

Power Programming with Java

Session: 7

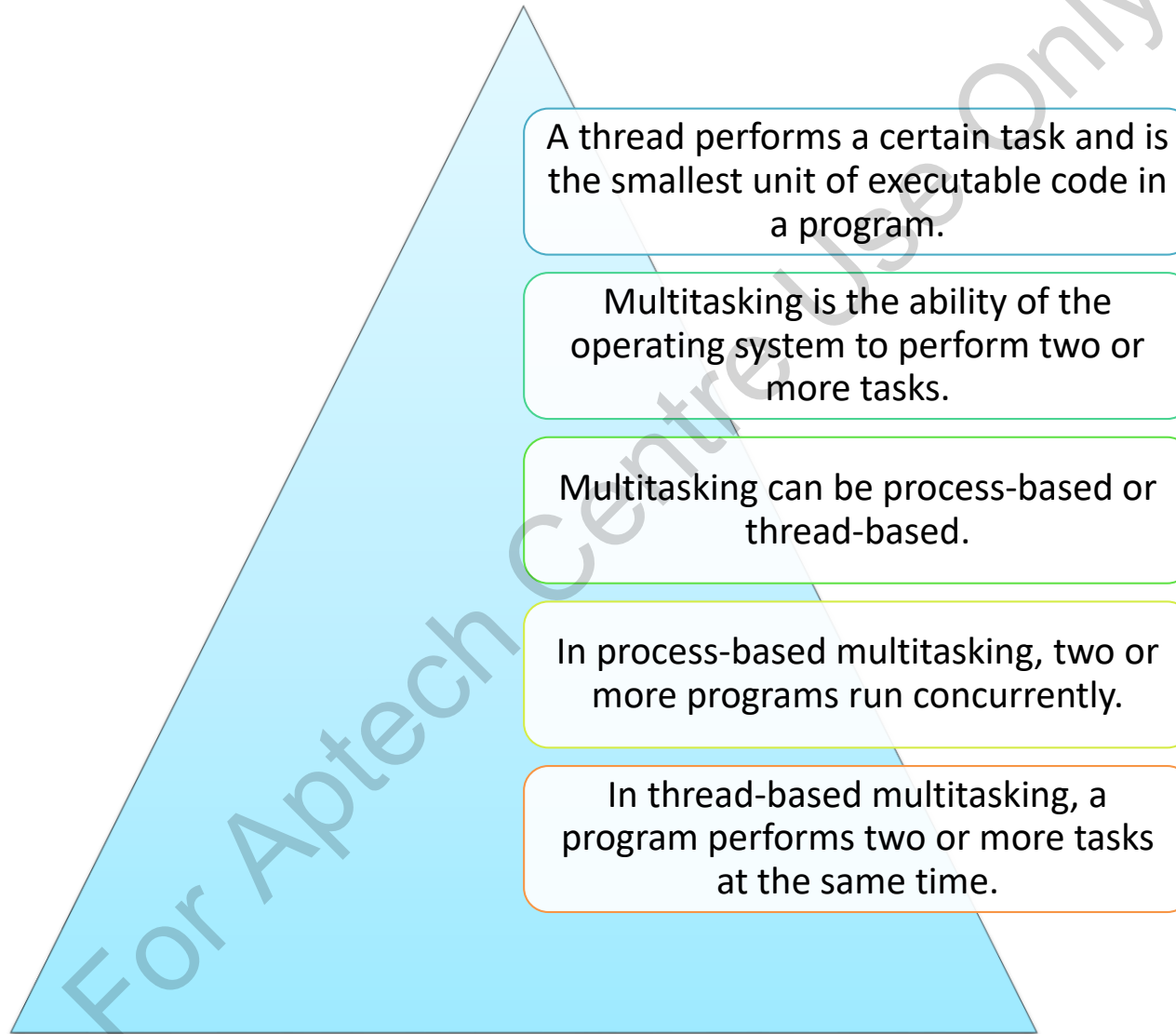
Multithreading and Concurrency





- ◆ Define multithreading
- ◆ Differentiate between multithreading and multitasking
- ◆ Explain the use of `isAlive()` and `join()` method
- ◆ Explain race conditions and ways to overcome them
- ◆ Describe intrinsic lock and synchronization
- ◆ Describe atomic access
- ◆ Describe the use of `wait()` and `notify()` methods
- ◆ Define deadlock and the ways to overcome deadlock
- ◆ Explain `java.util.concurrent` collections

