**Multithreading:**

**A Thread is flow of execution control**. In multithreading a no.of threads execute concurrently and usually asynchronously (unless explicitly synchronized).

In multithreading a main task swaps among no.of threads and each thread performs the process assigned to it.

In multithreading the **processor time** is allotted to all the threads that are ready to execute. This is done using timesharing technique. All the threads that are ready to execute will wait for processor availability, on availability a thread gets the processor time and executes for the allotted time period. If the task could not be completed at the given time the thread is suspended and another thread is loaded. The scheduling of the thread and their execution depends on many factors like priority, OS environment and synchronization etc.

**Jumping between one thread to another is called context switching** and it is of light weight because all threads share memory of parent task and utilize the same memory resources. Multithreading is in-built in Java and is achieved either by extending **Thread** class or by implementing **Runnable** interface.

Both Thread class and Runnable interface provide a method called **public void run()**. While main() is starting point of any program, **run() is starting point for thread.** run() has to be called by start() of Thread class. All methods required for thread handling and management are available in Thread class whereas Runnable provides only run().

**Thread class Methods:**

1. **sleep(int milliseconds):** It suspends the process for given no.of milliseconds.

2. **setName(String name):** Assigns the given name to the thread.

3. **String getName():** Returns the name of the specified thread.

4. **int activeCount():** Returns the count of the threads that are currently available.

5. **setPriority(int priority):** A**ssigns a priority values to the thread. The value can be between 1 and 10. Default value is 5.**

6. **getPriority():** Returns the priority value of the specified thread.

7. **join() throws InterruptedException:** joins the child threads to a thread group thus making the thread group to wait till child thread complete their task.

8. **boolean isAlive():** Returns true if the thread available in the memory or is running. But returns false if the thread is killed.

1. **boolean isDaemon():** Returns true if the thread is daemon thread otherwise, it returns false.
2. **setDaemon(boolean):** If true is specified then the thread is converted to a Daemon thread.
3. **wait():** Suspends the thread execution until **notify**() is called.
4. **notify():** Resumes the thread execution that has been suspended by wait().
5. **notifyAll():** Resumes all threads execution that are suspended by wait().
6. **start():** Invokes the **run** method thus beginning the thread execution.

//program that dispalys Thread information

class threadinfo implements Runnable

{

Thread x;

threadinfo(String str)

{

x=new Thread(this);

x.start();

x.setName(str);

x.setPriority(7);

}

public void run()

{

System.out.println(x);

}

}

class threaddemo1

{

public static void main(String args[])

{

threadinfo t1=new threadinfo(“Thread – 1”);

threadinfo t2=new threadinfo(“Thread – 2”);

threadinfo t3=new threadinfo(“Thread – 3”);

}

**Output:**

Thread[Thread-0,5,main]

Thread[Thread-2,5,main]

Thread[Thread-1,5,main]

**joining threads:**

**The parent thread group should wait and must not exit before the child threads have completed their task. In some Operating Systems if the parent thread exits before the completion of child threads execution, then the program will hang**. To resolve this, Java provides join(). join() joins threads to the method in which you have invoked join(). Consequently, the method will wait till all threads completes their execution. join() causes **InterrupedException**.

//program that demonstrates concurrent execution of Threads and sleep() and join()

class thread1 implements Runnable

{

Thread x;int d;

thread1(String name,int delay)

{

x=new Thread(this);

x.setName(name);

d=delay;

x.start();

}

public void run()

{

try

{

System.out.println(x.getName()+" started");

for(int i=1;i<=30;i++)

{

System.out.println(x.getName()+"'s i="+i);

x.sleep(d);

}

}catch(InterruptedException e){}

System.out.println(x.getName()+" exited");

}

}

class threaddemo2

{

public static void main(String args[])

{

System.out.println(">>>main started<<<");

thread1 t1=new thread1(“Thread – 1”,2000);

thread1 t2=new thread1(“Thread – 2”,2500);

thread1 t3=new thread1(“Thread – 3”,3000);

try

{

t1.x.join();

t2.x.join();

t3.x.join();

}catch(InterruptedException e){}

System.out.println(">>>main exited<<<");

}

}

**Daemon threads:**

Daemon threads are usually such threads that have no definite period of execution. They are interested will infinite tasks. But these threads should terminated automatically as and when the program started them closes. While a concrete thread continues its execution till the task interested is not completed (even if the main program closes). The daemon thread will shuts down when the main() closed.

