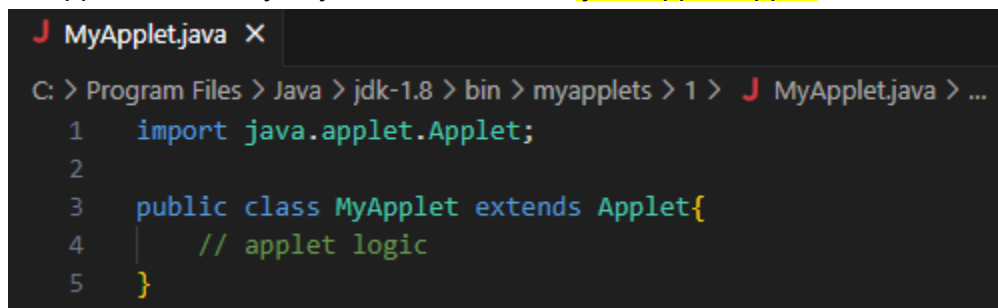


Unit-4 : Applets

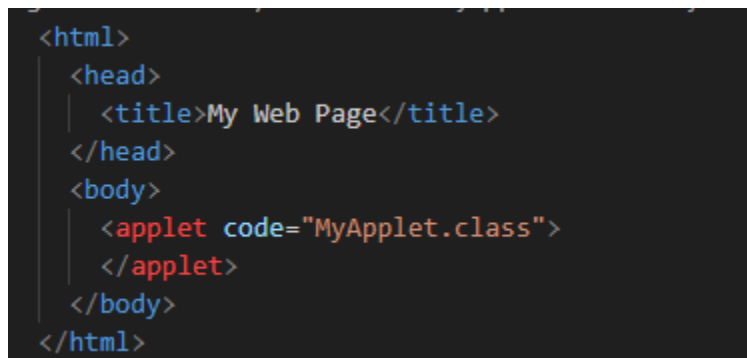
Java Applets:

- Java Applet is a java program that can be embedded into an HTML page to create dynamic web pages.
- These web pages can only be executed in Java-compatible web-browsers.
- Hence, initially before JavaScript, Java Applets were used as a programming language for creating web pages.
- Applets are used to provide interactive features to web applications that cannot be provided by HTML alone.
- For Example: They can capture user actions like mouse and keyboard events and provide the corresponding actions when the event triggers.
- Overtime, Java Applets have depreciated and today's web browsers that we use only support JavaScript as the primary language for the web.
- Creating web pages with applets:
 - + An applet is basically any class that extends `java.applet.Applet` class.



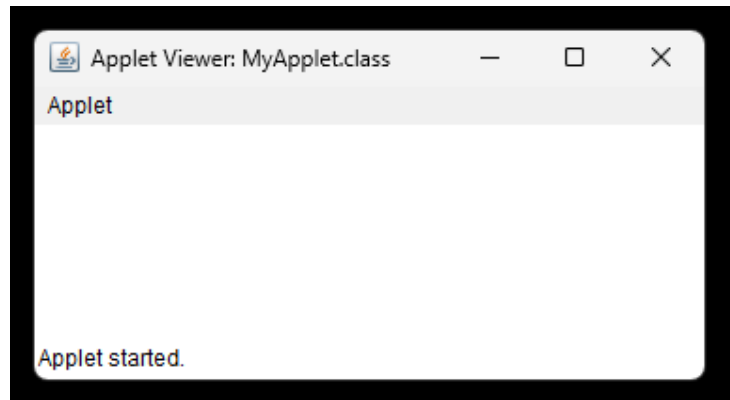
```
J MyApplet.java X
C: > Program Files > Java > jdk-1.8 > bin > myapplets > 1 > J MyApplet.java > ...
1  import java.applet.Applet;
2
3  public class MyApplet extends Applet{
4      // applet logic
5  }
```

- + This java file(whose main public class extends the Applet class), is compiled with `javac filename.java` command and the resultant class file is provided to an HTML file.



```
<html>
  <head>
    <title>My Web Page</title>
  </head>
  <body>
    <applet code="MyApplet.class">
    </applet>
  </body>
</html>
```

- + `<applet>` is the element used in an HTML file to embed the applet code.
- + The class file is provided to a "code" property of `<applet>` element.
- The above is the standard process of creating web pages using Java Applets.
- But, today's web browsers don't support the Java Applets as the primary language for web. Hence, it is not possible to render the web pages with Java Applets.
- Hence, Java provides another way of viewing those HTML pages. Which is by using `appletviewer`.
- `appletviewer` is a .exe file present in the Java Library, which is used as a tool to view the HTML pages with applets, browser independently.
- I.e., appletviewer is a tool provided by Java to view the Applet containing web pages as a standalone window application.
- appletviewer is independent of web browsers, but is dependent on the OS(Operating System) for UI(User-Interface).
- Run `appletviewer "page_name.html"` command to view the HTML page.



