Object Oriented Programming using Java

Prepared By:
Suyel, PhD
Assistant Professor
Dept. of CSE, NIT Patna

Outline

1. Basics of Threading
2. Thread Life-Cycle
3. Creating Threads

Basics of Threading

- **Process**
  - In computing, a process is the instance of a computer program that is being executed by one or many threads.
  - It contains the program code and its activity.
  - It can be seen as a program or application.
  - Java runtime environment runs as a single process.
- **Thread**
  - A thread is a very light-weighted process or we can say the smallest part of a process.
  - It allows a program to operate more efficiently by running multiple tasks simultaneously.

Basics of Threading

Thread (Cont…)

- When a thread is invoked, there are two paths of execution. One path executes the thread and the other path follows the statement after the thread invocation.
- Proper co-ordination is must required between threads accessing common variables for maintaining data consistency.
- Overuse of Java threads can be hazardous to program's performance and its maintainability.
- Java's multithreading system is built upon the Thread class.

Basics of Threading (Cont…)

- **Benefits to Use Thread**
  - Threads are lightweight compared to processes. It takes less time and resource to create a thread.
  - To execute complicated tasks in the background, we can use threads.
  - Here, all the tasks are executed without affecting the main program.
  - All the threads of a program or a process have their own separate path for execution, so each thread of a process is independent.
  - Threads share their parent process data and code.
  - If a thread has an exception or an error at the time of its execution, it doesn't affect the execution of the other threads.
  - All the threads share a common memory and have their own stack, local variables and program counter.

Thread Life-Cycle

- A Java thread at any point of time exists in any one of the following states:
  - New
  - Runnable
  - Waiting/Blocked
  - Timed Waiting
  - Terminated/Dead
- **New**
  - When a new thread is created, it is in the new state.
  - The thread has not yet started to run when it is in this state.

Thread Life-Cycle (Cont…)

- **Runnable**
  - A thread that is ready to run is moved to runnable state.
  - In this state, a thread might actually be running or it might be ready to run at any instant of time.
  - Thread scheduler is responsible to give the time to run.
  - A multi-threaded program allocates a fixed amount of time to each thread.
  - In a multithread environment, each and every thread runs for a short while, and then, pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run.

Thread Life-Cycle (Cont…)

- **Waiting/Blocked**
  - A thread is in the blocked state, when it tries to access a protected section of code that is currently locked by some other thread.
  - When the protected section is unlocked, the schedule picks one of the threads that is blocked for that section and moves it to runnable state.
  - A thread is in the waiting state when it is waiting for some other thread to perform a particular action/condition.
  - It is the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread.
  - A thread in this state cannot continue its execution any further until it is moved to runnable state.
  - Any thread in these states does not consume any CPU cycle.

Thread Life-Cycle (Cont…)

- **Timed Waiting**
  - A runnable thread can enter the timed waiting state for a specified interval of time.
  - A thread in this state transitions backs to the runnable state, when that time interval expires or when the event it is waiting for occurs or a notification is received.
  - Example: When a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

Thread Life-Cycle (Cont…)

- **Terminated (Dead)**
  - There are mainly two conditions for terminating a thread:
    1. It exists normally. This happens, when the code of thread has entirely executed by the program.
    2. There occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

Thread Life-Cycle (Cont…)

