Welcome

Java Multi-Threading

Prakash Badhe

prakash.badhe@vishwasoft.in

Copyright Notice

Process

In computer terms each of the running programs is a task or process.

Multitasking

- Multitasking or multiprocessing allows user to rapidly switch between different tasks or programs running simultaneously.
- For example you can have the computer to print a document while at the same time recalculate the spreadsheet, download your email and let you work in the word document as well.

Threads

- There is a thread associated With each of these running programs in the operating system.
- Whenever a new program/process is started a new thread is started by the operating system.
- Thread is a single sequential flow of control with in a program.

Thread Scheduler

- The program called '**Thread Scheduler**' in the OS manages the threads and controls the CPU time allocated for execution.

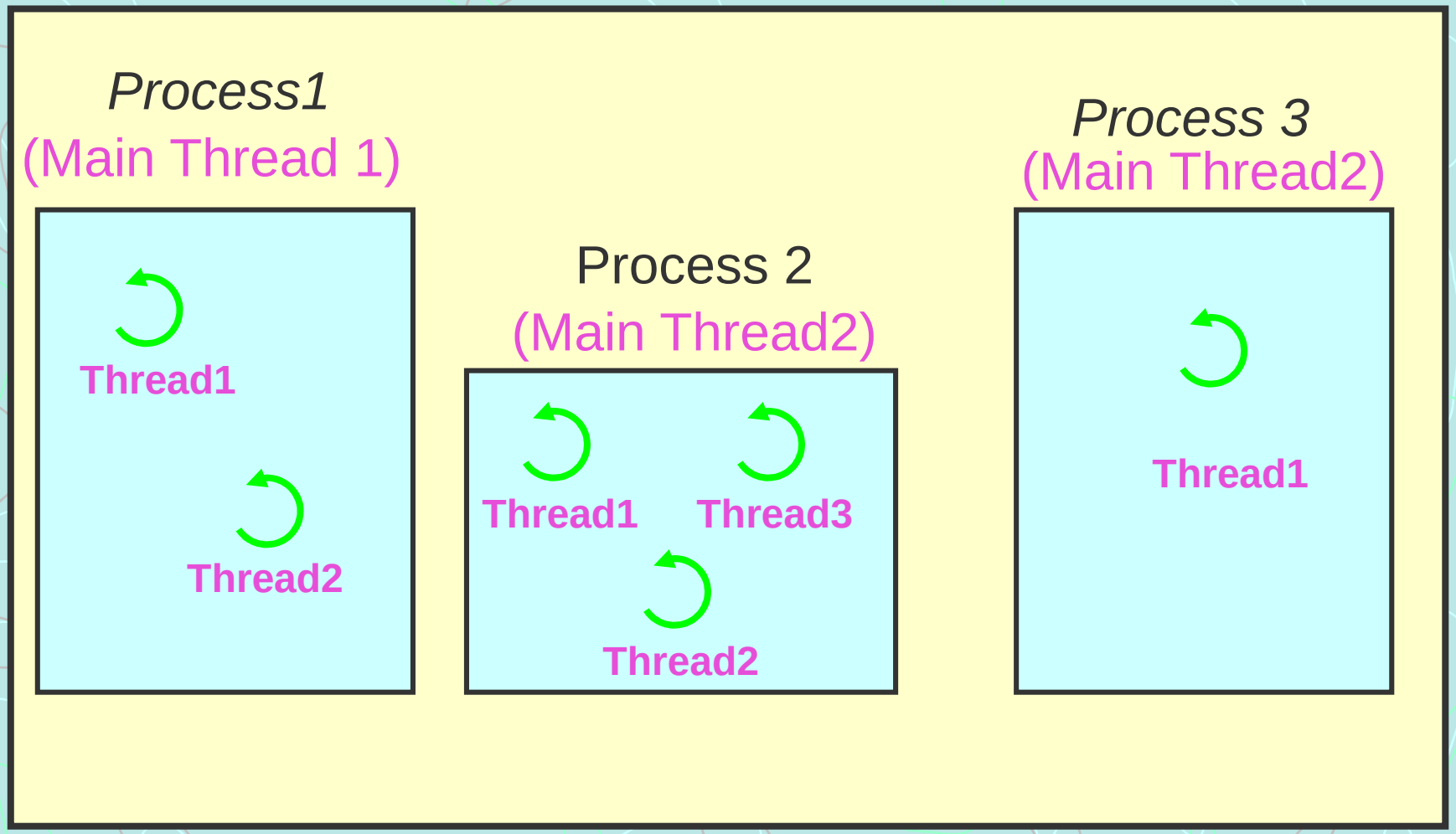
Multi Threading

Multitasking allows user to switch between the tasks.

