VPE Project

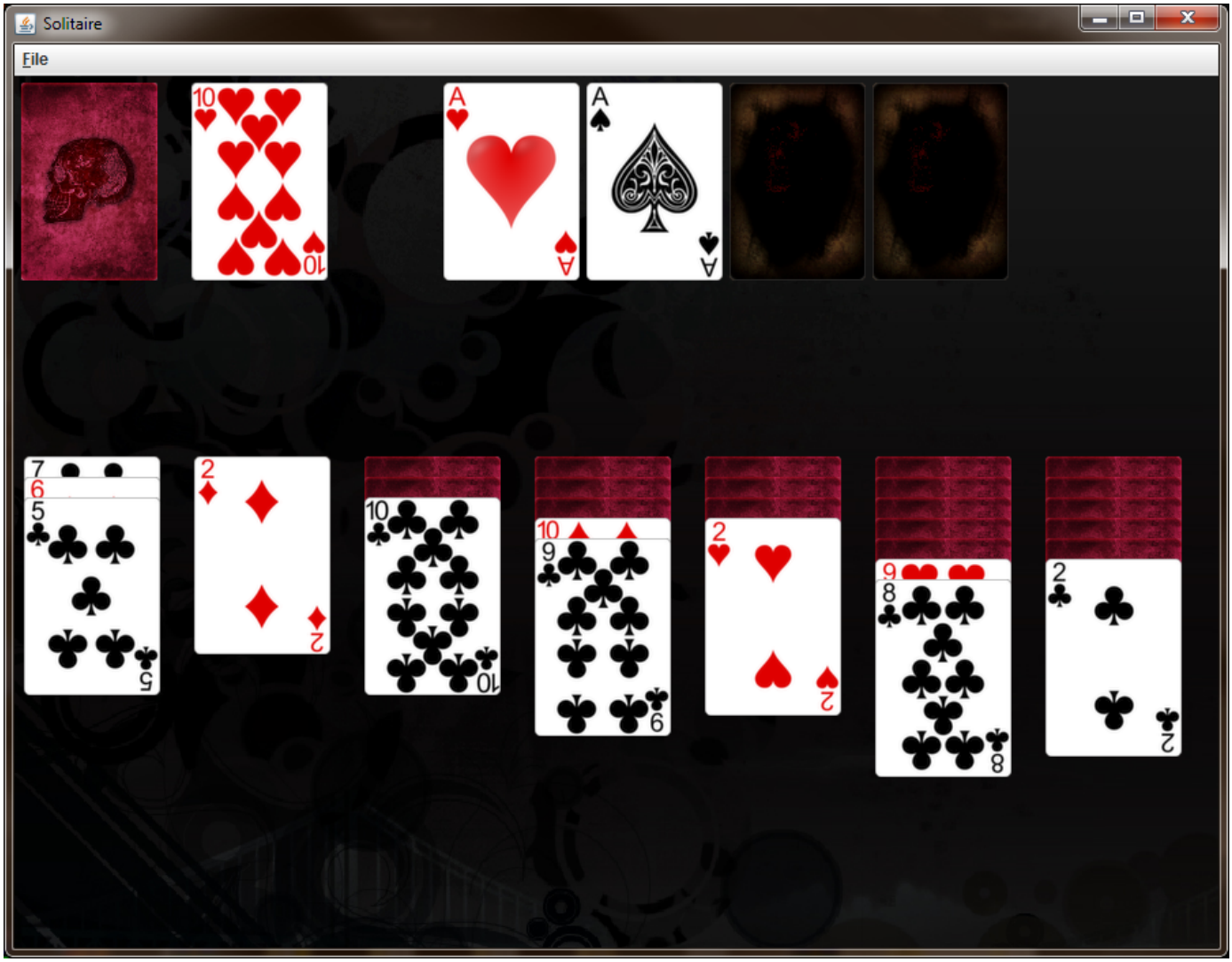
Solitaire

Cristian Buleandră
C.E. 10305 A

-> Mission Statement

Solitaire, or patience, is a genre of tabletop games, consisting of card games that can be played by a single player. This specific game of **Solitaire** is the classic version, in which the purpose is to sort all the cards into 4 different piles (each for every card suit) ascending from **Ace** to **King**.

