Applets

the applet lifecycle and more

The lifecycle of an applet can roughly be described through four methods: `init()`, `start()`, `stop()`, and `destroy()`. Each of these methods is inherited from the `Applet` class and invoked by the browser at predefined times. These methods are generally overridden by subclasses of the `Applet` class. This is a great example of the benefits of polymorphism. The browser simply creates an `Applet` object and invokes predefined methods associated with the class. However, the browser knows nothing about the underlying applet nor how the lifecycle methods are defined since they are overridden.

Lifecycle Methods

`init()`

This is the first method which is invoked after creating the applet. This method is, in some ways, similar to a constructor (though not the same thing). Generally, the `init()` method is used to read in information about the applet stored in the html file, download an image from the host server, set up the user interface, etc. The `init()` method is invoked only once during the life of the applet and should contain initial set up logic which only needs to be executed a single time. This will likely be the only lifecycle method which we will override.

`start()`

This method is invoked at least once in the lifecycle of the applet. It is first invoked immediately after the `init()` method and is meant to start any threads which will be used by the applet. To give some intuition about this: any animation that occurs in the context of the applet will generally use multithreading (or, concurrent programming). This allows multiple sub-programs to be executing, in some sense, at the same time (though, not in parallel). These threads will start their execution in a call from the `start()` method inherited from `Applet`. The browser may invoke the `start()` method at other times during the life of the applet, though this will only make sense after understanding the `stop()` method.

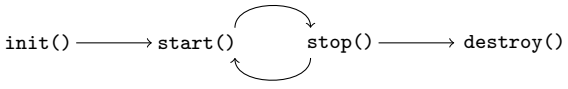
`stop()`

This method is used to stop any execution associated with the applet. This generally means: stop any threads that may be running in the background – i.e. for animation or otherwise. The applet generaly should not be executing at all when `stop()` is invoked. This is because the browser invokes the `stop()` method when it leaves the page into which the applet is embedded. If the browser returns to the page, it will invoke the `start()` method again restarting any threads which were previously stopped. And so, the `start()` and `stop()` methods are to be used in tandem. The first beginning execution of threads (used for animation, etc.), the second for pausing those threads.

`destroy()`

This method, like `init()` is invoked only once in the lifecycle of the applet. This method proceeds the last invocation of the `stop()` method. `stop()` and `destroy()` will get invoked when the browser exits, for instance. In general, `destroy()` is invoked when the browser shuts down the JVM which it is running.

To summarize, the lifecycle of an applet begins with a call to `init()` proceeded by a call to `start()`. This is followed by an alternating set of calls to `stop()` and then `start()` when the browser leaves and returns to the applet page. And finally, the applet completes execution after a call to `destroy()`.



Notice that the `paint()` method hasn't been mentioned. This is because, as we will see, `paint()` doesn't fit so cleanly into this lifecycle description and, furthermore, `paint()` is not a method particular to the `Applet` class. It is inherited from the `Container` class and is common to all `Containers` – more on this in later sections.

`paint()` and `repaint()`

The `paint()` method, as we know, is a method which is overridden as a means of drawing on the applet. This method is also called each time the applet needs to be redrawn. There are many scenarios when we may need to redraw our applet. For instance, when the applet is covered by another window, when the browser is minimized, etc. Once the applet window is visible again, it must be redrawn on the screen. This is accomplished by calls to `paint()`. However, calls to `paint()` should not happen explicitly. That is, whenever windows are minimized, covered, etc. `paint()` gets called by a JVM thread devoted to the GUI. This thread is called the *Event Dispatch Thread* (EDT) and is intended to manage any changes happening within the GUI.

As previously mentioned, `paint()` should not be called explicitly by your code. However, Java provides a way to give the programmer some control over when the GUI gets updated. The programmer may invoke a method called `repaint()` which lets the Event Dispatch Thread know that some `Component` needs to be updated – in our case the components have been `Applets`. A call to `repaint()` is said to schedule an event with the Event Dispatch Thread. Basically, `repaint()` allows all drawing to be handled by the EDT, while still giving the programmer some control over the process.

The `Component` Hierarchy

Just as we can draw on an applet through the `paint()` method, we can also build up the applet with prefabricated items. These items are available in Java's AWT framework and include classes like: `Button`, `CheckBox`, and `Label` among others. All of these classes are subclasses of `Component` and represent an element which can be used in a GUI. For instance, the `Applet` class inherits from `Panel`, which inherits from `Container`, which inherits from `Component`.

