COMP6700/2140 GUI and Event Driven Programming

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

April 2017
Topics

Graphical User Interface Programming

Asynchronous Control-Flow: Event-Driven Programming

Java Platform for GUI and Rich Client Programming:
- Abstract Window Toolkit (AWT)
- Swing (and Java Foundation Classes)
- JavaFX

JavaFX and Rich Client Applications

Scene Graph
Graphical User Interface

Nowadays, all computer programmes which are intended for interaction with the human user are created with Graphical User Interface (while “In the Beginning was Command Line”, by Neal Stephenson). According to Wiki: “allows for interaction with a computer or other media formats which employs graphical images, widgets, along with text to represent the information and actions available to a user. The actions are usually performed through direct manipulation of the graphical elements.” The history of GUI development so far is very worth reading. Since standard computers have a well defined and constrained (less constrained now, after introduction of iPhone and iPad) interface, the type of GUI which can be deployed on them is often called WIMP (window, icon, menu, pointing device).

One of the very first GUI as it was created by Xerox 6085 (“Daybreak”) workstation in 1985. It inspired the first mass-produced computer with native GUI — Macintosh, by Apple Comp. As Steve Jobs explained regarding this influence by quoting Picasso: “Ordinary people borrow, real artist steals”.

Alexei B Khorev and Josh Milthorpe (RSCS, ANU) COMP6700/2140 GUI and Event Driven Programming
April 2017 3 / 12
Java in the GUI Game

Was there from its birth

1995: – as the part of the first release, Java contained Application Window(ing?) Toolkit, **AWT**, a collection of containers, widgets, events and event handlers (Java API also contained `java.geom` package for 2D drawing, so called **Java2D**). Implementation of AWT was “heavy-weight”, *ie*, in native code. Strong OS dependence in look and feel. Still has some use today (*eg*, Apple programming advice on DialogBox use).

1998: – new, partially dependent on AWT, set of containers and widgets (also some events, but those from AWT are still in active use), called **Swing** (not sure why?) was included in Java SDK 1.2. Swing API “lives” in `javax.swing` and `javax.swing.*` packages. Swing is light weight, *eg*, most of its classes in implemented in Java. First releases had performance issues, but over the years achieved significant improvement and expansion. *Note*: reliance on AWT events results in otherwise pure Swing applications have OS specific traits of behaviour. Example: **MouseSpy.java**.

2012: – Java SDK 1.7 is released with a new, pure Java, GUI and Rich Client API called **JavaFX**. The actual version is 2, so there is some history:
  - Conceived by Chris Oliver (SeeBeyond) as a scripting language **FFF** and API (c. 2006).
  - Bought by Sun, and the language renamed as **JavaFX** (2007); versions 1.0-1.3 released between 2008 and Sun’s acquisition by Oracle in 2010. Had its own compiler to bytecode.
  - JavaFX 2 released as API only (“Forget that stupid little language”, S. Blackburn). Complete alternative to AWT/Swing.
  - JavaFX programs can be run as standalone, or deployed to be executed in a browser.
Command-line: Application has control at all stages of execution. User's interaction with the application is pre-programmed. Event-Driven: After initialising the user-interface, application passes control over to the operation system. System waits for the user actions and other events, then calls an appropriate function. This is Event-Driven Programming. GUI is type of ED application (non-GUI examples: servers, stream parsers etc).
Event-Driven Programming

Event-driven programming involve a new type of flow-of-controls — *events*. Events are *asynchronous* (happen at unpredictable time), they manifest themselves by altering the state of execution environment. They are detected though this change. Event-driven programs have the following distinctions:

- Control structure is inverted — system has control, not the application
- System waits for an event (like user action, or lapsing a time interval, or else), then calls the appropriate routine in the application
- This can make otherwise simple programs much more complicated
- In GUI applications, one needs a set of classes (widgets) which can detect user’s input, and a set of classes, called *listeners* or *event-handler*, which define what the system will do in response to an event
- Event-driven programs are not only GUI. Events can be generated by a stream of data. *Eg*, the XML-scanning API called SAX, in `org.xml.sax`, is an event-driven library. It provides several listeners which allow to program client’s response to events like start of an XML document, start of an XML element, a data element and so on. The event-driven approach makes the task of complex data processing easier.

GUI Application events:

- user input e.g. click on a button
- internal events e.g. timeouts
- window events e.g. expose
Event-Driven and (Functional) Reactive Programming

The technique called *(Functional) Reactive Programming* has gained traction in recent years with the advent of functional languages or functional language features in traditional languages like Java.

