



JavaOne

JSR-296: The Swing Application Framework

Hans Muller, Josh Marinacci

Swing Engineering
Sun Microsystems
<http://appframework.dev.java.net>

TS-3942



Learn Enough to `Application.launch()`!

Write less, finish more

Java™ Specification Request (JSR)-296,
Swing Application Framework: how it
started, what it is, how to use it, why you
want it

Agenda

How Did We Get Here? The Back Story

The Entire Application Framework API

Summary, Plans, Futures, Outstanding
Issues, Things You're Not Likely to
Remember

A Demo!

Helpful Links, Q&A

Agenda [font size reflects duration]

How did we get here? The back story

The entire Application Framework API

Summary, plans, futures, outstanding issues, things you're not likely to remember

A Demo!

Helpful links, Q&A

Agenda

How Did We Get Here? The Back Story

The Entire Application Framework API

Summary, Plans, Futures, Outstanding
Issues, Things You're Not Likely to
Remember

A Demo!

Helpful links, Q&A

How Did We Get Here?

- Swing version 0.1 debuts in 1997
- It's widely adopted
- Developers let us know that building Swing applications can be difficult. They tell us that a standard application framework would make the job easier.

How Did We Get Here? (Cont.)

- More than 8 years pass

How Did We Get Here? (Cont.)

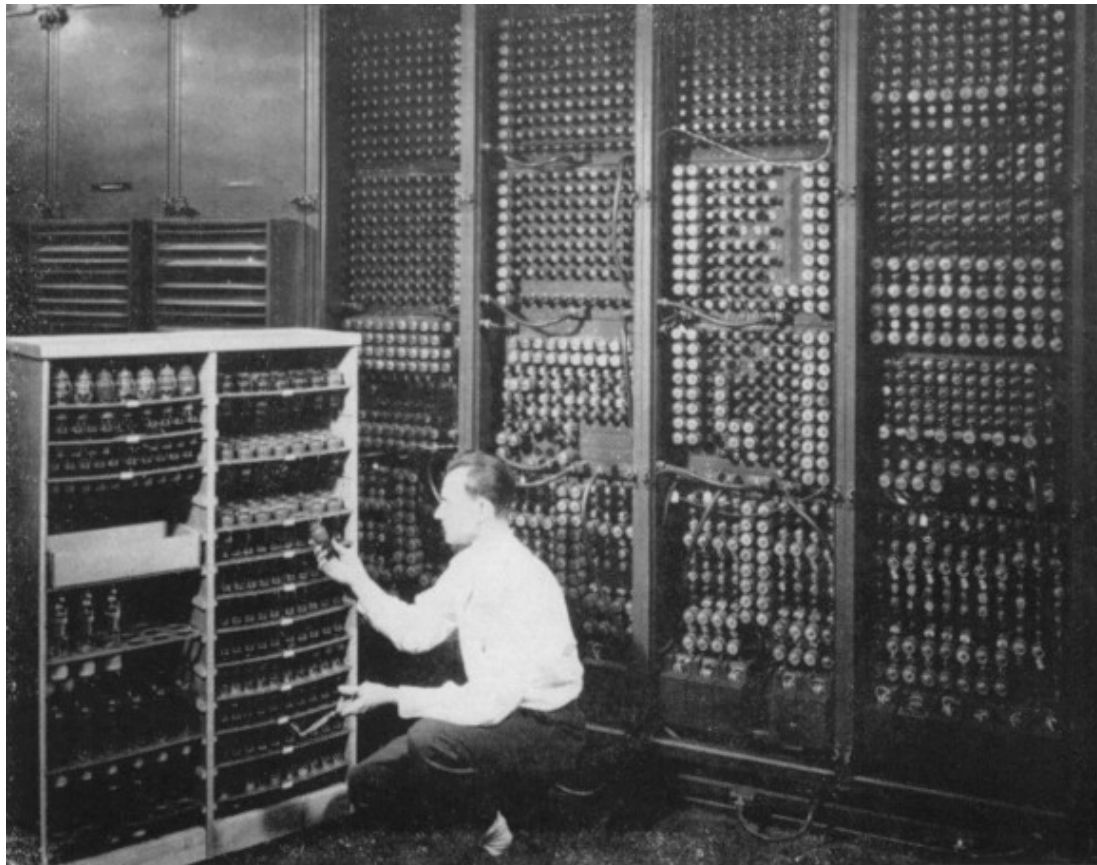
- Late Spring 2006
- Swing Team files two JSRs aimed at simplifying building applications:
 - JSR 295 Beans Binding
 - JSR 296 Swing Application Framework
- Both JSRs
 - Focus on issues common to typical Swing applications
 - Are intended for Java Platform, Standard Edition (Java SE platform) SE 7
 - Have prototypes available, java.net projects

This Presentation Is About JSR-296

- An Application Framework for Swing

But Aren't Application Frameworks Giant Scary Monsters?

