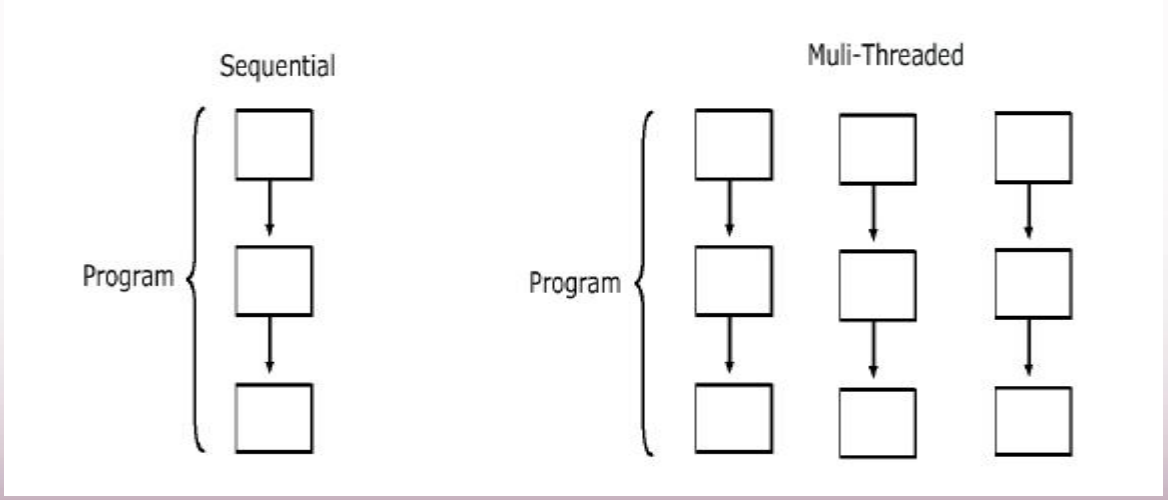
Threads

Why threads

- Need to handle concurrent processes
- Definition
  - Single sequential flow of control within a program
- Threads are like a processes executed by a program
- Example:
  - Operating System
  - HotJava web browser



Multi-threading in Java Platform

- Java Application start with just one thread, called the main thread.
  - This thread has the ability to create additional threads
- *When a new* thread is started , a new stack materializes and methods called from *that* thread run in a call stack that's separate from the main() call stack.
- That second new call stack runs concurrently with the main thread
- *A thread of execution* is an individual process (a "lightweight" process) that has its own call stack.
- *one thread per call stack—one call stack per thread*.

The Main Thread

- Every java program has at least one thread called main thread
- **Thread.*currentThread().getName()***
  - Returns the name current thread
  - setName() is used to set the name for the thread

Making a Thread

- **Done by extend the java.lang.Thread class**
  - To have specialized version of a thread class.
  - *That class can't extend another class*
- **Done By Implementing the Runnable interface**
  - Leaves class free to extend from some *other* class
  - Define behavior that will be run by a separate thread.
  - Thread Class is created with a Runnable argument.
  - The Runnable object is the *target* of the thread.

Making a Thread

- A thread begins as an instance of java.lang.Thread.
- public void run()
  - Has the Code that needs to be run in a separate thread
  - It can call other methods,
  - The thread of execution begins with the call to this method

Example

```
public class NumberService  {

public NumberService() {
     super();
}
public void sumNumbers(int countTo) {

        int count =0;

for(int i=0;i<=countTo;i++) {
    count =count +i;
}

System.out.println( " Sum of Number upto" + countTo +"Is ="+count +"Done
    By"+Thread.currentThread().getName());
  }
}
```

Implementing java.lang.Runnable

```
public class NumberAddingService  implements Runnable{

  private int countTo;
  private NumberService service;

public NumberAddingService(int countTo,String name) {
          this.countTo = countTo;
         this.service = new NumberService();

        Thread t1 = new Thread(this,name);
         t1.start();
}
@Override
public void run() {
        this.service.sumNumbers(countTo);
    }
}
```

Starting a Thread

- **start()**
  - Creates a new thread and makes it runnable
  - A new thread of execution starts (with a new call stack).
  - The thread moves from the *new* state to the *runnable* state.
  - When the thread gets a chance to execute, its target run() method will run.

Starting Multiple Thread

```
public static void main(String[] args) {

  new NumberAddingService(50,"Fifty Count");

  new NumberAddingService(500,"Five Hundred Count");

  new NumberAddingService(5000,"Five Thousand Count");

            try {

            System.out.println("Enter the Number");
                  int ch =System.in.read();

                  System.out.println(ch);
} catch (IOException e) {
    e.printStackTrace();
  }
}
```

The Thread Scheduler

- The thread scheduler is the part of the JVM
- Any thread in the *runnable* state can be chosen by the scheduler to be the one and only *running* thread.
- **java.lang.Thread Class**
  - public static void sleep(long millis) throws InterruptedException
  - public final void join()
- **java.lang.Object Class**
  - public final void wait()
  - public final void notify()
  - public final void notifyAll()

Thread States

- **New**
  - When the Thread has been instantiated
    - The start() method has not been invoked on the thread.
  - It is a live Thread object, but not yet a thread of execution.
  - At this point, the thread is considered *not alive*.

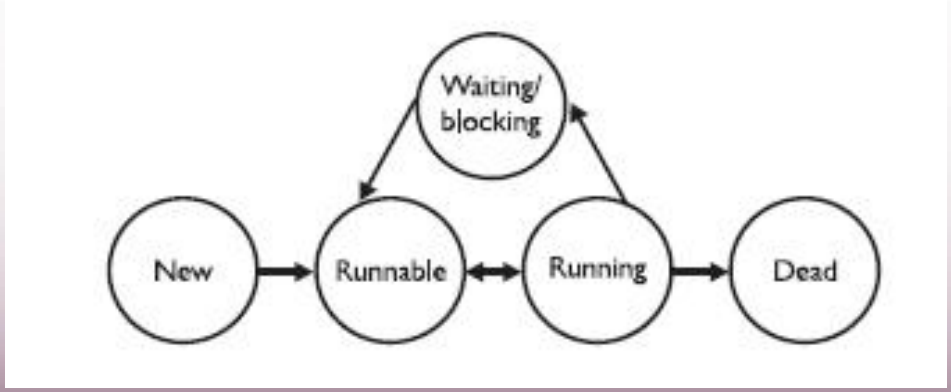
- **Runnable**
  - The state a thread is in when it's eligible to run,
  - Scheduler has not selected it to be the running thread.
  - A thread first enters the runnable state when the start() method is invoked
  - A thread can also return to the runnable state after either running or coming back from a blocked, waiting, or sleeping state.
  - When the thread is in the runnable state, it is considered *alive*.

