Basics of the Object-Oriented Graphical User interface

Associate Profesor Viorica Rozina Chifu
viorica.chifu@cs.utcluj.ro

Object Class

- **getClass**() method
  - You cannot override **getClass**
  - The **getClass**() method returns a **Class** object
    - **Class** class has methods you can use to get information about the class, such as:
      - Its name - **getSimpleName**() method
      - Its superclass - **getSuperclass**() method
      - The interfaces it implements - **getInterfaces**() method

Object Class

- **getClass**() method - Example of displaying the class name of an object:

```
public class Test
{
    public static void main(String[] args)
       {
          Object obj = new String("GeeksForGeeks");
          Class c = obj.getClass();
          System.out.println("Class of Object obj is : " + c.getName());
      }
}
```

>> Output: Class of Object **obj** is java.lang.String

Object Class

- **finalize**() method
  - Is called by the garbage collector before freeing up the memory allocated to an object that has become eligible for garbage collection
  - The purpose of the method is to provide an opportunity for the object to perform any necessary cleanup before it is destroyed
  - The **finalize**() method in the Object class does nothing, but you can override it in your own classes to provide cleanup code, such as
    - closing file descriptors, releasing network connections, or releasing other system resources
  - However, relying on the **finalize**() method to perform cleanup is not recommended, as there is no guarantee that it will be called, or when it will be called
    - In fact, the JVM may choose not to call the finalize() method at all, or it may delay calling it until later, which can cause resource leaks and other problems
  - Therefore, it's best practice to explicitly release system resources, such as file descriptors or network connections, by calling appropriate cleanup methods, rather than relying on the **finalize**() method to do it for you.

Object Class

- **hashCode**() method
  - For every object, JVM generates a unique number which is hashcode
  - It returns distinct integers for distinct objects
  - The method convert the internal address of object to an integer by using an algorithm
- By definition, if two objects are equal, their hash code must also be equal
- If you override the **equals**() method, you change the way two objects are compared to be equals and Object's implementation of **hashCode**() is no longer valid
- Therefore, if you override the **equals**() method, you must also override the **hashCode**() method as well

Object Class

- **hashCode**() method - **Use of hashCode()**
  - JVM(Java Virtual Machine) uses hashcode method while saving objects into hashing related data structures like HashSet, HashMap, Hashtable, etc.
  - The main advantage of saving objects based on hash code is that searching becomes easy

Object Class

- **hashCode**() method –Example of implementation

```
public class Person {
    private String name;
    private int age;

    // constructor and other methods here

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + age;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }

    // equals method here
}
```

- ✓ In this implementation, the hashCode() method calculates the hash code of the Person object based on its name and age fields.
- ✓ The prime number is used to provide a good distribution of hash codes and reduce the likelihood of collisions.
- ✓ It's important to note that the **hashCode**() method should be consistent with the **equals**() method, which means that two objects that are equal according to the equals() method should have the same hash code.
- ✓ Therefore, it's recommended to include all fields used in the equals() method in the calculation of the hash cod

Object Class

- **hashCode**() method – Example of implementation(available forJDK 7 and above)

```
import java.util.Objects;

public class Person {
    private String name;
    private int age;
    // constructor and other methods here

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Person)) return false;
        Person person = (Person) o;
        return age == person.age && Objects.equals(name, person.name);
    }
    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }
}
```

- ✓ In this implementation, the **equals**() method uses the **Objects.equals**() method to check for equality of the name and age fields and returns **true** if they are equal.
- ✓ It also checks if the given object is an instance of the Person class to avoid ClassCastException errors.
- ✓ The **hashCode**() method uses the **Objects.hash**() method to generate a hash code for the name and age fields.
- ✓ The **Objects.hash**() method combines the hash codes of the given fields using a formula that takes into account their values and types and produces a good distribution of hash codes

Object Class

- **clone**() method
  - Is used to create a copy of an object
  - It creates a new instance of the class of the current object and initializes all its fields with exactly the contents of the corresponding fields of this object
  - The clone() method is a protected method that is defined in the Object class, which means that it is inherited by all classes in Java
  - To use the clone() method, a class must implement the Cloneable interface, which is a marker interface that indicates that the class supports cloning.
  - The clone() method returns a reference to the cloned object, which must be cast to the appropriate type.
  - The clone() method throws a CloneNotSupportedException if the object's class does not support cloning.

