Parallelism

Slide Series 7

Content

Threads, Timers, TimerTasks
Atomic operations
Race conditions
Synchronization
Deadlock
Concurrent Collections
Swing, Swing Components and Swing thread
Blocking calls
Working threads

Review the Model

We're doing OOD/P that means ...

...build a model of something...

Things happens in the model, sometimes they happens concurrently (the world seems to be parallel?)

- Q: How to handle in model? A: Threads!

Threads

"...the Java virtual machine can support many threads of execution at once. These threads independently execute code that operates on values and objects residing in a shared main memory." // JLS 17

"The behavior of threads, particularly when not correctly synchronized [more to come], can be confusing and counterintuitive."

// JLS 17

Hmm.. this is an understatement..

Threads in Java Program

All Java programs have least one thread (executing the main method)

If using a GUI there will automatically be more threads, also if using a network, ...

- So normally a Java program is multithreaded, more to come...

Usage of Thread in Course

- Timers, scheduled events running in own thread
- Programming with Java Swing, threads involved
- Long running tasks running in own thread

Programming with Threads

Complicated always try to avoid

- State changed concurrently! Very complex!
- Hard to debug, if stopping one thread other still runs
- Program doesn't (normally) run faster, have to switch between threads, takes time (normally more threads than processors)

You'll have a complete course in parallel programming in the autumn... this is just what we need in the lab 2 and some basic general knowledge!

Representing Threads

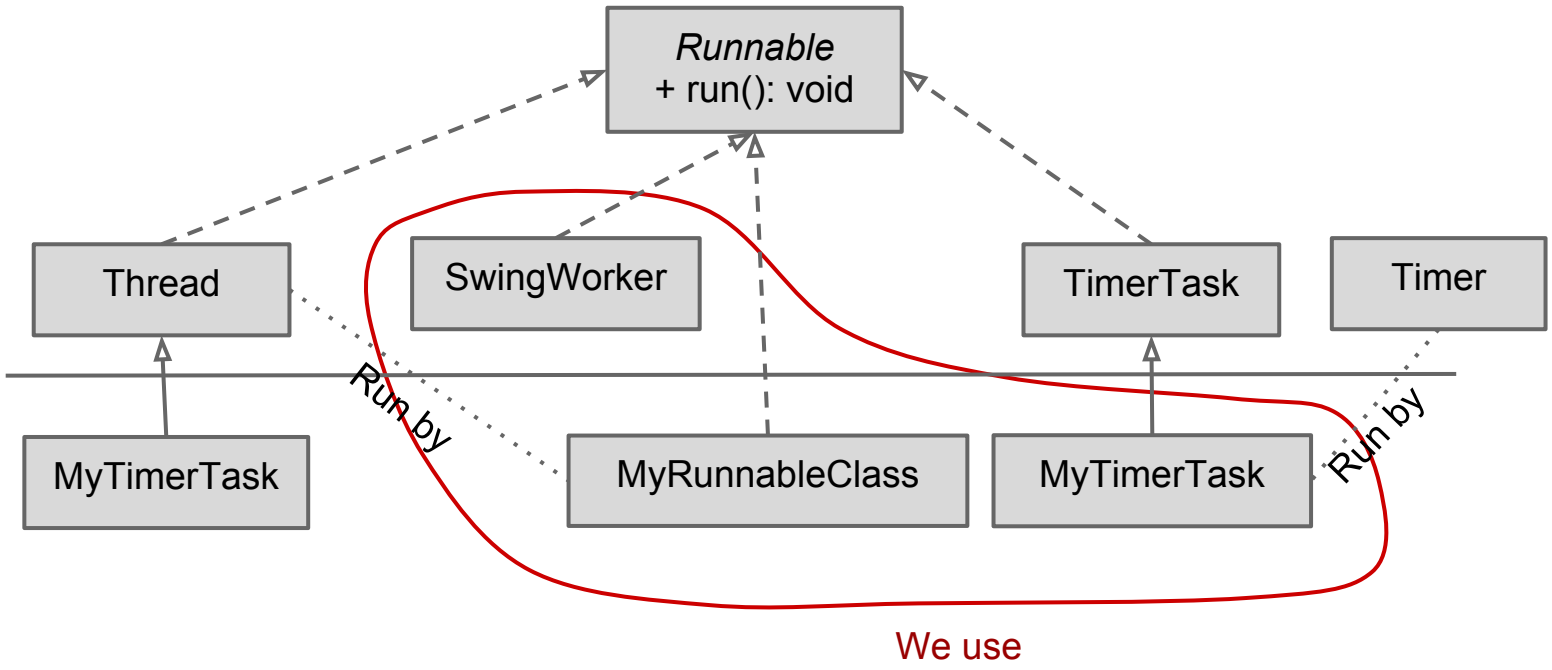
"Threads are represented by the Thread class. The only way for a user to create a thread is to create an object of this class [could be indirectly]; each thread is associated with such an object". // JLS 17

