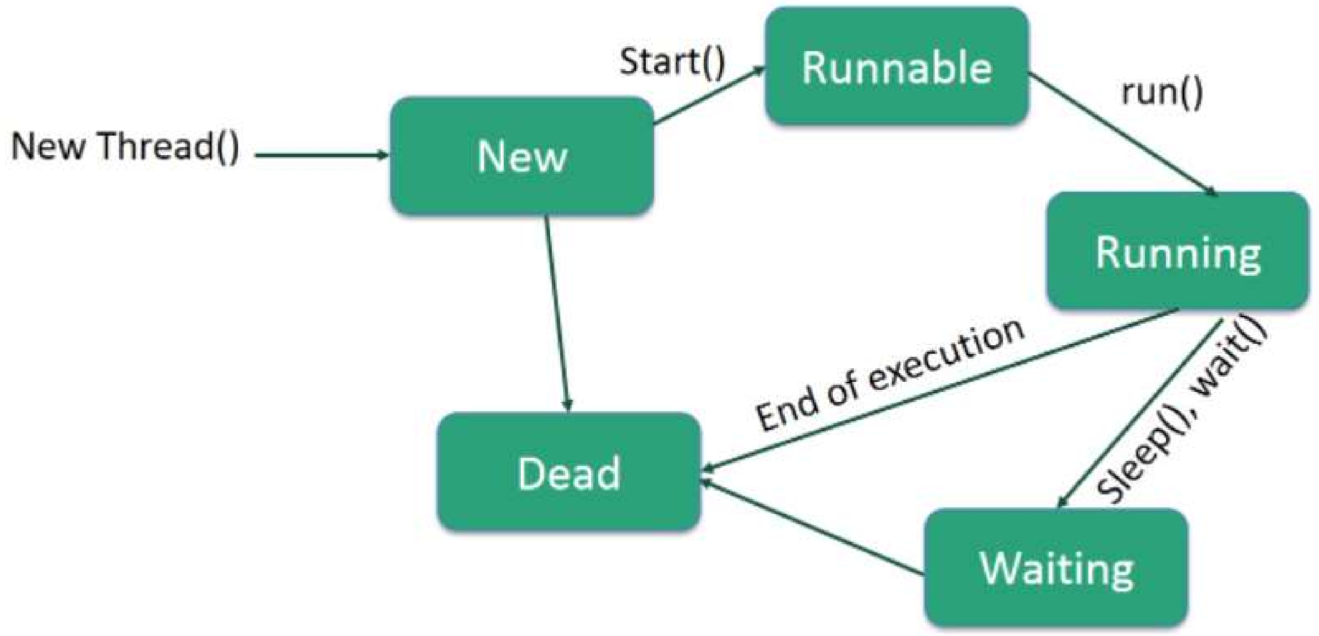
Concurrency in Java

Principles of Programming Languages

Recap of Course- Object Oriented Paradigm

On Threads

Life Cycle of a Thread



Thread Priorities

- Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

- Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

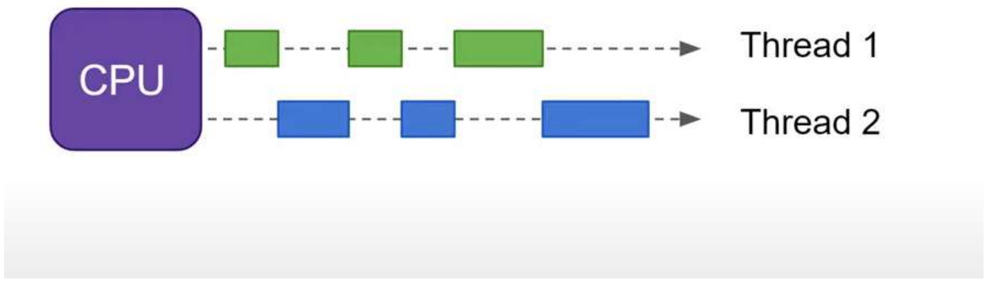
- Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Thread Creation in Java

- Create a Thread by Implementing a Runnable Interface
- Create a Thread by Extending a Thread Class

