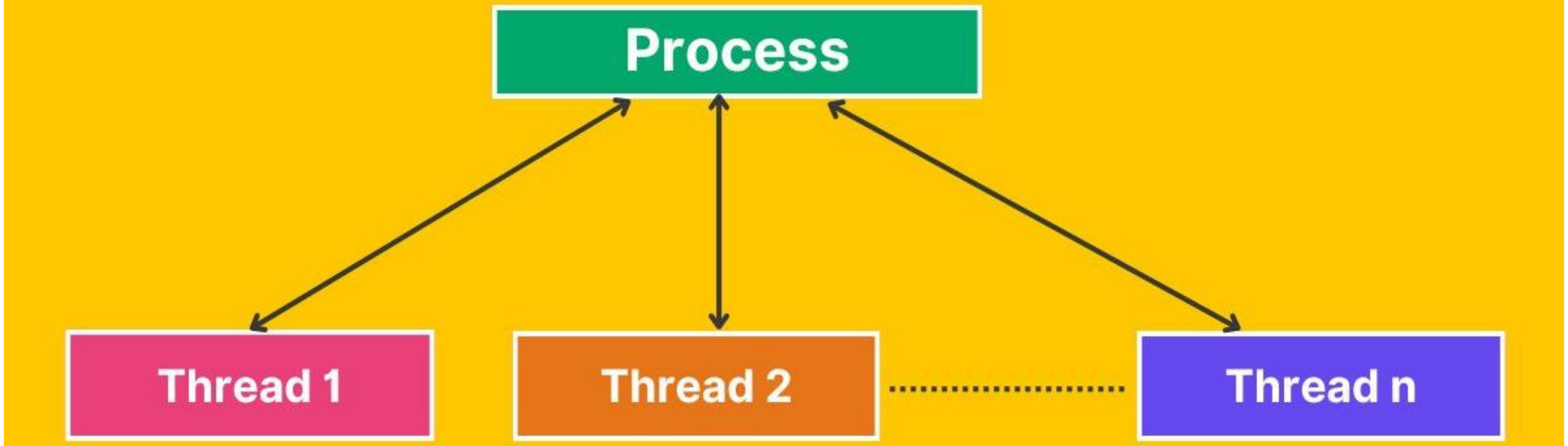
Understanding Multithreading In Java Programming



Process

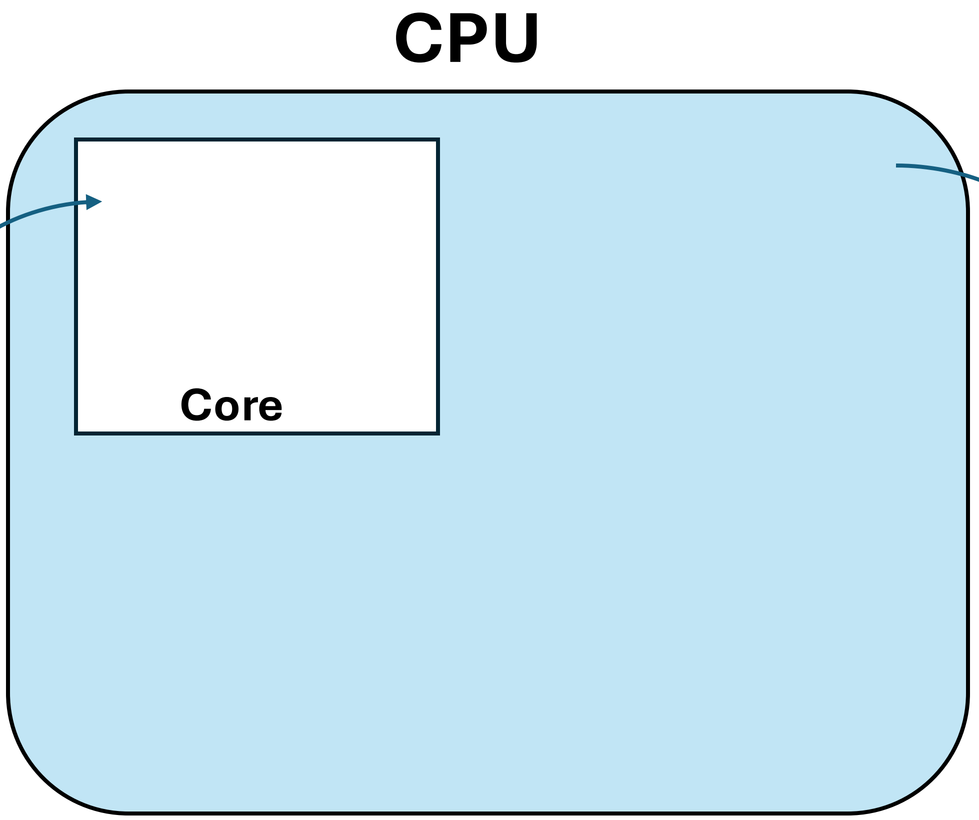
Thread 1

Thread 2

Thread n

Topics we will cover

- CPU, Core, Thread
- Multitasking
- Multithreading
- How to create thread?
- Lifecycle of threads
- Thread class methods
- Synchronization
- Daemon thread
- Inter-thread communication
- Locks
- Basic of Executor framework

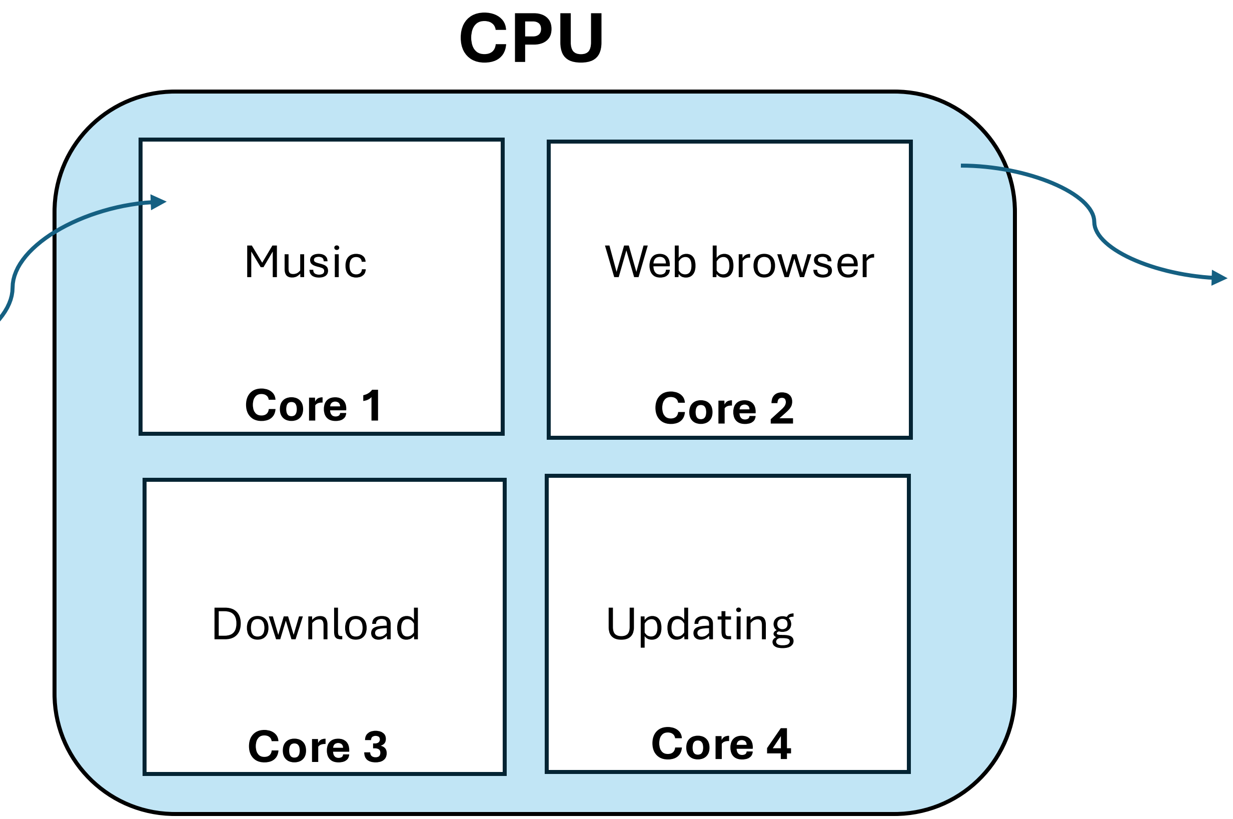


Core is an individual processing unit within a CPU

(Single core)