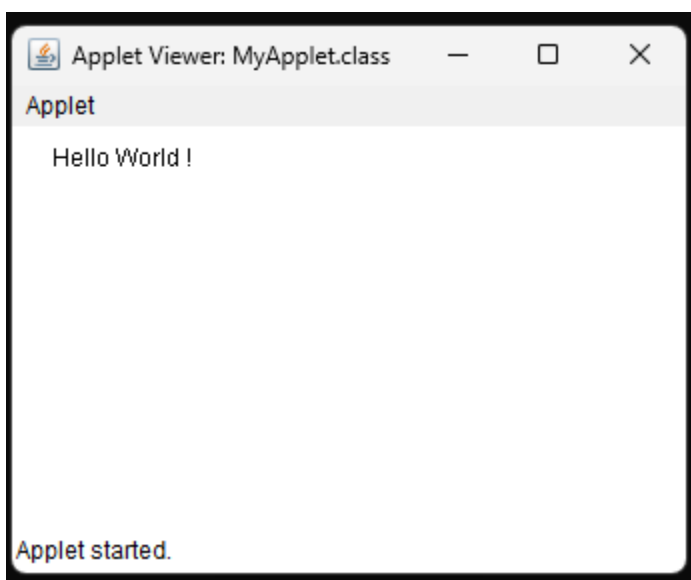
Ex: (simple Applet Program to display a string)

```
import java.applet.Applet;
import java.awt.Graphics;

public class MyApplet extends Applet{

    public void paint(Graphics g){
        g.drawString("Hello World !", 20, 20);
    }
}
```

```
<html>
<head>
  <title>My Web Page</title>
</head>
<body>
  <applet code="MyApplet.class" width="200" height="200"> </applet>
</body>
</html>
```



Life Cycle of an Applet:

- The life cycle of a Java applet is the process through which an applet is created, initialized, started, stopped, and destroyed.
- This process is controlled by the browser or applet viewer in which the applet is running.
- The following are the five stages in the life cycle of a Java applet:
 1. **Initialization** [`init()`]: This is the first method called when an applet is loaded. It is used to perform one-time initialization tasks, such as setting up variables, creating objects, and initializing the applet's state.
 2. **Starting** [`start()`]: The start method is called after the init method and also whenever the user returns to a web page containing the applet. It is used to start execution of the applet.
 3. **Running/Painting** [`paint()`]: The paint method is called whenever the applet needs to be redrawn, such as when it is first displayed or when it is uncovered by another window. Developers override this method to define how the applet should be visually rendered.
 4. **Stopping** [`stop()`]: The stop method is called after the start method and also whenever the user returns to a web page containing the applet. It is used to stop execution of the applet.
 5. **Destroying** [`destroy()`]: The destroy method is called when the browser completely removes the applet from memory. It is used to perform cleanup tasks, such as releasing resources and saving state.

```
import java.applet.Applet;
import java.awt.Graphics;

public class MyApplet extends Applet {

    // Initialization
    public void init() {
        // Perform initialization tasks
    }

    // Starting
    public void start() {
        // Start execution
    }

    // Stopping
    public void stop() {
        // Suspend execution
    }

    // Destroying
    public void destroy() {
        // Perform cleanup tasks
    }

    // Painting
    public void paint(Graphics g) {
        // Define how the applet should be visually rendered
    }
}
```

Adding Images to an Applet:

- An Image is a multimedia file which can be displayed on an HTML page.
- `java.awt` provides the `Image` class to store the data of an Image file.
- `getImage()` method provided by `java.applet.Applet`, can be used to get the image file data.
- `Graphics` class provided by `java.awt` provides a method called `drawImage()` to display an image on the provided class instance.

I.e, The last parameter is an instance on which the image is to be displayed.

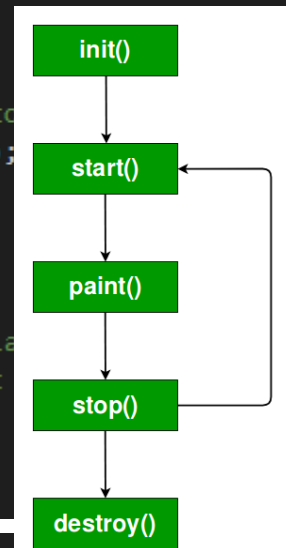
Ex:

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

public class MyApplet extends Applet{
    private Image image;

    public void init(){
        // Load the image using Applet.getImage()
        // Applet.getDocumentBase() returns the directory of the object file
        image = getImage(getDocumentBase(), "img.png");
    }

    public void paint(Graphics g){
        // Draw the image at 10px from top and left.
        // this is the current instance of MyApplet class
        // (i.e., draw the image on the same component)
        g.drawImage(image, 10, 10, this);
    }
}
```



