# Introduction to Threads

Mehrshad Saadatinia

# Contents

- Parallelism vs. Concurrency vs. Asynchronous
- What are Threads?
- Threads vs. Processes
- Threads in different programming languages
- Threads in Java
- Basic Threads methods
- Challenges of multi-threading

# • Parallelism vs. Concurrency vs. Asynchronous

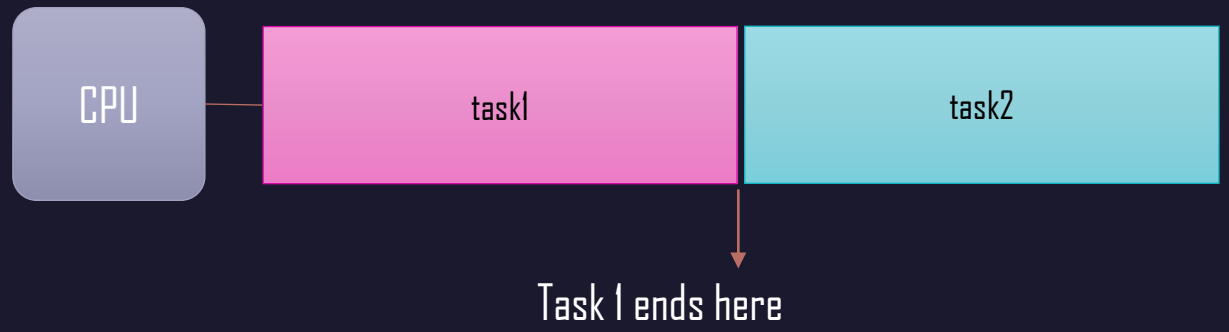
Can you run two tasks in parallel?

Is concurrent execution the same as parallel?

What is asynchronous then?

# Sequential execution

One task starts after the other finishes, like programs we wrote **so far**.



# Parallelism vs. Concurrency

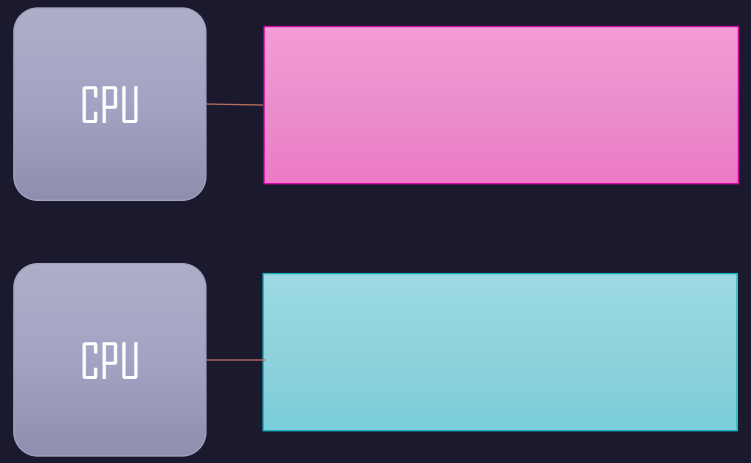
**Concurrency** means **executing** multiple tasks at the same time but not necessarily simultaneously.


Concurrent execution (on one CPU core)

# Parallelism vs. Concurrency

**Concurrency** means **executing** multiple tasks at the same time but not necessarily simultaneously.

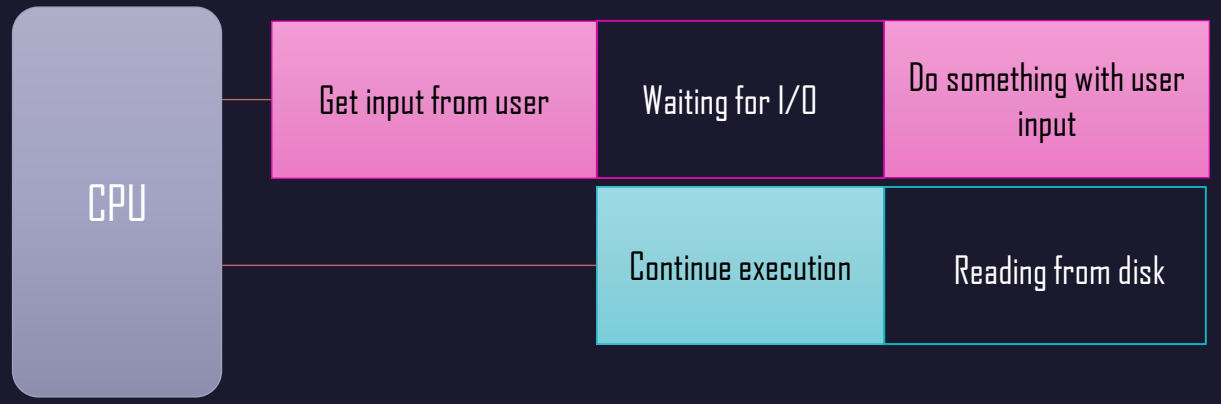

Concurrent execution (on one CPU core)



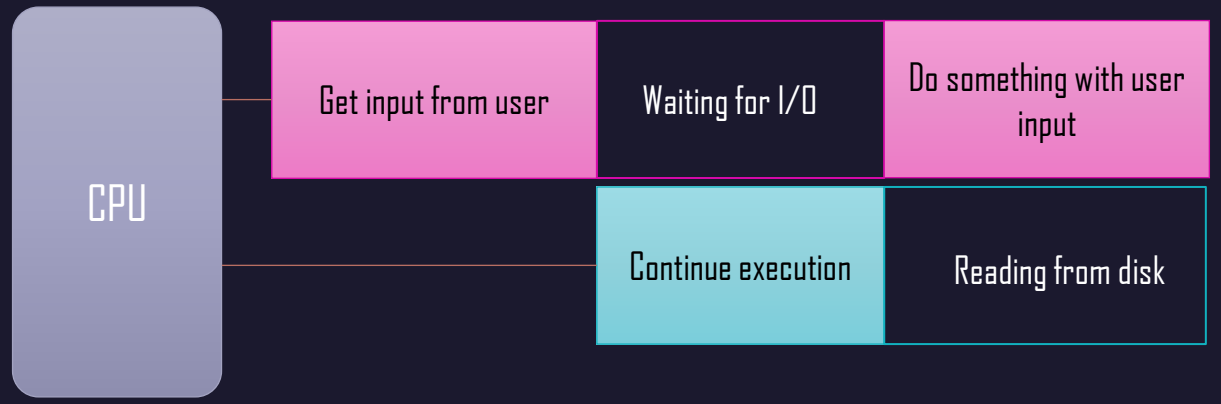

Parallel execution (on 2 CPU cores)

It is **also concurrent**

# Asynchronous execution



# Asynchronous execution

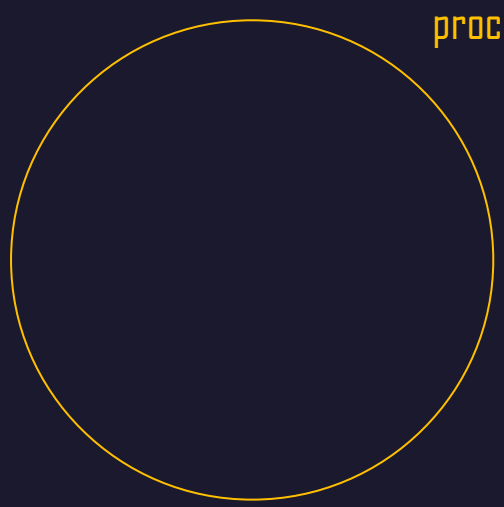



In asynchronous execution CPU is never Idle , therefore when one task is forced to wait for something, CPU is switched to other tasks until the wait is over.

**Asynchronous is also a special case of Concurrent execution**

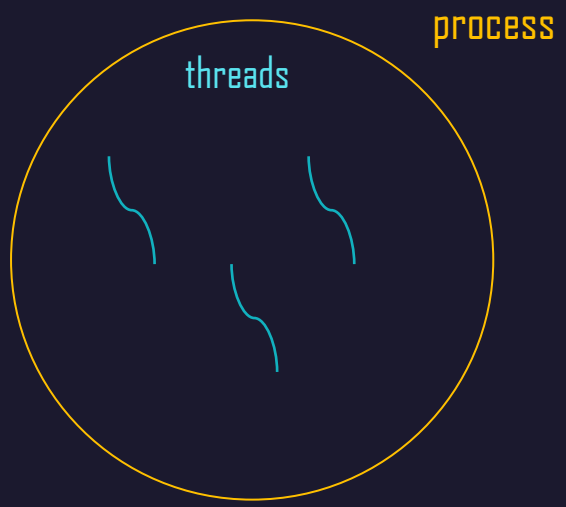

let's talk about
THREADS
What are threads?