Object Class

- Syntax of the clone() method is :
  - **protected** Object clone() **throws** CloneNotSupportedException

- **Creating a copy using the clone() method**
  - Step1: The **java.lang.Cloneable** interface must be implemented by the class whose object clone we want to create
    - » If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**
  - Step 2**: clone() method from Object class need to be override**

Object Class

- Here's an example of how to create a copy of an object using the clone() method:

```
public class Person implements
Cloneable {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public Object clone() throws
CloneNotSupportedException {
        return super.clone(); // Shallow copy
    }

    // Other methods here...
}
```

- ✓ In this example, the Person class implements the Cloneable interface to indicate that it supports cloning.
- ✓ The clone() method is overridden to call the super.clone() method, which creates a shallow copy of the object.
- ✓ Note that if Person had any mutable fields, we would need to create a deep copy of those fields in the clone() method to ensure that the cloned object is truly independent of the original object.

Object Class

- In Java, when you create a copy of an object using the clone() method, the copy created is a shallow copy
- A shallow copy means that a new object is created with the same values for its fields as the original object, but any reference variables in the original object are also copied by reference, not by value
- In other words, a shallow copy of an object only copies the object itself and its primitive data types, but not any objects referenced by the object
- If the object contains reference variables, only the references are copied, not the actual objects they refer to
- Therefore, the copied object and the original object will both refer to the same objects in memory

Object Class

- In Java, when you create a copy of an object using the clone() method, the copy created is a shallow copy
- A shallow copy means that a new object is created with the same values for its fields as the original object, but any reference variables in the original object are also copied by reference, not by value
- In other words, a shallow copy of an object only copies the object itself and its primitive data types, but not any objects referenced by the object
- If the object contains reference variables, only the references are copied, not the actual objects they refer to
- Therefore, the copied object and the original object will both refer to the same objects in memory

Object Class

- Here's an example to illustrate this concept

```
public class Person implements Cloneable {
    private String name;
    private int age;
    private Address address;

    public Person(String name, int age, Address address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone(); // Shallow copy
    }
    // Other methods here...
}
```

```
public class Address {
    private String street;
    private String city;

    public Address(String street, String city)
{
        this.street = street;
        this.city = city;
    }

    // Other methods here...
}
```

Object Class

- In this example, the Person class has a field called address, which is an object of the Address class
- If we create a shallow copy of a Person object using the clone() method, the copy will have a new Person object with the same name, age, and address values as the original Person object
- However, the address field in the copied Person object will refer to the same Address object as the address field in the original Person object
- This means that if we modify the Address object referred to by the address field in the copied Person object, the Address object referred to by the address field in the original Person object will also be modified
- This is because both objects are referring to the same Address object in memory

Object Class

- In object-oriented programming, deep copy refers to the process of creating a new object that is an exact copy of an existing object, but with all of its fields and sub-objects also copied over recursively
- In other words, when you create a deep copy of an object, all of its dependent objects are also copied and not just referenced
- Deep copying is useful when you want to create a completely independent copy of an object that can be modified without affecting the original object or any other copies of it

Object Class

- Example of deep copy

```
public class Person implements Cloneable {
    private String name;
    private int age;
    private Address address;
    public Person(String name, int age, Address address) {
        this.name = name;
        this.age = age;
        this.address = address;    }
    @Override
    public Object clone() throws CloneNotSupportedException {
        // Create a new Person object with the same name and age
        Person clonedPerson = (Person) super.clone();
        // Create a new Address object with the same street and city
        Address clonedAddress = (Address) address.clone();
        // Set the clonedAddress object in the clonedPerson object
        clonedPerson.setAddress(clonedAddress);
        return clonedPerson; // Deep copy
    }
    // Other methods here...}
```

```
public class Address implements Cloneable {
    private String street;
    private String city;

    public Address(String street, String city) {
        this.street = street;
        this.city = city;
    }

    @Override
    public Object clone() throws
CloneNotSupportedException {
        return super.clone(); // Shallow copy
    }

    // Other methods here...
}
```

Graphical User interface

Graphical User Interface (GUI)

- GUI
  - Refers to all visual communication types between a program and their users
  - Presents a friendly mechanism user interaction with the program
  - Give a look and feel to the program
  - It allows users to feel more familiar with the program even before it will be used
  - Reduce the learning time for usage
- Java language provides predefined classes to implement a GUI

Graphical User Interface (GUI)

- We can create java applications with a graphical user interface by using:
  - AWT(Abstract Windowing Toolkit) components
    - »Is an API for providing a graphical user interface (GUI) for a Java program
  - Swing components
    - »Part of the JFC (Java Foundation Classes) project created by Sun, Netscape si IBM
    - »Swing is built on top of AWT, and extends the functionality provided by AWT by adding new components and replacing old ones. It also has a pluggable look and feel, which means that developers can choose from different themes or styles for their GUI applications.
    - »Swing offers a large collection of classes and interfaces for developing GUI applications, including buttons, labels, text fields, menus, dialog boxes, and more. It also provides advanced features such as drag-and-drop support, data binding, and internationalization.
    - »Overall, Swing is a powerful and flexible GUI library that is widely used for developing desktop applications in Java.Is recommended to develop java application with SWING technology

