Difference between a Java Application and a Java Applet

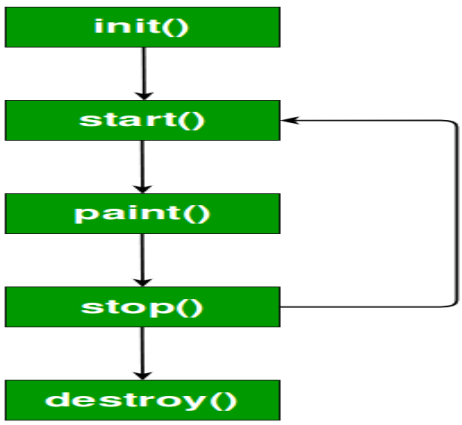
Java Application is just like a Java program that runs on an underlying operating system with the support of a virtual machine. It is also known as an application program. The graphical user interface is not necessary to execute the java applications, it can be run with or without it.

Java Applet is an applet is a Java program that can be embedded into a web page. It runs inside the web browser and works on the client-side. An applet is embedded in an HTML page using the APPLET or OBJECT tag and hosted on a web server. Applets are used to make the website more dynamic and entertaining.

Applets

All applets are sub-classes (either directly or indirectly) of java.applet.Applet class.
Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called applet viewer.
In general, execution of an applet does not begin at main() method.
Output of an applet window is not performed by System.out.println(). Rather it is handled with various AWT methods, such as drawString().

Life cycle of an applet:



When an applet begins, the following methods are called, in this sequence:

init( )

start( )

paint( )

When an applet is terminated, the following sequence of method calls takes place:

stop( )

destroy( )

Let's look more closely at these methods.

1. init( ) : The init( ) method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

2. start( ) : The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped. Note that init( ) is called once i.e. when the first time an applet is loaded whereas start( ) is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start( ).

3. paint( ) : The paint( ) method is called each time an AWT-based applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored.

paint( ) is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint( ) is called.

The paint( ) method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

public void paint(Graphics g)

where g is an object reference of class Graphics.

4. stop( ) : The stop( ) method is called when a web browser leaves the HTML document containing the applet—when it goes to another page.

5. destroy( ) : The destroy( ) method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The stop( ) method is always called before destroy ( ).

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;

/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/

public class AppletSkel extends Applet {
// Called first.
public void init() {
// initialization
}
/* Called second, after init(). Also called whenever
the applet is restarted. */
public void start() {
// start or resume execution
}
// Called when the applet is stopped.
public void stop() {
// suspends execution
}
/* Called when applet is terminated. This is the last
method executed. */
public void destroy() {
// perform shutdown activities
}
```

```
// Called when an applet's window must be restored.
public void paint(Graphics g) {
// redisplay contents of window
}
```

Overriding update( )

In some situations, the applet may need to override another method defined by the AWT, called **update( )**. This method is called when your applet has requested that a portion of its window be redrawn. The default version of **update( )** first fills an applet with the default background color and then calls **paint( )**. If you fill the background using a different color in **paint( )**, the user will experience a flash of the default background each time **update( )** is called—that is, whenever the window is repainted. One way to avoid this problem is to override the **update( )** method so that it performs all necessary display activities. Then have **paint( )** simply call **update( )**. Thus, for some
applications, the applet skeleton will override **paint( )** and **update( )**, as shown here:

```
public void update(Graphics g) {
// redisplay your window, here.
}
public void paint(Graphics g) {
update(g);
}
```

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

public class HelloWorld extends Applet
{
        public void paint(Graphics g)
    {
        g.drawString("Hello World", 20, 20);
    }
}
```

Inside paint( ) is a call to drawString( ), which is a member of the Graphics class. This method outputs a string beginning at the specified X,Y location. It has the following general form:

void drawString(String message, int x, int y)

Here, message is the string to be output beginning at x,y. In a Java window, the upper-left corner is location 0,0. The call to drawString( ) in the applet causes the message "Hello World" to be displayed beginning at location 20,20.

Using the Status Window

In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. To do so, call showStatus( ) with the string that you want displayed.

```
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/*
<applet code="StatusWindow" width=300 height=50>
</applet>
*/
public class StatusWindow extends Applet{
```

```
public void init() {
setBackground(Color.cyan);
}
// Display msg in applet window.
public void paint(Graphics g) {
g.drawString("This is in the applet window.", 10, 20);
showStatus("This is shown in the status window.");
}
}
```

HTML Tag

< APPLET
[CODEBASE = *codebaseURL*]
CODE = *appletFile*
[ALT = *alternateText*]
[NAME = *appletInstanceName*]
WIDTH = *pixels* HEIGHT = *pixels*
[ALIGN = *alignment*]
[VSPACE = *pixels*] [HSPACE = *pixels*]
>
[< PARAM NAME = *AttributeName* VALUE = *AttributeValue*>]
[< PARAM NAME = *AttributeName2* VALUE = *AttributeValue*>]
. . .
[*HTML Displayed in the absence of Java*]
</APPLET>

CODEBASE CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified.

CODE CODE is a required attribute that gives the name of the file containing your applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.

ALT The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser recognizes the APPLET tag but can't currently run Java applets.

NAME NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use **getApplet( )**, which is defined by the **AppletContext** interface.

WIDTH and HEIGHT WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.

ALIGN ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

VSPACE and HSPACE These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

PARAM NAME and VALUE The PARAM tag allows you to specify applet-specific arguments in an HTML page. Applets access their attributes with the getParameter( ) method.

Passing parameter to Applet

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

public class UseParam extends Applet{

public void paint(Graphics g){
String str=getParameter("msg");
g.drawString(str,50, 50);
}

}
```

MyApplet.html

```
<html>
<body>
<applet code="UseParam.class" width="300" height="300">
<param name="msg" value="Welcome to applet">
</applet>
</body>
</html>
```

getDocumentBase( ) and getCodeBase( )

Java will allow the applet to load data from the directory holding the HTML file that started the applet (the *document base*) and the directory from which the applet's class file was loaded (the *code base*). These directories are returned as **URL** objects by **getDocumentBase( )** and **getCodeBase( )**. They can be concatenated with a string that names the file you want to load.

The following applet illustrates these methods:

```
// Display code and document bases.
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="Bases" width=300 height=50>
</applet>
*/
public class Bases extends Applet{
// Display code and document bases.
public void paint(Graphics g) {
String msg;
URL url = getCodeBase(); // get code base
msg = "Code base: " + url.toString();
g.drawString(msg, 10, 20);
url = getDocumentBase(); // get document base
msg = "Document base: " + url.toString();
```