Brain of computer, responsible for executing instructions from programs

Core is an individual processing unit within a CPU



CPU

Music

Core 1

Web browser

Core 2

Download

Core 3

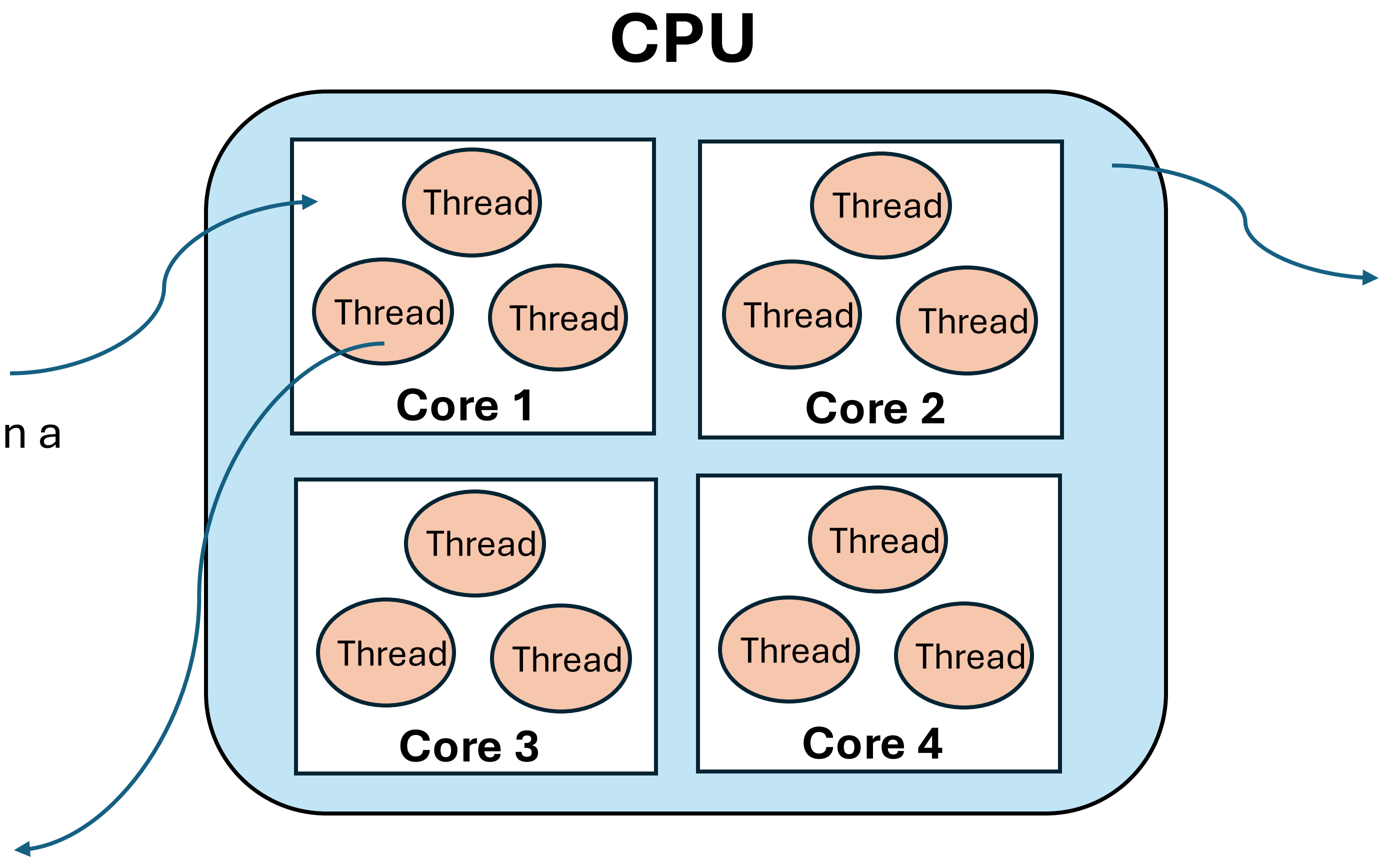
Updating

Core 4

Brain of computer, responsible for executing instructions from programs

(Multiple core)