Thread States

- **Running**
  - This is the state a thread is in when the thread scheduler selects it from the runnable pool to be the currently executing process.
  - A thread can transition out of a running state for several reasons, including because “the thread scheduler felt like it.”
- **Waiting/blocked/sleeping**
  - The thread is still alive, but is currently not eligible to run.
  - A thread may be *blocked* waiting for a resource
  - A thread may be *sleeping* because the thread’s run code *tells* it to sleep for some period of time
  - *waiting*, because the thread’s run code *causes* it to wait

Thread States

- **Dead**
- A thread is considered dead when its run() method completes.
- It may still be a viable Thread object, but it is no longer a separate thread of execution. Once a thread is dead, it can never be brought back to life
- A runtime exception will be thrown



Leave the running state

- **sleep()**
- Guaranteed to cause the current thread to stop executing
  - Duration is specified in milli seconds
  - It might be *interrupted* before its specified time
- **join()**
- Guaranteed to cause the current thread to stop executing until the thread it joins with the thread it calls wait on completes.
- **Leave the Running State:**
- Following scenarios
  - The thread's **run() method completes.**.
  - A thread **can't acquire the *lock* on the object** whose method code it's attempting to run

The sleep() method

- Static method of Thread Class
- Can be Placed anywhere in the Code
- Throws a Checked Exception – Interrupted Exception

```
try {
   Thread.sleep(5*60*1000); // Sleep for 5 minutes
   } catch (InterruptedException ex) { }
```

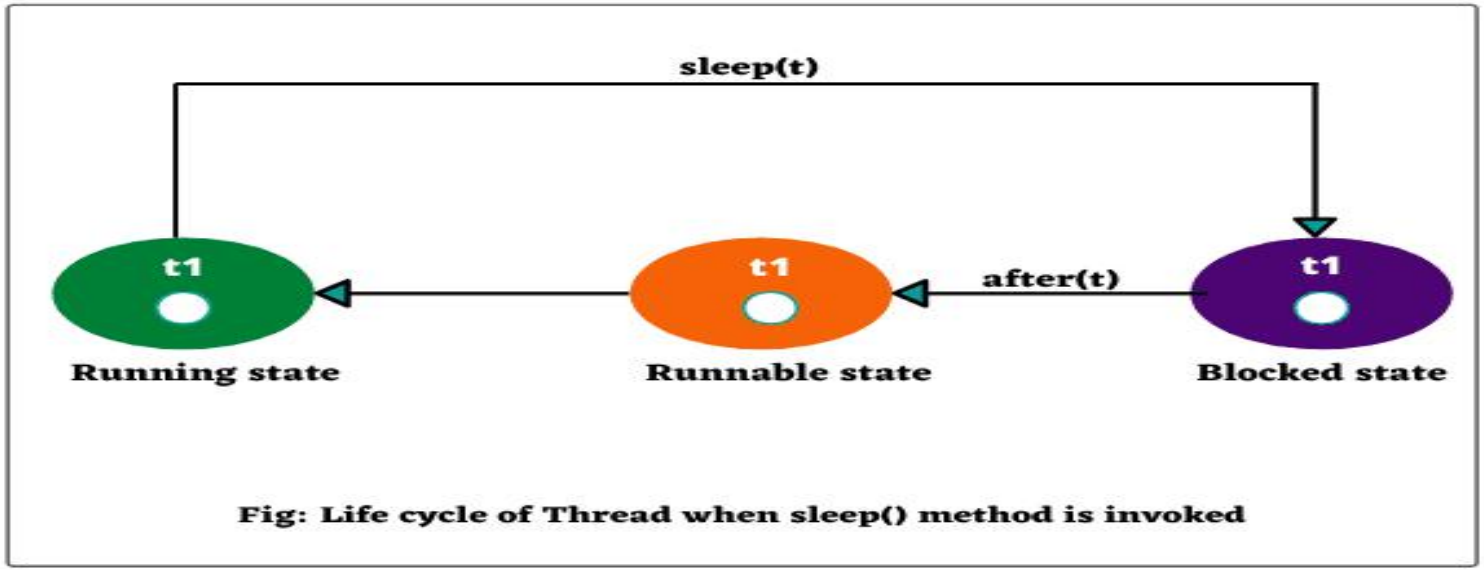
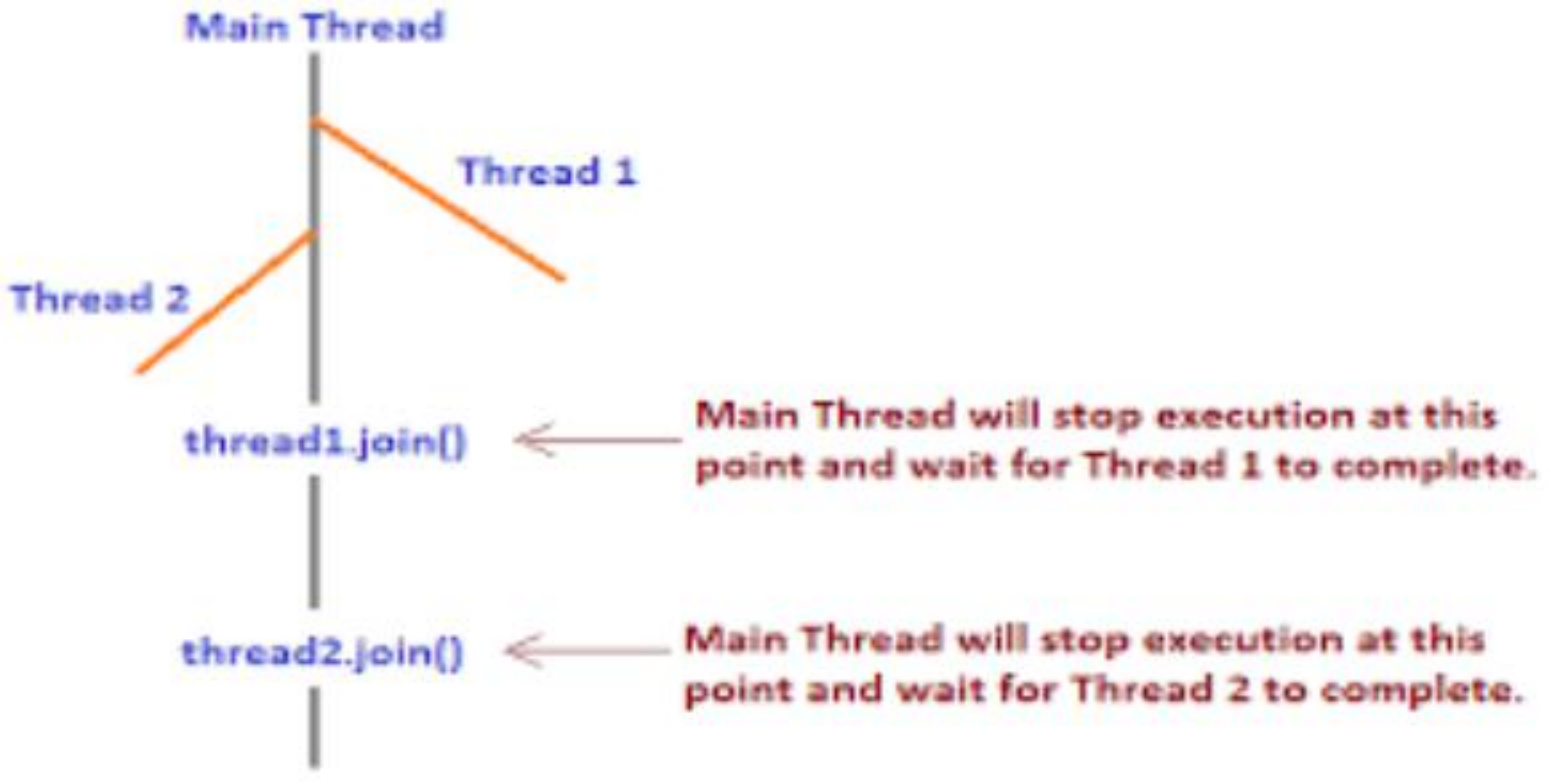