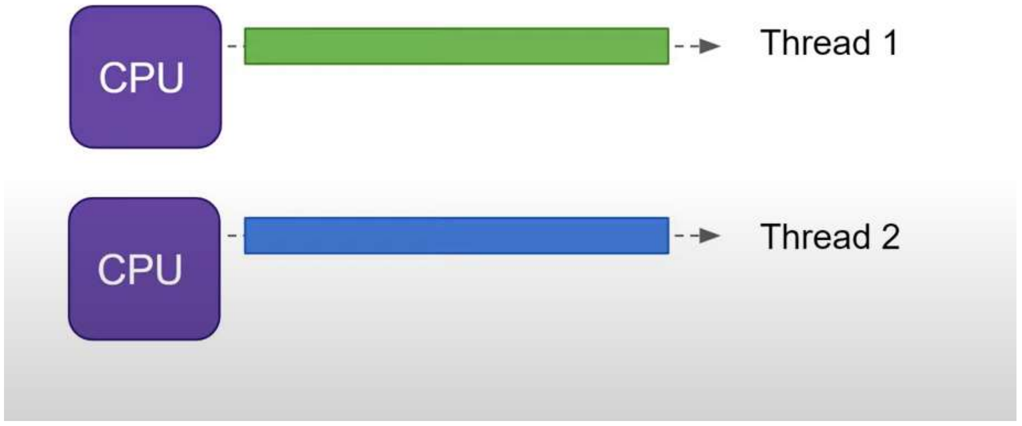
Concurrency

Making progress on more than one task – seemingly at the same time



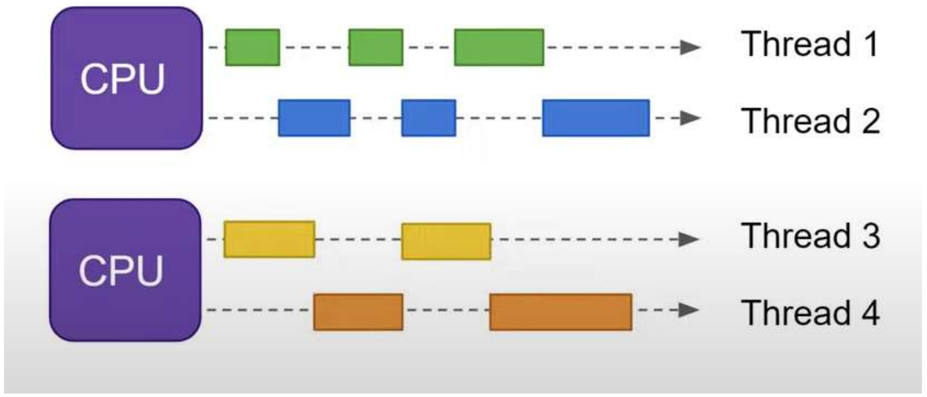
Parallel Execution

Making progress on more than one task at the exact same time



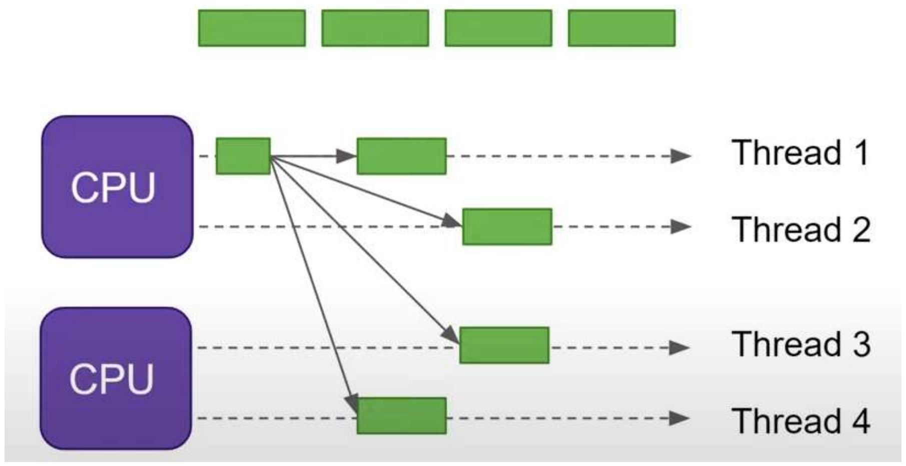
Parallel Concurrent Execution

Making progress on more than one task- seemingly at the same time – on more than one CPU.



Parallelism

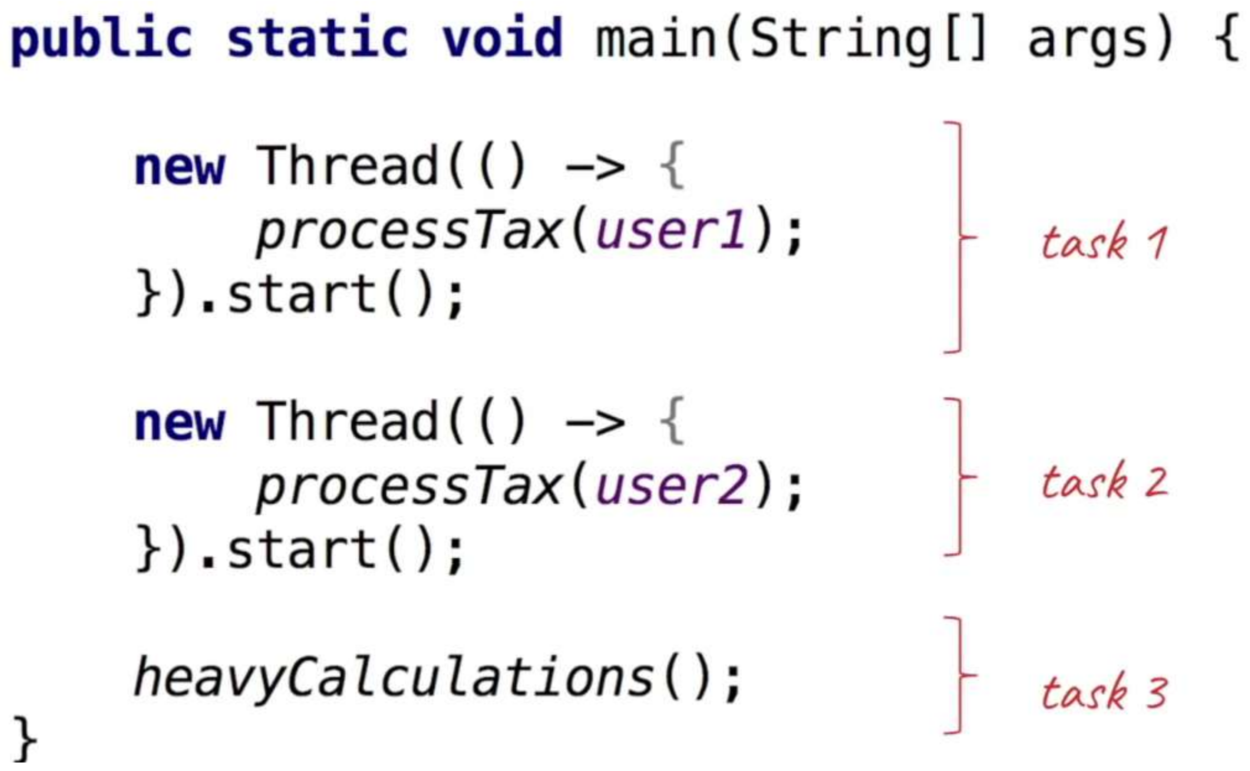
Splitting a single task into subtasks which can be executed in parallel



Combinations

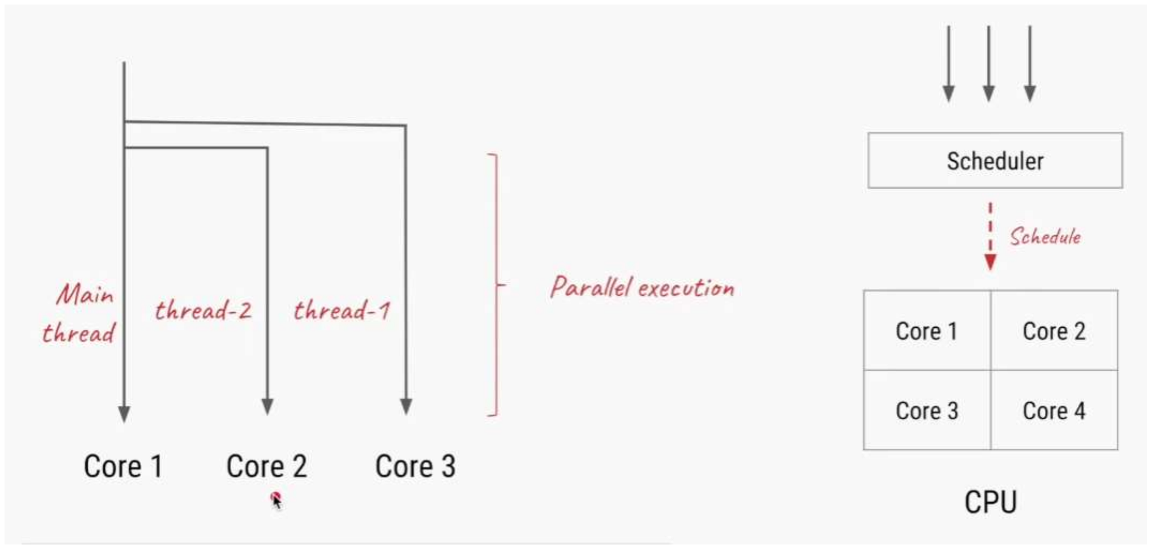
- Concurrent, Not Parallel
- Parallel, Not Concurrent
- Both Concurrent and Parallel
- Neither Concurrent, Nor Parallel

