Topics in Parallel and Distributed Computing: Introducing Algorithms, Programming, and Performance within Undergraduate Curricula*†‡

Chapter 10 – Parallel programming for interactive GUI applications

Nasser Giacaman[1] and Oliver Sinnen[2]

[1, 2]Parallel and Reconfigurable Computing Lab, Department of Electrical and Computer Engineering, The University of Auckland, New Zealand ,

n.giacaman@auckland.ac.nz, o.sinnen@auckland.ac.nz

*How to cite this book: Prasad, Gupta, Rosenberg, Sussman, and Weems. Topics in Parallel and Distributed Computing: Enhancing the Undergraduate Curriculum: Performance, Concurrency, and Programming on Modern Platforms, Springer International Publishing, 2018, ISBN : 978-3-319-93108-1, Pages: 337.

†Free preprint version of this book: `https://grid.cs.gsu.edu/~tcpp/curriculum/?q=cder_book_2`

‡Free preprint version of volume one: `https://grid.cs.gsu.edu/~tcpp/curriculum/?q=cedr_book`

Abstract

This chapter will help you understand the rules that you must adhere to when developing a concurrent application with a graphical user interface (GUI). Regardless of the technology you use (for example, developing an Android mobile app or a desktop application using the .NET Framework), the concepts presented here are standard for the GUI toolkits you will use. The most important aspect includes ensuring the application does not freeze or become unresponsive, by employing background threads. This in turn leads to the other important consideration, which relates to ensuring access to any GUI components does not introduce potential race conditions. Collectively, the concepts presented here relate to the single-thread rule that governs almost all GUI toolkits you will likely come across.

Relevant core courses: GUI Concurrency is a topic suitable for any CS2-equivalent course. The material covered in this chapter would be typically covered in around 3-4 hours of class time (about a week's worth of lectures). Rather than focusing on parallel programming, the focus is on thread-safety issues pertaining to GUI applications (and does not include general introductory threading). The topic is also suitable for any course that incorporates GUI development, as well as PDC courses at any level.

Relevant PDC topics: There is no specific categorization for GUI concurrency in the NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing [1]. However, the topic presented here is essential for CS2 courses that involve a GUI module. More specifically, this topic is vital for any software developer creating *interactive GUI applications*, and not just for the parallel programming enthusiast. Since GUI concurrency has much of its essence based on standard parallelization/concurrency concepts, we will be touching on the following subtopics – but directly in the context of GUI applications. According to Bloom's classification, students are expected to *Apply* these subtopics:

- Programming
  - Parallel programming paradigms (shared memory, task/thread spawning).
  - Semantics and correctness issues.

Learning outcomes:

- Students will understand the importance of concurrency and apply it in the context of GUI applications.
- Students will be able to discuss the main issues associated with GUI applications. This includes two fundamental primary themes:
  - Maintaining a responsive GUI application by introducing concurrency for event handlers with human-perceived delays (i.e. allowing the GUI-thread/EDT/UI-thread to promptly return to the event loop to avoid the backlog of events).
  - Ensuring thread-safety of GUI components (i.e. background threads must not access GUI components, only the GUI-thread/EDT/UI-thread may do this).
- In addition to the essential correctness themes above, students will also be able to supply intermittent updates (from the background threads) to the GUI-thread to support improved user-perceived performance (e.g. regular updates to a progress bar).
- Students will be able to understand the different ways in which a GUI application becomes unresponsive, and apply the correct techniques to overcome this:
  - The GUI-thread is never to invoke blocking functions, even when waiting for asynchronous tasks.
  - The GUI-thread is never to process any events that involve perceived delays.

Context for use: It is intended that this chapter be used directly by students to help them understand the underlying concepts of GUI concurrency. Rather than just declare the

rules of GUI concurrency, it is important to explain the bigger picture why those rules are in place. The analogy presented has been used in lectures for the above courses with positive feedback from students.

10.1 Essential concurrency definitions

Before we start discussing GUI concurrency, we will briefly mention the most relevant definitions. Most of these you would have come across already. A **thread** is a programming entity that allows a stream of instructions to be executed independent of (and at the same time as) other instructions. A **task** (or more specifically a **Runnable** in Java) is a packaged entity of code to be executed by a thread. **Locks** are a protection mechanism that ensures only one thread executes a piece of code at any one point in time. As this chapter progresses, more definitions (especially in the context of GUI applications) are introduced. See section 10.4 for a summary of these concepts.

10.2 The cash balance problem

To help us understand concurrency in a graphical user interface (GUI) application, we are going to develop some storylines to explain it in non-technical terms. The first storyline introduced in this section is a rather classical example that helps explain the major problem that concurrency introduces in general. As simple as it may seem, this problem is the fundamental issue underpinning GUI concurrency, so it is important we have a clear appreciation of the inherent problem.

Figure 10.1 illustrates a company's policy in maintaining the cash balance by using a book. The policy includes three primitive steps to be followed whenever an employee needs to update the balance. First, the employee must observe the balance on the open page of the book (for simplicity, we assume only one balance is written on each page of the book). Once the employee has taken a mental note of the current balance, the employee momentarily performs a simple calculation on their own calculator. With this new balance in mind, the employee returns to the book, flips the page (without taking notice of what page was open)

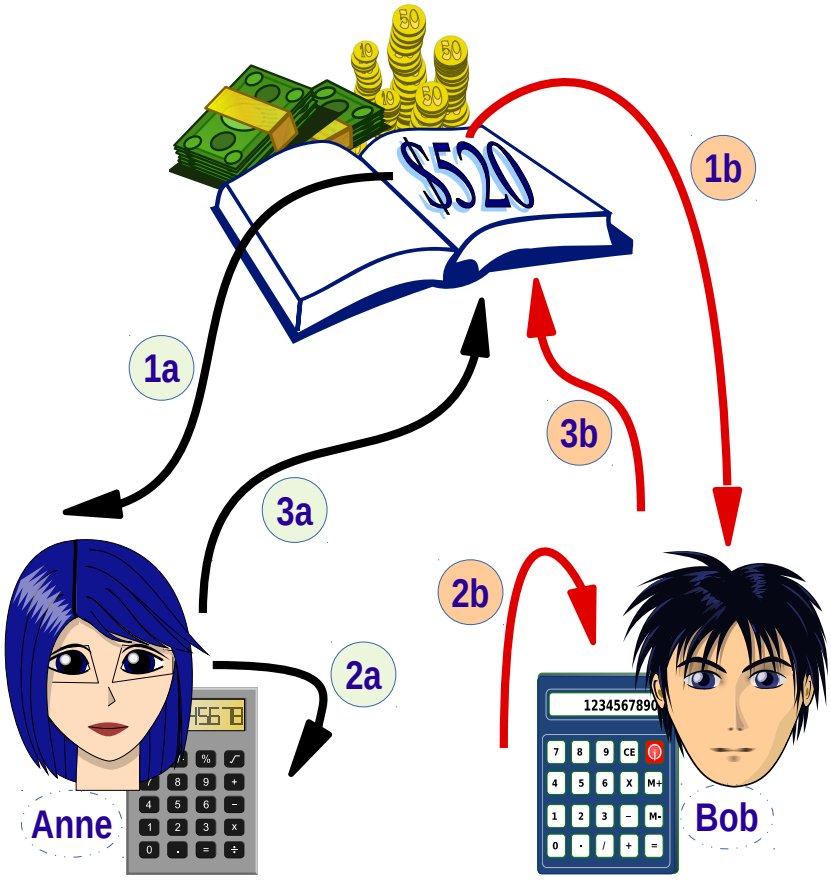


Figure (10.1) In this company, a record of the cash balance is maintained in a book. The company policy to maintain the book balance is rather primitive, and involves three simple steps that each employee follows. (1) An employee observes the balance on the currently open page in the book. (2) The employee turns away and calculates the new balance using a calculator. (3) The employee returns to the book, flips the page (without noticing if it had changed since being observed in Step 1 and writes the new balance on the next page.

and writes the new balance on the next empty page. This page becomes the new balance.