Fig: Life cycle of Thread when sleep() method is invoked

The Join() Method

- The non-static join() method of class Thread lets one thread “join onto the end” of another thread.
- The thread class join method waits until a thread is finished executing or waiting for a thread to die before returning .



Joining Thread

```
public class TaskForJoin implements Runnable {

public void run() {

 System.out.println("Reading");

        try {

            System.in.read( );

        } catch (java.io.IOException ex) {

           }

        System.out.println("Thread Finished.");
    }
}
```

Joining Thread

```
public class ParentThread {

public static void main(String[] args) {

System.out.println("Starting");

TaskForJoin task = new TaskForJoin();

Thread t = new Thread(task);

        t.start( );

        System.out.println("Joining");
```

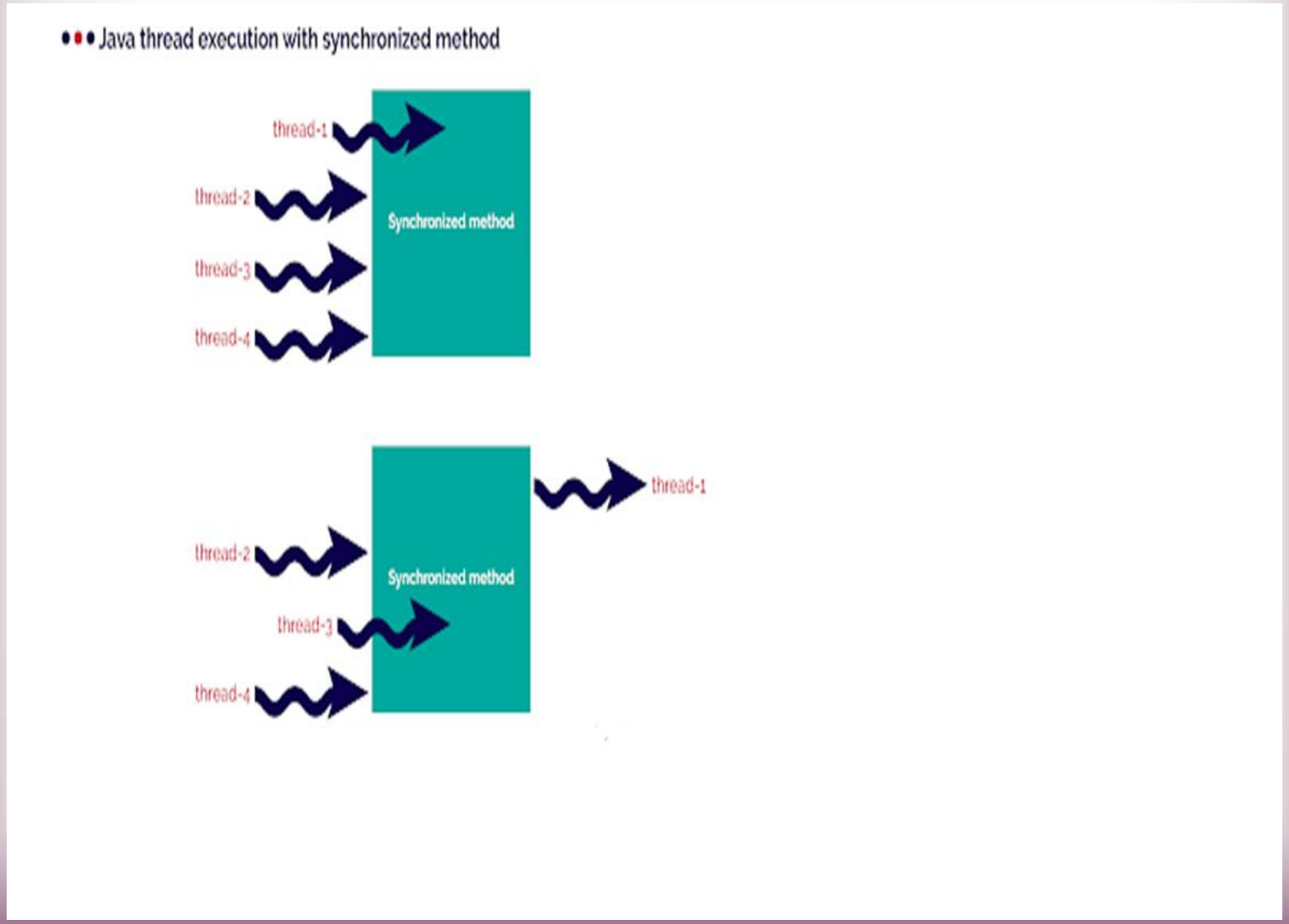
Joining Thread

```
        try {
            t.join( );
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        System.out.println("Main Finished.");
    }
}
```