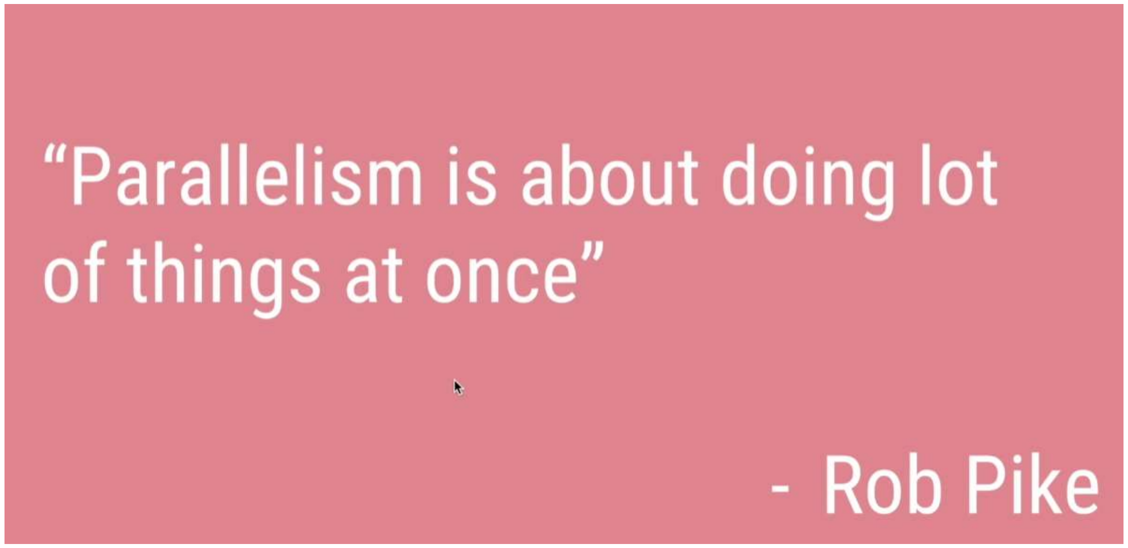
Parallelism



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

    new Thread(() -> {
        processTax(user1);
    }).start();

    new Thread(() -> {
        processTax(user2);
    }).start();

    heavyCalculations();
}
```

Parallelism

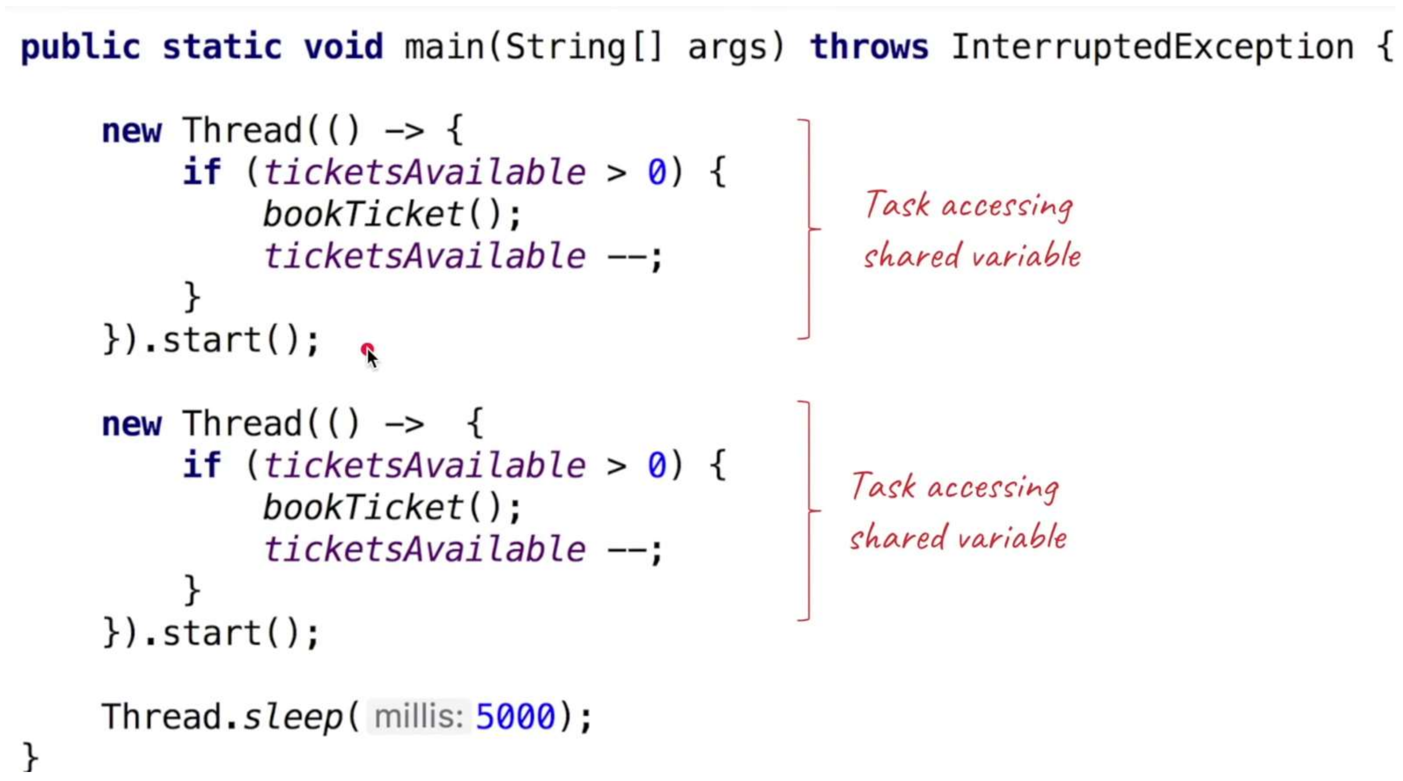




“Parallelism is about doing lot of things at once”

- Rob Pike

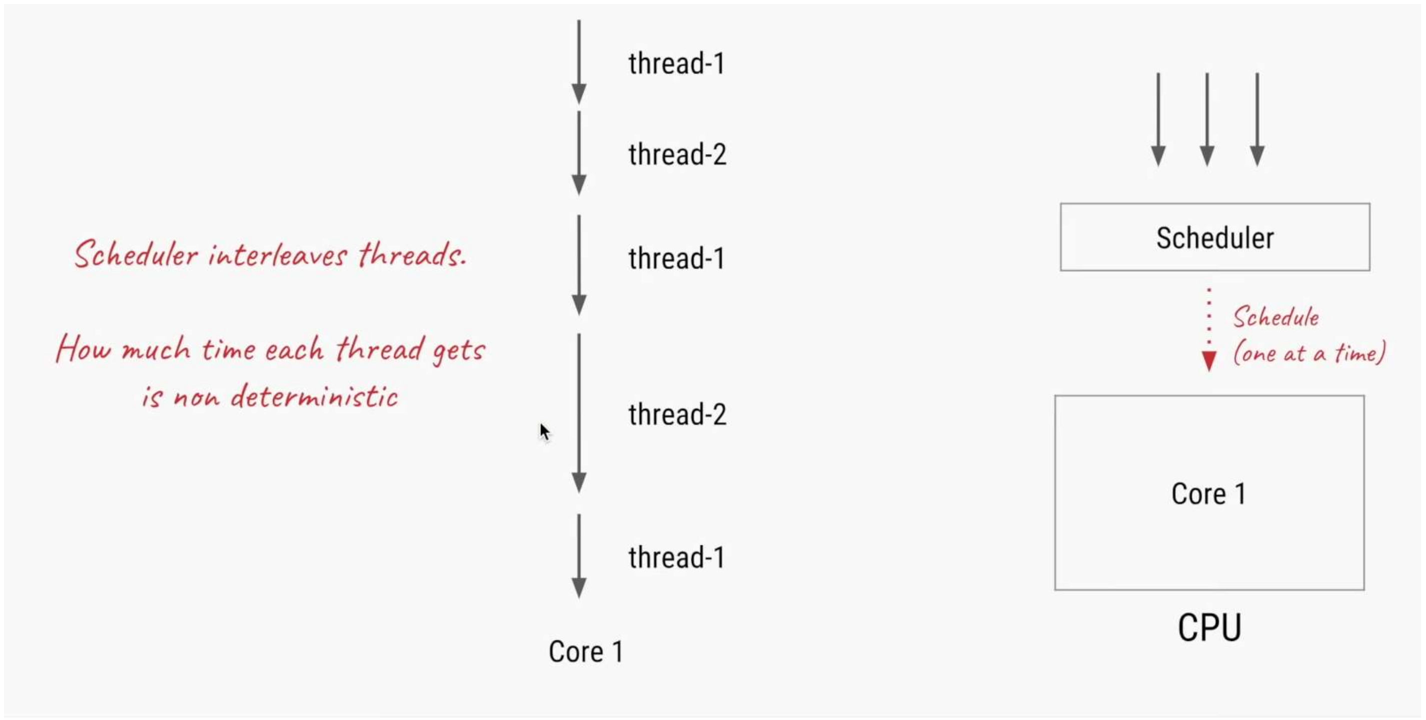
Concurrency



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

    new Thread(() -> {
        if (ticketsAvailable > 0) {
            bookTicket();
            ticketsAvailable --;
        }
    }).start();

    new Thread(() ->  {
        if (ticketsAvailable > 0) {
            bookTicket();
            ticketsAvailable --;
        }
    }).start();

    Thread.sleep( millis: 5000);
}
```