```
<html>
<head>
| <title>My Web Page</title>
</head>
<body>
| <applet code="MyApplet.class" width="200" height="200"> </applet>
</body>
</html>
```



Adding Sound to an Applet:

- An Audio/Sound is a multimedia file which can be listened to on the HTML page.
- **AudioClip** Interface provided by **java.applet** can be used to store the audio file data.
- **getClass().getResource("filename.wav")** provided by java.lang.Object class, is used to get the path of the audio file.
- To control the audio track, the **AudioClip** interface provides some methods:
 - play() - to start playing the audio track
 - stop() - to stop playing the audio track

```

import java.applet.*;
import java.net.*; // URL class

public class MyApplet extends Applet{
    private AudioClip sound;

    public void init(){
        // Get the path of sound.wav file
        URL soundUrl = getClass().getResource("sound.wav");
        // Load the sound.wav file to "sound" attribute, using Applet.getAudioClip()
        sound = getAudioClip(soundUrl);
    }

    public void start(){
        // Play the sound
        sound.play();
    }

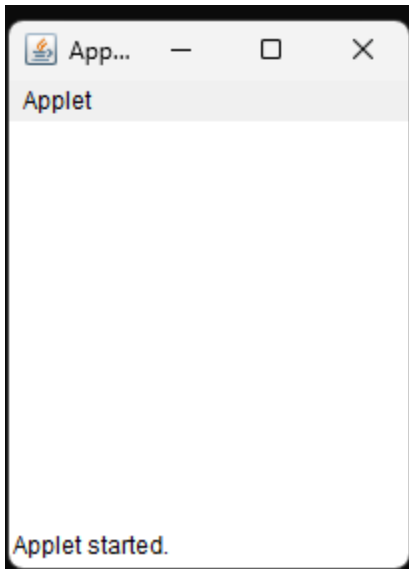
    public void stop{
        // Stop the sound
        sound.stop();
    }
}

```

```

<html>
<head>
| <title>My Web Page</title>
</head>
<body>
| <applet code="MyApplet.class" width="200" height="200"> </applet>
</body>
</html>

```



Passing Parameters to an applet:

- Parameters are the carriers used to pass data from HTML page to Applet Logic.
- This is useful for applications which take user inputs.
- A user always interacts with an HTML page and its components. Hence when user gives input, it can be provided to Applet logic to do computations with it.
- Every parameter has a unique name.
- In HTML, `<param>` element is used to set a parameter, where the name of the parameter is assigned to the "name" property, and the parameter value is assigned to the "value" property.
- `<param>` elements are specified under the `<applet>` element, to pass them to the corresponding Applet file.
- In Applet, `getParameter("parameter_name")` can be used to get the value of a parameter.
- `Graphics` class provided by `java.awt` provides a method called `drawString()` to display a string on the applet window.

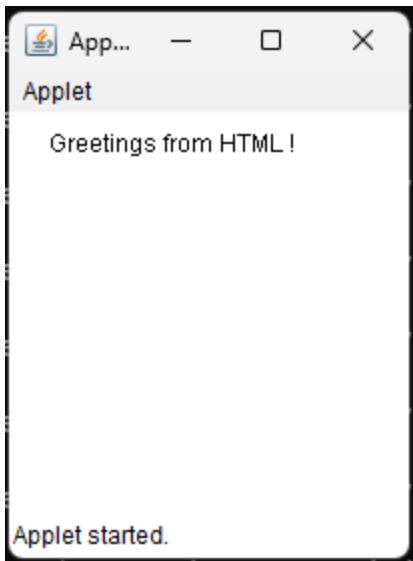
```
import java.applet.*;
import java.awt.*;

public class MyApplet extends Applet{
    private String message;

    public void init(){
        // Get the value of "message" parameter from HTML and assign it to "message" attribute
        message = getParameter("message");
    }

    public void paint(Graphics g){
        // display the message at 20px from top and left
        g.drawString(message, 20, 20);
    }
}
```