Synchronizing Code

- Threads can share resources.
- Sometimes it is desirable that only one thread at a time has access to a shared resource.
- Achieved by using the keyword: `synchronized`, Only *methods* can be synchronized, not variables.
- Protects access to code, not to data
  - Make data members private
  - Synchronize accessor methods
- A lock can be associated with a shared resource. Threads gain access to a shared resource by first acquiring the lock associated with the resource.
- one thread can hold the lock and thereby have access to the shared resource.

Synchronizing Code



Synchronizing Code

- A class can have both synchronized and no synchronized methods.
- If a class has both synchronized and nonsynchronized methods, *multiple threads can still access the nonsynchronized methods* of the class!
- *A thread can acquire more than one lock.*
- A thread can enter a synchronized method, thus acquiring a lock, and then immediately invoke a synchronized method on a different object, thus acquiring *that* lock as well.
- As the stack unwinds, locks are released again.

Thread Synchronization

```
class TwoStrings {

synchronized static void print(String str1, String
  str2) {

System.out.print(str1);

try {
Thread.sleep(500);
} catch (InterruptedException ie)
{
ie.printStackTrace();
}
System.out.println(str2);
}
}
```

Thread Synchronization

```
class PrintStringsThread implements Runnable {

String str1, str2;

PrintStringsThread(String str1, String str2) {

    this.str1 = str1;
    this.str2 = str2;

 Thread thread = new Thread(this);
 thread.start();
}

public void run() {
 TwoStrings.print(str1, str2);

 }
}
```

Thread Synchronization

```
public class Test {

public static void main(String args[])
{
new PrintStringsThread("Hello ", "there.");
 new PrintStringsThread("How are ", "you?");
 new PrintStringsThread("Thank you ","very much!");
 new PrintStringsThread("Bye ","Bye!");

}
}
```

Output- May be in Different Order

```
Hello there
How are you
Thank you very much
Bye,Bye
```

Thread Interaction

- The java.lang.Object class has three methods—wait(), notify(), and notifyAll()—that help threads communicate
- wait(), notify(), *and* notifyAll() *must be called from within a synchronized context!*
- If many threads are waiting on the same object, only one will be chosen (in no guaranteed order) to proceed with its execution.
- If there are no threads waiting, then no particular action is taken

notifyAll()

- This notifies all waiting threads and they start competing to get the lock. As the lock is used and released by each thread, all of them will get into action without a need for further notification.
- Object can have many threads waiting on it, and using notify() will affect only one of them.
- Which one exactly is not specified and depends on the JVM implementation,
- never rely on a particular thread In cases in which there might be a lot more waiting, the best way to do this is by using notifyAll().

Notify -Example

```
class BankAccount  {

private double balance=2000;

public synchronized double deposit(double amount){

balance=balance+amount;

notify();

return balance;
}
```

Notify -Example

```
public synchronized double withdraw(double amount)
{
if(balance <amount) {
System.out.println("In sufficent Balance ");
   try {

     wait();
} catch (InterruptedException e) {
              e.printStackTrace();
         }
}

 balance =balance-amount;

return balance;
}
```

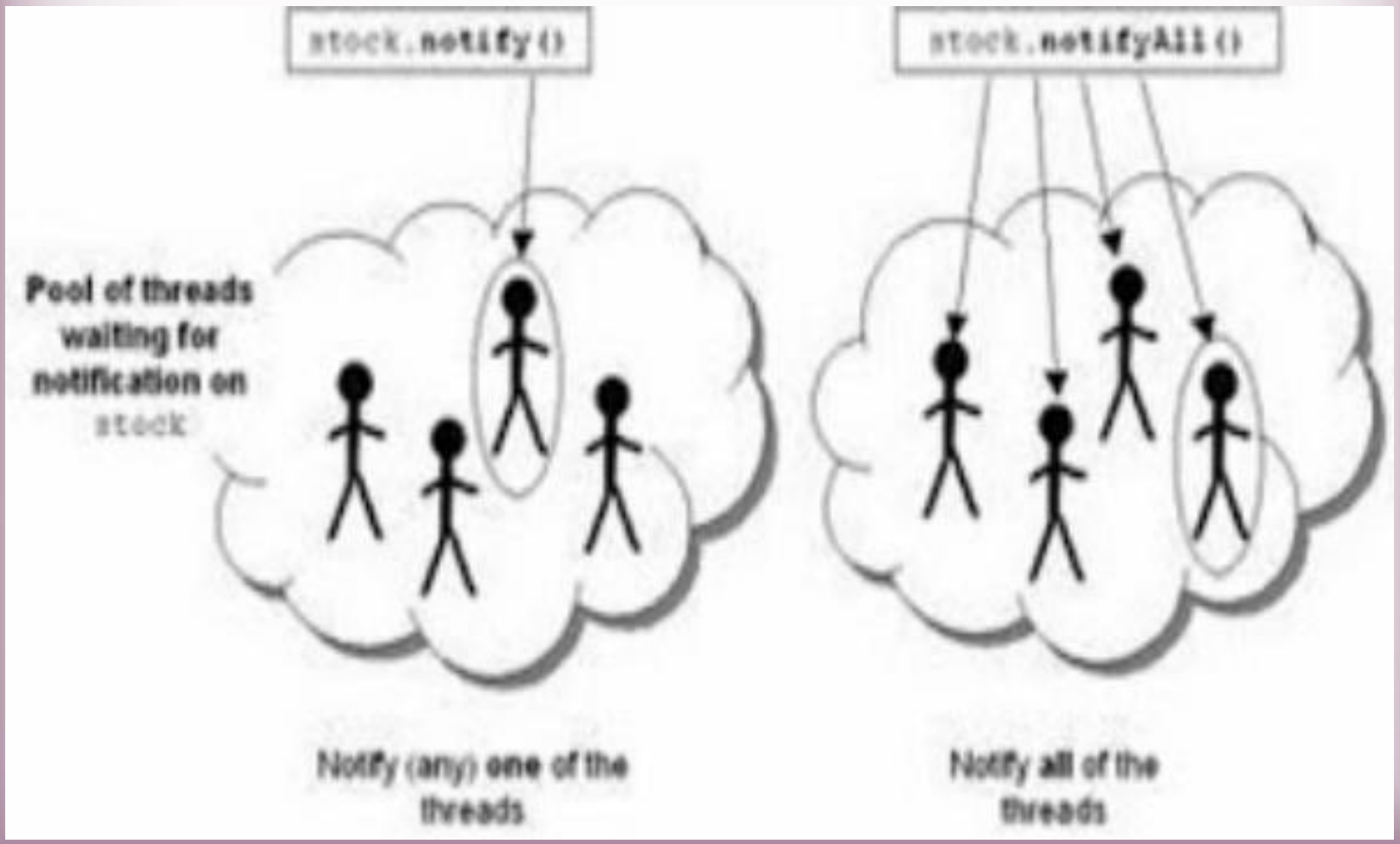
Notify -Example

```
public static void main(String[] args) {

final BankAccount  bk1 =new BankAccount();

Thread t1=new Thread(){
public void run(){
bk1.withdraw(4000);
}   };
t1.start();
Thread t2=new Thread(){
   public void run(){
          bk1.deposit(3000);
}
};
t2.start();
}
```