While it may seem like a rather straightforward and harmless set of steps, the obvious situation is when an employee performs steps 1-3 at the same time as another employee. For example, if Anne and Bob both observe the book balance (1a and 1b respectively), then they both enter $520 into their calculators. Anne adds 20 to her calculator (2a), which then reports 540. In the meantime, Bob is adding 30 to his calculator (2b), which then reports 550. Anne deposits the $20 into the pile of cash, flips the page and writes $540 on a new page (3a). Just as she finishes, Bob is also depositing his $30 into the pile of cash, flips the page and writes $550 on a new page. The book balance is inconsistent with the actual amount of cash, which is actually $570! Neither Anne or Bob is to blame – they were simply following company procedure!

As naive as the three steps seem in updating something as simple as the cash balance, these are the exact same steps involved in updating an integer in our program! A statement as simple as "`count++;`" expands to three instructions that the processor must execute:

1. Read the value from memory and into a register (a small amount of fast storage located on the processor), much in the same way Anne glanced the value in the book and recorded it into her calculator.

2. Perform the increment in the register, much in the same way Anne performed the addition on the calculator.

3. Write the result from the register back to memory (with disregard to the current value in memory), much in the same way Anne copied the result from her calculator onto a new page in the book (with disregard to the current value written in the book).

We can even simulate the cash balance problem in a simple program*:

```
int currentBookBalance = 520;
```

*The first code example (cache balance) is included in the Appendix, and all the example codes are downloadable from `http://parallel.auckland.ac.nz/files/gui-chapter-examples.zip`

```
...
// Anne
int observedAmount = currentBookBalance;     // 1a
blink();
int calculatedAmount = observedAmount + 20;  // 2a
blink();
currentBookBalance = calculatedAmount;       // 3a
...
// Bob
int observedAmount = currentBookBalance;     // 1b
blink();
int calculatedAmount = observedAmount + 30;  // 2b
blink();
currentBookBalance = calculatedAmount;       // 3b
...
System.out.println("Final balance = $" + currentBookBalance);
```

When we run this program (`eg01.CashBalanceProblem.java`), we experience what is known as a **race condition** (a programming bug where the output depends on the uncontrollable timing and intertwining of the steps since multiple threads are writing to the same memory location). In our example, we notice that sometimes the final result is $540, while at other times it is $550. It is never the expected $570. The `blink()` function is rather an exaggeration to help illustrate the point by forcing the intertwining of the three steps between Anne and Bob by introducing a time delay between steps 1, 2 and 3).

10.3 Solving the cash balance problem – without locks?

You probably anticipated this section to solve the cash balance problem by protecting access to the cash balance using fancy concepts such as mutual exclusion and locks, right?

Well, sorry to disappoint you, but we're not going to do that here[†]. *If* we were going to take this approach, then we would be talking about how we put the cash balance inside a room that has a lock on the inside of the door. The rule is that only one person is allowed inside the room, in which case they have full access to the books while everyone else waits outside. When either of Anne or Bob wants to deposit money into the cash balance, they enter the room, perform the three steps, then exit to allow another person to perform the steps.

While using locks seems like a reasonable solution to avoid corrupting the cash balance, the complexity of managing this approach quickly escalates as we introduce more and more items that need protecting. Imagine we have multiple account books, that somewhat relate to each other. We would need to protect each and every one of these books in the same manner. If each book was placed in a separate room with its own lock, how do we ensure we do not deadlock as Anne accesses a book then also wants another book locked by Bob (who in turn wants the book already locked by Anne)?

Devising a set of policies to manage all these books in a correct (let alone efficient) manner is very complicated. So, instead of allowing all the employees to have access to the books, we say that *none* of them is allowed direct access to the books! What we do is employ a new person, Gemma, to be solely responsible for any direct access to the books, including our original cash balance. If Gemma is the only employee that accesses the cash balance, then this will naturally ensure the balance remains correct at all times. Figure 10.2 illustrates the new policy in place, where the same three steps to modify the cash balance exist, only this time it is always performed by Gemma. We can also see how Gemma is responsible for the other books.

So, what about Anne and Bob when they want to modify the cash balance? It would not be such a good idea if they directly talked to Gemma, since she might be busy performing some other tasks. Instead, it would make more sense if they wrote their request on a memo and placed that memo in the pile next to Gemma. When she gets a chance,

[†]This isn't to say that locks cannot be used, but rather that we are going to solve this scenario without locks.

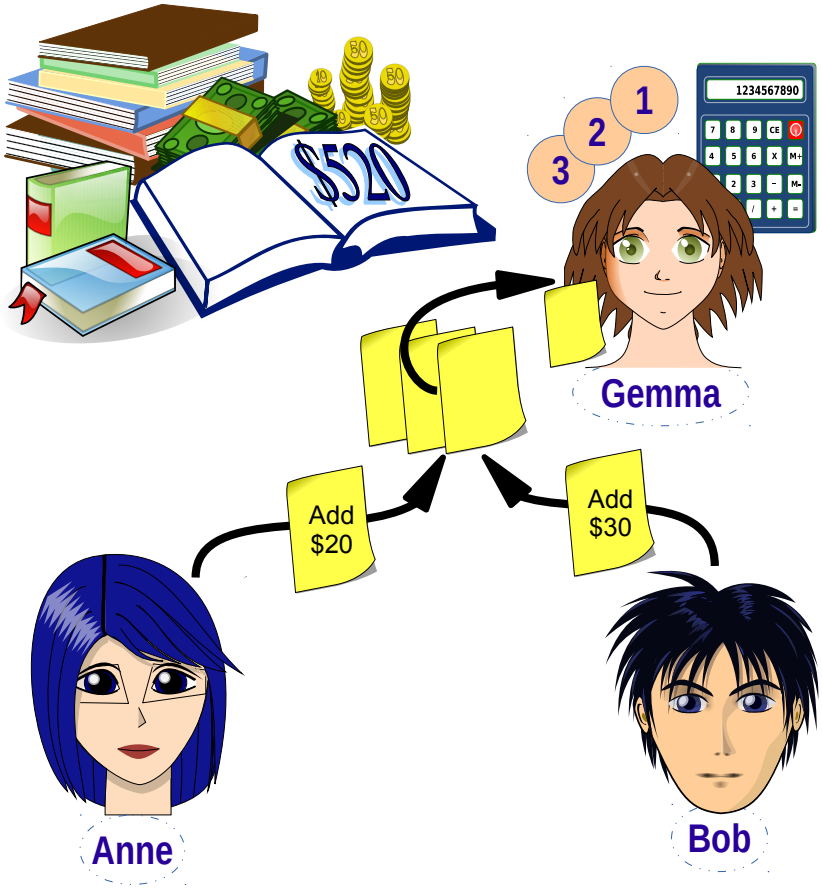


Figure (10.2) A new policy is put into place, not only to manage the book regarding the cash balance, but all the account books. The new rule states that only Gemma should touch the books, and that any access to them must be through her. This means that Anne and Bob must now write memos for Gemma to take action on the cash balance. If multiple employees wish to access any of the books at the same time, these requests (i.e. the memos) are queued up for Gemma to process one at a time.

