# Power Programming in Java

## Session: 6

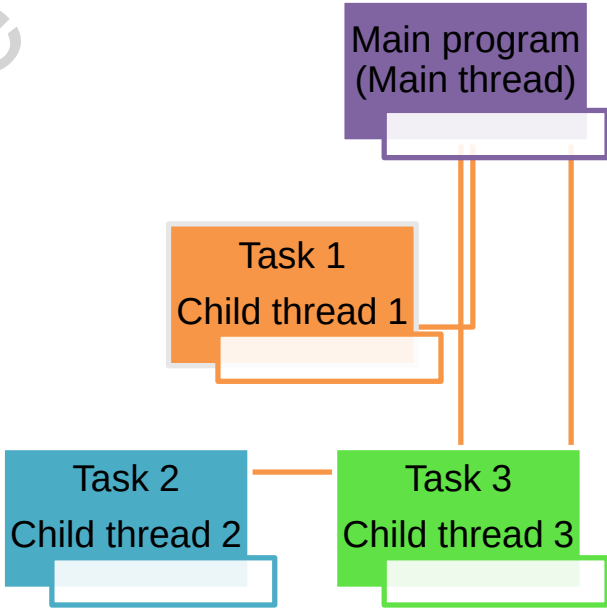

## Introduction to Threads

# Objectives

- Explain the Thread class
- Describe creating threads
- Identify Thread states
- List methods of Thread class
- Explain managing threads
- Describe daemon threads

# Introduction to Threads

- A process is a program that is executing.
- Each process has its own run-time resources, such as their own data, variables, and memory space.
- A thread is nothing but the basic unit to which the operating system allocates processor time.
- A thread is the entity within a process that can be scheduled for execution.
- A process is started with a single thread, often called the primary, default, or main thread.



# Characteristics of Threads

A thread has its own complete set of basic run-time resources to run it independently.

A thread is the smallest unit of executable code in an application that performs a particular job or task.

Several threads can be executed at a time, facilitating execution of several tasks of a single application simultaneously.

# Comparing Processes and Threads

## Similarities between processes and threads

- Threads share a central processing unit and only one thread is active (running) at a time.
- Threads within processes execute sequentially.
- A thread can create child threads or sub threads.
- If one thread is blocked, another thread can run.

## Differences between processes and threads

- Unlike processes, threads are not independent of one another.
- Unlike processes, all threads can access every address in the task.
- Unlike processes, threads are designed to assist one other.

# Application and Uses of Threads

## Applications of Threads

| | | |
|---|---|---|
| Playing sound and displaying images simultaneously. | Displaying multiple images on the screen. | Displaying scrolling text patterns or images on the screen. |

# Creating Threads

There are two ways to create `Thread` class:

- Creating a subclass of `Thread` class
- Implementing the `Runnable` Interface

# Subclassing `Thread`

Step by step procedure to create a new thread by extending `Thread` class:

- **Creating a Subclass of `Thread` class**
- **Overriding the `run()` Method**
- **Starting the Thread**

# Creating a Subclass

- Declare a class that is a subclass of the `Thread` class defined in the `java.lang` package.
- This creates a class `MyThread` which is a subclass of the `Thread` class.

Code Snippet

```
class MyThread extends Thread //Extending Thread class
{
    //class definition
     . . .
}
```

# Overriding the `run()` Method

- Inside the subclass, override the `run()` method defined in the `Thread` class.
- The code in the `run()` method defines the functionality required for the thread to execute.
- The `run()` method in a thread is analogous to the `main()` method in an application.

Code Snippet

```
class MyThread extends Thread  //Extending Thread class
{
        // class definition
        public void run()
        //overriding the run() method
        {
        // implementation
        }
        . . .
 }
```

# Starting the Thread

- The `main()` method creates an object of the class that extends the `Thread` class.
- Next, the `start()` method is invoked on the object to start the thread.
- This method will place the thread object in a runnable state.
- The `start()` method of a thread invokes the `run()` method which allocates the resources required to run the thread.

