Java Workshop

DDUC ACM STUDENT CHAPTER
DEEN DAYAL UPADHYAYAYA COLLEGE
UNIVERSITY OF DELHI

Multithreaded Programming

AN IMPORTANT CONCEPT HELPING IN WRITING EFFICIENT PROGRAMS THAT MAKE MAXIMUM USE OF THE PROCESSING POWER AVAILABLE IN THE SYSTEM

Multithreaded Programming

- Unlike some other computer languages, Java provides built-in support for multithreaded programming.
- Threads are parts of a program which can run concurrently.
- Threads share the same address space.
- Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost.

The Java Thread Model

- The value of a multithreaded environment is best understood in contrast to its counterpart.
- Single-threaded systems use an approach called an ***event-loop*** with ***polling***.
  - A single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.
  - In a single-threaded environment, when a thread *blocks*, because it is waiting for some resource, the entire program stops running.
- In Java's multithreaded environment, the main *loop/polling* mechanism is eliminated.
  - One thread can pause without stopping other parts of program.

Threads

- Java's multithreading feature work in both type of systems i.e. single-core and multi-core systems.
- In a single-core system, two or more threads do not actually run at the same time but receive slices of CPU time.
- At the same place, in a multi-core system, it is possible for two or more threads to actually execute simultaneously.

Threads

- Threads exist in several states.
  - A thread can be *running*.
  - A thread can be *ready to run* as soon as it gets CPU time.
  - A running thread can be *suspended*, which temporarily halts its activity.
  - A suspended thread can then be *resumed*, allowing it to pick up where it left off.
  - A thread can be *blocked* when waiting for a resource.
  - At any time, a thread can be *terminated*, which halts its execution immediately.
    - After termination, a thread cannot be resumed.

Thread Priorities

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.
- Thread priorities are integers that specify the relative priority of one thread to another.
- As an absolute value, a priority is meaningless.
- A thread's priority is used to decide when to switch from one running thread to the next. This is called ***context switch***.

Rules to Determine Context Switch

- A *thread can voluntarily relinquish control.*
  - This is done by explicitly yielding, sleeping, or blocking on pending I/O.
  - In this scenario, all other threads are examined, and the highest priority that is ready to run is given the CPU.
- A *thread can be preempted by a higher-priority thread.*
  - In this case, a lower-priority thread that does not yield the processor is simple preempted-no matter what it is doing-by a higher-priority thread.
  - Basically as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking*.
- Situation is a bit complicated when two threads with same priority are competing for CPU.

Synchronization

- Multithreading introduces an asynchronous behavior to programs, so there must be a way to enforce synchronicity when it is needed.
- Java implements an elegant twist on an age-old model of inter-process synchronization: the ***monitor*** to ensure synchronization.
- Monitor can be think like a very small box that can hold only one thread.
- Most multithreaded systems expose monitors as objects that a program must explicitly acquire and manipulative.
- But, Java provides a cleaner solution. There is **no** class "Monitor"; instead each object has its own implicit monitor that is automatically entered when one of the object's synchronized method is called.

Messaging

- Threads need to communicate with each other.
- Some languages provide a costly way for to establish communication by depending on operating system.
- Java provides a clean and low-cost way for this via calls to predefined methods that all object have.
- Java's messaging system mallows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**.
- **Thread** encapsulates a thread of execution.
- A running thread's state cannot be directly referred so one have to deal with it through its proxy, the **Thread** instance that spawned it.
- To create a new thread, either **Thread** class is to be extended or **Runnable** interface is to be implemented.

The Thread Class

- **Thread** class defines several methods that help manage threads. Some of these are:

Method	Meaning
getName	Obtain a thread's name.
setName	To set 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.

The Thread Class

- **Thread** class provide several constructors. Basic form of this constructors is: Thread(ThreadGroup *group*, Runnable *target*, String *name*).
  - Here, *group* is thread group. If null then either security manager decides the group or group is set to the current thread's thread group.
  - *target* is the object whose run method is invoked when this thread is started. If null, this thread's run method is invoked.
  - *name* is the name of the new thread. If null, then new name of the form "Thread-"+*n*, where *n* is an integer; is generated and is given to thread.

The Thread Class

- Different constructors of **Thread** class are:
  - Thread() equivalent to Thread(null, null, *gname*)
  - Thread(Runnable *target*) equivalent to Thread(null, *target*, *gname*)
  - Thread(String *name*) equivalent to Thread(null, null, *name*)
  - Thread(Runnable *target*, String *name*) equivalent to Thread(null, *target*, *name*)
  - Thread(ThreadGroup *group*, Runnable *target*) equivalent to Thread(*group*, *target*, *gname*)
  - Thread(ThreadGroup *group*, String *name*) equivalent to Thread(*group*, null, *name*)
  - Thread(ThreadGroup *group*, Runnable *target*, String *name*)
- For these constructors *gname* is automatically generated name of form "Thread-"+*n*, where *n* is an integer.