Swing

- Includes:
  - Swing components
    - » Replace and extend the old setoff components provided by AWT model
  - Look-and-Feel
    - » Allow to change the application look and interaction mode with application based on user preferences
  - Java 2D API
    - » Allows to create applications that use graphics at advanced level
  - Internationalization
    - » Allow to develop applications that can be configured to be explored in various part of the world

Swing - API

- Includes 17 packages consisting of classes and interfaces
  - javax.accessibility, javax.swing.plaf
  - javax.swing.text.html
  - javax.swing
  - javax.swing.plaf.basic
  - javax.swing.text.parser
  - javax.swing.border
  - javax.swing.plaf.metal
  - javax.swing.text.rtf
  - javax.swing.colorchooser
  - javax.swing.plaf.multi, javax.swing.tree
  - javax.swing.event, javax.swing.table
  - javax.swing.undo
  - javax.swing.filechooser
  - javax.swing.text

Swing - API

- **javax.swing**
  - Is the most important package from Swing
  - Contains the basic components to create graphical user interface
  - Atomic components
    - » **JLabel, JButton, JCheckBox, JRadioButton, JToggleButton, JScrollBar, JSlider, JProgressBar, JSeparator**
  - Complex components
    - » **JTable, JTree, JComboBox, JSpinner, JList, JFileChooser, JColorChooser, JOptionPane**
  - Components for editing text
    - » **JTextField, JFormattedTextField, JPasswordField, JTextArea, JEditorPane, JTextPane**
  - Menus
    - » **JMenuBar, JMenu, JPopupMenu, JMenuItem, JCheckboxMenuItem, JRadioButtonMenuItem**
  - Intermediate containers
    - » **JPanel, JScrollPane, JSplitPane, JTabbedPane, JDesktopPane, JToolBar**
  - High level containers
    - » **JFrame, JDialog, JWindow, JInternalFrame, JApplet**

SWING versus AWT

- Swing
  - Extends the AWT model with new components
  - Introduces a new package that describes the events for the SWING components
    - » **javax.swing.event**
  - Introduces new classes for position
    - » **BoxLayout, SpringLayout**
- AWT
  - Same classes (e.g. *Color*, *Font*, etc. from **java.awt** package) did not rewrite in SWING
  - **java.awt.event** package is still used event handling
  - The components positioning is taken from AWT

Graphical User Interface (GUI)

- Steps to create a GUI:
  - **Define a display area**: The first step is to create a display area for your GUI. This is typically done by creating a window or frame using the **JFrame** class, which provides a top-level container for your GUI components.
  - **Define and place graphical objects:** Once you have a display area, you can start defining the graphical objects that will be displayed in the window. This can include buttons, text fields, labels, and other components. You will use layout managers, such as FlowLayout or GridLayout, to position these components in the window.
  - **Define actions**: For each graphical object in your GUI, you will need to define the actions that should be executed when the user interacts with them. This can include things like button clicks, text input, or menu selections.
  - **Listen for events**: To capture user input, you will need to listen for events that are generated by the graphical objects in your GUI. This is done by attaching event listeners to the components, which will trigger a callback method when the user interacts with the component.
  - **Execute actions**: When an event is triggered, the corresponding event listener will call the appropriate action method to execute the desired behavior in response to the user's input.

Component class

- A component is any object that:
  - Has a graphical representation that can be displayed on the screen
  - Can interacts with the user
- **Component** class
  - Abstract class
  - Superclass for all non-menu-related elements that are used to develop a graphical user interface
  - Defines the basic functionality and behavior of a graphical component, such as its size, position, and visibility.
  - Provides methods for handling input events, painting the component, and managing focus.
  - All of the classes that describe graphical components (such as buttons, text fields, and labels) in Java extend the **Component** class, either directly or indirectly through other subclasses
  - By extending the **Component** class, these classes inherit the basic functionality and behavior of a graphical component, while also providing their own unique features and properties

Component class

- **Position**: The Component class provides several methods for managing the **position** of a component:
  - getLocation() returns the current x and y coordinates of the component's upper-left corner as a Point object.
  - getX() and getY() return the current x and y coordinates of the component's upper-left corner as separate integers.
  - setLocation(x, y) sets the position of the component's upper-left corner to the specified x and y coordinates.
  - setX(x) and setY(y) set the component's x and y coordinates to the specified values, respectively.