Notify and notifyAll()

- **notify**
- ***Object can have many threads waiting on it, and using notify() will affect only one of them.***
  - Which one exactly is not specified and depends on the JVM implementation,
  - The thread that receives notification may not able to proceed and starts waiting again application cannot progress further.
  - May result in deadlock if the thread which calls notify() goes into waiting for state after sending a notification because then there is no active thread to notify all waiting threads.
- **notifyAll()**
  - A better option than notify()
  - It sends a notification to all the threads.
  - If one is not able to proceed, there is still some more threads to do the job

Notify and notifyAll()



Notify -Example

```
class BankAccount  {

private double balance=2000;

public synchronized double deposit(double amount){

balance=balance+amount;

notify();

return balance;
}
```

Notify -Example

```
public synchronized double withdraw(double amount)
{
if(balance <amount) {
System.out.println("In sufficent Balance ");
   try {

     wait();
} catch (InterruptedException e) {
              e.printStackTrace();
         }
}

 balance =balance-amount;

return balance;
}
```

Notify -Example

```
public static void main(String[] args) {

final BankAccount  bk1 =new BankAccount();

Thread t1=new Thread(){
public void run(){
bk1.withdraw(4000);
}   };
t1.start();
Thread t2=new Thread(){
   public void run(){
          bk1.deposit(3000);
}
};
t2.start();
}
```

EXECUTOR FRAMEWORK

Executor Framework

- Released with the JDK 5 as part of java.util.concurrent package
- Helps to **decouple a command submission from command execution**.
- **Thread Creation**
  - **Has methods for creating a pool of threads to run tasks concurrently.**
- **Task submission and execution**
  - **Has methods for submitting tasks for execution in the thread pool**
  - **Can submit a task to be executed or schedule them to be executed later**
  - **Can also make them execute periodically.**

Executors framework

- **Executors**
  - Provides Factory methods to create Thread Pools of worker threads.
  - Tasks are submitted for execution.
  - Schedules and executes the submitted tasks
  - Return the results from the thread pool.

- **ExecutorService**
  - A sub interface of Executor
  - Has methods to manage lifecycle of threads
  - Used for running the submitted tasks
  - Has methods that produce a Future to get a result from an asynchronous computation.

Executor Service

- Has methods to start and terminate thread.

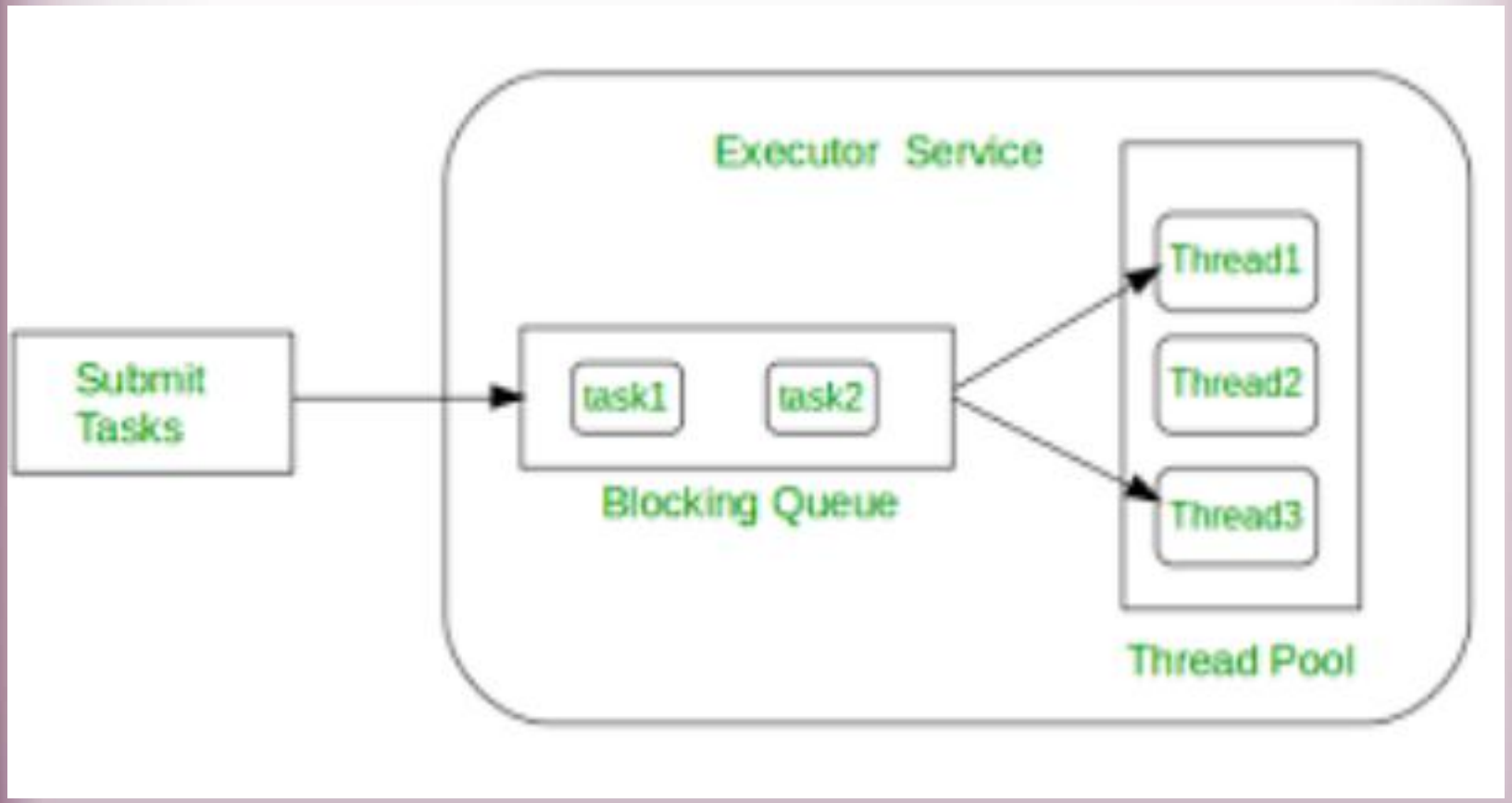
- **execute()**
  - Used for threads which are Runnable
- **submit()**
  - Used for Callable threads.
- **shutdown()**
  - Stops accepting new tasks, waits for previously submitted tasks to execute,
  - Then terminates the executor.
- **shutdownNow()**
  - Interrupts the running task and shuts down the executor immediately.