The purpose of this project is to create a fully functionable solitaire game using Java and the Swing library.



-> About the game

The game starts with a 52 deck of cards. The cards will be arranged into 7 columns (**normal piles**), each having a different number of cards (first column one car, second column two cards, etc.).

The rest of the cards are placed into a separate pile, let's call it the **draw pile**.

In the beginning there are also 5 empty piles:

- 4 for the final piles, where the cards should be in sorted order after the game is won (**final piles**)
- 1 to place (face-up) the cards drew from the draw pile. (let's call it **get pile**)

The player can do several different legal moves:

- Take first card from draw pile and place it into the get pile
- Take first card from the get pile and place it over a normal pile if **condition1** is met
- Take first card from a normal pile and place it over a final pile if **condition2** is met
- Take any number of cards from a normal pile and keeping them in exactly the same order place them over another normal pile, if **condition1** is met

Condition1: The first card from the newly added pile has **distinct color** and a **value smaller with exactly one** than the last card from the pile it is added on.

Condition2: The pile has only one card, it is the **same color** and has a value **larger with exactly one** than the last card from the final pile it is placed on. If the final pile is empty the card must be an **ace**.

The game is won the last card in every final pile is a **King**.

After the game has ended, or during the game, the player can start a new game.

The player can also save or load a game state during gameplay.

-> Use case

The following paragraph describes a possible use case that may be encountered while using the Solitaire program.

1. The User opens the program by double-clicking the Solitaire.jar file.
2. A new game is automatically started.
    a. The user chooses to continue the new game.
        i. He notices that he can place the **2 of hearts** under the **3 of spades**.
        ii. He drags the **2 of hearts** over the pile that contains the **3 of spades**. The two piles are now merged.
        iii. No other valid move is available, so the users decides to draw a new card from the pack by clicking the top-left pile (which is the draw pile).
        iv. He gets a **2 of spades**, which is not useful for him.
        v. He draws another card.
        vi. He gets and **Ace of diamonds**. He decides to place the **Ace of diamonds** by draging the card on one of the four available final piles (top right piles).
        vii. He continues to play the game until he gets bored, at this point he considers saving his progress by accessing the **File->Save** option.
    b. The user chooses to load a previous game.
        i. He choses the **File->Load** option and the previous game is automatically loaded from an **XML** file.
        ii. The user continues playing the game (as described in section **2.a**).
        iii. He finishes the game by dragging the last king over the last queen in the final piles.
        iv. He gets a congratulations messages and decides it's time to go outside and play.
3. The user closes the application by accessing **File->Exit**, or by clicking the **X** button in the top-right corner of the window.

-> Program architecture

The program main class is **Game**, which contains the main function and also creates singular instances of the **GUI** and **Engine**.

The game, as it should, does not use any **global variables**. All variables are members of some class. The GUI instances can access public members and methods from the Engine instance because the Engine is passed as an argument to the GUI's constructor. All the public methods and members can be seen in the UML diagram on the next page.

The **Engine** class contains functions which enable the interaction between one or two **Piles**.

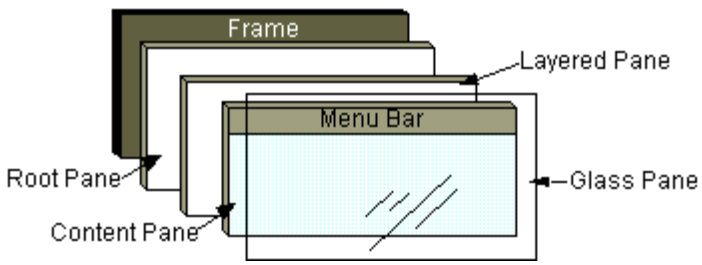
The **GUI** class, except for display all user interface elements and positioning the game objects, it also handles all the interactions done by the user using the mouse or keyboard input.

