Multithreading in Java

Multitasking:

Multitasking allows several activities to occur concurrently on the computer.

There are two types of multitasking:

1. Process Based Multitasking
2. Thread Based Multitasking

Process Based Multitasking

Allows processes (i.e. different programs) to run concurrently on the computer.

Example: Running MS word while also listening to music on Google Chrome.

Thread Based Multitasking

Allows parts of the same program to run concurrently on the computer.

Example: MS Word that is printing text, giving suggestions and formatting text at the same time.

Thread vs Process!

* Two threads share the same address space because multiple threads for the same program, for example MS word, there will be multiple threads running parallel or simultaneously to implement different functions, therefore they share same address space.
* Context switching between threads is usually less expensive than between processes.
* The cost of communication between threads is relatively low.

Why Multithreading?

* In a single-threaded environment, only one task at a time can be performed.
* CPU cycles are wasted, for example, when waiting for user input.
* Multitasking allows idle CPU time to be put to good use.
* Multithreading makes multitasking possible when it breaks programs into smaller, executable threads. Each thread has the programming elements needed to execute the main program, and the computer executes each thread one at a time.

Threads

* A thread is an independent sequential path of execution within a program.
* Java has its Main thread which we do not need to create explicitly. It is created automatically. When this Main thread stops executing, then the program flow stops.
* Many threads can run concurrently within a program.
* At runtime, threads in a program exist in a common memory space and can, therefore, share both data and code (i.e., they are lightweight compared to processes).

Main Thread

* When a standalone application is run, a user thread is automatically created to execute the main() method of the application. This thread is called the main thread.
* If no other threads are found, the program terminates when the main() method finishes executing.
* All other threads, called the child threads, are spawned from the main thread.
* If the user threads are running but the main() method finishes, the program will keep running until all user threads have completed.

Daemon Thread

* Calling the `setDaemon(boolean)` method in the Thread class marks the status of the thread as either daemon or user, but this must be done before the thread is started.
* If a user thread is alive, the JVM does not terminate.
* A daemon thread is stopped if there are no more user threads running, thus terminating the program.

CREATION OF THE THREAD

* A thread in JAVA is represented by an Object of the Thread class.
* Creating threads is achieved in two ways:

1. Implementing the `java.lang.Runnable Interface`

2. Extending the `java.lang.Thread Class`

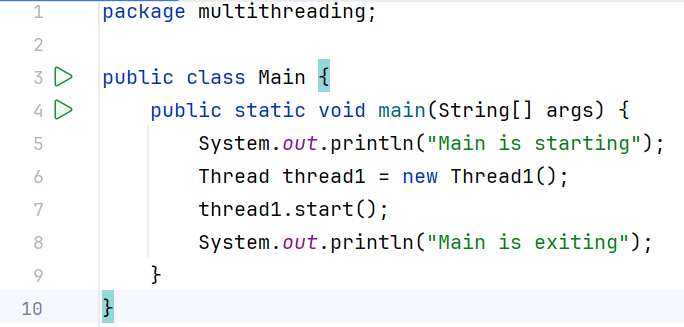
Extending the Thread Class

Thread 1 Class:

A screen shot of a computer code

Description automatically generated

Main Class:



Output:

A screenshot of a computer program

Description automatically generated

* There is NO ORDER that is being followed in the execution of threads. It is up to JVM in which order does it executes the code. If we run the same code in some other machine, it is possible that it will print Inside Thread1 0, Inside Thread1 1, Main is exiting, and after that the rest of the Inside Thread1 statements. It is platform dependent.
* Here, the main thread finished but the program didn’t terminate and allowed Thread1 to execute because it was a user thread.

We can also set the name of the Thread by passing a name in the constructor and overloading the constructor by the ThreadName, where super method, will be called and the thread name would be set.

If we want to print the name of the thread, there is a static method inside Thread class `currentThread()`which returns the instance of the thread which is running, and we can get the name of the thread by `getName()`.

A computer screen shot of a computer code

Description automatically generated

A screen shot of a computer code

Description automatically generated

Implementing the Runnable Interface

Thread 2 Class:

A screen shot of a computer code

Description automatically generated

While implementing Runnable Interface, we must override the run method, because it is a functional interface with one abstract method, and we must override it.

