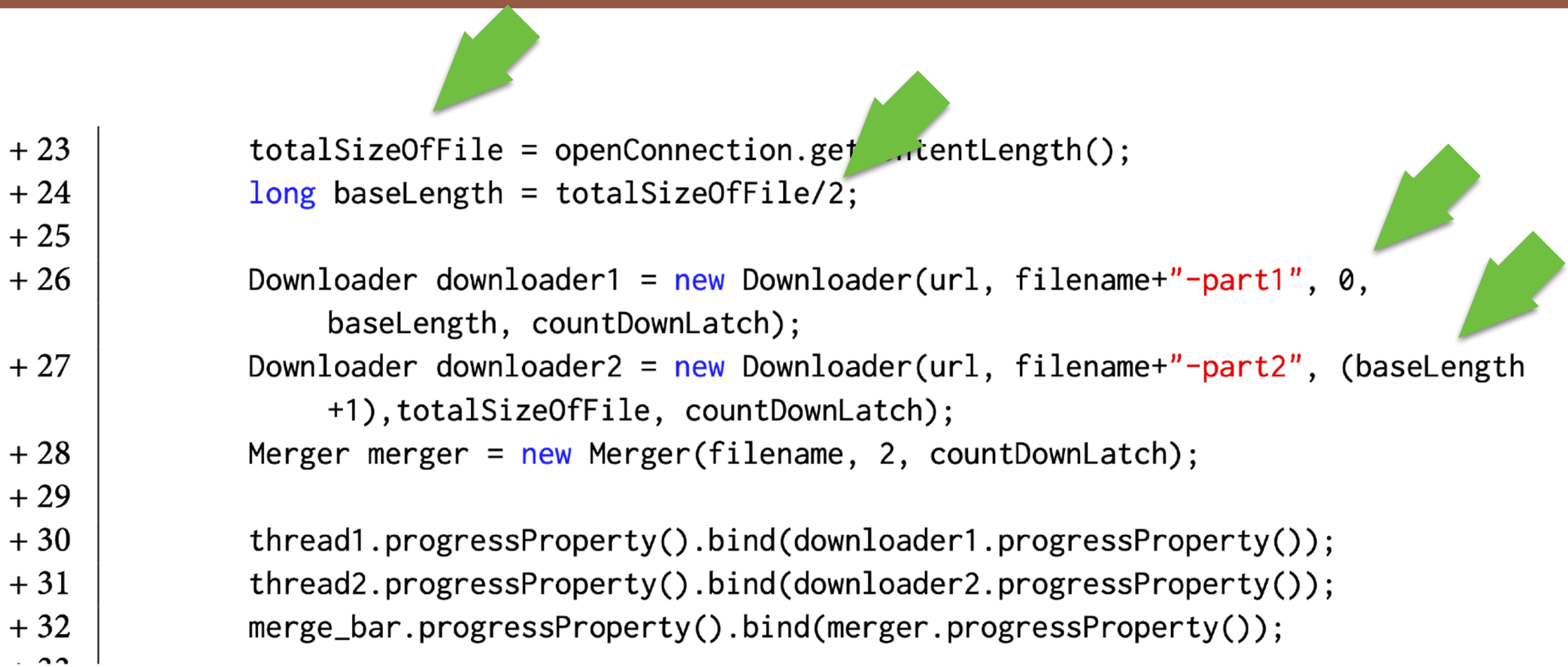
Create the list of fx:id, these will be linked with Java codes later

MainController

```
+ 1  //Imports are omitted
+ 2  public class MainController {
+ 3     @FXML
+ 4     private TextField urlField;
+ 5     @FXML
+ 6     private TextField fileFiel
+ 7     @FXML
+ 8     private ProgressBar thread1,thread2,merge_bar;
+ 9
+ 10    long totalSizeOfFile;
+ 11    private static ExecutorService executor;
```

The created fx:id components in the fxml file.