Threads are low level, we will try to avoid the explicit usage of

- We'll mostly use higher level classes (that will create threads)
- ... but some knowledge needed...

Runnable

"The general contract of the method run is that it may take any action whatsoever' // Javadoc



Code to Run in Thread

The code to run in the thread is located in some class implementing the Runnable interface (only method run())

```
public class MyThreadClass implements Runnable {
    @Override
    public void run(){
        // Code to run in thread, when finished thread dies
    }
}
// Create and start thread to run the code
Thread t = new Thread(new MyClass());
t.start(); // Will execute code in run-method in parallel
           // with the thread we're in
```

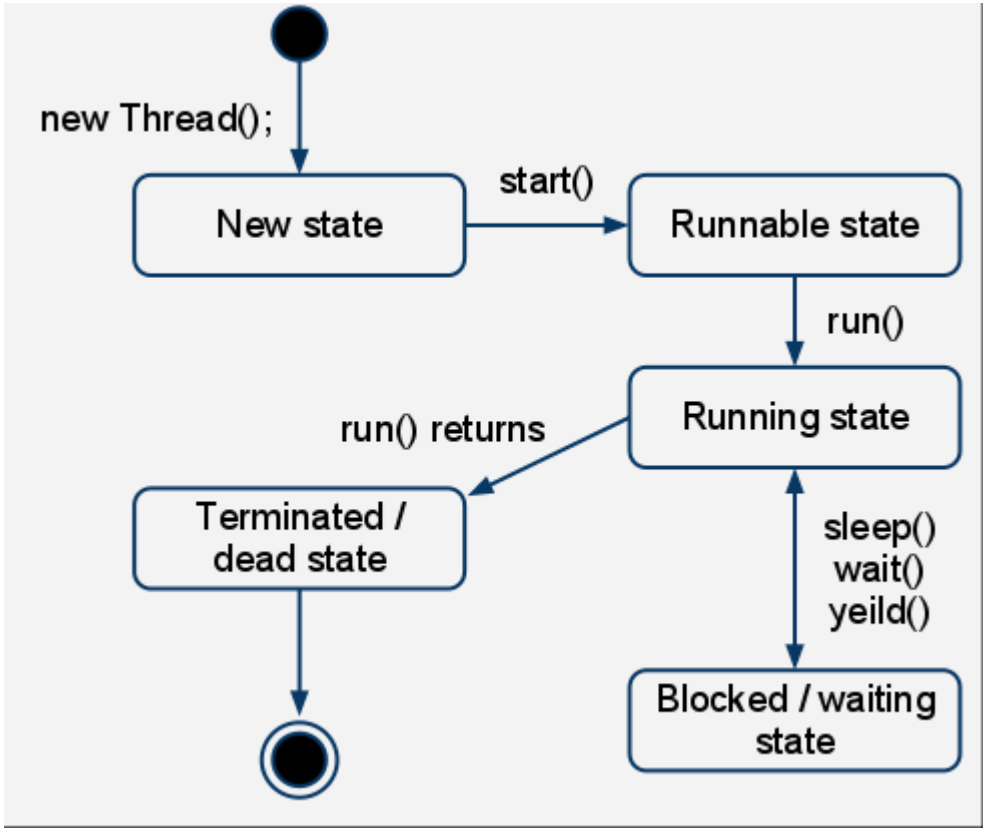
Example: Basic Thread*

```
// Starting three threads X, Y, Z implements Runnable
Thread t1 = new Thread(new X());
Thread t2 = new Thread(new Y());
t1.start();
t2.start();

// Z handles it's own thread (has Thread attribute)
new Z().start();

// t1.stop() is deprecated don't use
// To stop a thread you should break the loop in the
// run method (we don't need, we use higher level classes)
```