For Aptech Centre Use Only





There are basically two types of multitasking in use among operating systems:

Preemptive

- In this case, the operating system controls multitasking by assigning the CPU time to each running program.

Cooperative

- In this approach, related applications voluntarily surrender their time to one another.

Multithreading is a technique similar to multitasking and involves the creation of one or more threads within a program to enable number of tasks to run concurrently or in parallel.

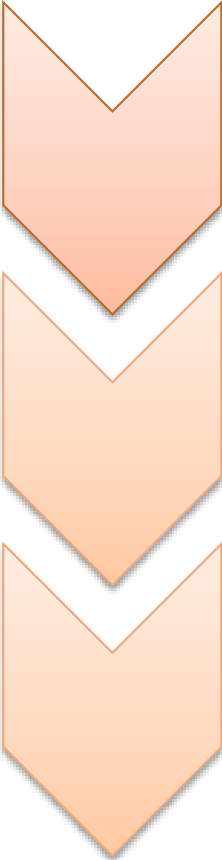


Differences Between Multithreading and Multitasking:

Multithreading	Multitasking
In a multithreaded program, two or more threads can run concurrently.	In a multitasking environment, two or more processes run concurrently.
Multithreading requires less overhead. In case of multithreading, threads are lightweight processes. Threads can share same address space and inter-thread communication is less expensive than inter-process communication.	Multitasking requires more overhead. Processes are heavyweight tasks that require their own address space. Inter-process communication is very expensive and the context switching from one process to another is costly.



Multithreading is essential for following reasons:

- 
- Multithreading increases performance of single-processor systems, as it reduces the CPU idle time.
 - Multithreading encourages faster execution of a program when compared to an application with multiple processes, as threads share the same data whereas processes have their own sets of data.
 - Multithreading introduces the concept of parallel processing of multiple threads in an application which services a huge number of users.

isAlive() Method



A thread is considered to be alive when it is running.

If the thread is alive, then the boolean value true is returned.
If the `isAlive()` method returns false, it is understood that the thread is either in new state or in terminated state.

Syntax: `public final boolean isAlive()`

Code Snippet

```
public class ThreadDemo extends Thread {  
    public static void main(String args[]) {  
        ThreadDemo Obj = new ThreadDemo();  
        Thread t = new Thread(Obj);  
        System.out.println("The thread is alive :" + t.isAlive());  
    }  
}
```



- ◆ `join()` method causes the current thread to wait until the thread on which it is called terminates.
- ◆ `join()` method performs the following operations:
 - ◆ This method allows specifying the maximum amount of time that the program should wait for the particular thread to terminate.
 - ◆ It throws `InterruptedException` if another thread interrupts it.
 - ◆ The calling thread waits until the specified thread terminates.
- ◆ `join()` method of the `Thread` class has two other overloaded versions:
- ◆ `void join(long timeout):`
 - ◆ In this type of `join()` method, an argument of type `long` is passed.
 - ◆ The amount of timeout is in milliseconds.
 - ◆ This forces the thread to wait for the completion of the specified thread until the given number of milliseconds elapses.
- ◆ `void join(long timeout, int nanoseconds):`
 - ◆ In this type of `join()` method arguments of type `long` and `integer` are passed.
 - ◆ The amount of timeout is given in milliseconds in addition to a specified amount of nanoseconds.
 - ◆ This forces the thread to wait for the completion of the specified thread until the given timeout elapses.




- ◆ In multithreaded programs, several threads may simultaneously try to update the same resource, such as a file.
- ◆ This leaves the resource in an undefined or inconsistent state. This is called race condition.