Gemma will pick up one of the memos from the pile and complete the instructions requested on it. This is known as the *single thread rule*, where a dedicated thread is assigned the sole responsibility of accessing unprotected data. The single thread rule is implemented in `eg01.CashBalanceWithMemos.java`:

```
BlockingQueue<Memo> pileOfMemos = new LinkedBlockingQueue<Memo>();
...
// Anne creates a Memo requesting $20 to be added
pileOfMemos.add(new Memo(20));
...
// Bob creates a Memo requesting $30 to be added
pileOfMemos.add(new Memo(30));
```

This program differs from the first one, in that Anne and Bob never directly access `currentBookBalance`. Instead, they each create a `Memo` and place it on the `pileOfMemos`. The `Memo` class is a `Runnable` instance, defining the three steps necessary to modify the `currentBookBalance`:

```
class Memo implements Runnable {
   private int amountToAdd;
   Memo(int a) {
      this.amountToAdd = a;
   }
   public void run() {
      int observedAmount = currentBookBalance;              // 1
      blink();
      int calculatedAmount = observedAmount + amountToAdd; // 2
      blink();
      currentBookBalance = calculatedAmount;                // 3
   }
}
```

Gemma then polls the `pileOfMemos`, taking one `Memo` at a time and completing the instructions on it:

```
// Gemma
Memo nextMemo = null;
while ((nextMemo = (Memo)pileOfMemos.poll(1,TimeUnit.SECONDS)) != null) {
   nextMemo.run();
}
```

If Gemma waits longer than 1 second, she assumes no more `Memos` will arrive and ends her work. Notice that locks were not necessary to protect the `currentBookBalance`, since Gemma is the only one that has direct access to it. Because she executes one `Memo` at a time, there is never any intertwining of the three instructions within a `Memo`. When we execute this program, we will always get the correct result of $570.

10.4 Here comes the auditor

We think of Gemma's role in the company as being the accountant; to ensure the correctness of the company books, she is the only one within the company allowed to access the books directly. At some stage, a tax auditor may contact the company and inquire about the state of the company's financial records (figure 10.3). Naturally, the auditor has authority (and skills) to inspect the company books without corrupting them (it is his job, after all). In this regards, the books become a medium of communication where the outside world sees the state of the company. If the auditor requires specific jobs from the company, he will telephone the company and explain what he needs. Gemma would answer the phone and take note of the request. Hopefully, Gemma will be able to fulfill the auditor's request in a short amount of time. Since there is only one telephone, and Gemma is the only company representative allowed to operate it, then prolonged handling of any request will mean that other (external) people trying to contact the company will be frustrated as they encounter a busy dial tone. Not only will this occur when Gemma is busy on the telephone, but it

will also occur during the time she is completing instructions on the memos given to her from Anne and Bob. Any attempt to call the company at this time will again frustrate the outside world, as the phone rings and rings without being answered.

So, how does this all correspond to a GUI application? In a GUI application, we have the same policies and interactions in place. Here is how the analogy relates to a GUI application:

- The *company* represents the **application**.
- The company *books* represent the **GUI components** that reflect the state of the application. Much like how there are many forms of books a company may maintain, there are many forms of GUI components an application may maintain. Some are forms of input (e.g. text fields and buttons), while others are forms of output (e.g. progress bars and message dialogs). Regardless, they are all GUI components and it is not safe for multiple access.
- The *auditor* (or anyone outside the company) represents the **users** of the application.
- A phone *call* represents an **event** from a user that requires attention. The *arrival of a new memo* to the pile also represents an event.
- The *employees* within the company represent the **threads** within the application. More specifically, *Gemma's* role as *accountant/receptionist* represents the **GUI thread**'s role of sole responsibility for the GUI components. *Anne and Bob* represent the **background threads**, and they should never access the GUI components.
- A *memo* represents a **Runnable (set of instructions)**.
- A *busy dial tone* experienced by the outside world represents an **unresponsive application** or **"frozen" GUI**. In fact, any time Gemma is doing any form of processing (e.g. on the phone, or executing a memo), this corresponds to the GUI thread being busy handling an event. Such processing should be kept to a minimum, ensuring Gemma is kept as free as possible. In other words, the GUI thread should be as idle as possible so that it can respond immediately to any new events without noticeable lag.

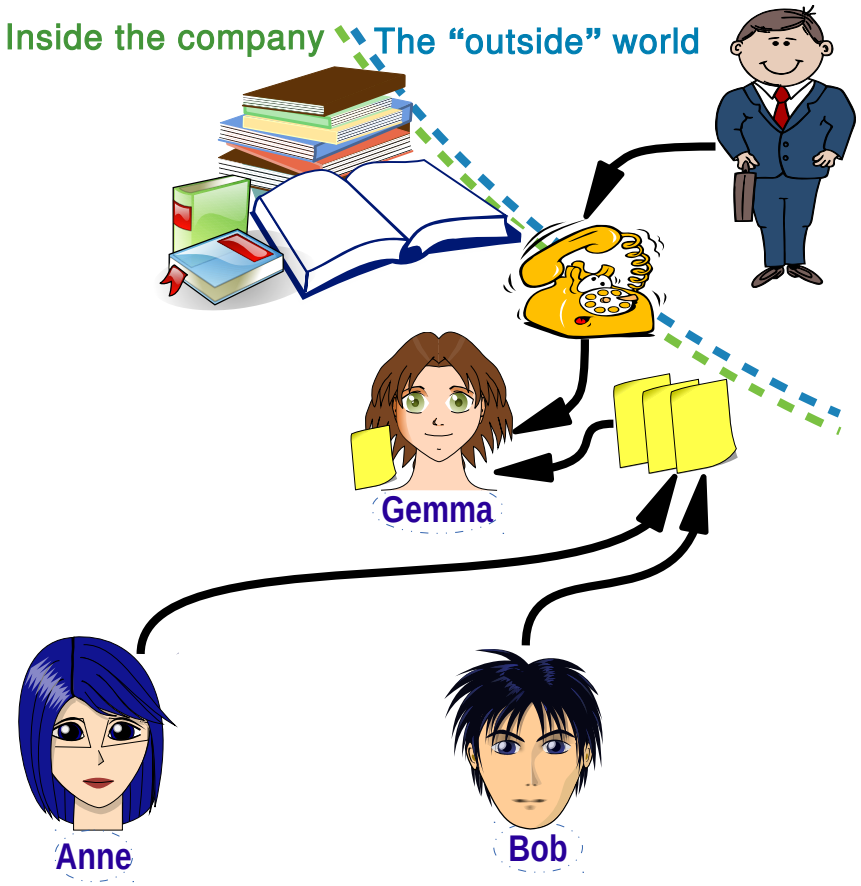


Figure (10.3) As the accountant for the company, Gemma is responsible for all the book keeping. The auditor represents an external entity, or client, interested in interacting with the company (i.e. the **users** of the application). The auditor is able to interpret the company's state from the book records and is able to communicate with the company by making phone calls. Gemma is also the only person within the company that responds to the phone calls. If Gemma is busy on the phone, then the outside world gets a busy dial tone. This will inevitably frustrate anyone from outside trying to communicate with the company. Anne and Bob do not interact with the phone, much in the same way they should not access the books.