Quad Processor – 4 core

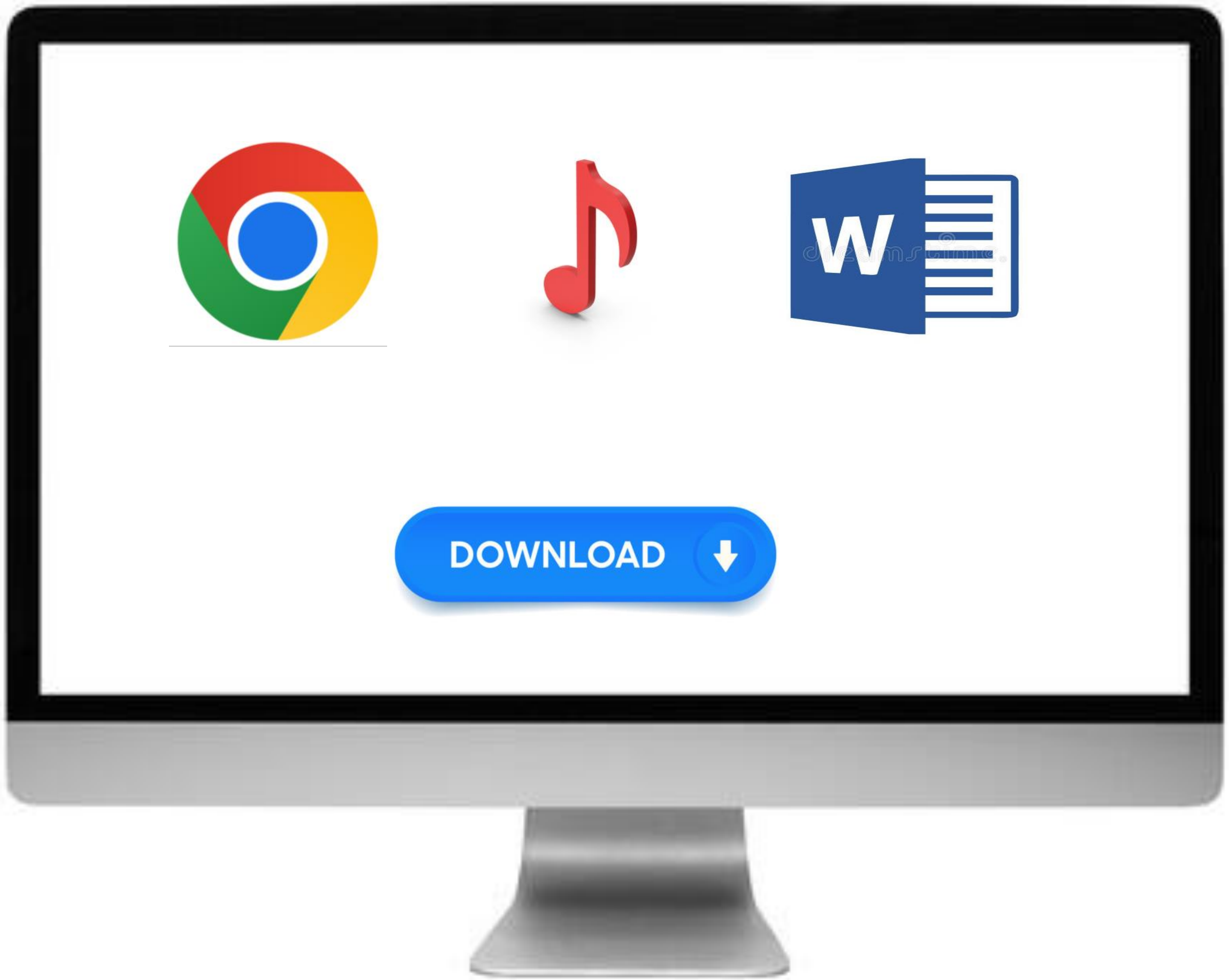


Core is an individual processing unit within a CPU

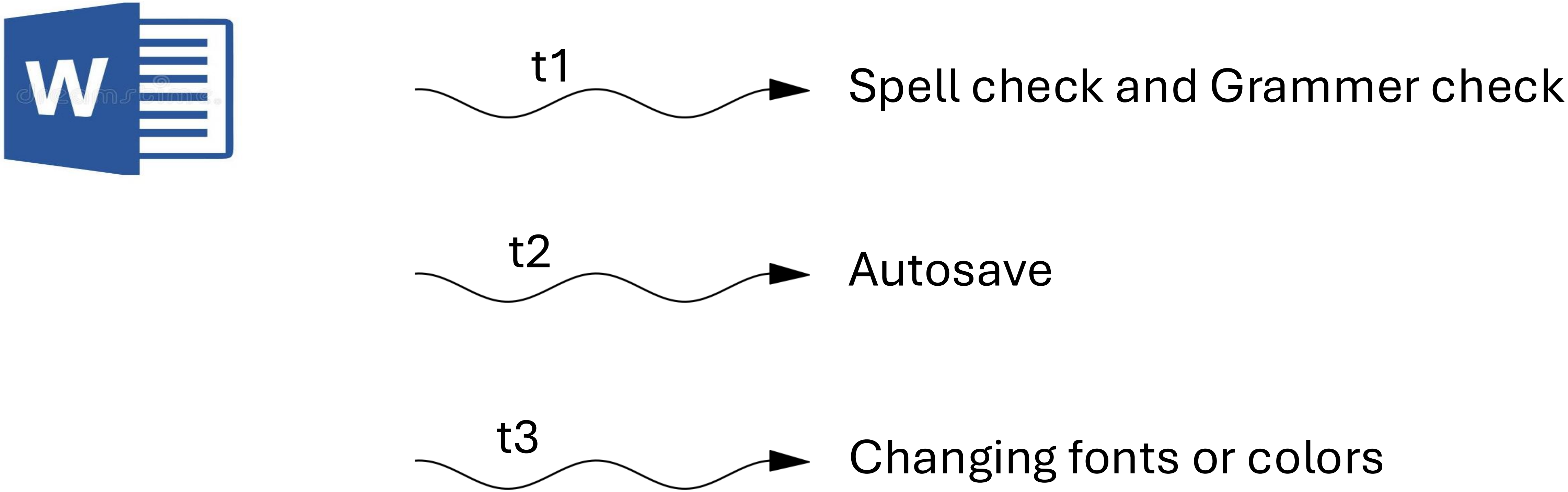
Brain of computer, responsible for executing instructions from programs

Thread is the smallest unit of execution within a process

Multitasking



Multithreading



Why use Multithreading ?

- To increase the performance

- Better utilization of CPU cores

- Real time applications like gaming, web browsers

How to create Thread in Java

Two ways to create thread in Java

- ☞ By extending Thread class
- ☞ By implementing Runnable Interface

Extending Thread class :-

```
public class MyThread extends Thread{

    @Override
    public void run() {
    System.out.println("Thread task");
    }

    public static void main(String[] args) {
    MyThread t = new MyThread();
    t.start();
    }

}
```

Output :

```
Thread task
```

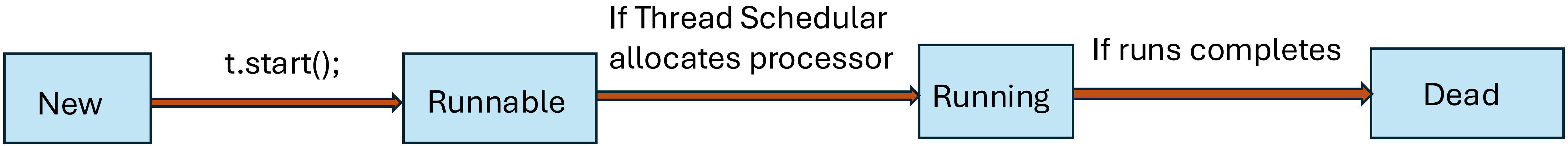
Implementing Runnable interface :-

```
public class MyThread implements Runnable{

   @Override
   public void run() {
   System.out.println("Thread task");
   }


    public static void main(String[] args) {
     MyThread t = new MyThread();
     Thread thread = new Thread(t);
     thread.start();
     }
}
```

Output :

Thread task

Lifecycle of Thread



New

t.start();

Runnable

If Thread Schedular allocates processor

Running

If runs completes

Dead

MyThread t = new MyThread();

Thread class

(implements Runnable)

- public String getName()
- public void setName(String name)
- public boolean isAlive()
- public void setPriority(int priority)
- public static void sleep(long millis) throws InterruptedException
- public static void yield()
- public void join() throws InterruptedException
- public void interrupt()

Get and set name of Thread

```
public class MyThread extends Thread{

    @Override
    public void run() {
        System.out.println("Run method");
        System.out.println(Thread.currentThread().getName())
    }

    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();

        System.out.println(Thread.currentThread().getName());

        Thread.currentThread().setName("Supriya");

        System.out.println(Thread.currentThread().getName());

    }
}
```

```
Output:
Main
Run method
Thread-0
```

```
Output:
Main
Supriya
Run method
Thread-0
```

To check if the Thread is alive

```java
public class MyThread extends Thread{

   @Override
   public void run() {
      System.out.println("Run method");

   }

   public static void main(String[] args) {
      MyThread t = new MyThread();
      t.start();

     System.out.println(Thread.currentThread().isAlive());
   }
  }
```

```
Output:
true
Run method
```

To set priorities

- Priorities are represented in the form of integer which ranges from 1-10.

1 >> MIN_PRIORITY

5 >> NORM_PRIORITY

10 >> MAX_PRIORITY

- Default priority of main thread is 5.
- Windows do not support priorities; it depends upon the platform.
- If multiple threads having same priorities which thread will be executed depends upon JVM.

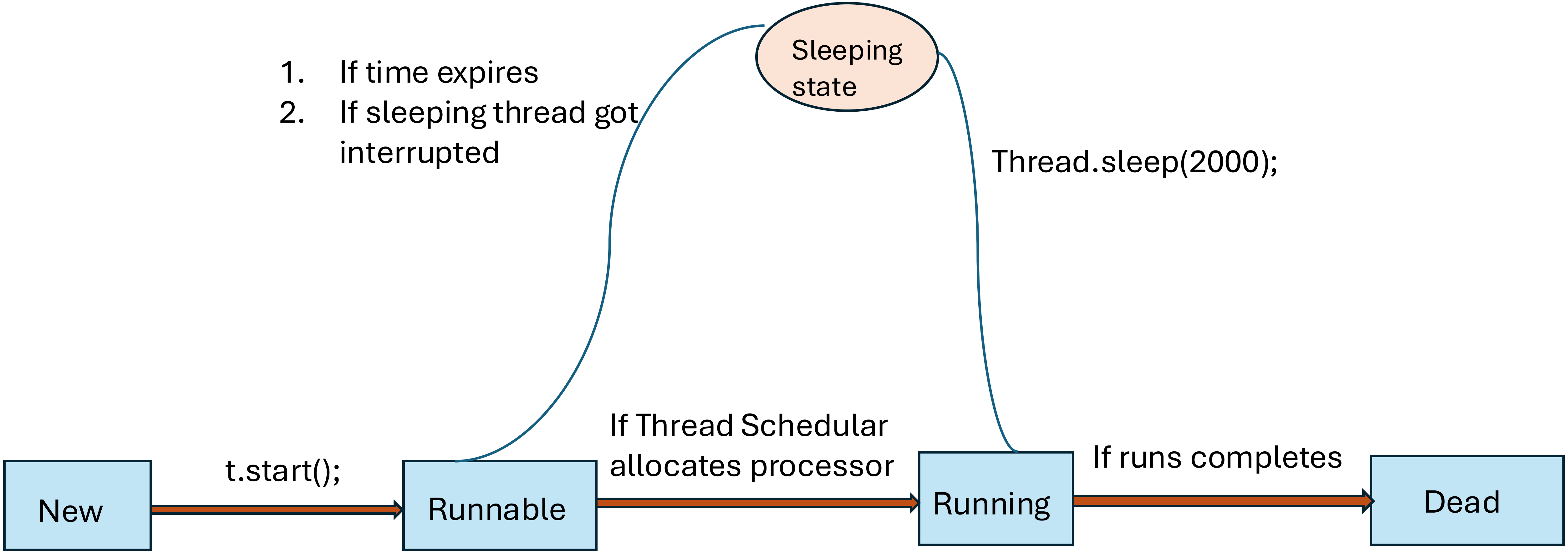
To set priorities

```
public class Priority extends Thread{

 @Override
 public void run() {
    System.out.println("Task priority thread : " + Thread.currentThread().getPriority());   //3
 }

 public static void main(String[] args) {
  System.out.println("Main thread old priority : " + Thread.currentThread().getPriority()); //5
  Thread.currentThread().setPriority(6);
  System.out.println("Main thread new priority : " + Thread.currentThread().getPriority()); //6
  Priority t = new Priority();

  t.setPriority(3);

  t.start();
 }

}
```