Use setDaemon(true) to convert a concrete thread as a daemon thread.

//program that demonstrates daemon threads

class daemon implements Runnable

{

Thread x;

daemon(String name)

{

x=new Thread(this,name);

x.setDaemon(true);

x.start();

}

public void run()

{

int i=1;

System.out.println(x.getName()+" started");

while(true)

{

try

{

System.out.println(x.getName()+"'s i="+i);

i++;

x.sleep(500);

}catch(InterruptedException e){}

}

}

}

class daemondemo

{

public static void main(String args[])

{

System.out.println(">>>Main Started<<<");

daemon d1=new daemon("abc");

daemon d2=new daemon("pqr");

daemon d3=new daemon("xyz");

try

{

Thread.sleep(10000);

}catch(InterruptedException e){}

try

{

d1.x.join();

d2.x.join();

d3.x.join();

}catch(InterruptedException e){}

System.out.println(">>>Main Exited<<<");

}

}

**Thread priorities:**

A priority is a relative value ranging from 1 to 10. The priority denotes relative importance of a thread over the other threads during the execution. The higher priority threads get more CPU time when compared to other. Due to this a higher priority threads get more amount of task than others. Based on the OS strategy handling of threads valid. In a cooperative multithreading environment when a higher priority thread is ready to execute the OS evaluates time required by the low priority thread completes its task. If time required is very less then the low priority thread completes its task first then higher priority thread next. If the time required is more the low priority thread is suspended and higher priority thread is given processor.

Use setPriority() to set priority. Java provides 3 constants namely

Thread.MIN\_PRIORITY

Thread.NORM\_PRIORITY

Thread.MAX\_PRIORITY

//program that demonstrates thread priorities

class threadp implements Runnable

{

Thread x;

int i=0;

boolean flag;

threadp(String name,int p)

{

x=new Thread(this,name);

x.setPriority(p);

}

void go()

{

flag=true;

x.start();

}

public void run()

{

while(flag==true)

{

i++;

}

}

void stop()

{

flag=false;

System.out.println(x.getName()+"'s priority="+x.getPriority()+" and it's i="+i);

}

}

class prioritydemo

{

public static void main(String args[])

{

threadp t1=new threadp(“thread-1",1);

threadp t2=new threadp("thread-2",3);

threadp t3=new threadp("thread-3",5);

threadp t4=new threadp("thread-4",9);

Thread m=Thread.currentThread();

m.setPriority(10);

t1.go();

t2.go();

t3.go();

t4.go();

try

{

m.sleep(5000);

}catch(InterruptedException e){}

t1.stop();

t2.stop();

t3.stop();

t4.stop();

try

{

t1.x.join();

t2.x.join();

t3.x.join();

t4.x.join();

}catch(InterruptedException e){}

}

}

**Output:**

abc's priority=1 and it's i=6066887

bbc's priority=3 and it's i=11324065

cgh's priority=5 and it's i=110631953

xyz's priority=9 and it's i=974697401

**Synchronization:**

By default, threads are asynchronous. Threads that compete for processor will acquires the processor on availability and have to yield the processor after the expire of its allotted time slice. But synchronization allows continues execution of thread till the task is not completed. Due to this the thread in the midway will not blocked by the other thread.

Java provides synchronized() which takes object as argument. The method of the object that you call within the synchronized() will executed in synchronization. Java also provides synchronized keyword. A method declared with this keyword will also execution in synchronization.

//program that demonstrates synchronization of threads

class table

{

synchronized void showtable(int n)

{

System.out.println(n+" table started");

for(int i=1;i<=10;i++)

System.out.println(n+"\*"+i+"="+(n\*i));

System.out.println(n+" table completed");

}

}

class tablegenerator implements Runnable

{

table tbl;

int tno;

Thread x;

tablegenerator(table tb,int no)

{

x=new Thread(this);

tno=no;

tbl=tb;

x.start();

}

public void run()

{

tbl.showtable(tno);

}

}

class synchdemo

{

public static void main(String args[])

{

table tbl=new table();

tablegenerator tg1=new tablegenerator(tbl,2);

tablegenerator tg2=new tablegenerator(tbl,3);

tablegenerator tg3=new tablegenerator(tbl,5);

try

{

tg1.x.join();

tg2.x.join();

tg3.x.join();