Component class

- **Dimension**: The Component class provides several methods for managing the size of a component:
  - getSize() returns the current width and height of the component as a Dimension object.
  - getWidth() and getHeight() return the current width and height of the component as separate integers.
  - setSize(width, height) sets the size of the component to the specified width and height.
  - setWidth(width) and setHeight(height) set the component's width and height to the specified values, respectively

Component class

- **Dimension and position:** The Component class also provides methods for managing both the position and size of a component:
  - getBounds() returns the current position and size of the component as a Rectangle object.
  - setBounds(x, y, width, height) sets the position and size of the component to the specified values

Component class

- **Color:** The Component class provides methods for managing the text and background colors of a component:
  - getForeground() returns the current text color of the component.
  - getBackground() returns the current background color of the component.
  - setForeground(color) sets the text color of the component to the specified color.
  - setBackground(color) sets the background color of the component to the specified color.

AWT Model - AWT Components

- Visibility: The Component class provides methods for managing the visibility of a component
  - setVisible(boolean visible): Sets the visibility of the component to the specified boolean value.
  - isVisible(): Returns a boolean indicating whether the component is currently visible or not.
- Interaction: The Component class provides methods for managing the interaction with a component
  - setEnabled(boolean enabled): Sets the enabled state of the component to the specified boolean value.
  - isEnabled(): Returns a boolean indicating whether the component is currently enabled or not.

Container Class

- A container is:
  - A surface on which we can place graphical objects
  - Instance of a class that extends the **Container** class
- The Container class is a subclass of the Component class in AWT.
- It is an abstract class that provides methods to add and remove components from a container, to set the layout manager for the container, and to manage the focus traversal policy for the container.
- Some of the commonly used methods in the Container class are:
  - add(Component comp): Adds the specified component to the container.
  - remove(Component comp): Removes the specified component from the container.
  - setLayout(LayoutManager mgr): Sets the layout manager for the container.
  - getComponents(): Returns an array of all the components in the container.
  - setFocusTraversalPolicy(FocusTraversalPolicy policy): Sets the focus traversal policy for the container.
  - validate(): Validates this container and all of its subcomponents.
  - repaint(): Requests that the container and all of its subcomponents be repainte

JComponent

- JComponent is a subclass of the Container class in Java that provides a more advanced and customizable set of graphical components for building user interfaces.
- JComponent provides a number of additional methods for handling events, customizing appearance, and managing layout.
- JComponent is the base class for many Swing components, such as JButton, JLabel, JTable, and JTree.
- It provides a number of methods that are common to all Swing components, such as setPreferredSize(), setEnabled(), setVisible(), and setToolTipText().

Layout manager

- Object that controls the size and position of the components within a container
- It is responsible for arranging the components in a container according to a specific layout strategy.
- A layout manager is an object that implements the LayoutManager interface
- When a container is instantiated, a default layout manager associated with the container is created
- However, this default layout manager can be changed to a different layout manager using the setLayout() method.
- Using layout managers is an important part of developing a GUI application, as it helps ensure that the components are arranged in a consistent and visually appealing manner across different platforms and screen size

Layout manager

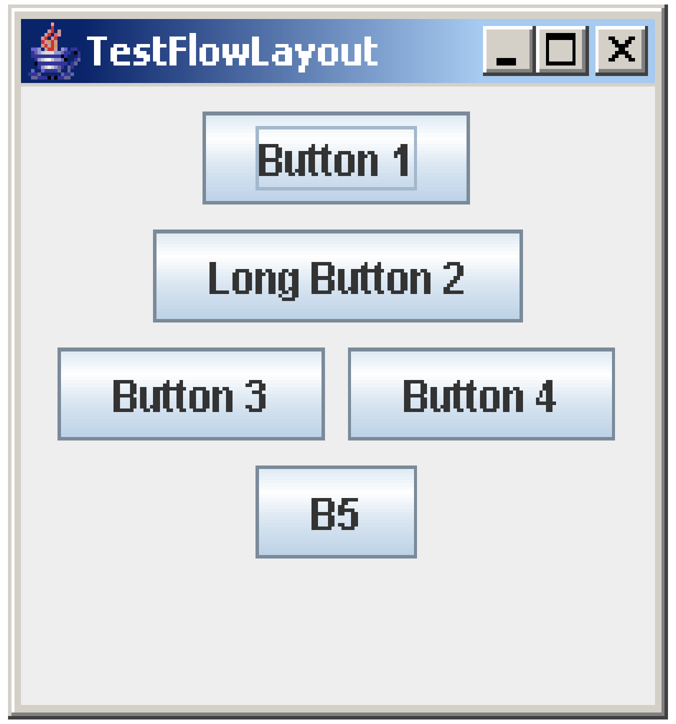
- Examples of Layout Manager
  - FlowLayout
  - BorderLayout
  - GridLayout
  - CardLayout
  - GridBagLayout