Thread Life Cycle*



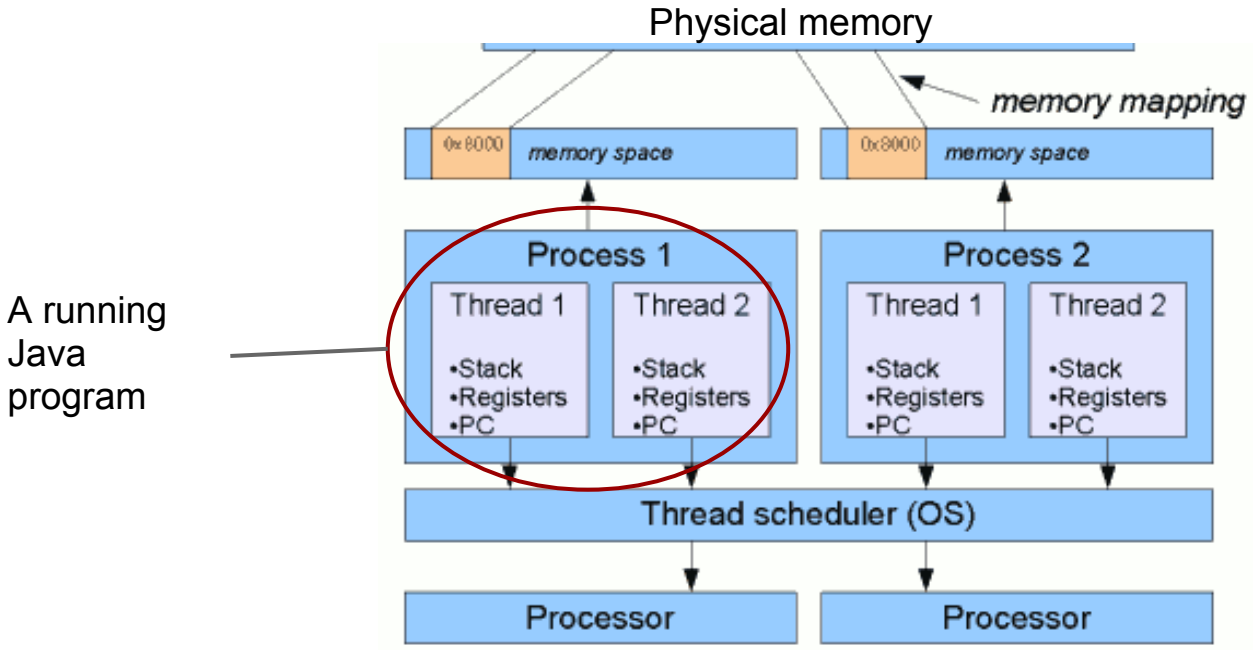
Thread Life Cycle, cont

- New state: this is when the Thread object is first created by calling the constructor of the Thread class. For example; Thread newThread = new (runnableObj);
- Runnable. After calling the start() method on the thread, it enters the runnable state. That means it is ready to run or to go to some other states.
- Running state: a thread is in this state while its run() method is executing. During this state, the thread can go to blocked state if it gets blocked or has to wait for other threads. When the run() method finishes, this state ends.
- Blocked/waiting state: while running a thread can be put to sleep, interrupted or blocked by other threads. There are many reasons why this may happen.
- Terminated /dead state: when a thread reaches this state, its life ends. It can never be revived again. Normally, a thread enters this state because its run() method has ended. However, a thread can also be terminated even before it is in the runnable state or during the waiting state by calling stop() or suspense() on the thread. I don't suggest using any of those methods to move a thread to its terminated state because those methods have been deprecated by Java and are not thread safe.

Threads and Portability

Threads scheduler is part of operating system (OS)

The behaviour of the thread scheduler, thread priorities, and Thread. yield are highly dependent on the Java Runtime implementation you happen to be using. You cannot rely on them to define the logic of your application (we will not use).



Atomic Operations

Some operations are by default guaranteed to be "atomic" i.e. a thread will be able to finish without any other thread interfering

- Single read/write of variable except long or double is atomic (there is a class AtomicLong,...)

```
// Is this atomic?
x++;
```

Race Conditions*

All threads have own stack so local variables no problem but...

