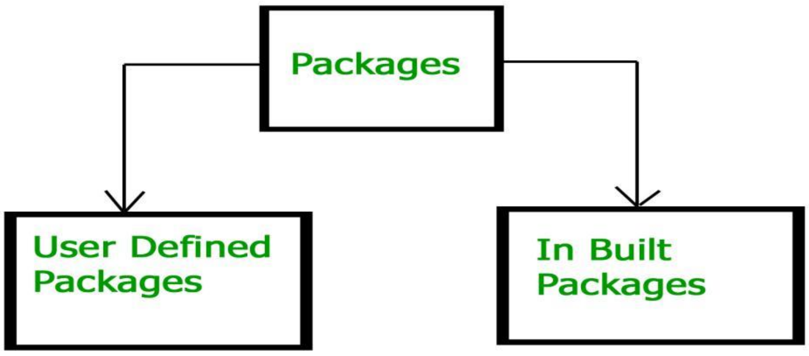
Packages

A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code.

Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)



Built-in Packages

These packages consist of a large number of classes which are a part of Java **API**.Some of the commonly used built-in packages are:

1. **java.lang:** Contains language support classes(e.g classes which defines primitive data types, math operations). This package is automatically imported.
2. java.io: Contains classes for supporting input / output operations.
3. java.util: Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
4. **java.applet:** Contains classes for creating Applets.
5. **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc). 6)
6. java.net: Contain classes for supporting networking operations.

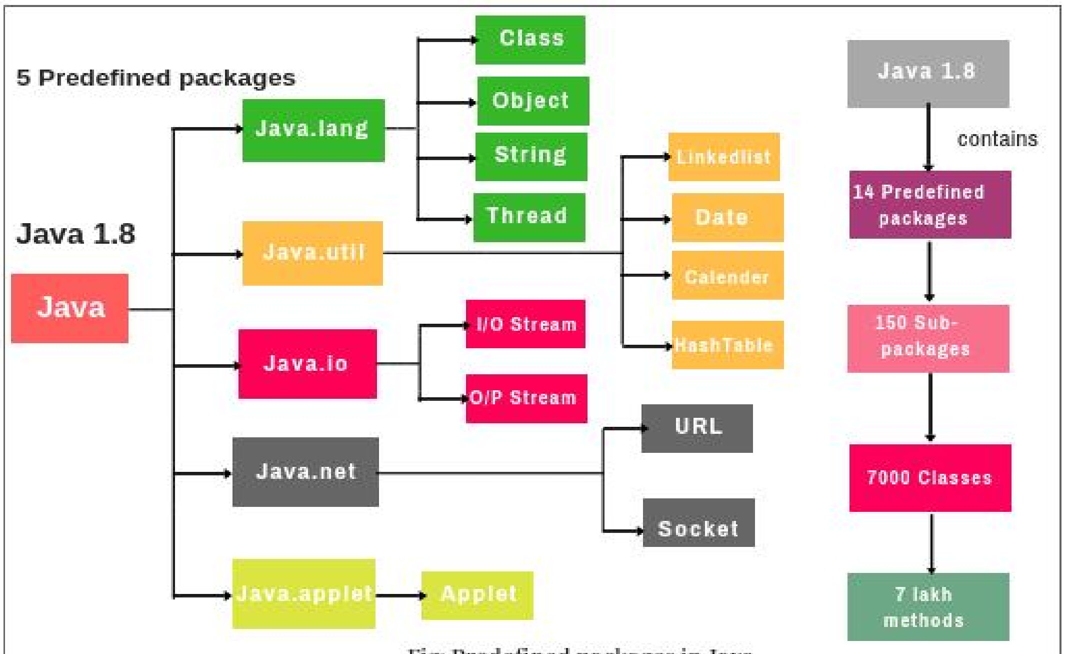


Fig: Predefined packages in Java

Java program of demonstrating use of java.lang package

```
class JavaLangExample {
   public static void main(String args []) {
      int a = 20, b =30;
      int sum = Math.addExact(a,b);
      int max = Math.max(a,b);
      double pi = Math.PI;
      System.out.printf("Sum = "+sum+", Max = "+max+", PI = "+pi);
    }
 }
```