To prevent thread execution : sleep()

```
public class Sleep extends Thread{

    @Override
    public void run() {

    for(int i = 1; i<=10; i++){
        try{
            Thread.sleep(2000);
           System.out.println(i);
        }catch (Exception e){
        }
       }
    }

    public static void main(String[] args) {

    Sleep t = new Sleep();
    t.start();
    }
}
```

To prevent thread execution : sleep()



1. If time expires
2. If sleeping thread got interrupted

Thread.sleep(2000);

MyThread t = new MyThread();

To prevent thread execution : yield()

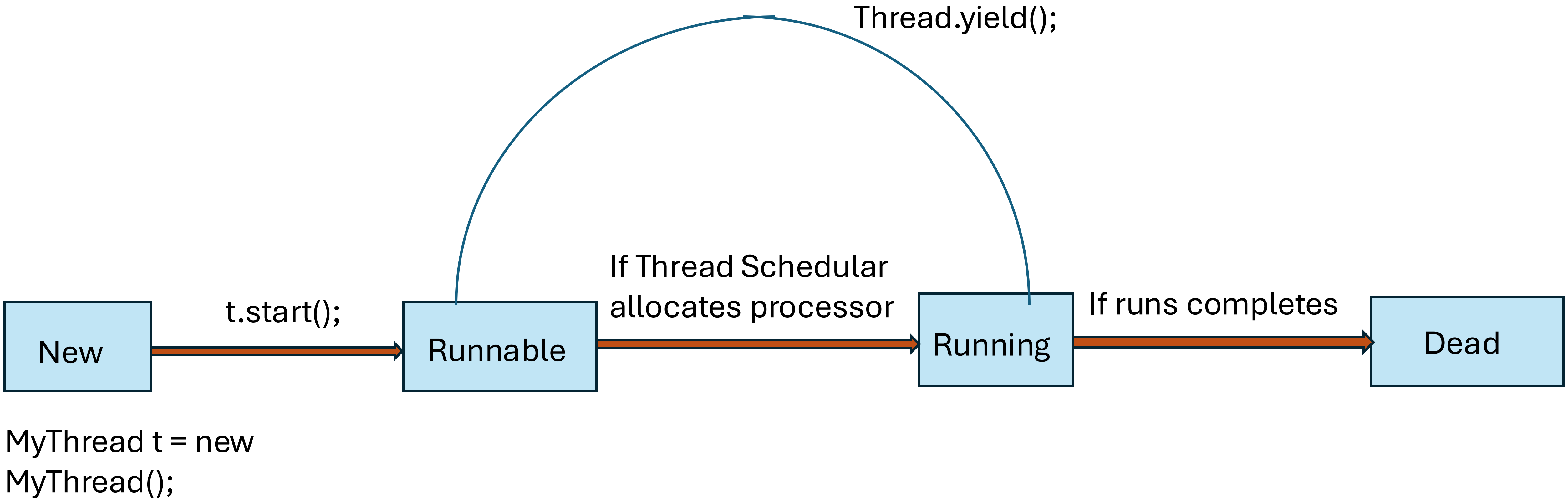
- yield() method stops the current executing thread and give a chance to other threads for execution.

- Till java 5 it internally used sleep() method
- From java 6 thread provides hint to the thread schedular, then it depends on thread schedular to accept or ignore it.

To prevent thread execution : yield()

```
public class MyThread extends Thread{

    @Override
    public void run() {
        for (int i = 0; i<=10 ; i++){
            System.out.println("Child Thread");
            Thread.yield();
        }
    }
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();

        for (int i =0; i<=10; i++){
            System.out.println("Main method");
        }
    }
}
```

To prevent thread execution : yield()



Thread.yield();

New

t.start();

Runnable

If Thread Schedular allocates processor

Running

If runs completes

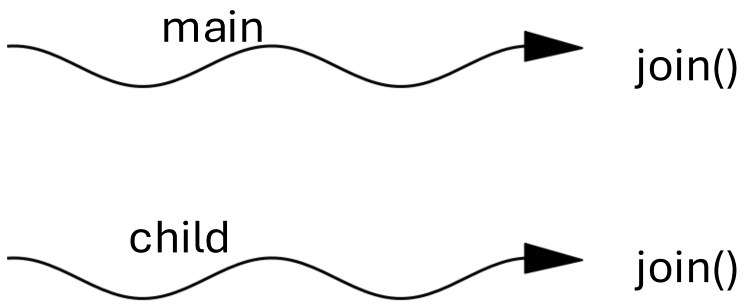
Dead

MyThread t = new MyThread();

To prevent thread execution : join()

- If a thread wants to wait for another thread to complete its task then we should use join() method.

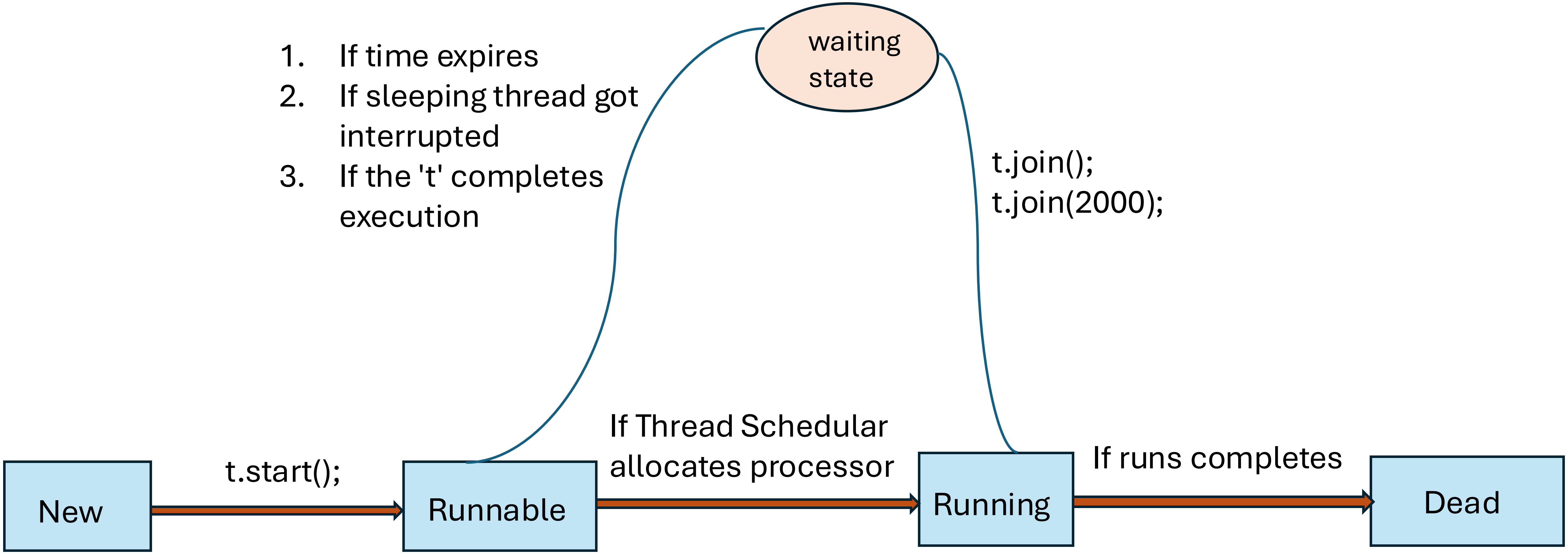
- If main method calls join method on child thread object and child thread calls main method on main thread object than both threads will wait forever, and program will be stuck (deadlock).



To prevent thread execution : join()

```
public class MyThread extends Thread {
    @Override
    public void run() {
        for (int i =0; i<=10; i++){
            System.out.println("child thread");
            try {
                Thread.sleep(2000);
            }catch (Exception e){
            }
        }
    }
    public static void main(String[] args) throws InterruptedException {
        MyThread t = new MyThread();
        t.start();
        t.join();

        for (int i = 0; i<=10; i++){
            System.out.println("main thread");
        }
    }
}
```

To prevent thread execution : join()



waiting state
1. If time expires
2. If sleeping thread got interrupted
3. If the 't' completes execution
t.join();
t.join(2000);
New
t.start();
Runnable
If Thread Schedular allocates processor
Running
If runs completes
Dead

MyThread t = new MyThread();

interrupt() :

- It is used to interrupt an executing thread.