```
g.drawString(msg, 10, 40);
}
}
```

Output

What is AWT in Java?

Abstract Window Toolkit acronymed as AWT is a toolkit of classes in Java which helps a programmer to develop Graphics and Graphical User Interface components. It is a part of JFC (Java Foundation Classes) developed by Sun Microsystems.

Features of AWT in Java

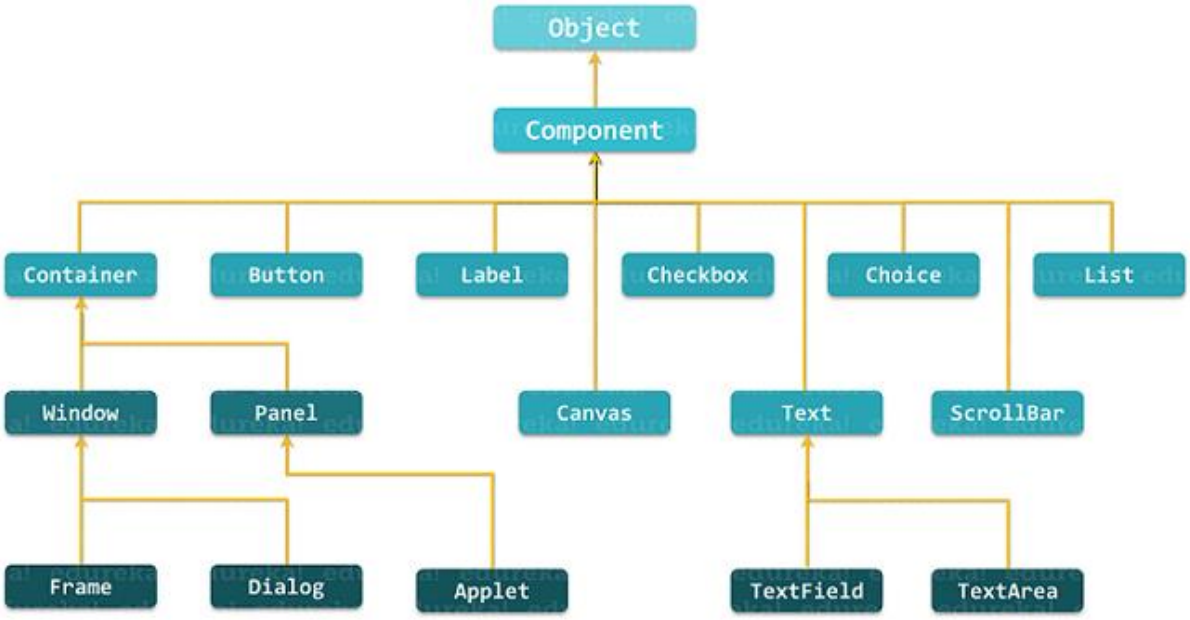
- AWT is a set of native user interface components
- It is based upon a robust event-handling model
- It provides Graphics and imaging tools, such as shape, color, and font classes
- AWT also avails layout managers which helps in increasing the flexibility of the window layouts
- Supports a wide range of libraries that are necessary for creating graphics for gaming products etc.

AWT UI Aspects

Any UI will be made of three entities:

- **UI elements**: These refers to the core visual elements which are visible to the user and used for interacting with the application.
- **Layouts**: These define how UI elements will be organized on the screen and provide the final look and feel to the GUI.
- **Behavior**: These define the events which should occur when a user interacts with UI elements.

Hierarchy Of AWT



AWT Components

1. Containers

Container in Java AWT is a component that is used to hold other components such as text fields, buttons, etc. It is a subclass of java.awt.Component and is responsible for keeping a track of components being added. There are four types of containers provided by AWT in Java.

Types of Containers

1. **Window**: It is an instance of the Window class having neither border nor title. It is used for creating a top-level window.
2. **Frame**: Frame is a subclass of Window and contains title, border and menu bars. It comes with a resizing canvas and is the most widely used container for developing AWT applications. It is capable of holding various components such as buttons, text fields, scrollbars, etc. You can create a Java AWT Frame in two ways:
   i. By Instantiating Frame class
   ii. By extending Frame class
3. **Dialog:** Dialog class is also a subclass of Window and comes with the border as well as the title. Dialog class's instance always needs an associated Frame class instance to exist.
4. **Panel**: Panel is the concrete subclass of Container and doesn't contain any title bar, menu bar or border. Panel class is a container for holding the GUI components. You need the instance of the Panel class in order to add the components.

2. Button

java.awt.Button class is used to create a labeled button. GUI component that triggers a certain programmed *action* upon clicking it.

The Button class has two constructors:

```
//Construct a Button with the given label
public Button(String btnLabel);

//Construct a Button with empty label
```

```
public Button();
```

A few of the methods provided by this class have been listed below:

```
//Get the label of this Button instance
public String getLabel();

//Set the label of this Button instance
public void setLabel(String btnLabel);

//Enable or disable this Button. Disabled Button cannot be clicked
public void setEnable(boolean enable);
```

3. Text Field

A java.awt.TextField class creates a single-line text box for users to enter texts.

The TextField class has three constructors which are:

```
//Construct a TextField instance with the given initial text string with the number of columns.
public TextField(String initialText, int columns);

//Construct a TextField instance with the given initial text string.
public TextField(String initialText);

//Construct a TextField instance with the number of columns.
public TextField(int columns);
```

A few of the methods provided by TextField class are:

```
Get the current text on this TextField instance
public String getText();

// Set the display text on this TextField instance
public void setText(String strText);

//Set this TextField to editable (read/write) or non-editable (read-only)
public void setEditable(boolean editable);
```

4. Label

The java.awt.Label class provides a descriptive text string that is visible on GUI. An AWT Label object is a component for placing text in a container. Label class has three constructors which are:

```
public Label(String strLabel, int alignment);

//Construct a Label with the given text String
public Label(String strLabel);

//Construct an initially empty Label
public Label();
```

This class also provides 3 constants which are:

```
 final LEFT; // Label.LEFT
 public static final RIGHT;  // Label.RIGHT
```

public static final CENTER; // Label.CENTER

Methods provided by this class:

public String getText();
public void setText(String strLabel);
public int getAlignment();
//Label.LEFT, Label.RIGHT, Label.CENTER
public void setAlignment (int alignment)

5. Canvas

A Canvas class represents the rectangular area where you can draw in an application or receive inputs created by the user.

6. Choice

Choice class is used to represent a pop-up menu of choices. The selected choice is shown on the top of the given menu.

7. Scroll Bar

The Scrollbar class object is used to add horizontal and vertical scrollbar in the GUI. It enables a user to see the invisible number of rows and columns.

8. List

The object of List class represents a list of text items. Using the List class a user can choose either one item or multiple items.

9. CheckBox

The Checkbox is a class is a graphical component that is used to create a checkbox. It has two state options; true and false. At any point in time, it can have either of the two.

The Delegation Event Model

The approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events.

An **event** can be defined as changing the state of an object or behavior by performing actions. Actions can be a button click, cursor movement, keypress through keyboard or page scrolling, etc.

The **java.awt.event** package can be used to provide various event classes.

Classification of Events

- Foreground Events
- Background Events

Its concept is quite simple: a source generates an event and sends it to one or more listeners.

The listener simply waits until it receives an event. Once received, the listener processes the event and then returns.

Source: Events are generated from the source. There are various sources like buttons, checkboxes, list, menu-item, choice, scrollbar, text components, windows, etc., to generate events.

Listeners: Listeners are used for handling the events generated from the source. Each of these listeners represents interfaces that are responsible for handling events.

In the delegation event model, listeners must register with a source in order to receive an event notification.

To perform Event Handling, we need to register the source with the listener.

Registering the Source With Listener

Different Classes provide different registration methods.

Syntax:

addTypeListener()

where Type represents the type of event.

Example: KeyEvent we use addKeyListener() to register.

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Sources of Events

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Listener Interfaces

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

- **The ActionListener Interface**

void actionPerformed(ActionEvent *ae*)

- **The AdjustmentListener Interface**

void adjustmentValueChanged(AdjustmentEvent *ae*)

- **The ComponentListener Interface**

void componentResized(ComponentEvent *ce*)
void componentMoved(ComponentEvent *ce*)
void componentShown(ComponentEvent *ce*)
void componentHidden(ComponentEvent *ce*)

- **The ContainerListener Interface**

void componentAdded(ContainerEvent *ce*)

void componentRemoved(ContainerEvent *ce*)

- **The FocusListener Interface**

void focusGained(FocusEvent *fe*)
void focusLost(FocusEvent *fe*)

- **The ItemListener Interface**

void itemStateChanged(ItemEvent *ie*)

- **The KeyListener Interface**

This interface defines three methods. The **keyPressed( )** and **keyReleased( )** methods are invoked when a key is pressed and released, respectively. The **keyTyped( )** method is invoked when a character has been entered.

For example, if a user presses and releases the A key, three events are generated in sequence:

key pressed, typed, and released. If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released.

The general forms of these methods are shown here:

void keyPressed(KeyEvent *ke*)
void keyReleased(KeyEvent *ke*)
void keyTyped(KeyEvent *ke*)

- **The MouseListener Interface**

This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked( )** is invoked. When the mouse enters a component, the **mouseEntered( )** method is called. When it leaves, **mouseExited( )** is called. The **mousePressed( )** and **mouseReleased( )** methods are invoked when the mouse is pressed and released, respectively.

The general forms of these methods are shown here:

void mouseClicked(MouseEvent *me*)
void mouseEntered(MouseEvent *me*)
void mouseExited(MouseEvent *me*)
void mousePressed(MouseEvent *me*)
void mouseReleased(MouseEvent *me*)

- **The MouseMotionListener Interface**

This interface defines two methods. The **mouseDragged( )** method is called multiple times as the mouse is dragged. The **mouseMoved( )** method is called multiple times as the mouse is moved. Their general forms are shown here:

void mouseDragged(MouseEvent *me*)
void mouseMoved(MouseEvent *me*)

- **The MouseWheelListener Interface**

This interface defines the **mouseWheelMoved( )** method that is invoked when the mouse wheel is moved. Its general form is shown here:

void mouseWheelMoved(MouseWheelEvent *mwe*)

- **The TextListener Interface**

This interface defines the **textChanged( )** method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

void textChanged(TextEvent *te*)

- **The WindowFocusListener Interface**

This interface defines two methods: **windowGainedFocus( )** and **windowLostFocus( )**. These are called when a window gains or loses input focus. Their general forms are shown here:

void windowGainedFocus(WindowEvent *we*)
void windowLostFocus(WindowEvent *we*)