Output:

```
Sum = 50, Max = 30, PI = 3.141592653589793
```

Java program of demonstrating use of java.io package

```
import java.io.Console;
 class JavaIOExample {
   public static void main(String args []) {
      Console cs = System.console();
      System.out.println("Enter your name : ");
      String name = cs.readLine();
      System.out.println("Welcome : "+name);
    }
 }
```

Output:

```
Enter your name : teja
Welcome : teja
```

Java program of demonstrating use of java.util package

```
import java.util.Arrays;
```

```
class JavaUtilExample {
  public static void main(String args []) {
    int[] intArray = {10,30,20,50,40};
    Arrays.sort(intArray);
    System.out.printf("Sorted array : %s", Arrays.toString(intArray));
  }
}
```
Output:
```
Sorted array : [10, 20, 30, 40, 50]
```

User-defined packages

User-defined packages are those packages that are designed or created by the developer to categorize classes and packages. They are much similar to the built-in that java offers. It can be imported into other classes and used the same as we use built-in packages. But If we omit the package statement, the class names are put into the default package, which has no name.

```
//write a java program using user defined packages.
import java.io.*;
class Greeting {
   public void sayHello() {
      System.out.println("Hello from mypackage!");
   }
}
class Farewell {
   public void sayGoodbye() {
      System.out.println("Goodbye from mypackage!");
   }
}
class ThankYou {
   public void sayThanks() {
      System.out.println("Thank you from mypackage!");
   }
}
public class Main {
   public static void main(String[] args) {
      Greeting greet = new Greeting();
      greet.sayHello();

      Farewell farewell = new Farewell();
      farewell.sayGoodbye();

      ThankYou thankYou = new ThankYou();
      thankYou.sayThanks();
   }
}
```
Output:

```
Hello from mypackage!
Goodbye from mypackage!
Thank you from mypackage!
```

Java File Path

java.io.File contains three methods for determining the file path, we will explore them in this tutorial.

1. getPath(): This file path method returns the abstract pathname as String. If String pathname is used to create File object, it simply returns the pathname argument. If URI is used as argument then it removes the protocol and returns the file name.
2. getAbsolutePath(): This file path method returns the absolute path of the file. If File is created with absolute pathname, it simply returns the pathname. If the file object is created using a relative path, the absolute pathname is resolved in a system-dependent way. On UNIX systems, a relative pathname is made absolute by resolving it against the current user directory. On Microsoft Windows systems, a relative pathname is made absolute by resolving it against the current directory of the drive named by the pathname, if any; if not, it is resolved against the current user directory.
3. [getCanonicalPath](https://docs.oracle.com/javase/7/docs/api/java/io/File.html#getCanonicalPath())(): This path method returns the canonical pathname that is both absolute and unique. This method first converts this pathname to absolute form if necessary, as if by invoking the getAbsolutePath method, and then maps it to its unique form in a system-dependent way. This typically involves removing redundant names such as "." and "..." from the pathname, resolving symbolic links (on UNIX platforms), and converting drive letters to a standard case (on Microsoft Windows platforms).

Java File Path Example

Let's see different cases of the file path in java with a simple program.

```
package com.journaldev.files;
import java.io.File;
import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
public class JavaFilePath {
    public static void main(String[] args) {
        try {
            // Absolute path
            File file = new File("/Users/pankaj/test.txt");
            printPaths(file);
            // Relative path
            file = new File("test.xsd");
            printPaths(file);
            // Complex relative path
            file = new File("/Users/pankaj/../pankaj/test.txt");
            printPaths(file);
            // URI path
            file = new File(new URI("file:///Users/pankaj/test.txt"));
            printPaths(file);
        } catch (IOException | URISyntaxException e) {
            e.printStackTrace();
        }
    }
    private static void printPaths(File file) throws IOException {
        System.out.println("Path: " + file.getPath());
        System.out.println("Absolute Path: " + file.getAbsolutePath());
        System.out.println("Canonical Path: " + file.getCanonicalPath());
        System.out.println("---------------------------");
    }
}
```

Output:

```
Path: /Users/pankaj/test.txt
Absolute Path: /Users/pankaj/test.txt
Canonical Path: /Users/pankaj/test.txt
---------------------------
Path: test.xsd
Absolute Path: /Users/pankaj/test.xsd
Canonical Path: /Users/pankaj/test.xsd
---------------------------
Path: /Users/pankaj/../pankaj/test.txt
Absolute Path: /Users/pankaj/../pankaj/test.txt
Canonical Path: /Users/pankaj/test.txt
---------------------------
Path: /Users/pankaj/test.txt
Absolute Path: /Users/pankaj/test.txt
Canonical Path: /Users/pankaj/test.txt
---------------------------
```

