

UNIT 4

Multithreading

Thread-Definition and Features:

Thread is process. When the process needs to be divided into sub processes and executed under control of programmer, that process can be made as a thread. Generally the process management is done by operating system. When process is developed as thread, the thread control is managed by programmers. As the threads won't give burden on operating system, they are called light weight components.

A Thread can be created by extending Thread class or by implementing Runnable interface of java.lang package. He process to be placed as definition of the overridden run() method.

The threads can be either single tasking or multi tasking. That is, multiple threadsmake a single process done or multiple processes done. In single tasking, we override one run() method and in multi tasking, we override multiple run() methods of Runnable interface.

ThreadPriority: the Thread priority specifies how much time the thread is to occupy processor within given time. It never indicates sequence of execution. It can be specified between 1 to 10. 1 is the minimum priority and the 10 is the maximum priority. The maximum priority threads occupy the processor much time within given time. But irrespective of the priority, the execution of thread is based on the other resources using processor. So the threads are inaccurate.

Uses of Threads:

Threads allow a program to operate more efficiently by doing multiple things at the same time. Threads can be used to perform complicated tasks in the background without interrupting the main program.

Terminating the Thread :

A Thread can be started its execution by calling the method start() of Thread class and it can be terminated by making a call to stop() method of Thread class.

Q. explain what is thread and how to use it.

Q. explain how the threads can be created and executed.

Q. explain how the single tasking and multitasking threads are managed.

Creating Threads using Thread Class and Runnable Interface:

A thread is a lightweight unit of execution within a program. Java provides two common ways to create a thread:

By extending the Thread class and By implementing the Runnable interface

1. Creating a Thread by Extending Thread Class

In this method, a class is created by extending the predefined Thread class. The run() method is overridden to define the task that the thread should perform.

The thread is started using the start() method.

Example:

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Thread is running...");  
    }  
}
```

```
public class ThreadExample {  
    public static void main(String[] args) {  
        MyThread t1 = new MyThread();  
        t1.start();  
    }  
}
```

Here:

MyThread extends the Thread class.

run() contains the code executed by the thread.

start() creates a new thread and internally calls run().

2. Creating a Thread using Runnable Interface

In this method, a class implements the Runnable interface and provides the implementation of the run() method. An object of this class is then passed to a Thread object, which is started using start().

Example:

```
class MyRunnable implements Runnable {  
    public void run() {  
        System.out.println("Thread is running...");  
    }  
}
```

```
public class RunnableExample {  
    public static void main(String[] args) {  
        MyRunnable obj = new MyRunnable();  
        Thread t1 = new Thread(obj);  
        t1.start();  
    }  
}
```

Here:

MyRunnable implements Runnable.
The run() method defines the thread's task.
The Runnable object is passed to the Thread constructor.
start() starts the execution of the thread.

Difference between the Two Methods :

Thread Class	Runnable Interface
Class extends Thread	Class implements Runnable
Cannot extend another class	Can extend another class
Simpler for small programs	More flexible and preferred
Thread and task are combined	Thread and task are separated

Conclusion: Both methods can be used to create threads in Java. Implementing the Runnable interface is generally more flexible because Java supports single inheritance, allowing the class to extend another class while still implementing Runnable.

Q. Explain how to create threads.

Q. Explain creating threads using Thread class and Runnable interface.

Controlling with sleep() and Thread Synchronization with wait(), notify() :

Inter-thread communication, also known as **thread cooperation**, is a mechanism that allows synchronized threads to communicate and coordinate with each other. When multiple threads are working with a shared resource, one thread may have to wait until another thread completes a particular operation. In such situations, Java provides a mechanism through which one thread can temporarily stop its execution and allow another thread to perform the required operation. This helps threads work together efficiently and avoids unnecessary execution or continuous checking of a condition.

Java provides three methods for inter-thread communication: **wait()**, **notify()**, and **notifyAll()**. These methods are defined in the **java.lang.Object** class because every object in Java has an associated monitor or lock. These methods are generally used with synchronized methods or synchronized blocks to coordinate the execution of multiple threads.

sleep() Method

Before discussing inter-thread communication, it is important to understand the sleep() method. The **sleep() method** is used to temporarily suspend the execution of the currently running thread for a specified amount of time. It is a **static method of the Thread class**. The time is specified in milliseconds.

The syntax of the method is:

```
Thread.sleep(milliseconds);
```

For example:

```
class SleepExample {  
    public static void main(String[] args) {  
        System.out.println("Thread starts");  
  
        try {  
            Thread.sleep(2000);  
        } catch (InterruptedException e) {  
            System.out.println(e);  
        }  
  
        System.out.println("Thread resumes");  
    }  
}
```

In this example, the currently executing thread is suspended for **2000 milliseconds**, or 2 seconds. After the specified time has elapsed, the thread becomes eligible to continue its execution. The sleep() method does not release any lock or monitor held by the thread during the sleeping period. Since the method can be interrupted by another thread, it may throw InterruptedException.

wait() Method

The **wait() method** causes the current thread to release the lock it holds and wait until another thread invokes the notify() or notifyAll() method on the same object. It can also wait for a specified amount of time when the timed version of wait() is used. The wait() method is a non-static method of the Object class.

The syntax is:

```
public final void wait() throws InterruptedException
```

When wait() is called without a time limit, the thread remains in the waiting state until another thread calls notify() or notifyAll() on the same object. The thread must own the object's monitor before calling wait(). Therefore, it is normally called from a synchronized method or synchronized block. If a thread calls wait() without owning the object's monitor, an IllegalMonitorStateException is thrown.

For example:

```
synchronized(obj) {  
    obj.wait();  
}
```

Here, the thread releases the lock of obj and waits until it is notified.

notify() Method

The **notify() method** is used to wake up a single thread that is waiting on an object's monitor. If several threads are waiting on the same object, one of the waiting threads is selected by the Java Virtual Machine to be awakened. The selection is arbitrary and is determined by the implementation.

The syntax is:

```
public final native void notify();
```

The notify() method must be called by a thread that owns the object's monitor. Therefore, it is normally used inside a synchronized method or synchronized block.

For example:

```
synchronized(obj) {  
    obj.notify();  
}
```

The notify() method does not immediately give the lock to the awakened thread. The awakened thread must first compete with other threads to acquire the object's lock.

notifyAll() Method

The **notifyAll() method** is used to wake up all threads that are waiting on an object's monitor. When notifyAll() is called, all waiting threads become eligible to continue execution. However, they must acquire the object's lock before they can actually proceed.

The syntax is:

```
public final native void notifyAll();
```

For example:

```
synchronized(obj) {  
    obj.notifyAll();  
}
```

The notifyAll() method is particularly useful when several threads are waiting for a condition and more than one of them may be able to proceed after the shared resource is changed.

synchronized Keyword

The synchronized **keyword** in Java is used to control the access of multiple threads to a shared resource. When a method or block is declared as synchronized, only **one thread at a time** can execute that synchronized code

for a particular object. This prevents multiple threads from accessing and modifying shared data simultaneously, thereby avoiding data inconsistency.

Difference Between wait() and sleep()

The wait() and sleep() methods are both used to suspend the execution of a thread, but they have important differences. The wait() method belongs to the **Object class**, whereas sleep() belongs to the **Thread class**. The wait() method is a non-static method, while sleep() is a static method. The most important difference is that **wait() releases the lock held by the thread, whereas sleep() does not release the lock.**

The wait() method is mainly used for **inter-thread communication** and normally works together with notify() or notifyAll(). On the other hand, sleep() is mainly used when a thread needs to pause its execution for a specified period of time. The wait() method requires the thread to own the object's monitor and is therefore normally used within a synchronized method or block, whereas sleep() does not require synchronization.

Thus, **sleep() is used for timed suspension of a thread, while wait(), notify(), and notifyAll() are used for communication and cooperation between threads.**

Q. Explain sleep(), Wait() and notify() methods

Q. Explain inter thread communication

Introduction to Swing

Features of Swing:

Java Swing

Swing is an **Advanced Java concept** used to create **advanced Graphical User Interfaces (GUI)** in Java applications. It provides a rich set of components such as buttons, labels, text fields, tables, menus, and dialog boxes. Swing is also a **lightweight component**, which means its components are mainly created and managed by Java rather than depending directly on the operating system's GUI components.

Swing follows the **MVC (Model-View-Controller) architecture**, which helps in separating the data, user interface, and application logic. This makes GUI applications easier to develop, maintain, and modify. **Model** manages the data and application information, **View** displays the data and GUI components to the user, and **Controller** handles user actions and connects the View with the Model.

Swing is widely used for designing **advanced and interactive GUI applications** such as desktop applications, forms, calculators, banking applications, and management systems. It provides flexible and customizable components, making it suitable for developing user-friendly Java applications.

Q. Explain what are java swings?

Q. What is MVC Model

Swing Components are defined in javax.swing package. It can be understood as advanced concept of java.awt package. The different components are:

JFrame is similar to awt Frame object, but it has 2 panel called “root pane” and “content pane”. we obtain the content pane with the method getContentPane() and do setBackground(), setLayout() , add() etc.

JLabel : is similar to Label of java.awt package but provides good look and feel.

JButton : is similar to java.awt’s Button object, but we can also set an image using the ImageIcon class to the JButton.

JTextField : this is very similar to TextField of java.awt . but this class doesn’t have setEchoChar() to design it for passwords. For passwords, we use JPasswordField class.

JTextArea : this is like awt’s TextArea, but it has no size control with it. So we take help of class JScrollPane to set boundaries and scrollbars to the JTextArea.

JCheckBox and JRadioButton : in awt we used Checkbox objects grouped using CheckboxGroup to turn them into radio buttons. But in advanced java, i.e. in swings, we have different classes for these two components. We use JRadioButton used with ButtonGroup class to make up radio buttons.

JComboBox : this is like Choice class of java.awt. But the getSelectedItem() of Choice returns data as string instance and the same of JComboBox object returns data as Object class instance, so we must use toString() method to convert into string instances to do process on data.

Q. Explain the differences between awt and swing components.

Working with menus: the java.awt package has classes MenuBar, Menu and MenuItem to work with menus. Similarly the swings too have classes JMenuBar, JMenu and JMenuItem to work with menus.

MenuBar/ JMenuBar : this class provides a bar onto which Menu objects can be added. The menubar is set to Frame/JFrame with the method setMenuBar() / setJMenuBar().

Methods : add(Menu instance)

Menu / JMenu : this class provides Menu objects that can be added to the menu bar and get added with menu item objects.

Methods : add(MenuItem instance)
 addSeparator()

MenuItem / JMenuItem : this class provides to develop the objects to present menu options. These are added to the menu objects.

The menu related objects raise ActionEvent that can be handled by implementing ActionListener interface (similar to buttons).

Q. Explain working with menus.

Layout Managers : layout is a concept used to specify how the components are to be arranged on container. Layout can be set to any container with the method setLayout() of the Container class.

FlowLayout :this layout when set, the components are arranged as a flow. The location of the components is not fixed, it is based on the size of the container.

BorderLayout : this layout when set, it divides the container into 5 parts and only 5 components can be added. We can add 5 Panel class objects and add other components to them.

When this layout is set, the add() takes a different form :

add(component, int LOCATION)

the location is specified with built in static/final variables of the same class viz., NORTH, EAST, SOUTH, WEST and CENTER being default.

GridBagLayout :

When this layout is set, it divides container into a logical grid that has same number of columns in each row. the no. of rows / col.s are based on no. of components added in that col / row.

when this layout is set, the add() takes a different form,
add(component, location);

The location can be specified with the help of class GridBagConstraints.
The GridBagConstraints class has no. of variables to specify location of component.

gridx : used to specify col. no

gridy : used to specify row no.

gridwidth : no. of col.s to occupy

gridheight : used to specify no. of rows to occupy.

fill : used to specify how the component is to be filled in logical cell. its values are mentioned with built in symbolic constants (static & final variables) viz., BOTH, VERTICAL etc.

For flexible working with this layout, we can write a user defined method and add components.

Q. Explain what is layout manager and different layouts in java