Can be too much frame, not enough work



Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

Not Scary



- Swing Application Framework goals
 - As small and simple as possible (not more so)
 - Explain it all in one hour
 - Work very well for small/medium apps
- No module system, integral docking framework, generic data model, scripting language, GUI markup schema

Agenda

How Did We Get Here? The Back Story

The Entire Application Framework API

Summary, Plans, Futures, Outstanding Issues, Things You're Not Likely to Remember

A Demo!

Helpful links, Q&A

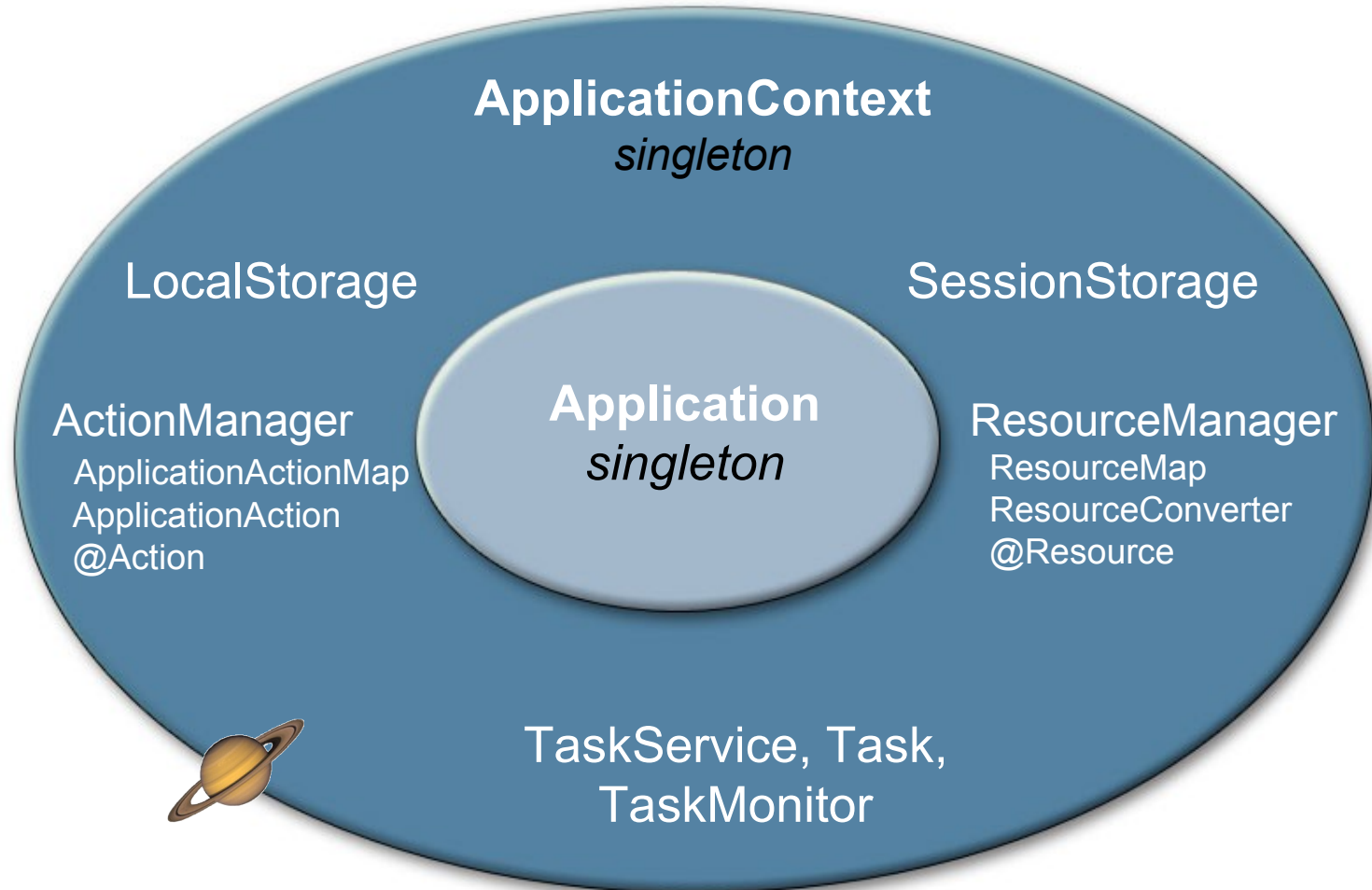
Disclaimer

- This is a review of my prototype
- The details will almost certainly change
- The fundamentals could change too

What the Framework Does

