Helwan University

Faculty of Computer Science

Programming Language 2

Object oriented design using JAVA

Dr. Ayman Ezzat

Email: ayman@fcih.net

Web: www.fcih.net/ayman

Unit 8: threads

Outline

1. Introduction

2. Creating threads

1. Introduction

- **Threads** are separate activities within a program, each of which has a beginning and an end.
  - Example: You might want to monitor the keyboard for a key being pressed by a user and, at the same time, track the movement of the mouse by the user and repaint the screen. Each of these tasks can be thought of as a single thread in program.

- The Java system has a component known as the **scheduler** that makes decisions about which thread to run next.

- Threads **are not the actual static code** itself but rather they **are the dynamic process** of executing that code

- So far, all of the programs that you have written have had only **one thread**
  - and this has been the thread started by the **main method**.

1. Introduction

- Concurrent systems
  - It is an area of computer science that represents the concepts surrounding the use of threads.
  - **a concurrent system** consists of a collection of separate activities or tasks that are simultaneously at some stage between their starting and finishing point.

- In a concurrent system, each task is made into a thread and …
  - **on a multiprocessor computer**, each thread can truly run at the same time as all of the other threads (up to the number of processors available, of course).
  - **on a single-processor computer**, the CPU can carry out only one set of instructions at a time. In this situation the threads share the CPU, and the operating system will allocate small blocks of time to each thread.

2. Creating threads

- A thread is treated as an object in Java and the **`Thread`** class is found in the **`java.lang`** library.
  - The Thread class has a number of methods for creating, destroying and modifying threads.

- Threaded classes can be defined in two ways:

  (1) by **inheriting** from the **`Thread`** class
  - This method is used if you wish to inherit from only one class, the Thread class.
  - Note that Java does not allow multiple inheritance

  (2) or by **implementing** the **`Runnable`** interface.
  - Defining threads by implementing the Runnable interface is used when you want to inherent from a class other than the Thread class.

2. Creating threads

(1) Inheriting from the `Thread` class

- The general approach is
  - **(1) Define a class** by **extending** the **`Thread`** class and **overriding** the **`run`** method.
    - In the run method, you should write the code that you wish to run when this particular thread has started.
  - **(2) Create an instance** of the above class
  - **(3) Start running the instance** using the **`start`** method that is defined in **`Thread`**.

Example

-

```
public class WhereAmI extends Thread {
    int n;
// constructor
    public WhereAmI(int number) {
        n = number;
    }
// override the run method
    public void run() {
        for (int i = 0; i < 100; i++)
            System.out.println("I'm in thread " + n);

    }
}
```

```
public class ThreadTester {
    public static void main(String[] args) {
        // create the threads
        WhereAmI place1 = new WhereAmI(1);
        WhereAmI place2 = new WhereAmI(2);
        WhereAmI place3 = new WhereAmI(3);
        // start the threads
        place1.start();
        place2.start();
        place3.start();
    }
}
```

The output

```
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
```

2. Creating threads

(2) Implementing the `Runnable` interface

- The general approach is
  - **(1) Define a class** that **implements** `Runnable` and **overriding** the `run` method.
  - **(2) Create an instance** of the above class.
  - **(3) Create a thread** that runs this instance.
  - **(4) Start running the instance** using the `start` method.