- It only works when the thread is in sleeping or waiting state. When we use this method it throws InterruptionException.
- If a thread is not in sleeping or waiting state then calling an interrupt() method will perform normal behaviour.

interrupt() :

```
public class Interrupt extends Thread{

        @Override
        public void run() {
            try {
                for (int i = 1; i<= 5; i++){
                        System.out.println(i);
                        Thread.sleep(1000);
                }
           }catch (Exception e){
                System.out.println("Thread interrupted : " + e);
            }

        }

        public static void main(String[] args) {

            Interrupt t = new Interrupt();
            t.start();
            t.interrupt();
        }
}
```

Output :

```
1
Thread interrupted :
java.lang.InterruptedException:
sleep interrupted
```

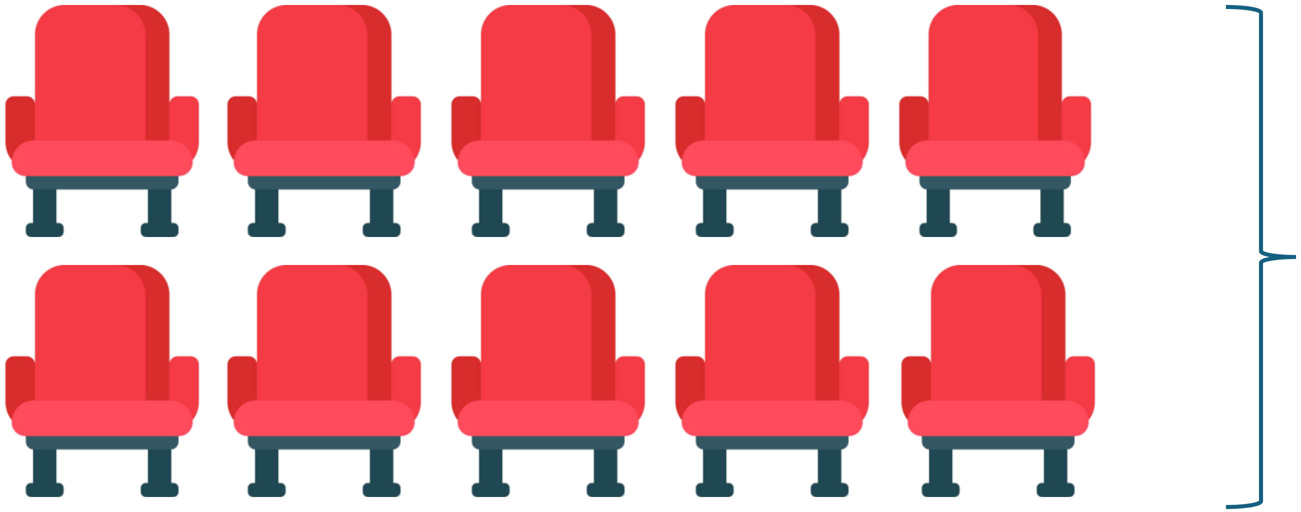
Synchronization

- **Synchronization** in Java is a mechanism used to control access to shared resources by multiple threads. It ensures that only one thread can access a critical section of code at a time, preventing race conditions and ensuring data consistency.

- **Advantages** : No data inconsistency problem
  No thread interference

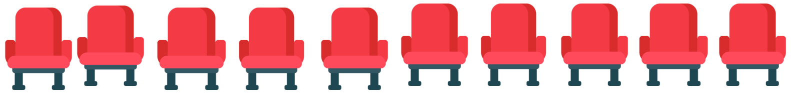
- **Disadvantages** : Increase waiting period of threads
  Create performance problems

Let us consider one example where we are developing application for booking seats in theatre.

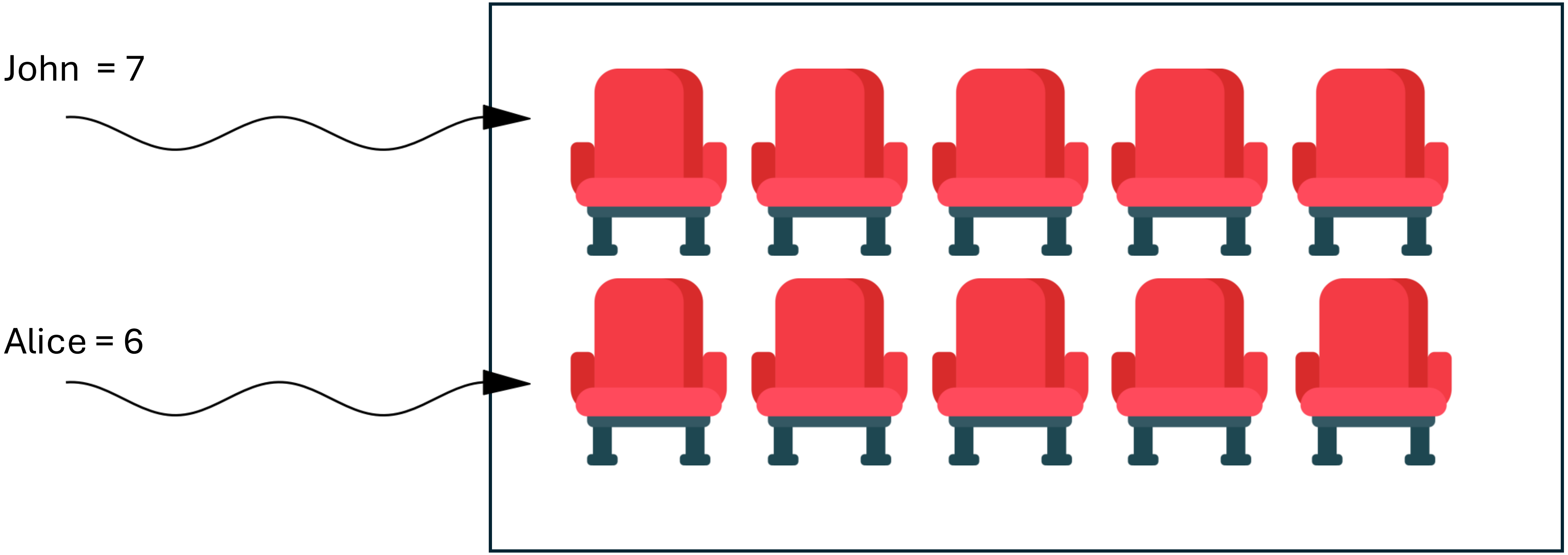


Total Seats = 10

```java
public class Seat {

    int total_seats = 10;

        void bookSeat(int seats){
            if (total_seats >= seats){
                System.out.println(seats + " seats booked successfully");
                total_seats = total_seats - seats;
                System.out.println(total_seats + " seats are left");
            }else {
                System.out.println("Sorry seats cannot be booked....!");
                System.out.println("Seats left : " + total_seats);
            }
    }
}
```



```java
public class BookSeat extends Thread {

    static Seat s;
    int seats;

    @Override
     public void run() {
        s.bookSeat(seats);
     }

     public static void main(String[] args) {
       s = new Seat();

       BookSeat john = new BookSeat();
       john.seats = 7;
       john.start();
       BookSeat alice = new BookSeat();
       alice.seats = 6;
       alice.start();
     }
}
```