- **Lifecycle**
- Resources
- Actions
- Tasks
- Session state

Real Framework Architecture



Using the Framework

- Create a subclass of Application
- Create and show your GUI in the **startup** method
- Use **ApplicationContext** services to
 - define/manage actions and tasks
 - load/inject resources
 - save/restore session state
- Call **Application.launch** from your main method

Using the Framework: Hello World

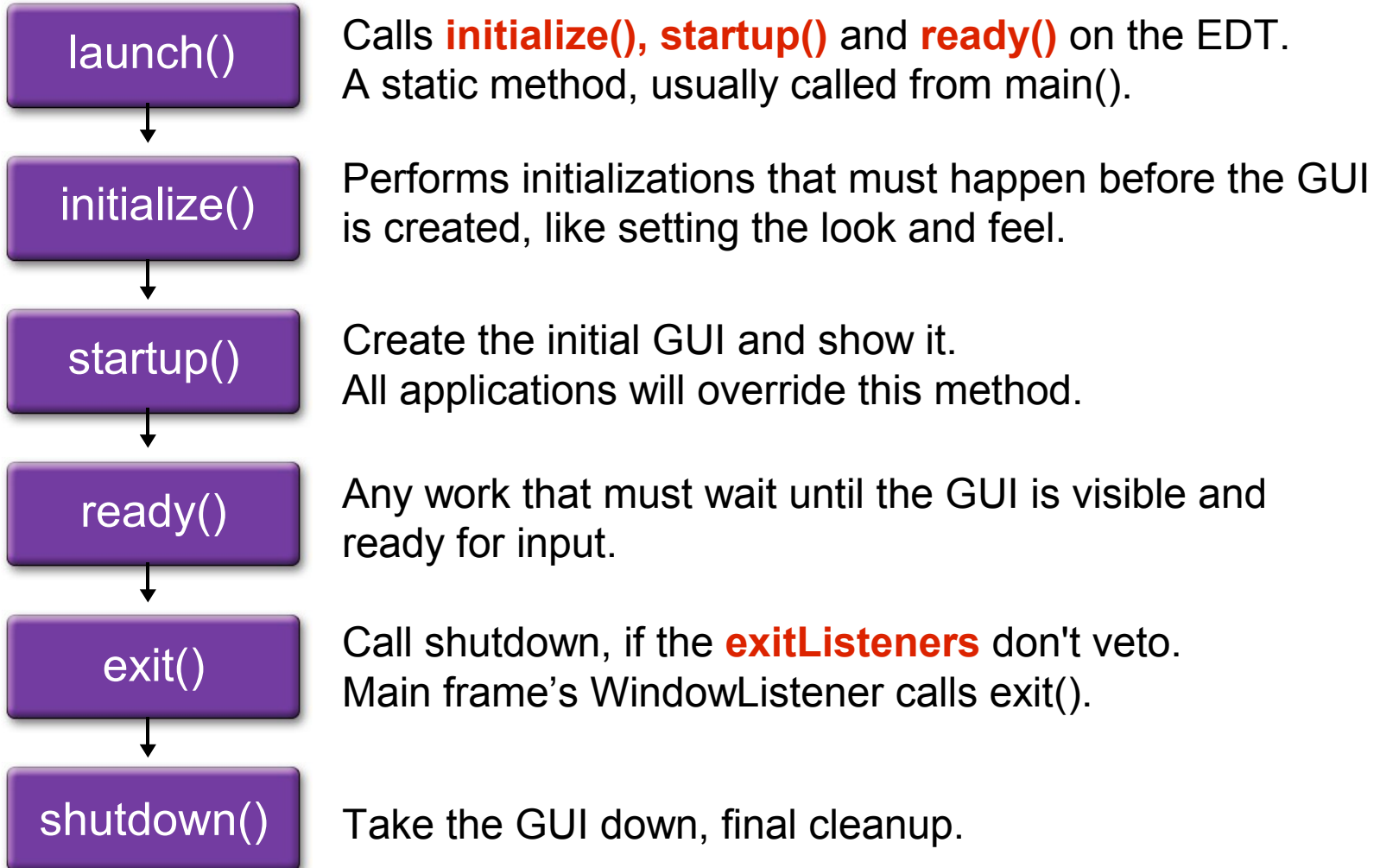
```
public class MyApp extends SingleFrameApplication {  
    @Override protected void startup() {  
        JLabel label = new JLabel("Hello World");  
        show(label);  
    }  
    public static void main(String[] args) {  
        Application.launch(MyApp.class, args);  
    }  
}
```