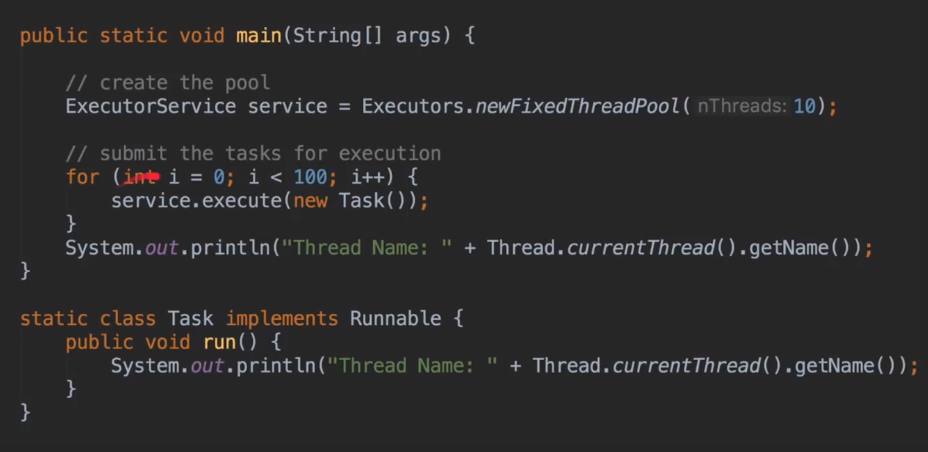
}catch(InterruptedException e){}

}

}

**Executor Framework:**

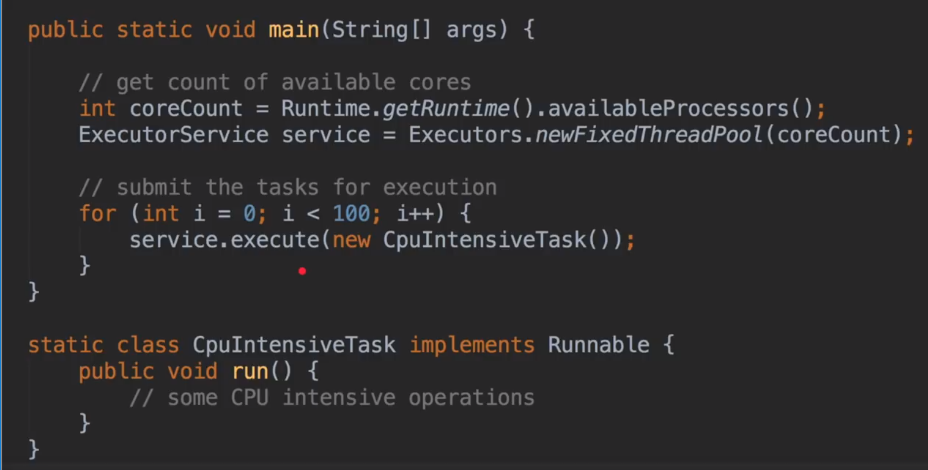
* For creating more of threads to run asynchronously is an expensive operation, so we go for concept of thread pool.
* Executor service will create a pool of threads and tasks need to be submitted to service. Service handles the tasks concurrently using **blocking Queue principle which is thread safe.**
* Blocking queue stores all the tasks in it. Thread after completing a task, it fetches new task from queue and executes it.



A screenshot of a cell phone

Description automatically generated

* Ideal pool size is no of cpu cores in system in case of cpu intensive operations.



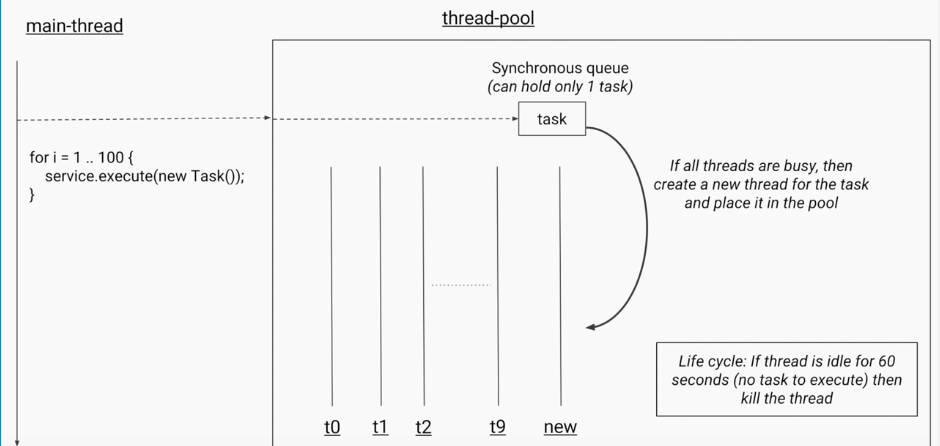
* In case of IO operations(database calls, network calls) we should define a larger pool