# Process vs. Threads




- Any program we execute is **a process**

# Process vs. Threads




- Any program we execute is **a process**
- Each process may consist of one or multiple **threads** of execution
- Each thread is created to do a task in the program
- Threads are run **concurrently** so multiple tasks are executed (progress) at the same time
- In some programming languages threads could be executed in parallel (like **Java**)

# Threads in different programming languages

**multithreading** is a general programming concept and multiple programming languages implement it in different ways.

# Threads in different programming languages

**multithreading** is a general programming concept and multiple programming languages implement it in different ways.


**Java :** Threads are run **concurrently** and could also run in **parallel** on different CPU cores

# Threads in different programming languages

**multithreading** is a general programming concept and multiple programming languages implement it in different ways.


**Java :** Threads are run **concurrently** and could also run in **parallel** on different CPU cores


**JavaScript :** does **not** support multi-threading at all, but supports **asynchronous** execution

# Threads in different programming languages

**multithreading** is a general programming concept and multiple programming languages implement it in different ways.


**Java :** Threads are run **concurrently** and could also run in **parallel** on different CPU cores


**JavaScript :** does **not** support multi-threading at all, but supports **asynchronous** execution

**Python :** threads in python are run **concurrently** but **not in parallel**

# Threads in different programming languages

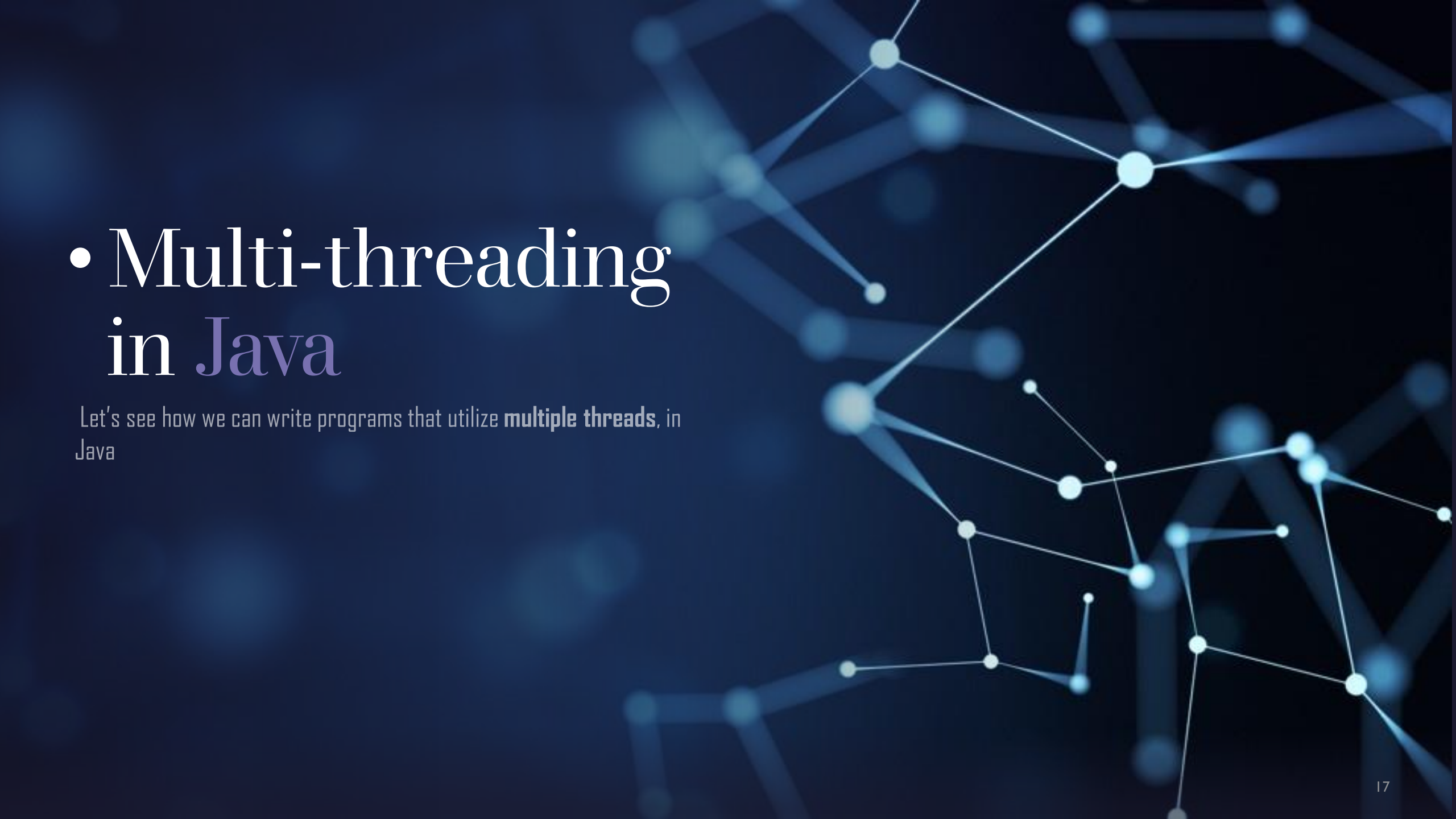
**multithreading** is a general programming concept and multiple programming languages implement it in different ways.


**Java :** Threads are run **concurrently** and could also run in **parallel** on different CPU cores


**JavaScript :** does **not** support multi-threading at all, but supports **asynchronous** execution

**Python :** threads in python are run **concurrently** but **not in parallel,** also supports **asynchronous** execution


**Go :** supports **multithreading** and **parallelism ,** but it's different with Java's implementation of threads



# • Multi-threading in Java

Let's see how we can write programs that utilize **multiple threads**, in Java

# Creating a Thread

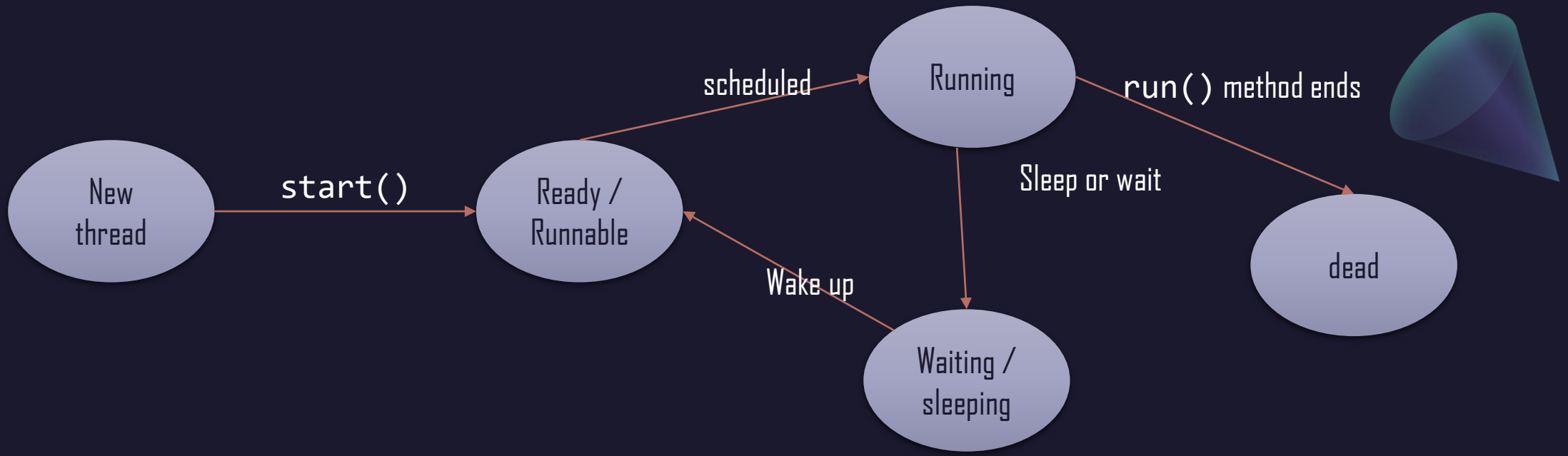
We have two choices:

1. Extending Threads class (java.lang.Thread)
2. Implementing Runnable interface (java.lang.Runnable)

Then we implement run() method to describe our thread's tasks.