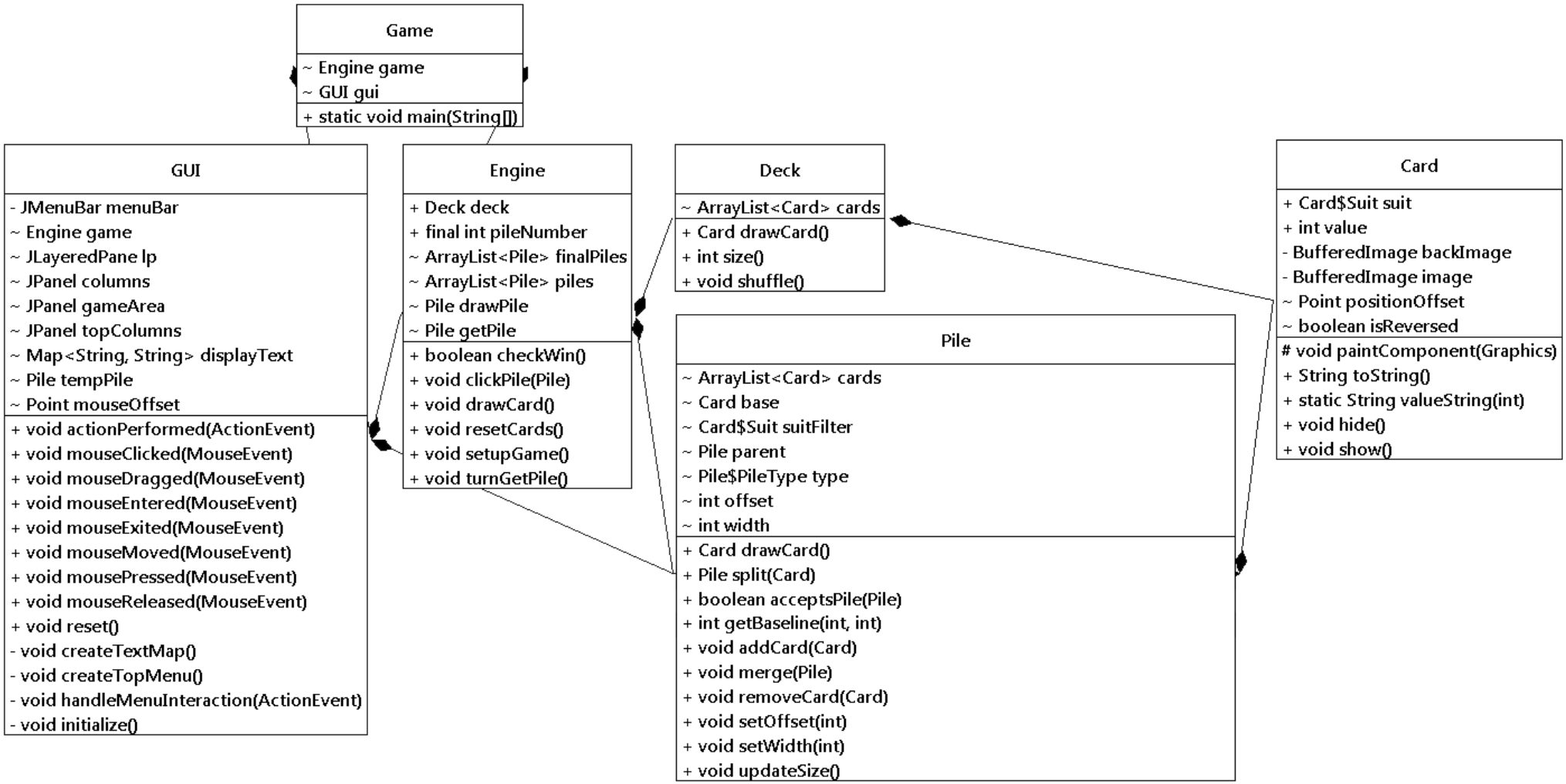
The **Pile** class is a container class which hold zero, one or more **Cards**. This class contains solitaire-specific logic to make sure only legal moves can be performed.

The **Card** class holds information about a single playing card (including the image) and the **Deck** class contains a list of 52 cards.

To easily handle the dragging event, when a drag event is started the **pile** that was clicked gets split into two piles (one that remains still, and another temporarily pile that is absolute positioned). The temporarily file gets merged with another pile when the drag event is complete, or is appended back to the original pile.

The GUI implementation uses the root content pane to display all game content and the default frame layered pane to display the top menu bar. The **layered pane** is also used to display the absolute-positioned **temp pile** while it is being dragged.