Task accessing shared variable
Task accessing shared variable

Concurrency

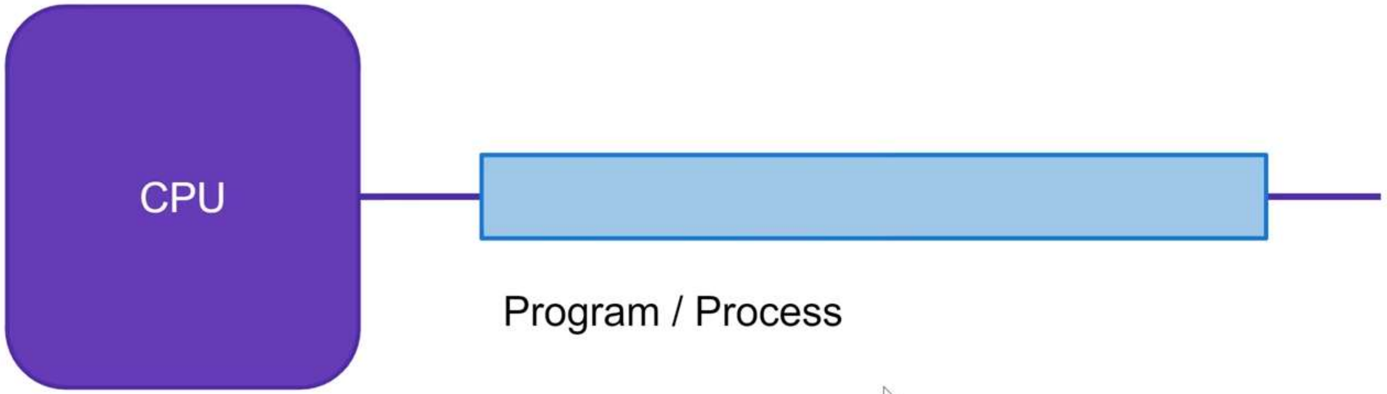


Threads

Process Vs Threads

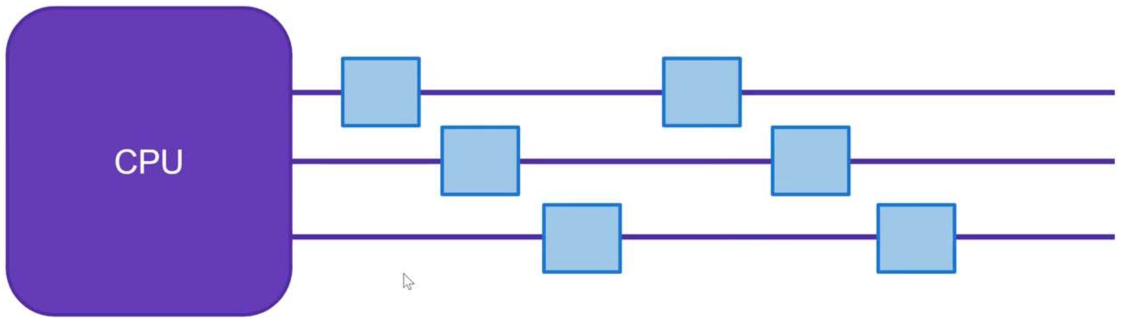
- A process runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process, e.g., memory and CPU time, are allocated to it via the operating system.
- A thread is a so-called lightweight process. It has its own call stack but can access shared data of other threads in the same process. Every thread has its own memory cache. If a thread reads shared data, it stores this data in its own memory cache.
- A thread can re-read the shared data.
- A Java application runs by default in one process. Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.