- The *pile of memos*[‡] (and list of phone messages) represents the **event queue**, containing events yet to be handled by the GUI thread.

- When Gemma is idle and "*on the lookout*" for memos and messages to arrive, this corresponds to the GUI thread being in the **event loop**. This is the ideal situation, meaning the GUI thread is ready and waiting to respond instantaneously as soon as an event arrives.

So, what does it mean having the "outside" world interacting with the application? In the analogy, these represent customers or auditors that will interact with the company through the phone. We need to ensure this is all responsive. In terms of the application, this represents the user interaction. To be truly responsive, the outside world demands:

1. Continuous responsiveness that never results in a frozen GUI, and

2. Frequent updates for tasks that take a long time (i.e. an update at the end is insufficient for long-processing tasks).

We can appreciate these points from our own personal experience using GUI applications. How many times have you pressed a button on an application and it immediately freezes? If you are unfamiliar with the application, chances are you are wondering if the entire application has crashed and if you should kill it using the operating system's task manager. If you are familiar with the application, you may have the patience to wait for it to complete its actions and come back to life. Nonetheless, this behavior refers to the first point above and is undesirable to say the least. Even when this first requirement is fulfilled, is it sufficient? Again, from your own personal experience, how many times have you clicked a button and the application displays a "Processing, please wait" message but gives no other hint as to the progress it is making? What we want is some sort of clue that quantifies the remaining time, either a determinate progress bar or a constant message updates that confirm to us "yes, the application is making progress".

[‡]Although "pile" is used in this analogy, the memos will be processed in a first in first out (FIFO) manner.

10.5 Single-thread GUI fundamentals

This section will present the two fundamentals pertaining to GUI concurrency. While the examples are presented in the context of Java, these fundamentals are consistent with almost all GUI toolkits you will encounter. Following these rules will ensure our applications are both correct (without race conditions) and responsive.

10.5.1 Fundamental 1: Correctness

So, what is the relationship between the single-thread rule of section 10.3 and the GUI aspects discussed in section 10.4? As hinted in section 10.4, the GUI components of an application must be protected from possible corruption due to potential race conditions. The easiest way to protect these components is to use the single-thread rule. Almost all the popular GUI toolkits you will come across follow this rule [234234234], where they dedicate a specific thread to access the GUI components (just like in our analogy where Gemma was dedicated to access the books). This thread is most commonly called the **UI Thread**, the **GUI Thread**, or the **Event Dispatch Thread** (EDT) as in Java [5].

To simulate the race condition using a real GUI application, we create our own `ProgressBar` class. This class represents an actual GUI component (it extends Java Swing), which means we can add it to any GUI application. The purpose of this class is to represent the functionality of a real progress bar (for example, `javax.swing.JProgressBar`), but also to illustrate the potential race condition that may arise in using such a GUI component. Only a snippet of this class is shown, but you can have a look at the complete code in `eg02.ProgressBar.java`:

```
public class ProgressBar extends JLabel {
   private int value = 0;
   private double max = 100;
   ...
   public void increment(int delta) {
      int oldValue = value;        // read from memory to CPU register
```

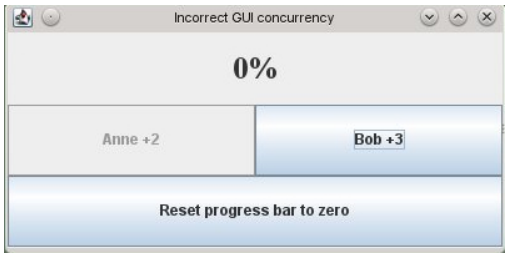
```
        minorCPUstall();
        oldValue = oldValue + delta; // update value in register
        minorCPUstall();
        value = oldValue;            // write to memory from CPU register
        setText(toString());         // update GUI
    }
    public int getPercent() {
        return (int)(100*value/max);
    }
    public String toString() {
        return getPercent()+"%";
    }
}
```

Figure 10.4 shows a simple GUI application (`eg02.BadGUI.java`) that makes use of this progress bar. There are two buttons below the progress bar:

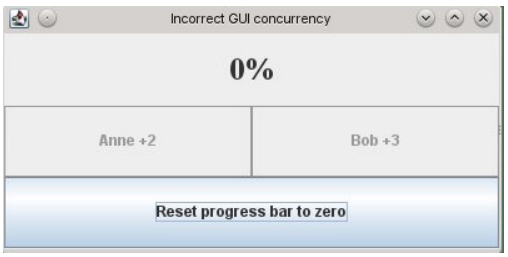
- **Anne +2**: create a new thread that does some work then increments the progress bar by 2.
- **Bob +3**: create a new thread that does some work then increments the progress bar by 3.

If we have a look at how the code is implemented to achieve this seemingly innocent behavior, we see that both threads have direct access to the progress bar instance:

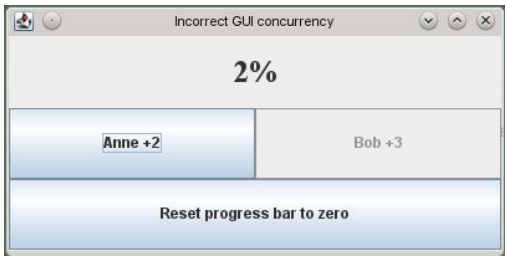
```
public class BadGUI extends JFrame implements ActionListener {
    private JButton btnAnne = new JButton("Anne +2");
    private JButton btnBob = new JButton("Bob +3");
    private ProgressBar progressBar = new ProgressBar();
    ...
```



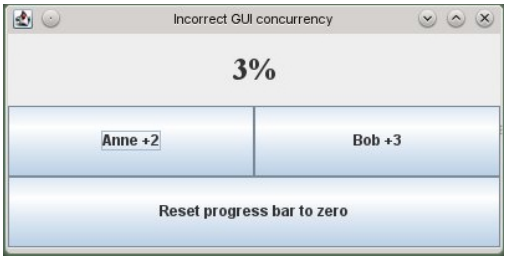
(a) Anne's thread starts to add 2 to the current progress bar value.



(b) Close behind, Bob's thread starts to add 3 to the current progress bar value.



(c) Anne's thread finishes updating the progress bar.



(d) Bob's thread finishes, and overrides the update made by Anne's thread.

Figure (10.4) Bad practice: a race condition when updating a GUI component (`eg02.ProgressBar`) from multiple threads. Since both Anne's and Bob's threads update the progress bar directly themselves, there is the likelihood that the incorrect value results in the progress bar. Instead of showing 5%, it will either show 2% or 3%, depending on which thread was last. Full example in `eg02.BadGUI.java`.

```
    public void actionPerformed(ActionEvent e) {
       if (e.getSource() == btnAnne) {
          btnAnne.setEnabled(false);

          Thread anne = new Thread() {
             public void run() {
                doWork();  // This is correctly performed by non-GUI-thread
                progressBar.increment(2);  // Bad! Accessed by non-GUI thread
                btnAnne.setEnabled(true);  // Also bad!
             }
          };
          anne.start();

       } else if (e.getSource() == btnBob) {
          ... // equivalent code for Bob's thread
       }
    }
}
```

If pressed one at a time, with sufficient time between the completion of each action (i.e. wait for the button to be enabled again), then there is no problem; the value of the progress bar increases to 5%. However, if we were to quickly press the two buttons (as in figures 10.4(a) to 10.4(d)), then one of the threads will override the value of the progress bar that the other thread has written (rather than incrementing onto the updated value). This is because both threads have access to the same GUI component, and the 3 steps of updating a value might with some (bad) luck be interleaved. Not only is this program incorrect since the threads access our custom-made `ProgressBar`, but they also perform the re-enabling on the buttons!