Main Class:

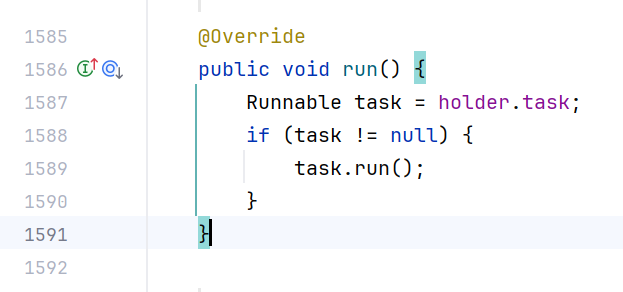
A computer code with colorful text

Description automatically generated

There is another constructor where we can pass an object of the runnable interface, and we can also pass thread name (its optional).

Rest is the same as in extending the Thread Class.

Internal Implementation of run method when we are implementing Runnable.



Whenever we pass an object of the runnable interface, it set the task by the constructor and here the task != null, there it calls the run method and executes the thread.

However, when we extend the Thread class, our run method which we have overridden will get executed because of method overriding in JAVA.

Life Cycle of a Thread



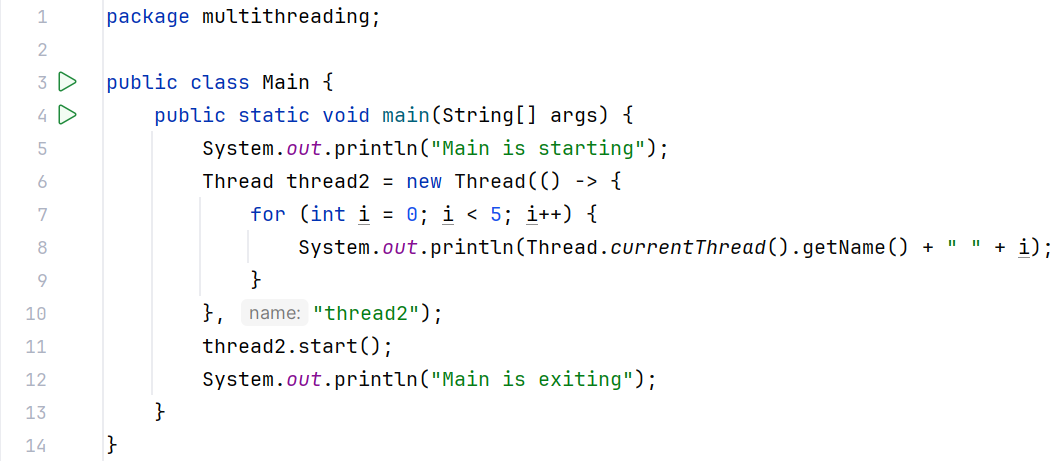
1. New: Instance of thread is created which is not yet started by invoking `Thread.start()`
2. Runnable: After invocation of `Thread.start() `and before it is selected to be run by the thread scheduler.
3. Running: After thread scheduler has selected it.
4. Blocked / Non-runnable: Thread is alive, but not eligible to run.
5. Terminated: The thread has completed its execution.

Which is better approach in creating Thread? Extending Thread Class or Implementing Runnable Interface?

The better way is to implement Runnable Interface because:

* When we extend Thread class, we can’t extend any other class even if we require it because multiple inheritance in JAVA is not allowed and when we implement Runnable, we can save a space for our class to extend any other class in future or now.
* When we extend Thread class, each of our threads creates a unique object and associates with it. When we implement Runnable, it shares the same object with multiple threads.

Using Lambda function to write code.



Synchronization

Threads share the same memory space, i.e. they share resources (objects).

However, there are critical situations where it is desirable that only one thread at a time has access to a shared resource.

Example: While booking tickets for a movie, multiple threads work simultaneously to book tickets. Let’s assume that only one ticket is remaining and there are 3 different threads that are trying to book tickets simultaneously. Here comes the problem known as RACE CONDITION.