// SingleFrameApplication is a subclass of Application

Application Class

- Subclass defines your application's lifecycle
- Most apps will start with a class that extends Application, like SingleFrameApplication
- Subclasses override lifecycle methods
 - Application instance is constructed on the EDT
 - Lifecycle methods run on the EDT (except launch)
- Most lifecycle methods do little by default
 - ...other than being called at the appropriate time

Application Lifecycle Methods



May I Exit? Application Exit Listeners

- To quit the app, call `exit()`
- The `exit()` method checks `exitListeners` first

```
public interface ExitListener extends EventListener {  
    boolean canExit(EventObject e);  
    void willExit(EventObject e);  
}
```

- If they all return true:
 - call `Application.shutdown()`
 - `System.exit()`

ExitListener Example

```
class MaybeExit implements Application.ExitListener {
    public boolean canExit(EventObject e) {
        JFrame f = getMainFrame();
        String s = "Really Exit?"; // *ResourceMap
        int o = JOptionPane.showConfirmDialog(f, s);
        return o == JOptionPane.YES_OPTION;
    }
    public void willExit(EventObject e) { }
}
```

```
Application app = Application.getInstance();
app.addExitListener(new MaybeExit());
```

What the Framework Does

- Lifecycle
- **Resources**
- Actions
- Tasks
- Session state

Application Resources

- Defined with ResourceBundles
- Organized in resources subpackages
- Used to initialize properties specific to:
 - locale
 - platform
 - a few related values...

ResourceMaps

- Encapsulate list of ResourceBundles whose names are based on a class:
 - Generic ResourceBundle; just the class name
 - Per OS platform, class_os, e.g., MyForm_OSX
- Automatically parent-chained
 - Package-wide resources
 - Application-wide resources
- Support **string to type** resource conversion
 - Extensible

Getting a ResourceMap

- Resources for `mypackage/MyForm.java`
 - `mypackage/resources/MyForm.properties`
 - ~~`mypackage/resources/PackageResources.properties`~~
 - `mypackage/resources/MyApp.properties`
- ResourceMap for MyForm:



```
ApplicationContext c = ApplicationContext.getInstance();
```

```
ResourceMap r = c.getResourceMap(MyForm.class);
```

Using ResourceMaps: Example

```
# resources/MyForm.properties
  aString = Just a string
  aMessage = Hello {0}
  anInteger = 123
  aBoolean = True
  anIcon = myIcon.png
  aFont = Arial-PLAIN-12
  colorRGBA = 5, 6, 7, 8
  color0xRGB = #556677
```

```
ApplicationContext c = ApplicationContext.getInstance();
ResourceMap r = c.getResourceMap(MyForm.class);
```

```
r.getString("aMessage", "World") => "Hello World"
r.getColor("colorRGBA") => new Color(5, 6, 7, 8)
r.getFont("aFont") => new Font("Arial", Font.PLAIN, 12)
```

Resource Injection

`ResourceMap.injectComponents(Component root)`
`ResourceMap.injectComponent(Component target)`

