Dr. Ambedkar Institute of Technology, Bengaluru

Department of Computer Science & Engineering

UG Programme

Subject Name: JAVA Programming

Subject Code: 18CS52

PRESENTATION BY

Dr. SMITHA SHEKAR B
ASSOCIATE PROFESSOR
DEPT. OF C.S.E
DR. AMBEDKAR INSTITUTE OF TECHNOLOGY
BANGALORE-560056

Multi Threaded Programming

- Thread model - Life Cycle, Thread Priorities, Synchronization, Messaging, **The Thread Class and the Runnable Interface**
- The Main Thread
- Creating a Threads
- Using isAlive() and join()
- Thread priorities
- Synchronization
- Inter-thread communication
- Deadlock

Multithreading is a **Java** feature that allows concurrent execution of two or more parts of a **program** for maximum utilization of CPU.

Each part of such **program** is called a **thread**.

So, **threads** are light-weight processes within a process.

We create a class that extends the **java**.lang.**Thread** class

TB-1

Differences between Multi - Threading and Multi- Tasking

Multi-Tasking

- Two kinds of multi-tasking:
  1) process-based multi-tasking
  2) thread-based multi-tasking

- Process-based multi-tasking is about allowing several programs to execute concurrently, e.g. Java compiler and a text editor.

- Processes are heavyweight tasks:
  1) that require their own address space
  2) inter-process communication is expensive and limited
  3) context-switching from one process to another is expensive and limited

PROCESS

- Definition – executable program loaded in memory
- Has own address space : Variables and data structures (in memory)
- Each process may execute a different program
- Communicate via operating system, files, network
- May contain multiple threads

THREAD

- Definition – sequentially executed stream of instructions
- Shares address space with other threads
- Has own execution context : Program counter, call stack (local variables)
- Communicate via shared access to data
- Multiple threads in process execute same program
- Also, known as "*lightweight process*"

Thread-Based Multi-Tasking

- Thread-based multi-tasking is about a single program executing concurrently
  - several tasks e.g. a text editor printing and spell-checking text.
- Threads are lightweight tasks

  1) they share the same address space
  2) they cooperatively share the same process
  3) inter-thread communication is inexpensive
  4) context-switching from one thread to another is low-cost
- Java multi-tasking is thread-based

Reasons for Multi-Threading

- Multi-threading enables to write efficient programs that make the maximum use of the CPU, keeping the idle time to a minimum.

- There is plenty of idle time for interactive, networked applications:

  1) the transmission rate of data over a network is much slower than the rate at which the computer can process it