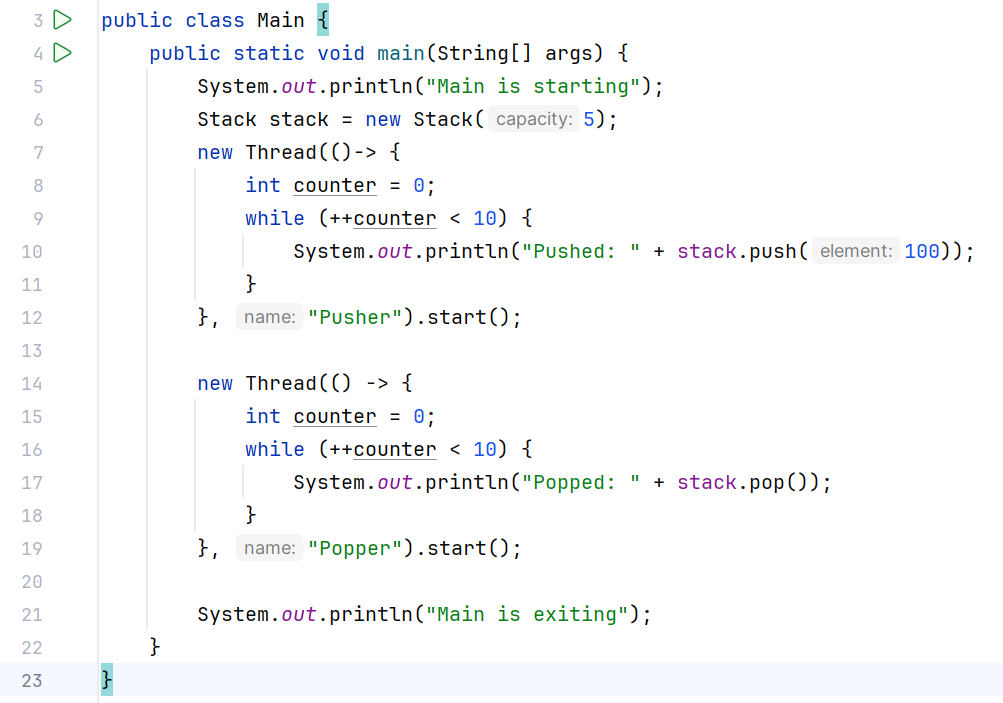
RACE CONDITION

It occurs when two or more threads simultaneously update the same value and, consequently, leave the value in an undefined or inconsistent state.

Example: STACK CLASS



MAIN CLASS:



Output:

A screenshot of a computer

Description automatically generated

Here, we get Array Index out of Bound exception because it might happen when the push thread goes to sleep after `++stackTop`, the pop thread executes `stackTop--` which is causing this exception. We can’t allow multiple threads to change the state of the object at the same time.

To achieve this, we must use synchronized block that will allow only one thread to access the shared resource at a time.

Synchronized Block

A synchronized block in Java is synchronized on some object. All synchronized blocks synchronize on the same object and can only have one thread executed inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

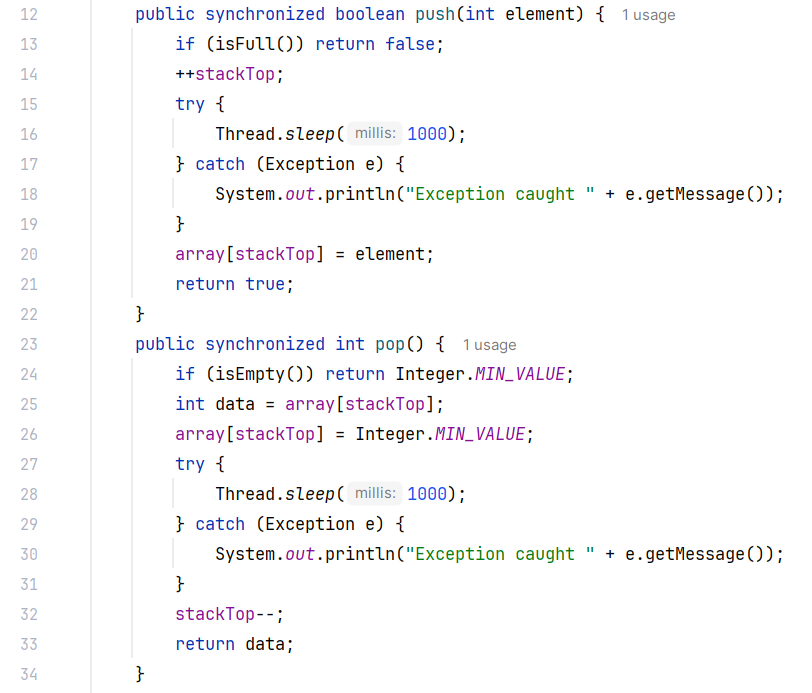
This synchronization is implemented in Java with a concept called monitors or locks. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.