**Code Snippet**

```
public class TestThread
{
. . .
   public static void main(String args[])
   {
        MyThread t=new MyThread(); //creating thread object
        t.start();      //Starting the thread
   }
}
```

# Constructors of Thread Class

The ThreadGroup class represents a group of threads and is used in constructors of Thread class.

| Constructor | Description |
|---|---|
| Thread() | Default constructor |
| Thread(Runnable objRun) | Creates a new Thread object, where objRun is the object whose run() method is called. |
| Thread(Runnable objRun, String threadName) | Creates a new named Thread object, where objRun is the object whose run() method is called and threadName is the name of the thread that will be created. |
| Thread(String threadName) | Creates a new Thread object where threadName is the name of the thread that will be created. |
| Thread(ThreadGroup group, Runnable objRun) | Creates a new Thread object, where group is the ThreadGroup and objRun is the object whose run() method is called. |
| Thread (ThreadGroup group, Runnable objRun, String threadName) | Creates a new Thread object so that it has objRun as its run object threadName as its name and belongs to ThreadGroup referred to by group. |
| Thread (ThreadGroup group, Runnable objRun, String threadName, long stackSize) | Creates a new Thread object so that it has objRun as its run object, threadName as its name belongs to the ThreadGroup referred to by group, and the specified stack size. |
| Thread (ThreadGroup group, String threadName) | Creates a new Thread object with group as the ThreadGroup and threadName as the name of the thread that will be created. |

# Methods of `Thread` Class

Some methods of the `Thread` class are:

| Method | Description |
|---|---|
| `static int activeCount()` | Returns the number of active threads among the current threads in the program |
| `static Thread currentThread()` | Returns a reference to the currently executing thread object |
| `ThreadGroup getThreadGroup()` | Returns the `ThreadGroup` to which this thread belongs |
| `static boolean interrupted()` | Tests whether the current thread has been interrupted |
| `boolean isAlive()` | Tests if this thread is alive |
| `boolean isInterrupted()` | Tests whether this thread has been interrupted |
| `void join()` | Waits for this thread to die |
| `void setName(String name)` | Changes the name of this thread to be equal to the argument name |

# Constructor and Methods of `Thread` Class [1-3]

- Following example demonstrates the creation of a new thread by extending `Thread` class and using some of the methods of `Thread` class:

Code Snippet

```
/**
 * Creating threads using Thread class and using methods of the class
 */
package demo;
/**
 * NamedThread is created as a subclass of the class Thread
 */
 public class NamedThread extends Thread {
 /* This will store name of the thread */
       String name;
/**
  * This method of Thread class is overridden to specify the action
 * that will be done when the thread begins execution
  */
public void run() {
    // Will store the number of threads
       int count = 0;
     while(count<=3) {
      //Display the number of threads
           System.out.println(Thread.activeCount());
     //Display the name of the currently running thread
          name = Thread.currentThread().getName();
          count++;
          System.out.println(name);
```

# Constructor and Methods of `Thread` Class [2-3]

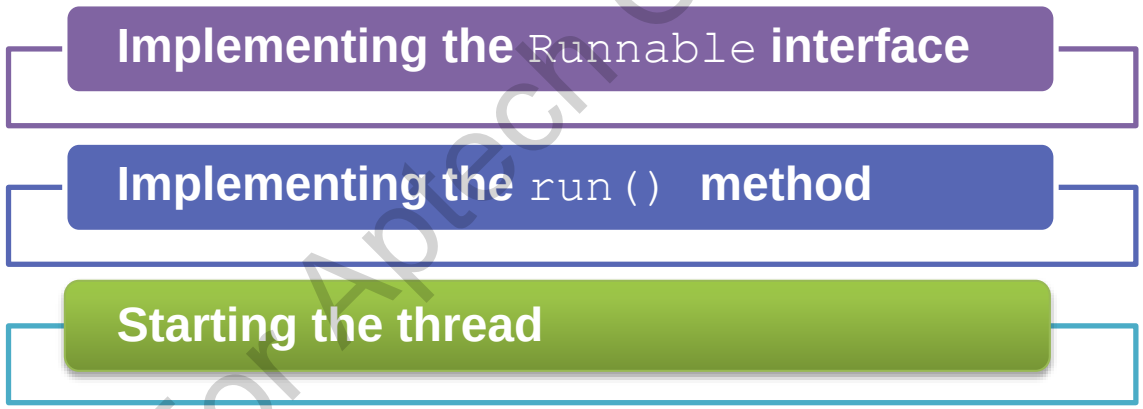
```
    if (name.equals ("Thread1"))
         System.out.println("Marimba");
     else
         System.out.println("Jini");
   }
}
public static void main(String args[]) {
     NamedThread objNamedThread = new NamedThread();
     objNamedThread.setName("Thread1");
   //Display the status of the thread, whether alive or not
     System.out.println(Thread.currentThread().isAlive());
     System.out.println(objNamedThread.isAlive());
   /*invokes the start method which in turn will call
     run and begin thread execution
    */
     objNamedThread.start();
    System.out.println(Thread.currentThread().isAlive());
    System.out.println(objNamedThread.isAlive());
   }
}
```