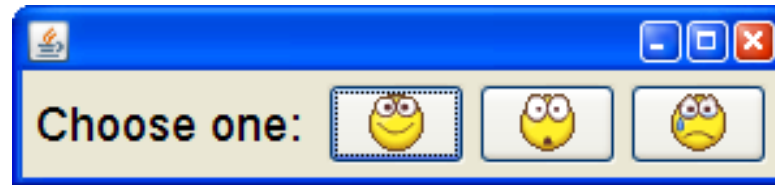
- Initialize properties from like-named resources
 - `myPanel.setForeground(Color c)`
 - `myLabel.setIcon(Icon i)`

`ResourceMap.injectFields(Object target)`

- Initialize marked fields from like-named resources
 - `@Resource Color foreground;`
 - `@Resource Icon icon;`

Resource Injection Example

- `resourceMap.injectComponents(myPanel)`



`component.getName():`

label

button1

button2

button3

```
# resources/MyPanel.properties
label.text = Choose one:
label.font = Lucida-PLAIN-18
button1.icon = smiley.gif
button2.icon = scared.gif
button3.icon = sad.gif
```

Resource Injection Advantages

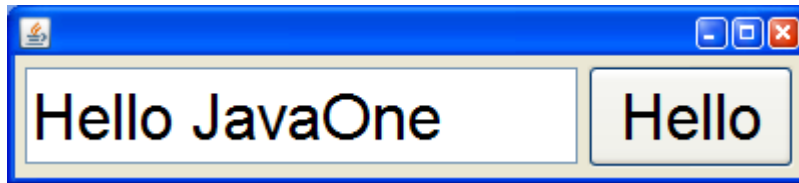
- Localizable by default
- No need to explicitly lookup/set resources
 - Easy to reconfigure visual app properties
 - review visual app properties
- But:
 - not intended to be a “styles” mechanism
 - not intended for general purpose GUI markup
 - @Resource injection requires privileges

Resource Injection in the SingleFrameApplication class

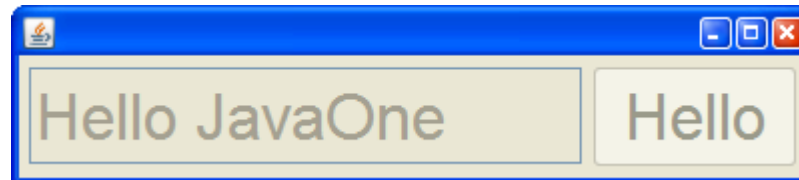
```
public class MyApp extends SingleFrameApplication {
    @Override protected void startup() {
        JLabel label = new JLabel();
        label.setName("label");
        show(label); // creates JFrame "mainFrame"
    }
    public static void main(String[] args) {
        Application.launch(MyApp.class, args);
    }
}
```

```
# resources/MyApp.properties
label.text = Hello World
label.font = Lucida-PLAIN-64
mainFrame.title = Hello
```

The sayHello Action in action



- Disable the sayHello Action:
`sayHello.setEnabled(false);`



What the Framework Does

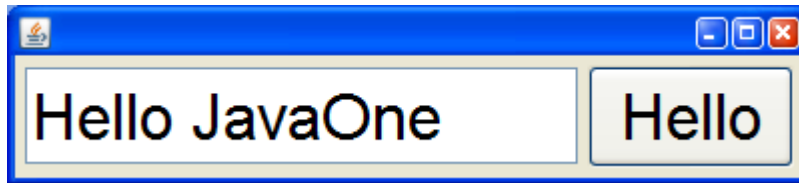
- Lifecycle
- Resources
- **Actions**
- Tasks
- Session state

Actions: A (very) brief review

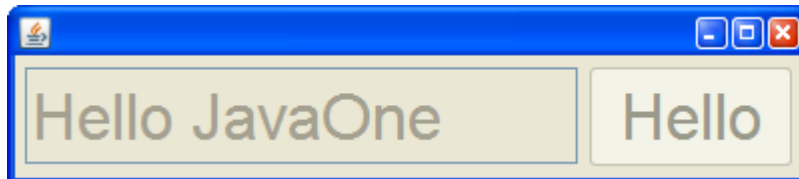
- Encapsulation of an ActionListener and:
 - some purely visual properties
 - enabled and selected boolean properties

```
// define sayHello Action – pops up message Dialog
Action sayHello = new AbstractAction("Hello") {
    public void actionPerformed(ActionEvent e) {
        String s = textField.getText();
        JOptionPane.showMessageDialog(s);
    }
};
// use sayHello – set the action property
textField.setAction(sayHello);
button.setAction(sayHello);
```