**Let's see some code**

# Thread states



# Thread methods

- `Thread.currentThread()` : returns the currently running thread
- `Thread.sleep(millis)` : causes the currently executing thread to sleep for the specified number of milliseconds
- `Thread.yield()` : transfer control to another thread determined by scheduler
- `thread1.join()` : waits for `thread1` to finish execution
- `thread1.setPriority(Thread.MAX_PRIORITY)`

# Thread methods

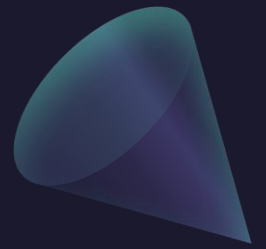
- `Thread.currentThread()` : returns the currently running thread
- `Thread.sleep(millis)` : causes the currently executing thread to sleep for the specified number of milliseconds
- `Thread.yield()` : transfer control to another thread determined by scheduler
- `thread1.join()` : waits for `thread1` to finish execution
- `thread1.setPriority(Thread.MAX_PRIORITY)`

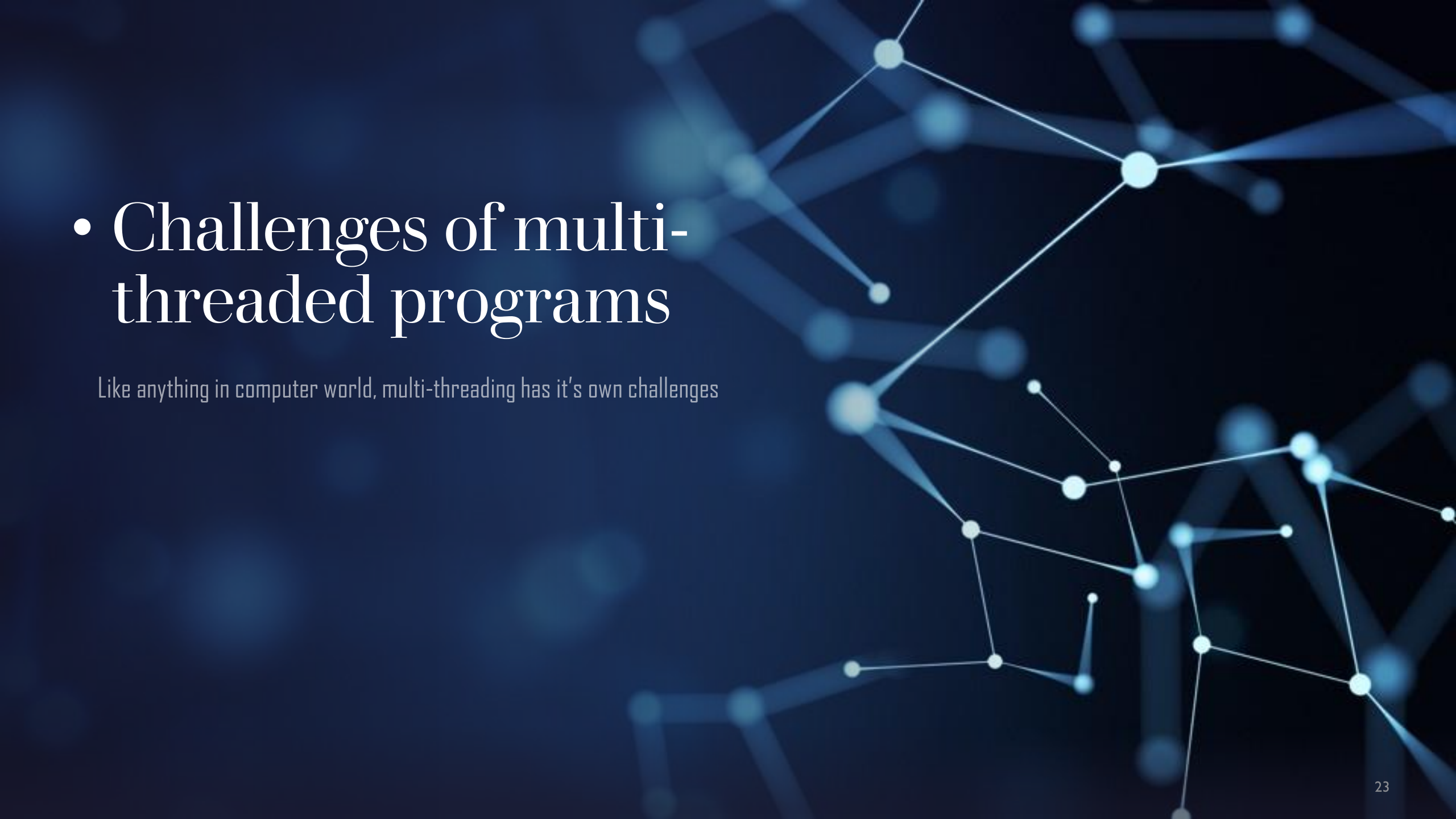
```
MAX_PRIORITY = 10
MIN_PRIORITY = 1
NORM_PRIORITY= 5
```

# Thread methods (2)

- `thread1.setDaemon(true)` : we can make threadl a daemon thread (running in the background)

Example for daemon thread : garbage collector





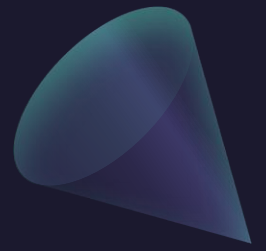
- # Challenges of multi-threaded programs

Like anything in computer world, multi-threading has it's own challenges

# critical sections

Sections in a program which we don't want multiple threads to enter at the same time

- If one threads entered critical section, no other threads must be allowed to enter
- Second thread must wait until the first one exits critical section

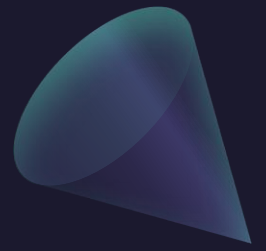


**Why?**

# critical sections

Sections in a program which we don't want multiple threads to enter at the same time

- If one thread entered critical section, no other threads must be allowed to enter
- Second thread must wait until the first one exits critical section



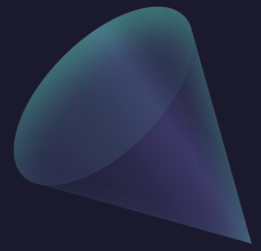
## Why?

- **Shared resources**:
  an entity (object , variable, ...) used in multiple threads in a shared manner

- **Race condition**:
  multiple threads trying to access a shared resource and at least one of them trying to change it

# Race condition

```
//shared by both threads
void critical(){
 int a = 5;
 if (a == 6) {
    print(“this must be printed”);
  }
 else if (a < 7){
    a++;
  }
 else {
   throw error;
 }
}
```



# Race condition

```
//shared by both threads
void critical(){
 int a = 5;
 if (a == 6) {
    print(“this must be printed”);
  }
 else if (a < 7){
    a++;
  }
 else {
   throw error;
 }
}
thread1 runs critical()
thread2 runs critical()
```

# Race condition

```
//shared by both threads
void critical(){
 int a = 5;
 if (a == 6) {
    print(“this must be printed”);
  }
 else if (a < 7){
    a++;
  }
 else {
   throw error;
 }
}
thread1 runs critical()
thread2 runs critical()
```

Imagine the following scenario:

1. thread 1 runs critical, finds a to be 5 and executes `else if (a < 7)`
2. Before it can increment a scheduler gives control to thread2
3. thread2 finds a to be still 5 and executes `else if (a < 7) a++;`
4. Now thread1 continues and runs a++

# Race condition

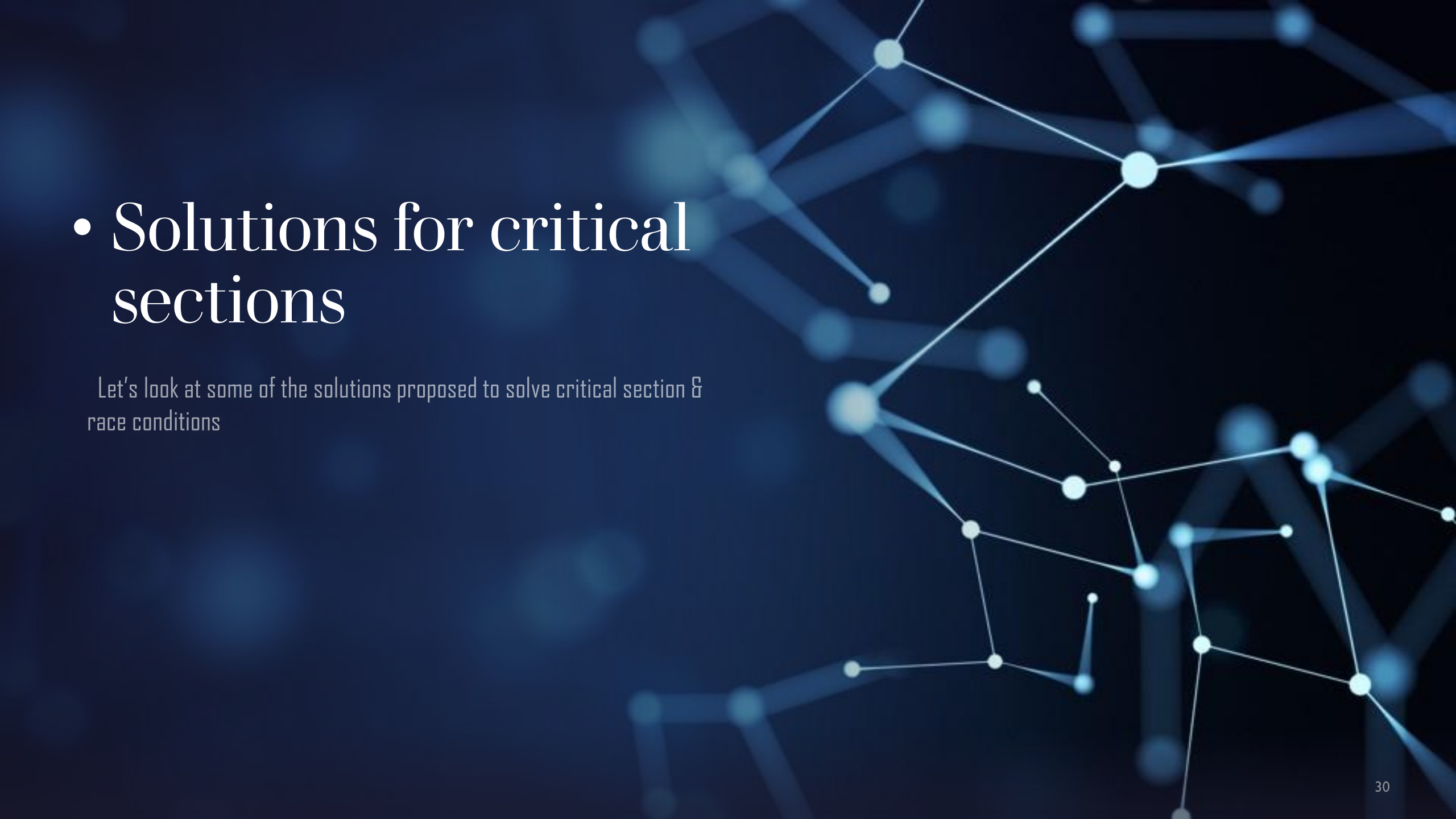
```
//shared by both threads
void critical(){
 int a = 5;
 if (a == 6) {
    print("this must be printed");
  }
 else if (a < 7){
    a++;
  }
 else {
   throw error;
 }
}
thread1 runs critical()
thread2 runs critical()
```