To overcome this problem, we must conform to the single-thread rule discussed in section 10.3. The same program is repeated again (`eg02.GoodGUI.java`), only this time using

the correct approach by conforming to the single-thread rule:

```
public class GoodGUI extends JFrame implements ActionListener {
   ...
   public void actionPerformed(ActionEvent e) {
      if (e.getSource() == btnAnne) {
         btnAnne.setEnabled(false);

         Thread anne = new Thread() {
            public void run() {
               // This is correctly performed by non-GUI-thread
               doWork();

               // GUI-related work moved to a "memo" for the GUI thread
               SwingUtilities.invokeLater(new Runnable() {
                  public void run() {
                     progressBar.increment(2);
                     btnAnne.setEnabled(true);
                  }
               });
            }
         };
         anne.start();

      } else if (e.getSource() == btnBob) {
         ... // equivalent code for Bob's thread
      }
   }
}
```

The difference here is that the updating of the progress bar and button are no longer performed by Anne's and Bob's threads. Instead, the instructions to update the GUI are wrapped inside a `Runnable` instance (representing the memo in our analogy) and passed to the GUI toolkit, requesting that the GUI thread invoke these instructions. This memo is submitted by the respective background thread (e.g. `anne`) after the `doWork()` computation is completed. Regardless of which memo will be picked up by the GUI thread, they will always be executed one at a time since they get piled up in the GUI thread's event queue. Go ahead and modify the `minorCPUstall()` function inside the `ProgressBar` class to increase this stall amount. The correct result is always achieved!

10.5.2 Fundamental 2: Responsiveness

Section 10.5.1 demonstrated the single-thread rule in the context of GUI applications in order to protect the GUI components. In other words, the purpose was to ensure program *correctness*. This section will now demonstrate the single-thread rule with another purpose in mind: ensuring a *responsive application*. While this is not required in contributing towards the correctness or functionality of the application, it is essential in contributing towards a positive user experience and therefore overall user satisfaction. In fact, you could even consider an unresponsive application as dysfunctional!

Purpose of concurrency Based on section 10.4, we already understand that we need to allow the GUI thread to be idle as much as possible in order for it to patrol the event loop and therefore react to new events without delay. To achieve this, the GUI thread employs background threads that perform all the long-lasting processing that would otherwise preoccupy the GUI thread for an unacceptable amount of time. Ultimately, this allows the GUI thread to immediately return to the event loop in anticipation of new events arriving, while the background threads are doing the real work. This is the concept of *GUI concurrency*, where the background thread (executing time-consuming computation) is working concurrently with (i.e. at the same time as) the GUI thread (patrolling the event loop). You will

also hear terminology such as "the time-consuming computation is executed *asynchronously*", which is a fancy way of saying the time-consuming computation is progressing independently of the patrolling of the event loop (i.e. on its own time and in "its own little world").

Classifying a GUI's streams of instructions As already hinted in the previous section, the underlying concept behind concurrency is that of a thread. By having multiple threads, we can logically perform multiple streams of instructions at the same time. In the context of a GUI application, we are interested in separating the instructions into two particular streams of instructions. Each of these streams of instructions will be executed by a thread in order for the streams to progress concurrently:

- **Event management mechanism**: this refers to the instructions that define the administration relevant to the event loop, including the enqueuing, dequeuing and handling of events on the event queue. Fortunately, GUI toolkits typically provide an implementation for this event handling mechanism so that programmers do not need to manage it (or even see it!). This also includes the nomination of the GUI thread to manage all of this communication in the event loop. All that programmers need to do is specify the stream (or block) of instructions for the handling of those events (i.e. the response to a particular event).

- **Event handling logic**: this refers to the programmer-defined stream of instructions that depict what should happen when an event is encountered. The GUI thread will initially commence handling the event, but it is the responsibility of the programmer to determine if the computation will be time-consuming. If so, then the programmer needs to "free the GUI thread" by creating a new thread to take over. The GUI thread therefore classifies the event as being "sorted out", and immediately returns to the event loop. This will avoid any "freezing" of the application.

In Java, the most primitive approach to achieve this is using `Thread`s and the `SwingUtilities` class to hook into the event management system whenever necessary. This is shown by the

Figure (10.5) Good and Bad application. This application contains a standard progress bar (top row), 2 buttons with time-consuming tasks, and a third button to test responsiveness of the application (it changes color as soon as it is pressed). When either of the "Good" or "Bad" buttons is pressed, the progress bar is incremented. The only difference is that the "Bad" button freezes the entire application until the action is completed, whereas the "Good" button maintains application responsiveness, allowing other buttons to be pressed. The full code is found in `eg03.GoodAndBadGUI.java`.

program of figure 10.5, which demonstrates both a responsive and unresponsive handling of events within the same application. The code snippet below refers to the event handling logic when any of the 3 buttons are pressed:

```
public void actionPerformed(ActionEvent e) {

   // the GUI thread can quickly create a new color
   if (e.getSource() == btnResponsive) {
      btnResponsive.setBackground(createRandomColour());
      return;
   }

   // the other buttons involve some time-consuming work being performed
   if (e.getSource() == btnBad) {
```

```
            // The current thread (the GUI thread) does the work itself...
            doWork();
            // ... and then updates the progress bar
            progressBar.setValue(progressBar.getValue()+1);
        } else {
            // The GUI thread asks a background thread to take over...
            Thread bob = new Thread("Bob") {
                public void run() {
                    // the work is performed by the background thread...
                    doWork();
                    // ... and the background thread asks the GUI thread to
                    //                                  update the progress bar
                    SwingUtilities.invokeLater(new Runnable() {
                        public void run() {
                            progressBar.setValue(progressBar.getValue()+1);
                        }
                    });
                }
            };
            bob.start();
        }
    }
```

The actionPerformed() function is the event handler that responds to any of the buttons being pressed. The GUI thread always enters this function. Here, we first query to check if the event received was in regards to the responsiveness button. If this is the case, then the color of the button is updated. Since this computation can be performed without any noticeable lag, it is fine for the GUI thread to execute this and then end the event handler. If the event was in response to the "Bad" button, then the GUI thread decides to perform the time-consuming

`doWork()` function, and then increment the progress bar. This ultimately preoccupies the GUI thread, meaning other events cannot be responded to. The final situation refers to the desired behavior, where the `doWork()` function is being assigned to another thread ("`Bob`"), which allows the GUI thread to end the event handler and respond to other events. In the meantime, when thread "`Bob`" completes `doWork()`, it requests the GUI thread to update the progress bar (since only the GUI thread should access GUI components). When you run the examples, notice the output printed that state the name of the thread executing the respective sections of code.

