Unit IV

1. Types of I/O Streams in Java

Java provides several types of I/O streams that can be used to read and write data. These include:

- Byte Streams: These streams are used to read and write raw bytes. They are commonly used for reading and writing binary data, such as images and audio files. Examples of byte streams include FileInputStream and FileOutputStream.
- Character Streams: These streams are used to read and write characters. They are commonly used for reading and writing text data, such as files and input from the keyboard. Examples of character streams include FileReader and FileWriter.
- Buffered Streams: These streams are used to improve the performance of I/O operations by buffering data in memory. Examples of buffered streams include BufferedInputStream and BufferedOutputStream.
- Object Streams: These streams are used to read and write objects. They are commonly used for serializing and deserializing objects. Examples of object streams include ObjectOutputStream and ObjectInputStream.

---

1. Serialization and Deserialization in Java

Serialization is the process of converting an object into a byte stream, while deserialization is the process of converting a byte stream into an object. In Java, serialization and deserialization can be achieved using the ObjectOutputStream and ObjectInputStream classes.

Serialization is useful for storing and retrieving objects, sending objects over a network, and caching objects. Deserialization is useful for reconstructing objects from a byte stream.

Here is an example of how to serialize and deserialize an object in Java:

```
import (link unavailable)*;

class Person implements Serializable {
   private String name;
   private int age;

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

```
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

public class SerializationExample {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        Person person = new Person("John Doe", 30);

        // Serialize the person object
        FileOutputStream fileOutputStream = new FileOutputStream("person.ser");
        ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);
        objectOutputStream.writeObject(person);
        objectOutputStream.close();

        // Deserialize the person object
        FileInputStream fileInputStream = new FileInputStream("person.ser");
        ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);
        Person deserializedPerson = (Person) objectInputStream.readObject();
        objectInputStream.close();

        System.out.println("Name: " + deserializedPerson.getName());
        System.out.println("Age: " + deserializedPerson.getAge());
    }
}
```

---

1. Wrapper Classes in Java

Wrapper classes are classes that provide a way to use primitive types as objects. In Java, wrapper classes are provided for each primitive type, including Integer, Double, Boolean, and Character.

Wrapper classes provide several benefits, including:

- They allow primitive types to be used as objects, which is necessary for many Java features, such as collections and generics.
- They provide methods for converting between primitive types and objects.
- They provide methods for performing common operations, such as parsing and formatting.

Here is an example of how to use the Integer wrapper class:

```
public class WrapperClassExample {
   public static void main(String[] args) {
      int primitiveInt = 10;
      Integer wrapperInt = Integer.valueOf(primitiveInt);

      System.out.println("Primitive int: " + primitiveInt);
      System.out.println("Wrapper int: " + wrapperInt);

      // Convert wrapper int to primitive int
      int convertedPrimitiveInt = wrapperInt.intValue();
      System.out.println("Converted primitive int: " + convertedPrimitiveInt);
   }
}
```

---

1. Scanner and StringTokenizer Classes in Java

The Scanner and StringTokenizer classes are used for parsing input in Java.

The Scanner class is used for parsing input from a variety of sources, including files, input streams, and strings. It provides methods for scanning input and parsing it into different types, such as integers, doubles, and strings.

The StringTokenizer class is used for breaking a string into tokens. It provides methods for tokenizing a string and iterating over the tokens.

Here is an example of how to use the Scanner class:

```
import java.util.Scanner;

public class ScannerExample {
   public static void main(String[] args) {
```

```
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter your name: ");
        String name = scanner.nextLine();

        System.out.print("Enter your age: ");
        int age = scanner.nextInt();

        System.out.println("Name: " + name);
        System.out.println("Age: " + age);

        scanner.close();
    }
}
```

---

1. Date, Calendar, and Timer Classes in Java

The Date, Calendar, and Timer classes are used for working with dates and times in Java.

The Date class represents a specific date and time. It provides methods for getting and setting the date and time, as well as for comparing dates.

The Calendar class provides methods for manipulating dates and times. It allows you to set and get the year, month, day, hour, minute, and second of a date.

The Timer class is used for scheduling tasks to run at a later time. It provides methods for scheduling tasks to run once or repeatedly.

Here is an example of how to use the Date and Calendar classes:

```
import java.util.Date;
import java.util.Calendar;

public class DateCalendarExample {
    public static void main(String[] args) {
        // Create a Date object
        Date date = new Date();
        System.out.println("Current date: " + date);