The Main Thread

- When a Java program starts up, one thread begins running immediately.
- This thread is usually called the *main thread* of program.
- This thread is important for two reasons:
  - It is the thread from which other "child" threads will be spawned.
  - Often, it must be the last thread to finish execution because it performs various shutdown actions.
- Although the main thread is created automatically when program starts, it can be controlled through a **Thread** object.
- currentThread() method of **Thread** class can be used to get a reference for the main thread.
  - General form is Thread currentThread().
  - This is a public static member of **Thread** class.

The Main Thread

```
// Controlling the main Thread.
class CurrentThreadDemo {
   public static void main(String args[]) {
      Thread t = Thread.currentThread();
     System.out.println("Current thread: " + t);
     // change the name of the thread
     t.setName("My Thread");
     System.out.println("After name change: " + 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");
     }
   }
}
```

- Output of the code on left is:

```
Current Thread: Thread[nain, 5, main]
After name change: Thread[My Thread, 5, main]
5
4
3
2
1
```

- sleep takes its arguments in milliseconds.
- Another implementation of sleep also accepts period in terms of milliseconds and nanoseconds.
- Default priority is 5 for a thread, maximum priority is 10, and minimum priority is 1. These priorities are defined in **Thread** class with names **NORM_PRIORITY**, **MAX_PRIORITY**, and **MIN_PRIORITY** respectively.
- toString method of **Thread** class is overridden in this way:
  - "Thread[" + getName() + "," + getPriority() + "," +group.getName() + "]";

Creating a Thread by Implementing Runnable

- This is the easiest way to create a thread.
- **Runnable** abstracts a unit of executable code.
- To implement **Runnable**, a class need to implement a single method called **run()** only.
  - General form of this method is void run().
- Inside **run()**, the code which constitutes the new thread is defined.
- **run()** can call other methods, use other classes, and declare variables, just like a main thread can.
- Main difference is that **run()** establishes the entry point for another, concurrent thread of execution within the program.
- This thread will end when **run()** returns.

Creating a Thread by Implementing Runnable

- After creating a class that implements **Runnable**, an object of type **Thread** is to be instantiated.
- After the new thread is created, it will not start running until you call its **start()** method which is declared within **Thread** class.
  - General form of the method is void start().

Create a Thread by Implementing Runnable

```
// Create a second thread.
class NewThread implements Runnable {
    Thread t;
    NewThread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread
    }
    // This is the entry point for the second thread.
    public void run() {
        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("Exiting child thread.");
    }
}
```

```
class ThreadDemo {
    public static void main(String args[]) {
        new NewThread(); // 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 exiting.");
    }
}
```

Create a Thread by Implementing Runnable

- Output of this program might be:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Create a Thread by Implementing Runnable

- Passing **this** as the first argument indicates that new thread will cal **run()** method of **this** object.
- **start()** starts the thread of execution beginning at the **run()** method. This causes execution of child thread.
- After calling **start()**, **NewThread**'s constructor returns to main. When main thread resumes, its starts execution.
- Both threads then continue running simultaneously.

Create a Thread by Extending Thread

- Second way to create a thread is to create an instance of a class that extends **Thread**.
- The extending class must override the **run()** method, which is the entry point for the new thread.
- All other things for this method remain same as previous case.
- Example for this case is on the next slide and its out will also be somewhat similar to previous example.

Create a Thread by Extending Thread

```
// Create a second thread.
class NewThread extends Thread {
    NewThread() {
        // Create a new, second thread
        super("Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread
    }
    // This is the entry point for the second thread.
    public void run() {
        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("Exiting child thread.");
    }
}
```

```
class ThreadDemo {
    public static void main(String args[]) {
        new NewThread(); // 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 exiting.");
    }
}
```

Choosing an Approach

- Now the question arises why Java has two ways to create child threads, which one is better.
- **Thread** class defines several methods that can be overridden by a derived class. One method that *must* be overridden is **run()**.
- If no other method is to be overridden in derived class, its good to implement **Runnable** rather than extending **Thread** as we are not enhancing or modifying something.
- Also by implementing **Runnable**, our thread class does not need to inherit **Thread**, so we are free to inherit a different class.
- Which approach should be used is in this way is the programmer's wish.

Creating Multiple Threads

```
// Create multiple threads.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e) {
            System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}
```

```
class MultiThreadDemo {
    public static void main(String args[]) {
        new NewThread("One"); // start threads
        new NewThread("Two");
        new NewThread("Three");
        try {
            // wait for other threads to end
            Thread.sleep(10000);
        }
        catch (InterruptedException e) {
            System.out.println("Main thread
                                        Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}
```

Creating Multiple Threads

- Sample output of previous program (which may vary) is:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

Using isAlive() and join()

- To force main thread to finish last then call **sleep()** within **main()**, with a long enough delay to ensure that all child thread terminates prior to main thread.
- But this is not a satisfactory solution and raises another question.
- How can one thread know whether another threaded has ended or not?
- **Thread** provides isAlive method to answer this question.
  - General form of the method is boolean isAlive().
  - It returns true if thread upon which it is called is running, otherwise false.