# Constructor and Methods of `Thread` Class [3-3]

- In the `main()` method of this class, a thread object `objNamedThread` is created by instantiating `NamedThread` and its name is set to Thread1.
- The code checks if the current thread is alive by invoking the `isAlive()` method and displays the return value of the method.
- This will result in `true` being printed because the `main(default)` thread has begun execution and is currently alive.
- The code also checks if `objNamedThread` is alive by invoking the same method on it.
- Next, the `start()` method is invoked on `objNamedThread` which will cause the thread to invoke the `run()` method which has been overridden.
- The `run()` method prints the total number of threads running then checks the name of the thread running and prints Marimba if the currently running thread's name is Thread1.

# Runnable Interface

- The Runnable interface provides a common set of rules for objects that wish to execute a code while a thread is active.
- Another way of creating a new thread is by implementing Runnable interface.
- This approach can be used because Java does not allow multiple class inheritance.
- Step by step procedure for creating and running a new thread by implementing the Runnable interface are:



# Implementing the `Runnable` Interface and the `run()` Method

- Declare a class that implements the `Runnable` interface.

**Code Snippet**

```
// Declaring a class that implements Runnable interface
class MyRunnable implements Runnable
{
      . . .
}
```

- The `Runnable` interface defines a method, `run()`, to contain the code that will be executed by the thread object.
- The class implementing the `Runnable` interface should override the `run()` method. Following code snippet implements the `run()` method:

**Code Snippet**

```
// Declaring a class that implements Runnable interface
class MyRunnable implements Runnable
{
    public void run()    // Implementing run()
      {
            . . .
          // implementation
     }
}
```

# Starting the Thread

- In the `main()` method, create an object of the class that implements the `Runnable` interface.
- Next, pass this object to the constructor to create an object of `Thread` class.
- Finally, invoke the `start()` method on the thread object to start the thread.