The sayHello Action in Action



- Disable the sayHello Action:
`sayHello.setEnabled(false);`



Actions: What We Like

- Encapsulation of default GUI + behavior
- The enabled and selected properties
- Reusability

What We're Not So Happy About

- Overhead: creating Action objects is a pain
- Visual properties should be localized!
- Asynchronous Actions are difficult
- Proxy linkages can be messy
- It's tempting to make a little spaghetti:
 - Backend logic that depends on Actions: find all the actions you need to enable/disable

The new `@Action` annotation

```
// define sayHello Action – pops up message Dialog
@Action public void sayHello() {
    String s = textField.getText();
    JOptionPane.showMessageDialog(s);
}
// use sayHello – set the action property
Action sayHello = getAction("sayHello");
textField.setAction(sayHello);
button.setAction(sayHello);
```

- ActionEvent argument is optional
- Used to define a “sayHello” ActionMap entry
- Encapsulation of default GUI + behavior

Where did the Action come from?

// utility method: look up an action for this class

```
Action getAction(String name) {  
    ApplicationContext c = ApplicationContext.getInstance();  
    ActionMap actionMap = c.getActionMap(getClass(), this);  
    return actionMap.get(name);  
}
```

- `ApplicationContext.getActionMap()`
 - creates an Action for each `@Action` method
 - default key is the action's method name
 - creates and caches an ActionMap

Action Resources

- Loaded from the target class's ResourceMap

```
# resources/MyForm.properties
```

```
sayHello.Action.text = Say &Hello  
sayHello.Action.icon = hello.png  
sayHello.Action.accelerator = control H  
sayHello.Action.shortDescription = Say hello modally
```

```
public class MyApp extends SingleFrameApplication {
    @Action public void sayHello() {
        JLabel label = new JLabel();
        label.setName("label");
        show(JOptionPane.createDialog(label));
    }
    @Override protected void startup() {
        show(new JButton(getAction("sayHello")));
    }
    public static void main(String[] args) {
        Application.launch(MyApp.class, args);
    }
}
```

```
# resources/MyApp.properties
sayHello.Action.text = Say&Hello
sayHello.Action.shortDescription = say hello
label.text = Hello World
mainFrame.title = Hello
```

@Action enabled/selected linkage

- @Action parameter names bound property
 - The rest of the app depends on the property, not the Action object
- @Action(enabledProperty = “name”)
- @Action(selectedProperty = “name”)

```
// Defines 3 Actions: revert, save, delete
```

```
public class MyForm extends JPanel {
    @Action(enabledProperty = "changesPending")
    public void revert() { ... }
```

```
    @Action(enabledProperty = "changesPending")
    public void save() { ... }
```

```
    @Action(enabledProperty = "!selectionEmpty")
    public void delete() { ... }
```



```
// These properties are bound, when they change
// PropertyChangeEvents are fired
public boolean getChangesPending() { ... }
public boolean isSelectionEmpty() { ... }
```

```
// ...
}
```

Enabled Property: Beans Binding

```
public MyHomePanel {  
    @Action(enabledProperty = "purchaseEnabled")  
    public void purchase() { ... }  
}
```

```
public boolean isPurchaseEnabled() { }  
public void setPurchaseEnabled(boolean b) { }  
}
```

```
new Binding(  
    aHomeListing, "${available}",  
    myHomePanel, "purchaseEnabled").bind();
```

```
new Binding(  
    homeListingTable, "${selectedElement.available}",  
    myHomePanel, "purchaseEnabled").bind();
```

One Action, Multiple Looks



Note: all three buttons share the sayHello Action

```
# resources/MyForm.properties  
sayHello.Action.text = Say Hello  
sayHello.Action.icon = hello.png
```

```
button2.text = ${null}  
button3.icon = ${null}
```

- Override Action's visual properties
 - action resource is set first
 - other resources override action's visuals
- Common case: Menu/Toolbar/Button

What the Framework Does

- Lifecycle
- Resources
- Actions
- **Tasks**
- Session state

Don't Block the EDT

- Use a background thread for
 - Computationally intensive tasks
 - Tasks that might block, like network or file IO
- Background thread monitoring/management
 - Starting, interrupting, finishing
 - Progress
 - Messages
 - Descriptive information
- SwingWorker: most of what we need

Tasks Inherit the SwingWorker API

- Task *isa* SwingWorker *isa* Future
 - Futures compute a value on thread
 - They can be canceled/interrupted
- SwingWorker adds:
 - EDT done() and PropertyChange methods
 - Publish/process for incremental results
 - Progress property percent complete
- Tasks add...