Here all the critical code, that is sharing the same resources is put in the synchronized block which takes a lock as an Object. The thread which has the lock will only be able to access the critical code and after executing the whole code, it will come out and handle the lock to JVM, and then JVM will decide which thread is to be given the lock.

Here, the threads are bound by the same lock. Therefore, if thread1 has the lock and is executing push function, thread2 cannot access the pop function because the lock is with thread1.

We can also make the whole method synchronized.



When we make the whole method synchronized, the compiler behind the scenes is using the instance of this current Object (this) as the lock. It means that only one synchronized method will be executed by a thread at a time.

In case of static methods, there is no Object, therefore in that case we can use `ClassName.class`as the lock.

* While a thread is inside a synchronized method of an object, all other threads that wish to execute this synchronized method or any other synchronized method of the object will have to wait.
* This restriction does not apply to the thread that already has the lock and is executing a synchronized method of the object.
* Such a method can invoke other synchronized methods of the object without being blocked.
* The non-synchronized methods of the object can always be called at any time by any thread.

Thread Safety

It is a term used to describe the design of classes that ensure that the state of their objects is always consistent, even if the objects are used concurrently by multiple threads.

Example: StringBuffer

VOLATILE KEYWORD

A diagram of a computer keyword

Description automatically generated

The volatile keyword in Java is used to indicate that a variable’s value can be modified by different threads. Used with the syntax, volatile `dataType variableName = x;` It ensures that changes made to a volatile variable by one thread are immediately visible to other threads

Example:

Volatile Example Class

A screen shot of a computer program

Description automatically generated

Main Class

A screen shot of a computer code

Description automatically generated

Output:

A screenshot of a computer

Description automatically generated

In this example, VolatileExample is a thread that runs a loop if running is true. The stopRunning() method sets running to false, which should stop the loop.

In the main method, we start the VolatileExample thread, wait for 1 second, and then call stopRunning(). Because running is volatile, the change is immediately visible to the VolatileExample thread, and it stops running.

PRODUCER-CONSUMER PROBLEM

In computing, the producer-consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer, and the consumer, which share a common, fixed-size buffer used as a queue.

* The producer’s job is to generate data, put it into the buffer, and start again.
* At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Problem

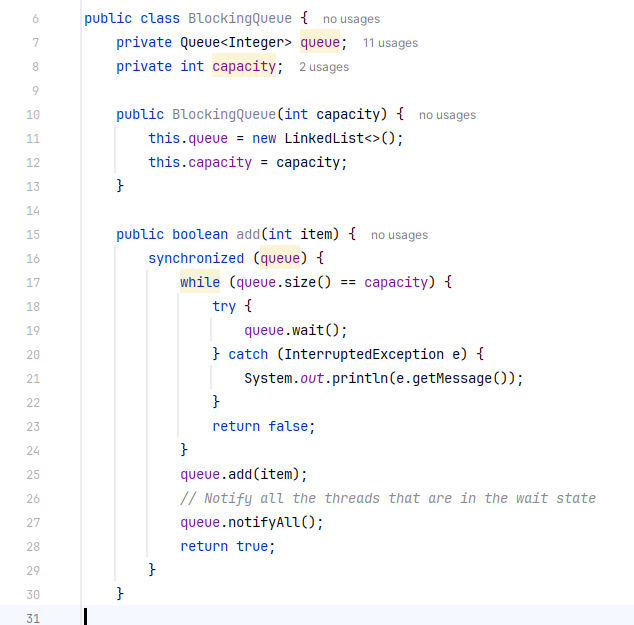
To make sure that the producer won’t try to add data into the buffer if it’s full and that the consumer won’t try to remove data from an empty buffer.

Solution

The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Example:



A screenshot of a computer code

Description automatically generated

Wait Method()

wait() method is a part of `java.lang.Object` class. When wait() method is called, the calling thread stops its execution until notify() or notifyAll() method is invoked by some other Thread.