**Code Snippet**

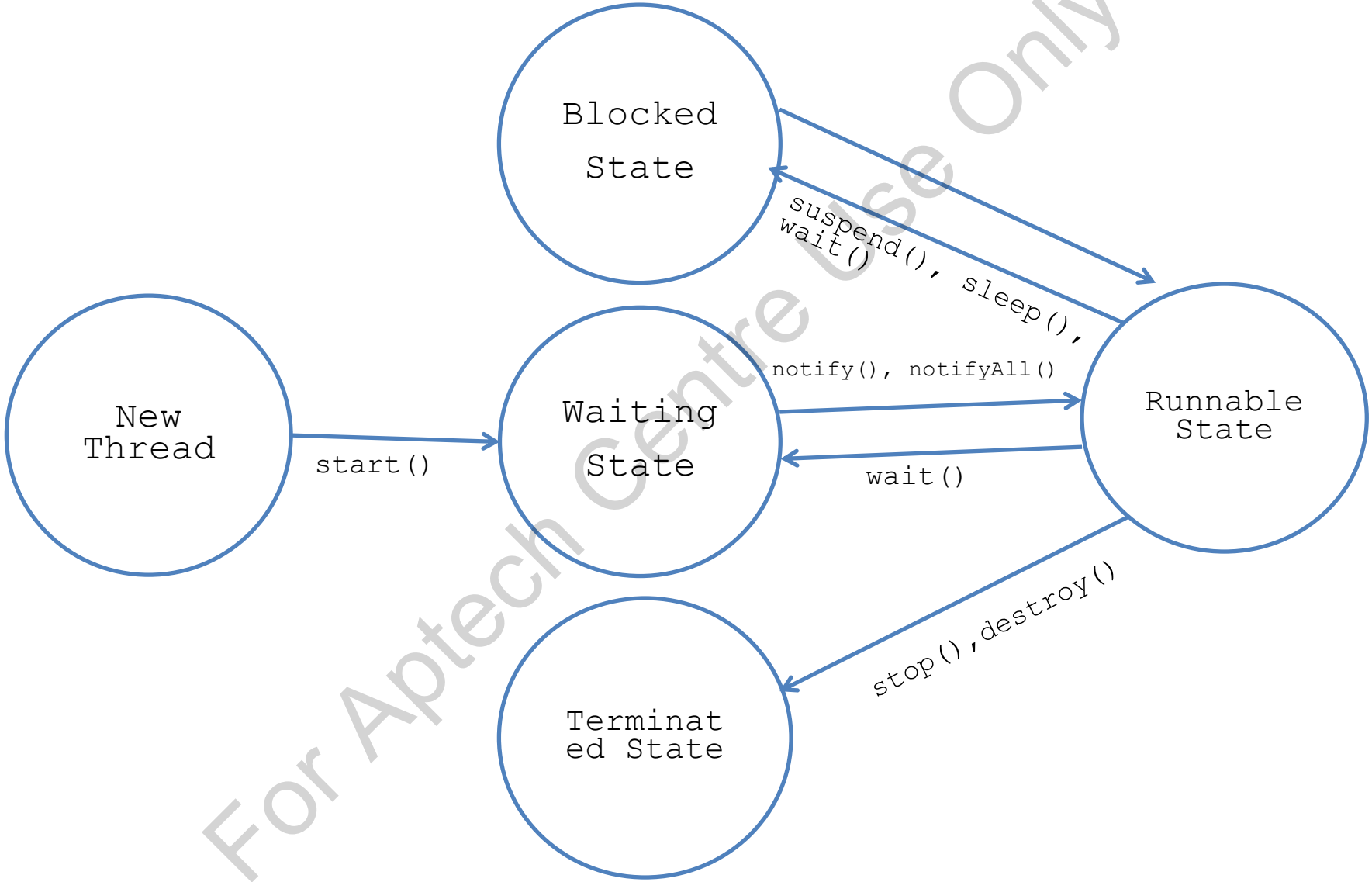
```
class ThreadTest
{
    public static void main(String args[])
    {
      MyRunnable r=new Runnable();
      Thread thObj=new Thread(r);
      thObj.start(); //Starting a thread
    }
}
```

# Example Using `Runnable` Interface

## Code Snippet

```
/*
 *Creating threads using Thread class and using methods of the class
 */
package test;
/**
* NamedThread is created so as to implement the interface Runnable
*/
class NamedThread implements Runnable {
        /* this will store name of the thread */
        String name;
         /* This method of Runnable is implemented to specify the action
         that will be done when the thread begins execution.     */
         public void run()  {
                int count = 0; //will store the number of threads
                while(count < 3){
                        name = Thread.currentThread().getName();
                        System.out.println(name);
                        count++;
                        }
                }
 }
public class Main {
        public static void main(String args[])     {
                NamedThread objNewThread= new NamedThread()
                Thread objThread = new Thread(objNewThread);
                objThread.start();
        }
}
```

# Life Cycle of a Thread



# New State of a `Thread`

- The thread can exists in various state – `new`, `runnable`, `blocked`, `waiting`, and `terminated` according to its various phases in program.
- When a thread is newly created, it is a new thread and is not alive.
- In this state, it is an empty thread object with no system resources allocated.
- So, the thread is left as a 'new thread' state till the `start()` method is invoked on it.
- When a thread is in this state, you can only start the thread or stop it.
- Calling any method before starting a thread raises an `IllegalThreadStateException`.

Code Snippet

```
. . .
Thread thObj = new Thread();
. . .
```

# Runnable State of a Thread

- A new thread can be in a runnable state when the `start()` method is invoked on it.
- A thread in this state is alive.
- A thread can enter this state from running or blocked state.
- Threads are prioritized because in a single processor system all runnable threads cannot be executed at a time.

**Code Snippet**

```
. . .
MyThreadClass myThread = new MyThreadClass();
myThread.start();
. . .
```

# Blocked State of a `Thread`

Blocked state is one of the states in which a thread:

- Is alive, but currently not eligible to run as it is blocked for some other operation.
- Is not runnable but can go back to the runnable state after getting the monitor or lock.

A thread in blocked state waits to operate on resource or object which at the same time is being processed by another thread.

A running thread goes to blocked state when `sleep()`, `wait()`, or `suspend()` method is invoked on it.