... attributes shared

- What if many threads do non atomic reads/writes the same attribute concurrently?
- Can't specify which thread will run, done by system, imagine; random!
- Other benefit of immutable classes, they are thread safe!

Race Conditions, cont

Race conditions occur when multiple threads perform non-atomic operations (outcome depends of timing of threads)

```
// Possible race condition
int x = 10;
x = x + 1
```

Thread A	Thread B
read x (x == 10)	
	read x (x == 10)
x +1, write x (x == 11)	
	x + 1, write x (x == 11) **One update lost**

Monitors

"Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor. Any other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor. A thread t may lock a particular monitor multiple times; each unlock reverses the effect of one lock operation." // JLS 17

Synchronization

To avoid race condition we "synchronize" the critical code sections by locking/unlocking the monitor

- New keyword **synchronized,** use to "get the lock" (if fail thread put into wait-set)
- Used on statement or method
- synchronized will turn a method or code block into an atomic operation
- synchronized not part of method signature
- Constructors and initializers can't be synchronized

Synchronized Statements*

*"The synchronized statement (§14.19) computes a reference to an object; it then attempts to perform a lock action on **that object's monitor** and does not proceed further until the lock action has successfully completed. After the lock action has been performed, the body of the synchronized statement is executed. If execution of the body is ever completed, either normally or abruptly, an unlock action is automatically performed on that same monitor."*

// JLS 17

'Acquiring the lock associated with an object does not in itself prevent other threads from accessing fields of the object or invoking un-synchronized methods on the object".

// JLS 14.19

Synchronized Methods*

*"A synchronized method (§8.4.3.6) automatically performs a lock action when it is invoked; its body is not executed until the lock action has successfully completed. If the method is an instance method, it locks the monitor associated with the instance for which it was invoked (that is, the object that will be known as **"this"** during execution of the body of the method). If the method is static, it locks the monitor associated with the **Class object** that represents the class in which the method is defined. If execution of the method's body is ever completed, either normally or abruptly, an unlock action is automatically performed on that same monitor."*
// JLS 17

At un-lock scheduler choose any waiting thread to execute (assume random)

Changes to shared memory not visible to other thread util thread leaves synchronized method (or statement)

Composing Thread Safe Methods*

Synchronization doesn't compose

```
// Assume both methods synchronized
o.doIt();
o.doOther();    // The sequence isn't thread safe albeit
                // both methods are
```

Deadlock*

If using too much synchronization threads possible end up mutually waiting on each other; Deadlock!

- Never call out to other object from synchronized code, code (in other thread) possibly will call back, ... deadlock
- A single thread can call synchronized methods on same object from inside synchronized method

Other forms; Livelock, ...

Util Timer and TimerTask*

To perform some action at fixed interval running in own thread

- Implement a subclass of TimerTask. The run method contains the code that performs the task.
- Create an object of type java.util.Timer class
- Create an object of TimerTask sub class
- Use timer object to schedule the sub class object for execution
- To stop task: Invoke cancel on the timer

Utility methods in Thread*

Thread.sleep()

"Thread.sleep causes the currently executing thread to sleep (temporarily cease execution) for the specified duration, subject to the precision and accuracy of system timers and schedulers. The thread does not lose ownership of any monitors, and resumption of execution will depend on scheduling and the availability of processors on which to execute the thread."

//JLS 17

Thread.currentThread()

Returns a reference to the currently executing thread object.

Concurrent Collections*

Possible to delegate thread safety if methods use thread safe Collections

- ConcurrentHashMap - A highly concurrent, high-performance ConcurrentMap implementation based on a hash table. This implementation never blocks when performing retrievals and allows the client to select the concurrency level for updates. It is intended as a drop-in replacement for Hashtable: in addition to implementing ConcurrentMap, it supports all of the "legacy" methods peculiar to Hashtable.
- ConcurrentLinkedQueue , ConcurrentSkipListSet, ...

