Why are GUIs single-threaded

Modern GUI frameworks create a dedicated event dispatch loop (EDT) for handling GUI events. The reason is that multithreaded GUI frameworks tend to be susceptible to deadlock:

- Unfortunate interaction between input event processing and any sensible object oriented modeling of GUI components. Actions initiated by the user tend to "bubble up" from the OS to the application. Application-initiated actions "bubble down" from the application to the OS. Combining this tendency for activities to access the same GUI objects in the opposite order with the requirement of making each object thread-safe yields a recipe for inconsistent lock ordering, which leads to deadlock.
- The prevalence of Model-View-Controller pattern. Facroting user interactions into cooperating model, view and controller objects greatly simplifies implementing GUI applications, but raises the risk of inconsistent lock ordering.

Sequential event processing

GUI applications are oriened around processing fine-grained events such as mouse clicks.

- Because there is only a single thread for processing GUI tasks, they are processed sequentially and no tasks overlap. This makes writing tasks easier — you don't have to worry about interference from other tasks.
- The downside is that if one task takes too long to execute, other tasks must wait unil it is finished. If those other tasks are responsible for responding to user input or providing visual feedback the UI will appear frozen, even "Cancel" button will become unresponsible. You must run such tasks in another thread. Nevertheless, you will have to send notifications to GUI thread and handle cancelation, so things can get complicated quickly.

Thread confinement in Swing

All Swing components and data model objects are kept consistent not by synchronization but by thread confinement: they are confined to the event thread. Any code that accesses these objects must be run in the event thread.

- You don't need to worry about synchronization
- You can't access presentation objects from outside the event thread at all

Few exceptins, of course:

- **SwingUtilities.isEventDispatchThread** — determines whether the current thread is the event thread
- **SwingUtilities.invokeLater** — schedules a **Runnable** for execution on the event thread

- **SwingUtilities.invokeAndWait** — schedules a **Runnable** for execution on the event thread and blocks the current thread until it completes (callable from a non-GUI thread only)
- methods to enqueue a repaint or revalidation request on the event queue
- methods for adding or removing listeners

The Swing event thread can be thought of as a single-threaded Executor that processes tasks from the event queue. The worker thread sometimes dies and is replaced by a new one, but this should be transparent to tasks.

Short-running GUI tasks

In a GUI application, events originate in the event thread and bubble up to application-provided listeners, which will probably perform some computation that affects the presentation object. For simple, short-running tasks, the entire action can stay in the event thread; for longer-running tasks, some of the processing should be offloaded to another thread.

Long-running GUI tasks

Sophisticated GUI applications may execute tasks that take longer than the user is willing to wait. These tasks must run in another thread so that the GUI remains responsive while they run.

Prior to Java 6 Swing doesn't provide any mechanism for helping GUI tasks execute code in other threads. You can create your own **Executor** for processing long-running tasks. A cached thread pool is a good choice for long-running tasks: only rarely do GUI applications initiate a large number of long-running tasks, so there is a little risk of the pool growing without bound.

- You can use an executor to submit a long-running task from an event listener in a "fire-and-forget" manner. This does not include cancellation, progress indication or visual completion feedback.
- When the computation in another thread completes it may queue another task to run in the event thread to inform the UI about completed operation.
- To cancel a long running task you can save a reference to the **Future** object represnting the result and invoke **cancel** method on it with **mayInterruptIfRunning** argument set to true.
- Use **FutureTask** to implement cancellation, progress and completion indication altogether. Implement cancellation with "cancel" button listener that cancels **FutureTask**.

Shared data models

Swing presentation objects (**TableModel**, **TreeModel**) are confined to the event thread.

In simple GUI programs, all the mutable state is held in the presentation objects, and the the single-thread rule is easy: don't access the data model or presentation components from the main thread.

More complicated programs may use other threads to move data to or from persistent store, so as not to compromise responsiveness. The presentation model may be just a view onto another data source such as a database. In this case, more than one thread is likely to touch the data as it goes into or out of the application.

Thread safe data models

As long as responsiveness is not unduly affected by blocking, the problem of multiple threads operating on data can be addressed with a thread-safe data model.

- If the data model supports fine-grained concurrency, the event thread and background thread should be able to share it without responsiveness problems.
- The downside is that it does not offer a consistent snapshot of data
- You can use versioned data models such as **CopyOnWriteArrayList**. However such structures offer good performance on frequent traversals and infrequent updates. More specialized implementations are possible but are not easy to build.

Split data models

A program that has both a presentation-domain and an application-domain data model is said to have a split-model design. In a split-model design:

- the presentation model is confined to the event thread
- the other model, the shared model, is thread-safe and may be accessed by both the event thread and application threads
- the presentation model registers listeners with the shared model so it can be notified of updates

The presentation model can be updated from the shared model

- By embedding a snapshot of relevant state in the update message. This approach is simple, but works well when the data model is small and is not updated frequently, and the structure of two models is similar.
- By having the presentation model retreive the data directly from the shared model when it receives an update event. More complex approach, works best when the data is large or updates are frequent, or when any of both sides contains information not visible to the other side. Another advantage is that the finer-grained updates can improve the percieved quality of the display if only a small portion of data has changed.

Other forms of single-threaded subsystems

Thread confinement is not restricted to GUIs: it can be used wherever a facility is implemented as a single-threaded subsystem.

Sometimes thread confinement is forced on the developer for reasons that have nothing to do with avoding synchronization or deadlock. You can create a dedicated thread or a single-threaded executor and provide a proxy object that intercepts calls to the thread-confined object and submits them as tasks to the dedicated thread.

Future and **newSingleThreadExecutor** work together to make this easy. The proxy method can submit the task and immediately call **Future.get** to wait for the result.