Game
~ Engine game
~ GUI gui
+ static void main(String[])
GUI
- JMenuBar menuBar
~ Engine game
~ JLayeredPane lp
~ JPanel columns
~ JPanel gameArea
~ JPanel topColumns
~ Map<String, String> displayText
~ Pile tempPile
~ Point mouseOffset
+ void actionPerformed(ActionEvent)
+ void mouseClicked(MouseEvent)
+ void mouseDragged(MouseEvent)
+ void mouseEntered(MouseEvent)
+ void mouseExited(MouseEvent)
+ void mouseMoved(MouseEvent)
+ void mousePressed(MouseEvent)
+ void mouseReleased(MouseEvent)
+ void reset()
- void createTextMap()
- void createTopMenu()
- void handleMenuInteraction(ActionEvent)
- void initialize()
Engine
+ Deck deck
+ final int pileNumber
~ ArrayList<Pile> finalPiles
~ ArrayList<Pile> piles
~ Pile drawPile
~ Pile getPile
+ boolean checkWin()
+ void clickPile(Pile)
+ void drawCard()
+ void resetCards()
+ void setupGame()
+ void turnGetPile()
Deck
~ ArrayList<Card> cards
+ Card drawCard()
+ int size()
+ void shuffle()
Pile
~ ArrayList<Card> cards
~ Card base
~ Card$Suit suitFilter
~ Pile parent
~ Pile$PileType type
~ int offset
~ int width
+ Card drawCard()
+ Pile split(Card)
+ boolean acceptsPile(Pile)
+ int getBaseline(int, int)
+ void addCard(Card)
+ void merge(Pile)
+ void removeCard(Card)
+ void setOffset(int)
+ void setWidth(int)
+ void updateSize()
Card
+ Card$Suit suit
+ int value
- BufferedImage backImage
- BufferedImage image
~ Point positionOffset
~ boolean isReversed
# void paintComponent(Graphics)
+ String toString()
+ static String valueString(int)
+ void hide()
+ void show()

Java AWT Events

Summary

Java AWT package contains all of the classes for creating user interfaces and for painting graphics and images.

The Abstract Window Toolkit (AWT) is Java's original platform-independent windowing, graphics, and user-interface widget toolkit. The AWT is part of the Java Foundation Classes (JFC) — the standard API for providing a graphical user interface (GUI) for a Java program. AWT is also the GUI toolkit for a number of Java ME profiles. For example, Connected Device Configuration profiles require Java runtimes on mobile telephones to support AWT.