How you run a single process (program) to perform two different things at the same time.

Multithreading provides the ability to run different (child) threads in the single process.

Runtime Execution



Thread switching

Are these multiple programs executed simultaneously by the OS or CPU ?
Actually the CPU is switched to execute each program code periodically at defined time interval and this switching is so fast that the programs appear to run at the same time.
Each programs logical context is the **Thread.**

Thread Properties

- Each Thread has its own memory and process and parent process allocated.
- Each thread has its priority of execution
- The running threads are controlled by Task Manager and can be forcibly terminated.

Thread priority

Each thread has certain priority of execution.
Depending on this priority the thread may not start to execute or the time allotted for execution will vary.
The thread inherits the priority from the parent process or you can change the priority.
Priorities are defined as minimum, normal and maximum priority.
This priority value that hints the thread scheduler how much it should be cared in case of many threads are running.

Thread.MIN_PRIORITY = 1
Thread.NORM_PRIORITY = 5
Thread.MAX_PRIORITY = 10

Thread states

Ready: The thread is created but not started execution by CPU.

Running: The thread is running.

Waiting: The thread is waiting for certain user /program response.

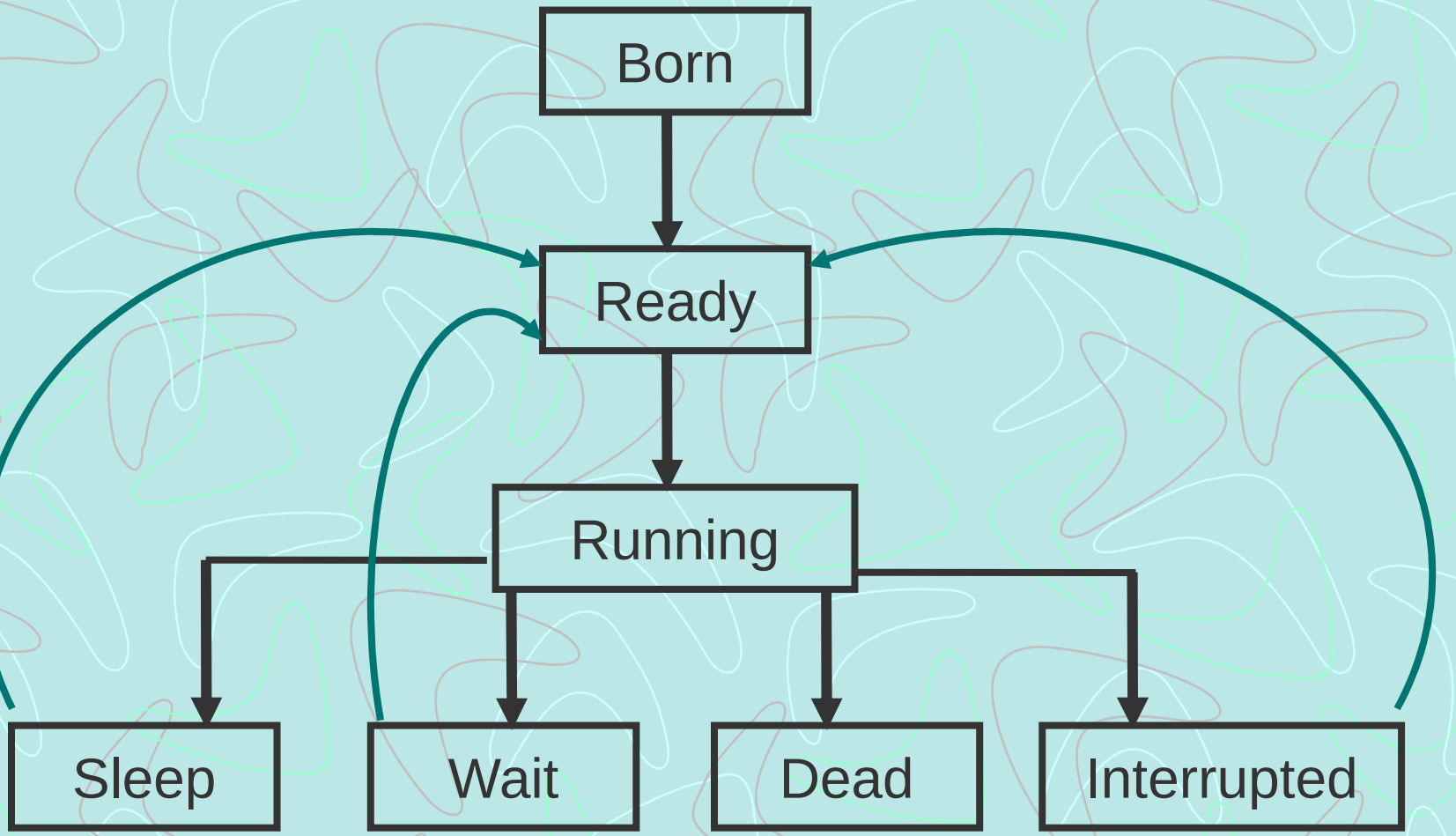
Blocked: The thread is blocked.

Suspended: The thread is suspended temporarily.

Sleeping: The thread is stopped for fixed time

Dead: The thread is terminated

Thread States



Java Threading Model

Java implements multithreading features as part of the language in two ways Thread class and Runnable interface defined in *java.lang* package.

You can create a thread class by extending from Thread class(**override** run( ) method)

OR

By implementing ***Runnable*** interface and provide *implementation* for run( ) method.

Java Threads

Both approaches revolve around ***providing*** a run method, which is where all the execution processing takes place. When a thread is created and initialized, its *run* method is called and given the opportunity to perform whatever processing the thread is designed to run. Because it provides all the processing logic, the ***run*** method is the heart of a thread.

Deriving from Thread Class


```
public class myThread extends Thread
{
 public void run( )
  {
   //Program execution code
  }
}
```

To run this thread

myThread t = new myThread ( );

t.start();

start( ) will call run( ) and starts the thread execution

Implementing Runnable interface


```
public class myRun implements Runnable
{
 public void run( )
  {
   //Program execution code
  }  }
```

To run this Runnable thread pass this runnable object to the Thread constructor

```
Thread  t = new Thread (new myRun ( ));
t.start( );
```

start( ) will call run( ) and starts the thread execution

Thread class methods

- Thread getCurrentThread()
- sleep(int milliseconds)
- String getName()
- setName(String newName)
- int getPriority()
- setPriority(int newPriority)
- boolean isAlive()
- boolean isDaemon()
- join()
- stop()
- suspend()

ThreadGroup

ThreadGroup is a class that groups related threads as a single unit and allows to perform some operations on a group as a whole, rather than with each separate thread, like start, stop etc.

ThreadGroup Methods

- activeCount(): returns an estimate of the number of active threads in the thread group and its subgroups.
- activeGroupCount(): returns an estimate of the number of active groups in the thread group and its subgroups.
- destroy(): destroys the thread group and all of its subgroups.

ThreadGroup Methods..

- enumerate(Thread[] list): copies into the specified array every active thread in this thread group and its subgroups.
- getMaxPriority(): returns the maximum priority of the thread group.
- interrupt(): interrupts all threads in the thread group.
- isDaemon(): tests if the thread group is a daemon thread group.
- setMaxPriority(int priority): sets the maximum priority of the group.

Thread applications

- Running parallel processes
- Graphics UI render and event handling
- Image animation
- Providing concurrent client responses oin server side applications
- Handle multiple database connections.
- Multiple threads can be grouped together in ***'ThreadGroup'*** for common operations on all the threads in the group.

Daemon Thread

- Java defines two types of thread: user thread (normal thread) and daemon thread.
- The default thread created is user thread.
- The Java Virtual Machine (JVM) won't terminate if there are still user threads running.
- But it will exit if there are only daemon threads running.
- The daemon threads are running in background even if jvm terminates.

Daemon Thread Usage

- Daemon threads have lower priority than normal ones, so they are used for running background services that serve user threads.
- An example of daemon thread in the JVM is the garbage collector thread that runs silently in the background to free unused memory.
- To make a thread daemon by calling Thread.**setDaemon**(true) and check daemon status by using **isDaemon()**.
- *The setDaemon() should be called before the thread is started.*

Thread synchronization

- While running simultaneous multiple threads it may happen that more than one thread can access ***same*** resource which may get *undesired* results, like corrupt the file,hang he system etc.
- To avoid this,need to control the access to the
- resource in code execution method in such a
- way that if one thread is accessing a resource (in side method) another thread has to wait for that resource code in the method until the first thread finishes the method code execution, this is thread ***synchronization*** mechanism.

Synchronized operations

- Synchronized or thread safe methods are defined using synchronized keyword on methods of resource i.e.
  public synchronized void getResult(int num)
- The existing resource methods which are not
- Defined as synchronized can be put under
- synchronized block in calling threads.

```
synchronized(myResourceObj)
{
   //non - synchronized  code
}
```

This is Resource synchronization.

Synchronized Access to the Threads…

- When one synchronized method/block is being executed by a thread, the same or the other synchronized method/block on the same object cannot be accessed by another thread till the first running thread finishes/comes out of synchronized block or method

…Under the hood

- When a thread is running a *synchronized* method on a shared object, the thread is said to have acquired the lock on the object.
- If another thread tries to call the *synchronized* method on the same shared object,it will have to wait till the first thread has released the lock on the shared object by finishing the execution/coming out of the synchronized method .

Class Level Lock..?

- **Similar to object level locking by synchronized instance methods, can there be class level lock..** ?

The Locks and synchronization

- **Mutual Exclusion:** Only one thread or process can execute a block of code (critical section) at a time.
- **Visibility**: The changes made by one thread to shared data are visible to other threads.
- Java's synchronized keyword guarantees both mutual exclusion and visibility of shared data.

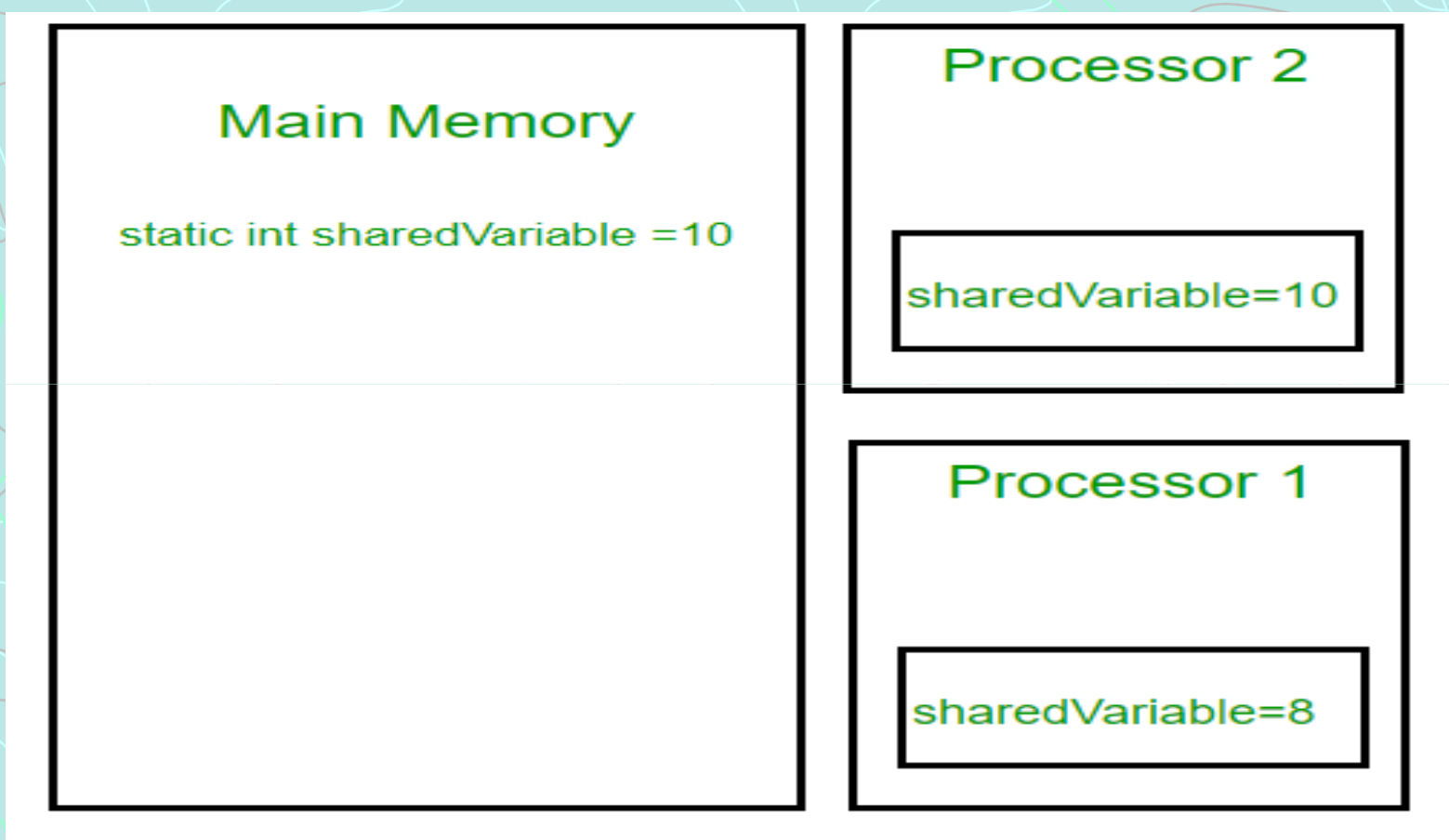
Thread Safe

- The synchronized methods or blocks which controls/limits the simultaneous access across the threads is called as **thread safe** which avoids the resource corruption and undesired results.

Locked Resources by the threads

- Common resource files
- Configuration files
- Log files
- Database connections
- Network sockets
- System resources like console

Shared variables across threads



For the threads running on different processors, value of **sharedVariable** may be different in different threads.

Synchronized as overkill..

- In some cases we may only desire the visibility and not atomicity.
- Use of synchronized in such situation is an overkill and may cause scalability problems.
- Here *volatile* comes to the rescue.
- Volatile variables have the visibility features of synchronized but not the atomicity features.
- The values of volatile variable will never be cached and all writes and reads will be done to and from the main memory.

The volatile variable

- The compiler makes the optimization for volatile variable to always read and write from master copy and not the thread's local copy.
- The volatile variable used to inform the compiler that a particular field is subject to be accessed by multiple threads, which will prevent the compiler from doing any reordering or any kind of optimization which is not desirable in a multi-threaded environment.
- Without volatile variable compiler can re-order the code, free to cache value of volatile variable instead of always reading from main memory.

Volatile vs. Synchronized

- The volatile variable can be used as an alternative way of achieving synchronization
- In some cases, like visibility,with volatile variable, it's guaranteed that all reader thread will see updated value of the volatile variable once write operation completed, without volatile keyword different reader thread may see different values.
- Synchronized method affects performance more than a volatile keyword in Java.

The usage of volatile

- The volatile keyword is only to a variable and using volatile keyword for methods or class.
- volatile keyword in guarantees that value of the volatile variable will always be read from main memory and not from Thread's local cache.
- The changes made to a volatile variable are always visible to other threads.

More with volatile

- Access to a volatile variable in Java never has a chance to block, since only doing a simple read or write, so unlike a synchronized block will never hold on to any lock or wait for any lock.
- If a variable is not shared between multiple threads, you don't need to use volatile keyword with that variable.
- Threads can be blocked for waiting for any monitor in case of synchronized, that is not the case with the volatile keyword .

Thread Safe Collections

- Only two collections are internally defined as thread-safe: Vector and Hashtable.
- These two collections have poor performance in multi-threaded programs.
- The other collections methods need to be called from within synchronized blocks in threads.

Synchronized Wrappers

- The basic collection implementations are not thread-safe in order to provide maximum performance in single-threaded applications.
- We should not use non-thread-safe collections in concurrent context, as it may lead to undesired behaviors and inconsistent results.
- We can use synchronized blocks manually to safeguard the code.
- it's always wise to use thread-safe collections instead of writing synchronization code manually.

Factory Synchronized collections

- Java Collections Framework provides factory methods for creating thread-safe collections. These methods are in the following form:
- **Collections.synchronizedXXX(collection)**
- These factory methods wrap the specified collection and return a thread-safe implementation.
- The supported collections are are Collection, List, Map, Set, SortedMap and Sorted Set implementations.

Synchronized collections

- List<String> safeList = Collections.synchronizedList(new ArrayList<>());
- Map<Integer, String> unsafeMap = new HashMap<>();
- Map<Integer, String> safeMap = Collections.synchronizedMap(unsafeMap);
- In the wrapped collection all the methods are synchronized to provide thread-safety.
- The synchronized collection delegates all work to the wrapped collection inside it.

Sync Hits the Performance

- Synchronization requires locks which always take time to monitor, and that reduces the performance.
- Define the methods as synchronized only when you critically need it.
- It is the threads that have to call non-synchronized methods in synchronized blocks to make it thread-safe.

Inter thread communication

- To make the multiple threads ware about the shared
- synchronized resource access availability,
- communication between different threads is done
- with wait( ) and notify( ) methods defined in the
- super class Object.
- These wait( ) and notify( ) methods are to be called
- only from within the synchronized methods/blocks.

Wait for access..

When a running thread calls wait in a synchronized method/block on a shared object, it releases the lock on the shared object temporarily so that another thread can access this method and calling thread waits indefinitely.

Resource is ready for access : notify

When another thread gets the lock on shared object and starts executing the synchronized method/block and when the execution is done and to release the lock calls notify( )from within a synchronized method/block to indicate to other waiting thread that this synchronized code is available for execution.

The waiting thread starts the synchronized code execution.

Fail-fast iterators with Collections

- Define collection's Iterator to traverse through elements in the collection.
- When two threads are executing the iteration on same collection, if the second thread is modifying the collection (adding or removing elements) while the first thread is still iterating over the elements.. what happens…
- The iteration code in the first thread throws *ConcurrentModificationException* and fails immediately, this is '*fail-fast iterators*'.

Why Fail fast iterator..?

- Iterating a collection while it is being modified by another thread is very dangerous.
- The collection may have more, less or no elements after the iterator has been obtained, that leads to unexpected behavior and inconsistent result.
- And this should be avoided as early as possible, thus the iterator must throw an exception to stop the execution of the current thread.

Stop when fail-fast..

- Since the fail-fast behavior is not guaranteed. If this exception is thrown, the program should stop immediately, instead of continuing the execution

Iterator in synchronized

- Using the iterator of a synchronized collection, use *synchronized* block to safeguard the iteration code because the iterator itself is not thread-safe.

Collections performance…

- Drawback of synchronized collections is that their synchronization mechanism uses the collection object itself as the lock object.
- That means when a thread is iterating over elements in a collection, all other collection's methods block, causing other threads having to wait.
- Other threads cannot perform other operations on the collection until the first thread release the lock. This causes overhead and reduces performance.

Concurrent Collections

- Java 5 introduced ***concurrent collections*** that provide much better performance than synchronized wrappers.
- The concurrent collection classes are organized in the **java.util.concurrent** package.
- They are categorized into 3 groups based on their thread safety mechanisms.

Where read is more than write..

- ***copy-on-write* collections:** thread-safe collections stores values in an immutable array.
- Any change to the value of the collection results in a new array being created to reflect the new values.
- These collections are designed for situations in which read operations are more than write operations.
- The implementations of this are *CopyOnWriteArrayList and CopyOnWriteArraySet*.

Check before the update

- ***Compare-And-Swap* or *CAS* collections:** implement an algorithm called Compare-And-Swap (CAS)
- To perform calculation and update value of a variable, it makes a local copy of the variable and performs the calculation without getting exclusive access.
- When it is ready to update the variable, it compares the variable's value with its value at the start and, if they are the same, updates it with the new value

Get back and retry If updated..

- If they are not the same, the variable must have been changed by another thread.
- In this situation, the CAS thread can try the whole computation again using the new value, or give up, or continue.
- Collections using CAS are ConcurrentLinkedQueue and ConcurrentSkipListMap .

Collections with Special Lock

- **Concurrent collections using a special lock object java.util.concurrent.lock.Lock:** This mechanism is more flexible than classical synchronization.
- A lock has the same basic behavior as classical synchronization but a thread can also acquire it under special conditions: only if the lock is not currently held, or with a timeout, or if the thread is not interrupted.

Unlock the lock..