Thread Pool

- A pool of worker threads
  - Executors use *thread pools* to execute tasks.
  - Executors Utility class provides different methods to create thread pool.

- **Need for Thread Pool**
  - Creating a new thread for a new task leads to overhead of thread creation and tear-down.
    - Managing thread life-cycle adds to the execution time.
  - New thread for each process without any throttling leads to the creation of many threads.
    - They occupy memory and cause wastage of resources.

Working of Executors

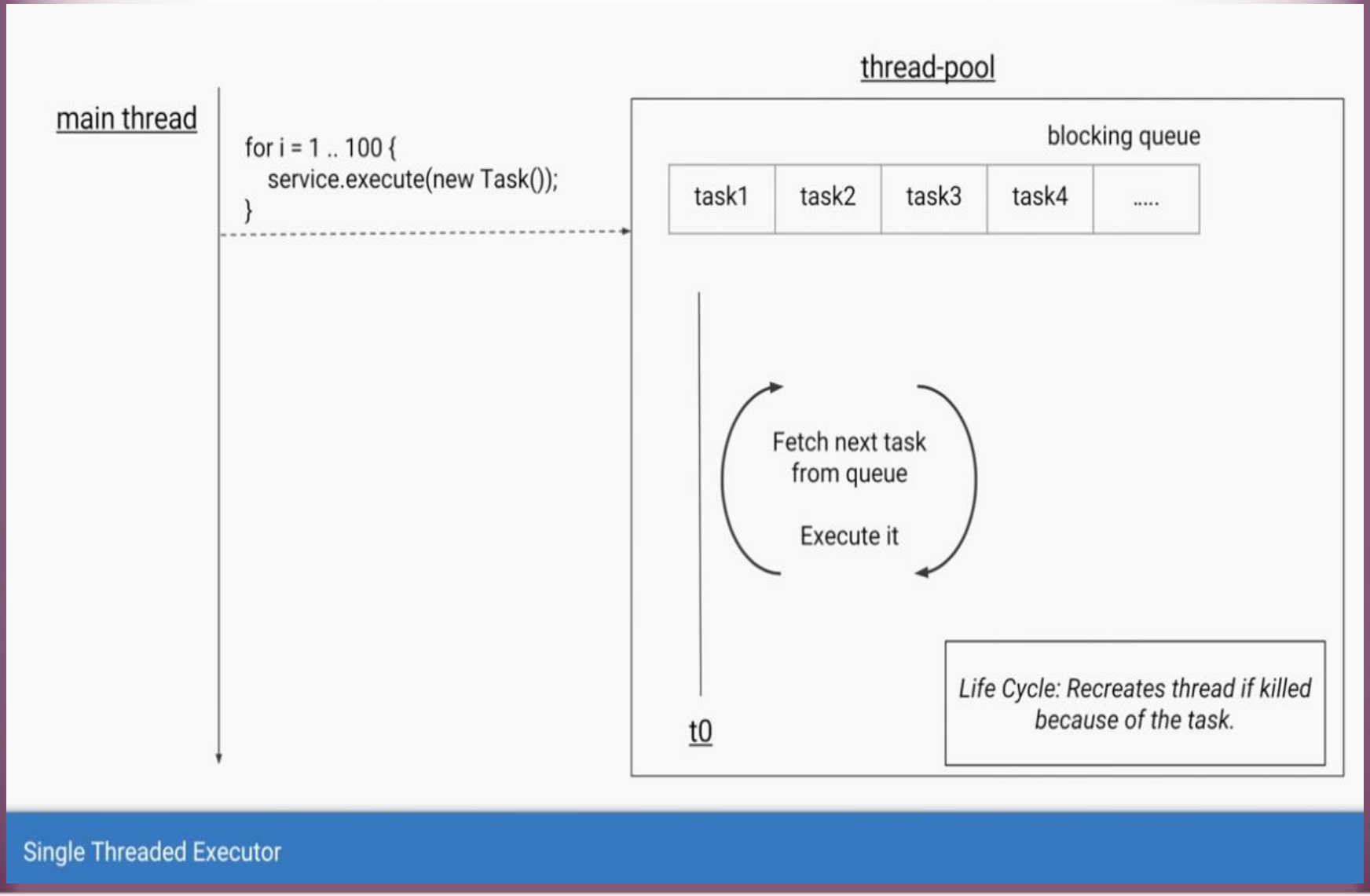


Types of Executors –Single Thread Executor

- **newSingleThreadExecutor()**
  - Executor has only a single thread.
  - Used to execute tasks in a sequential manner.
  - When the submitted task is in execution
    - The new task will wait in a queue until the thread is free to execute it.
    - If the thread dies due to an exception while executing a task, a new thread is created to replace the old thread and the subsequent tasks are executed in the new one.

```
ExecutorService executorService =
                    Executors.newSingleThreadExecutor()
```