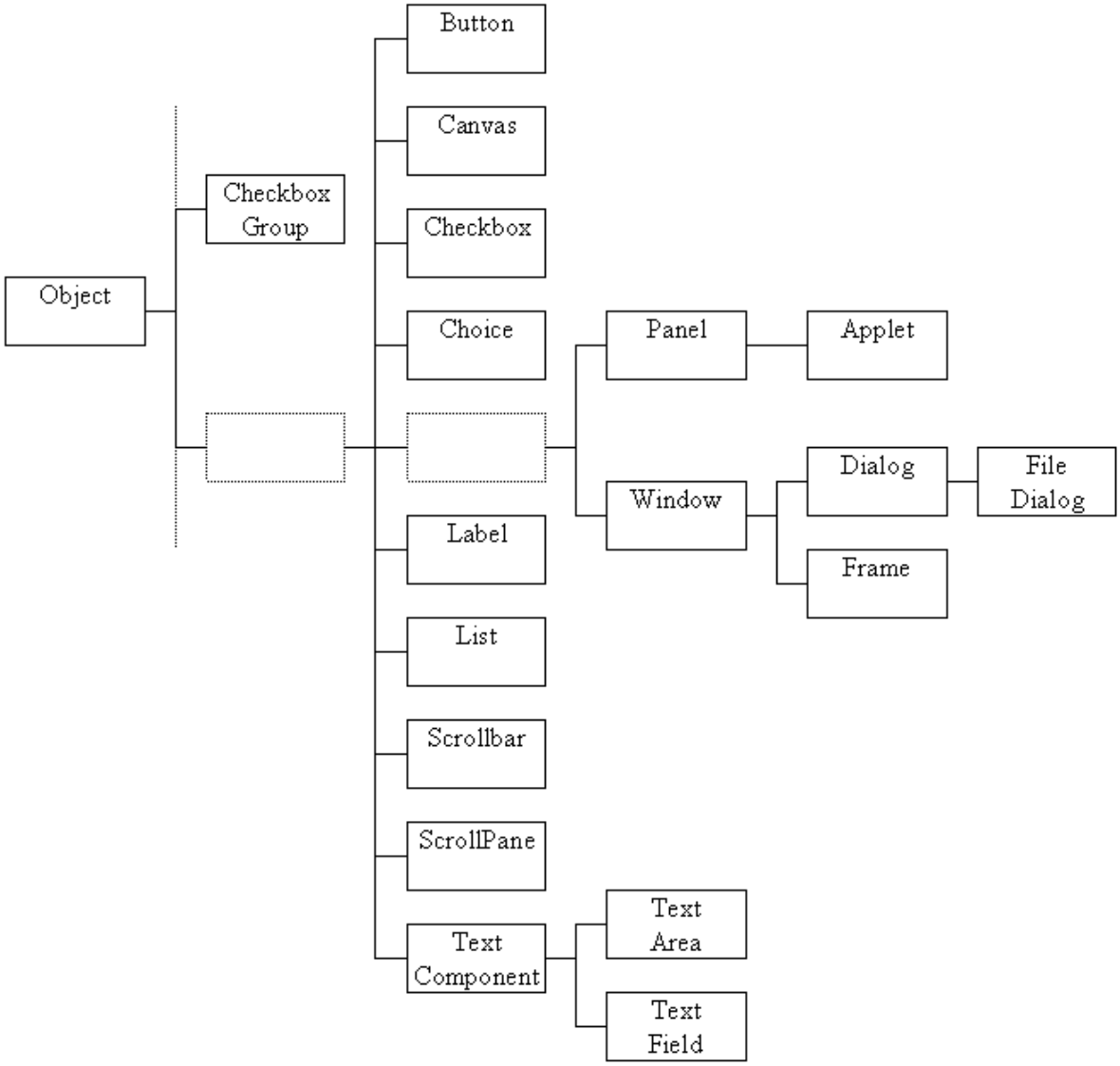
The AWT provides two levels of APIs:

- A general interface between Java and the native system, used for windowing, events, and layout managers. This API is at the core of Java GUI programming and is also used by Swing and Java 2D. It contains:
    - The interface between the native windowing system and the Java application;
    - The core of the GUI event subsystem;
    - Several layout managers;
    - The interface to input devices such as mouse and keyboard; and
    - A java.awt.datatransfer package for use with the Clipboard and Drag and Drop.
- A basic set of GUI widgets such as buttons, text boxes, and menus. It also provides the AWT Native Interface, which enables rendering libraries compiled to native code to draw directly to an AWT Canvas object drawing surface.

AWT also makes some higher level functionality available to applications, such as:

- Access to the system tray on supporting systems; and
- The ability to launch some desktop applications such as web browsers and email clients from a Java application.

The Java AWT Component Class Hierarchy

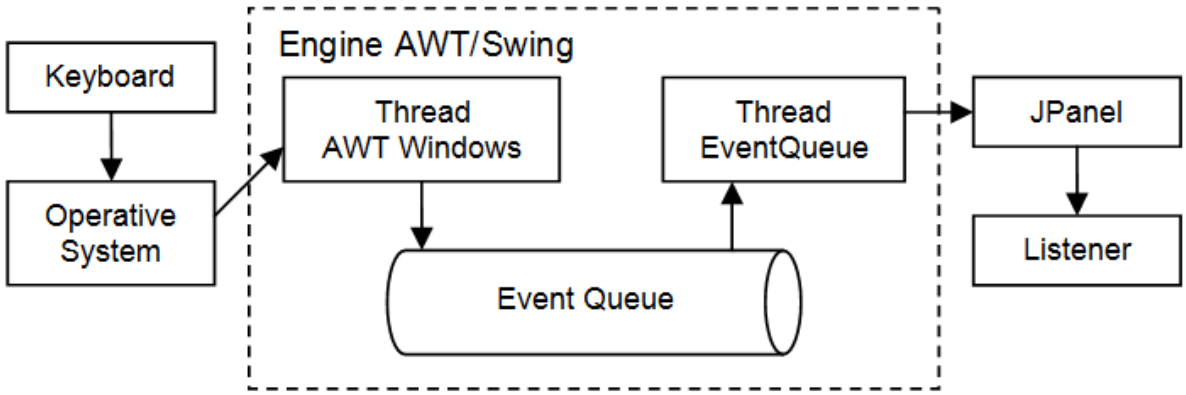


Java AWT Events

The **AWT Event** package provides interfaces and classes for dealing with different types of events fired by AWT components.