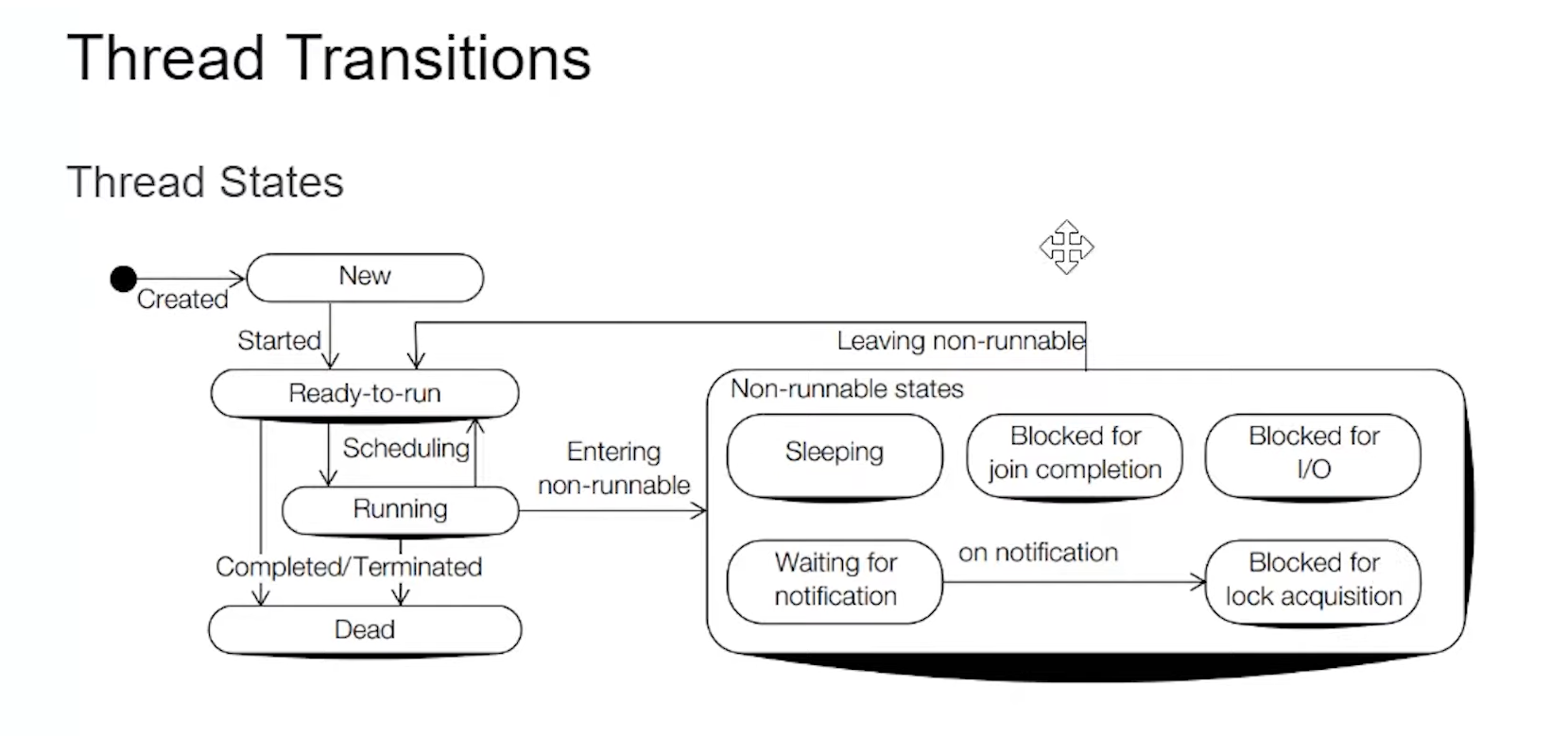
Notify and NotifyAll Method()

We use the notify() method for waking up threads that are waiting for access to this object’s monitor.

There are two ways of notifying waiting threads:

1. **Notify():** For all threads waiting on this object’s monitor (by using any one of the wait() methods), the method notify() notifies any one of them to wake up arbitrarily. The choice of exactly which thread to wake is nondeterministic and depends upon the implementation. Since notify() wakes up a single random thread, we can use it to implement mutually exclusive locking where threads are doing similar tasks. But in most cases, it would be more viable to implement notifyAll().
2. NotifyAll(): This method simply wakes all threads that are waiting on this object’s monitor. The awakened threads will compete in the usual manner, like any other thread that is trying to synchronize on this object.

THREAD TRANSITION & STATES



Thread Joining

`java.lang.Thread` class provides the join() method which allows one thread to wait until another thread completes its execution. If t is a Thread object whose thread is currently executing, then t.join() will make sure that t is terminated before the next instruction is executed by the program.