Single tasking in Early Computing



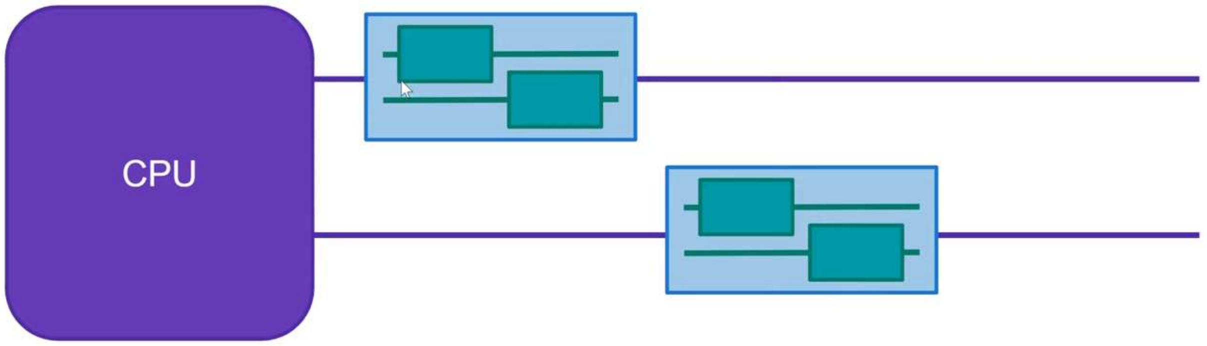
One CPU / Computer can run
one program (process)
at a time.

Multitasking in Early Computing



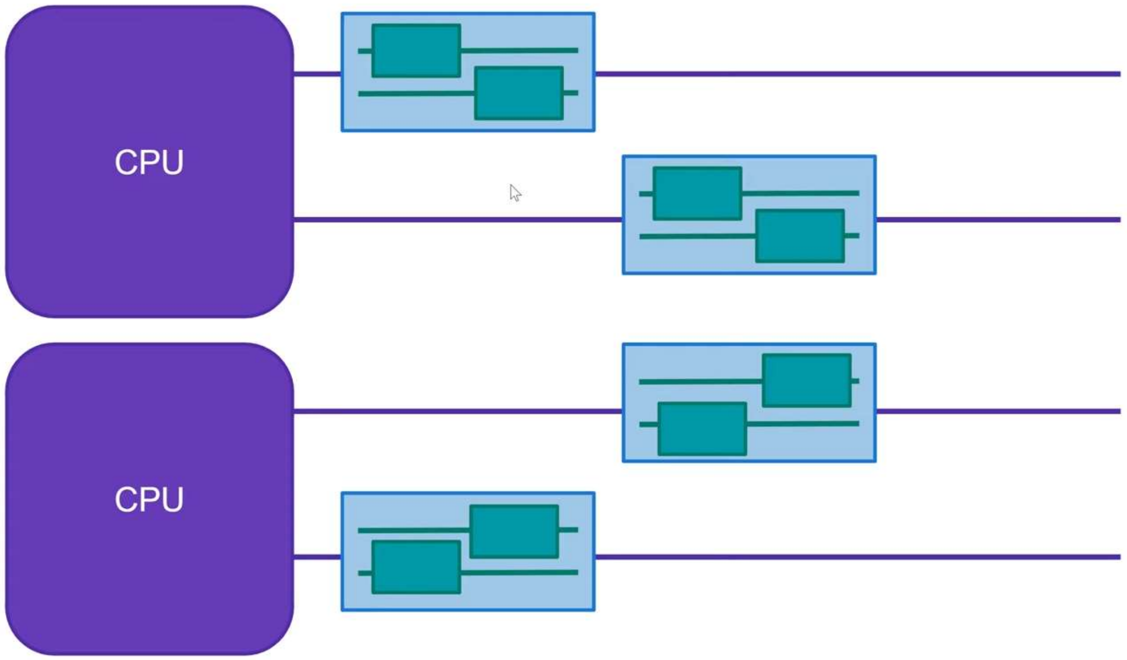
One CPU / Computer can run
Multiple programs (process)
at a time - by switching between
executing one program at a time for a
little time, and then switch to the next.

Multithreading



One CPU / Computer can run
Multiple programs (process)
at a time, with multiple threads of
execution inside.