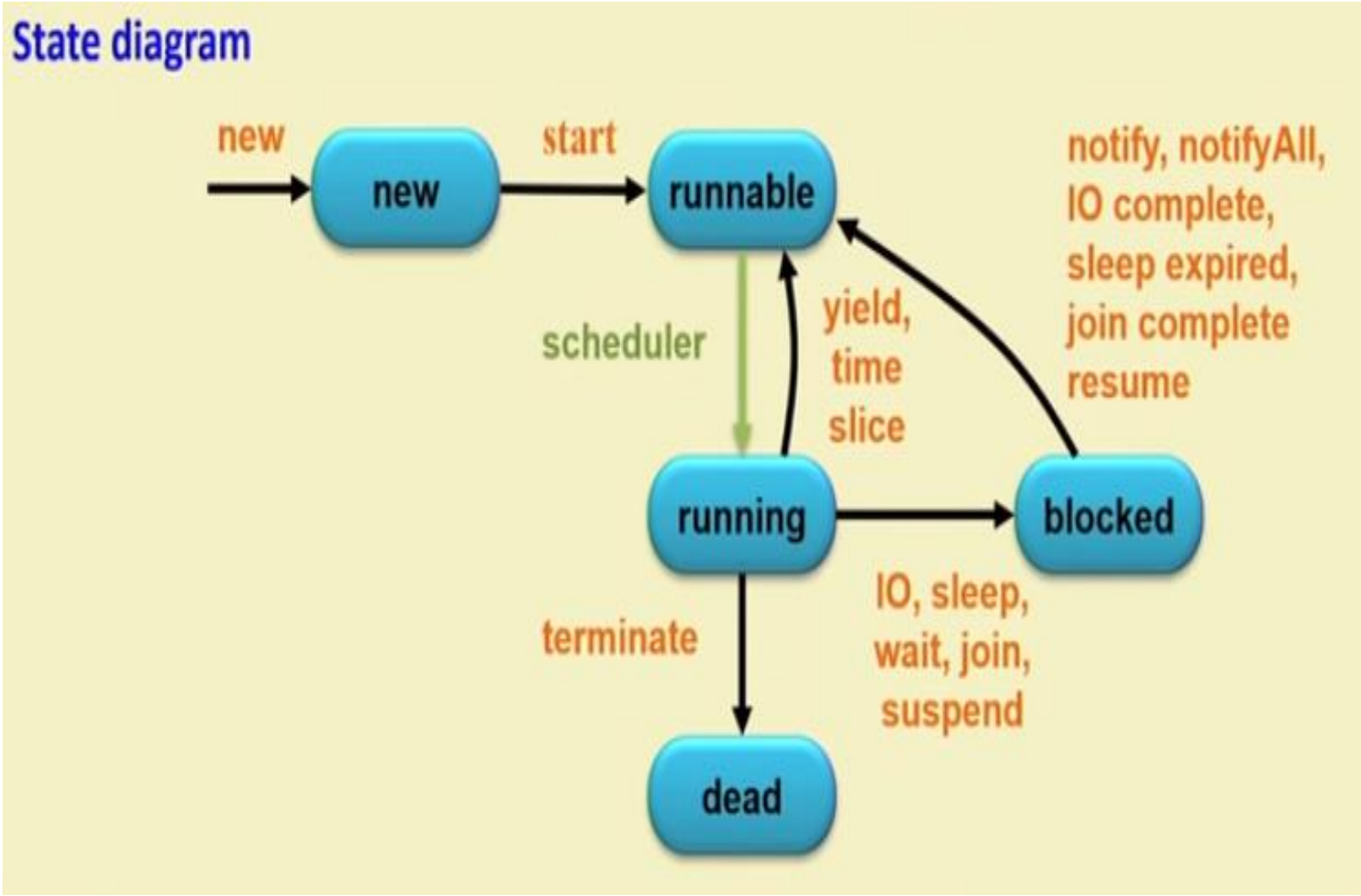
  2) local file system resources can be read and written at a much slower rate than can be processed by the CPU

  3) user input is much slower than the computer

Thread Model / Thread Life-Cycle

- Thread exist in several states
  1) ready to run
  2) running
  3) a running thread can be suspended
  4) a suspended thread can be resumed
  5) a thread can be blocked when waiting for a resource
  6) a thread can be terminated
- Once terminated, a thread cannot be resumed.

Threads exist in several states. Here is a general description.
A thread can be *running*.
It 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.
Once terminated, a thread cannot be resumed.


State diagram
new
new
start
runnable
notify, notifyAll,
IO complete,
sleep expired,
join complete
resume
scheduler
yield,
time
slice
running
blocked
IO, sleep,
wait, join,
suspend
terminate
dead

- **New state –** After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.
- **Runnable (Ready-to-run) state –** A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.
- **Running state –** A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.
- **Dead state –** A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.
- **Blocked -** A thread can enter in this state because of waiting the resources that are hold by another thread.

Thread Control methods

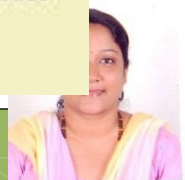
start () :→ A newborn thread with this method enter into Runnable state and Java run time create a system thread context and starts it running. This method for a thread object can be called once only

suspend() :→ This method is different from stop( ) method. It takes the thread and causes it to stop running and later on can be restored (by resume())

resume() :→ This method is used to revive a suspended thread. There is no gurantee that the thread will start running right way, since there might be a higher priority thread running already, but, resume() causes the thread to become eligible for running

sleep(int n):→ This method causes the run time to put the current thread to sleep for n milliseconds

yield() :→ This method causes the run time to switch the context from the current thread to the next available runnable thread. This is one way to ensure that the threads at lower priority do not get started



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 higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running.

Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a ***context switch***.

The rules that determine when a context switch takes place are simple:

- *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 thread that is ready to run is given the CPU.

- *A thread can be pre-empted by a higher-priority thread.*

In this case, a lower-priority thread that does not yield the processor is simply pre-empted—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 *pre-emptive multitasking*.

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated.
For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

TB-1

Synchronization

Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it.

For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other.