10.6 More elegant library support: SwingWorker

The code snippets of section 10.5 demonstrated how we can achieve a responsive and thread-safe application by resorting to using primitive libraries existing in the Java library (in this case the `Thread` and `SwingUtilities` classes). While this approach got the job done and met our requirements, it does pose some disadvantages:

- It contained a large amount of boilerplate code to create background threads and send memos back to the GUI thread. This problem will be exacerbated should we need to send intermittent memos to the GUI thread (i.e. not just at the very end).

- Notice how we are creating a new thread every time the "Good" button is pressed. In most cases, this will not be an issue if we are not expecting to have too many tasks. However, if we end up having lots of threads that perform a large amount of computation, then we risk the chance of reducing performance of the application since a lot of time will be dedicated to managing the threads rather than executing the work. A smarter solution would create a fixed number of threads, and instead queue the work to be executed as a thread frees up.

To solve the points above, yet to retain respect to the single-thread GUI model, Java introduced the `SwingWorker` class. This provides a more elegant solution by dealing with the creation and management of a team of background threads, while also reducing the

boilerplate code required. The code snippet below demonstrates how the event handler is modified:

```
public void actionPerformed(ActionEvent e) {

   // ... same as before

   if (e.getSource() == btnBad) {
      // ... same as before
   } else {
      Memo memo = new Memo();
      memo.execute();
   }
}
```

What has changed? The code remains essentially identical to that of section 10.5.2, except now we create a `Memo` instance and tell it to `execute()`. It definitely looks more elegant than the code we had before! All the logic in regards to doing the work and updating the progress bar we define in `Memo`:

```
class Memo extends SwingWorker {
   protected Void doInBackground() {
      doWork();
      return null;
   }
   protected void done() {
      progressBar.setValue(progressBar.getValue()+1);
   }
}
```

You will notice that this class is not too complicated at all. In fact, `SwingWorker` helps guide us by specifying which functions we should be implementing. In our simple example, `doInBackground()` is the place we specify any time-consuming computation that will be passed on to the background thread. The `done()` function refers to any computation that must be performed by the GUI thread when `doInBackground()` is completed.

You are probably wondering, where is the background thread? This is the other elegance to this solution, in that the programmer does not need to create or manage the background threads that will execute the `SwingWorker` instances. This is all managed automatically by the library using a pool of threads that are dedicated to processing `doInBackground()` functions. If you execute the example code provided (`eg04.GoodAndBadSwingWorker.java`), the only difference you will notice is in the output printed. Notice how the names of the threads are now something like "`SwingWorker-pool-1-thread-5`" or "`SwingWorker-pool-1-thread-2`", which refers to the threads that are being automatically managed to execute background work. In order to see the multiple background threads being managed by the SwingWorker class, try pressing the "Good" button as fast as you can 15 times. Every time you press the button, it enqueues a memo that will eventually be processed by one of the threads. By reading the thread names, we notice that the same 10 threads are being recycled (the exact number might be slightly different for you).

You will also notice that it is still the GUI thread that is executing the `done()` function. Figure 10.6 illustrates how we can visualize SwingWorker in terms of our company analogy. The newly submitted jobs refer to the `execute()` function being performed on a newly created SwingWorker instance. When one of the SW-threads is idle, it picks up the next memo from the pile and completes the top blue section. Upon completion, that memo is passed on to the GUI thread to complete the bottom orange section. Concurrency is achieved by having multiple background threads that are available to execute the long-processing computations. By separating the GUI-related computation in a different section, this also achieves responsiveness since the GUI thread is not unnecessarily occupied.

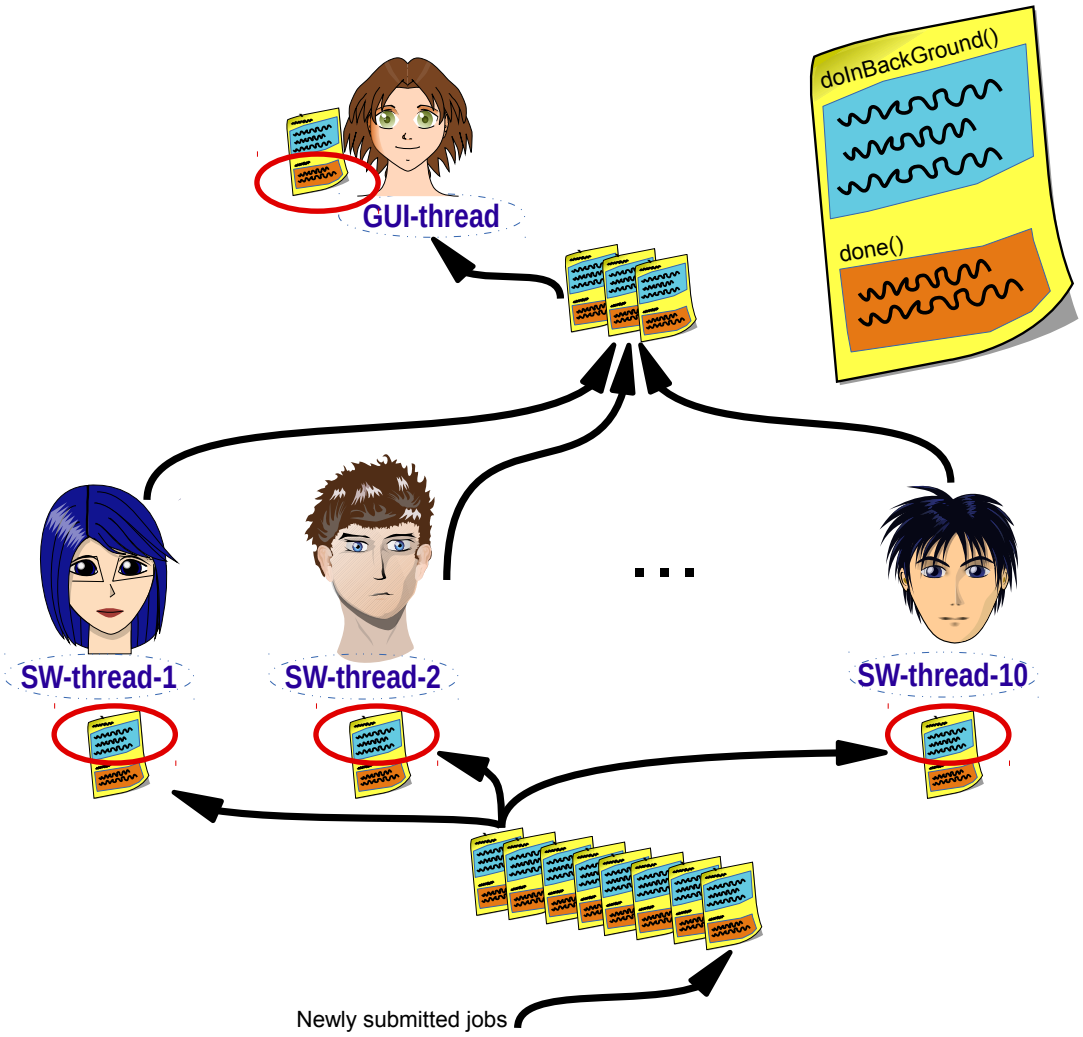


Figure (10.6) SwingWorker is designed to meet both the correctness and responsiveness fundamentals of GUI concurrency. The top right corner shows how we can visualize a SwingWorker instance as a memo containing 2 sections. The blue section (`doInBackground()`) is reserved for one of the SW threads, while the orange section (`done()`) is reserved for the GUI thread. There are 2 queues: the first is when the job is submitted and it waits for a SW thread, while the second is when the background portion is completed and the memo is passed on to the GUI thread to execute the GUI-related portion.