THREADS

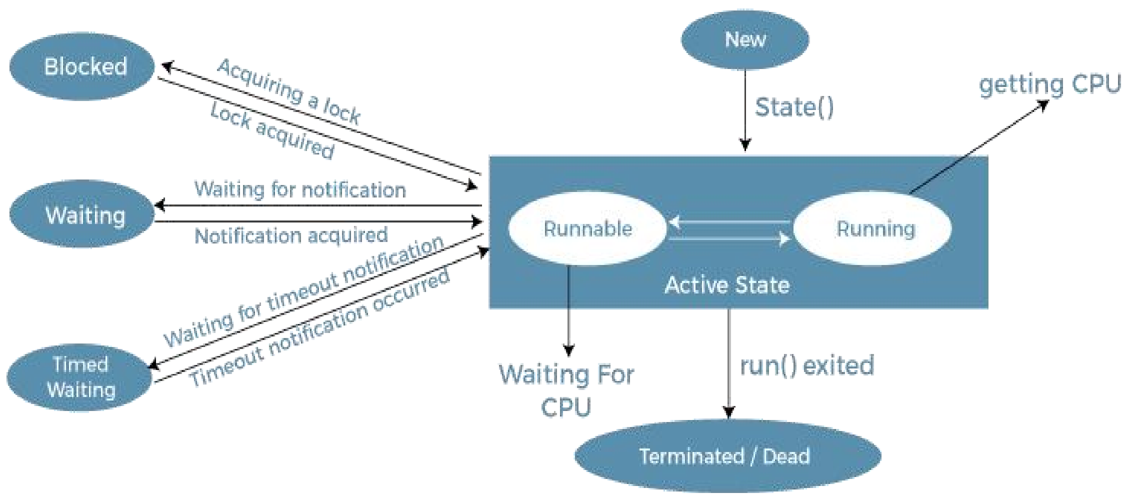
- A thread in Java is the direction or path that is taken while a program is being executed. Generally, all the programs have at least one thread, known as the main thread, that is provided by the JVM or Java Virtual Machine at the starting of the program's execution. At this point, when the main thread is provided, the main() method is invoked by the main thread.

- **A thread is an execution thread in a program**. Multiple threads of execution can be run concurrently by an thread varies. Higher priority threads are executed before lower priority threads.

- Thread is critical in the program because it enables multiple operations to take place within a single method. Each thread in the program often has its own program counter, stack, and local variable.

- Thread in Java enables concurrent execution, dividing tasks for improved performance. It's essential for handling operations like I/O and network communication efficiently. Understanding threads is crucial for responsive Java applications. Enroll in a Java Course to master threading and create efficient multithreaded programs.

LIFE CYCLE OF THREAD:



Life Cycle of a Thread

Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:

1. New

2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

Explanation of Different Thread States

New: Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

Active: When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.

- **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.
  A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.
- **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

Blocked or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

For example, a thread (let's say its name is A) may want to print some data from the printer. However, at the same time, the other thread (let's say its name is B) is using the printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus, thread A is in the blocked state. A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU). Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state.

When the main thread invokes the join() method then, it is said that the main thread is in the waiting state. The main thread then waits for the child threads to complete their tasks. When the child threads complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.

If there are a lot of threads in the waiting or blocked state, then it is the duty of the thread scheduler to determine which thread to choose and which one to reject, and the chosen thread is then given the opportunity to run.

Timed Waiting: Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

Terminated: A thread reaches the termination state because of the following reasons:

- When a thread has finished its job, then it exists or terminates normally.
- **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

Implementation of Thread States

In Java, one can get the current state of a thread using the **Thread.getState()** method. The **java.lang.Thread.State** class of Java provides the constants ENUM to represent the state of a thread. These constants are:

- public static final Thread.State NEW

It represents the first state of a thread that is the NEW state.

- public static final Thread.State RUNNABLE

It represents the runnable state.It means a thread is waiting in the queue to run.

- **public static final** Thread.State BLOCKED

  It represents the blocked state. In this state, the thread is waiting to acquire a lock.