The members of this package fall into three categories:

- **Events -** The classes with names ending in "Event" represent specific types of events, generated by the AWT or by one of the AWT components.
- **Listeners -** The interfaces in this package are all "event listeners"; their names end with "Listener." These interfaces define the methods that must be implemented by any object that wants to be notified when a particular event occurs. Note that there is a Listener interface for each Eventclass.
- **Adapters** - Each of the classes with a name ending in "Adapter" provides a no-op implementation for an event listener interface that defines more than one method. When you are only interested in a single method of an event listener interface, it is easier to subclass an Adapter class than to implement all of the methods of the corresponding Listener interface.



What is an Event?

Change in the state of an object is known as event i.e. event describes the change in state of source. Events are generated as result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard,selecting an item from list, scrolling the page are the activities that causes an event to happen.

Types of Event

The events can be broadly classified into two categories:

- **Foreground Events** - Those events which require the direct interaction of user.They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard,selecting an item from list, scrolling the page etc.
- **Background Events** - Those events that require the interaction of end user are known as background events. Operating system interrupts,

hardware or software failure, timer expires, an operation completion are the example of background events.

What is Event Handling?

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.
- **Listener** - It is also known as event handler.Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event and then returns.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model ,Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- the method is now get executed and returns.

Points to remember about listener

- In order to design a listener class we have to develop some listener interfaces.These Listener interfaces forecast some public abstract callback methods which must be implemented by the listener class.
- If you do not implement the any if the predefined interfaces then your class can not act as a listener class for a source object.

Callback Methods

These are the methods that are provided by API provider and are defined by the application programmer and invoked by the application developer. Here the callback methods represents an event method. In response to an event java jre will fire callback method. All such callback methods are provided in listener interfaces.

If a component wants some listener will listen to it's events the the source must register itself to the listener.

Here is a list of the commonly used **AWT event types**, with use examples:

- **ActionEvent**

```
int uniqueId = System.currentTimeMillis().intValue();
String commandName = ""; //it can be like "show" or "hide" or whatever else;
                         //you can get this string with getActionCommand() method
                         //and make some actions based on its value
                         //... but you don't need it now
ActionEvent event = new ActionEvent(this, uniqueId, commandName);
```

- **KeyEvent**

```
   /** Handle the key typed event from a text field. */
   public void keyTyped(KeyEvent e) {
       displayInfo(e, "KEY TYPED: ");
   }
```