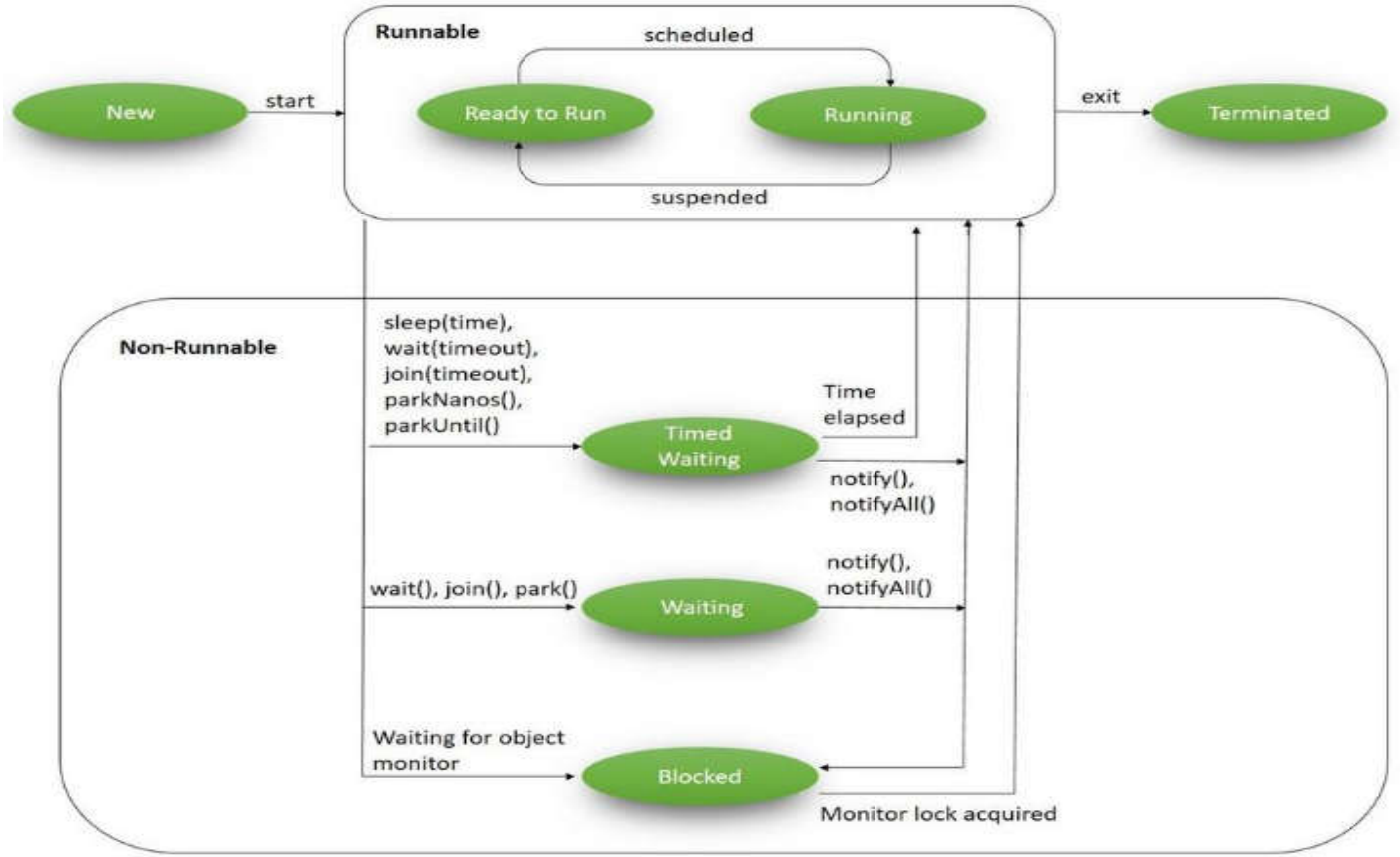


Fig. 1: Life-cycle of a thread

Creating Threads

- There are two ways to create a thread:
  1. Implementing the Runnable interface
  2. Extending the Thread class

- Each Java program creates at least one thread, i.e. main() thread. The main thread is important for two reasons:
  1. It is the thread from which other threads is spawned.
  2. Often, it must be the last thread to finish execution because it performs various shutdown actions.

Creating Threads

- Thread class has many methods that help to manage threads. Some are:

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Creating Threads (Cont…)

- Although the main thread is created automatically, when our program is started, it can be controlled by a Thread object.
- To do so, we must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread. Its general form is shown here:

```
static Thread currentThread( )
```

This method returns a reference to the thread in which it is called.

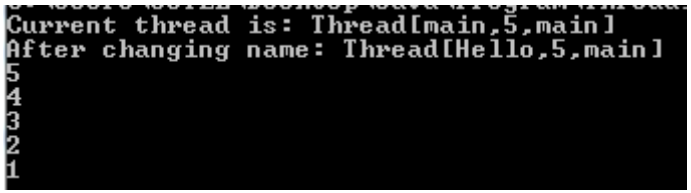
- Main is also the name of the group of threads to which this thread belongs.
- A **thread group** is a data structure that controls the state of a collection of threads as a whole.

Creating Threads (Cont...)

```
class Thread1
{
   public static void main(String args[])
   {
      Thread t = Thread.currentThread();
      System.out.println("Current thread is: " + t);
      t.setName("Hello");
      System.out.println("After changing name: " + t);
      try{
            for(int n = 5; n > 0; n--)
            {
               System.out.println(n);
               Thread.sleep(1000);
            }
         }catch (InterruptedException e)
          {
             System.out.println("Main thread interrupted");
          }
   }
}
```

Creating Threads (Cont...)