Dividing the tasks



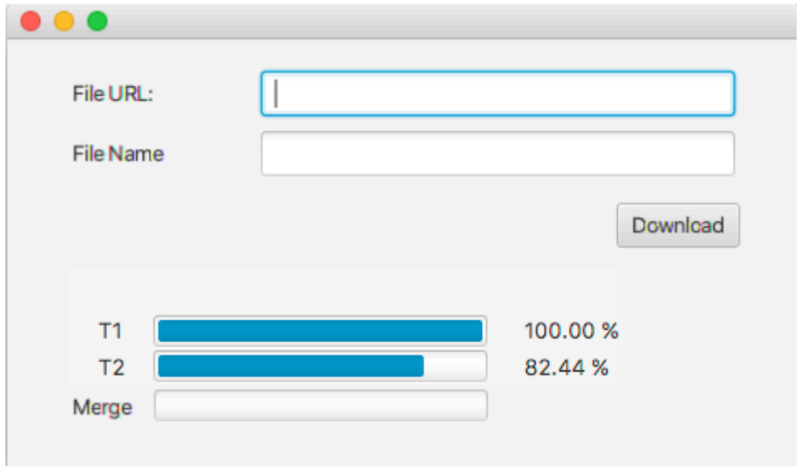
```
+ 23        totalSizeOfFile = openConnection.getContentLength();
+ 24        long baseLength = totalSizeOfFile/2;
+ 25
+ 26        Downloader downloader1 = new Downloader(url, filename+"-part1", 0,
                baseLength, countDownLatch);
+ 27        Downloader downloader2 = new Downloader(url, filename+"-part2", (baseLength
                +1),totalSizeOfFile, countDownLatch);
+ 28        Merger merger = new Merger(filename, 2, countDownLatch);
+ 29
+ 30        thread1.progressProperty().bind(downloader1.progressProperty());
+ 31        thread2.progressProperty().bind(downloader2.progressProperty());
+ 32        merge_bar.progressProperty().bind(merger.progressProperty());
```

Utilizing an (Executor)Service

```
+10    long totalSizeOfFile;
+11    private static ExecutorService executor;
+12
+13    @FXML
+14    private void handleDownloadAction() {
+15       CountDownLatch countDownLatch = new CountDownL     (2);
+16       try {
+17          executor = Executors.newFixedThreadPool(2);
+18
+19          URL url = new URL(urlField.getText());
```

```
+34          executor.submit(downloader1);
+35          executor.submit(downloader2);
+36          executor.submit(merger);
+37          executor.shutdown();
+38       } catch (Exception e) {
+39          e.printStackTrace();
+40       }
+41    }
```

Now we can download the divided file



Next is to merge

- The merge task starts after all the download threads have completed their tasks
- How can the program know it at runtime?