- **public static final** Thread.State WAITING

  It represents the waiting state. A thread will go to this state when it invokes the Object.wait() method, or Thread.join() method with no timeout. A thread in the waiting state is waiting for another thread to complete its task.

- **public static final** Thread.State TIMED_WAITING

  It represents the timed waiting state. The main difference between waiting and timed waiting is the time constraint. Waiting has no time constraint, whereas timed waiting has the time constraint. A thread invoking the following method reaches the timed waiting state.

  - sleep
  - join with timeout
  - wait with timeout
  - parkUntil
  - parkNanos

- **public static final** Thread.State TERMINATED

  It represents the final state of a thread that is terminated or dead. A terminated thread means it has completed its execution.

Java Program for Demonstrating Thread States

FileName: ThreadState.java

```
class ABC implements Runnable
{
public void run()
{
try
{
    Thread.sleep(100);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}
  System.out.println("The state of thread t1 while it invoked the me
thod join() on thread t2 -   "+ ThreadState.t1.getState());
try
{
Thread.sleep(200);
}
```

```
.   catch (InterruptedException ie)
.   {
.   ie.printStackTrace();
.   }
.   }
.   }
.   public class ThreadState implements Runnable
.   {
.   public static Thread t1;
.   public static ThreadState obj;
.   public static void main(String argvs[])
.   {
.   obj = new ThreadState();
.   t1 = new Thread(obj);
.   System.out.println("The state of thread t1 after spawning it -
    " + t1.getState());
.   t1.start();
.   System.out.println("The state of thread t1 after invoking the
.   method start() on it - " + t1.getState());
.   }
.   public void run()
.   {
.   ABC myObj = new ABC();
.   Thread t2 = new Thread(myObj);
.   System.out.println("The state of thread t2 after spawning it -
    "+ t2.getState());
.   t2.start();
.   System.out.println("the state of thread t2 after calling the method
    start() on it - " + t2.getState());
.   try
.   {
.   Thread.sleep(200);
.   }
.   catch (InterruptedException ie)
.   {
.   ie.printStackTrace();
.   }
.   System.out.println("The state of thread t2 after invoking the
.   method sleep() on it - "+ t2.getState() );
.   try
.   {
.   t2.join();
.   }
.   catch (InterruptedException ie)
.   {
.   ie.printStackTrace();
.           }
.           System.out.println("The state of thread t2 when it has completed
.           It's execution - " + t2.getState());
.           }
.           }
```
Output:
```
The state of thread t1 after spawning it - NEW
The state of thread t1 after invoking the method start() on it -
RUNNABLE
The state of thread t2 after spawning it - NEW
the state of thread t2 after calling the method start() on it - RUNNABLE
The state of thread t1 while it invoked the method join() on thread t2 -
TIMED_WAITING
The state of thread t2 after invoking the method sleep() on it -
TIMED_WAITING
The state of thread t2 when it has completed it's execution -
TERMINATED
```

Priority of a Thread in Java

- ❖ Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled. You can get and set the priority of a Thread. Thread class provides methods and constants for working with the priorities of a Thread.
- ❖ Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Built-in Property Constants of Thread Class

- ❖ Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).
- ❖ **MIN_PRIORITY**: Specifies the minimum priority that a thread can have.
- ❖ **NORM_PRIORITY**: Specifies the default priority that a thread is assigned.
- ❖ **MAX_PRIORITY**: Specifies the maximum priority that a thread can have.

Types of synchronization:

There are two types of synchronization that are as follows:

1. Process synchronization
2. Thread synchronization

Here we will be mainly focusing on thread synchronization.

Thread synchronization basically refers to The concept of one thread execute at a time and the rest of the threads are in waiting state. This process is known as thread synchronization. It prevents the thread interference and inconsistency problem.

Synchronization is build using locks or monitor. In Java, a monitor is an object that is used as a mutually exclusive lock. Only a single thread at a time has the right to own a monitor. When a thread gets a lock then all other threads will get suspended which are trying to acquire the locked monitor. So, other threads are said to be waiting for the monitor, until the first thread exits the monitor. In a simple way, when a thread request a resource then that resource gets locked so that no other thread can work or do any modification until the resource gets released.

Thread Synchronization are of two types:

1. **Mutual Exclusive**
2. Inter-Thread Communication

A. Mutual Exclusive

While sharing any resource, this will keep the thread interfering with one another i.e. mutual exclusive. We can achieve this via

- Synchronized Method
- Synchronized Block
- Static Synchronization

Synchronized Method

We can declare a method as synchronized using the *"synchronized"* keyword. This will make the code written inside the method thread-safe so that no other thread will execute while the resource is shared.