# Waiting State of a `Thread`

A thread is in waiting state when it is waiting for another thread to release resources for it.

When two or more threads run concurrently and only one thread takes hold of the resources all the time, other threads ultimately wait for this thread to release the resource for them.

In this state, a thread is alive but not running.

A call to the `wait()` method puts a thread in this state.

Invoking the `notify()` or `notifyAll()` method brings the thread from the waiting state to the runnable state.

# Terminated State of a `Thread`

A thread, after executing its `run()` method dies and is said to be in a terminated state.

This is the way a thread can be stopped naturally.

Once a thread is terminated, it can not be brought back to runnable state.

Methods such as `stop()` and `destroy()` can force a thread to be terminated, but in JDK 1.5, these methods are deprecated.

# Methods of `Thread` Class

Some important methods of `Thread` Class are:

- `getName()`
- `start()`
- `run()`
- `sleep()`
- `interrupt()`

# getName() Method

- All threads have a name associated with it.
- At times, it is required to retrieve the name of a particular thread.
- The `getName()` method helps to retrieve the name of the current thread.
- Code snippet demonstrates the use of `getName()` method to obtain the name of the thread object.

**Code Snippet**

```
public void run()
 {
for(int i = 0;i < 5;i++)
{
Thread t = Thread.currentThread();
 System.out.println("Name = " + t.getName());
 ...
}
}
```

# start() Method

- A newly created thread remains idle until the `start()` method is invoked.
- The `start()` method allocates the system resources necessary to run the thread and executes the `run()` method of its target object.
- At the time of calling the `start()` method:
  - A new thread execution starts
  - The thread moves from the new thread to runnable state

**Syntax**: `void start()`

Code Snippet

```
. . .
NewThread thObj = new NewThread();
thObj.start();
. . .
```

# run() Method

- The life of a thread starts when the `run()` method is invoked.
- The characteristics of the `run()` method are:
  - It is public
  - Accepts no argument
  - Does not return any value
  - Does not throw any exceptions
- The `run()` method contains instructions, which are executed once the `start()` method is invoked.

**Syntax:** `public void run()`

Code Snippet

```
class myRunnable implements Runnable {
      . . .
    public void run() {
   System.out.println("Inside the run method.");
    }
   ....}
```

# sleep() Method

- The `sleep()` method has the following characteristics:
  - It suspends the execution of the current thread for a specified period of time.
  - It makes the processor time available to other threads of an application or other applications that might be running on the computer system.
  - It stops the execution if the active thread for the time specified in milliseconds or nanoseconds. It raises `InterruptedException` when it is interrupted using the `interrupt()` method.

**Syntax**: `void sleep(long millis)`

Code Snippet

```
try
 {
    myThread.sleep (10000);
 }
catch (InterruptedException e)
 {
}
```

# interrupt() Method

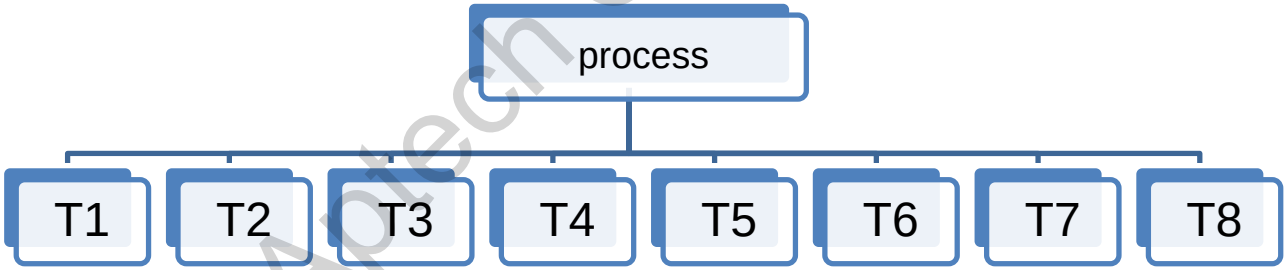
- The `interrupt()` method interrupts the thread.
- The method tells the thread to stop what it was doing even before it has completed the task.
- The `interrupt()` method has the following characteristics:

1. An interrupted thread can die, wait for another task, or go to next step depending on the requirement of the application.
2. It does not interrupt or stop a running thread; rather it throws an `InterruptedException` if the thread is blocked, so that it exits the blocked state.
3. If the thread is blocked by `wait()`, `join()`, or `sleep()` methods, it receives an `InterruptedException`, thus terminating the blocking method prematurely.