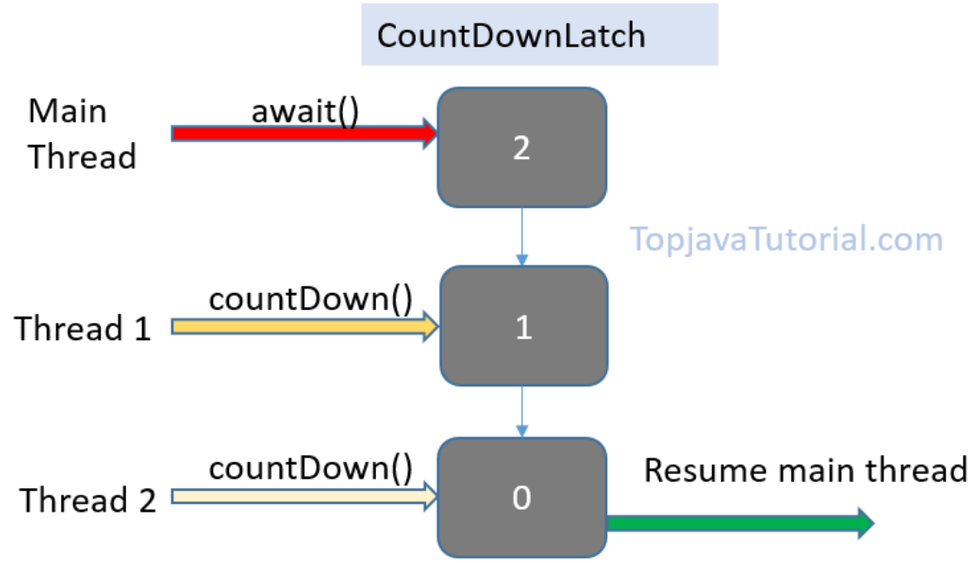
Thread symchornization

- There are many approaches to do synchronization. Different approaches are suitable for different situations

- In this course, let us use one of the simplest approaches namely CountDownLatch

CountDownLatch

- Mimic an electronic latch.
- It waits until the countdown reaches 0 before it continues



CountDownLatch

java.util.concurrent

Class CountDownLatch

java.lang.Object
    java.util.concurrent.CountDownLatch

```
public class CountDownLatch
extends Object
```

A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

A `CountDownLatch` is initialized with a given *count*. The `await` methods block until the current count reaches zero due to invocations of the `countDown()` method, after which all waiting threads are released and any subsequent invocations of `await` return immediately. This is a one-shot phenomenon -- the count cannot be reset. If you need a version that resets the count, consider using a `CyclicBarrier`.

A `CountDownLatch` is a versatile synchronization tool and can be used for a number of purposes. A `CountDownLatch` initialized with a count of one serves as a simple on/off latch, or gate: all threads invoking `await` wait at the gate until it is opened by a thread invoking `countDown()`. A `CountDownLatch` initialized to *N* can be used to make one thread wait until *N* threads have completed some action, or some action has been completed N times.

A useful property of a `CountDownLatch` is that it doesn't require that threads calling `countDown` wait for the count to reach zero before proceeding, it simply prevents any thread from proceeding past an `await` until all threads could pass.

Coding — Declaration — MainController

```
+10    long totalSizeOfFile;
+11    private static ExecutorService executor;
+12
+13    @FXML
+14    private void handleDownloadAction() {
+15       CountDownLatch countDownLatch = new CountDownLatch(2);
+16       try {
+17          executor = Executors.newFixedThreadPool(2);
+18
+19          URL url = new URL(urlField.getText());
+20          String filename = fileField.getText();
+21          URLConnection openConnection = url.openConnection();
+22
+23          totalSizeOfFile = openConnection.getContentLength();
+24          long baseLength = totalSizeOfFile/2;
+25
```

In steps

- Each downloader thread decrement the latch value by 1 after the download is complete.
- When the latch value reaches 0, the merger thread start.

Coding — Decrement the latch value

```
   2  public class Downloader extends Task<Void
          ...
+  5     private long startByte, endByte;
+  6     private CountDownLatch doneSignal;
-  7     public Downloader(URL url, String fileName) {
+  8     public Downloader(URL url, String fileName, long startBit, long endBit,
              CountDownLatch doneSignal) {
           ...
+ 11        this.startBit = startByte;
+ 12        this.endBit = endByte;
+ 13        this.doneSignal = doneSignal;
  14     }
```

```
+ 39           in.close();
  40        }
- 41        byte[] response = out.toByteArray();
- 42        try (FileOutputStream fos = new FileOutputStream(fileName))
- 43           fos.write(response);
- 44     } catch (Exception e) { }
+ 45     doneSignal.countDown();
  46     return null;
  47  }
```