Have to read Javadoc carefully for the meaning of thread safe (have to synchronize iterators)

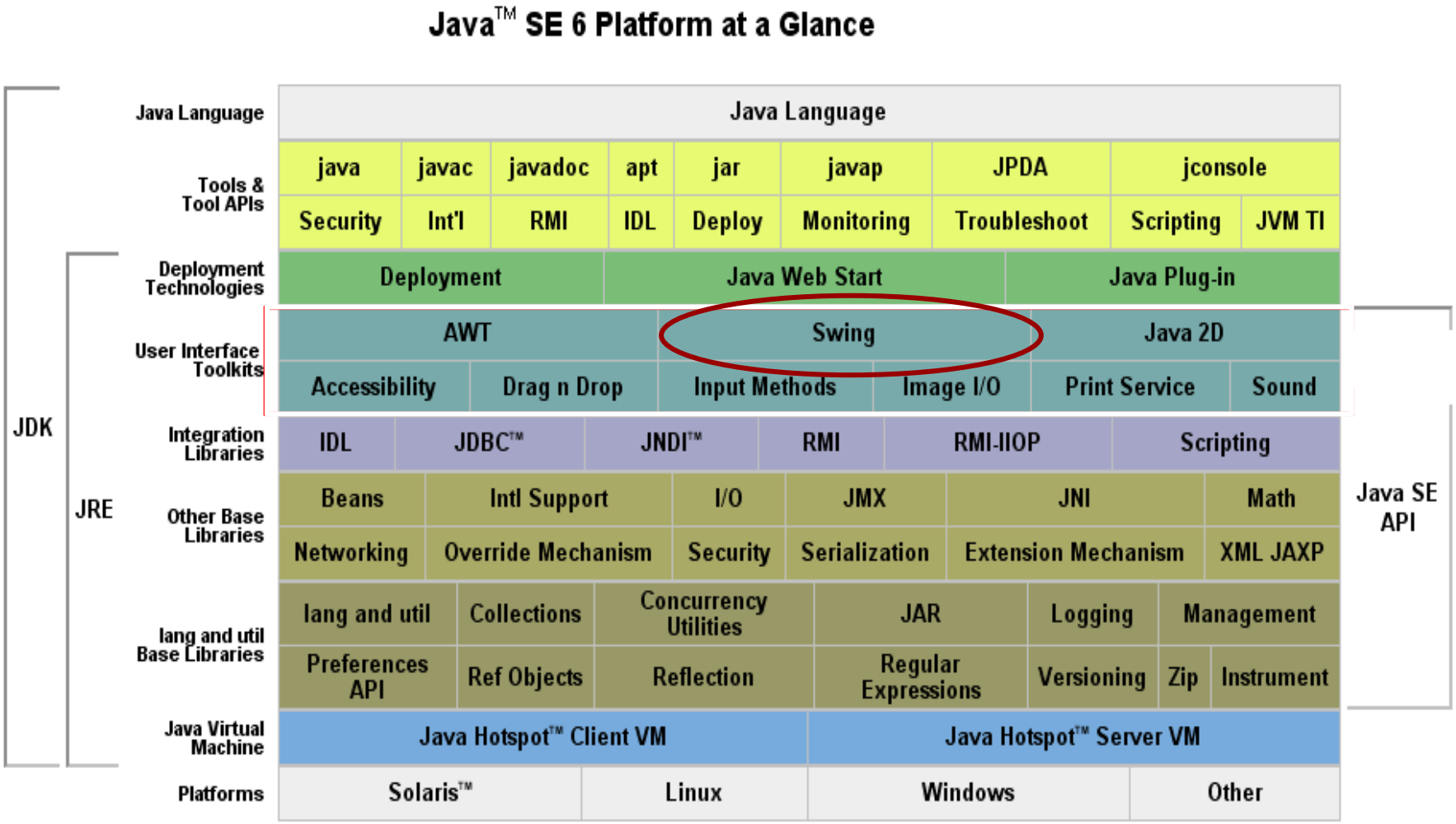
Java Swing

A GUI widget toolkit (GUI framework)