Critical section

Imagine the following scenario:

1. thread 1 runs critical, finds a to be 5 and executes `else if (a < 7)`
2. Before it can increment a scheduler gives control to thread2
3. thread2 finds a to be still 5 and executes `else if (a < 7) a++;`
4. Now thread1 continues and runs a++

**Now a is 7 and we never printed anything and**
**If we execute critical() one more time we get an error**



# • Solutions for critical sections

Let's look at some of the solutions proposed to solve critical section & race conditions

# Synchronized blocks

```
public class BankAccount {
     private float balance;
     public synchronized void deposit(float amount) {
        balance += amount;
     }
     public synchronized void withdraw(float amount) {
        balance -= amount;
     }
  }
```

# Synchronized blocks

```
public class BankAccount {
     private float balance;
     public synchronized void deposit(float amount) {
        balance += amount;
     }
     public synchronized void withdraw(float amount) {
        balance -= amount;
     }
  }
```

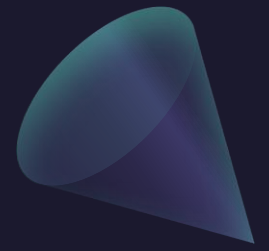
Deposit and withdraw methods are critical sections here

Two threads can't simultaneously call synchronized methods (enter synchronized block) on the **same object**

# Inter-thread Communication

Two threads can communicate using notify() and wait() methods:

Sometimes a thread needs to wait until another thread notify it to continue



There can be multiple threads on in waiting state on each object, notifyAll() will notify all of them to continue running

*** wait and notify must be called inside a synchronized block , otherwise **IllegalMonitorStateException** is thrown***

- Notify and wait are native methods which means they have a **low level implementation**

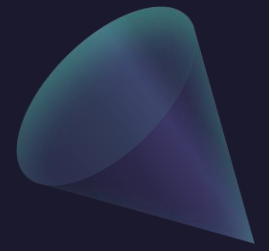
# notify() & wait()

```
synchronized( lockObject )
{
    while( ! condition )
    {
        lockObject.wait();
    }

    //take the action here;
}
```

```
synchronized(lockObject)
{
    //establish_the_condition;

    lockObject.notify();

    //any additional code if needed

}
```

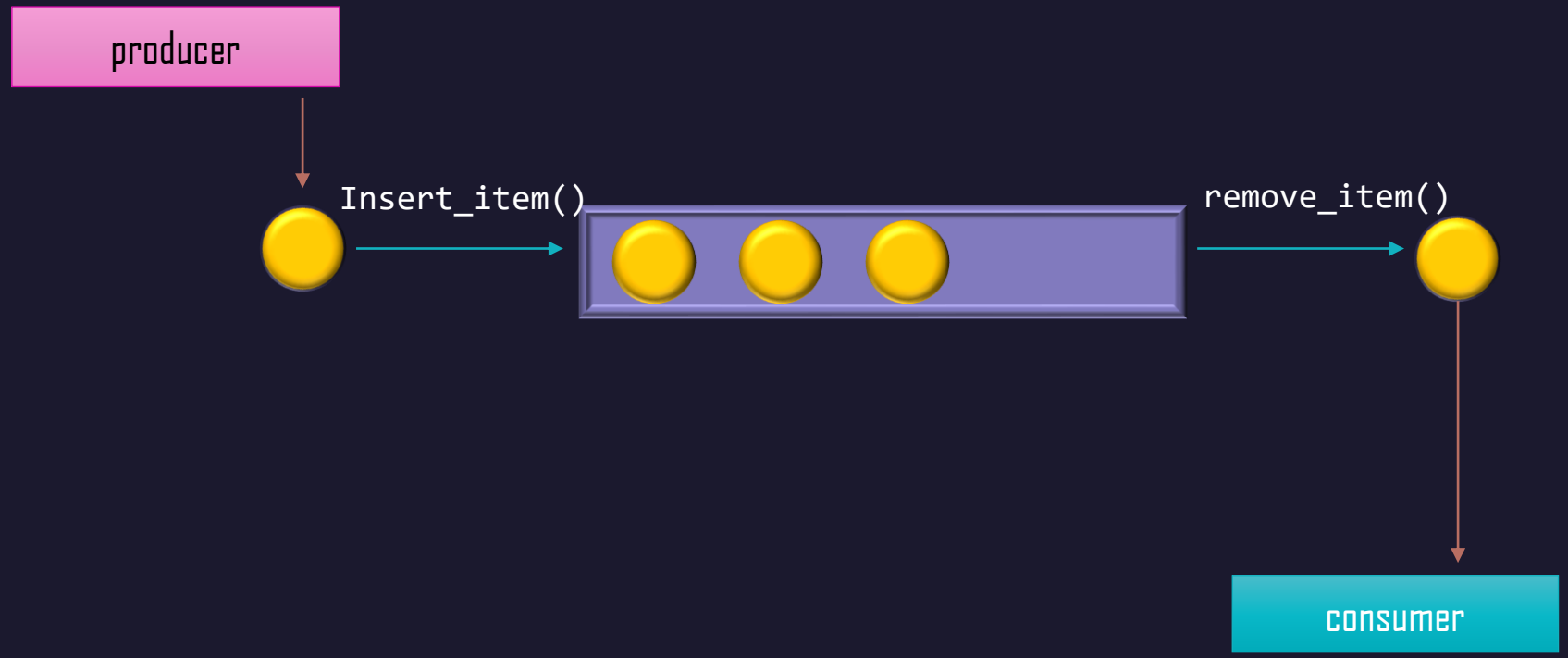
# interrupt()

A thread might be in a waiting status, if we call interrupt on it , thread **exits waiting state** and **InterruptedException** will be thrown