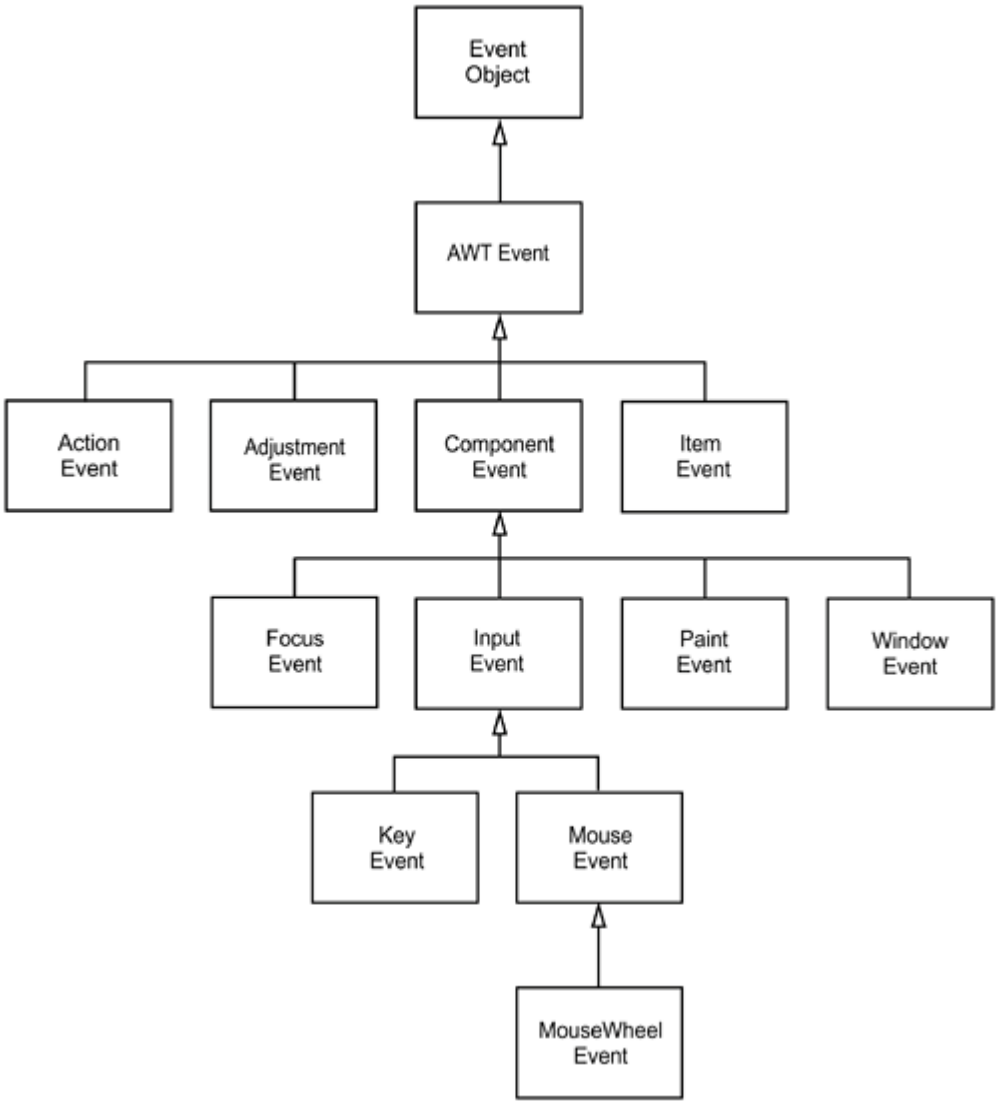
- **AdjustmentEvent**

```
 AdjustmentListener vListener = new AdjustmentListener() {
   @Override
   public void adjustmentValueChanged(AdjustmentEvent event) {
       System.out.println("Vertical: ");
       dumpInfo(event);
   }
 };
  JScrollBar vscrollBar = scrollPane.getVerticalScrollBar();
  vscrollBar.addAdjustmentListener(vListener);
```

- **MouseEvent**

```
public class MouseEventDemo ... implements MouseListener {
       //where initialization occurs:
       //Register for mouse events on blankArea and the panel.
       blankArea.addMouseListener(this);
       addMouseListener(this);
   ...

   public void mousePressed(MouseEvent e) {
      saySomething("Mouse pressed; # of clicks: "
                   + e.getClickCount(), e);
   }
}
```