Race Condition

- ◆ In general, race conditions in a program occur when:



- Two or more threads share the same data between them.



- Two or more threads try to read and write the shared data simultaneously.



- ◆ The synchronized block contains code qualified by the `synchronized` keyword.
- ◆ A lock is assigned to the object qualified by `synchronized` keyword.
- ◆ A lock allows only one thread at a time to access the code. When a thread starts to execute a synchronized block, it grabs the lock on it.
- ◆ Any other thread will not be able to execute the code until the first thread has finished and released the lock. The lock is based on the object and not on the method.

Syntax

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

where,

object is the reference of the object being synchronized.



The synchronized method obtains a lock on the class object.

This means that at a single point of time only one thread obtains a lock on the method, while all other threads have to wait to invoke the synchronized method.

Syntax

```
synchronized method(...)  
{  
    // body of method  
}
```



Synchronization is built around the concept of an in-built monitor, which is also referred to as intrinsic lock or monitor lock.

The monitor lock enforces exclusive access to the thread objects and creates a relationship between the thread action and any further access of the same lock.

Every object is connected to an intrinsic lock.

Typically, a thread acquires the object's intrinsic lock before accessing its fields, and then releases the intrinsic lock.

In this span of acquiring and releasing the intrinsic lock, the thread owns the intrinsic lock.

No other thread can acquire the same lock.

The other thread will block when it attempts to acquire the lock.

When a thread invokes a synchronized method, the following occurs:

- It automatically acquires the intrinsic lock for that method's object.
- It releases when the method returns.



This occurs when a thread acquires a lock that it already owns.

In this event, the synchronized code invokes a method directly or indirectly that also contains synchronized code.

Both sets of code use the same lock.

With reentrant synchronization, it is easy for synchronized code to avoid a thread cause itself to block.



In programming, an atomic action occurs all at once.

Following are the features of an atomic action:

- It occurs completely or it does not occur at all.
- Effects of an atomic action are visible only after the action is complete.

Following are the actions that can be specified as atomic:

- Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
- Reads and writes are atomic for all variables declared volatile.

Atomic actions cannot be interleaved.

There will no thread interference when atomic actions are used.



- ◆ The wait-notify mechanism acts as the traffic signal system in the program.
- ◆ It allows the specific thread to wait for some time for other running thread and wakes it up when it is required to do so.
- ◆ In other words, the wait-notify mechanism is a process used to manipulate the `wait()` and `notify()` methods.
- ◆ This mechanism ensures that there is a smooth transition of a particular resource between two competitive threads.
- ◆ It also oversees the condition in a program where one thread is:

Allowed to wait for the lock of a synchronized block of resource currently used by another thread.

Notified to end its waiting state and get the lock of that synchronized block of resource.



- ◆ The `wait()` method causes a thread to wait for some other thread to release a resource.
- ◆ It forces the currently running thread to release the lock or monitor, which it is holding on an object.
- ◆ Once the resource is released, another thread can get the lock and start running.
- ◆ The `wait()` method can only be invoked only from within the synchronized code.
- ◆ Following points should be remembered while using the `wait()` method:

- The calling thread gives up the CPU and lock.
- The calling thread goes into the waiting state of monitor.



- ◆ The `notify()` method alerts the thread that is waiting for a monitor of an object.
- ◆ This method can be invoked only within a synchronized block.
- ◆ If several threads are waiting for a specific object, one of them is selected to get the object.
- ◆ The scheduler decides this based on the requirement of the program.
- ◆ `notify()` method functions in the following ways:

The waiting thread moves out of the waiting space of the monitor and into the ready state.

The thread that was notified is now eligible to get back the monitor's lock before it can continue.



Deadlock describes a situation where two or more threads are blocked forever, waiting for the others to release a resource.

At times, it happens that two threads are locked to their respective resources, waiting for the corresponding locks to interchange the resources between them.

In that situation, the waiting state continues forever as both are in a state of confusion as to which one will leave the lock and one will get into it.