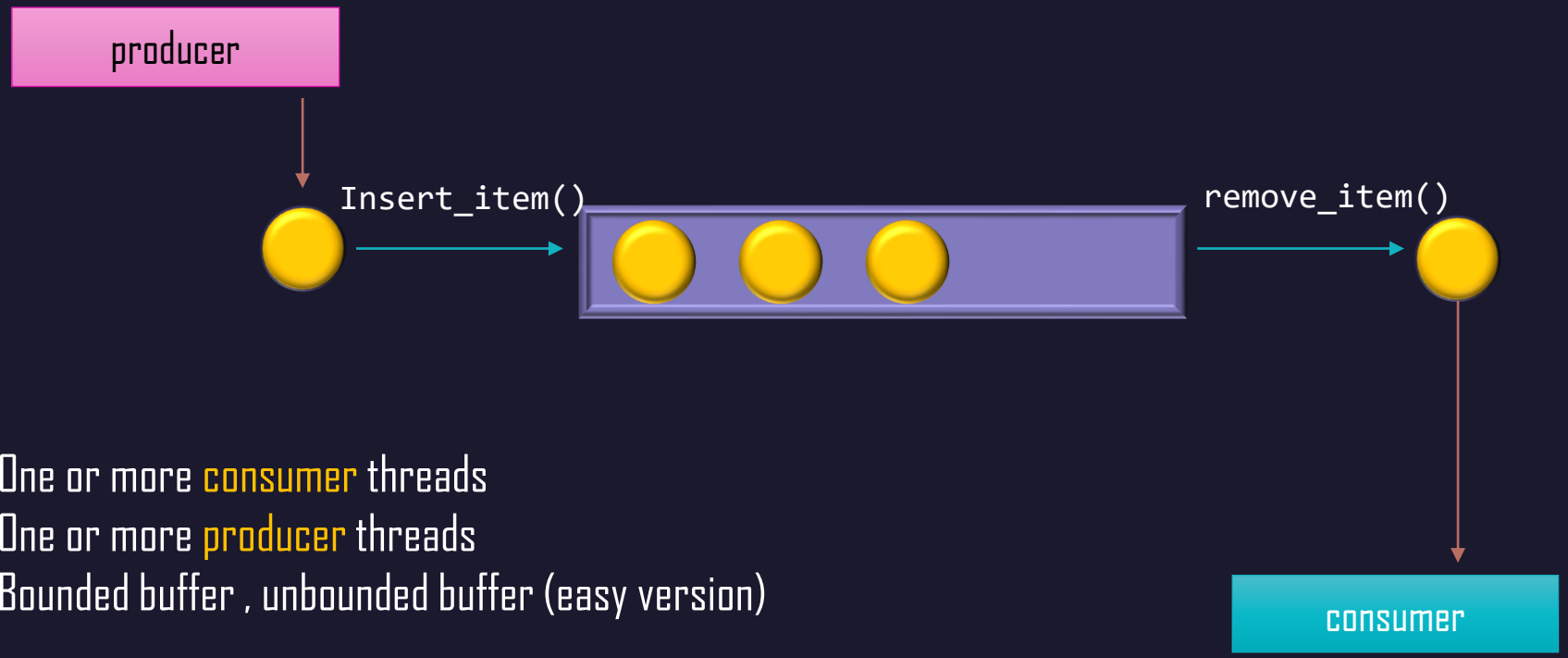


If the thread is not in the sleeping or waiting state then calling the interrupt() method performs a normal behavior and doesn't interrupt the thread but sets the interrupt flag to true.

# Producer Consumer problem



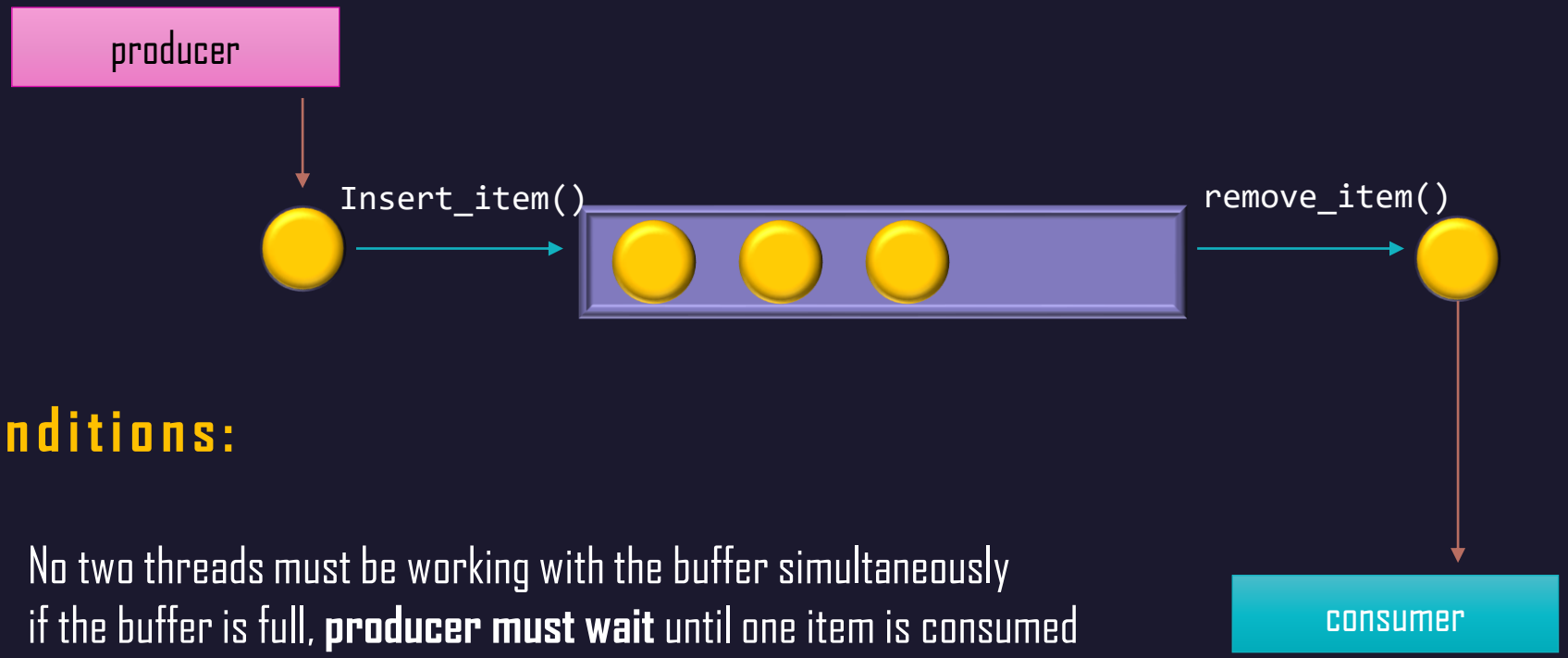
# Producer Consumer problem




- One or more consumer threads
- One or more producer threads
- Bounded buffer , unbounded buffer (easy version)

- **A traditional and important problem in concurrency settings (Operating Systems , ….)**

# Producer Consumer problem




## Conditions:

1. No two threads must be working with the buffer simultaneously
2. if the buffer is full, **producer must wait** until one item is consumed
3. If the buffer is empty the **consumer must wait** until an item is produced

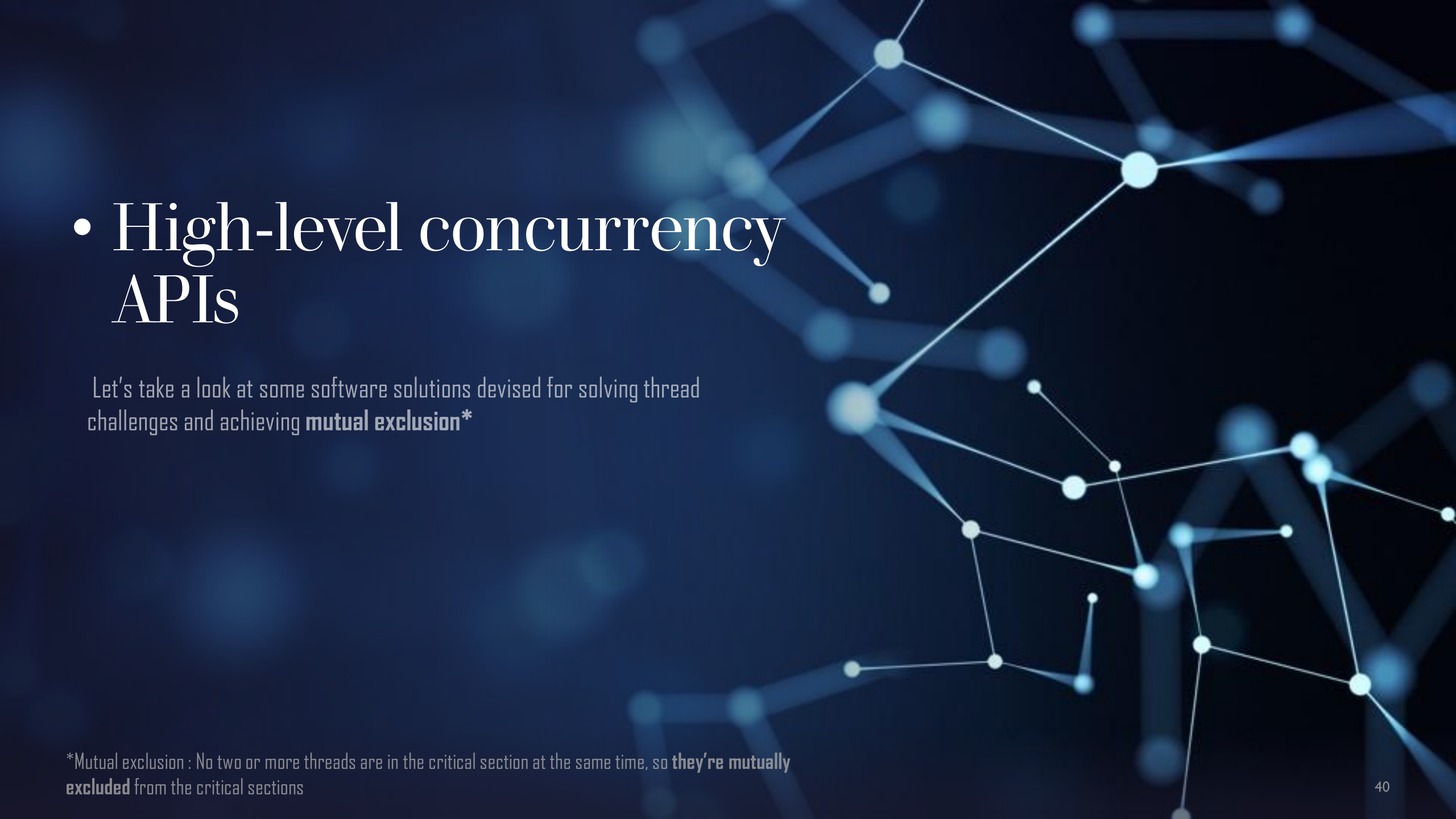
**Lets see some code;**

# Thread safe Collections

When using thread safe Objects we don't to use **synchronized** because they are **thread safe**

| thread safe | non thread safe |
| --- | --- |
| Vector | Arraylist |
| StringBuffer | StringBuilder |
| ConcurrentHashMap | HashMap |

And all **immutable** Objects such as String , Integer, ...

