Paste HTML element by inspecting here!

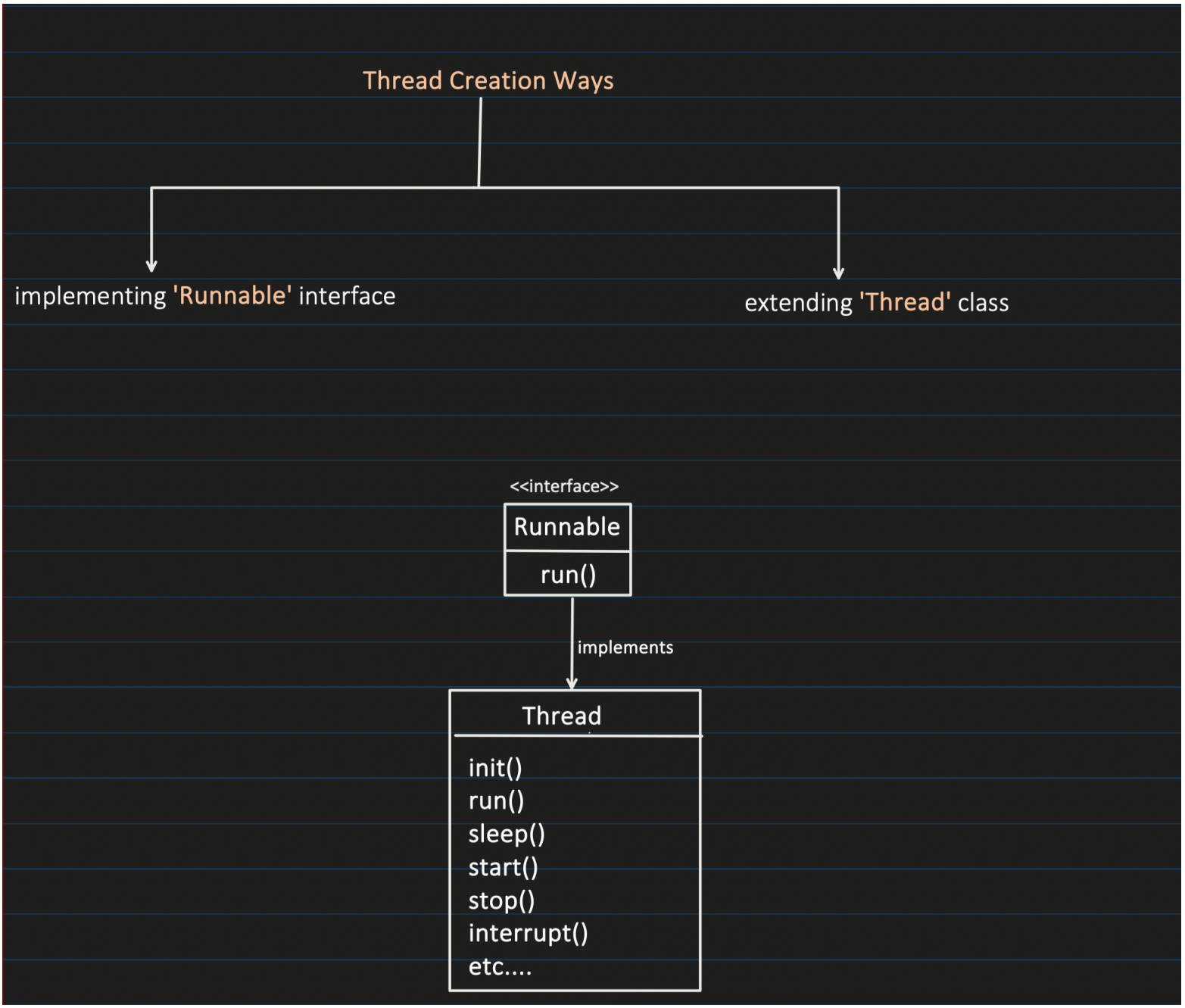
es/74tdo93036fb7dc344bce811fddc55665a092" class="notecardImageClass"
d="74tdo93036fb7dc344bce811fddc55665a092">

