Assignment 4

CompSci 230 S2 2015

Clark Thomborson

No submissions will be accepted after 4pm Wednesday 3 June

No lateness penalties will be imposed on submissions after 4pm Friday 29 May

Version 1.01: clarification of experimental units in question 3. Note: if you had already completed your experimental measurements before reading this version you should not make any new measurements, instead you should analyse the data you collected.

Total marks on this assignment: 20. This assignment counts 2% of total marks in COMPSCI 230.

Lateness penalties are significantly relaxed on this assignment:

- -10% of total marks, if you submit prior to 4pm Monday 1 June but after the submission deadline
- -20% of total marks, if you submit prior to 4pm Wednesday 3 June, but after 4pm Monday 1 June.
- Submissions will not be accepted after the final submission deadline of 4pm Wednesday 3 June.

Learning goals. Basic level of understanding of the behaviour of a multi-threaded GUI, with respect to its performance advantages and its implementation defects.

Getting help. You may get assistance from anyone if

- you become confused by Java syntax or semantics,
- you don't understand anything I have written in this assignment handout,
- you have difficulty doing the initial (unassessed) steps in a question, or
- you have any other difficulty "getting started" on this assignment.

Working independently. The homework assignments of this course are intended to help you learn some important aspects of software development in Java, however they are also intended to assess your mastery (or competence) of these aspects. You must construct ***your own answer to every question***.

English grammar and spelling. You will not be marked down for grammatical or spelling errors in your submission. However if your meaning is not readily apparent to the marker, then you will lose some marks.

Submissions. You must submit electronically, using the Assignment Drop Box (https://adb.auckland.ac.nz/). Your submission must have **one document** (in PDF, docx, doc, or odt format) with your written answers to the questions in this assignment. You should not submit any jarfiles.

Time budgeting. I'd suggest you spend 4 to 5 hours on this assignment, so that you still have a significant amount of time (within a 10 hour/week budget for this course) for your independent course-related study, and for your lecture and tutorial attendance.

ADB response time. The ADB system is sometimes unresponsive, when it is under heavy load due to an impending stage-1 assignment deadline. I will extend your submission deadline by up to one hour, if I see any evidence (from the submission logfiles) of ADB overloading. So: please be patient if the ADB seems slow to respond.

Resubmissions. You may resubmit at any time prior to the final deadline of 4pm Wednesday 3 June 2015. However your markers will see only your last submission.

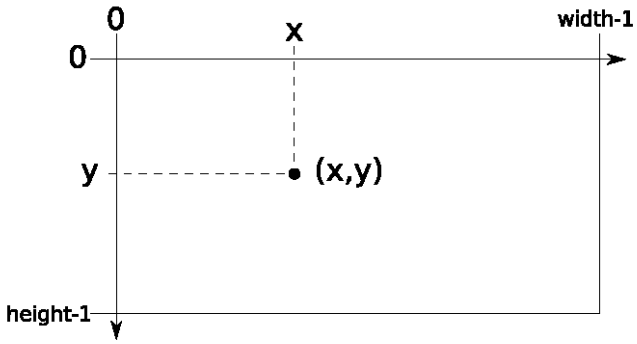
Part 1: Reporting a Responsiveness Defect

1) **(8 marks)** *Initial steps (unassessed):* Import `a4.jar` into your development environment in a new project which I suggest you call A4v0. Compile and run it, in order to confirm your import, and also to discover how to "drive" its user-interface. Note: `a4.jar` is being released to you under GPL v3 licensing, because it was derived from Tim Vaughan's Mandelscape.

Also import the release version of Mandelscape, either from its zipfile or (if you're feeling adventurous and have a few more hours to devote to these initial steps) you could take this opportunity to learn about Git by signing up for a free account at GitHub, and working through its Bootcamp lessons or Vogella's tutorial.

Read the following background information about Mandelbrot sets. Note: this information is reproduced from Tim Vaughan's assignment handout in a previous offering of CompSci 230.

