A **GUI component** is an object that can be represented visually on the computer screen and can be acted upon by a human user. GUI stands for *graphical user interface*. Windows, buttons, combo boxes and scroll bars are examples of GUI components. In Java, such an object must be an instance of the class `java.awt.Component` or its many subclasses.

**Rendering** is the process of transforming the GUI component’s internal data into a visual image. In Java, a *graphics context* object holds the data needed to do this transformation. Such data includes:

- The component’s color
- The component’s font
- The transform mapping used to convert component coordinates into pixel positions on the display device

`java.awt.Graphics` is the abstract superclass for all graphics context objects in Java. You can use its methods to get and set the data within a graphics context.

The following two methods get and set the graphics context’s current color. The `Color` class is explained in topic *Color*.

```
java.awt.Graphics Methods for Colors

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color getColor( )</td>
</tr>
<tr>
<td>void setColor( Color c )</td>
</tr>
</tbody>
</table>
```

These two methods get and set the graphics context’s current font. The `Font` class is explained in topic *Fonts*.

```
java.awt.Graphics Methods for Fonts

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Font getFont( )</td>
</tr>
<tr>
<td>void setFont( Font f )</td>
</tr>
</tbody>
</table>
```

This method visually renders a string using the graphics context’s current color and font.

```
void drawString( String str, int x, int y )
// Render the text in 'str' at position (x, y).
```
How to Use the Graphics Context to Render Components

`java.awt.Component` contains an instance method that renders the contents of the component object onto the computer screen. To guide the rendering, it requires the component’s graphics context as a parameter.

```java
void paint( Graphics g )
// Uses the given graphics context to render this component.
```

Objects that are instances of class `java.awt.Component` or a subclass will inherit this method and have it available to morph.

Whenever the JVM determines that the GUI component needs to be rendered – such as when it is resized or maximized – it automatically calls the component’s `paint` and passes the graphics context to parameter `g`. Thus, to customize the rendering of a GUI component:

- Create your own subclass for the component.
- Morph its inherited `paint` method.
- Within `paint` add content to the component’s graphics context by calling the appropriate `Graphics` methods within object `g`.

**Example**
A Java applet is a GUI component; its place in Java’s class hierarchy is shown at right. Thus, the class inherits the `paint` method.

Let’s write an applet the creates this window:

Since applet inherits the `paint` method, we morph it, setting the graphics context’s color and font as desired and then adding the string to it. The code is shown below.
import java.applet.Applet;
import java.awt.*; // Graphics, Color, Font
public class GraphicsAppletDemo extends Applet
{
    public void paint( Graphics g )
    {
        super.paint( g ); // call Applet paint
        g.setColor( Color.RED );
        g.setFont( new Font( "Trebuchet MS", Font.PLAIN, 24 ) );
        g.drawString( "Welcome to Graphics", 20, 50 );
    }
}

The User Coordinate Space
Two-dimensional GUI objects are positioned on an x-y axis called the user coordinate space. In this space, the origin (0, 0) is in the upper-left corner. The x-axis is horizontal and the y-axis vertical. The x values count up from 0 going left to right; the y values from top to bottom. The values are in pixels.

Example
Applet GraphicsAppletDemo draws the message 20 pixels across and 50 down.
**Repainting**

Since it is meant to be called by the JVM, your Java program should never explicitly call the `paint` method. Your program can, however, request that the JVM render the component by calling its `repaint` method, which arranges for the JVM to call `paint`. Here are the overloaded versions of the `repaint` method.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>repaint()</code></td>
<td>Causes a call to this component's paint method as soon as possible.</td>
</tr>
<tr>
<td><code>repaint(long msec)</code></td>
<td>Causes a call to this component's paint method in 'msec' milliseconds.</td>
</tr>
<tr>
<td><code>repaint(int x, int y, int width, int height)</code></td>
<td>Repaints a rectangular area of this component whose upper-left corner is (x,y) and is the given width and height.</td>
</tr>
</tbody>
</table>

**Example**

We’re going to design a simple applet that reads messages from its HTML file and displays them one at a time. The messages are communicated from the HTML file to the applet through applet parameters (see the topic *Applet Parameters*). The applet’s execution is suspended after each message by displaying a dialog box that requires a user’s response. Here’s the algorithm:

```html
<applet code="Repainting.class">
  <param name="count" value="n">
  <param ...message 1... >
  <param ...message 2... >
  . . .
  <param ...message n... >
</applet>
```

- fetch the parameter named "count"
- \( n \) = scan it to an integer
- repeat \( n \) times
- fetch the next message
- call repaint to paint the message
- display model dialog and wait for user

The parameter named “count” gives the number of messages to print and is coded into the HTML file using a `param` element. For example, the following line defines a count of 4:
The algorithm must be coded as the applet’s life cycle method `start`. Its first two lines – fetching the parameter named “count” and scanning it into an integer – are accomplished with this Java statement:

```
int count = Integer.parseInt( this.getParameter( "count" ) );
```

Having done this, the loop can be written as a `for` statement:

```
for ( int k = 0; k < count; k++ )
{
    ...
}
```

Each message is coded into the HTML file using a `param` element whose name is `msgX`, where `X` numbers the message. For example, here sample lines for messages 1 and 2:

```
<param name="msg1" value="Happy birthday to you">
<param name="msg2" value="You live in a zoo">
```

Thus, on the first loop cycle, the applet accesses message #1 by concatenating the value of `k+1` onto the string "msg" and fetching the parameter with that name.

```
msg = this.getParameter( "msg" + (k + 1) );
```

A call to `repaint` causes the JVM to call the `paint` method, which draws the `String` object `msg` onto the applet.

```
public void paint( Graphics g )
{
    super.paint( g);
    g.drawString( msg, 0, 20 );
}
```

`msg` must be given class-level scope so that both `start` and `paint` methods can access it.

Back in the `start` method’s `for` loop, a message dialog is displayed so that the applet pauses. Here’s the entire loop:

```
for ( int k = 0; k < count; k++ )
{
    msg = this.getParameter( "msg" + (k + 1) );
    this.repaint( 0, 0, this.getWidth( ), 40 );
    showMessageDialog( null, "Click OK" );
}
```
Exercises

1. Modify GraphicsAppletDemo so that it uses a JOptionPane input dialog to read the user’s name and displays a message with the name, such as shown to the right.

2. Following the implementation details explained in the repainting example on pages 4 and 5, write a complete Java applet and associated HTML file that fetches messages from the HTML via applet parameters and displays them, one at a time, on the applet background. After each message is displayed, use a JOptionPane message dialog to suspend the applet’s execution and wait for the user’s response.