Multithreading with Multiple CPUs

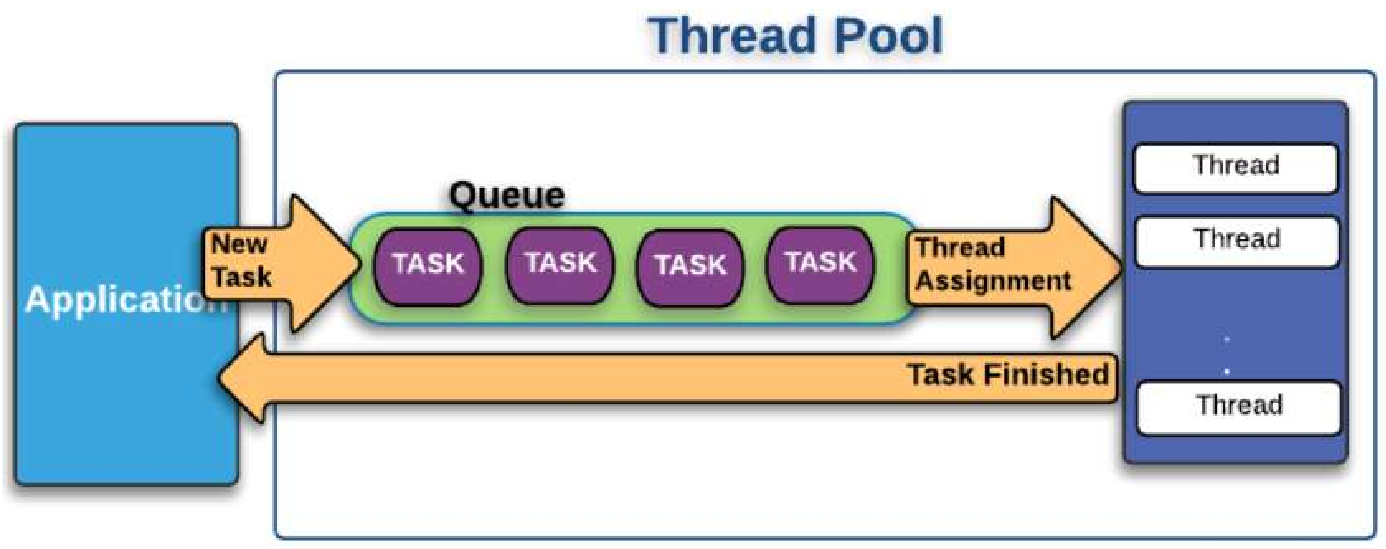


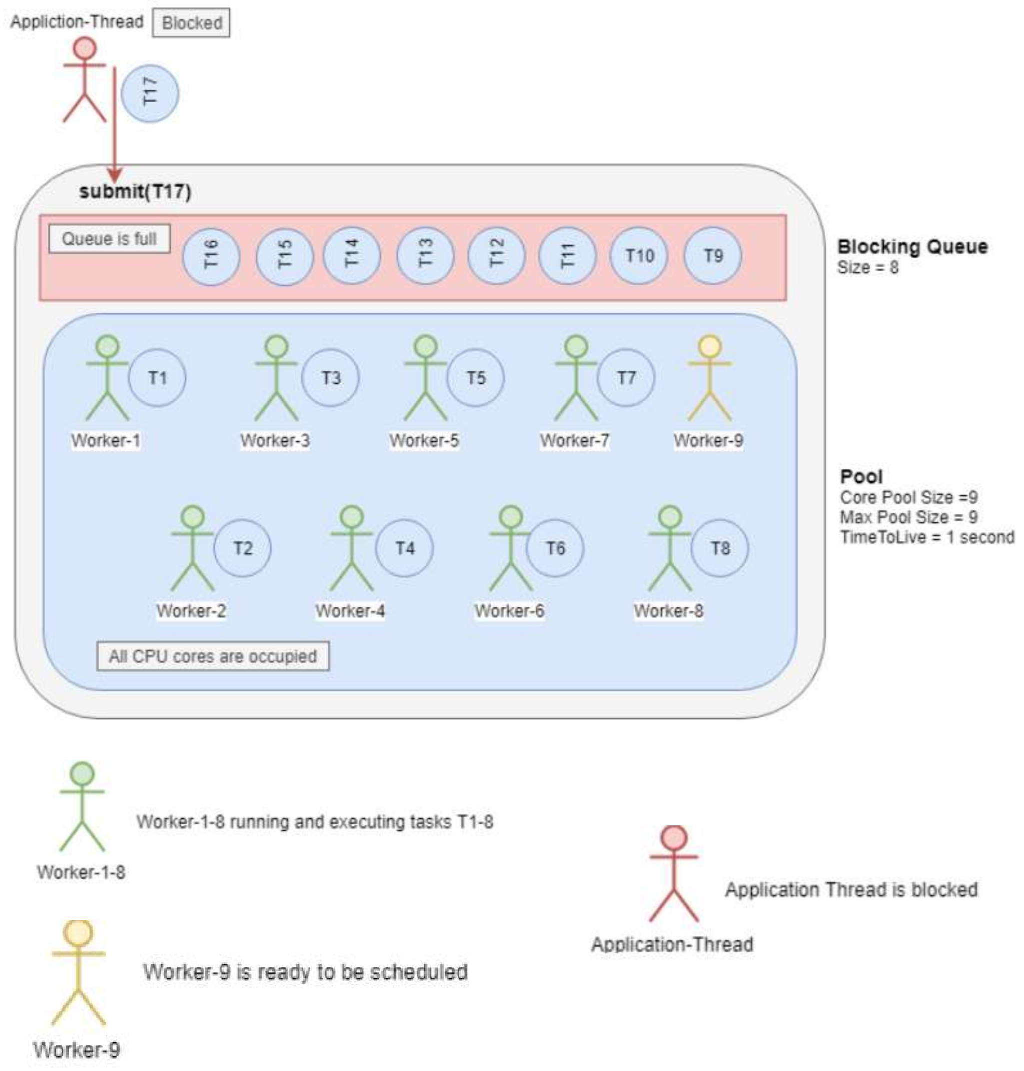
What is thread pool?

- A thread pool is a collection of pre-initialized threads.
- Generally, the size of the collection is fixed, but it is not mandatory.
- It facilitates the execution of N number of tasks using the same threads.
- If there are more tasks than threads, then tasks need to wait in a queue like structure (FIFO – First in first out).
- When any thread completes its execution, it can pickup a new task from the queue and execute it. When all tasks are completed, the threads remain active and wait for more tasks in the thread pool.

Thread Pool

A watcher keep watching queue (usually BlockingQueue) for any new tasks. As soon as tasks come, threads again start picking up tasks and execute them.





ThreadPool

1. Application thread submitted 16 tasks to the thread pool
2. 8 worker threads in the pool have taken 8 tasks from the queue and are executing the tasks
3. Remaining tasks are queued as workers are busy
4. The 9th worker thread is ready to be scheduled. But it cannot run yet as all the CPU cores are occupied by Workers 1-8
5. Remaining 8 tasks are queued in the Blocking Queue. Effectively, the application is allowed to submit 8+8=16 tasks to the thread pool. After this application thread is blocked.
6. When application thread tried submitting the 17th task, application thread is blocked by the queue as the queue is full

ThreadPoolExecutor

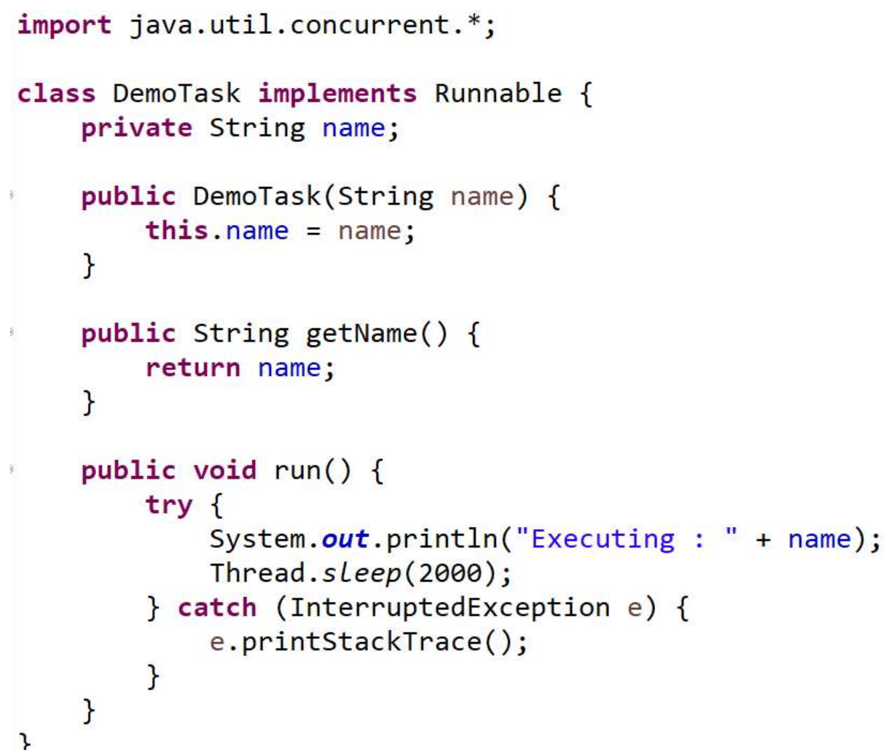
- Since Java 5, the Java concurrency API provides a mechanism Executor framework. This is around the Executor interface, its sub-interface ExecutorService, and the ThreadPoolExecutor class that implements both interfaces.
- ThreadPoolExecutor separates the task creation and its execution. With ThreadPoolExecutor, you only have to implement the Runnable objects and send them to the executor. It is responsible for their execution, instantiation, and running with necessary threads.
- The Executor framework relieved Java application developers from the responsibility of creating and managing threads.