Tasks: Support for Monitoring

- Task title, description properties
 - For users (humans)
 - Initialized from ResourceMap
- Task message property, method
 - `myTask.setMessage("loading " + nThings)`
 - `myTask.message("loadingMessage", nThings)`
 - (resource) `loadingMessage = loading {0} things`
- Task start/done time properties
- Task `userCanCancel` property

Tasks: Completion

- `SwingWorker.done` overrides can be tricky
 - call `SwingWorker.get()` for exception
 - check for cancellation, interruption
- Task completion methods:
 - `protected void succeeded(T result)`
 - `protected void failed(Throwable cause)`
 - `protected void cancelled()`
 - `protected void interrupted(InterruptedException e)`
 - `protected void finished()`

Asynchronous @Action example

```
@Action public Task sayHello() { // Say hello repeatedly
    return new SayHelloTask();
}
```

```
private class SayHelloTask extends Task<Void, Void> {
    @Override protected Void doInBackground() {
        for(int i = 0; i <= 10; i++) {
            progress(i, 0, 10); // calls setProgress()
            message("hello", i); // resource defines format
            Thread.sleep(150L);
        }
        return null;
    }
    @Override protected void succeeded(Void result) {
        message("done");
    }
    @Override protected void cancelled() {
        message("cancelled");
    }
}
```



Tasks That Block the GUI

- Task BlockingScope property:
 - NONE, ACTION, COMPONENT, ANCESTOR, WINDOW, APPLICATION
 - ANCESTOR blocks tagged ancestor
- Application deals with blocking the GUI
 - protected void block(Task task, Object... target)
 - protected void unblock(Task task)
- Reasonable default behavior

TaskService

- Defines how a Task is executed, e.g.
 - serially
 - by a thread pool
 - etc...
- TaskService is a ExecutorService
 - named, constructed lazily
 - `@Action(taskService = "database")`

ApplicationContext provides Task Service Access

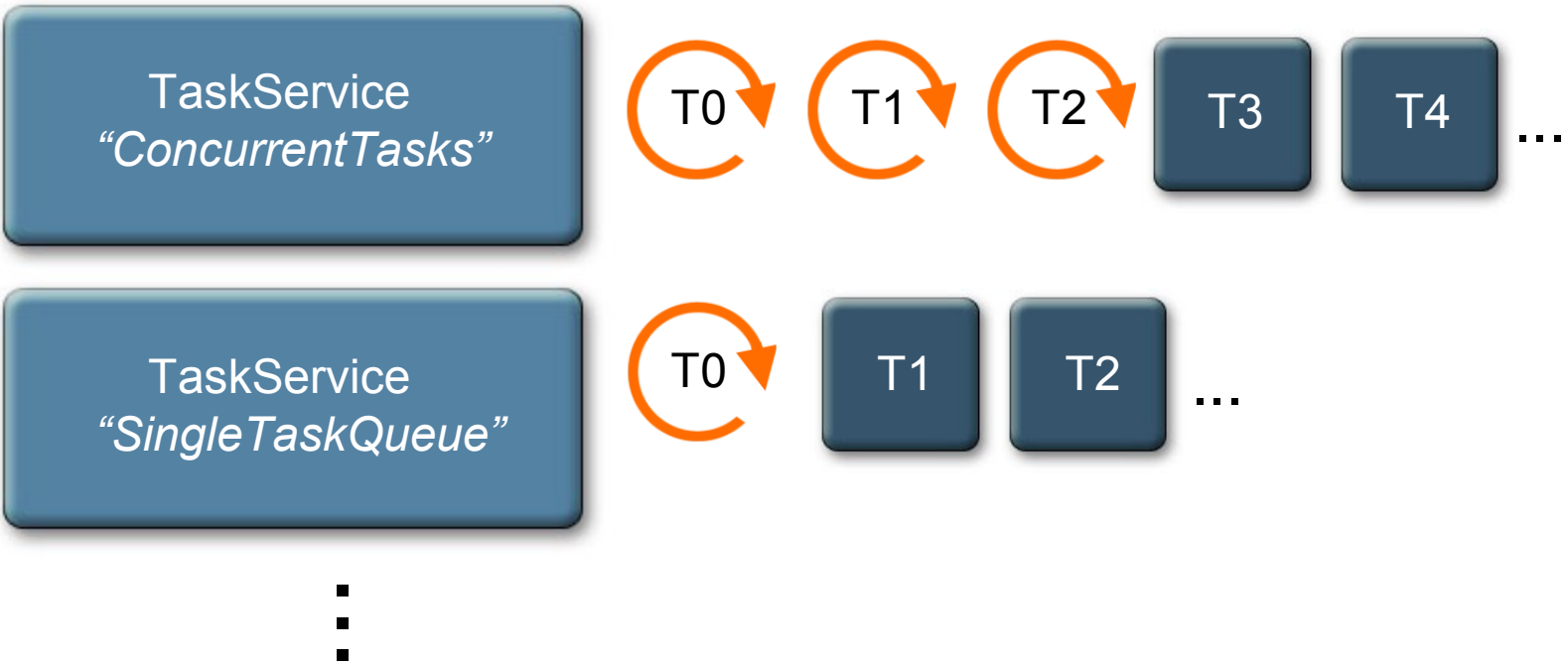
TaskService `getTaskService(String name)`