- **Output**



There will be a delay, when printing 5, 4, 3, 2 and 1.

Creating Threads (Cont...)

Implementing the Runnable Interface

- The easiest way to create a thread is to create a class that implements the Runnable interface.
- We can construct a thread on any object that implements Runnable.
- If a class implements the Runnable interface, the thread can be run by passing an instance of the class to a Thread object's constructor, and then, calling the thread's start().
- There are three main steps:

  Step 1: Here, we need to implement a run() method provided by a Runnable interface. This method provides an entry point for the thread and we must give logic/code inside this method. Syntax:

  public void run( )

Creating Threads (Cont...)

- **Implementing the Runnable Interface (Cont…)**
  - There are three main steps (Cont…):

Step 2: In this step, we instantiate an object of type Thread from within the class. Thread defines several constructors. One is:

Thread(Runnable threadOb, String threadName)

Here, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.

Step 3: After the new thread is created, it will not start running until we call its start( ) method, which is declared within Thread. In essence, start( ) executes a call to run( ). Syntax:

void start( )

Creating Threads (Cont...)

- **Implementing the Runnable Interface (Cont…)**

```
class ThreadTest implements Runnable
{
   Thread t;
   ThreadTest()
   {
      t = new Thread(this, "Demo Thread");  //Creating the second thread
      System.out.println("Child thread: " + t);
      t.start();           // Start the thread
   }
   public void run()      //Entry point of the second thread
   {
      try{
            for(int i = 5; i > 0; i--)
            {
               System.out.println("Child Thread: " + i);
               Thread.sleep(500);
            }
         }catch (InterruptedException e)
          {
             System.out.println("Child interrupted.");
          }
      System.out.println("Child thread exit");
   }
}

class Thread2
{
   public static void main(String args[])
   {
      new ThreadTest(); //create a new thread
      try{
            for(int i = 5; i > 0; i--)
            {
               System.out.println("Main Thread: " + i);
               Thread.sleep(1000);
            }
         }catch (InterruptedException e)
          {
             System.out.println("Main thread interrupted.");
          }
      System.out.println("Main thread exit");
   }
}
```

Creating Threads (Cont...)

- **Implementing the Runnable Interface (Cont…)**
  - **Output**

```
Child thread: Thread[Demo Thread,5,main]
Child Thread: 5
Main Thread: 5
Child Thread: 4
Child Thread: 3
Main Thread: 4
Child Thread: 2
Main Thread: 3
Child Thread: 1
Child thread exit
Main Thread: 2
Main Thread: 1
Main thread exit
```

Creating Threads (Cont...)

Extending the Thread Class

- If a class extends the Thread class, the thread can be run by creating an instance of the class and call its start().
- This approach provides more flexibility in handling multiple threads created using available methods in Thread class. There are two steps:

  Step 1: In the first step, we need to override run( ) method of the Thread class. This method provides an entry point for the thread and we give codes inside this method. Syntax:

  public void run( );

  Step 2: Once Thread object is created, we can start it by calling start() method that executes a call to run( ) method. Syntax:

  void start( );

Creating Threads (Cont...)

- **Extending the Thread Class (Cont…)**

```
class Thread3 extends Thread
{
   public void run()
   {
      int x = 5, y = 10, z;
      z = x+y;
      System.out.println("Thread is started");
      System.out.println("Sum of x and y is: " + z);
   }
   public static void main(String args[])
   {
      Thread3 t = new  Thread3();
      t.start();
   }
}
```

Creating Threads (Cont...)

- **Extending the Thread Class (Cont…)**
  - **Output**

```
Thread is started
Sum of x and y is: 15
```

Creating Threads (Cont…)

Comparing Both Techniques

Thread Class	Runnable Interface
It supports comparatively advanced functionality by using suspend(), resume(), etc.	It provides basic functionality.
Thread class is not recommended for better object-oriented design.	It supports better object-oriented design and consistency and also avoid the single inheritance problems.
It has simple code structure.	Here, code structure is comparative complex.
Extending the Thread class makes our class unable to extend other classes because of the single inheritance feature of Java.	It supports to achieve multiple inheritance.

thank you . . .

Slides are prepared from various sources, such as Book, Internet Links and many more.