> This assignment concerns a program that displays a visualization of a beautiful mathematical object known as the Mandelbrot set. The Wikipedia page at http://en.wikipedia.org/wiki/Mandelbrot_set contains a lot of interesting and useful information regarding this set and its place in mathematics. I'd encourage you to read this especially if you've never encountered the Mandelbrot set before. However, you need to know very little about the set to successfully complete this assignment. What you do need to know is detailed here. A visualization of the Mandelbrot set involves a rectangular array of pixels:



> These pixels are mapped onto an equivalent array of points in a two-dimensional space known as the complex plane, which is where the Mandelbrot set lives. This mapping is controlled by the coordinates in this plane at the edge of our pixel grid. For instance, the top-left pixel (0, 0) maps to the complex coordinate (crMin, ciMin), while the bottom-right pixel (width-1, height-1) maps to the complex coordinate (crMax, ciMax). The values of crMin, crMax, ciMin and ciMax therefore determine which portion of the Mandelbrot set is visible in our application window.
>
> In order to display an image, a visualizer needs to perform the following for every point (x, y) in the pixel array:
>
> 1. Determine the corresponding complex coordinate (cr, ci).
> 2. Evaluate a simple but *computationally intensive* algorithm that returns an integer as its result.
> 3. Assign a colour to the pixel depending on the value of the integer returned.
>
> The computationally intensive algorithm is parameterized by another integer: the maximum number of "iterations". This can be regarded as the accuracy with which the computation is done, but also directly affects the worst-case time complexity of the algorithm; higher maximum iteration values will give better-looking results (particularly at higher "magnification" levels) but will take longer to compute.

Assessed work:

Compare the performance of A4v0 with Mandelscape, with respect to GUI responsiveness. You should test A4v0 in two modes. A4v0.0 is A4v0 being run with 0 worker. A4v0.1 is A4v0 with 1 worker.

If you notice a GUI-responsiveness defect, you should attempt to "reproduce" this defect, that is, you should attempt to find a reliable way to cause the application to exhibit this defect. For example, the spinner labelled "Max iter." may be unresponsive in either or both of these applications, and this unresponsiveness may be evident only in A4v0.1 but not in A4v0.0 (or vice versa). Note that a description of "what can go wrong" in a defective application doesn't reveal the conditions under which the defect can be reliably observed it – and developing a reasonably-concise but easily-understandable description of "how to observe" a responsiveness defect should be your primary focus when you are answering this question.

You may run across one or more correctness defects in this code. In particular, a SwingWorker may throw an uncaught exception which has the effect of killing off that worker but not the application, and the uncompleted task may result in visible defects in the display. Another known defect (which is also present in Tim's codebase) is that multiple mouse-clicks in rapid succession may not have the (desired and expected) effect of causing multiple zooms, but instead may have the effect of resetting the view to the default zoom. Please do your best to "work around" any correctness defects you observe. I will issue a new version of a4.jar only if it has fatal defects in correctness, and (after running it on a fast desktop and a slow laptop) I'm reasonably confident it has no fatal defects.

Rather than getting side-tracked into correctness defects, when answering this question you should focus your attention on responsiveness defects. Users generally expect the controls (= widgets or affordances) on their GUIs to always react, immediately, to their mouse-clicks and mouse-drags. In particular, it should always be possible for a user to "spin" a spinner, or to click a button – so you should concentrate your testing on the responsiveness of the three spinners and one button on the GUI of a4v0.

Submit: a paragraph discussing *one* responsiveness defect you discover in *either* version of the Mandelscape application. Your paragraph should describe a specific defect (2 marks), and it should also describe a reliable method for reproducing this defect in each of the three systems under test (Mandelscape, a4v0.0, a4v0.1; for 2 marks each). If the defect is not reproducible in a system under test, you should state this clearly.

Part 2: Thread Performance

2) **(6 marks)** *Initial steps:* Examine the console output of a4v0, as this application responds to a few mouseclicks, to gain some first-hand experience with the console view of my "instrumentation" of Tim Vaughan's Mandelscape code. You have probably already noticed my addition of progress bars: these are intended to give you a visual indication of what each `SwingWorker` is doing at any point in time. You should have already experimented with the "Number of workers" spinner, and I hope you noticed that this control affects the number of progress bars being displayed. Visual indicators of performance can be very helpful in gaining informal understanding, however they are ephemeral i.e. they don't leave a written record which can be analysed and discussed.