- Unlike synchronization code, in which an object lock is held while a code block or a method is executed, a Lock is held until its unlock() method is called.
- The *LinkedBlockingQueue* has separate locks for the head and the tail ends of the queue, so that elements can be added and removed in parallel.
- Other collections using this lock are *ConcurrentHashMap* and most of the implementations of *BlockingQueue*.
- Collections in this group have weakly consistent iterators and do not throw *ConcurrentModificationException*.

HashMap and Hashtable

- Hashtable is synchronized, HashMap is not. This makes HashMap better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones.
- Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.

LinkedHashMap with Iteration

- One of HashMap's subclasses is LinkedHashMap,
- To get predictable iteration order (which is insertion order by default), use LinkedHashMap over HashMap.
- This wouldn't be as easy with Hashtable.

ConcurrentHashMap

- This is a replacement of synchronized hash-based map implementations e.g. Hashtable and synchronized HashMap.
- It implements Map and ConcurrentMap (a sub-interface of Map) interface which allows to store key-value pairs.
- This is similar to HashMap or Hashtable but it's more **scalable** and the right fit for concurrent Java applications

Scalability with ConcurrentHashMap

- Unlike Hashtable which achieves its thread-safety by compromising the scalability, ConcurrentHashMap uses advanced techniques e.g. dividing the map into segments to remain thread-safe and scalable at the same time.
- Because of its performance and scalability as well as thread-safety, it is the choice of Map in concurrent Java applications.

Deadlock Situation

- If two threads are trying to call each other's methods which are synchronized; the first thread which has acquired the lock starts the execution.
- If now second thread tries to call synchronized method on first thread, it will have to wait till the first thread has finished the execution (release the lock) whereby it is waiting on second thread while this thread is waiting for first thread to release the lock which will never succeed.

This creates a deadlock situation and should be avoided.

Avoid deadlock..?

- Java doesn't have anything to escape deadlock state when it occurs, so design the program to avoid deadlock situation.
- Avoid acquiring more than one lock at a time. If not, make sure that you acquire multiple locks in consistent order.

Livelock

- ***A situation where two threads are busy responding to actions of each other***.
- They keep repeating a particular code so the program is unable to make further progress:
- *Thread 1 acts as a response to action of thread 2*
- *Thread 2 acts as a response to action of thread 1*
- Unlike deadlock, threads are not blocked when livelock occurs.
- They are simply too busy responding to each other to resume work.
- The program runs into an infinite loop and cannot proceed further.

A Thread is Starving for resources..

- ***Starvation is a situation where a greedy thread holds a resource for a long time so other threads are blocked forever***.
- The blocked threads keep waiting to acquire the resource but they never get a chance and they starve to death.
- In starvation situation occurs, the program still runs but doesn't run to completion because some threads are not executed.

Reasons for Starvation

- Threads are blocked infinitely because a thread takes long time to execute some synchronized code (e.g. heavy I/O operations or infinite loop).
- A thread doesn’t get CPU’s time for execution because it has low priority as compared to other threads which have higher priority.
- Threads are waiting on a resource forever but they remain waiting forever because other threads are constantly notified instead of the hungry ones.

Parallel vs. Concurrent execution

- In parallel execution, each thread is executed in a separate processing core. Therefore, tasks are really executed in true parallel fashion.
- In concurrent execution, the threads are executed on a same core of CPU. That means tasks are actually executed in interleave fashion, sharing processing time of a processing core.

Fork/Join Framework in Java7

Fork/Join framework is a set of Java APIs that allow to take advantage of parallel execution supported by multi-core processors.

Fork/Join execution

- It uses ‘divide-and-conquer’ strategy: divide a very large problem into smaller parts, which in turn, the small part can be divided further into smaller ones, recursively until a part can be solved directly. This is a ‘fork’.
- Then all parts are executed in parallel on multiple processing cores. The results of each part are ‘joined’ together to produce the final result. Hence the name of the framework ‘Fork/Join’.

Thread POOL

- In terms of performance, creating a new thread is an expensive operation because it requires the operating system allocates resources need for the thread.
- Therefore, in practice thread pool is used for large-scale applications that launch a lot of short-lived threads in order to utilize resources efficiently and increase performance.
- Example : DataSource connections maintained by server manager.

Use Threads from the POOL

- Instead of creating new threads when new tasks arrive, a thread pool keeps a number of idle threads that are ready for executing tasks as needed.
- After a thread completes execution of a task, it does not die. Instead it remains idle in the pool waiting to be chosen for executing new tasks.
- The definite number of concurrent threads in the pool can be limited which is useful to prevent overload.
- If all threads are busily executing tasks, new tasks are placed in a queue, waiting for a thread becomes available.

Thread Pool Types

- The Java Concurrency API supports the following types of thread pools:
- **Cached thread pool**: keeps a number of alive threads and creates new ones as needed.
- **Fixed thread pool**: limits the maximum number of concurrent threads. Additional tasks are waiting in a queue.
- **Single-threaded pool**: keeps only one thread executing one task at a time.
- **Fork/Join pool**: a special thread pool that uses the Fork/Join framework to take advantages of multiple processors to perform heavy work faster by breaking the work into smaller pieces recursively.

Using the thread pool

- In practice, thread pool is used widely in web servers where a thread pool is used to serve client's requests.
- Thread pool is also used in database applications where a pool of threads maintaining open connections with the database.
- It is majorly implemented on server side and custom frameworks to manage the performance.

Executor

- An **Executor** is an object that is responsible for threads management and execution of Runnable tasks submitted from the client code.
- It decouples the details of thread creation, scheduling, etc from the task submission so you can focus on developing the task's business logic without caring about the thread management details.

Executor Usage

- Instead of creating a thread to execute a task..
- Thread t = new Thread(new RunnableTask());
- t.start();
- submit tasks to an executor
- Executor executor = anExecutorImplementation;
- executor.execute(new RunnableTask1());
- executor.execute(new RunnableTask2());

Executor Types

- The Java Concurrency API defines the following 3 base interfaces for executors:
- **Executor**: is the super type of all executors. It defines only one method execute(Runnable).
- **ExecutorService**: is an Executor that allows tracking progress of value-returning tasks (Callable) via Future object, and manages the termination of threads. Its key methods include submit() and shutdown().
- **ScheduledExecutorService**: is an ExecutorService that can schedule tasks to execute after a given delay, or to execute periodically. Its key methods are schedule(), scheduleAtFixedRate() and scheduleWithFixedDelay().

Create the Executor

- Create an executor by using one of several factory methods provided by **Executors** class
- In case the factory methods do not meet the need, construct an executor directly as an instance of either ThreadPoolExecutor

  or ScheduledThreadPoolExecutor, which gives additional options such as pool size, on-demand construction, keep-alive times, etc.
- For creating a Fork/Join pool, construct an instance of the ForkJoinPool class.

Thread Dump

- A Java thread dump is a way of finding out what every thread in the JVM is doing at a particular point in time.
- A thread dump is a list of all the Java threads that are currently active in a Java Virtual Machine (JVM).
- This is especially useful if your Java application sometimes seems to hang when running under load, as an analysis of the dump will show where the threads are stuck.
- To generate a thread dump under Unix/Linux by running kill -QUIT <pid>, and under Windows by hitting Ctl + Break.

When to take the thread dumps..?

- Several ways to take thread dumps from a JVM.
- It is highly recommended to take more than 1 thread dump.
- A good practice is to take 10 thread dumps at a regular interval (for example, one thread dump every ten seconds).

How to take thread dump of java process

- **Get the PID of your Java process**
- The jps command which lists all Java process ids
- On Windows: Press Ctrl+Shift+Esc to open the task manager and find the PID of the Java process.
- **Request a thread dump from the JVM**
- **jstack** :prints thread dumps to the command line console.
- **jstack -l <pid>**

Thread dumps to file

- Output consecutive thread dumps to a file by using the console output redirect/append directive:
  jstack -l <pid> >> threaddumps.log
- Also with psexec to run jstack as SYSTEM user,
  psexec -s jstack <*pid*> >> threaddumps.log
- If the java process isn't responding, use the option **-J-d64** (on 64 bit systems), :
  jstack -J-d64 -l <*pid*> >> threaddumps.log

Analyze the thread dumps

- Read a thread dump with a text editor.
- Thread Dump Analyzer on https://github.com/irockel/tda/releases/download/2.3.3/tda-bin-2.3.3.zip
- It is a Java Swing application for analyzing thread dumps.
- It provides statistics about the thread dump, displays individual threads and gives information about locked monitors and waiting threads, flagging monitors that have many threads waiting on it.