Layout manager - FlowLayout

- This layout manager arranges components horizontally from left to right, one by one, on lines
- If there is not enough space in a row, the next component is moved to the next row
- Is the default layout for the classes that extends the **Panel** class
- The dimension of the displayed components are automatically received by layout by means of **getPreferredSize** which is implemented by all standard component

Layout manager - Example of using FlowLayout

```
import javax.swing.*;
public class TestLayout {
public static void main(String args[]) {
   JFrame f = new JFrame(“TestFlowLayout");
   //FlowLayout f1= new  FlowLayout ();
  // f.setLayout(f1);
   f.setLayout(new FlowLayout());
   JButton b1 = new JButton("Button 1");
   JButton b2 = new JButton(“LongButton2");
   JButton b3 = new JButton("Button 3");
   JButton b4 = new JButton(" Button 4");
   JButton b5 = new JButton("B5");
   f.add(b1);
   f.add(b2);
   f.add(b3);
   f.add(b4);
   f.add(b5);
   f.setVisibile(true);} }
```



Layout manager - BorderLayout

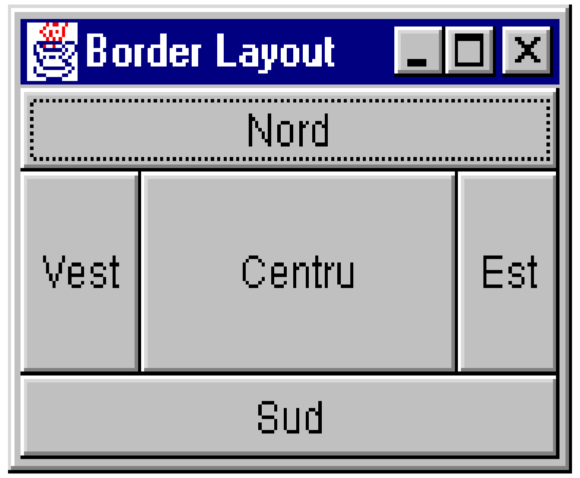
- This layout manager divides the container into five areas: north, south, east, west, and center.
- When you add a new component, you need to specifiy the position

panel.**add**(component, **BorderLayout.NORTH**)

- Each region may contain no more than one component
  - To add more than one component into the same region, you must use a panel to group the components; then you add the panel in that region
  - Is the default layout JFrame class

Layout manager - Example of using BorderLayout

```
port javax.swing.*;
public class TestBorderLayout {
  public static void main(String args[]) {
     JFrame f = new JFrame("Border Layout");
     f.setLayout(new BorderLayout());
     f.add(new JButton("Nord"), BorderLayout.NORTH);
     f.add(new JButton("Sud"), BorderLayout.SOUTH);
     f.add(new JButton("Est"), BorderLayout.EAST);
     f.add(new JButton("Vest"), BorderLayout.WEST);
     f.add(new JButton("Centru"),
BorderLayout.CENTER);
     f.setVisible(true); } }
```

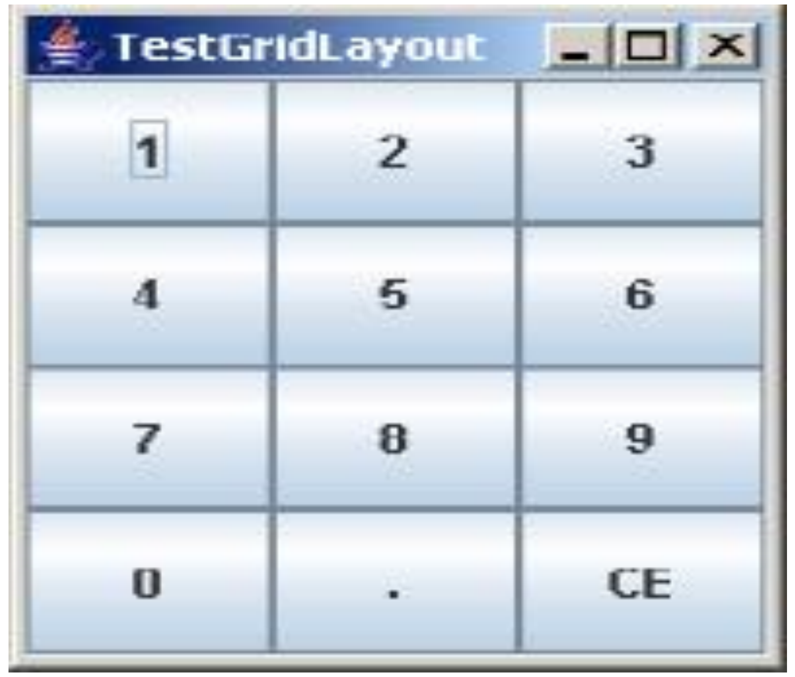


Layout manager - Grid Layout

- Places components in a grid of cells (the number of rows and columns are fixed)
- Each component takes all the available space within its cell, and each cell is exactly the same size
- If the window is resized, the **GridLayout** object changes the cell size so that the cells are as large as possible, given the space available to the container
- Add the components one by one from left to right