   On the console output stream, you'll find some analysable information about the performance of a4v0. In particular, whenever a `SwingWorker` has completed its task, a one-line summary is printed to the console. For example:

```
Task number 1 completed on update number 4 with latency 282.  Work rate = 2.84 mega-iters per second.
```

   I'd interpret this task-completion summary as follows: during the fourth update of the screen, 282 milliseconds elapsed between the first execution of a `SwingWorker`'s task and its completion of this task. (Each SwingWorker gets exactly one task during each update.) During these 282 milliseconds, this `SwingWorker` had completed slightly more than 800,000 iterations of its inner loop (as described in the last paragraph of Tim's description on the previous page). My instrumentation reports all "useful work" not as a count of iterations, but

instead as an average rate of iteration. This conversion is simple arithmetic, and it results in a measurement of performance where "higher is better":

(800,000 iterations)/(0.282 seconds) = 2.84 mega-iterations per second.

Note: unless you're running this application on a rather slow laptop, you should see much higher values for "mega-iterations per second", and much lower values for "latency", than in my example above.

In the console output, you'll also find one-line reports on task cancellations. SwingWorker tasks are cancelled if they have become irrelevant to the current model. Such "useless work" should be terminated as soon as possible. For example, if two or more "zoom" or "pan" requests made before the model has been updated to reflect the first such request, then only the tasks associated with the latest request describe some "useful work". Any work done by a `SwingWorker` on a cancelled tasks does *not* count toward the "mega-iters per second" metric of performance that is reported to the console.

In the console output, you'll also find one-line reports on updates. An update is any calculation (or recalculation) required to repaint the display. If there are sixteen workers, then there are sixteen tasks: each task does the computation required to repaint a vertical stripe that is one-sixteenth of the width of the display. You'd see these repaints occur in some arbitrary order -- unless (and this seems unlikely!) your CPU supports sixteen threads of sufficiently high performance that the repaint is near-immediate. If you are running on a very high-performance CPU, you can dial-up the difficulty of the computation by increasing the "Max iters" value.

A summary report is printed when an update is completed (and is being deleted from the list of pending updates). This report indicates the latency (elapsed time, in milliseconds) between this invocation of update() and its completion; and it also indicates the rate at which useful work was completed (summing over all workers) during this period. For example, if an update requires 1,600,000 iterations and is completed with a latency of 1000 milliseconds, then it was accomplished at a rate of 1.6 mega-iters per second.

You should look at the code in a4v0, if you require any additional information about "what its console output means". Most of this console output is generated by methods in the `MandelModel` class. Note: after you locate one `println()` invocation, you can find all of the other invocations of this method easily in Eclipse by selecting the method name, right-clicking to bring up a context menu, and invoking the References/Project affordance.

Assessed work:

Discover an (approximate value for) the optimal number of workers for a4v0 on your platform, by determining the lowest value of "Number of workers" that reliably delivers near-optimal "mega-iterations per second" on the updates. Note that allocating slightly more than the optimal number of workers will not significantly affect the time required for an update, and allocating many more than the optimal number of workers will increase the overheads of task-formation and task-cancellation – possibly to the point of adversely affecting responsiveness.

To estimate the optimal number of workers, you should select a display size and "Max iter" value which causes your platform to spend about 4 seconds (= 4000 milliseconds) when computing an update. You should then adjust the Number of workers from 1 to 16, then down from 16 to 1, then from 1 up to 16 again, slowly enough to ensure that each update is completed before the next update is requested. Your console listing will be quite long – you should cut-and-paste it into a word-processing document, retaining the entire listing as a single document (for reference). Then you should edit-down the listing until it consists solely of reports on update-deletions of the following form: "Update number X deleted. There are 0 pending updates. Latency Y. Work rate = Z mega-iters per second." Note that if there are any pending updates in the completion reports on your experimental trace (for X = 1, 2, 3, … 15, 16, 15, 14, 13, …, 2, 1, 2, …, 15 16) then the reported work-rates are unreliable – because some `SwingWorker`s are doing useless work, thereby consuming CPU resources that are unavailable to the usefully-working `SwingWorker`s. Also note that any report of a 0 work-rate is referring to an

update which had been cancelled. If your first attempt does not provide you with a complete experimental trace (due to some pending updates), you should collect a second experimental trace, adjusting the number of workers then waiting until the update is completed (as indicated in its console report) before making another adjustment to the number of workers. If your second attempt fails, you will have reproduced a serious correctness error in a4v0. If you have discovered a reproducible correctness defect, you should document it briefly, then you should construct a (partial) experimental trace which contains deletion records of updates which contain performance records of updates which weren't cancelled, weren't running concurrently with pending updates, and which didn't immediately follow an update that completed while another update was pending.