Single Threaded Executor



Task

```
public class PrintNames {

public synchronized static void print(String
str1,String str2) {

 System.out.println(str1);

 try {
Thread.sleep(200);
} catch (InterruptedException e) {
e.printStackTrace();
}
 System.out.println(str2);
}
}
```

Runnable

```
public class ExecutorPrintService implements Runnable {

private String str1;
private String str2;

public ExecutorPrintService(String str1, String str2) {
super();
this.str1 = str1;
this.str2 = str2;
}
@Override
public void run() {
PrintNames.print(str1, str2);
}
}
```

Submit to Executor

```
public static void main(String[] args) {

ExecutorService executor =
Executors.newSingleThreadExecutor();

ExecutorPrintService tasks[] = {
 new ExecutorPrintService("Idly","sambar"),
         new ExecutorPrintService("Parantha","achar"),
         new ExecutorPrintService("pizza","coke")};

for(int i=0;i<tasks.length;i++) {
executor.submit(tasks[i]);
}
executor.shutdown();
}
```

Runnable Task

```
public class NameManager implements Runnable {

private List<String> list;
private List<String> nameToAdd ;

public NameManager(List<String> list,String ... names) {
    this.list = list;
    this.nameToAdd = Arrays.asList(names);
}

public List<String> getList() {
    return list;
}
@Override
public void run() {
this.list.addAll(nameToAdd);
  }
}
```

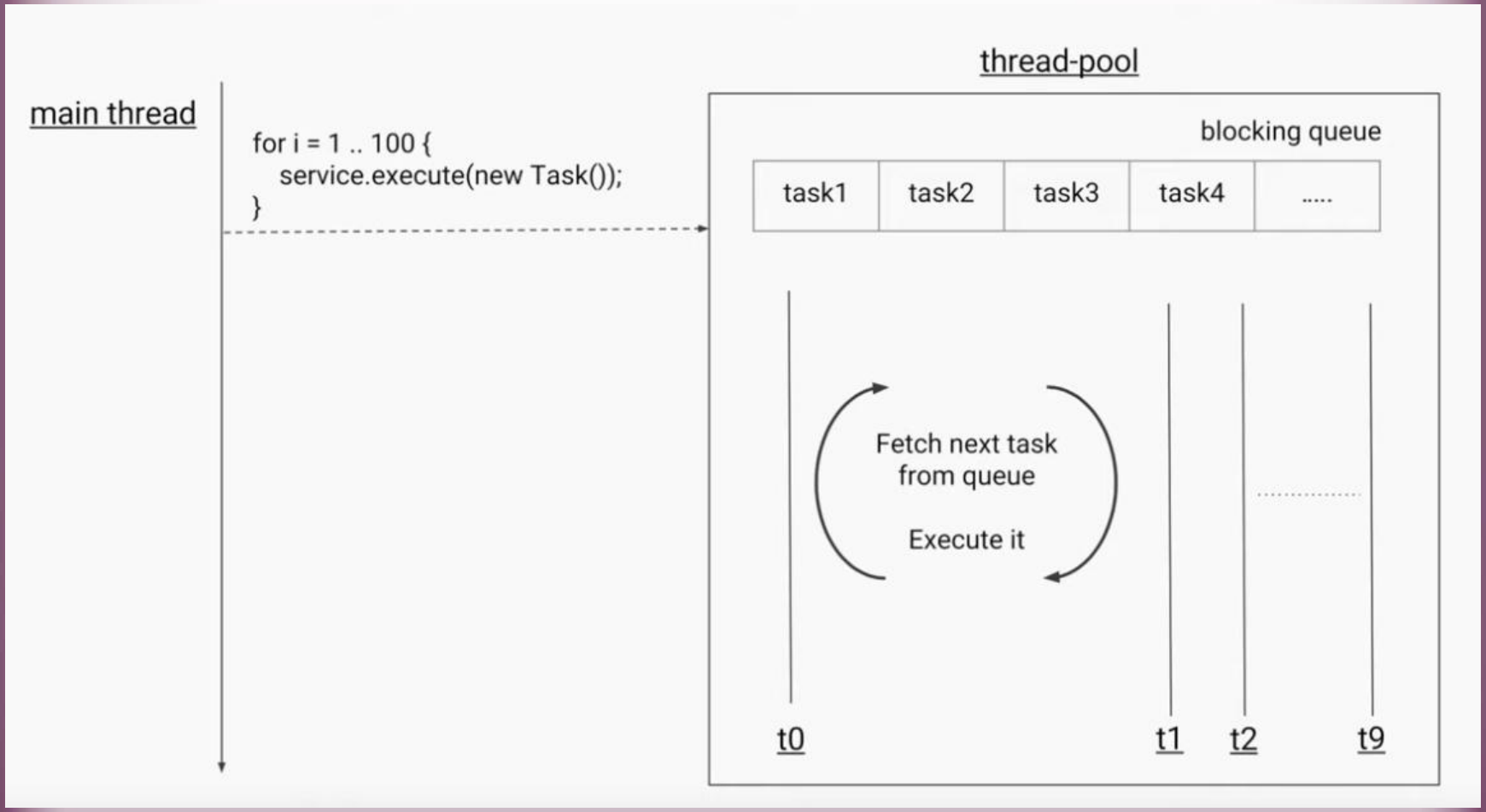
Single Thread Executor

```
ExecutorService service =
                    Executors.newSingleThreadExecutor();

List<String> list = new ArrayList<>();

NameManager task1 =new NameManager(list,"Ram",“Vijay",”Vicky");

NameManager task2 =new NameManager(list,"Vidya","vani","Rani");

 service.submit(task1);
 service.submit(task2);

 task1.getList().forEach(System.out::println);

service.shutdown();
```

Executors.newFixedThreadPool

- Create a thread pool of fixed size,
- Executor service picks one of the available threads from the pool and executes the task on that thread.
- Makes sure that the pool always has the specified number of threads running.
- If any thread dies it is replaced by a new thread immediately.
- If all the threads are busy , the new tasks will wait for their turn in a queue.