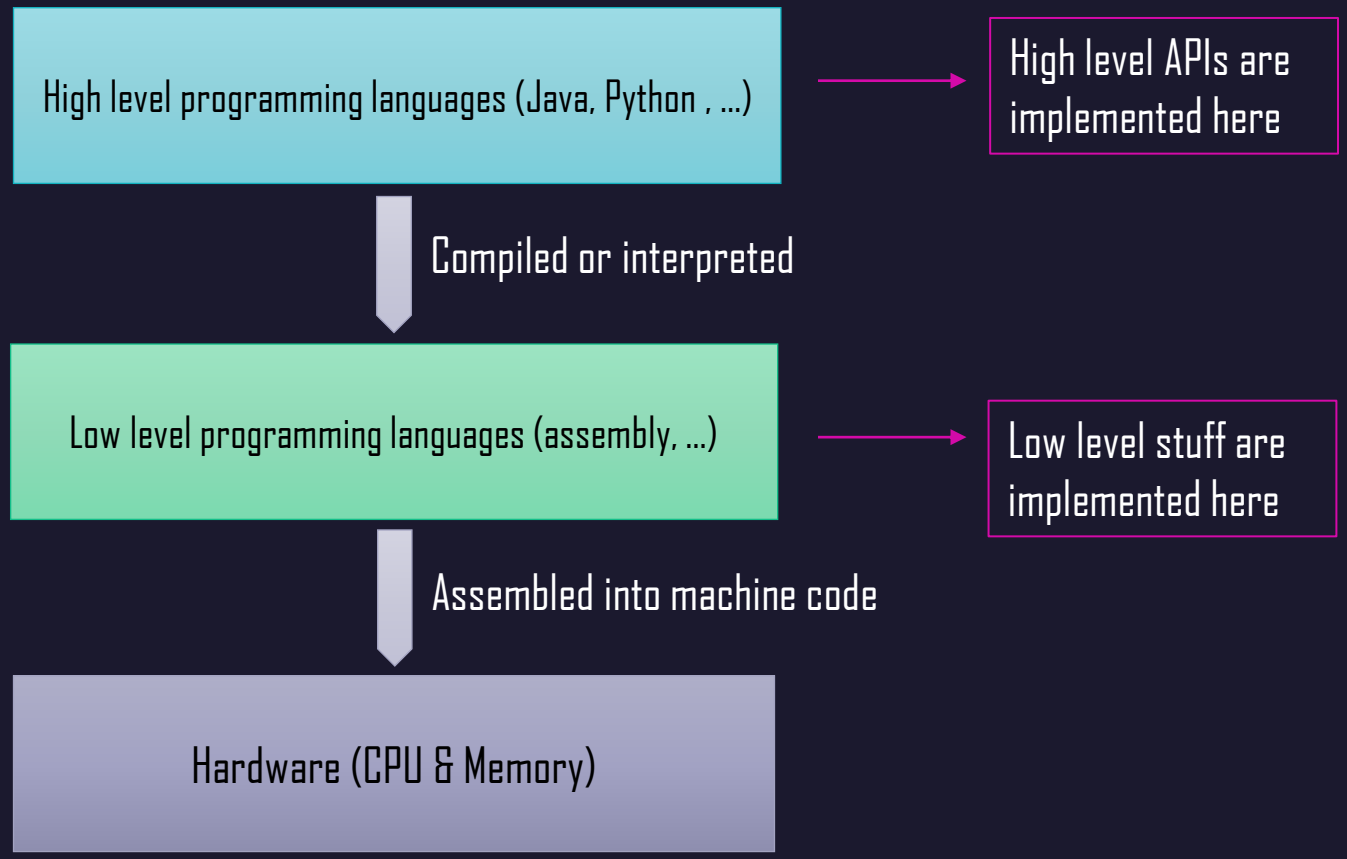


- # High-level concurrency APIs

Let's take a look at some software solutions devised for solving thread challenges and achieving **mutual exclusion***

*Mutual exclusion : No two or more threads are in the critical section at the same time, so **they're mutually excluded** from the critical sections

# What do we mean by high-level?



# Synchronizers

Used in order to synchronize multiple thread `(java.util.concurrent)`

1. **Semaphore**
2. CountDownLatch
3. Exchanger
4. CyclicBarrier

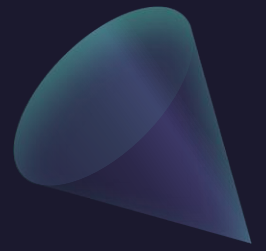
Semaphore is a common solution to achieve mutual exclusion, and is not only  limited to Java.
Semaphores are widely used in Unix based **Operation Systems**

# Semaphore ;

Methods : `acquire(), release()`

A semaphore controls access to a shared resource through the use of a counter.

If the counter is greater than zero, then access is allowed.



# Semaphore ;

Methods : `acquire(), release()`
A semaphore controls access to a shared resource through the use of a counter.
If the counter is greater than zero, then access is allowed.

**In producer /consumer problem:**

```
Semaphore semaphore = new Semaphore(0);
```

Permits 0 threads to acquire lock

```
semaphore.acquire();
synchronized (list)
 { obj = list.remove(0); }
```

```
synchronized(list)
{ list.add(obj); }
semaphore.release();
```

# Atomic classes

Remember this code?

```
void critical(){
 int a = 5;
 if (a == 6) {
    print(“this must be printed”);
  }
 else if (a < 7){
    a++;
  }
 else {
   throw error;
 }
}
```

What if we could never take execution away from Thread 1 while executing else if ?

# Atomic classes

An **atomic action** is an action which is executed as a whole, and we can not take the control away while executing it.

Java supports atomic operations on variables through atomic classes (`java.util.concurrent.atomic`)

Some atomic classes:

- `AtomicBoolean`
- `AtomicInteger(extends Number)`
- `AtomicIntegerArray`
- `AtomicLongArray`
- `AtomicReference`

And so on …

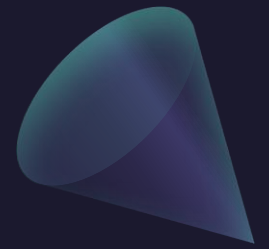
Atomic variables are **thread-safe**, therefor no synchronization is needed

# Lock interface

Gives us more control over how we want to execute critical sections

**Lock** : controlled by programmer explicitly

**Synchronized**: automatically locks critical section

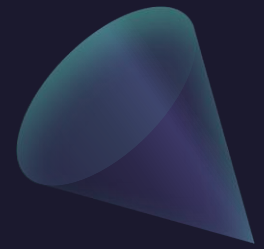


# Lock interface

Gives us more control over how we want to execute critical sections

**Lock** : controlled by programmer explicitly

**Synchronized**: automatically locks critical section



```
Lock l = new ReentrantLock(); // ReentrentLock implements Lock
 l.lock();
 try { ... // critical section }
finally { l.unlock(); }
```

# Executor framework

Contemporary way of creating threads

program determines that a part needs a new thread

In large scale enterprise application thread management is not done by programmer
the programmer only defines tasks.
**Executor framework is used in this case**

```
interface Executor {
  void execute(Runnable command);
}
```

The executor might define a new thread for the given task , or use an existing idle thread, it depends on the type of executor and the state of **thread pool**