Produce two scatterplots from the (X, Y, Z) triples in your experimental trace. Your (X, Y) plot will indicate how update-completion latency varies as a function of the number of workers. Your (X, Z) plot will indicate how the max-iter performance varies as a number of workers. Consider what these plots tell you about the optimal number of workers (for this particular view of a Mandelbrot set, on your particular platform). Now shift your viewpoint on the Mandelbrot set, and choose a somewhat smaller or larger "Max iter" value, selecting the optimal number of workers, to get a rough indication of whether or not the "mega-iters per second" performance of your platform (when running with the optimal number of workers) is reasonably constant over viewpoint settings. (This is called a "sensitivity analysis" – you're discovering whether your finding is "sensitive" to parameters you weren't directly testing. In this case your experimental trace varied only the number of workers, but there are many other parameters which *could conceivably* have a significant effect on performance.)

Submit: your two scatterplots (1 mark each), accompanied by your discussion (4 marks). Your discussion should briefly describe any experimental difficulties (such as a correctness defect), and it should focus on *your* interpretation of your experimental findings regarding the optimal number of workers.

Part 3: Injecting a Defect

3) **(6 marks)** *Initial steps:* Modify the `cancelWork()` method of `MandelUpdate` in `a4v0`, in a new project called `a4v1`. Your modified `cancelWork()` method should read as follows:

```
void cancelWork() {
      isCancelled = true;
}
```

This change has the effect of defeating the cancellation feature in a4v0 – because it no longer sends cancellation requests to its workers. Please be aware that the removal of the cancellation feature in your `a4v1` is likely to introduce some correctness defects, in addition to the (I hope obvious) possibility that it will negatively impact the responsiveness and performance of this version. Indeed, `a4v1` has one fairly obvious defect, in that its console output contains incorrect reports about cancellation requests being sent to all workers. Unsurprisingly, my instrumentation does not detect this defect – after all, its purpose is to instrument the performance of this application, not to perform correctness tests on it! You should not bother to repair this correctness defect. However I do hope this example helps you remember, whenever you are working with an instrumented program such as `a4v1`, that any modifications to its code may invalidate some (or all!) of its reports.

Assessed work: determine whether or not the responsiveness defect you identified in Part 1 is present in `a4v1.0` (= your `a4v1` with no workers), and whether it is present in `a4v1`.1 (= your `a4v1` with 1 worker).

Also determine whether or not your `a4v1` has similar performance to a4v0 under two runtime scenarios. In scenario r1, you collect an experimental trace as in Part 2 – although you need only test four cases: `a4v1r1.1`

(in which you request an update in a session on `a4v1`, with no pending updates and 1 worker), `a4v1r1.5`, `a4v1r1.11`, and `a4v1r1.16`. In scenario r2, you should request a series of three updates by clicking three times in rapid succession. The experimental units in scenario r2 should be `a4v0r2.1`, `a4v0r2.5`, `a4v0r2.11`, `a4v0r2.16`, `a4v1r2.1`, `a4v1r2.5`, `a4v1r2.11`, `a4v1r2.16`. This 2x2x4 experimental design (= {a4v0, a4v1} × {r1, r2} × {1, 5, 11, 16}) will allow you to compare the performance of `a4v0` and `a4v1` under the rapid-click scenario r2 (which you hadn't tested in Part 2 of this assignment). You should measure the latency and performance (in mega-iters per second) of the application on the last of the three requested updates. The paucity of data (only 16 performance measurements, with no repetitions) in this experimental design means that your findings will be only indicative – certainly not definitive, as there are many factors you are not controlling and at least one of these factors must be important (because, as I suspect you have noticed already, the performance measurements are rather noisy).

Submit: A paragraph discussing the presence or absence of your part-1 responsiveness defect in `a4v1` (3 marks), and a paragraph discussing your performance findings on `a4v1` (2 marks) which refers (in some relevant and clear way) to a table or plot of your performance measurements (1 marks).