How to create ThreadPoolExecutor

We can create following 5 types of thread pool executors with pre-built methods in java.util.concurrent.Executors interface.

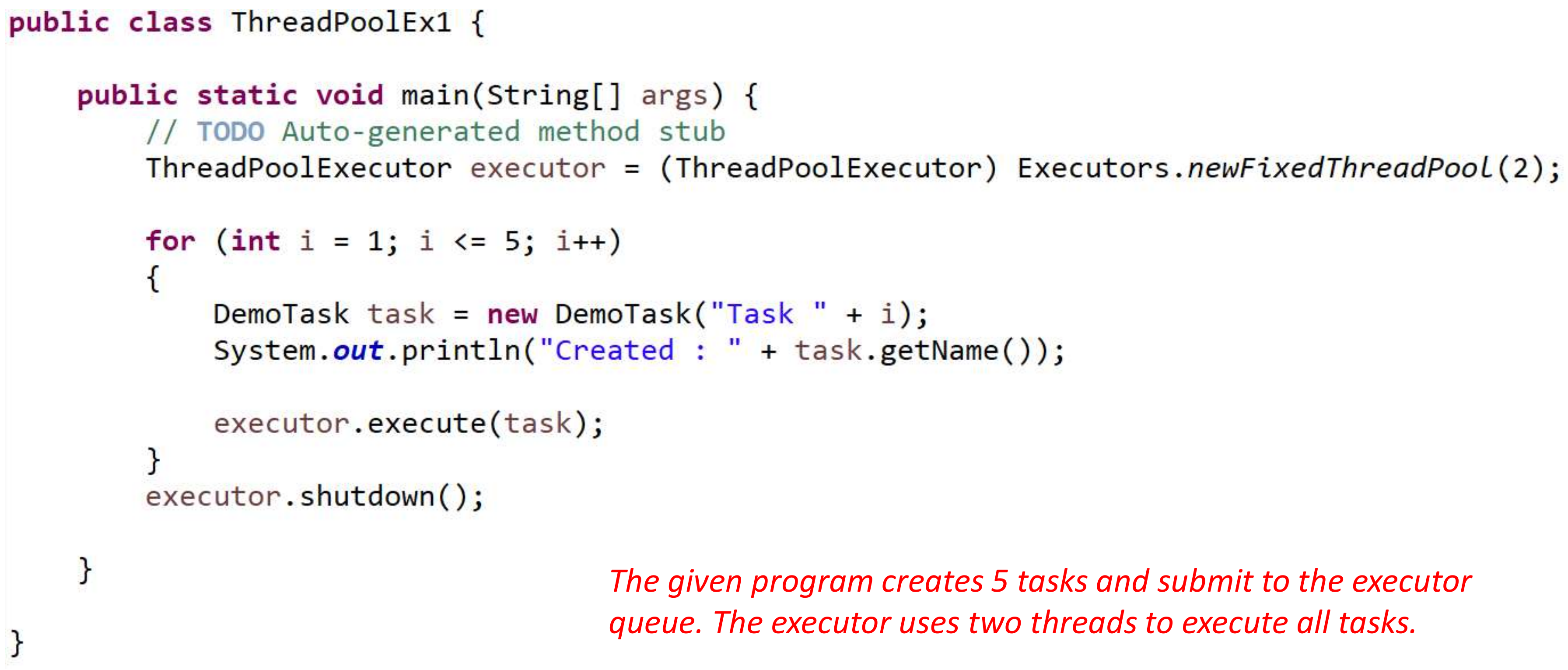
1. Fixed thread pool executor
2. Cached thread pool executor
3. Scheduled thread pool executor
4. Single thread pool executor
5. Work stealing thread pool executor

Example – Create task



```
import java.util.concurrent.*;

class DemoTask implements Runnable {
    private String name;

    public DemoTask(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void run() {
        try {
            System.out.println("Executing : " + name);
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Example - Execute tasks with thread pool executor

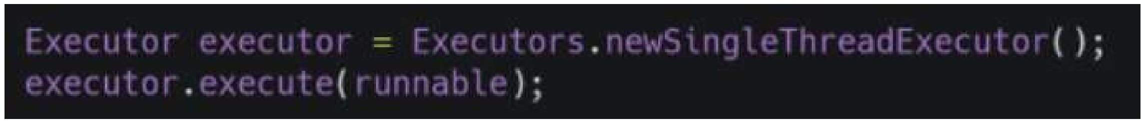


```java
public class ThreadPoolEx1 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(2);

        for (int i = 1; i <= 5; i++)
        {
            DemoTask task = new DemoTask("Task " + i);
            System.out.println("Created : " + task.getName());

            executor.execute(task);
        }
        executor.shutdown();

    }

}
```

The given program creates 5 tasks and submit to the executor queue. The executor uses two threads to execute all tasks.

Executor Framework

- The Executor framework is the official Java support for ThreadPools.