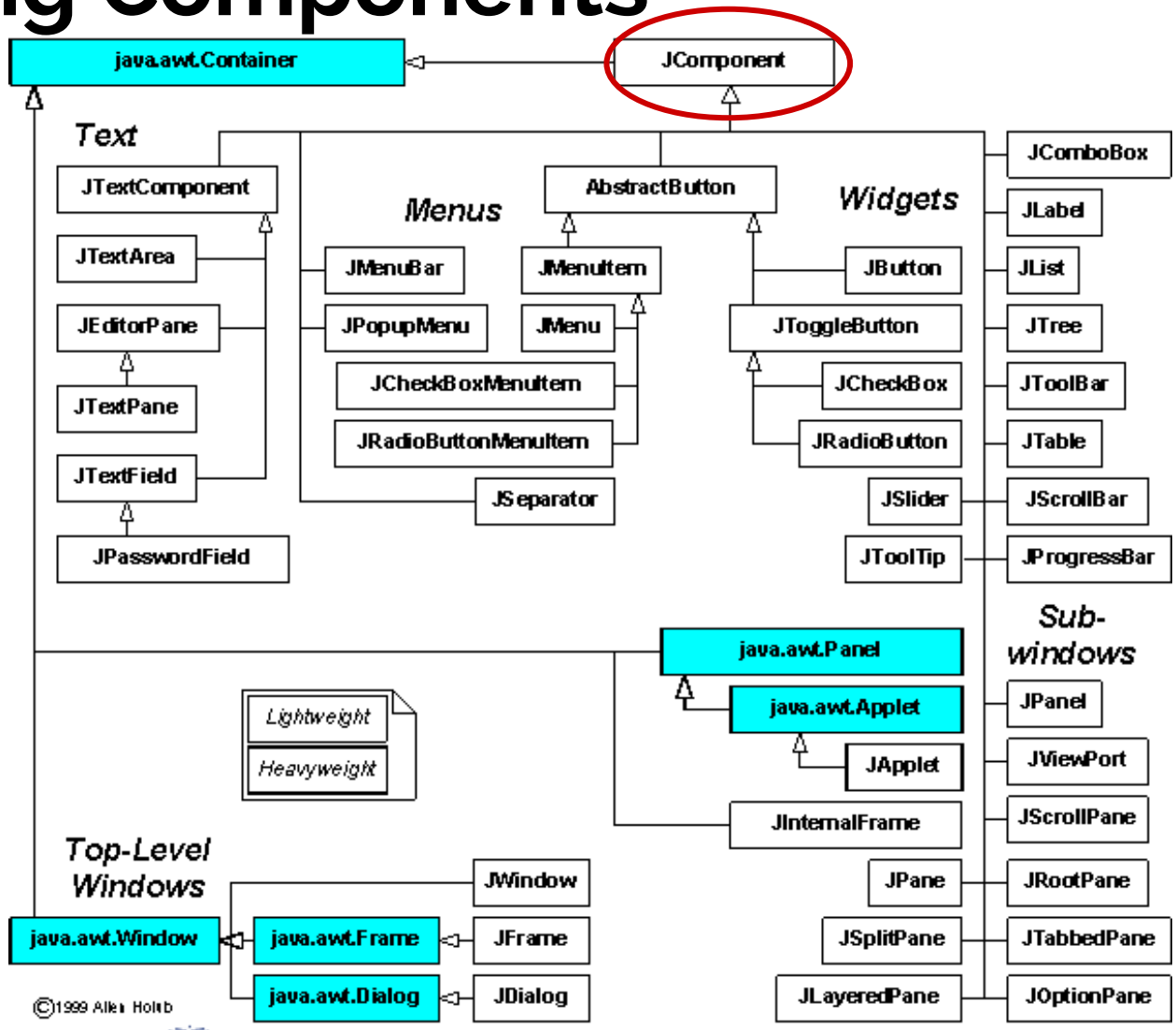
- Descendant of the Java Advanced Window Toolkit (AWT). Some things (like event handling) remains from AWT (java.awt.event.* package)
- Many components
- Uses an "internal" MVC architecture
- Pluggable look-n-feel
- Single threaded (not thread safe)
- Programming Swing is a subtopic in it's own right

There are other frameworks : SWT (used by Eclipse), SwingX, JavaFX, Apache Pivot, Qt Jambi, ...