```
<html>
<head>
  <title>My Web Page</title>
</head>
<body>
  <applet code="MyApplet.class" width="200" height="200">
    <!-- Pass the parameter, named "message", to MyApplet, with a value -->
    <param name="message" value="Greetings from HTML !" />
  </applet>
</body>
</html>
```



Event Handling:

Event Handling is a mechanism to **control the events** and to **decide what should happen after an event** occurs.

To handle the events, Java follows the ***Delegation Event model***.

In earlier versions of java, event handling is assigned to the sole components.

The Major disadvantage with this approach was that the GUI components related code and event handling related code used to be written together at same place.

Hence if one wants to modify the GUI(appearance) of a component, then event handling code messes up too.

Hence, Delegation Event Model was introduced to separate the GUI components from event handling process.

Delegation Event Model

This Model divides the concept of event handling into two parts, viz: -

1. Source: The Event Generating Components, such as button, checkbox, textfield, scrollbar, etc... are termed as the sources.
2. Listeners: The built-in, Event-Listening interfaces, which are used for listening to/identifying an event and handling them, are known as listeners.

The Idea of this model is *“A source generates an event and sends it to one or more listeners. The listener waits for an event and processes the event, once it is received.”*

The advantage of this model is that the Event handling process for a component is separated from the separated from that GUI component, so that the GUI code can be modified without affecting the event handling code.

Events

The changing state of a source is known as an event.

An event can be occurred with or without the user interaction with the sources.

An Event occurred with user interaction, is due to the action performed by the user, by using I/O devices, such as key-press, mouse hover, mouse click, etc...

An Event occurred implicitly (without user interaction), can be due to the pre-loaded instructions, such as when timer expires, a counter reaches a value, etc...

Event Sources

An object that generates an event is known as the Source of that generated event (I.e., Event Source).

A single source can generate more than one type of event.

Ex: button can generate a click event, and mouse hover event.

Event Listeners:

A listener is an interface which listens to an event and provides methods for handling specific types of events.

Each Event has its own listening interface.

Event handling is performed under the method provided by every type of interface.

A source must be registered with a method of a listener, in-order to allow a listener to receive notifications(status) about the specific type of event performed on that source.

Each type of event has its own registration method with the below syntax: -

Source_obj.addTypeListener(listenerClassObject);

'*ListenerClassObject*' is an instance of the class which has implemented the Event-Listening Interface and is listening for an event.

'*Type*' is a placeholder for an event type, such as *Key*, *MouseMotion*, etc...

The syntax for un-registering a listener is: -

Source_obj.removeTypeListener(listnerClassObject);

The Event Listening Interfaces are provided by the java.awt.event package.

To add an event listener to a source object, the interface (of that type) must be implemented by a class.

The method registration and event handling must be performed under the implemented class.

Event Classes:

Java provides the definition of an Event in the form of classes.

Each event has its own class.

The Event classes are provided by the java.awt.event package.

Event class must be specified as an object type to the parameter of the method (provided by event listener interface) which is implicitly invoked when that type of event occurs.

Some of the event classes with the corresponding type of listeners (interfaces) are as shown below: -

Event Class	Listener Interface	Description
ActionEvent	ActionListener	An event that indicates that a component-defined action occurred like a button click or selecting an item from the menu-item list.
AdjustmentEvent	AdjustmentListener	The adjustment event is emitted by an Adjustable object like Scrollbar.
ComponentEvent	ComponentListener	An event that indicates that a component moved, the size changed or changed its visibility.
ContainerEvent	ContainerListener	When a component is added to a container (or) removed from it, then this event is generated by a container object.
FocusEvent	FocusListener	These are focus-related events, which include focus, focusin, focusout, and blur.
ItemEvent	ItemListener	An event that indicates whether an item was selected or not.
KeyEvent	KeyListener	An event that occurs due to a sequence of keypresses on the keyboard.
MouseEvent	MouseListener & MouseMotionListener	The events that occur due to the user interaction with the mouse (Pointing Device).
MouseWheelEvent	MouseWheelListener	An event that specifies that the mouse wheel was rotated in a component.
TextEvent	TextListener	An event that occurs when an object's text changes.
WindowEvent	WindowListener	An event which indicates whether a window has changed its status or not.

The methods corresponding to the listeners (interfaces), which are invoked implicitly when the specified event occurs on the specified source object, are as shown below: -

Listener Interface	Methods
ActionListener	<ul style="list-style-type: none"> • actionPerformed()
AdjustmentListener	<ul style="list-style-type: none"> • adjustmentValueChanged()
ComponentListener	<ul style="list-style-type: none"> • componentResized() • componentShown() • componentMoved() • componentHidden()
ContainerListener	<ul style="list-style-type: none"> • componentAdded() • componentRemoved()
FocusListener	<ul style="list-style-type: none"> • focusGained() • focusLost()
ItemListener	<ul style="list-style-type: none"> • itemStateChanged()
KeyListener	<ul style="list-style-type: none"> • keyTyped() • keyPressed() • keyReleased()
MouseListener	<ul style="list-style-type: none"> • mousePressed() • mouseClicked() • mouseEntered() • mouseExited() • mouseReleased()
MouseMotionListener	<ul style="list-style-type: none"> • mouseMoved() • mouseDragged()
MouseWheelListener	<ul style="list-style-type: none"> • mouseWheelMoved()
TextListener	<ul style="list-style-type: none"> • textChanged()
WindowListener	<ul style="list-style-type: none"> • windowActivated() • windowDeactivated() • windowOpened() • windowClosed() • windowClosing() • windowIconified() • windowDeiconified()

Introducing AWT: