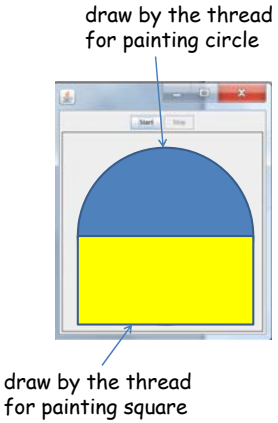
Concurrency in Java Swing

Event-dispatching thread

- By default all AWT and Swing-based applications start off with two threads.
  - One is the main application thread which handles execution of the main() method.
  - The other, referred to as the *event-dispatching thread,* is responsible for handling events, painting, and layout.
- All events are processed by the listeners that receive them within the event-dispatching thread.
  - the code you write inside the body of an actionPerformed() method is executed within the event-dispatching thread automatically

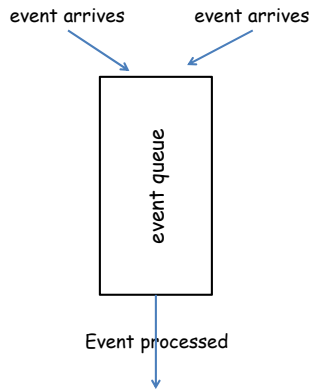
Updating Component

- All updates to any component's state should be executed from within the event-dispatching thread only
- If you have multiple threads updating the states of the components at the same time, the components might end up in an inconsistent state, e.g. half of the panel is repainted according to the instructions from one thread while the other half might be drawn according to the instructions from another thread



system event queue

- Associated with the event-dispatching thread is a FIFO (first in first out) queue of events called the *system event queue* (an instance of java.awt.EventQueue).
- The queue gets filled up in a serial fashion.
- Each request takes its turn executing event-handling code.
  - All events are processed serially



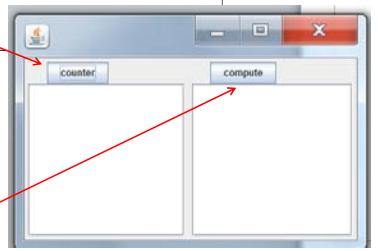
Make GUI responsive

- The event-dispatching thread executes all listener methods, painting and layout.
- It is important that event-handling, painting, and layout methods be executed quickly.
- Otherwise, the whole system event queue will be blocked waiting for one event process to finish
  - Your application will appear to be frozen or locked up.

5

```
counterButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
                // do work for counterButton
        }
    });
```

After the compute button is clicked, the GUI becomes un-responsive for at least 5 seconds




```
computeButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
                // do work for computeButton
                Thread.sleep(5000); // have a rest
        }
    });
```

6

Achieve responsiveness through Multithreading

- To ensure that event handling methods get executed quickly, we need to use separate thread for doing time-consuming work.
  - Do the time-consuming work first
  - Then, add the event for updating the component, e.g. repaint, to the system event queue to ask the event-dispatching thread to process the event
- Swing provides a class called SwingWorker that makes managing the multithreading of the time-consuming tasks easy
- Read Java tutorial's Concurrency in Swing http://docs.oracle.com/javase/tutorial/uiswing/index.html

7

Define your own worker class

- If the handling of a GUI event takes a long time to complete, you should create a worker object to carry out the task in a thread other than the Event Dispatch Thread
- To take advantage of the thread creation and scheduling facility available in Java, you should make you worker class a subclass of Java's SwingWorker class.
  - You need to override several methods of the SwingWorker class to make your worker class behave in the way that you expect

8

```
class MySwingWorker extends SwingWorker<XXX> {

    @Override
     public XXX doInBackground() {
       …
      }

    @Override
     protected void process(XXX) {
       …
     }

    @Override
     protected void done() {
       …
     }
}
```

9

Executing a SwingWorker object

- A SwingWorker object is created when you need to handle a time consuming GUI event.
- The execution of a SwingWorker object normally involves two or more threads
- Event Dispatch Thread
  - creates a SwingWorker object and registers the object with the runtime
  - Execute some operations to update the GUI after the SwingWorker object completes its execution
- Worker thread
  - The runtime assigns a worker thread to carry out the execution of the time-consuming tasks that the SwingWorker object wants to perform

10

The execute method

- This method is used to register the SwingWorker object with the JVM. JVM schedules SwingWorker object for execution on a thread
- This method returns immediately.
- This method is normally called by the Event Dispatch Thread

```
public void actionPerformed(java.awt.event.ActionEvent evt) {
        // create a worker object
        MySwingWorker msw = new MySwingWorker();
        // start the worker object
        msw.execute();
}
```

11

The doInBackground method

- This method is called by the runtime and runs in the worker thread. This is where all background activities happen.
- This method can return a value. If the method returns a value, the value can be retrieved by calling the get method on the worker object.

```
@Override
public Double doInBackground() {
        Double x;
        // do a lot of work to generate the value for x
        return x;
}
```

12

The get method

- This method is used to obtain the value produced by the doInBackground method.
- If the doInBackground method has not terminated, this method blocks the execution of the thread that calls the get method.
  - If it is called without any argument, the thread wait until the doInBackground method terminate.
    - get()
  - It can be called with arguments specifying the timeout period for waiting the result.
    - get(50L, TimeUnit.MILLISECONDS): wait for at most 50 milliseconds

13

The done method

- When the doInBackground method terminates, the done method will be executed by the Event Dispatch Thread
- In this method, you specify the operations for updating the GUI components, e.g. write the result produced by the doInBackground method to a textfield.

```
@Override
protected void done() {
        Double value = super.get();
        computeText.append(value.toString()+"\n");
        computeText.append("done "+"\n");
}
```