A computer screen shot of a computer program

Description automatically generated

Thread Priorities

* Threads are assigned priorities that the thread scheduler can use to determine how the threads will be scheduled.
* The thread scheduler can use thread priorities to determine which thread gets to run first.
* Priorities are integer values from **1** (lowest priority given by the constant **Thread.MIN\_PRIORITY**) to **10** (highest priority given by the constant **Thread.MAX\_PRIORITY**).
* The default priority is **5** (**Thread.NORM\_PRIORITY**).
* A thread inherits the priority of its parent thread.
* The priority of the thread can be set using the `setPriority()` method and read using the `getPriority()` method, both of which are defined in the Thread class.
* The `setPriority()` method is an advisory method, meaning that it provides a hint from the program to JVM, which the JVM is in no way obliged to honor. The JVM favors the priority, but it does not guarantee it.

A close up of a computer screen

Description automatically generated

Thread Scheduler

Schedulers in JVM implementations usually employ one of the two following strategies:

1. Preemptive scheduling
2. Time-Sliced or Round-Robin scheduling

**Preemptive scheduling**

The name of the scheduling algorithm denotes that the algorithm is related to the priority of the threads.

A diagram of a computer program

Description automatically generated

Suppose there are multiple threads available in the runnable state. The thread scheduler picks that thread that has the highest priority. Since the algorithm is also pre-emptive, therefore, time slices are also provided to the threads to avoid starvation. Thus, after some time, even if the highest priority thread has not completed its job, it must release the CPU because of pre-emption.

**Time-Sliced scheduling**

Usually, the First Come First Serve algorithm is non-preemptive, which is bad as it may lead to infinite blocking (also known as starvation). To avoid that, some time-slices are provided to the threads so that after some time, the running thread must give up the CPU. Thus, the other waiting threads also get time to run their job.

A diagram of a process

Description automatically generated

In the above diagram, each thread is given a time slice of 2 seconds. Thus, after 2 seconds, the first thread leaves the CPU, and the CPU is then captured by Thread2. The same process repeats for the other threads too.

Deadlocks

* A deadlock is a situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds.
* Since each thread is waiting for the other thread to relinquish the lock, they both remain waiting forever in the Blocked-for-lock-acquisition state.
* The threads are said to be **deadlocked**.

Example:

A screenshot of a computer program

Description automatically generated

Here, initially thread1 has acquired lock1 and thread2 has acquired lock2.

Now, thread1 needs lock2 and thread2 needs lock1 which are currently with thread2 and thread1 respectively. Therefore, it is a deadlock situation. The thread1 and thread2 will never end and the program will keep executing.

EXECUTOR SERVICE

The `java.util.concurrent.Executors` provide factory methods that are being used to create Thread Pools of worker threads. Thread pools overcome this issue by keeping the threads alive and reusing the threads. Any excess tasks flowing in, that the threads in the pool can’t handle are held in a Queue. Once any of the threads get free, they pick up the next task from this queue. This task queue is essentially unbounded for the out-of-box executors provided by the JDK.

Some types of Java Executors are listed below:

1. SingleThreadExecutor
2. FixedThreadPool(n)+
3. CachedThreadPool
4. ScheduledExecutor

**Executor 1: SingleThreadExecutor**

A thread pool of single thread can be obtained by calling the static newSingleThreadExecutor() method of the Executors class. It is used to execute tasks sequentially.

A white rectangular object with black text

Description automatically generated

Executor 2: FixedThreadPool(n)

As the name indicates, it is a thread pool of a fixed number of threads. The tasks submitted to the executor are executed by the n threads and if there are more tasks they are stored on a LinkedBlockingQueue. It uses Blocking Queue.

A close-up of a white box

Description automatically generated

Executor 3: CachedThreadPool

Creates a thread pool that creates new threads as needed but will reuse previously constructed threads when they are available. Calls to execute will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. It uses a SynchronousQueue queue.

A white rectangular object with black text

Description automatically generated

Executor 4: ScheduledExecutor

Scheduled executors are based on the interface ScheduledExecutorService which extends the ExecutorService interface. This executor is used when we have a task that needs to be run at regular intervals or if we wish to delay a certain task.

A close-up of a computer code

Description automatically generated

Future Object

The result of the task submitted for execution to an executor can be accessed using `java.util.concurrent`. The future object returned by the executor. Future can be thought of as a promise made to the caller by the executor. The future interface is mainly used to get the results of Callable results. Whenever the task execution is completed, it is set in this Future object by the executor.