List<TaskService> `getTaskServices()`

void `addTaskService(TaskService taskService)`

void `removeTaskService(TaskService taskService)`

Tasks and TaskServices



Watching Tasks That Aren't Done: TaskMonitor

- Desktop apps often have many running Tasks
- TaskMonitor provides a summary
 - Bound properties, same as Task
 - Foreground task: first one started
- Handy for StatusBar implementations

Actions and Tasks Summary

- Define Actions with `@Actions`, resources
 - Link enabled/selected to a property
 - `@Action(enabledProperty = name)`
 - `@Action(selectedProperty = name)`
- Asynchronous `@Actions` return Tasks
 - Provide title/description resources
 - Use message/progress methods/properties
 - Use the block parameter and resources
- Connect your status bar to a TaskMonitor

What the Framework Does

- Lifecycle
- Resources
- Actions
- Tasks
- **Session state**

Session State

- Make sure the application remembers where you left things
- Most applications should do this
 - But they don't
 - What state to save?
 - Where to store it (and what if you're unsigned)?
 - How to safely restore the GUI

SessionStorage, LocalStorage

- `ApplicationContext.getSessionStorage()`
- `Save(rootComponent, filename)`
 - Supported types, named components only
 - Window bounds, JTable column widths, etc
 - Archived with `XMLEncoder`
- `Restore(rootComponent, filename)`
 - Conservative
 - Restored with `XMLDecoder`
- `LocalStorage` abstracts per-user files
 - Works for unsigned apps too!

Using SessionStorage

- SingleFrameApplication saves/restores automatically, at startup/shutdown time
- You can customize what's stored/restored
 - Create SessionStorage.Property implementations
 - SessionStorage.putProperty(forClass, ssp)
 - Usually, you will not need to
- Save/restore arbitrary objects too:

```
ApplicationContext c = ApplicationContext.getInstance();  
c.getSessionStorage().save("data.xml", myHashMap);
```



DEMO

Building an Application With NetBeans™ Software!



Summary

- Swing Application Framework supports
 - Actions, resources, tasks, sessions
- Application and ApplicationContext singletons
 - You subclass Application, SingleFrameApplication
 - Call Application.launch() in main
- JSR-296 expert group is responsible
 - for defining the framework's final form
 - finishing in time for Java platform 7
- Get the code and learn more on the project site:
<http://appframework.dev.java.net>

For More Information

- Source code, docs, and binaries on java.net project: <http://appframework.dev.java.net>
- Questions, comments, etc to: users@appframework.dev.java.net
- Related sessions
 - TS-3316—Why Spaghetti Is Not Tasty: Architecting Full-Scale Swing Apps, Thursday, 2:50–3:50PM
 - TS-3569—Beans Binding: Thursday, 4:10–5:10PM



Q&A

Hans Muller, Josh Marinacci