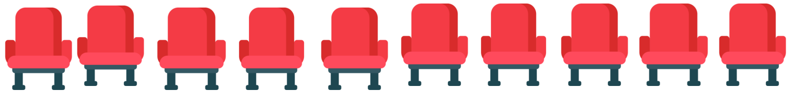


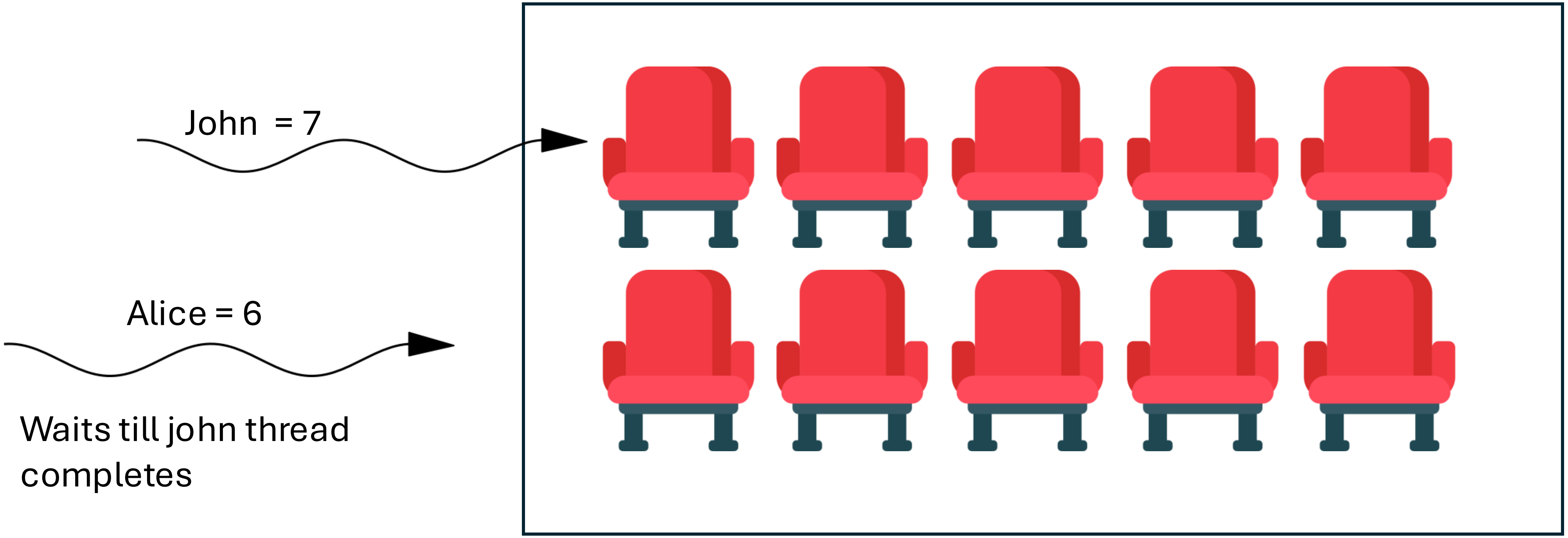
John = 7

Alice = 6

Seats

```
public class Seat {

    int total_seats = 10;

        synchronized void bookSeat(int seats){
            if (total_seats >= seats){
                System.out.println(seats + " seats booked successfully");
                total_seats = total_seats - seats;
                 System.out.println(total_seats + " seats are left");
            }else {
                 System.out.println("Sorry seats cannot be booked....!");
                 System.out.println("Seats left : " + total_seats);
             }
    }
}
```



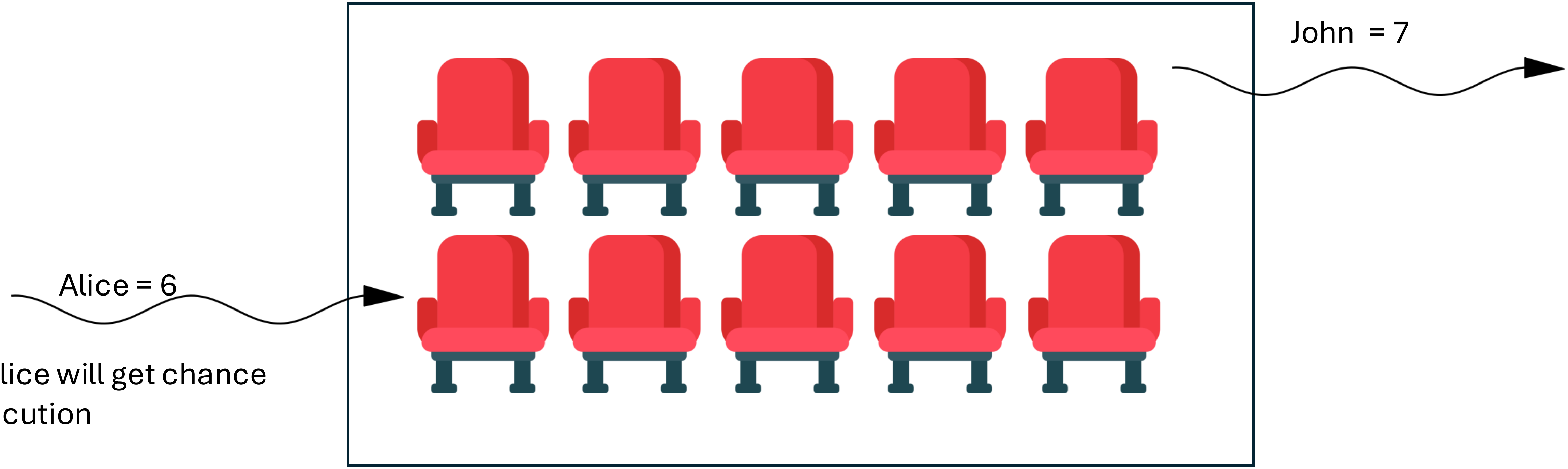


John = 7

Alice = 6

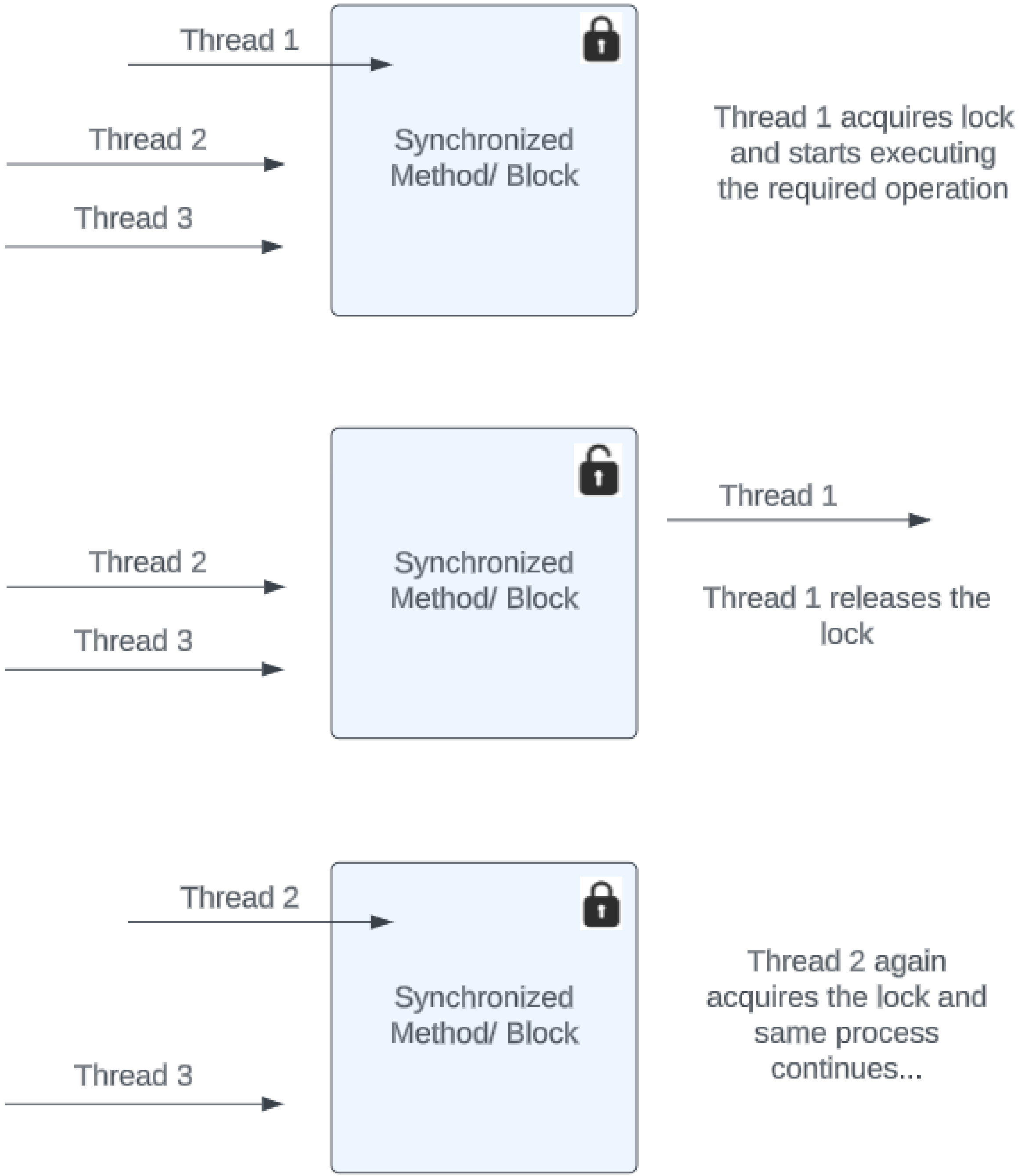
Waits till john thread completes

Seats



Now Alice will get chance for execution

Seats


Thread 1
Thread 2
Thread 3
Synchronized Method/ Block
Thread 2
Thread 3
Synchronized Method/ Block
Thread 1
Thread 2
Thread 3
Synchronized Method/ Block

Thread 1 acquires lock and starts executing the required operation

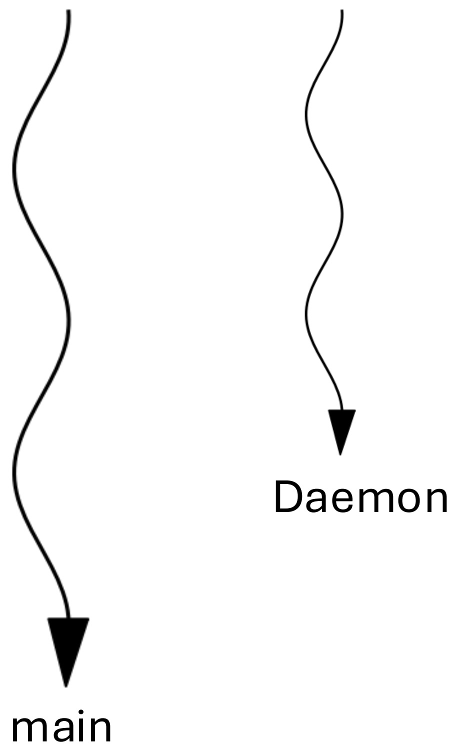
Thread 1 releases the lock

Thread 2 again acquires the lock and same process continues...

Daemon Thread

- Daemon thread in java is a service provider thread that provides services to the user thread, or which runs at the background of another thread.

- Best example : Garbage collector



```java
public class Test extends Thread{

   @Override
    public void run() {
       System.out.println("Child Thread");
    }

    public static void main(String[] args) {
      System.out.println("Main thread");
      Test t = new Test();
      t.setDaemon(true);
      System.out.println(t.isDaemon());
      t.start();
    }
}
```

Output :

```
Main thread
true
```

Daemon thread (Important notes) :

- We must create the daemon thread before starting of the thread if not it throws IllegalThreadStaticException.
- Most of the times daemon thread have low priority.
- Here in above example if we do not write System.out.println ("Main thread");
  then it will not execute run as daemon runs behind main and main is doing nothing so daemon can't provide service.
- Life of daemon thread depends upon another thread.
- In this example, the child thread does not print because the main thread finishes execution before the JVM schedules the child thread

Inter – thread communication

- It is a mechanism in which a thread releases the lock and enter paused state, and another thread acquires the lock and continue to executed.

- It is implemented by the following methods present in java.lang.Object

  wait();

  notify();

  notifyAll();

- To call this methods on any object, thread should own the lock of that object i.e the thread should be inside synchronized area.

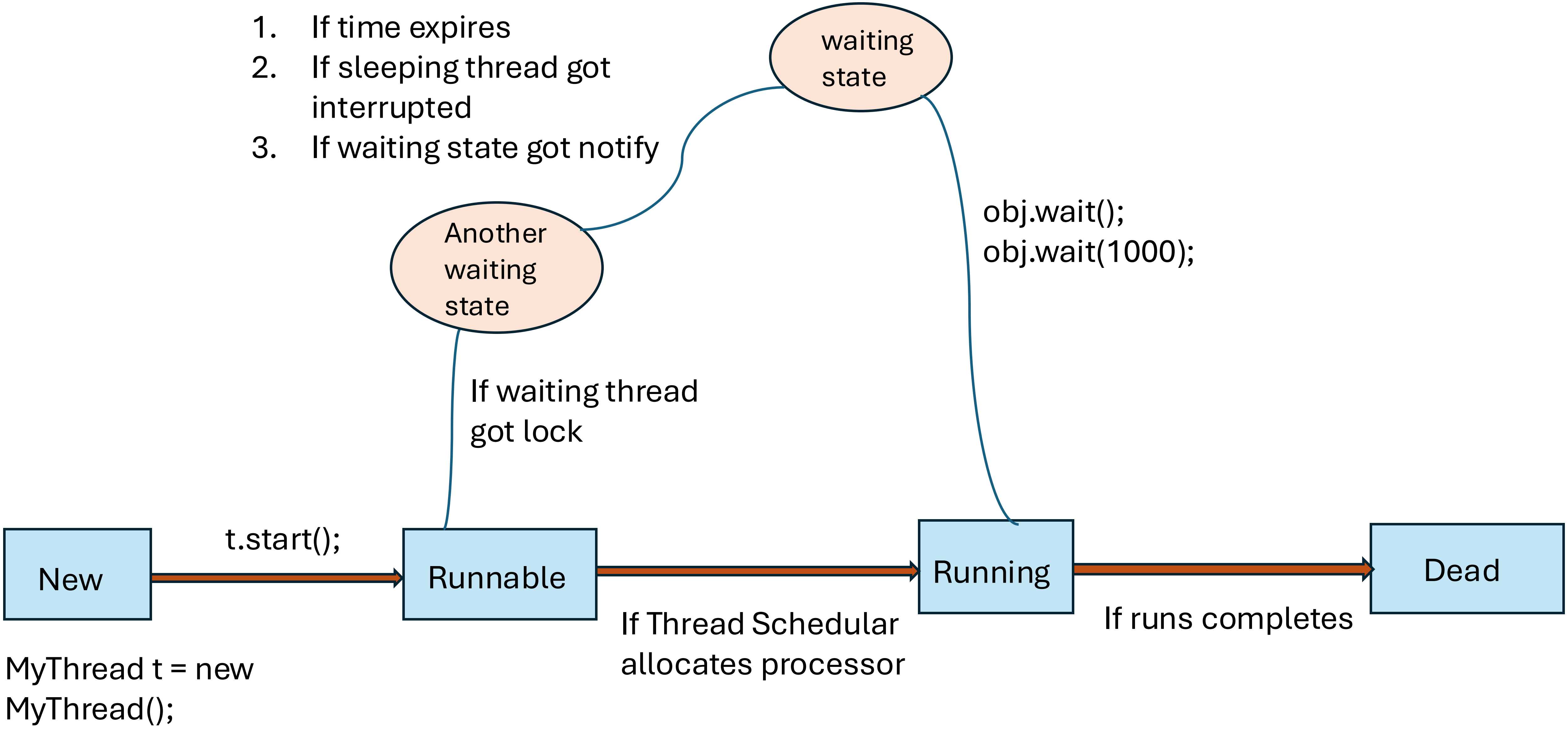
Inter – thread communication

- wait() - If any thread calls wait() method, it causes the current thread to release the lock and wait until another thread invokes the notify() or notifyAll() method for an object, or a specified amount of time has elapsed.

- notify() - This method is used to wake up a single thread and releases the object lock.

- notifyAll() - This method is used to wake up all threads that are in waiting state.

Inter – thread communication



1. If time expires
2. If sleeping thread got interrupted
3. If waiting state got notify

waiting state

Another waiting state

obj.wait();
obj.wait(1000);

If waiting thread got lock

New

t.start();

Runnable

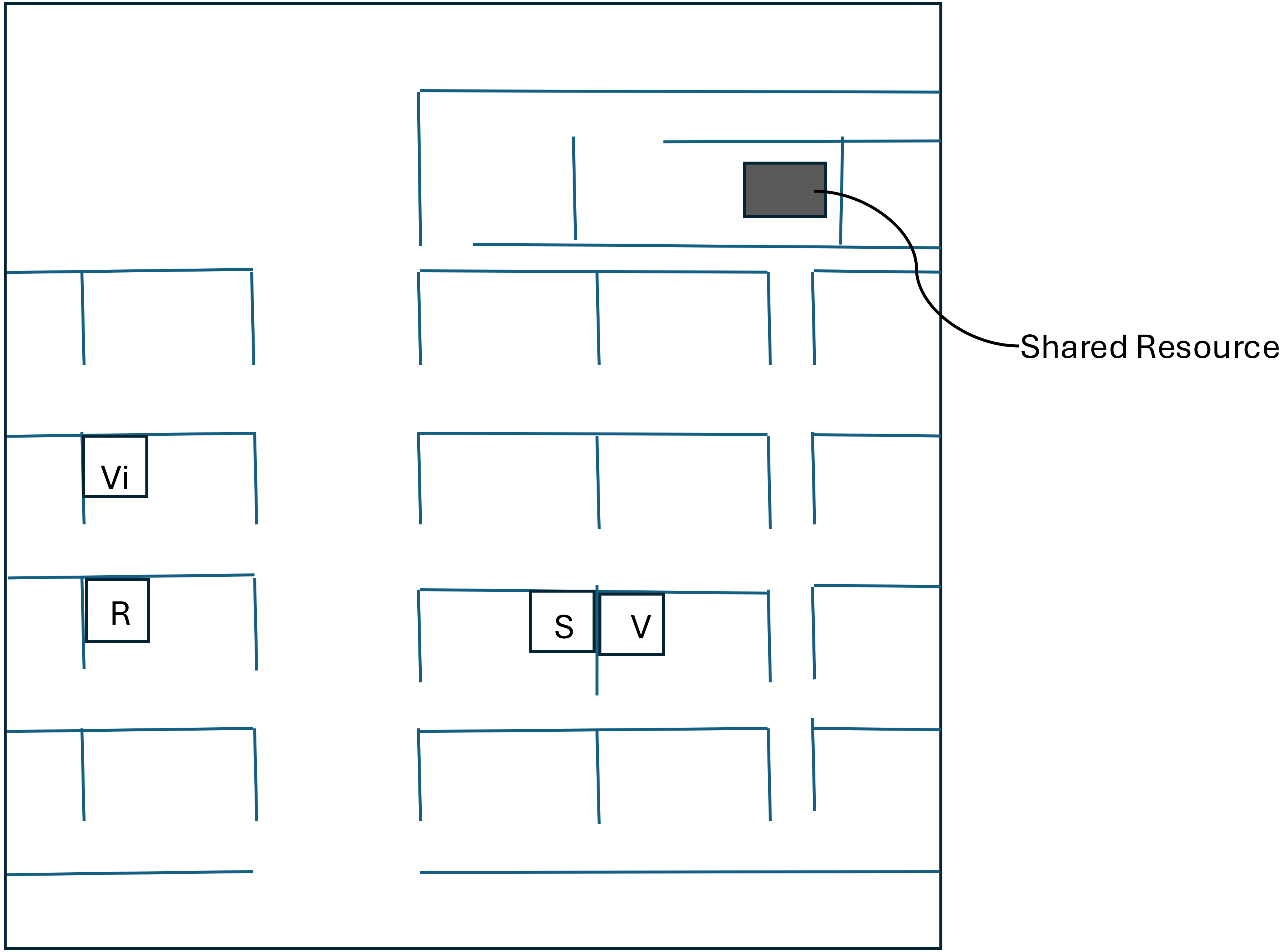
Running

Dead

If Thread Schedular allocates processor

If runs completes

MyThread t = new MyThread();


Shared Resource
Vi
R
S
V

Lock

- In java 1.5 java.util.concurrent.locks package was introduced.

- Lock is an interface.

- Lock implementations provide more extensive operations than traditional implicit locks.

- Methods : void lock();
  boolean tryLock();
  boolean tryLock(long time, TimeUnit unit);
  void unlock();

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class MyThread extends Thread {

    private static final Lock lock = new ReentrantLock();
    @Override
     public void run() {
      // Acquire the lock
         lock.lock();
        try {
         System.out.println(Thread.currentThread().getName() + " has acquired the lock.");
         // Simulate some work
         Thread.sleep(6000);
         System.out.println(Thread.currentThread().getName() + " is executing.");
         } catch (InterruptedException e) {
         e.printStackTrace();
         } finally {
         // Release the lock
         System.out.println(Thread.currentThread().getName() + " has released the lock.");
         lock.unlock();
         }
     }

    public static void main(String[] args) {
     MyThread t1 = new MyThread();
     MyThread t2 = new MyThread();

     t1.start();
     t2.start();
     }
}
```

Reentrant Lock

- It is the implementation class of Lock interface, and it is direct child class of Object.

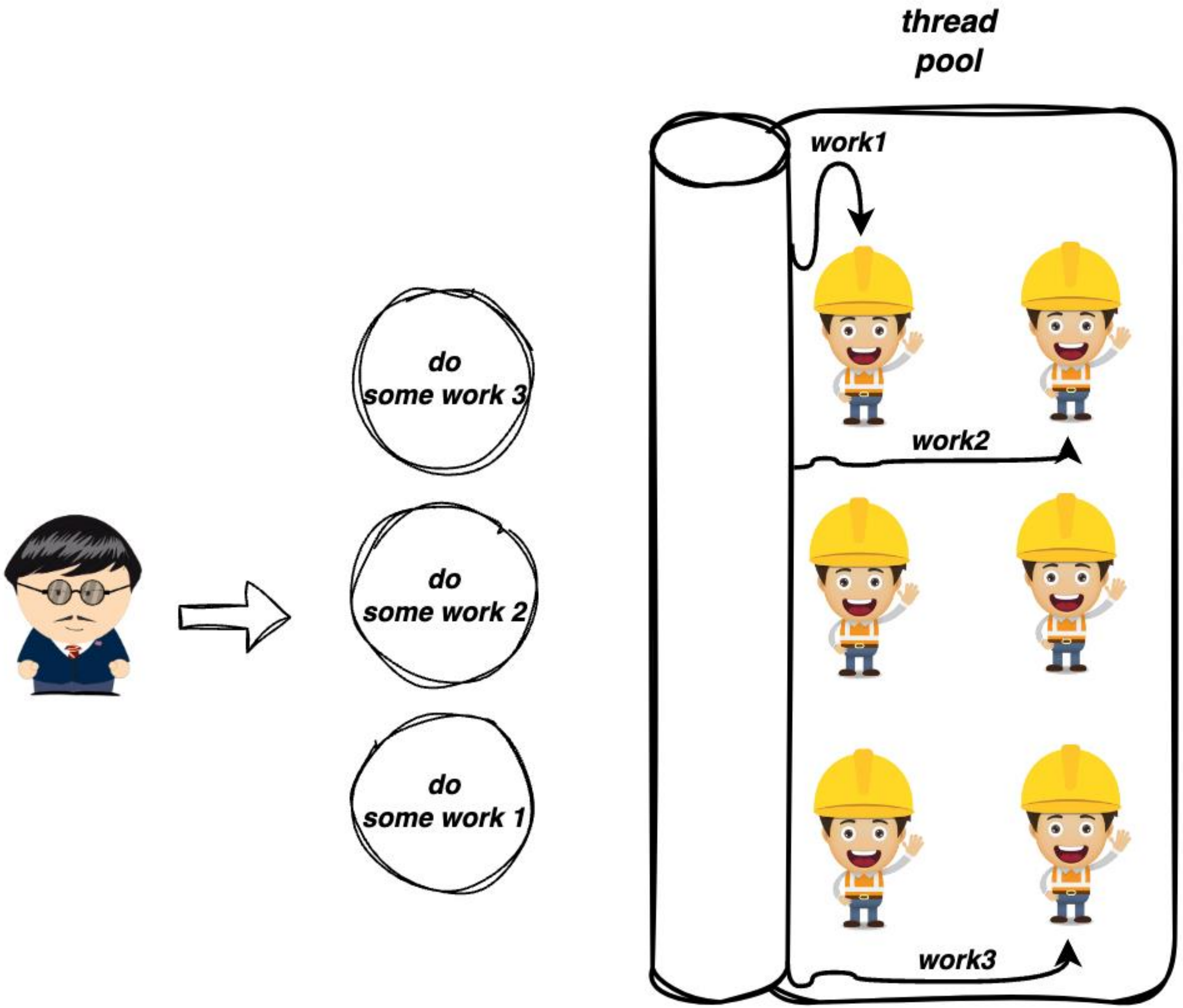
- Reentrant means a thread can acquire same lock multiple times without any issue.

- Internally ReentrantLock increments threads personal count whenever we call lock method and decrement the count value whenever threads call unlock method, and lock will be released when count reaches 0.

```java
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockDemo {
     public static void main(String[] args) {
         ReentrantLock l = new ReentrantLock();
         l.lock();
         l.lock();
         System.out.println(l.isLocked());                   //true
         System.out.println(l.getHoldCount());               //2
         System.out.println(l.isHeldByCurrentThread());      //true
         l.unlock();
         System.out.println(l.getHoldCount());               //1
         System.out.println(l.isLocked());                   //true
         l.unlock();
         System.out.println(l.isLocked());                   //false
         System.out.println(l.isFair());                     //false

     }
}
```

Thread Pools (Executor Framework)

- Creating a new thread for every job may create performance on memory problems. To overcome this we should go for Thread pool.
- Thread pool is the pool of already created threads ready to do our job.
- Java 1.5 version introduces Thread pool framework to implement Thread pools.
- Thread pool framework is also known as Executor framework.



thread pool

do some work 3

do some work 2

do some work 1

work1

work2

work3