A screenshot of a cell phone

Description automatically generated

4 kinds of Thread Pools:

1. **Fixed Thread Pool**: above one.
2. **Cached Thread Pool**: it has a synchronous Queue which can hold only one task. This Queue searches for the Thread which is free in Pool and give the task to execute, if it doesn’t find free thread it will create a new thread and executes the task. It can create any number of threads.
   1. If any thread is idle for 60secs then it kills the thread,
   2. If no tasks coming in, the it kills the threads and thread pool size keeps shrinking.



A picture containing text

Description automatically generated

1. **Scheduled Thread Pool:** 
   1. Used if you want to schedule a task after certain delay.
   2. Can use if you want to perform checks for logging and security at a fixed rate or after a fixed delay.
   3. It stores all tasks in delay Queue. It stores tasks based on the time delay.

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

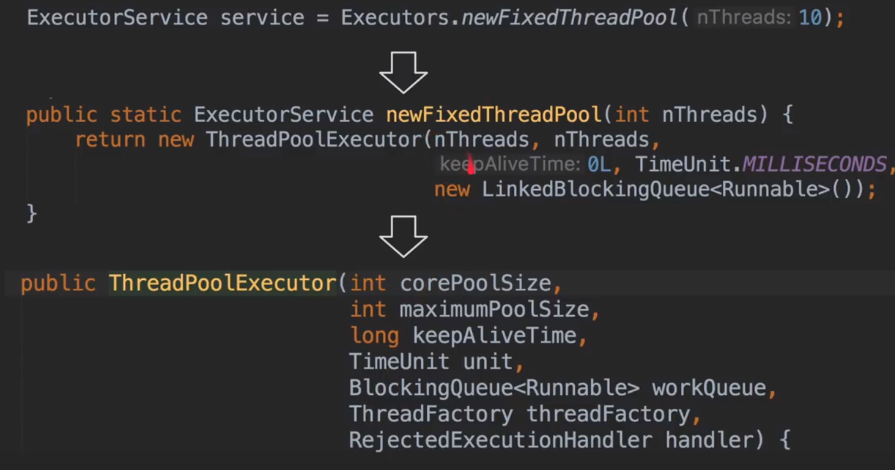
Description automatically generated

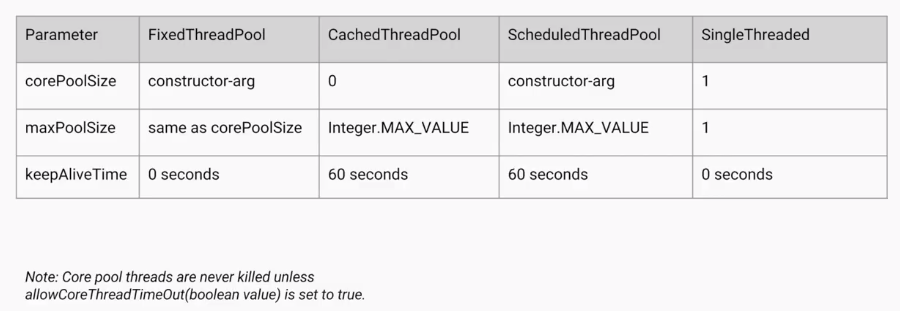
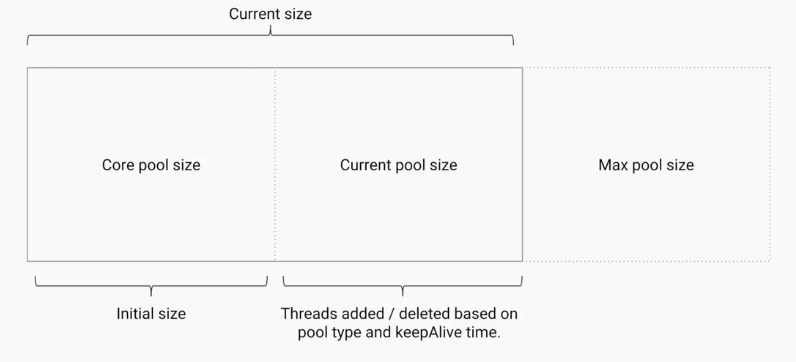
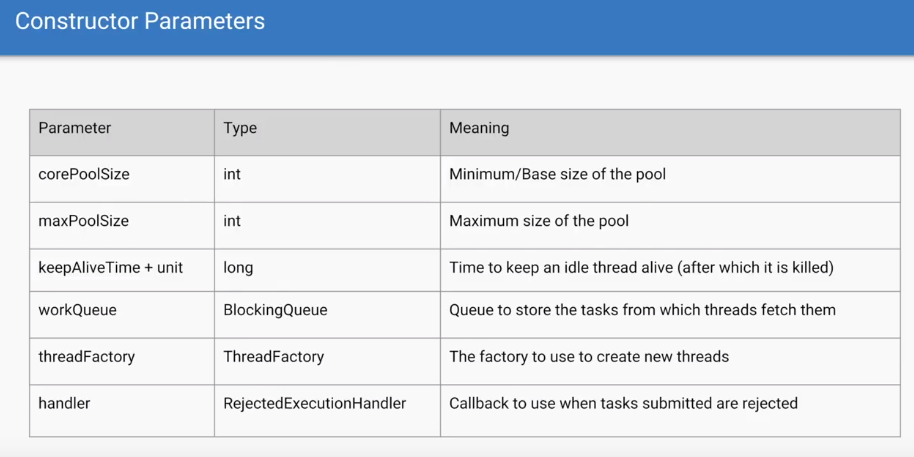
1. **Single Threaded Executor:**
   1. It works same as fixed thread executor but has only single thread.
   2. If thread got killed because of any exception in the run method, then pool creates another thread to execute.