14

The publish method

- Some applications might want to display the intermediate results produced by the worker thread while executing doInBackground.
- This method is called from inside the doInBackground method to deliver intermediate results for processing on the *Event Dispatch Thread*
- This method is final.

```
@Override
public Double doInBackground() {
        while condition {
                … // code for computing z
                publish(new Double(z));
        }
}
```

15

The process method

- This method is executed by the *Event Dispatch Thread*.
- It is used to process the data chunks received asynchronously from the publish method.
  - Update the GUI components.
  - The parameter is a collection of the type of objects sent by the publish method
- It is executed in the *Event Dispatch Thread*.

```
@Override
protected void process(java.util.List<Double> numbers) {
        // only display the last one in the list
        Double num = numbers.get(numbers.size() - 1);
        computeText.append(num.toString()+"\n");
}
```

16

The cancel method

- This method is used to cancel the background task
- This method changes the cancellation status of the object to true.
  - The background task can check its cancellation status by calling the isCancelled method.
- If it is called with true as an argument, an interrupt is sent to the background task.
- This is a final method.

17

Cancelling Background Tasks

- If you want the background task to be cancelled before it terminates normally, the background task must be programmed to cooperate with the cancelling event
- There are two ways for the background task to do this
  - terminate when it receives an interrupt
  - Invoking SwingWorker.isCancelled at short intervals. This method returns true if cancel has been invoked for this SwingWorker.

18

Cancelling by interrupt

```
public void actionPerformed(ActionEvent e) {
      if ("Start" == e.getActionCommand()) {
        msw = new MySwingWorker();
        msw.execute();
      } else if ("Stop" == e.getActionCommand())
        msw.cancel(true);
  }
```

```
@Override
public void doInBackground() {
        while (true) {
                if (Thread.interrupted()) return null;
                … // code for computing z
                publish(new Double(z));
        }
}
```

19

Cancelling by checking status

```
public void actionPerformed(ActionEvent e) {
      if ("Start" == e.getActionCommand()) {
        msw = new MySwingWorker();
        msw.execute();
      } else if ("Stop" == e.getActionCommand())
        msw.cancel(false);
  }
```

```
@Override
public void doInBackground() {
        while (!isCancelled()) {
                … // code for computing z
                publish(new Double(z));
        }
        return null;
}
```

20

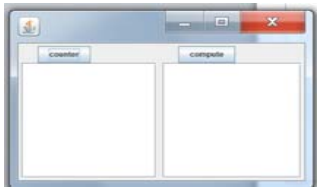
Revisit your worker class

class MySwingWorker extends SwingWorker<Double, Double>

- SwingWorker is a generic class, with two type parameters.
- The first type parameter specifies a return type for doInBackground,
  - also the type for the get method
- SwingWorker's second type parameter specifies a type for interim results returned while the background task is still active.

21

- The "compute" button's action needs a long time to process. The task is implemented as a subclass of SwingWorker.



```
counterButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
            counter++;
            counterText.setText(counter + "");
        }
    });
```

```
computeButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
            MySwingWorker msw = new MySwingWorker();
            msw.execute();
        }
    });
```

22

```
class MySwingWorker extends SwingWorker<Double, Double> {

    public Double doInBackground() {
        ...
        while (i > 0) {
            ...
            publish(new Double(z));
        }
        return new Double(z);
    }

    protected void process(java.util.List<Double> numbers) {
        Double num = numbers.get(numbers.size() - 1);
        computeText.append(num.toString()+"\n");
    }

    protected void done() {
            ...
        computeText.append("done "+computeCount+"\n");
    }
}
```

23

- When the "compute" button is clicked, a SwingWorker type object is created and is registered with runtime.
- The runtime creates a worker thread to run the doInBackground method of the SwingWorker object.
- When the publish method is called by the SwingWorker object, a event is added to the system event queue.
  - The event-dispatch thread process the event of the SwingWorker object by executing the process method of the SwingWorker object.
  - The process method does not contain any long-running computation. Thus, it won't freeze the GUI.
- When the "counter" button is clicked, an event is added to the system event queue.
  - The event will be processed by the event-dispatch thread.
- When the SwingWorker object's doInBackground method terminates, an event is added to the system event queue.
  - The event will be processed by the event-dispatch thread by executing the done method of the SwingWorker object.

24

Inversion of control

- When using SwingWorker, it can be seen that the time for executing the code of a SwingWorker object is not determined by the programmer.
  - The programmer just needs to specify the code that need to be executed and register the SwingWorker object with the runtime
    - Override the doInBackground, process and done method.
    - Call the execute method.
- The runtime schedules the execution of the code.
- This kind of design pattern is called inversion of control.
  - You specify what to do
  - Someone else (e.g. the runtime, another object) controls when the code is executed

25

reviews

- By default, how many threads does the Java runtime use to execute a Swing application? What are these threads called? What tasks does each of the threads do?
- Why is it not a good idea to use multithreading to update the states of the GUI component simultaneously?
- The event-dispatching thread uses a system event queue to hold the event to be processed. What is the property of the queue? How does the property affect the responsiveness of the GUI?
- Use an example to explain why handling the event generated by a GUI component might make the GUI appear frozen? What do you need to do to avoid this from happening?
- Understand how to define your worker class as a subclass of the SwingWorker class, e.g. the meaning of the parameters, the methods that need to be overridden, etc.
- Understand how the methods of a SwingWorker object are executed.

26