Layout manager - Example of using GridLayout

```
import java.awt.*;
public class TestBorderLayout {
  public static void main(String args[]) {
     JFrame f = new JFrame(“TestGirdLLayout");
     JPanel panel = new JPanel();
     JButton button1= new JButton(“1”);
     JButton button2= new JButton(“2”);
     JButton button3= new JButton(“3”);
     JButton button4= new JButton(“4”);
      …….
     panel.setLayout(new GridLayout(4, 3));
     panel.add(button1);
     panel.add(button2);
     panel.add(button3);
     panel.add(button4);
     …..
     f.add(panel);
     f.setVisible(true);}}
```



Layout manager - CardLayout

- Treats each component in the container as a card
- Only one card is visible at a time, and the container acts as a deck of cards
- The first component added to a **CardLayout** object is the visible component when the container is first displayed
- Class has methods to:
  - Display a specific component for the deck
  - Traverse the deck sequentialy, the order in which the components are in the deck is inner to the layout

Layout manager - CardLayout

```
import javax.swing.*;
import java.awt.*;

public class CardLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Card Layout Example");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 400);

        // create a panel with card layout
        JPanel cardPanel = new JPanel(new CardLayout());

        // create two panels with different colors
        JPanel panel1 = new JPanel();
        panel1.setBackground(Color.RED);
        JPanel panel2 = new JPanel();
        panel2.setBackground(Color.BLUE);
```

```
// add the panels to the card panel with unique names
        cardPanel.add(panel1, "panel1");
        cardPanel.add(panel2, "panel2");

        // create a button to switch between the panels
        JButton button = new JButton("Switch Panels");
        button.addActionListener(e -> {
            CardLayout cardLayout = (CardLayout)
cardPanel.getLayout();
            cardLayout.next(cardPanel);
        });

        // add the button and the card panel to the frame
        frame.add(button, BorderLayout.NORTH);
        frame.add(cardPanel, BorderLayout.CENTER);

        frame.setVisible(true);
    }
}
```

Layout manager - GridBagLayout

- This layout manager is the most flexible and powerful of all the layout managers.
- It arranges components in a grid, but each component can take up different amounts of space and be positioned in different ways.
  - Not all rows necessarily have the same height
  - Similarly, not all columns necessarily have the same width

Layout manager - GridBagLayout

- GridBagConstraints
  - is a class in Java that is used to define constraints for placing components in a GridBagLayout.
  - It specifies how much space a component should occupy, where the component should be placed, and how much padding should be added around the component.

Layout manager – GridBagLayout

- Types of constraints that can be specified by means of the variables of **GridBagConstraints** class:
  - gridx, gridy
    - » Specify the row and column at the upper left of the component
    - » The left most column has gridx = 0 and the top row has gridy=0
  - gridwidth, gridheight
    - » Specify the number of columns (for gridwidth) or rows (for gridheight) in the component's display area
    - » These constraints specify the number of cells the component uses, not the number of pixels it uses. The default value is 1.

Layout manager – GridBagLayout

- Types of constraints that can be specified by means of the variables of **GridBagConstraints** class:
  - fill
    - Used when the component's display area is larger than the component's requested size to determine whether and how to resize the component
    - Valid values (defined as GridBagConstraints constants) include
      - NONE (the default)
      - HORIZONTAL (make the component wide enough to fill its display area horizontally, but do not change its height)
      - VERTICAL (make the component tall enough to fill its display area vertically, but do not change its width)
      - BOTH (make the component fill its display area entirely)
  - insets
    - Specify the distance between the component and the bounds of its display area

Layout manager – GridBagLayout

- Types of constraints that can be specified by means of the variables of **GridBagConstraints** class:
  - **anchor**
    - Used when the component is smaller than its display area to determine where (within the area) to place the component
      - Valid values (defined as GridBagConstraints constants) are
        - CENTER (the default)
        - PAGE_START
        - PAGE_END, LINE_START
        - LINE_END
        - FIRST_LINE_START
        - FIRST_LINE_END
        - LAST_LINE_END
        - LAST_LINE_START