This is the deadlock situation in a thread based Java program.

The deadlock situation brings the execution of the program to a halt.

For Apteck Centre Use Only



- ◆ Following things in a program can be done to avoid deadlock situations in it:

- Avoid acquiring more than one lock at a time.
- Ensure that in a Java program, you acquire multiple locks in a consistent and defined order.

- ◆ Concurrency in Threads:

- ◆ Java platform has added a rich concurrency library for large applications executed on multiple processors environment.
- ◆ The concurrency library is the new addition in the `java.util` package. It also provides new concurrent data structures in the Collections Framework.



Following are some of these collections that are categorized by the collection interfaces:

- **BlockingQueue**: This defines a FIFO data structure that blocks or times out when data is added to a full queue or retrieved from an empty queue.
- **ConcurrentMap**: This is a subinterface of `java.util.Map` that defines useful atomic operations.
- **ConcurrentNavigableMap**: This is a subinterface of `ConcurrentMap` that supports approximate matches.

For Aptech Centre Use Only



`java.util.concurrent`
`.atomic` package defines
classes that support atomic
operations on single
variables.

All classes include get and
set methods that
functions similar to reads
and writes on volatile
variables.

Therefore, a set has a
happens-before
relationship with any
successive get on the
same variable.

For Aptech Certified Use Only



Objects that separate thread management and creates them from the rest of the application are called executors.

`java.util.concurrent` package defines the following three executor interfaces:

- **Executor**: This helps launch new tasks.
- **ExecutorService**: This is a subinterface of **Executor** and helps manage the development of the executor tasks and individual tasks.
- **ScheduledExecutorService**: This is a subinterface of **ExecutorService** and helps periodic execution of tasks.



Thread pools have worker threads that help create threads and thus minimize the overhead.

Certain executor implementations in `java.util.concurrent` use thread pools.

Thread pools are often used to execute multiple tasks.

Allocating and deallocating multiple thread objects creates a considerable memory management overhead in a large-scale application.

Fixed thread pool is a common type of thread pool that includes the following features:

- There are a specified number of threads running.
- When in use if a thread is terminated, it is automatically replaced with a new thread.
- Applications using fixed thread pool services HTTP requests as quickly as the system sustains.



This is an implementation of the `ExecutorService` interface.

The framework helps work with several processors to boost the performance of an application.

It uses a work-stealing algorithm and is used when work is broken into smaller pieces recursively.

The Fork/Join framework allocates tasks to worker threads in a thread pool.

There is the `ForkJoinPool` class in the fork/join framework.

The class is an extension of the `AbstractExecutorService` class.

The `ForkJoinPool` class implements the main work-stealing algorithm and executes `ForkJoinTask` processes.



Before looking into stack walking and traversing stack frames, it is crucial to understand what is a stack frame.

Each JVM thread has a private JVM stack that is created when a thread is created.

By now you may be aware that a stack is a linear data structure in memory based on Last In First Out (LIFO) principle.

Whenever a method is called, a new stack frame is created and pushed to the top of the stack.

When the method call completes, the stack frame is destroyed (that is, 'popped' from the stack)

A stack trace is a representation of a call stack at a specific point in time, with each element representing a method call.

This is extremely useful in exception handling to identify the root cause of the failure.



- ◆ Multithreading is nothing but running of several threads in a single application.
- ◆ The `isAlive()` method tests whether the thread is in runnable, running, or terminated state.
- ◆ The `join()` method forces a running thread to wait until another thread completes its task.
- ◆ Race condition can be avoided by using synchronized block.
- ◆ The `wait()` method sends the running thread out of the lock or monitor to wait.
- ◆ The `notify()` method instructs a waiting thread to get in to the lock of the object for which it has been waiting.
- ◆ Deadlock describes a situation where two or more threads are blocked forever, waiting for each to release a resource.
- ◆ Java 9 introduced the StackWalker API which provides capabilities to walk through the stack and helps to trace root causes of exceptions.
- ◆ The `StackWalker` class is the main component of StackWalker API.