- FocusEvent
- MouseWheelEvent
- ItemEvent
- WindowEvent

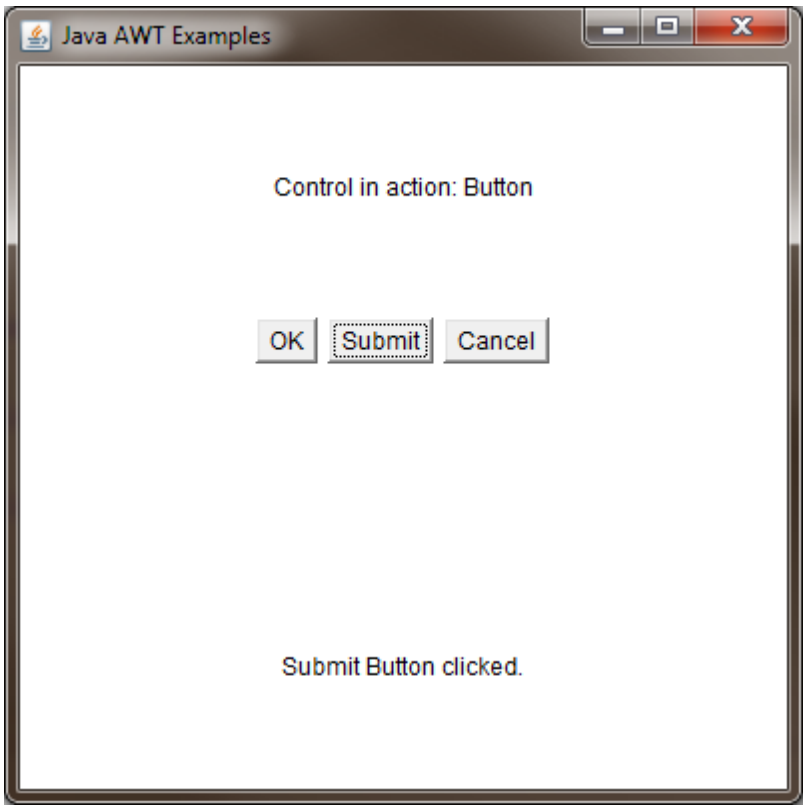

Event Object
AWT Event
Action Event
Adjustment Event
Component Event
Item Event
Focus Event
Input Event
Paint Event
Window Event
Key Event
Mouse Event
MouseWheel Event

Event Handling Example

```
import java.awt.*;
import java.awt.event.*;

public class AwtControlDemo {

  private Frame mainFrame;
  private Label headerLabel;
  private Label statusLabel;
  private Panel controlPanel;

  public AwtControlDemo(){
    prepareGUI();
  }

  public static void main(String[] args){
    AwtControlDemo  awtControlDemo = new AwtControlDemo();
    awtControlDemo.showEventDemo();
  }

  private void prepareGUI(){
    mainFrame = new Frame("Java AWT Examples");
    mainFrame.setSize(400,400);
    mainFrame.setLayout(new GridLayout(3, 1));
    mainFrame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent windowEvent){
        System.exit(0);
      }
    });
    headerLabel = new Label();
    headerLabel.setAlignment(Label.CENTER);
    statusLabel = new Label();
    statusLabel.setAlignment(Label.CENTER);
    statusLabel.setSize(350,100);

    controlPanel = new Panel();
    controlPanel.setLayout(new FlowLayout());

    mainFrame.add(headerLabel);
    mainFrame.add(controlPanel);
    mainFrame.add(statusLabel);
    mainFrame.setVisible(true);
  }

  private void showEventDemo(){
    headerLabel.setText("Control in action: Button");

    Button okButton = new Button("OK");
    Button submitButton = new Button("Submit");
    Button cancelButton = new Button("Cancel");

    okButton.setActionCommand("OK");
    submitButton.setActionCommand("Submit");
    cancelButton.setActionCommand("Cancel");

    okButton.addActionListener(new ButtonClickListener());
    submitButton.addActionListener(new ButtonClickListener());
    cancelButton.addActionListener(new ButtonClickListener());
```

```
    controlPanel.add(okButton);
    controlPanel.add(submitButton);
    controlPanel.add(cancelButton);

    mainFrame.setVisible(true);
  }

  private class ButtonClickListener implements ActionListener{
    public void actionPerformed(ActionEvent e) {
      String command = e.getActionCommand();
      if( command.equals( "OK" ))  {
        statusLabel.setText("Ok Button clicked.");
      }
      else if( command.equals( "Submit" ) )  {
        statusLabel.setText("Submit Button clicked.");
      }
      else  {
        statusLabel.setText("Cancel Button clicked.");
      }
    }
  }
}
```

The result of compiling and executing this file is:



In this example every button has an **actionListener** attach, which actually is the same listener for all. Each button sends a different command when clicked, so when the listener receives the **actionPerformed** it knows which button was clicked by checking the **event.getActionCommand()** string.

Another event present in this example, and most of the Java AWT applications, is the window event for closing. The main frame adds a window listener which listens to a **new WindowAdapter** that implements the **windowClosing** method which is called when the window gets closed by the user or the system. In this method the whole program is shut down by using the **System.exit(0)** function call.