10.6.1 Improving user experience with intermittent results/updates

A big part of GUI applications is ensuring a positive user experience. In this regards, responsiveness not only means avoiding a freezing user interface, but also providing *regular* updates to the user. This is especially important for background jobs that take a long amount of time. Examples include displaying search results (e.g. searching through emails) as they are found, or progressively rendering thumbnails of images in a folder. Bear in mind that we want to still conform to the GUI concurrency fundamental of correctness. This means that it must be the GUI thread updating the user interface with the information – but how does the GUI thread know about the background thread's progress on a given task? The general idea is simple:

- The background thread, as it processes the `doInBackground()` section, decides that it has accomplished a significant amount of work that warrants celebration. Since it is not allowed to access the GUI components directly itself, it simply publishes this achievement and resumes processing the remainder of the `doInBackground()` section.
- The GUI thread, upon hearing the update request, takes the published data and displays it on the GUI.

How is this achieved using SwingWorker? Before seeing the code, lets have a look at the general concept with the help of figure 10.7. As before, we have the `doInBackground()` and `done()` sections that are executed by the background SW-threads and the GUI thread respectively. There is a new section, `process(List)`, which is executed by the GUI thread whenever an "attachment" is added on the SwingWorker "memo". How do these attachments get there? This is the job of the background thread, to `publish()` these items whenever it feels it has made substantial progress in the background processing. Rather than waiting for the GUI-thread to acknowledge receipt of the attachment, the background thread continues processing the remainder of the `doInBackground()`. This is how the attachments potentially "pile up" for the GUI thread to `process()` (hence a List of intermittent results to process).

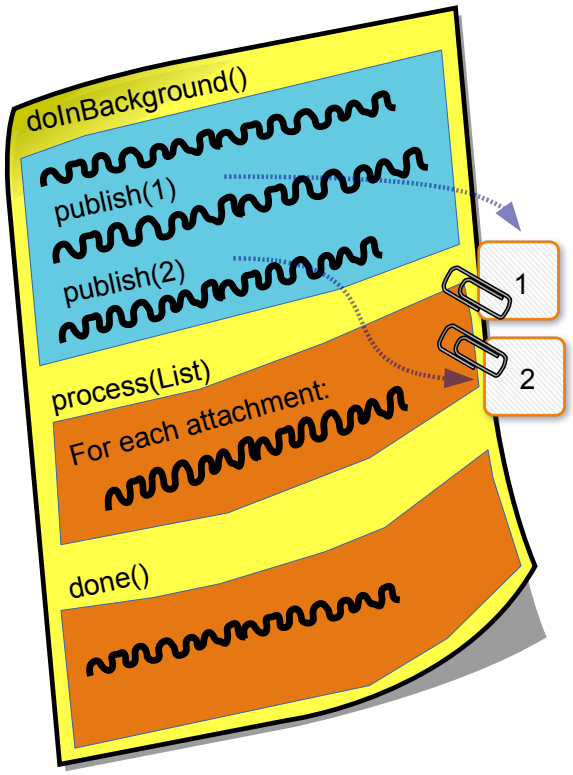
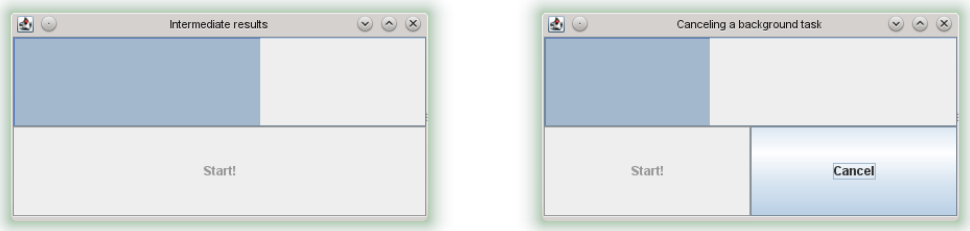


Figure (10.7) Our final visualization of Swingworker to include how intermittent results/updates are propagated from a background thread to the GUI thread. As the background thread processes the doInBackground(), it frequently decides to publish intermittent data. We imagine this as an attachment to the SwingWorker instance (i.e. the memo). The GUI thread sees these attachments, and executes the process() section for the list of attachments as they come along. As before, the orange sections are GUI-related (to be executed by the GUI thread), while the blue section refers to non-GUI and time-consuming work.



(a) (b)

Figure (10.8) (a) An example application demonstrating a background computation with intermittent updates via the progress bar. (b) The same application is extended to allow canceling of the background computation.

One of the most common cases of publishing intermittent results is when a progress bar is used. Figure 10.8 (a) shows such an example, while below is the code snippet (full code in `eg03.ManyUpdates.java`) that demonstrates the correct way to frequently update the status of a progress bar:

```
class Memo extends SwingWorker<Void, Integer> {
   protected Void doInBackground() {
      for (int i = 1; i <= 10; i++) {
         doWork();
         publish(i); // create a new ‘‘attachment’’
      }
      return null;
   }
   protected void process(List<Integer> attachments) {
      // process all the attachments that have piled up
      for (int attachment : attachments) {
         progressBar.setValue(10*attachment);
      }
   }
   protected void done() {
```

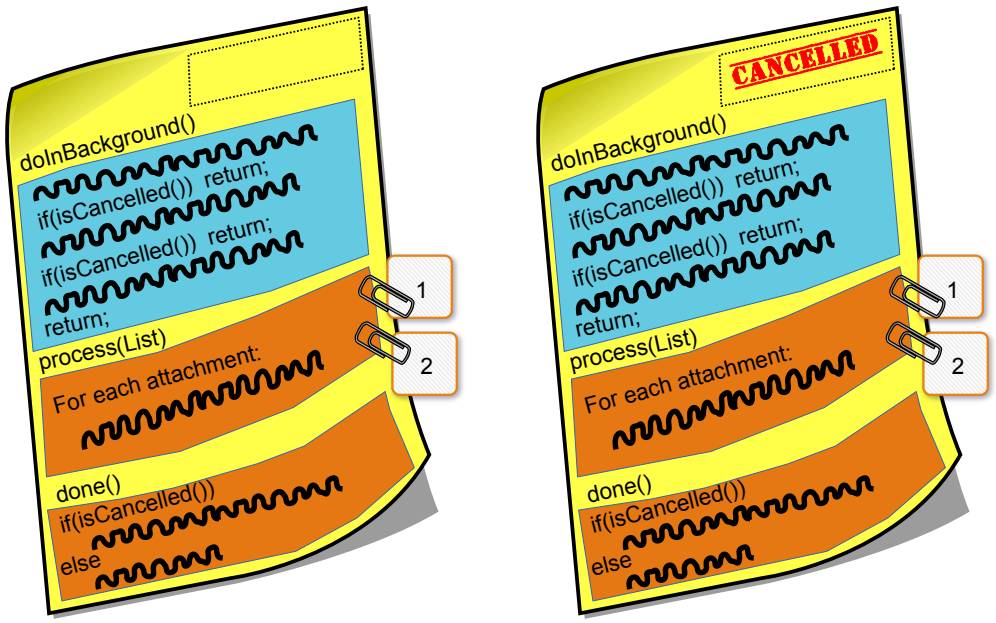
```
            // doInBackground() ended, so re-enable start button
            btnStart.setEnabled(true);
        }
    }
```