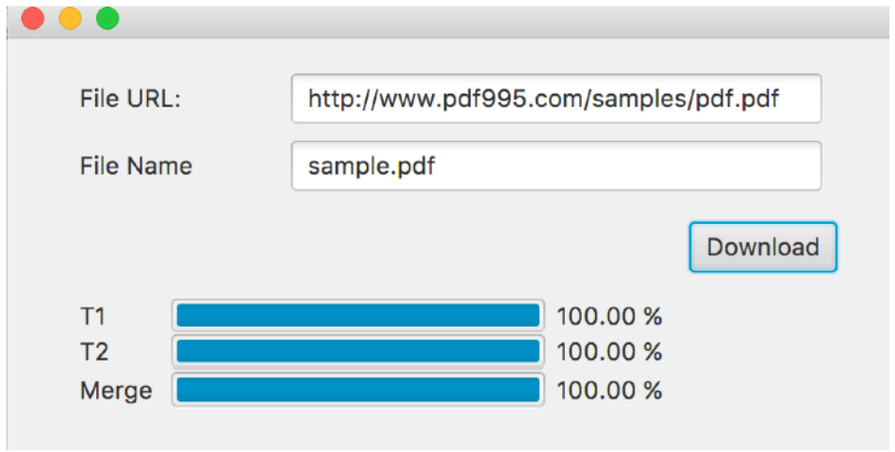
Decrement by one once the task is complete

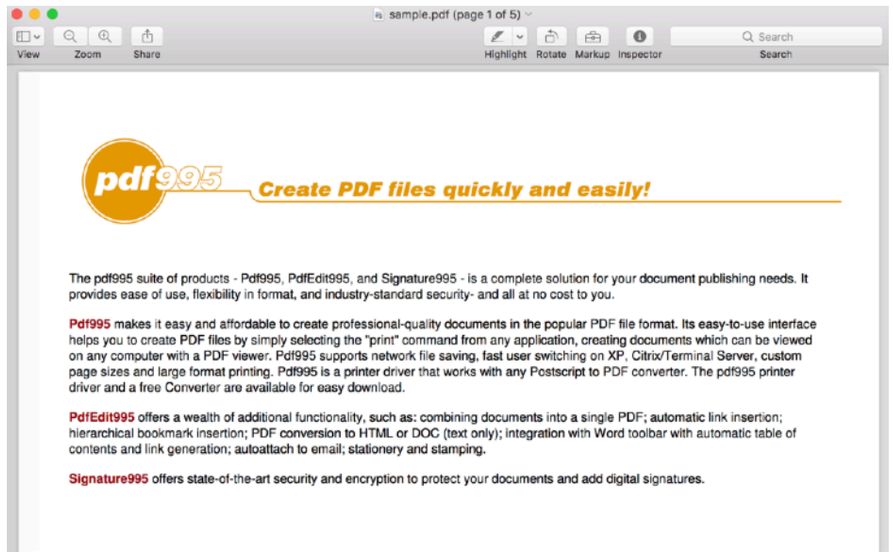
Coding — The merger

```
+ 1  //Imports are omitted
+ 2  public class Merger extends Task<Void> {
+ 3     private String fileName;
+ 4     private int totalParts;
+ 5     private CountDownLatch startSignal;
+ 6     public Merger(String fileName, int totalParts, CountDownLatch startSignal) {
+ 7        this.fileName = fileName;
+ 8        this.totalParts = totalParts;
+ 9        this.startSignal = startSignal;
+10     }
+11     @Override
+12     protected Void call() {
+13        byte[] buf = new byte[4096];
+14        OutputStream out = null;
+15        InputStream in = null;
+16        File f = null;
+17        try {
+18           startSignal.await();
+19           out = new FileOutputStream(this.fileName);
+20           for (int i = 0; i < this.totalParts; i++) {
+21              int toWrite;
+22              f = new File(this.fileName + "-part" + (i+1));
```

Wait until the latch value reaches 0

Now we can download the divided file





Summary

- Multithread and multiprocess are difference
- Multithreaded for GUI programming mainly aims at responsiveness.
- Multithreaded for accelerated computing mainly aims at increasing the computing performance.
- The JavaFX concurrent framework is a toolset providing the necessary resources to support concurrent application development.

Question