That is, you must prevent one thread from writing data while another thread is in the middle of reading it.

For this purpose, Java implements an elegant twist on an age-old model of inter-process synchronization: the *monitor*.

TB-1

The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread.

Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

Most multithreaded systems expose monitors as objects that your program must explicitly acquire and manipulate. 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 methods is called.

Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object.

This enables you to write very clear and concise multithreaded code, because synchronization support is built into the language.

Messaging

After you divide your program into separate threads, you need to define how they will communicate with each other.

When programming with some other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead.

By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have.

Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

TB-1

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.

Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it.

To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

TB-1

The **Thread** class defines several methods that help manage threads.

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.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the ***main thread*** of your program, because it is the one that is executed when your program begins. The main 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.

TB-1

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object.

To do so, you 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. Once you have a reference to the main thread, you can control it just like any other thread.

Controlling 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");
}
}
}
```


In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread( )**, and this reference is stored in the local variable **t**.

Next, the program displays information about the thread. The program then calls
setName( ) to change the internal name of the thread.

Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line.

The pause is accomplished by the **sleep( )** method. The argument to **sleep( )** specifies the delay period in milliseconds. Notice the **try/catch** block around this loop.

The **sleep( )** method in **Thread** might throw an **InterruptedException**. This would happen if some other thread wanted to interrupt this sleeping one.

This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently.

Here is the output generated by this program:

Current thread: Thread[main,5,main]

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

Creating Threads

- To create a new thread, a program will
  1) implement the **Runnable interface**
  or
  2) extend the **Thread class**

- Thread class encapsulates a thread of execution.

- The whole Java multi-threading environment is based on the **Thread class**.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.

Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1.public void run(): is used to perform action for a thread.
2.public void start(): starts the execution of the thread.JVM calls the run() method on the thread.
3.public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4.public void join(): waits for a thread to die.
5.public void join(long miliseconds): waits for a thread to die for the specified miliseconds.
6.public int getPriority(): returns the priority of the thread.
7.public int setPriority(int priority): changes the priority of the thread.
8.public String getName(): returns the name of the thread.
9.public void setName(String name): changes the name of the thread.
10.public Thread currentThread(): returns the reference of currently executing thread.
11.public int getId(): returns the id of the thread.
12.public Thread.State getState(): returns the state of the thread.
13.public boolean isAlive(): tests if the thread is alive.
14.public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.
15.public void suspend(): is used to suspend the thread(depricated).
16.public void resume(): is used to resume the suspended thread(depricated).
17.public void stop(): is used to stop the thread(depricated).
18.public boolean isDaemon(): tests if the thread is a daemon thread.
19.public void setDaemon(boolean b): marks the thread as daemon or user thread.
20.public void interrupt(): interrupts the thread.
21.public boolean isInterrupted(): tests if the thread has been interrupted.
22.public static boolean interrupted(): tests if the current thread has been interrupted.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1.public void run(): is used to perform action for a thread.

The **run()** method of thread class is called if the thread was constructed using a separate Runnable object otherwise this method does nothing and returns. When the run() method calls, the code specified in the run() method is executed. You can call the run() method multiple times.

The run() method can be called using the start() method or by calling the run() method itself. But when you use run() method for calling itself, it creates problems.

Return
It does not **return** any value.

Starting a thread:

•**start() method** of Thread class is used to start a newly created thread. It performs following tasks:

•A new thread starts(with new callstack).
•The thread moves from New state to the Runnable state.
•When the thread gets a chance to execute, its target run() method will run.

call the run() method using the start()method

```
public class RunExp1 implements Runnable
{
    public void run()
    {
        System.out.println("Thread is running...");
    }
    public static void main(String args[])
    {
        RunExp1 r1=new RunExp1();
        Thread t1 =new Thread(r1);
        // this will call run() method
        t1.start();
    }
}
```

running...

call the run() method using the run() method itself

```
public class RunExp2 extends Thread
{
    public void run()
    {
        System.out.println("running...");
    }
    public static void main(String args[])
    {
        RunExp2 t1=new RunExp2 ();
        // It does not start a separate call stack

        t1.run();
    }
}
```

call the run() method more than one time

```
public class RunExp3 extends Thread
{
   public void run()
   {
      for(int i=1;i<6;i++)
      {
         try
         {
            Thread.sleep(500);
         }catch(InterruptedException e){System.out.println(e);}
      System.out.println(i);
      }
   }
   public static void main(String args[])
   {
      RunExp3 t1=new RunExp3();
      RunExp3 t2=new RunExp3();
      t1.run();
      t2.run();
   }
}
```

there is no content switching because here t1 and t2 is treated as a normal object not thread object.

1) Java Thread Example by extending Thread class