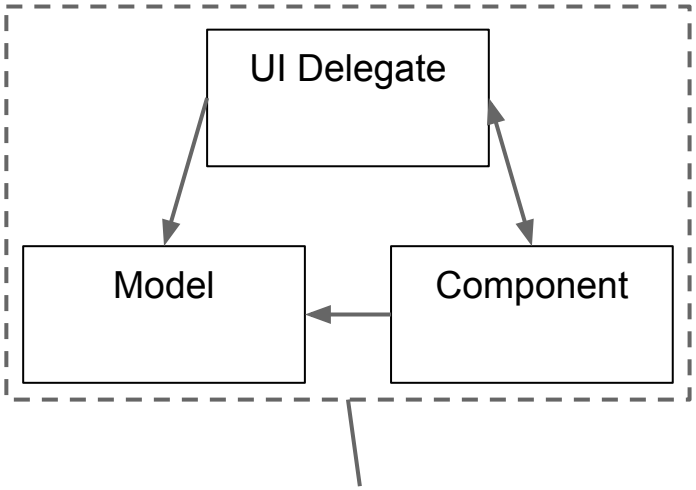
Aside: Swing is Part of the Java SE Platform (there's also Java EE, ...)



Swing Components



Swing Internal MVC*



Any JComponent is constructed like this

Use F3 in Eclipse to inspect

Model is the data for the component (customizable)

UI Delegate (class ComponentUI) is responsible for getting data from the Model and rendering it to the screen (can be replaced runtime, i.e. change look-n-feel)

Component generally coordinates the actions of the Model and Delegate, while also acting as glue to the AWT windowing system

Also: Complex components can use custom Renderers

Swing Model Interfaces

Component	Model Interface	Model Type
JButton	ButtonModel	GUI
JToggleButton	ButtonModel	GUI/data
JCheckBox	ButtonModel	GUI/data
JRadioButton	ButtonModel	GUI/data
JMenu	ButtonModel	GUI
JMenuItem	ButtonModel	GUI
JCheckBoxMenuItem	ButtonModel	GUI/data
JRadioButtonMenuItem	ButtonModel	GUI/data
JComboBox	ComboBoxModel	data
JProgressBar	BoundedRangeModel	GUI/data
JScrollBar	BoundedRangeModel	GUI/data
JSlider	BoundedRangeModel	GUI/data
JTabbedPane	SingleSelectionModel	GUI
JList	ListModel	data
JList	ListSelectionModel	GUI
JTable	TableModel	data
JTable	TableColumnModel	GUI
JTree	TreeModel	data
JTree	TreeSelectionModel	GUI
JEditorPane	Document	data
JTextPane	Document	data
JTextArea	Document	data
JTextField	Document	data
JPasswordField	Document	data

Swing Default Models*

Used default by corresponding component (implements previous interfaces)

- DefaultComboBoxModel, DefaultListModel DefaultTableModel, DefaultTreeModel, DefaultTableColumnModel, ...
- If creating complex components normally create custom model for efficiency and possible other issues (i.e. SpreadSheet)

Swing Components Listeners

Swing components uses Observer pattern

```
// In JComponent
protected EventListenerList listenerList = new EventListenerList();

// In AbstractButton extends JComponent
public void addItemListener(ItemListener l) {
        listenerList.add(ItemListener.class, l);
}
protected void fireStateChanged() {
   Object[] listeners = listenerList.getListenerList();
   for (int i = listeners.length-2; i>=0; i-=2) {
     if (listeners[i]==ChangeListener.class) {
       if (changeEvent == null)
            changeEvent = new ChangeEvent(this);
         ((ChangeListener)listeners[i+1]).stateChanged(changeEvent);
       }
   }
}
```

Swing Pluggable Look-n-feel*

Because of MVC-design for components it's possible to render components in very different styles.

- "Look" refers to the appearance of GUI widgets (more formally, JComponents) and "feel" refers to the way the widgets behave
- Some available L&F's
- Have to install look and feel on all computers..??

Blocking Calls*

Blocking calls

- A method called in a thread will block the thread until method finished (returns)
- If method very slow (long running task), rest of program have to wait (example: GUI will freeze if downloading big file)

Why is Swing Single Threaded?

"Multithreaded GUI frameworks tend to be particularly susceptible to deadlock, partially because of the 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 a mouse click is detected by the OS, is turned into a "mouse click" event by the toolkit, and is eventually delivered to an application listener as a higher level event such as a "button pressed" event. On the other hand, application-initiated actions "bubble down" from the application to the OS changing the background color of a component originates in the application and is dispatched to a specific component class and eventually into the OS for rendering. 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. And this is exactly what nearly every GUI toolkit development effort rediscovered through experience." //Web

Most (all?) GUI frameworks are single threaded: Qt, NextStep, MacOS Cocoa, X Window, Windows(?), ...

Swing Single Thread Rule

"Once a Swing component has been realized, all code that might affect or depend on the state of that component should be executed in the event-dispatching thread ***[=EDT = the Swing thread]****.*

Realized means that the component's paint [paintComponent] method has been or might be called. A Swing component that's a top-level window is realized by having setVisible(true), show, or pack called on it. Once a window is realized, all of the components that it contains are realized. Another way to realize a Component is to add it to a Container that's already realized."

This is "The Swing Single Thread Rule"

Swing Thread Handling

A Swing programmer must handle

- Initial threads, the threads that execute initial application code (thread created automatically)
- The event dispatch thread, where all event-handling code/painting is executed (thread created automatically)
- Worker threads, also known as background threads, where time-consuming background tasks are executed (created by us using higher level classes)

EDT: Create the GUI*

```
// Assume main-thread running here
```

```
// Causes run() to be executed in the EDT (after any
pending // events)
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        createAndShowGUI();
    }
});
// main thread continues before call returned
(asynchronous)
```

```
// If using SwingUtilities.invokeAndWait main waits until
// run finished
```

InvokeLater vs InvokeAndWait

- InvokeLater is non blocking call while InvokeAndWait will block until task is completed.
- If run method of Runnable target throws an Exception then in case of invokeLater EDT threads unwinds while in case of invokeAndWait exception is caught and rethrown as InvocationTargetException.
- InvokeLater can be safely called from Event Dispatcher thread while if you call invokeAndWait from EDT thread you will get an error because as per java documentation of invokeAndWait it clearly says that "this request will be processed only after all pending events" and if you call this from EDT this will become one of pending event so its a deadlock because caller of InvokeAndWait is waiting for completion of invokeAndWait while EDT is waiting for caller of InvokeAndWait.
- InvokeLater is more flexible in terms of user interaction because it just adds the task in queue and allow user to interact with system while invokeAndWait is preffered way to update the GUI from application thread.

EDT: Event Handling*

EDT uses simple queue for events, any event will be processed in incoming order

- If many events queue may filled up
- If slow handling of events queue may fill up

Event Handling should be as quick as possible

- If slow use a worker thread, more later...

EDT: Painting*

There is a default painting mechanism ... if not satisfied override paintComponent()

- Parameter is: Graphics g, a graphical toolbox
  - Methods, drawLine(), drawRectangle(), setColor(), fillRect(), ...
- Sometimes possible need to call repaint() or validate() to force painting(?)

The Graphics Object

"A Graphics object encapsulates state information needed for the basic rendering operations that Java supports. This state information includes the following properties: The Component object on which to draw, translation origin for rendering and clipping coordinate, the current clip, the current color, the current font, the current logical pixel operation function (XOR or Paint), the current XOR alternation color"

Javadoc for paintComponent()

"If you override this (paintComponent) in a subclass you should not make permanent changes to the passed in Graphics (i.e. not use it as an out parameter). For example, you should not alter the clip Rectangle or modify the transform. If you need to do these operations you may find it easier to create a new Graphics from the passed in Graphics and manipulate it."

Swing GUI Animations*

If cool GUI is on wishlist...

- ...combine paintComponent(Graphics g)..
- with a Swing timer (will run in EDT)

Note: Games are usually not implemented like this.

Worker Threads*

"SwingWorker is designed for situations where you need to have a long running task run in a background thread and provide updates to the UI either when done, or while processing." // Javadoc

Use javax.swing.SwingWorker<T>

Worker Threads, cont

There are three threads involved in the life cycle of a SwingWorker :

- *Current* thread: The execute() method is called on this thread. It schedules SwingWorker for the execution on a *worker* thread and returns immediately. One can wait for the SwingWorker to complete using the get methods.
- *Worker* thread: The doInBackground() method is called on this thread. This is where all background activities should happen.
- *Event Dispatch Thread*: All Swing related activities occur on this thread. SwingWorker invokes the process and done() methods and notifies any PropertyChangeListeners on this thread.

Often, the *Current* thread is the *Event Dispatch Thread*.

Switching Threads*

From GUI to backend and, if needed, back to GUI

new SwingWorker(){...}

From backend to GUI (network threads)

SwingUtilities.invokeLater(...)

Summary

Using threads and related problems; synchronization race conditions, deadlock

Usage of threads in Java Swing applications