Layout manager – Example of Using GridBagLayout

```
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.*;
public class GridBagLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("GridBagLayout Example");
        JPanel panel = new JPanel(new GridBagLayout());

        GridBagConstraints constraints = new GridBagConstraints();
        constraints.gridx = 0;
        constraints.gridy = 0;
        constraints.gridwidth = 2;
        constraints.fill = GridBagConstraints.HORIZONTAL;

        JButton button1 = new JButton("Button 1");
        panel.add(button1, constraints);

        constraints.gridx = 2;
        constraints.gridy = 0;
        constraints.gridwidth = 1;
        constraints.fill = GridBagConstraints.NONE;
```

```
 JButton button2 = new JButton("Button 2");
        panel.add(button2, constraints);

        constraints.gridx = 0;
        constraints.gridy = 1;
        constraints.gridwidth = 3;
        constraints.fill = GridBagConstraints.BOTH;
        constraints.weightx = 1.0;
        constraints.weighty = 1.0;

        JButton button3 = new JButton("Button 3");
        panel.add(button3, constraints);

        frame.add(panel);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Arrange components on a window

- In this example, we create a JFrame and a JPanel with a GridBagLayout.
- We then create three JButton components and add them to the panel using GridBagConstraints
- The first button spans two columns, the second button is in the third column, and the third button spans all three columns and both rows
- The fill and weightx/weighty properties are also used to specify how the components should be sized and aligned

Arrange components on a window

- Steps for an effective arrangement
  - Groups the related components into the same panel
  - Use a layout manager to arrange the components on a panel
  - Specify a layout manager for the window in order to arrange the panels on the window

JPanel Class

- Is a container that can hold other components, such as buttons, labels, text fields, and other JPanels
- Is a subclass of JComponent, which means that it inherits all of the methods and properties of JComponent
- Has a default layout manager, which is usually a **FlowLayout**, but it can be changed to other layout managers
- Can be used as a building block for creating complex GUIs by combining multiple JPanels within each other
- Can be customized with various properties like background color, foreground color, border, tooltip, etc.
- Can respond to user events like mouse clicks, key presses, etc., by using event listeners

JPanel Class - Example of grouping the component using **Panel** class

```
public class MyPanel extends JPanel {
    // constructor
    public MyPanel() {
        // set the size of the panel
        setPreferredSize(new Dimension(400, 300));

        // add some components to the panel
        JButton button = new JButton("Click me!");
        add(button);
        JLabel label = new JLabel("Hello, World!");
        add(label);
    }

    // main method to test the panel
    public static void main(String[] args) {
        JFrame frame = new JFrame("MyPanel");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(new MyPanel());
        frame.pack();
        frame.setVisible(true);
    }
```

- In this example, MyPanel is a subclass of JPanel that adds a button and a label to the panel
- The panel is then added to a JFrame

Events Handling

- Event
  - Is produced by user action on a graphic component
  - Is a mechanism by which the user communicates with the program
- Example of events
  - Press a button
  - Close a window, etc.
- In Java, every object can consume events generated by a certain graphic component
- Source of events
  - Is the component generating the event
  - Notify the event listener when an event occurs

Events Handling

- Event listener
  - Is a listener class
  - Listen for events generated by program’s components
  - Is notify when an event occurs
  - It belongs to a class provided by the application programmer
  - Its methods describe the actions executed when on event occurs
  - Is a class that specify in its declaration that it wants to listen a certain type of event by:
    - Implementing a certain interface that is specific to each event
      - ex: **ActionListener** interface is implemented for **ActionEvent** events
      - ex: **TextListener** interface is implemented for **TextEvent** events

Events Handling

- A listener (i.e., class) can listen more types of events

class Listener **implements ActionListener**, **TextListener**

- All interfaces specific to each event extends **EventListener** interface

Events Handling

- Example of handling events
  - We consider a J**Button** component
  - We consider a class that implements the **ActionListener** interface
    - The class provides implementation for the **actionPerformed** method
      - **actionPerformed** method contains the code that is executed when we press the button
  - We declare a listener object that is attached to the button

```
JButton button= new JButton(“Press me”);
 ......
 ActionListener listener = new ClickListener();
 button.addActionListener(listener);