Fixed Thread Pool



Submit to Executor

```
public static void main(String[] args) {

 int poolSize = Runtime.getRuntime().availableProcessors();

ExecutorService executor =
Executors.newFixedThreadPool(poolSize);

ExecutorPrintService tasks[] = {
 new ExecutorPrintService("Idly","sambar"),
         new ExecutorPrintService("Parantha","achar"),
         new ExecutorPrintService("pizza","coke")};
for(int i=0;i<tasks.length;i++) {
executor.submit(tasks[i]);
}
executor.shutdown();
}
```

Size of the Fixed Thread Pool

- Decided based on the use-case of the java program.
- It also depends on number of CPU cores

- **CPU Bound**
  - Tasks which involves mathematical calculations.
  - Ideal size is same as number of CPU cores.

- **IO Bound**
  - Tasks which involve communication to other application **through network calls** like database, web services
  - Ideal size depends on the wait time of the IO task.
  - More number of threads can be used to ensure maximum CPU utilization.

SingleThreadExecutor vs FixedThreadPool

- **Similarity**
  - Returns ExecutorService
  - If thread terminates, new thread will be created.
- **Difference**
  - newSingleThreadExecutor()
    - can never increase its thread pool size more than one.
  - newFixedThreadPool(1)
    - (1) stands for the number of threads in the pool
    - can increase its thread pool size more than one at run time
  - Done by setCorePoolSize() of the class **ThreadPoolExecutor**.

Difference between execute and submit

- execute
  - Starts the task
  - Used for fire and forget calls
  - Throws un-handled Exception.
- submit
  - returns a Future object to manage the task.
  - Can be Used to submit a Callable
  - Callable can return a value
  - Hides un-handled Exception in framework itself.
  - Can Wait for the Callable task to finish executing
  - Can get the return value on completion , with get.

CALLABLE

Callable

- Similar to Runnable
- Can be used with an executor service for execution.
- A functional Interface with one Method

- **call()**
  - Method contains the code that is executed by a thread.
  - Can return a result and throws a checked exception.

Callable With Executor

- **submit()**
  - submits the task for execution by a thread.
  - returns a java.util.concurrent.Future

- *A task submitted to the executor*
  - Submitted task must implement Callable.
  - *Whenever the task execution is completed, it is set to Future object by the executor.*

Future

- **Future**
  - Object returned by the executor.
  - The result of the task submitted for execution to an executor
  - Its like a promise made to the caller by the executor.

- **Future.isDone**
  - Returns a boolean value true if the task completed.

Future.get()

- **Future.get()**
  - Waits for the computation to complete, and then retrieves its result.
  - This method is used to get the result of the submitted task when its completed

- **get(long timeout, TimeUnit unit)**
  - Throws a TimeoutException if the result is not returned in the stipulated timeframe.
  - The caller can handle this exception and continue with the further execution of the program.

Cancelling a Future

- **Future.cancel(boolean mayInterruptIfRunning)**
  - Used to cancel a future
  - returns true if it is cancelled successfully, otherwise, it returns false.

- **mayInterruptIfRunning.**
  - True
    - the thread that is currently executing the task will be interrupted
  - False
    - in-progress tasks will be allowed to complete.

- **Future.isCancelled()**
  - To check if a task is cancelled or not.

Callable Example

```
public class Task implements Callable<String> {

    private String message;

    public Task(String message) {
        this.message = message;
    }

    @Override
    public String call() throws Exception {
        return "Hello " + message + "!";
    }
}
```

Callable Example

```
public class ExecutorExample {
    public static void main(String[] args) {

        Task task = new Task("World");

        ExecutorService executorService =
Executors.newFixedThreadPool(4);
        Future<String> result = executorService.submit(task);

        try {
            System.out.println(result.get());
        } catch (InterruptedException | ExecutionException e) {
            System.out.println("Error occured while executing the submitted
task");
            e.printStackTrace();
        }
```

Callable Example

```
public class FactorialCalculator implements Callable<Integer> {
    private Integer number;

    public FactorialCalculator(Integer number) {
        this.number = number;        }

    @Override
    public Integer call() throws Exception {
        int result = 1;
        if ((number == 0) || (number == 1)) {
            result = 1;
        } else {
            for (int i = 2; i <= number; i++) {
                result *= i;
                TimeUnit.MILLISECONDS.sleep(20);
            }
        }
        return result;
    } }
```

Callable Example

```
ExecutorService executor =  Executors.newFixedThreadPool(2);

    FactorialCalculator calculator  = new FactorialCalculator(5);

  try  {

   Future<Integer> task = executor.submit(calculator);

System.out.println("Future result is - " + " - " + task.get() + "; And Task
 done is " + task.isDone());
        }
        catch ( ExecutionException | InterruptedException e)
        {
            e.printStackTrace();
        }
            executor.shutdown ();
```

Cancelling the Future

```
public static void main(String[] args) {

ExecutorService executorService = Executors.newSingleThreadExecutor();

long startTime = System.nanoTime();
Future<String> future = executorService.submit(() -> {
Thread.sleep(2000); return "Hello from Callable"; });

while(!future.isDone()) {
System.out.println("Task is still not done...");
try {
Thread.sleep(200);
double elapsedTimeInSec = (System.nanoTime() - startTime)/1000000000.0;
if(elapsedTimeInSec > 1) {
future.cancel(true);
}
} catch (InterruptedException e) {
e.printStackTrace();   }
```

Cancelling the Future

```
System.out.println("Task completed! Retrieving the result");
String result;
try {
result = future.get();
System.out.println(result);
} catch (InterruptedException e) {
e.printStackTrace();
} catch (ExecutionException e) {
e.printStackTrace();
}
executorService.shutdown();
}
}
```

Cancelling the Future

```
if(!future.isCancelled()) {

     System.out.println("Task completed! Retrieving the result");
    String result = future.get();
    System.out.println(result);

}  else {
    System.out.println("Task was cancelled");
}
```