# Executor framework

Contemporary way of creating threads

program determines that a part needs a new thread

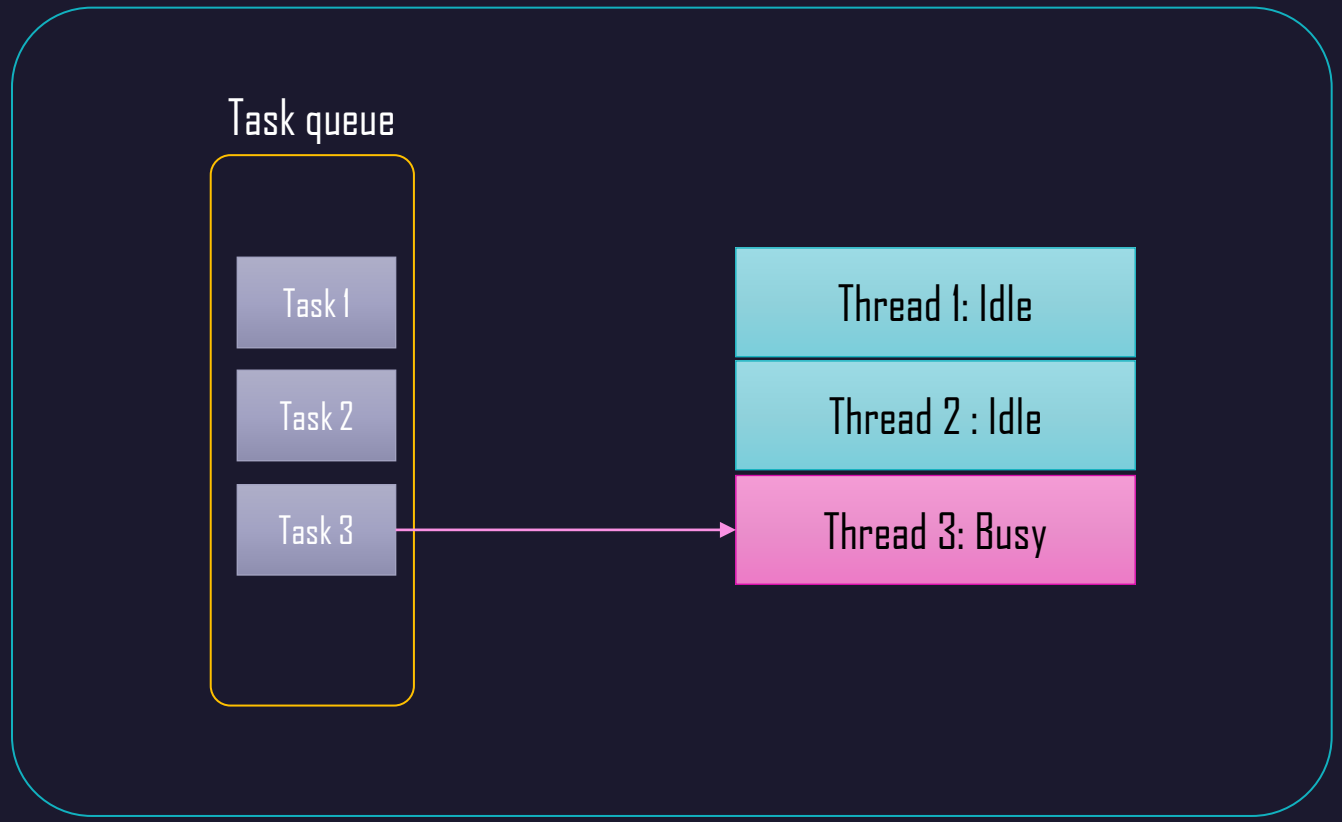
In large scale enterprise application thread management is not done by programmer
the programmer only defines tasks.
**Executor framework is used in this case**

```
interface Executor {
  void execute(Runnable command);
}
```

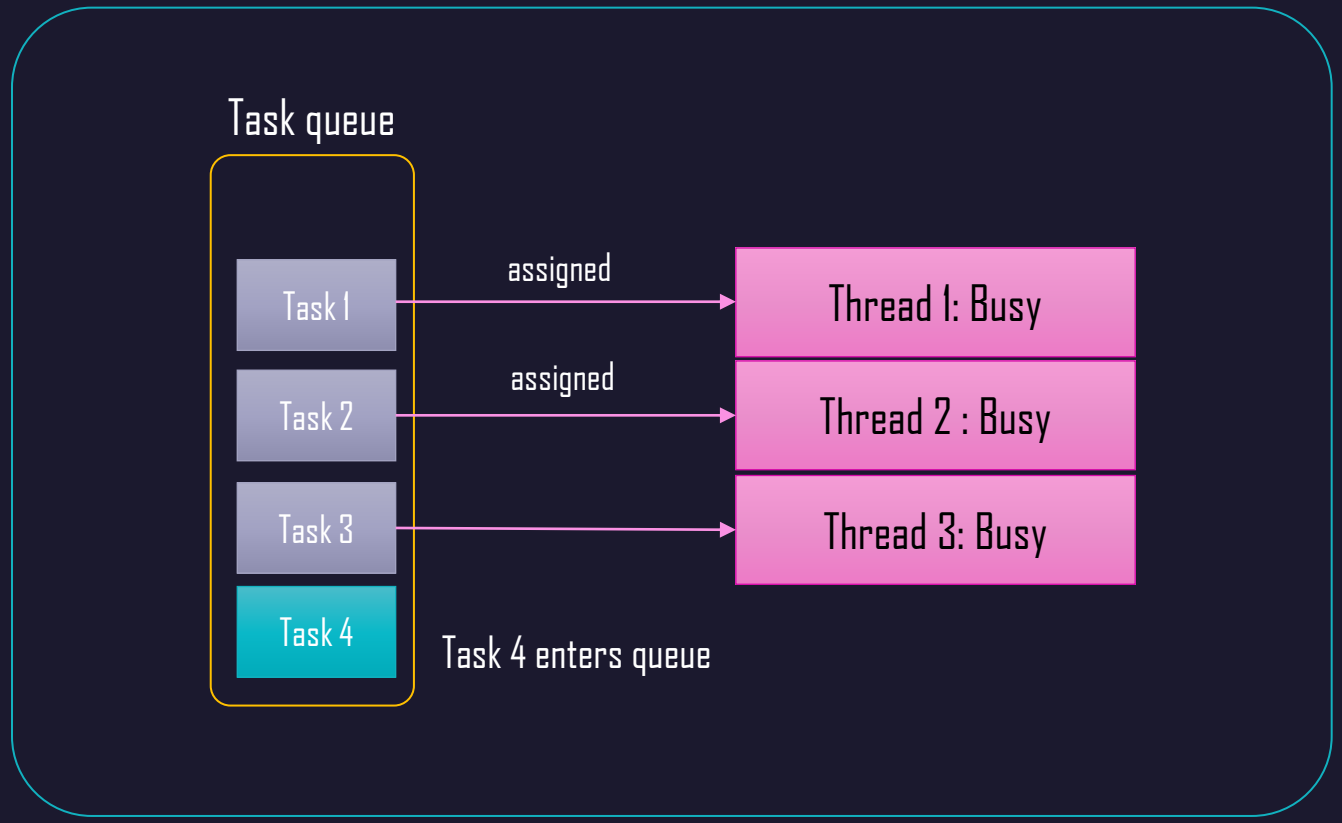
The executor might define a new thread for the given task , or use an existing idle thread, it depends on the type of executor and the state of **thread pool**

Thread what?!

