Goals Practice with concurrency in a Java GUI application.

Preparation

Download links and other details can be found via the course web page:

`http://www.win.tue.nl/~wstomv/edu/sc-hse/`

1. In the book *Introduction to Programming Using Java* by David Eck, read §12.1, 12.2.(1–3).
2. Review the slides of Lecture 12 (two slide sets).

Assignments

1. (*Iterable IntRelation (2)*) Do the assignment *Iterable IntRelation (2)*, which is an improved version of *Iterable IntRelation*. Note that the handout on *Test-Driven Development* and the code given in `TDDExamples.zip` have been updated (the data representation now has **`protected`** instead of **`private`** visibility; see Downloads for Series 4).

   Furthermore, some skeleton code has been given for writing your own test cases: `TDDForIterableIntRelationSkeleton.zip` (see Downloads for Series 7). Finally, the description in peach[3] has been improved.

2. Inspect your submission for *Undo-Redo facility* of Series 11, taking the feedback on Slides 3 and 4 of Lecture 12 into account.

3. Download `SwingWorkerDemo` (made with NetBeans). Inspect the code. Compile and run it. This illustrates

   - how to use a `SwingWorker` to run a calculation in the background;
   - how to obtain a final result.

   This involves `execute()`, `doInBackground()`, `done()` and `get()`.

4. Download `SwingWorkerWithPublish` (made with NetBeans). Inspect the code. Compile and run it.

   This will prepare you for adding a `SwingWorker` to the *Simple Kakuro Helper 2*. In particular, it illustrates

   - how to make a background calculation interruptable, using `cancel()`;
   - how to publish intermediate results for processing by the GUI thread, using `publish()` and `process()`.

5. (*Simple Kakuro Helper 3*) We want to run the generator of the *Simple Kakuro Helper 2* in a background thread, and (optionally) make it interruptable. We want to reuse as much of the code as possible without changing it. Adapt your code as follows.

   (a) In `KakuroCombinationGenerator`, change the visibility of the instance variables `maxNumber`, `observer`, and `object` from **`private`** to **`protected`**. That way, they can be accessed directly in subclasses.

   (b) Add a subclass `KCGeneratorProxy` that **`extends`** the (modified) class `KakuroCombinationGenerator`. It will serve as a *proxy*[1] for a Kakuro combination generator running in the background. Also see the given skeleton code.

   An instance of `KCGeneratorProxy` can be used as if it is a regular `KakuroCombinationGenerator`, but its `generate(s, n)` is overridden to create a special instance of a `SwingWorker` that will run a "real" `KakuroCombinationGenerator.generate(s, n)` in another thread. The proxy can be observed, and its observer will receive values generated by the "real" generator.

   There are two ways to handle the observer. Here, both ways can work, but in general you need to analyze the situation carefully.

      i. The proxy registers its observer directly with the "real" generator.

      ii. The proxy registers a special observer with the "real" generator. That observer retrieves the generated object using `getObject()` on the "real" generator, and publishes it to the proxy thread using `publish()`. The proxy in turn processes each published value, by storing it in its instance variable `object` and notifying its observer. That observer can then retrieve the value from the proxy through `getObject()`, as usual.

   This way, all the infrastructure for observing a generator can be reused "as is" on the proxy.

   Because the proxy's `generate()` returns asynchronously (without waiting for the generation process to complete), the client (like a GUI) must have a way of finding out that generation is done, so that it can do its post-processing (like obtaining a total count from an observer that counts). Provide this way through a special result listener (a callback; the given skeleton code already contains the relevant definitions).

   (c) In `MainFrame`, do the following.

   - Let the initialization code create *two* main frames, so that you can experience the concurrency.

[1] See, for instance, `http://en.wikipedia.org/wiki/Proxy_pattern`.

- Add a text field labeled Max, to let the user enter the maximum number allowed in a Kakuro combination. Use `setMaxNumber` to set this value on the (proxy) generator.
- Handle a Generate button click, by setting up and calling a generator proxy, instead of a "real" generator. The biggest difference is that you need to set a result listener on the proxy, to show the final values of the counter, intersector, and eliminator. Note that these observers either need to be made **`final`** inside the button handler (otherwise, the result listener cannot access them), or they should be moved out of the button handler and made into instance variables of `MainFrame`.
- To avoid complications, disable the Generate button while the generator runs, and enable it again when the generator is done;
- Optionally, add an Abort button, which is enabled if, and only if, the generator runs. So, initially it should be disabled. When clicked, it aborts the proxy. Note that the result listener will still be called.

(d) (Optional) The "real" generator must be modified slightly, to become interruptable.

You can test with maximum number 30, sum 433, and length 28. There are 14 combinations, having `[1, 16]` as intersection. To generate these takes a while, if you do not do *Branch & Bound*.

Note that the two windows can be active at the same time.

Submit your work to peach[3].

6. (Optional: *Kakuro Puzzle Assistant Continued*)

   (a) Make your *Kakuro Puzzle Assistant* run its solver in a background thread.

   (b) Make the solver interruptable.

Deadlines: See peach[3]