The Reactive Programming is an event-driven programming (ie, programming with an asynchronous event streams) in which you are given “standard” functional programming tricks like

- treating a stream — a sequence of ongoing events ordered in time — as an object or as code (a function) so it can be passed to another stream;
- composing (merging) two (many) streams into one;
- filtering streams;
- mapping one stream (consisting of objects of one kind) into another (consisting of objects of a different kind)

In FRP, a stream can emit events using the interface:

- `value()` (or `next()`)
- `error()`
- `completed()`

FRP is a functional (better?) alternative to the programming which utilises the *Observer* pattern (discussed later).
Event Handling Models

**Polling** (old, single-threaded)

- Application has the *main loop* that checks repeatedly for input or timeouts, then calls the appropriate routine
- Examples include most (pre OS-X) Macintosh programs, Unix programs that just use basic X window
- Leads to a more complex application, messy interfaces to the library
- Largely replaced by the callback mechanism

**Callbacks** (modern, multi-threaded)

- A *callback* is when a routine in the application is called *not* from within the application, but *from the library* (by a widget when it detects a signal)
- The application must first tell the toolkit what events it wants to handle and which routines to call
- Main loop and decoding of events are done by the library
- This happens in more modern higher-level Unix GUI libraries like Xt, Qt, Motif or GTK; also the Microsoft Foundation Classes, Cocoa Framework (MacOSX)
- In Java programming callbacks is done using events and event listeners
**GUI Concepts**

**widget**  a GUI component like a window, a push button, a label, a scroll-bar, a menu item...

**pixel**  a single dot on the display screen; displays typically have $800 \times 600$, $1024 \times 768$, $1280 \times 1024$ pixels

**depth**  the number of bits per pixel - depth 1 = black & white - depth $8 = 2^8 = 256$ colours - depth $24 = 2^{24} = 16,777,216$ colours

**colour map**  translation between pixel values and actual screen colours on a low-depth display

**bitmap**  an image one pixel deep

** pixmap**  an image of arbitrary depth (effectively as a 2D array of pixels, although you may not be able to access them like array elements)

**font**  a mapping from character (or symbol) shapes to bitmaps

**event**  change in computation environment caused by user or smth else which induces application response. An event has the *source* (widget which registers it) and the *target*, an application component which will react to it (change as the result)

**event handler**  part of application which defines how and what is changed as the result of event
JavaFX: Scene Graph

The JavaFX *Scene Graph* (modelled on graphical applications and libraries like vector editing tools, 3D libraries, and video games) is the *model* (discussed later, in Model-View-Controller architecture) of all graphical objects which exist in an application. It contains information about what objects to display, what areas of the screen need repainting, and how to render it all.

Individual objects (buttons, shapes, text etc) are *leaves*; groups of objects are *nodes*. The scene graph is rooted by a container, usually `javafx.scene.Group` or `javafx.scene.Region`, which is “embedded” into a scene object (a window). Once set, an entire scene graph is passed as a stage parameter to the (overridden) method `javafx.application.Application.Application.start()` method as the starting point of execution.
The Structure of a JavaFX Program

1. Your main class must extend `javafx.application.Application`.
2. Optionally include the overridden method called `Application.init()` (a public void method with an empty body), which can be overridden, but must not include “UI statements” (like constructing `Stage` or `Scene`, defining callbacks etc), and may include `fxml-` and `css-` bindings. Note that a different (can be private, take parameters, and even have another name) method `init` can be defined inside the application class and called inside `start()` (like private void `init(Stage primaryStage)` in `MouseEvents.java`) — it’s just an ordinary, coder-defined auxiliary method to modularise the stage assembly.
3. `Application.start(Stage stage)` is the method which launches a JavaFX application. In principle, one has to include `main` for a full ceremony:
   ```java
   public static void main(String[] args) {
     launch(args); }
   ```
   But it is only necessary if the run-time platform (an IDE) cannot deploy your application properly. An IDE like Netbeans ignores `main`; the latter is still required, however, when you need to pass command-line arguments, in which case, you should use the method `javafx.application.Application.getParameters()`.

   ![Diagram](image)

   - `application.init()`
   - `application.start()`
   - `application.stop()`
   - `Platform.exit()`

4. Once launched, a JavaFX application will start main, event-processing and other threads, and run until a callback will cause `Platform.exit()` to execute, which will call `Application.stop()` to shut down the application.
Where to look for this topic in the textbook?

- Hortsmann’s Core Java for the Impatient (not covered)
- Hortsmann’s Java SE 8 for the Really Impatient, Ch. 4.
- Oracle’s JavaFX Tutorial