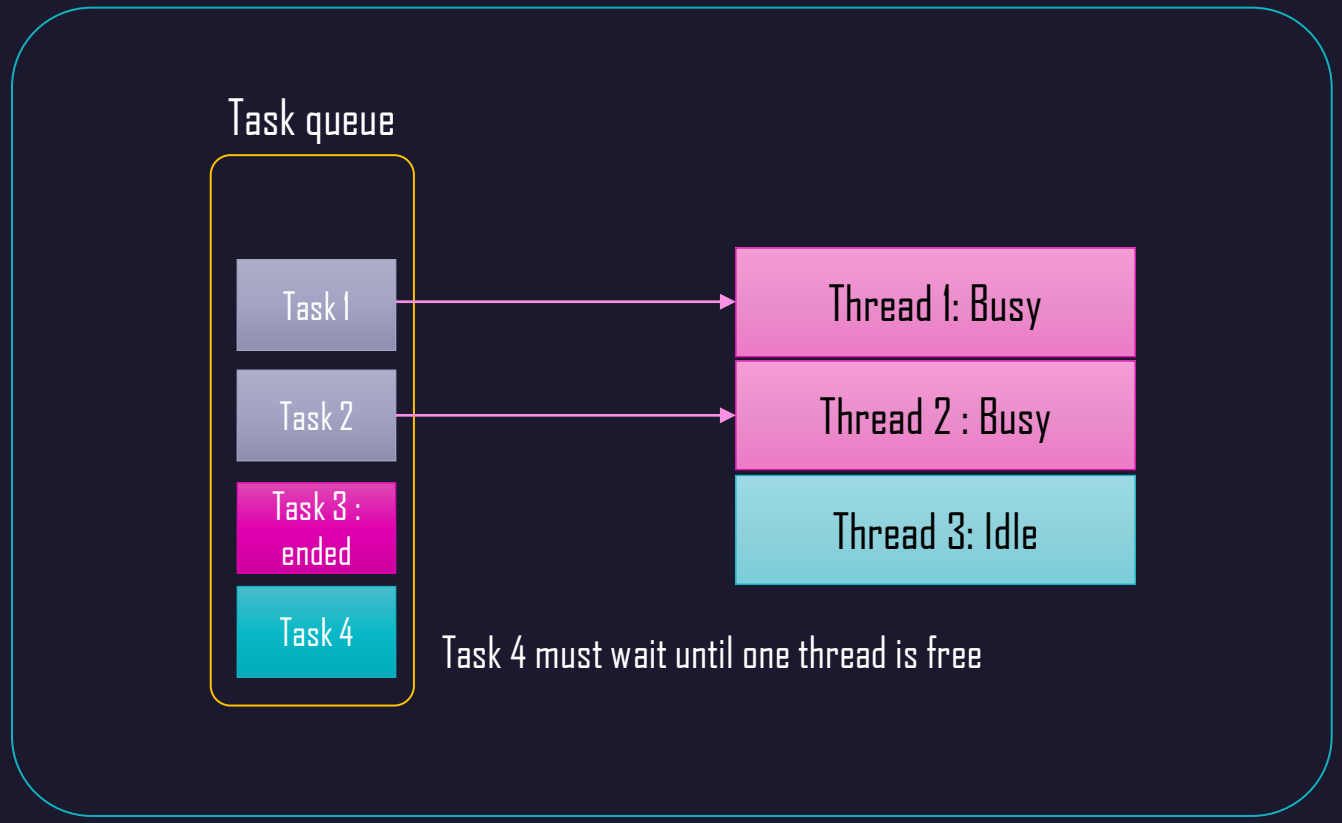
# Thread pool


Task queue
Task 1
Task 2
Task 3
Thread 1: Idle
Thread 2 : Idle
Thread 3: Busy

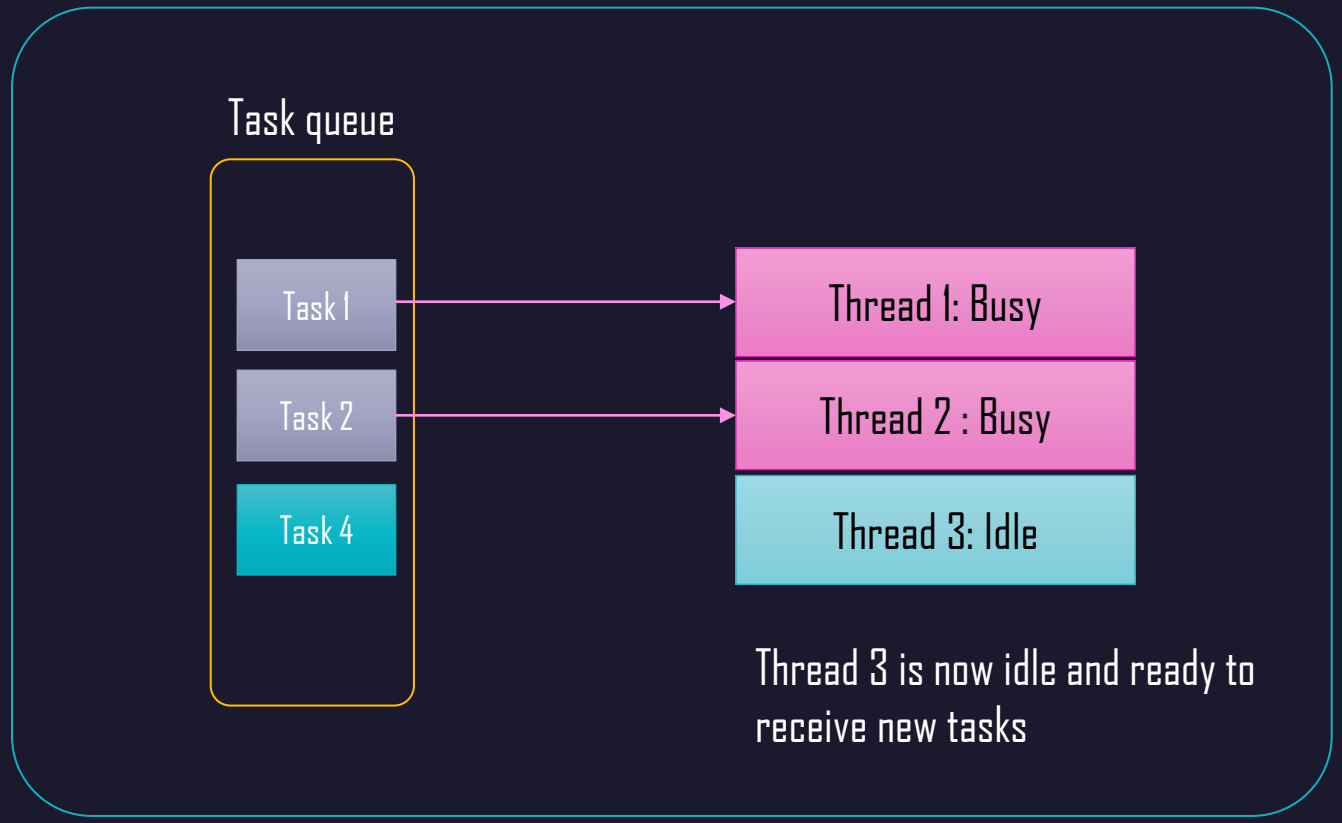
# Thread pool


Task queue
Task 1
Task 2
Task 3
Task 4
assigned
assigned
Thread 1: Busy
Thread 2 : Busy
Thread 3: Busy
Task 4 enters queue

# Thread pool


Task queue
Task 1
Task 2
Task 3 : ended
Task 4
Thread 1: Busy
Thread 2 : Busy
Thread 3: Idle
Task 4 must wait until one thread is free

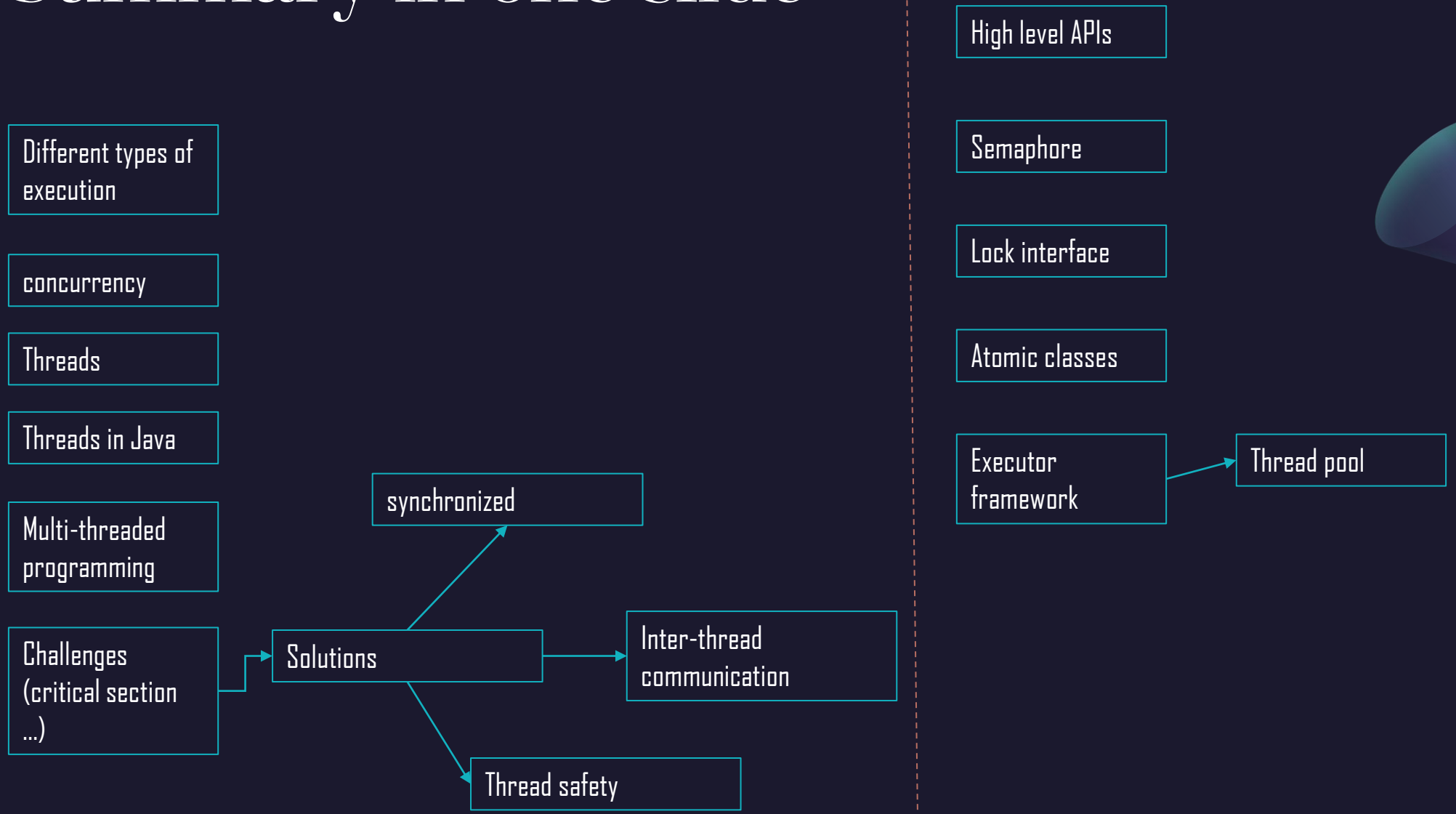
# Thread pool




By using thread pool, executors limit the number of threads created, because **creating a new thread for every single task is costly**

**Let's see some code;**

# Summary in one slide



# The end