```
class MultithreadingDemo extends Thread{
  public void run(){
    System.out.println("My thread is in running state.");
  }
  public static void main(String args[]){
    MultithreadingDemo obj=new MultithreadingDemo();
    obj.start();
  }
}
```

2) Java Thread Example by implementing Runnable interface

```
class MultiThreadRun implements Runnable{
  public void run(){
    System.out.println("My thread is in running state.");
  }
  public static void main(String args[]){
     MultiThreadRun obj=new MultiThreadRun();
     Thread tobj =new Thread(obj);
     tobj.start();
 }
}
```

NOTE

If you are not extending the Thread class, your class object would not be treated as a thread object.

So you need to explicitly create Thread class object.

We are passing the object of your class that implements Runnable so that your class run() method may execute.

Thread Methods

- Start: a thread by calling start its run method
- Sleep: suspend a thread for a period of time
- Run: entry-point for a thread
- Join: wait for a thread to terminate
- isAlive: determine if a thread is still running
- getPriority: obtain a thread's priority
- getName: obtain a thread's name

New Thread: **Runnable interface**

- To create a new thread by implementing the **Runnable interface**

1) create a class that implements the **run method**
public void run()

(Inside **run()**, define the code that constitutes the new thread. It is important to understand that **run()** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run()** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run()** returns**)**

2) instantiate a Thread object within that class, a possible constructor is

Thread(Runnable threadOb, String threadName)

(In this constructor, *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***)**

3) call the start method on this object

void start()

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

Example: New Thread 1

- A class NewThread that implements Runnable

```
class NewThread implements Runnable {
Thread t;
//Creating and starting a new thread. Passing this to the
// Thread constructor – the new thread will call this
// object's run method:
NewThread() {
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start();
}
```

Example: New Thread 2

```
//This is the entry point for the newly created thread – a five-
    iterations loop
//with a half-second pause between the iterations all within
    try/catch:
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.");
}
}
```

Example: New Thread 3

```
class ThreadDemo {
public static void main(String args[]) {
//A new thread is created as an object of
// NewThread:
new NewThread();
//After calling the NewThread start method,
// control returns here.
```

Example: New Thread 4

```
//Both threads (new and main) continue concurrently.
//Here is the loop for the main 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.");
}
}
```

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement:

t = new Thread(this, "Demo Thread");

Passing **this** as the first argument indicates that you want the new thread to call the **run( )** method on **this** object.

Next, **start( )** is called, which starts the thread of execution beginning at the **run( )** method. This causes the child thread's **for** loop to begin.

After calling **start( )**, **NewThread**'s constructor returns to **main( )**. When the main thread resumes, it enters its **for** loop. Both threads continue running, sharing the CPU in single core systems, until their loops finish. The output produced by this program is as follows.

(Your output may vary based upon the specific execution environment.)
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.

As mentioned earlier, in a multithreaded program, often the main thread must be the last thread to finish running.

In fact, for some older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may "hang."

The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds.

This causes the child thread to terminate earlier than the main thread.

Shortly, you will see a better way to wait for a thread to finish.

TB-1

New Thread: Extend Thread

- The second way to create a new thread:
  1) create a new class that extends Thread
  2) create an instance of that class
- Thread provides both run and start methods:
  1) the extending class must override run
  2) it must also call the start method

Example: New Thread 1

- The new thread class extends Thread:

```
class NewThread extends Thread {
   //Create a new thread by calling the Thread's
   // constructor and start method:

NewThread() {
super("Demo Thread");
System.out.println("Child thread: " + this);
start();
}
```


Example: New Thread 2

NewThread overrides the Thread's run method:

```
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.");
}
}
```

Example: New Thread 3

```
class ExtendThread {
public static void main(String args[]) {
//After a new thread is created:
new NewThread();
//the new and main threads continue
//concurrently…
```

Example: New Thread 4