Using isAlive() and join()

- Another method which waits until the thread on which it is called terminates also exist. This method is join() and overloaded in many ways. General form of the method are:
  - void join().
  - void join(long *milliSeconds*)
  - void join(long *milliSeconds*, int *nanoSeconds*)
  - It throws IllegelArgumentException if value of *milliSeconds* is negative, or value of *nanoSeconds* is not in range 0-999999.
  - It throws InterruptedException if any thread has interrupted the current thread.

Using isAlive() and join()

```
// Create multiple threads.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }

    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e) {
            System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}
```

```
class MultiThreadDemo {
    public static void main(String args[]) {
        NewThread ob1=new NewThread("One");
        NewThread ob2=new NewThread("Two");
        NewThread ob3=new NewThread("Three");
        System.out.println("ob1 is alive: "+ob1.isAlive());
        System.out.println("ob2 is alive: "+ob2.isAlive());
        System.out.println("ob3 is alive: "+ob3.isAlive());
        //wait for threads to finish
        try {
            System.out.println("Waiting for threads to
                                                finish");
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread
                                        Interrupted");
        }
        System.out.println("ob1 is alive: "+ob1.isAlive());
        System.out.println("ob2 is alive: "+ob2.isAlive());
        System.out.println("ob3 is alive: "+ob3.isAlive());
        System.out.println("Main thread exiting.");
    }
}
```

Using isAlive() and join()

- Sample output of previous program (which may vary) is:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One: 5
New thread: Thread[Three,5,main]
Two: 5
ob1 is alive: true
ob2 is alive: true
ob3 is alive: true
Waiting for threads to finish.
Three: 5
One: 4
Two: 4
Three: 4
Three: 3
One: 3
Two: 3
One: 2
Two: 2
Three: 2
Three: 1
One: 1
Two: 1
One exiting.
Three exiting.
Two exiting.
ob1 is alive: false
ob2 is alive: false
ob3 is alive: false
Main thread exiting.
```

ThreadGroup

- **ThreadGroup** creates a group of threads.
- Its constructors are:
  - ThreadGroup(String *groupName*)
    - Here, *groupName* specifies the name of thread group.
    - Current thread becomes parent of this group.
  - ThreadGroup(ThreadGroup *parentObject*, String *groupName*)
    - Here, *groupName* specifies the name of thread group.
    - *parentObject* specifies the parent of this group.

Methods of ThreadGroup Class

- int activeCount()
  - Returns approximate number of active threads in the invoking group and subgroups.
- int activeGroupCount()
  - Returns approximate number of active groups including subgroups for which the invoking thread is a parent.
- final void destroy()
  - Destroys the thread group and any child group on which it is called.
- int enumerate(Thread *group[]*)
  - Puts the active threads that comprise the invoking thread group and subgroups into the *group* array.
- int enumerate(Thread *group[]*, boolean *all*)
  - Puts active threads of group in *group* array and if *all* is **true** then threads of subgroups will also be put in *group*.

Methods of ThreadGroup Class

- int enumerate(ThreadGroup *group[]*)
  - Puts the active subgroups (and subgroups of subgrooups and so on) of the invoking thread group into the *group* array.
- int enumerate(ThreadGroup *group[]*, boolean *all*)
  - Puts active subgroups of invoking thread group in *group* array and if *all* is **true** then all active subgroups of the subgroups (and so on) are also put in *group*.
- final int getMaxPriority()
  - Returns the maximum priority setting for the group.
- final String getName()
  - Returns the name of the group.
- final ThreadGroup getParent()
  - Returns parent of invoking object if exists, otherwise **null**.
- void list()
  - Displays information about the group.

Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one threat at a time.
- The process by which it is achieved is called *synchronization*. Java provides language-level, unique support for it.
- **synchronized** is the keyword used wherever you will see synchronization.
- Synchronization can be achieved by using either a synchronized method or synchronized block.

Synchronization

- Any method which might be accessing any resource must be synchronized so that only one thread at time can access the resource with help of that method.
- To declare a method as a synchronized one just put synchronized keyword before it.
  - For example: synchronized void access(String *resource*)
- When more than one thread race each other to complete a method, *race condition* occurs.
- Race condition might be more subtle and less predictable hence we must *serialize* access to the method for which a race condition occurs.

Synchronization

- Sometimes it is not possible for you to have synchronized methods in a class because those classes might not be meant for that purpose or you might not be having the source code.
- Fortunately we have a solution for this problem also. Simply put calls to the methods defined by this class inside a synchronized block.
  - For example:

```
synchronized(object){
    //statements to be synchronized
}
```

  - Here, *object* is reference to the object being synchronized. A synchronized block ensures that each thread waits for the prior one to finish before proceeding.