Thread Priority Setter and Getter Methods

❖ **Thread.getPriority() Method: This method is used to get the priority of a thread.**

❖ **Thread.setPriority() Method**: This method is used to set the priority of a thread, it accepts the priority value and updates an existing priority with the given priority.

PROGRAM:

```
package com.tutorialspoint;
class ThreadDemo extends Thread {
  ThreadDemo() { }
  public void run() {
    System.out.println("Thread Name: " +
        Thread.currentThread().getName()
      + ", Thread Priority: " + Thread.currentThread().getPriority());
```

```
         for(int i = 4; i > 0; i--) {
         System.out.println("Thread: " + Thread.currentThread().getName()
           + ", Countdown: " + i);
         try {
                    Thread.sleep(50);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }
     public void start() {
      super.start();
   }
}
public class TestThread {
   public static void main(String args[]) {
          ThreadDemo thread1 = new ThreadDemo();
      ThreadDemo thread2 = new ThreadDemo();
      ThreadDemo thread3 = new ThreadDemo();
        thread1.start();
      thread2.start();
      thread3.start();
   }
}
```

OUTPUT:

```
Thread Name: Thread-0, Thread Priority: 5
Thread: Thread-0, Countdown: 4
Thread Name: Thread-1, Thread Priority: 5
Thread: Thread-1, Countdown: 4
Thread Name: Thread-2, Thread Priority: 5
Thread: Thread-2, Countdown: 4
Thread: Thread-0, Countdown: 3
Thread: Thread-1, Countdown: 3
Thread: Thread-2, Countdown: 3
Thread: Thread-0, Countdown: 2
Thread: Thread-1, Countdown: 2
Thread: Thread-2, Countdown: 2
Thread: Thread-0, Countdown: 1
Thread: Thread-1, Countdown: 1
Thread: Thread-2, Countdown: 1
```

GUI(Graphic User Interface)

GUI stands for Graphical User Interface. It refers to an interface that allows one to interact with electronic devices like computers and tablets through graphic elements. It uses icons, menus, and other graphical representations to display information, as opposed to text-based commands. The graphic elements enable users to give commands to the computer and select functions by using the mouse or other input devices.