```
//This is the loop of the main 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.");
}
}
```

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**.

Notice the call to **super( )** inside **NewThread**. This invokes the following form of the **Thread** constructor:

public Thread(String *threadName*)

Here, *threadName* specifies the name of the thread.

Choosing an Approach

At this point, you might be wondering why Java has two ways to create child threads, and which approach is better. The answers to these questions turn on the same point.

- The **Thread** class defines several methods that can be overridden by a derived class.
  Of these methods, the only one that *must* be overridden is **run( )**.
  This is, of course, the same method required when you implement **Runnable**.
- Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way.
  So, if you will not be overriding any of **Thread**'s other methods, it is probably best simply to implement **Runnable**.
  Also, by implementing **Runnable**, your thread class does not need to inherit **Thread**, making it free to inherit a different class.

Ultimately, which approach to use is up to you. However, throughout the rest of this chapter, we will create threads by using classes that implement **Runnable**.

Creating Multiple Threads

For example, the following program creates three child 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.");
}
}
```


ample output from this program

```
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.
```

As you can see, once started, all three child threads share the CPU. Notice the call to sleep(10000) in main( ). This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Using isAlive() and join()

As mentioned, often you will want the main thread to finish last. In the preceding examples, this is accomplished by calling **sleep( )** within **main( )**, with a long enough delay to ensure that all child threads terminate prior to the main thread.

How can one thread know when another thread has ended?

Fortunately, **Thread** provides a means by which you can answer this question.

Using isAlive() and join()

Two ways exist to determine whether a thread has finished.

- First, you can call **isAlive( )** on the thread.
  - This method is defined by **Thread**, and its general form is shown here,

    final boolean isAlive( )
  - The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.
- While **isAlive( )** is occasionally useful, the method that you will more commonly use to **wait for a thread to finish** is called **join( )**, shown here

  final void join( ) throws InterruptedException
  - This method waits until the thread on which it is called terminates
  - Its name comes from the concept of the calling thread waiting until the specified thread *joins* it.
- Additional forms of **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

uses **join()** to ensure that the main thread is the last to stop. It also demonstrates the **isAlive( )** method.

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


Sample output from this program

- New thread: Thread[One,5,main]
- New thread: Thread[Two,5,main]
- New thread: Thread[Three,5,main]
- Thread One is alive: true
- Thread Two is alive: true
- Thread Three is alive: true
- Waiting for threads to finish.
- One: 5
- Two: 5
- Three: 5
- One: 4
- Two: 4
- Three: 4
- One: 3
- Two: 3
- Three: 3
- One: 2
- Two: 2
- Three: 2
- One: 1
- Two: 1
- Three: 1
- Two exiting.
- Three exiting.
- One exiting.
- Thread One is alive: false
- Thread Two is alive: false
- Thread Three is alive: false
- Main thread exiting.
- As you can see, after the calls to **join( )** return, the threads have stopped executing.

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- In theory, higher-priority threads get more CPU time than lower-priority threads.
- In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.)
- A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.
- In theory, threads of equal priority should get equal access to the CPU.

To set a thread's priority,

- use the **setPriority( )** method, which is a member of **Thread**.
- This is its general form:

  final void setPriority(int *level*)

Here, *level* specifies the new priority setting for the calling thread.

- The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5.

These priorities are defined as **static final** variables within **Thread**.

- You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**, shown here:

  final int getPriority( )

TB-1

Threads: 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 thread at a time. The process by which this is achieved is called *synchronization*.
- Java provides unique, language-level support for it.
- Key to synchronization is the concept of the monitor.
- A *monitor* is an object that is used as a mutually exclusive lock. Only one thread can *own* a monitor at a given time.
- When a thread acquires a lock, it is said to have *entered* the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor.
- A thread that owns a monitor can re-enter the same monitor if it so desires.

TBF

Threads: Synchronization

- Multi-threading introduces asynchronous behavior to a program.

- How to ensure synchronous behavior when we need it?
- For instance, how to prevent two threads from simultaneously writing and reading the same object?

- Java implementation of monitors:

  1) classes can define so-called synchronized methods

  2) each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called.

  3) once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object.

TB-1

Using Synchronized Methods

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.
- To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.
- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

TB-1

Java synchronized method

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

TB-1

Synchronization in Java

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization

The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

- Process Synchronization
- Thread Synchronization

TB-1

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

Mutual Exclusive

Synchronized method.(Table pgm)

Synchronized block.(Tb1-pgm)

Synchronized block is used to lock an object for any shared resource.

Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

synchronized (object reference expression) {

//code block }

static synchronization.

Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

by synchronized method

by synchronized block

by static synchronization

TB-1

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor.

Every object has an lock associated with it.

By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

TB-1

Thread Synchronization

- Language keyword: `synchronized`
- Takes out a monitor lock on an object
  - Exclusive lock for that thread
- If lock is currently unavailable, thread will block

TB-1

Thread Synchronization

- Protects access to code, not to data
  - Make data members private
  - Synchronize accessor methods
- Puts a "force field" around the locked object so no other threads can enter
  - Actually, it only blocks access to other synchronizing threads

TB-1

Syntax

The general form of the **synchronized** statement:

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

Here, *objRef* is a reference to the object being synchronized.

A synchronized block ensures that a call to a synchronized method that is a member of *objRef*'s class occurs only after the current thread has successfully entered *objRef*'s monitor.

TB-1

Inter-Thread communication

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

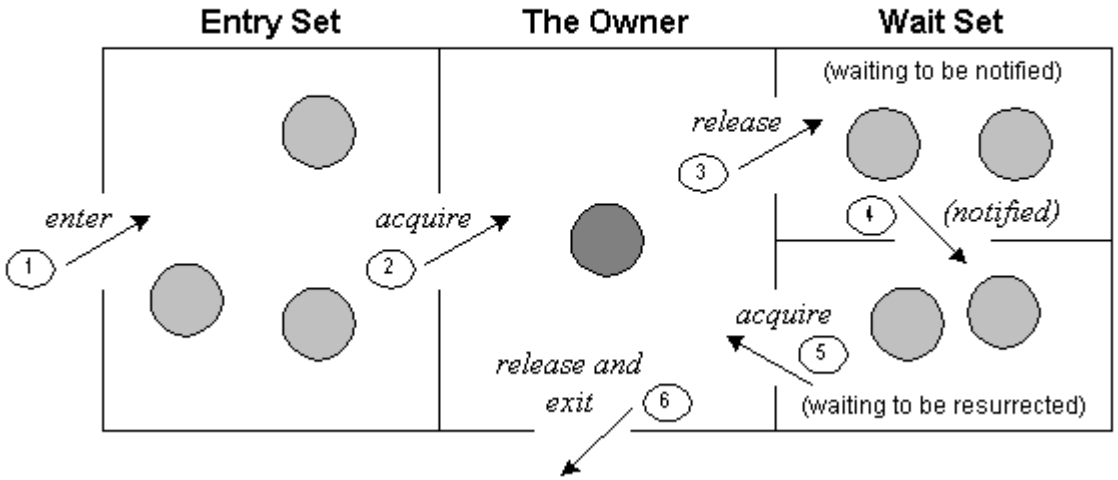
It is implemented by following methods of **Object class**:

wait()
notify()
notifyAll()

It is because they are related to lock and object has a lock.

TB-1

Understanding the process of inter-thread communication



1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	waits until object is notified.
public final void wait(long timeout)throws InterruptedException	waits for the specified amount of time.

Difference between wait and sleep?

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

2) notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

public final void notify()

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

Syntax:

public final void notifyAll()

Using Synchronized Method

The following program(Synch1.java) has three simple classes.
The first one, **Callme**, has a single method named **call( )**. The **call( )** method takes a **String** parameter called **msg**.

This method tries to print the **msg** string inside of square brackets. The interesting thing to notice is that after **call( )** prints the opening bracket and the **msg** string, it calls **Thread.sleep(1000)**, which pauses the current thread for one second.

The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this object's **run( )** method. The thread is started immediately.

The **run( )** method of **Caller** calls the **call( )** method on the **target** instance of **Callme**, passing in the **msg** string.

Finally, the **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.

TB-1

Producer-Consumer Problem in Java Concurrency-pattern problem/Bounded-buffer problem

The producer-consumer problem is a classic example of a multi-process synchronization problem

Problem Statement

There are two processes, a producer and a consumer, that share a common buffer with a limited size. The producer "produces" data and stores it in the buffer, and the consumer "consumes" the data, removing it from the buffer. Having two processes that run in parallel, we need to make sure that the producer will not put new data in the buffer when the buffer is full and the consumer won't try to remove data from the buffer if the buffer is empty.