10.6.2 Canceling background tasks

If you ran the example of figure 10.8(a), you probably eventually felt like something was not quite complete. Did you notice how we had no way to cancel the task? Clearly, there was no cancel button on the GUI. If we were to introduce such a cancel button (as in figure 10.8(b)), what does this mean in the context of a background task that was executed concurrently? Well, first of all, whoever wishes to `cancel()` the background task obviously needs access to the very same SwingWorker instance that was initially told to `execute()`. In other words, we need to declare our SwingWorker memo at a scope such that it is still accessible to the event handler:

```
public class ManyUpdatesWithCancel extends JFrame implements ActionListener {
    private JButton btnStart = new JButton("Start!");
    private JButton btnCancel = new JButton("Cancel");

    private Memo memo;  // instance at a scope accessible to all handlers
    ...
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == btnCancel) {
            memo.cancel(true);
        } else if (e.getSource() == btnStart) {
            memo = new Memo();
            memo.execute();
        }
```



(a) (b)

Figure (10.9) Once a SwingWorker memo has started executing, it needs to periodically check if it has been requested to cancel. This is to allow the background SW-thread to tidy up and end the `doInBackground()` in a clean manner. (a) Before a cancel request has been made. (b) After a SwingWorker memo instance has been instructed to `cancel()`. The act of canceling a memo essentially means it is stamped; however, this has no effect unless it is checked for, and acted upon.

```
        }
    }
```

Figure 10.9 shows how canceling is implemented for background tasks. Assume the SwingWorker memo instance is being executed by one of the background SW-threads. In the meantime, the cancel button was pressed, so we invoke `cancel()` on that same memo instance. Abruptly killing the background SW-thread as it is executing `doInBackground()` is an unsafe practice. Instead, what happens is the memo is stamped with a cancel request. However, for this to really take effect, the background SW-thread needs to frequently check the memo status just in case it has been stamped with cancel. This is achieved by calling `isCancelled()`, which checks for the status. If this returns `true`, then the `doInBackground()` method is ended with an early `return` statement.

Since the `doInBackground()` method has essentially ended (regardless of whether it was canceled or not), the GUI-thread takes over the memo by executing the `done()` method. It even executes any remaining attachments in the `process()` method. For this reason, we sometimes might want to take alternative decisions inside `done()` and `process()` depending on whether the memo had been canceled. In this case, the GUI-thread may also use the `isCancelled()` method to check if the memo had been canceled. The example of `eg03.ManyUpdatesWithCancel.java` demonstrates this by discarding attachments if the memo was canceled, and only displaying a final message if `done()` is processed without having received a cancel request.

10.7 Wrapping up

In this chapter, we appreciate the necessity of multi-threading in the context of applications that possess a graphical user interface (GUI). This is especially important as multi-core processors have become the norm for desktop and mobile devices, since the sorts of applications running on these systems will interact with users via the GUI. The limitations of the GUI toolkits available for these paradigms means that programmers need to adhere to the single-thread rule. This ultimately means two things. First, the event handlers that the GUI thread responds to must be kept minimal without noticeable lag to ensure a responsive application. This is achieved by off-loading the long processing computation to a background thread. Second, the background thread must never directly access any GUI component during that time, and should instead request the GUI thread to do so. Ultimately, when developing your next GUI application, you need to keep in mind the user-perceived performance of responsiveness, by implementing the concurrency features discussed in this chapter. This includes intermittent progress updates from background threads, as well as canceling the background tasks. But, as you do so, always remember the single-thread rule.

REFERENCES

[1] S. K. Prasad, A. Chtchelkanova, M. G. F. Dehne, A. Gupta, J. Jaja, K. Kant, A. L. Salle, R. LeBlanc, A. Lumsdaine, D. Padua, M. Parashar, V. Prasanna, Y. Robert, A. Rosenberg, S. Sahni, B. Shirazi, A. Sussman, C. Weems, and J. Wu, "NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates, Version 1," http://www.cs.gsu.edu/~tcpp/curriculum, 2012.

[2] D. Lea, *Concurrent programming in Java: design principles and patterns*, 2nd ed. Addison-Wesley, 1999.

[3] P. Hyde, *Java Thread Programming.* Sams, 2001.

[4] E. Ludwig, "Multi-threaded user interfaces in java," Ph.D. dissertation, University of Osnabrück, Germany, May 2006.

[5] Oracle. (2017) Lesson: Concurrency in Swing. http://docs.oracle.com/javase/tutorial/uiswing/concurrency.

Appendix

eg01.CashBalanceProblem.java

```
package eg01;

public class CashBalanceProblem {

  private static int currentBookBalance = 520;

  public static void main(String[] args) {

    // Anne will execute the three steps, in order to add $20
    Thread anne = new Thread() {
      public void run() {
        int observedAmount = currentBookBalance;     // 1a
        blink();
        int calculatedAmount = observedAmount + 20; // 2a
        blink();
        currentBookBalance = calculatedAmount;       // 3a
      }
    };
    anne.start();

    // Bob will execute the three steps, in order to add $30
    Thread bob = new Thread() {
      public void run() {
        int observedAmount = currentBookBalance;     // 1b
        blink();
        int calculatedAmount = observedAmount + 30;    // 2b
        blink();
        currentBookBalance = calculatedAmount;       // 3b
      }
    };
    bob.start();
```

```
    // Wait for both Anne and Bob to finish the three steps
    try {
      anne.join();
      bob.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println("Final balance = $"+currentBookBalance);
  }

  // Simulate blinking
  public static void blink() {
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

eg01.CashBalanceWithMemos.java

```
package eg01;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;

public class CashBalanceWithMemos {

  private static int currentBookBalance = 520;

  // the queue represents a pile of memos
```

```
private static BlockingQueue<Memo> pileOfMemos = new LinkedBlockingQueue<
    Memo>();

// a definition on a Memo, which states what needs to be done (i.e. the
    three steps to modify the cash balance)
static class Memo implements Runnable {
  private int amountToAdd;
  Memo(int a) {
    this.amountToAdd = a;
  }

  @Override
  public void run() {
    int observedAmount = currentBookBalance;        // 1
    blink();
    int calculatedAmount = observedAmount + amountToAdd;     // 2
    blink();
    currentBookBalance = calculatedAmount;          // 3
  }
}

public static void main(String[] args) {

  // Anne creates a new Memo, requesting $20 to be added
  Thread anne = new Thread() {
    public void run() {
      pileOfMemos.add(new Memo(20));
    }
  };
  anne.start();

  // Bob creates a new Memo, requesting $30 to be added
  Thread bob = new Thread() {
    public void run() {
```

```
        pileOfMemos.add(new Memo(30));
      }
    };
    bob.start();

    // Gemma goes to the pile of Memos, takes one at a time. If the pile is
        empty for more than 1 second, Gemma stops.
    Thread gemma = new Thread() {
      public void run() {
        Memo nextMemo = null;
        try {
          while ((nextMemo = (Memo)pileOfMemos.poll(1, TimeUnit.SECONDS)) !=
              null) {
            nextMemo.run();
          }
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      }
    };
    gemma.start();

    // wait for Anne, Bob and Gemma to finish
    try {
      anne.join();
      bob.join();
      gemma.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println("Final balance = $"+currentBookBalance);
  }
```

```
  // Simulate blinking
  public static void blink() {
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```