Program:

```
import java.awt.*;
import java.awt.event.*;
// Calculator class that extends WindowAdapter and implements
ActionListener for handling events
class MyCalc extends WindowAdapter implements ActionListener {
   Frame f;
   Label l1;
   Button b1, b2, b3, b4, b5, b6, b7, b8, b9, b0;
   Button badd, bsub, bmult, bdiv, bmod, bcalc, bclr, bpts, bneg, bback;
   double num1, num2, result;
   int operation;
   MyCalc() {
      f = new Frame("MY CALCULATOR");
      // Label to display input and result
      l1 = new Label();
      l1.setBackground(Color.LIGHT_GRAY);
      l1.setBounds(50, 50, 260, 60);
      // Initialize number buttons with positions
        b1 = createButton("1", 50, 340);
         b2 = createButton("2", 120, 340);
         b3 = createButton("3", 190, 340);
        b4 = createButton("4", 50, 270);
         b5 = createButton("5", 120, 270);
         b6 = createButton("6", 190, 270);
         b7 = createButton("7", 50, 200);
         b8 = createButton("8", 120, 200);
         b9 = createButton("9", 190, 200);
         b0 = createButton("0", 120, 410);
      // Initialize special function buttons
      bneg = createButton("+/-", 50, 410);
      bpts = createButton(".", 190, 410);
      bback = createButton("back", 120, 130);
      // Initialize arithmetic operation buttons
      badd = createButton("+", 260, 340);
        bsub = createButton("-", 260, 270);
      bmult = createButton("*", 260, 200);
        bdiv = createButton("/", 260, 130);
      bmod = createButton("%", 190, 130);
        bcalc = createButton("=", 245, 410);
      bclr = createButton("CE", 50, 130);
      // Add components to the frame
      f.add(l1);
      f.setSize(360, 500);
      f.setLayout(null);
      f.setVisible(true);
      f.addWindowListener(this);  // To handle window close
      // Add action listeners for each button
      addListeners();
   }
   // Utility method to create a button and set bounds
   private Button createButton(String label, int x, int y) {
      Button button = new Button(label);
      button.setBounds(x, y, 50, 50);
      f.add(button);
      return button;
   }
   // Method to add action listeners for buttons
   private void addListeners() {
      b1.addActionListener(this); b2.addActionListener(this);
b3.addActionListener(this); b4.addActionListener(this);
      b5.addActionListener(this); b6.addActionListener(this);
b7.addActionListener(this); b8.addActionListener(this);
      b9.addActionListener(this); b0.addActionListener(this);
      badd.addActionListener(this); bsub.addActionListener(this);
bmult.addActionListener(this); bdiv.addActionListener(this);
      bmod.addActionListener(this); bcalc.addActionListener(this);
bclr.addActionListener(this);
      bpts.addActionListener(this); bneg.addActionListener(this);
bback.addActionListener(this);
   }
   // Closing window
   public void windowClosing(WindowEvent e) {
      f.dispose();
   }
   // Method to handle button actions
   public void actionPerformed(ActionEvent e) {
      String label = ((Button)e.getSource()).getLabel();
      String currentText = l1.getText();
      // Handling number and decimal point buttons
      if ("0123456789.".contains(label)) {
          l1.setText(currentText + label);
      } else if (label.equals("CE")) {
          clear();
      } else if (label.equals("+/-")) {
          l1.setText(currentText.startsWith("-") ? currentText.substring(1) :
"-" + currentText);
      } else if (label.equals("back")) {
          if (currentText.length() > 0) {
              l1.setText(currentText.substring(0, currentText.length() - 1));
          }
      } else if ("+-*/%".contains(label)) {
          handleOperator(label);
      } else if (label.equals("=")) {
          calculateResult();
      }
   }
   // Method to handle arithmetic operations
   private void handleOperator(String operator) {
      try {
          num1 = Double.parseDouble(l1.getText());
      } catch (NumberFormatException e) {
          l1.setText("Invalid Format");
          return;
      }
      l1.setText("");
      switch (operator) {
          case "+" -> operation = 1;
          case "-" -> operation = 2;
          case "*" -> operation = 3;
          case "/" -> operation = 4;
          case "%" -> operation = 5;
      }
   }

   // Method to calculate and display the result
   private void calculateResult() {
      try {
          num2 = Double.parseDouble(l1.getText());
      } catch (Exception e) {
          l1.setText("Enter a number");
          return;
      }
      switch (operation) {
          case 1 -> result = num1 + num2;
          case 2 -> result = num1 - num2;
          case 3 -> result = num1 * num2;
          case 4 -> result = num1 / num2;
          case 5 -> result = num1 % num2;
      }
      l1.setText(String.valueOf(result));
   }
   // Method to clear the display and reset variables
   private void clear() {
      num1 = 0;
      num2 = 0;
      operation = 0;
      result = 0;
      l1.setText("");
   }
   // Main method to launch the calculator
   public static void main(String args[]) {
      new MyCalc();
   }
}
```

Output:

```
Calculator started.
Button pressed: 1
Display updated: 1
Button pressed: 2
Display updated: 12
Button pressed: 3
Display updated: 123
Button pressed: +
Display cleared, waiting for second number.
Button pressed: 4
Display updated: 4
Button pressed: 5
Display updated: 45
Button pressed: =
Calculation performed: 123 + 45 = 168
Display updated: 168

Button pressed: 6
Display updated: 6
Button pressed: *
Display cleared, waiting for second number.
Button pressed: 3
Display updated: 3
Button pressed: =
Calculation performed: 6 * 3 = 18
Display updated: 18

Button pressed: 7
Display updated: 7
Button pressed: -
```

```
Display cleared, waiting for second number.
Button pressed: 2
Display updated: 2
Button pressed: =
Calculation performed: 7 - 2 = 5
Display updated: 5

Button pressed: 5
Display updated: 5
Button pressed: +/-
Display updated: -5
Button pressed: CE
Display cleared.

Button pressed: 1
Display updated: 1
Button pressed: +
Display cleared, waiting for second number.
Button pressed: 0
Display updated: 0
Button pressed: /
Display cleared, waiting for second number.
Button pressed: 0
Display updated: 0
Button pressed: =
Error: Division by zero. Enter a number first.
Display updated: Enter a number first.
```