```
Executor executor = Executors.newSingleThreadExecutor();
executor.execute(runnable);
```

- We are asking the executor to execute a runnable task. Creating the thread that runs the task and its management is now delegated to the framework.
- Since we told the executor to execute the tasks submitted to it using only one thread — newSingleThreadExecutor(), it will make sure that one thread (worker thread) is always available for asynchronous task execution.
- If the worker thread terminates due to exceptional conditions, Executors.newSingleThreadExecutor() will create a new one automatically.

Difference between Executor, ExecutorService and Executors class

- All three classes Executor, ExecutorService, and Executors are part of Java's Executor framework which provides thread pool facilities to Java applications.
- Since the creation and management of Threads are expensive and the operating system also imposes restrictions on how many Threads an application can spawn, it's a good idea is to use a pool of threads to execute tasks in parallel, instead of creating a new thread every time a request comes in.
- This not only improves the response time of the application but also prevent resource exhaustion errors like "java.lang.OutOfMemoryError: unable to create new native thread".

Difference

- The main difference between Executor, ExecutorService, and Executors class is that Executor is the core interface which is an abstraction for parallel execution.
- It separates tasks from execution, this is different from java.lang.Thread class which combines both task and its execution.
- ExecutorService is an extension of the Executor interface and provides a facility for returning a Future object and terminate or shut down the thread pool. Once the shutdown is called, the thread pool will not accept new tasks but complete any pending task.
- ExecutorService also provides a submit() method which extends Executor.execute() method and returns a Future.

Difference

- One of the key differences between Executor and ExecutorService interface is that the former is a parent interface while ExecutorService extends Executor, i.e., it's a sub-interface of Executor.
- Executor defines execute() method which accepts an object of the Runnable interface, while submit() method can accept objects of both Runnable and Callable interfaces.
- The execute() method doesn't return any result, its return type is void but the submit() method returns the result of computation via a Future object.

Callable and Future

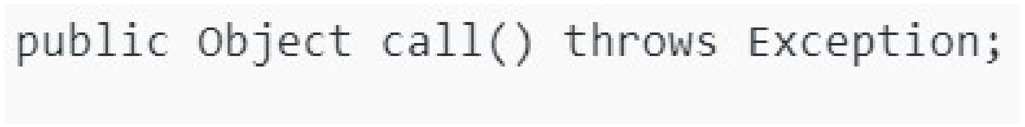
Note that a thread can’t be created with a Callable, it can only be created with a Runnable.

Runnable Vs Callable

- There are two ways of creating threads – one by extending the Thread class and other by creating a thread with a Runnable.
- However, one feature lacking in Runnable is that we cannot make a thread return result when it terminates, i.e. when run() completes.
- For supporting this feature, the Callable interface is present in Java.
- For implementing Runnable, the run() method needs to be implemented which does not return anything, while for a Callable, the call() method needs to be implemented which returns a result on completion.
- Another difference is that the call() method can throw an exception whereas run() cannot.

Callable

- Method signature that has to overridden for implementing Callable.



```
public Object call() throws Exception;
```

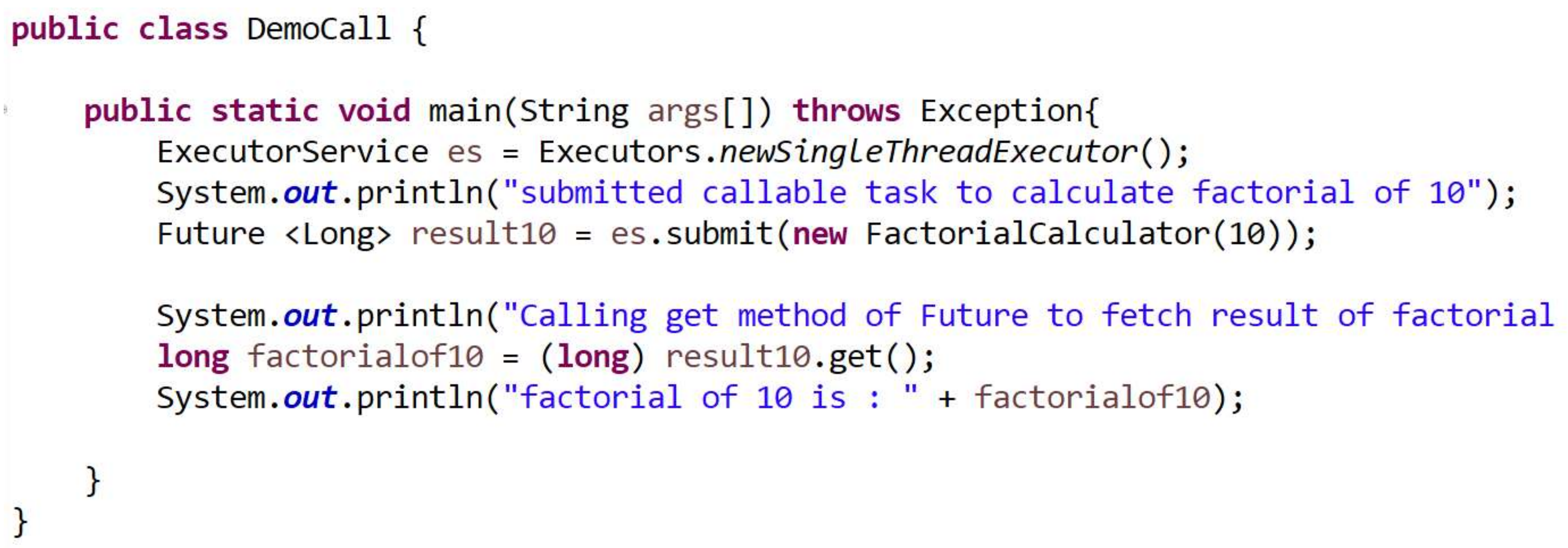
Callable and Future

- When you pass a Callable to a thread pool, it chooses one thread and executes the Callable.
- It immediately returns a Future object which promises to hold the result of computation once done.
- You can then call the get() method of Future, which will return the result of computation or block if the Computation is not complete.

Example

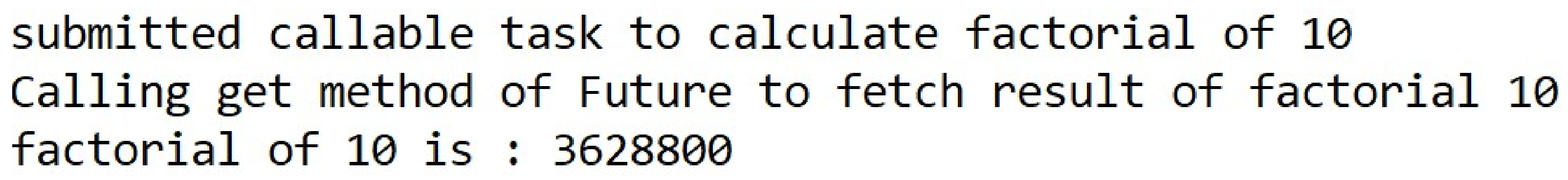
```java
class FactorialCalculator implements Callable<Long>
{
    private int number;
    public FactorialCalculator(int number)
    {
        this.number = number;
    }
    @Override
    public Long call() throws Exception
    { return factorial(number); }
    private long factorial(int n) throws InterruptedException
    {
        long result = 1;
        while (n != 0)
        {
            result = n * result; n = n - 1; Thread.sleep(100);
        }
        return result;
    }
}
```

Example



```
public class DemoCall {

    public static void main(String args[]) throws Exception{
        ExecutorService es = Executors.newSingleThreadExecutor();
        System.out.println("submitted callable task to calculate factorial of 10");
        Future <Long> result10 = es.submit(new FactorialCalculator(10));

        System.out.println("Calling get method of Future to fetch result of factorial
        long factorialof10 = (long) result10.get();
        System.out.println("factorial of 10 is : " + factorialof10);

    }
}
```

Example-O/P



```
submitted callable task to calculate factorial of 10
Calling get method of Future to fetch result of factorial 10
factorial of 10 is : 3628800
```