Solution

For solving this concurrency problem, the producer and the consumer will have to communicate with each other. If the buffer is full, the producer will go to sleep and will wait to be notified. After the consumer will remove some data from the buffer, it will notify the producer, and then, the producer will start refilling the buffer again. The same process will happen if the buffer is empty, but in this case, the consumer will wait to be notified by the producer.

If this communication is not done properly, it can lead to a deadlock where both processes will wait for each other.

It consists of four classes: **Q**, the queue that you're trying to synchronize; **Producer**, the threaded object that is producing queue entries; **Consumer**, the threaded object that is consuming queue entries; and **PC**, the tiny class that creates the single **Q**, **Producer**, and **Consumer**.

https://codepumpkin.com/producer-consumer-design-pattern-1/

Inside **get( )**, **wait( )** is called.

This causes its execution to suspend until **Producer** notifies you that some data is ready. When this happens, execution inside **get( )** resumes.

After the data has been obtained, **get( )** calls **notify( )**. This tells **Producer** that it is okay to put more data in the queue.

Inside **put( )**, **wait( )** suspends execution until **Consumer** has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and **notify( )** is called.

TB-1

This tells **Consumer** that it should now remove it.

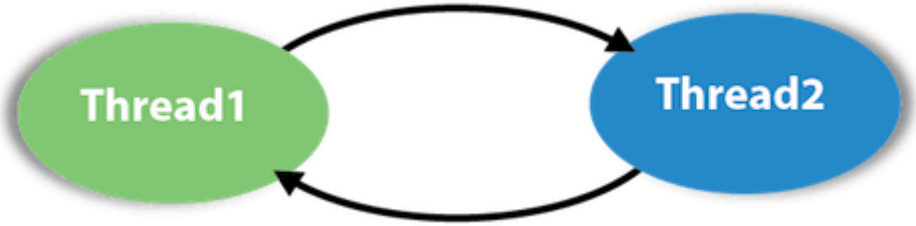
Deadlocks

Deadlock in java

Deadlock in java is a part of multithreading.

Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread.

Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



TB-1

Daemon Threads

- Any Java thread can be a *daemon* thread.
- Daemon threads are service providers for other threads running in the same process as the daemon thread.
- The run() method for a daemon thread is typically an infinite loop that waits for a service request. When the only remaining threads in a process are daemon threads, the interpreter exits. This makes sense because when only daemon threads remain, there is no other thread for which a daemon thread can provide a service.
- To specify that a thread is a daemon thread, call the setDaemon method with the argument true. To determine if a thread is a daemon thread, use the accessor method isDaemon.

Thread Groups

- Every Java thread is a member of a *thread group*.
- Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.
- For example, you can start or suspend all the threads within a group with a single method call.
- Java thread groups are implemented by the "ThreadGroup" class in the java.lang package.
- The runtime system puts a thread into a thread group during thread construction.
- When you create a thread, you can either allow the runtime system to put the new thread in some reasonable default group or you can explicitly set the new thread's group.
- The thread is a permanent member of whatever thread group it joins upon its creation--you cannot move a thread to a new group after the thread has been created

TB-1

The ThreadGroup Class

- The "ThreadGroup" class manages groups of threads for Java applications.
- A ThreadGroup can contain any number of threads.
- The threads in a group are generally related in some way, such as who created them, what function they perform, or when they should be started and stopped.
- ThreadGroups can contain not only threads but also other ThreadGroups.
- The top-most thread group in a Java application is the thread group named main.
- You can create threads and thread groups in the main group.
- You can also create threads and thread groups in subgroups of main.

TB-1

Creating a Thread Explicitly in a Group

- A thread is a permanent member of whatever thread group it joins when its created--you cannot move a thread to a new group after the thread has been created. Thus, if you wish to put your new thread in a thread group other than the default, you must specify the thread group explicitly when you create the thread.
- The Thread class has three constructors that let you set a new thread's group:

  public Thread(ThreadGroup *group*, Runnable *target*)

  public Thread(ThreadGroup *group*, String *name*)

  public Thread(ThreadGroup *group*, Runnable *target*, String *name*)
- Each of these constructors creates a new thread, initializes it based on the Runnable and String parameters, and makes the new thread a member of the specified group.

  For example:

```
ThreadGroup myThreadGroup = new ThreadGroup("My Group of Threads");
Thread myThread = new Thread(myThreadGroup, "a thread for my group");
```

TB-1

Thank You