These classes are really convenience classes which build up a framework so that the programmer does not need to build their own buttons, checkboxes, etc. Those things have already been created by Java with most of the functionality that we need. However, in the case that you need something more customized, Java allows you to extend these classes so that you may change them as you please. Of course, you may also build your own from scratch.

The nice thing about using the `Component` class and it's subclasses is that we do not have to write a `paint()` method for any of these classes. That is, a `paint()` method has already been written for them, so when we add them to an applet, they paint (and, repaint) themselves as necessary. For instance, consider the code below which displays the text: "Hello, World" in an applet.

```
import java.awt.*;
import java.applet.*;

public class HelloWorld extends Applet {
    public void init() {
        Label l = new Label("Hello, World!");
        this.add(l);
    }
}
```

Notice, as mentioned before, that there is no need for a `paint()` method. We simply create a `Label` object – which inherits from `Component` – and add it to the applet. A `Label` is a

particular type of `Component` on which you can simply write text which can be given to the object through it's constructor. The `String` provided to the constructor will be displayed on the `Label`.

Next, we add the `Label` to the applet with the following line:

```
this.add(l);
```

Notice that `add()` is a method inherited from the `Applet` class. The `add()` method is actually originally inherited from the `Container` class. A `Container` represent a GUI `Component` which can hold other `Component`s inside of it. An applet happens to be a type of `Container`, so we can `add()` other `Component`s into it.

So, where does the `Label` get placed? Well, each `Container` has an object of the class `LayoutManager`. The `LayoutManager` decides where things go in the `Container` and provides a means for the programmer to arrange the `Components`. We will save this for a later section.

Now, since the `Label` has it's own `paint()` method, when the `repaint()` method from the applet gets called, it implicitly calls the `repaint()` methods of all the `Component`s which it contains. So, when the applet gets repainted, all of it's `Component`s get repainted as well.

Notice, however, that we haven't defined a `paint()` method. This does not matter, as we inherit one from the `Applet` class. Of course, the inherited method is empty and will only have functionality if we override it in our subclass. Regardless, if the `repaint()` method of the applet is invoked, all of it's `Component`s `repaint()` methods will also be invoked.

Furthermore, it can be seen that the `Label` was created and added to the applet in the `init()` method. This is because, *there is no reason to do this more than once.* We would not want to create a new `Label` object and add it to the GUI each time that the applet got repainted for instance. We would then have another `Label` each time we minimized and then mazimized the window in which the applet resides. This would not only be a waste of memory, but it would ruin the appearance of our GUI. So, for this reason, we set up the groundwork for our GUI in the `init()` method.

To get a better feel for how all of this works, examine the source code from the applets provided on the course website. Also, you can look at the Java API for `Component`s, `Button`s, etc.

Parameters

While applets are not ever running from the command line, we can still pass arguments to them from outside of the program. These are very similar to command line arguments, though we have no `main()` method. The way these are passed to the applet is through the HTML file into which the applet is embedded.

The HTML `<applet>` tag, used to embed an applet into an HTML file, may be used in conjunction with the `<param>` tag. For instance, consider the following applet and corresponding HTML file.

```
public class Greeting extends Applet {
    String greeting;

    public void init() {
        String param = this.getParameter("theGreeting");
        if(param == null) greeting = "Hello, World";
        else greeting == param;

        this.add(new Label(greeting));
    }
}
```

```
<applet code="Greeting.class" height=200 width=200>
    <param name="theGreeting" value="Hola, Mundo">
    <param name="someOtherParam" value="Blah">
</applet>
```

Notice the line in the HTML file that specifies the parameter name and associated value:

```
<param name="theGreeting" value="Howdy, there!">
```

From here the applet extracts the parameter. This can be seen in the following line from the applet code:

```
String param = this.getParameter("theGreeting");
```

The method `getParameter()` is inherited from the `Applet` class and allows the applet to extract a particular value, by giving a name. The value which is associated with the name given is extracted and if the name given does not match any of the parameters, then `null` is returned.

Once again, pay attention to how this logic happens in the `init()` method. This is because we really only want to extract the parameter's once and then store them in the object somehow. If we were to extract parameters in the `paint()` method, the extraction would happen each time the window had to be repainted.

A Final Note

It is very important to be able to discern the sort of logic that should go into each of the methods mentioned above. Poor choices in this regard can lead to a very inefficient GUI. It is important to remember that almost all the work of setting up the GUI will be done in the `init()` method of an applet. The `paint()` method is really for redisplaying the applet each time it gets covered up.

Furthermore, it is important get comfortable with the framework that Java has provided for constructing a GUI. Namely, all the subclasses of `Component` will be extremely useful in this regard. There is no need to reinvent the wheel, as they say.