A screenshot of a cell phone

Description automatically generated

**Constructor:**





**Type of Queues in Pool:**

A screenshot of a cell phone

Description automatically generated

**Rejection handler:**

* If there is no memory on pool, then it rejects the task based on certain policies.

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**To Shut down the thread pool:**

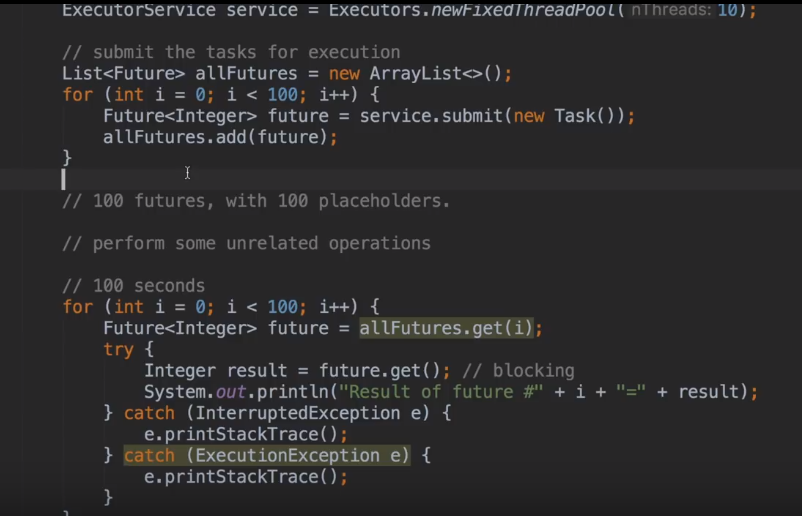
A screenshot of a cell phone

Description automatically generated

* If we implement Runnable method on the class, we need to implement run() method, by this we cannot return anything. For this purpose, if it needed to return anything from thread, we need to implement callable interface and should override call() method.
* In order to catch the returned value, we use Future type Place Holder which will arrives sometime in the future. Also, for future.get() method, main thread executes in blocking way, no error even if doesn’t have any value by that time of execution.

A screenshot of a cell phone

Description automatically generated



* Let say we have a service and submitted a task of type callable, then we get a return variable of type Future. This Future is a placeholder that is immediately returned by the Thread pool and rest of the pool works as it is.
* Once the result is ready after executing the task it will update the placeholder with that specific value.
* When callable method is running, if you call f.get() method(Blocking method). As there will be no value in the placeholder, main thread will be blocked. Main thread comes back to runnable mode once the value is updated.
* One way to overcome this problem is to use a overloaded method of get taking wait time, if it task doesn’t return any value in the time specified it will raise an time out exception.

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**Other method of get():**

A screenshot of a cell phone

Description automatically generated

**Future methods:**