Example

```
public class WhereAmI2 implements Runnable{
    int n;
    // constructor
    public WhereAmI2(int number) {
        n = number;
    }
    // override the run method
    public void run() {
        for (int i = 0; i < 100; i++)
            System.out.println("I'm in thread " + n);

    }
}
```

```
public class ThreadTester2 {
    public static void main(String[] args) {
        // create a runnable objects,
        // and the thread to run them.
        WhereAmI2 place1 = new WhereAmI2(1);
        Thread thread1 = new Thread(place1);
        WhereAmI2 place2 = new WhereAmI2(2);
        Thread thread2 = new Thread(place2);
        WhereAmI2 place3 = new WhereAmI2(3);
        Thread thread3 = new Thread(place3);
        // start the threads
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

The output

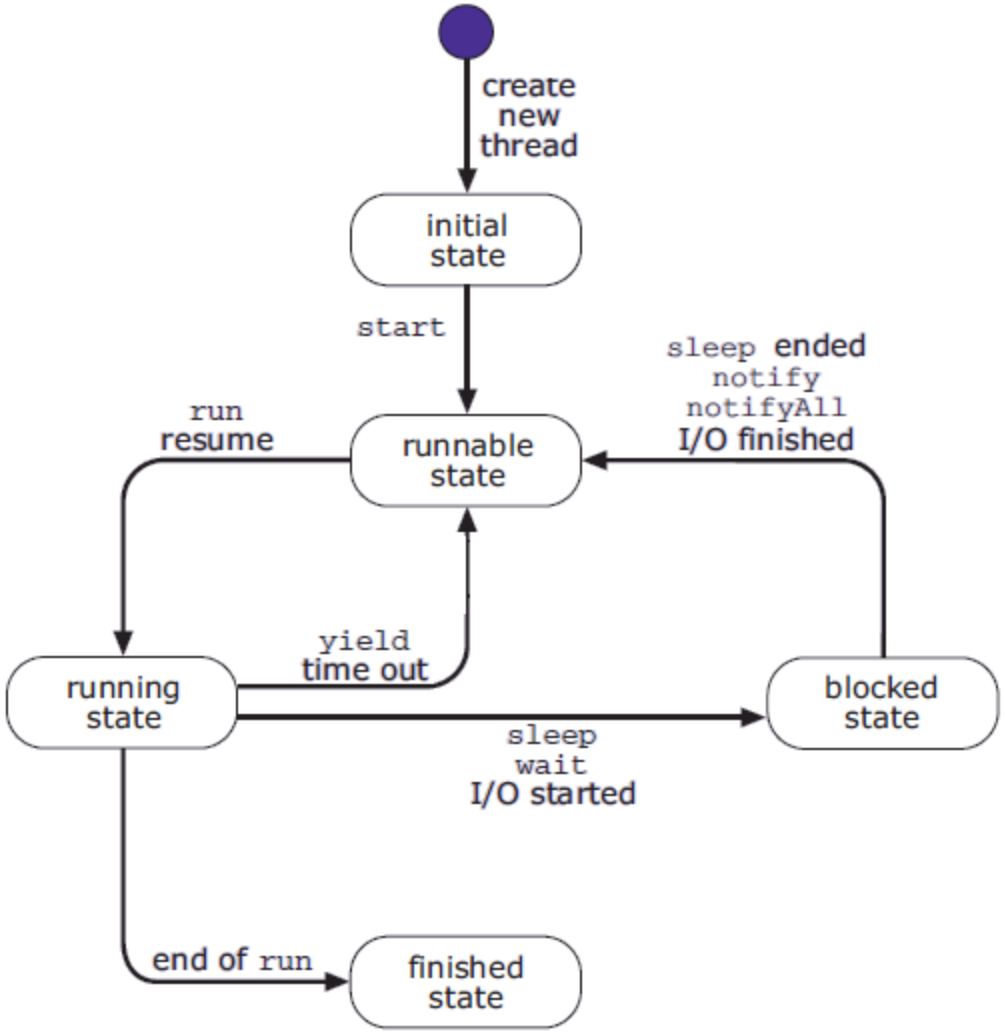
```
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
```

2. Creating threads

Thread states

- We need to look at the various states that a thread can be in and how a thread might move from one state to another.
- A thread is in one of **five states** at any moment in time:
  (1) initial state;
  (2) runnable state;
  (3) running state;
  (4) blocked state;
  (5) finished state.

The figure shows the possible transitions between states.



2. Creating threads

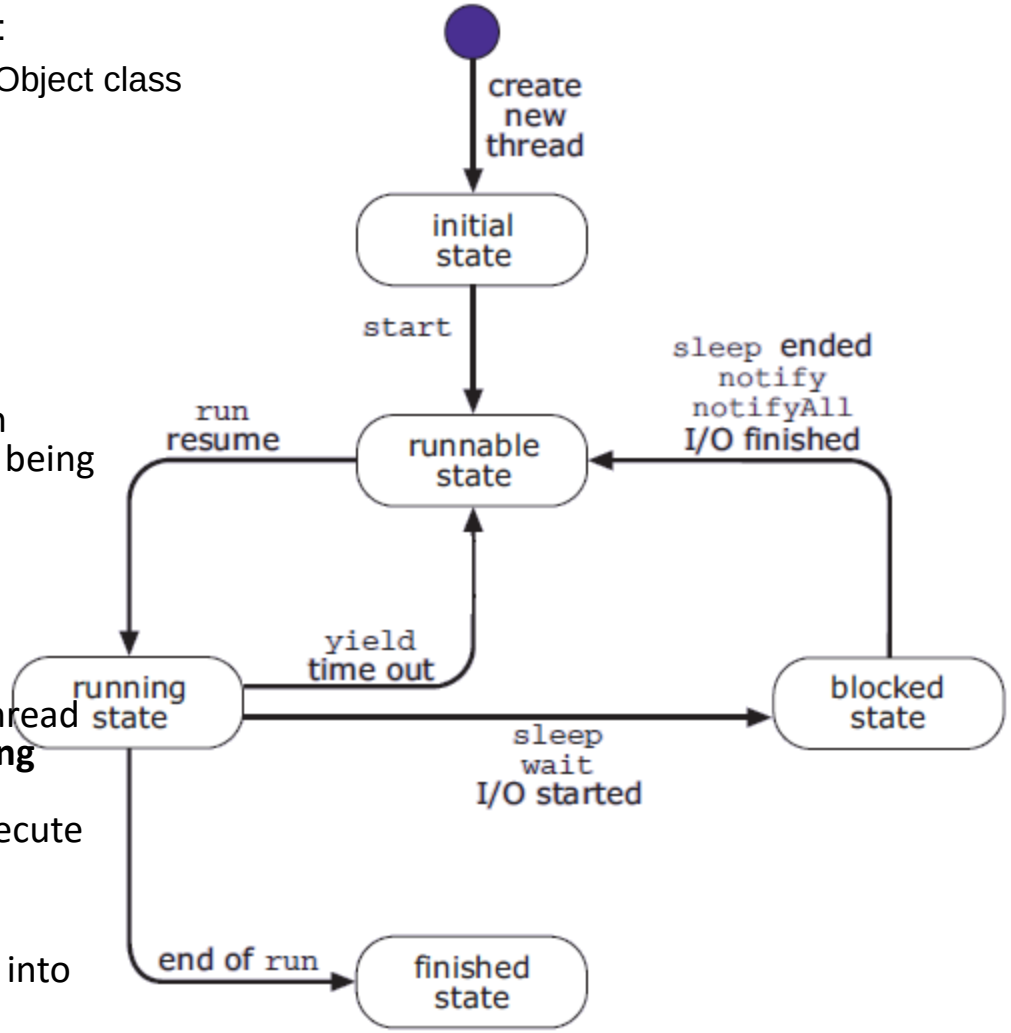
Movement from one state to another is governed by:

(1) a number of methods from Thread class and the Object class

(2) the operating system and other external factors.

Note the following in the figure:

- A thread is put into the **initial state** by means of it being created using **`new`**.
- the **`start`** method moves the thread into the **runnable state**. This does not mean that it will run immediately (there may be another thread that is being run by the CPU)
- At certain times during the execution of a program that contains threads, the Java system will have a look at all the runnable threads and will select one to execute. Only the thread that is actually executing is said to be in the **running state**. When a particular thread is chosen, its run method is invoked and the thread can begin to execute the code. It is now in the running state.
- When a running state finishes executing, it moves into the **finished state**.



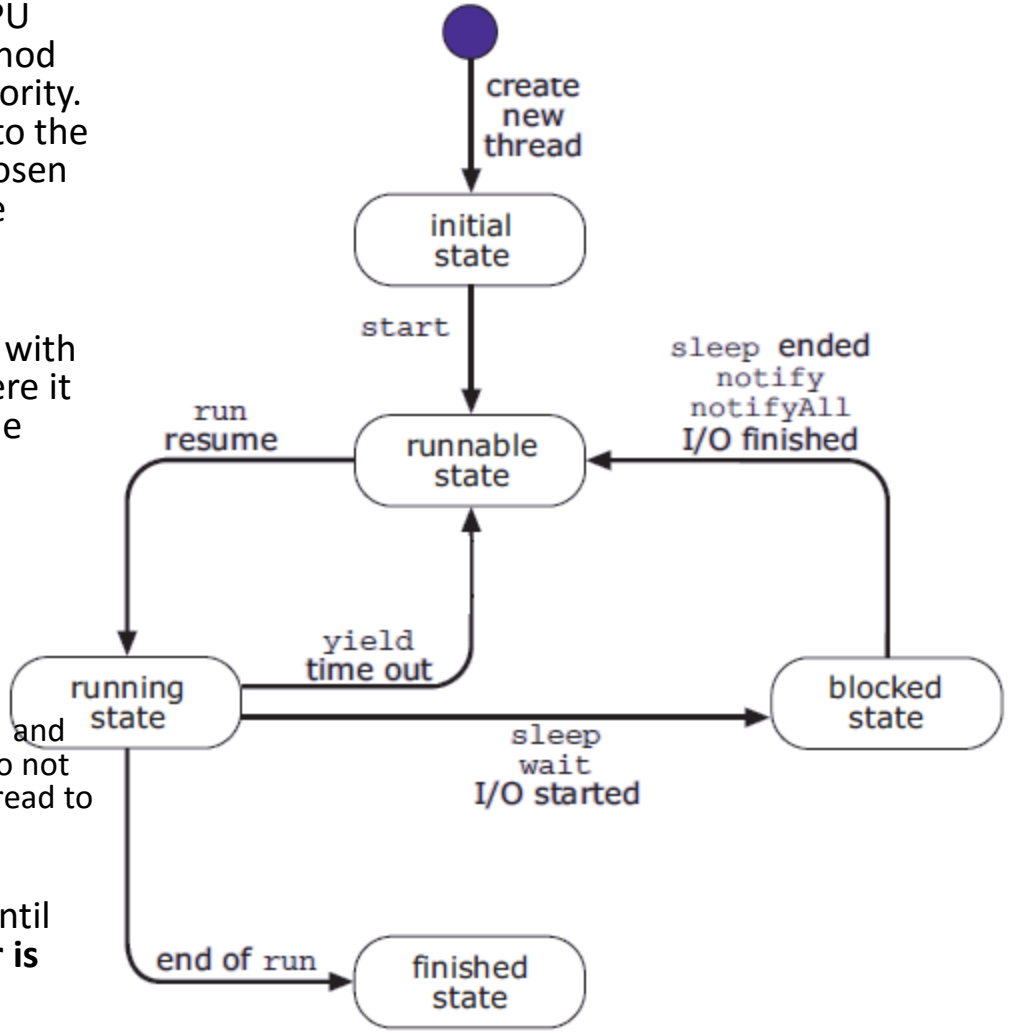
2. Creating threads

- However, the thread may run out of its allotted CPU time. Alternatively, it may invoke the `yield` method to give way to other threads of equal or higher priority. Both scenarios will cause the thread to go back into the **runnable state** where it can, at some point, be chosen by the Java run-time system to move back into the **running state.**

- Once it is back in the **running state** it can carry on with the execution of its `run` method at the point where it left off. This is indicated by the **resume** label in the figure.

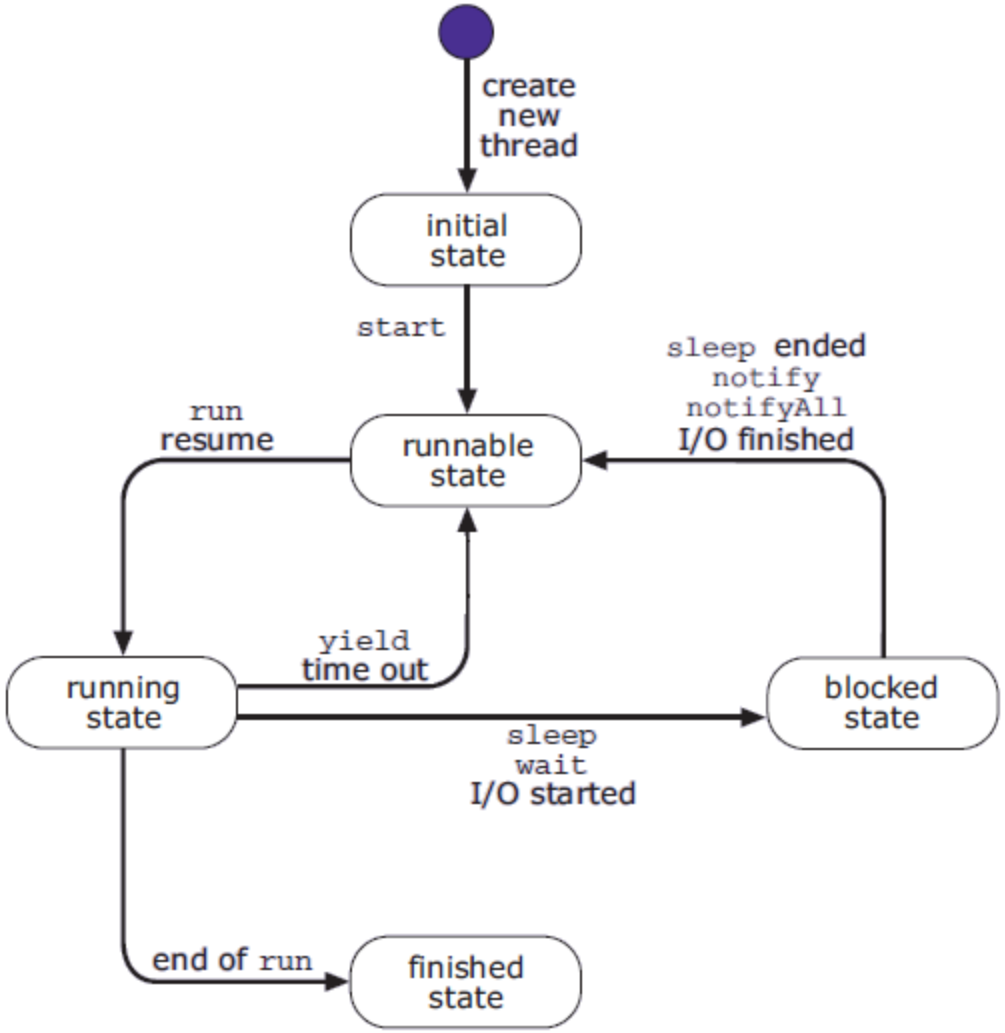
- From the **running state**, a thread may move to the **blocked state** because of **delays in I/O**, or because the `sleep` method or `wait` method is invoked.
  - The `sleep` method is a static method of `Thread` and so can be called in any program, even those that do not use the Thread class. When invoked, it puts the thread to sleep for the specified number of milliseconds.

- In the blocked state, the thread cannot progress until something happens – for example, an **I/O transfer is completed**.



2. Creating threads

- The **`wait`**, **`notify`** and **`notifyAll`** methods:
  - All three methods are inherited from the **Object** class rather than the **Thread** class
  - The **`wait`** method causes a thread to wait for notification by another thread,
  - The **`notify`** and **`notifyAll`** provide such notification.



Outline

1. Introduction
2. Creating threads
3. The problem with shared objects
4. The blocked state

3. The problem with shared objects

Accessing shared resources

- The problem of **shared objects** is relating to the problem of access to **shared resources** that contain shared data.
  - The problem of access to shared resources is not just a problem that affects Java in particular or even just object-oriented programming languages; it has occurred in programming languages ever since computer hardware first enabled us to share resources amongst a number of processes.
  - **Example:** an airline booking system must, somewhere, have a single record of how many seats on a plane have been booked. Yet that single record must be available to hundreds, if not thousands, of computers in travel agent offices around the world who want to check availability and then book seats for their clients.

- To illustrate this problem, we give an **example** in the next few slides!

3. The problem with shared objects

Example:

Consider the following three classes:

- `AirlineBookingTester`
- `NumberOfBookedSeats`
- `TravelAgent`

```
public class AirlineBookingTester {
    public static void main(String[] args) throws InterruptedException {
        NumberOfBookedSeats amount = new NumberOfBookedSeats();
        for (int count = 0; count < 100; count++) {
            TravelAgent thisone = new TravelAgent(amount);
            thisone.start();
        }
        Thread.sleep(1000);
        System.out.println("end total = " + amount.getTotal());
    }
}
```

```
public class NumberOfBookedSeats {
    private int total, newSum;
    public NumberOfBookedSeats() {
        total = 0;
    }
    public void addOne() throws InterruptedException {
        newSum = getTotal() + 1;
        Thread.sleep(2);
        setTotal(newSum);
    }
    public int getTotal() {
        return total;
    }
    public void setTotal(int value) {
        total = value;
    }
}
```

```
public class TravelAgent extends Thread {
    private NumberOfBookedSeats sum;
    public TravelAgent(NumberOfBookedSeats number){
        sum = number;
    }
    public void run() {
        try {sum.addOne();}
        catch (InterruptedException e) {}
    }
}
```

3. The problem with shared objects

Example (Cont'd):

- The class **`AirlineBookingTester:`**
  - Consists of a main method. In the main method, we create an instance of a class called **`NumberOfBookedSeats`**. We also create 100 instances of a class called **`TravelAgent`** which, from the use of the start method, we can see is a **threaded** class. The main method then **sleeps for a second** to allow all of the threads time to complete their processing **before printing out** the total in amount.
    - **Note that in ALL `TravelAgent` classes, we try to add 1 to the same object `amount`.**
- The class **`NumberOfBookedSeats`**
  - As well as a constructor that initializes total to zero, it consists of three methods
    - getter and setter methods, which handle access to **`total`**,
    - and a method called **`addOne`**, which gets the value of total, adds 1 to it, sleeps for a short while to simulate some sort of complex processing or I/O delay, and then uses the set method to update the value of total.
- The **`TravelAgent`** class, which extends Thread.
  - On its creation, it expects to be passed an instance of **`NumberOfBookedSeats`**. The run method is overridden and it invokes the **`addOne`** method of the **`NumberOfBookedSeats`** instance.

3. The problem with shared objects

Example (Cont'd):

- The output:
  - we would expect the `println` in the main method at the end to report that end total = 100.
  - However, when we **actually run the code** this is not what happens. Instead, the total value typically comes out at between 10 and 23 – it varies with each run.
- **The problem** is the use of a **shared data object** – in this case, it is the single instance of the `NumberOfBookedSeats` class.
  - The problem lies in the addOne method of `NumberOfBookedSeats`. Each of the threads invokes this method when they are set running. When the addOne method is invoked by a particular thread, the following three things happen:
    - the `getTotal` method is invoked to fetch the current value of total and add 1;
    - the thread then sleeps to simulate some I/O processing;
    - the `setTotal` method is invoked to update total to the new value.
  - Problems arise because each thread has to share the time with the CPU and so threads are continually being switched between running, runnable and blocked. \
    - Sometimes a thread gets an "old" value of `total` that should have been updated by another thread, but the other thread didn't have the time to update it before it is put in blocked state. After both threads finish executing, the value of `total` is incremented by only 1 instead of 2.

3. The problem with shared objects

Synchronization and locks

- The solution to the shared object problem is to **designate certain methods that access shared data as synchronized**. This is achieved by prefacing the method with the Java keyword `synchronized`.

- Each object in Java is said to have a **lock**. A thread entering a synchronized method gains the lock on the object that contains the method and **no other thread can access any synchronized method in that object until the lock is released** – in this case, by the thread completing the processing inside the method.

- We need to ensure that all methods that access the shared data are modified by the keyword `synchronized` because non-synchronized methods can still have access to the shared data if they are invoked.

3. The problem with shared objects

Synchronization and locks (Cont'd)

- How would you modify the code in the previous example in order to fix the problem and get an output of 100??
  - To add the keyword `synchronized` to all methods that access the variable `total NumberOfBookedSeats` in the class

```
public class NumberOfBookedSeats {
    private int total;
    private int newSum;
    public NumberOfBookedSeats() {total = 0;}
    public synchronized void addOne() {
        newSum = getTotal() + 1;
        try {
            Thread.sleep(2);
        } catch (InterruptedException e) {
            System.out.println("Error " + e.getMessage());
        }
        setTotal(newSum);
    }
    public synchronized int getTotal() {return total;}
    public synchronized void setTotal(int value) {total = value;}
}
```