        // Create a Calendar object
```

```
        Calendar calendar = Calendar.getInstance();
        System.out.println("Current year: " + calendar.get(Calendar.YEAR));
        System.out.println("Current month: " + calendar.get(Calendar.MONTH));
        System.out.println("Current day: " + calendar.get(Calendar.DAY_OF_MONTH));

        // Set the year, month, and day of the calendar
        calendar.set(Calendar.YEAR, 2022);
        calendar.set(Calendar.MONTH, 11);
        calendar.set(Calendar.DAY_OF_MONTH, 25);

        System.out.println("New date: " + calendar.getTime());
    }
}
```

---

1. Regular Expressions in Java

Regular expressions are a way to describe a pattern in a string. In Java, regular expressions can be used with the Pattern and Matcher classes.

The Pattern class represents a regular expression. It provides methods for compiling a regular expression and for matching a regular expression against a string.

The Matcher class represents a matcher for a regular expression. It provides methods for finding matches for a regular expression in a string.

Here is an example of how to use regular expressions in Java:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegularExpressionExample {
    public static void main(String[] args) {
        // Create a regular expression pattern
        Pattern pattern = Pattern.compile("hello");

        // Create a matcher for the pattern
        Matcher matcher = pattern.matcher("hello world");

        // Find matches for the pattern
```

```
      while (matcher.find()) {
         System.out.println("Match found: " + matcher.group());
      }
   }
}
```

Unit V

1. Abstract Window Toolkit (AWT) Class Hierarchy in Java

The Abstract Window Toolkit (AWT) is a set of classes in Java that provide a way to create graphical user interfaces (GUIs). The AWT class hierarchy includes the following classes:

- Component: This is the top-level class in the AWT class hierarchy. It represents a GUI component, such as a button or a text field.
- Container: This class extends the Component class and represents a container for other GUI components.
- Panel: This class extends the Container class and represents a panel for holding other GUI components.
- Frame: This class extends the Panel class and represents a top-level window for a GUI application.
- Dialog: This class extends the Dialog class and represents a dialog box for a GUI application.

Here is an example of how to create a GUI application using the AWT class hierarchy:

```
import java.awt.Frame;
import java.awt.Label;
import java.awt.Button;

public class AWTExample {
   public static void main(String[] args) {
      // Create a frame
      Frame frame = new Frame("AWT Example");

      // Create a label
      Label label = new Label("Hello, World!");

      // Create a button
      Button button = new Button("Click me!");

      // Add the label and button to the frame
```

```
        frame.add(label, "North");
        frame.add(button, "South");

        // Set the size of the frame
        frame.setSize(300, 200);

        // Make the frame visible
        frame.setVisible(true);
    }
}
```

---

1. Comparison of AWT and Swing Libraries in Java

The AWT and Swing libraries are two sets of classes in Java that provide a way to create graphical user interfaces (GUIs). Here are some key similarities and differences between the two libraries:

Similarities:

- Both AWT and Swing provide a way to create GUI components, such as buttons, labels, and text fields.
- Both libraries provide a way to handle events, such as button clicks and keyboard input.
- Both libraries provide a way to customize the look and feel of GUI components.

Differences:

- AWT is older and less powerful than Swing. AWT was introduced in Java 1.0, while Swing was introduced in Java 1.2.
- AWT components are heavyweight, meaning that they are implemented using native code and are therefore less flexible and less customizable. Swing components, on the other hand, are lightweight, meaning that they are implemented using Java code and are therefore more flexible and more customizable.
- AWT does not provide a way to use a pluggable look and feel, while Swing does. This means that Swing GUIs can be customized to look and feel like native GUIs on different platforms.

Here is an example of how to create a GUI application using Swing:

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JButton;

public class SwingExample {
    public static void main(String[] args) {
```

```
        // Create a frame
        JFrame frame = new JFrame("Swing Example");

        // Create a label
        JLabel label = new JLabel("Hello, World!");

        // Create a button
        JButton button = new JButton("Click me!");

        // Add the label and button to the frame
        frame.getContentPane().add(label, "North");
        frame.getContentPane().add(button, "South");

        // Set the size of the frame
        frame.setSize(300, 200);

        // Make the frame visible
        frame.setVisible(true);
    }
}
```

---

1. Layout Managers in Java

Layout managers are used to arrange GUI components in a container. Java provides several layout managers, including:

- FlowLayout: This layout manager arranges components in a horizontal row.
- BorderLayout: This layout manager divides a container into five regions: north, south, east, west, and center.
- GridLayout: This layout manager arranges components in a grid of rows and columns.
- GridBagLayout: This layout manager is a more complex and flexible layout manager that allows you to specify the size and position of components in a grid.
- CardLayout: This layout manager is used to implement a deck of cards, where only one card is visible at a time.

Here is an example of how to use the BorderLayout:

```
import java.awt.BorderLayout;
import java.awt.Button;
```

```
import java.awt.Frame;
import java.awt.Label;
import java.awt.TextField;

public class BorderLayoutExample {
   public static void main(String[] args) {
      // Create a frame
      Frame frame = new Frame("BorderLayout Example");

      // Create a label
      Label label = new Label("Hello, World!");

      // Create a text field
      TextField textField = new TextField();

      // Create a button
      Button button = new Button("Click me!");

      // Create a border layout
      frame.setLayout(new BorderLayout());

      // Add the label to the north region
      frame.add(label, BorderLayout.NORTH);

      // Add the text field to the center region
      frame.add(textField, BorderLayout.CENTER);

      // Add the button to the south region
      frame.add(button, BorderLayout.SOUTH);

      // Set the size of the frame
      frame.setSize(300, 200);

      // Make the frame visible
      frame.setVisible(true);
   }
}
```

---

1. Event Handling in Java

Event handling is the process of responding to events that occur in a GUI application. Events can include things like button clicks, keyboard input, and mouse movements.

Java provides several ways to handle events, including:

- Using event listener interfaces: These interfaces define methods that are called when an event occurs.
- Using event adapter classes: These classes provide default implementations for event listener interfaces.
- Using anonymous inner classes: These classes can be used to implement event listener interfaces.

Here is an example of how to use an event listener interface to handle a button click:

```
import java.awt.Button;
import java.awt.Frame;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class EventHandlerExample {
   public static void main(String[] args) {
      // Create a frame
      Frame frame = new Frame("EventHandler Example");

      // Create a button
      Button button = new Button("Click me!");

      // Add an action listener to the button
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            System.out.println("Button clicked!");
         }
      });

      // Add the button to the frame
      frame.add(button);

      // Set the size of the frame
      frame.setSize(300, 200);

      // Make the frame visible
      frame.setVisible(true);
   }
}
```

1. Handling Mouse and Keyboard Events in Java

Mouse and keyboard events are handled in Java using event listener interfaces. The MouseListener interface is used to handle mouse events, while the KeyListener interface is used to handle keyboard events.

Here is an example of how to use the MouseListener interface to handle mouse events:

```
import java.awt.Frame;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class MouseEventHandlerExample {
   public static void main(String[] args) {
      // Create a frame
      Frame frame = new Frame("MouseEventHandler Example");

      // Add a mouse listener to the frame
      frame.addMouseListener(new MouseAdapter() {
         public void mouseClicked(MouseEvent e) {
            System.out.println("Mouse clicked!");
         }

         public void mousePressed(MouseEvent e) {
            System.out.println("Mouse pressed!");
         }

         public void mouseReleased(MouseEvent e) {
            System.out.println("Mouse released!");
         }
      });

      // Set the size of the frame
      frame.setSize(300, 200);

      // Make the frame visible
      frame.setVisible(true);
   }
```

}

---

1. JDBC in Java

JDBC (Java Database Connectivity) is a set of classes and interfaces in Java that provide a way to interact with databases. JDBC provides a way to connect to a database, execute SQL statements, and retrieve results.

Here is an example of how to use JDBC to connect to a database and execute a SQL statement:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JdbcExample {
   public static void main(String[] args) {
      // Load the JDBC driver
      try {
         Class.forName("com.mysql.jdbc.Driver");
      } catch (ClassNotFoundException e) {
         System.out.println("Error loading JDBC driver: " + e.getMessage());
         return;
      }

      // Connect to the database
      Connection connection = null;
      try {
         connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase",
"myusername", "mypassword");
      } catch (SQLException e) {
         System.out.println("Error connecting to database: " + e.getMessage());
         return;
      }

      // Execute a SQL statement
      Statement statement = null;
      try {
```

```
            statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery("SELECT * FROM mytable");
            while (resultSet.next()) {
                System.out.println("Name: " + resultSet.getString("name"));
                System.out.println("Age: " + resultSet.getInt("age"));
            }
        } catch (SQLException e) {
            System.out.println("Error executing SQL statement: " + e.getMessage());
        } finally {
            // Close the statement and connection
            try {
                if (statement != null) {
                    statement.close();
                }
                if (connection != null) {
                    connection.close();
                }
            } catch (SQLException e) {
                System.out.println("Error closing statement and connection: " + e.getMessage());
            }
        }
    }
}
```