class ClickListener implements ActionListener{
            public void actionPerformed(ActionEvent  ev)
                 {  // code that is executed when we press the
button component }
  }
```

Events Handling

- Types of events:
  - Low level events
    - » Represent a low-level interaction
    - » e.g., key press, an operation on a window
  - Semantic events
    - » Represent the interaction with a GUI component
    - » e.g., press a button, select an item in a list,...

Events Handling – Example of Low-level events

ComponentEvent	These events are generated when a GUI component is added, removed, or resized. The low-level events generated by the component include component added, component removed, component shown, component hidden, component resized, and component moved.
ContainerEvent	Is an event that occurs when a container is added or removed from a parent container
FocusEvent	This event is generated when a component gains or loses focus, such as when the user clicks on a component or moves the focus from one component to another
KeyEvent	These events are generated when the user interacts with the keyboard, such as pressing or releasing a key. The low-level events generated by the keyboard include key pressed, key released, and key typed.
MouseEvent	These events are generated when the user interacts with the mouse, such as clicking, double-clicking, or dragging the mouse. The low-level events generated by the mouse include mouse pressed, mouse released, mouse clicked, mouse entered, and mouse exited
WindowEvent	These events are generated when a window is opened, closed, activated, or deactivated. The low-level events generated by the window include window opened, window closing, window closed, window activated, and window deactivated.

Events Handling – Example of Semantic events

ActionEvent	This event is generated when the user performs an action that requires a response from the application, such as clicking a button, selecting a menu item, or pressing Enter on the keyboard.
AdjustmentEvent	This event is generated when a scroll bar, slider, or other adjustable component is changed. This event is typically used to monitor changes in a user interface component's value, such as the position of a scroll bar.
ItemEvent	This event is generated when the state of an item in a choice or list component changes, such as when a checkbox is checked or unchecked or when an item is selected in a combo box.
TextEvent	This event is generated by text components when the text is modified. It indicates that a change has occurred in the text and provides information about the type of change.

Events Handling –Example of types of events generated by Swing components

Swing Component	**Generated events**
JButton	ActionEvent, FocusEvent, MouseEvent, KeyEvent, ComponentEvent
JTexArea	FocusEvent, MouseEvent, KeyEvent
JList	ListSelectionEvent, MouseEvent, KeyEvent, ComponentEvent, focusEvent
JRadioButton	ActionEvent, ItemEvent
JFrame	WindowEvent, ComponentEvent, FocursEvent, MouseEvent, KeyEvent
JCheckBox	ItemEvent, ActionEvent

Events Handling

- **getSource**() method
  - Is inherited by all types of events
  - Returns the object that is responsible for generating the event
  - Is used when a **listener** object handle events of the same type generated by different components
    - »Is necessary can find out the source of event we treat

Events Handling - Example of using **getSource**() method

```
public class Fereastra implements
ActionListener{
 JFrame fereastra;
 JPanel panou;
 JButton ok;
 JTextField nume;
 JLabel lb;
 Fereastra()
     {
         initComponent();
     }
  public void initComponent(){
    fereastra= new JFrame("First Application");
    fereastra.setSize(300,200);
    panou = new JPanel();
    ok = new JButton("Buton");
    nume= new JTextField(20);
    lb = new JLabel();
```

```
 panou.add(nume);
   panou.add(ok);
   panou.add(lb);
   fereastra.add(panou);
   nume.addActionListener(this);
   ok.addActionListener(this);
   fereastra.setVisible(true);
}
public void actionPerformed(ActionEvent e) {
     if (e.getSource() == ok)
       { lb.setText(" a fost apasat butonul"); }
     if(e.getSource() == nume)
        {lb.setText("a fost editata componenta texfield");}}
public static void main(String[] args) {
        new Fereastra();
  }
}
```

Events Handling

- **getActionCommand**() method
  - Return the label of the pressed component

```
 class Exemplu implements ActionListener{
   Button bt=  new Button(“button”);
    ……
   public void actionPerformed(ActionEvent e) {
      if (e.getActionCommand() ==“button”) {
       // A fost apasat butonul ’ok’
      }
...}
```

Events Handling - Example of using **getActionCommand**() method

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
public class Frame implements ActionListener{
  private JButton b1, b2,b3;
  private JFrame f;
 Frame()
   {  initComponent(); }
```

```
 public void initComponent(){
    f= new JFrame("First Application");
    f.setLayout(new FlowLayout());
    f.setSize(300,200);
    b1= new JButton("Bt1");
    b2= new JButton("Bt2");
    b3= new JButton("Bt3");
    b1.addActionListener(this);
    b2.addActionListener(this);
    b3.addActionListener(this);
    f.add(b1);
    f.add(b2);
    f.add(b3);
    f.setVisible(true);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

Events Handling

```
@Override
   public void actionPerformed(ActionEvent e) {
           // TODO Auto-generated method stub
           if(e.getActionCommand()=="Bt1")
                      JOptionPane.showMessageDialog(f, "button1 pressed");
           else
                      if(e.getActionCommand()=="Bt2")
                                 JOptionPane.showMessageDialog(f, "button2 pressed");
                      else
                                 JOptionPane.showMessageDialog(f, "button3 pressed");
                      }
           public static void main(String args[]) {
                      new Frame();
           }
}
```