**Syntax**: `public void interrupt()`

# Managing Threads

- Threads are self-executing entities inside a program.
- In a single program, several threads can execute independent of each other.
- Sometimes, it may happen that a particular run-time resource has to be shared by many threads, running simultaneously.
- This forces other running threads to enter the blocked state.
- So, in such situations some internal control or management of the threads is required so that the threads are executed simultaneously.




**Managing Threads**

# Necessity for `Thread` Priority

While creating multi-threaded applications, situations may come up where a thread is already running and you must run another thread of greater importance.

This is where thread priorities play an important role.

Priorities are used to express the importance of different threads.

Priorities play an important part when there is a heavy contention among threads trying to get a chance to execute.

This prioritizing process is managed by the scheduler which assigns priority to the respective threads.

# Types of `Thread` Priority

- Thread priority helps the thread scheduler to decide which thread to run.
- Thread priorities are integers ranging from **MIN_PRIORITY** to **MAX_PRIORITY**.
- Higher the integers, the higher are the priorities.
- Higher priority thread gets more CPU time than a low priority thread.
- `Thread` priorities in Java are constants defined in the `Thread` class.
- They are:
  - `Thread.MAX_PRIORITY`: It has a constant value of 10. It has got the highest priority.
  - `Thread.NORM_PRIORITY`: It has a constant value of 5. It is the default priority.
  - `Thread.MIN_PRIORITY`: It has a constant value of 1. It has the lowest priority.

# setPriority() Method

- A newly created thread inherits the priority from the thread that created it.
- To change the priority of a thread the `setPriority()` method is used.
- The `setPriority()` method changes the current priority of any thread.
- The `setPriority()` method accepts an integer value ranging from 1 to 10.

**Syntax**: `public final void setPriority(int newPriority)`

Code Snippet

```
. . .
Thread threadA = new Thread("Meeting deadlines");
threadA.setPriority(8);
. . .
```

# `getPriority()` Method

- The `getPriority()` method helps to retrieve the current priority value of any thread.
- A query to know the current priority of the running thread to ensure that the thread is running in the required priority level.

  **Syntax**: `public final int getPriority()`

# Daemon Threads [1-2]

- A daemon thread runs continuously to perform a service, without having any connection with the overall state of the program.
- In general, the threads that run system codes are good examples of daemon threads.
- The characteristics of good daemon threads are:

1. They work in the background providing service to other threads.
2. They are fully dependent on user threads.
3. Java virtual machine stops once a thread dies and only daemon thread is alive.

# Daemon Threads [2-2]

The `Thread` class has two methods related to daemon threads. They are:

- `setDaemon(boolean value)`
  - The `setDaemon()` method turns a user thread to a daemon thread.
  - It takes a boolean value as its argument.
  - To set a thread as daemon, the `setDaemon()` method is invoked with true as its argument.
  - By default every thread is a user thread unless it is explicitly set as a daemon thread previously.

    **Syntax:** `void setDaemon(boolean val)`
- `isDaemon()`
  - The `isDaemon()` method determines if a thread is a daemon thread or not.
  - It returns true if this thread is a daemon thread, else returns false.

    **Syntax**: `boolean isDaemon()`

# Necessity for Daemon Threads

Tasks performed by the `Daemon` threads are:

1. Daemon threads are service providers for other threads running in the same process.
2. Daemon threads are designed as low-level background threads that perform some tasks such as mouse events for Java program.

# Summary

- A process consists of many semi-processes known as threads.
- A thread is the smallest executable code in a Java program, which contributes in making it possible to run multiple tasks at a time.
- A thread object can be created by extending the `Thread` class that defines the `run()` method.
- It can also be created by declaring a class that implements the `Runnable` interface and passing the object to the `Thread` constructor.
- A newly created thread can be in following states: new, runnable, waiting, blocked, and terminated.
- The `getName()` method returns the name of the thread while `start()` allows a thread to be in runnable state.
- The `setPriority()` and `getPriority()` methods are used to assign and retrieve the priority of any thread respectively.
- The daemon thread runs independently of the default user thread in a program, providing background services such as mouse event, garbage collection, and so on.