/div>

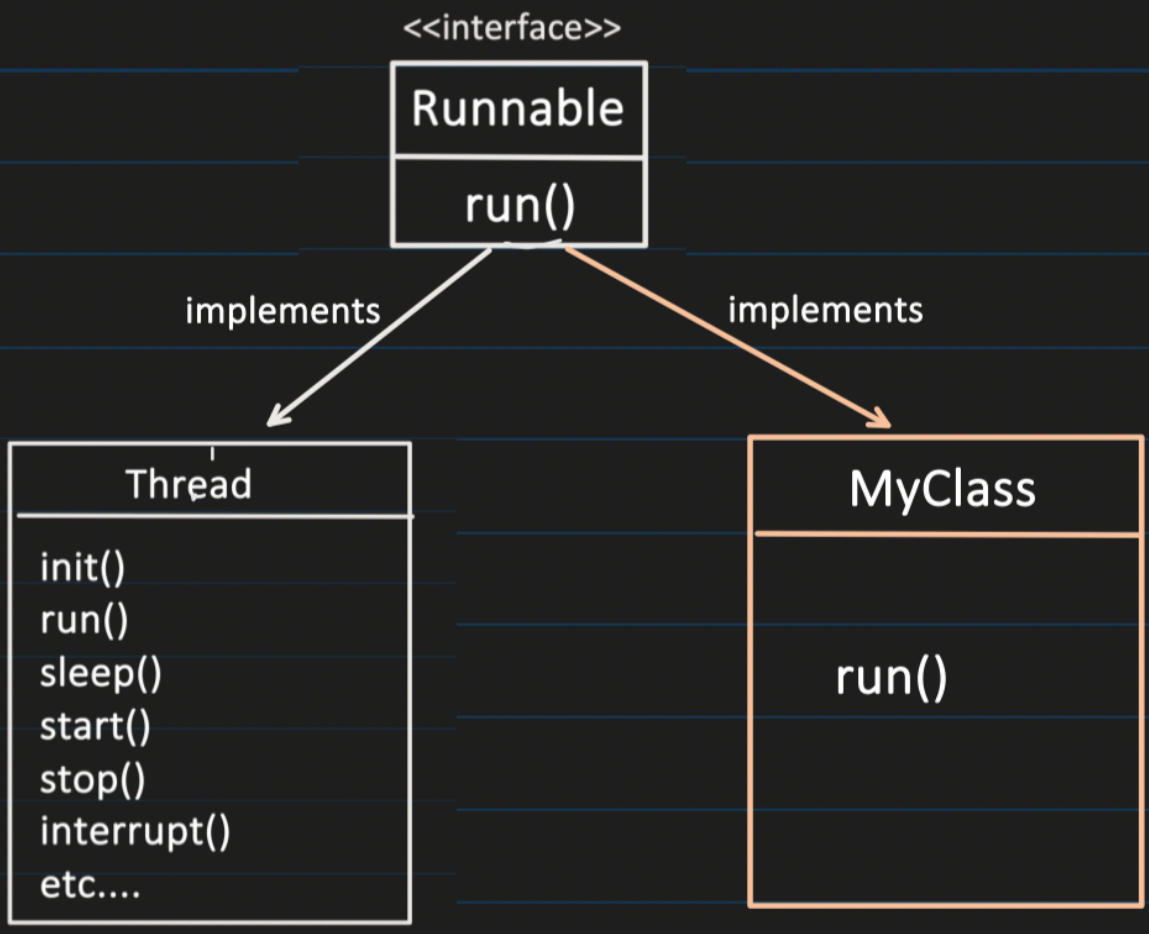
Download as PDF

Multithreading : Part2

"Concept && Coding" YT Video Notes



Implementing 'Runnable' interface:



Step1: Create a Runnable Object

- Create a class that implements 'Runnable' interface.
- Implement the 'run()' method to tell the task which thread has to do.

```
public class MultithreadingLearning implements Runnable{

    @Override
    public void run() {
        System.out.println("code executed by thread: " + Thread.currentThread().getName());
    }
}
```

Step2: Start the thread

- Create an instance of class that implement 'Runnable'.
- Pass the Runnable object to the Thread Constructor.
- Start the thread.

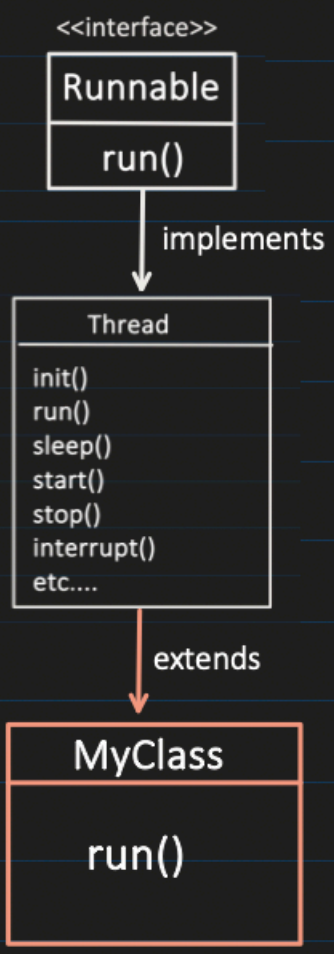
```
public class Main {
    public static void main(String args[]){

        System.out.println("Going inside main method: " + Thread.currentThread().getName());
        MultithreadingLearning runnableObj = new MultithreadingLearning();
        Thread thread = new Thread(runnableObj);
        thread.start();
        System.out.println("Finish main method: " + Thread.currentThread().getName());
    }
}
```

Output:

```
Going inside main method: main
Finish main method: main
code executed by thread: Thread-0
```

extending 'Thread' class:



Step1: Create a Thread Subclass

- Create a class that extends 'Thread' class.
- Override the 'run()' method to tell the task which thread has to do.

```java
public class MultithreadingLearning extends Thread{

    @Override
    public void run() {
        System.out.println("code executed by thread: " + Thread.currentThread().getName());
    }
}
```

Step2: Initiate and Start the thread

- Create an instance of the subclass.
- Call the start() method to begin the execution.

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

        System.out.println("Going inside main method: " + Thread.currentThread().getName());
        MultithreadingLearning myThread = new MultithreadingLearning();
        myThread.start();
        System.out.println("Finish main method: " + Thread.currentThread().getName());
    }
}
```

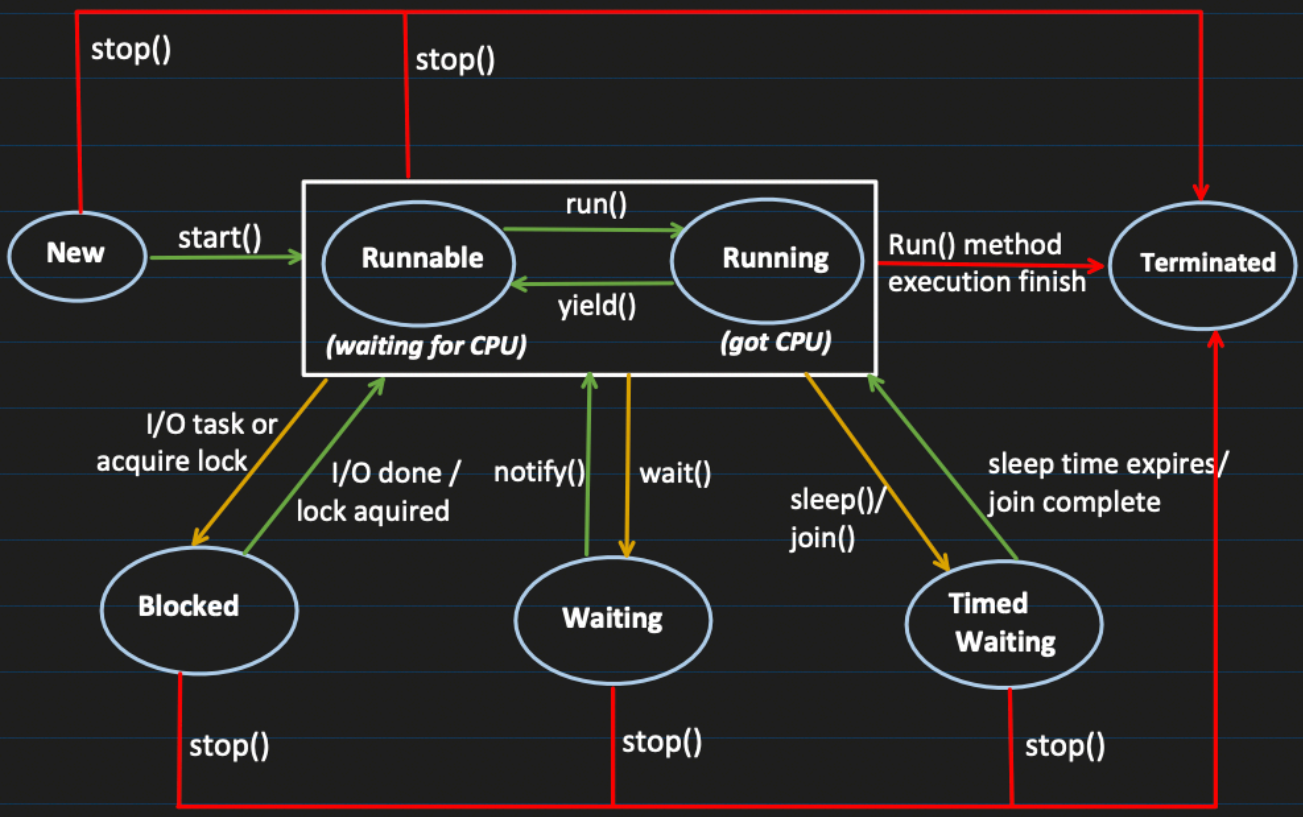
Output:

```
Going inside main method: main
Finish main method: main
code executed by thread: Thread-0
```

Why we have 2 ways to create threads?

- A class can implement more than 1 interface
  but
- A class can extend only 1 class.

Thread Lifecycle



Lifecycle State	Description
New	• Thread has been created but not started • Its just an Object in memory
Runnable	• Thread is ready to run. • Waiting for CPU time.
Running	• When thread start executing its code.
Blocked	Different scenarios where runnable thread goes into the Blocking state: - I/O : like reading from a file or database. - Lock aquired: if thread want to lock on a resource which is locked by other thread, it has to wait. • Releases all the MONITOR LOCKS
Waiting	• Thread goes into this state when we call the wait() method, makes it non runnable. • Its goes back to runnable, once we call notify() or notifyAll() method. • Releases all the MONITOR LOCKS
Timed Waiting	• Thread waits for specific period of time and comes back to runnable state, after specific conditions met. like sleep(), join() • Do not Releases any MONITOR LOCKS
Terminated	• Life of thread is completed, it can not be started back again.

Before seeing an example, lets first understand "MONITOR LOCK"

MONITOR LOCK:
It helps to make sure that only 1 thread goes inside the particular section of code (a synchronized block or method)

```java
public class MonitorLockExample {

    public synchronized void task1() {
        //do something
        try {
            System.out.println("inside task1");
            Thread.sleep( millis: 10000);
        } catch (Exception e) {
            //exception handling here
        }
    }

    public void task2() {
        System.out.println("task2, but before synchronized");
        synchronized (this) {
            System.out.println("task2, inside synchronized");
        }
    }

    public void task3() {
        System.out.println("task3");
    }
}
```

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

    MonitorLockExample obj = new MonitorLockExample();

    Thread t1 = new Thread(() -> {obj.task1();});
    Thread t2= new Thread(() -> { obj.task2();});
    Thread t3 = new Thread(() -> {obj.task3();});

    t1.start();
    t2.start();
    t3.start();
}
```

Now lets see an Example

```java
public class SharedResource {

    boolean itemAvailable = false;

    //synchornized put the monitor lock
    public synchronized void addItem(){
        itemAvailable = true;
        System.out.println("Item added by: " + Thread.currentThread().getName() + " and invoking all threads which are waiting");
        notifyAll();
    }

    public synchronized void consumeItem(){
        System.out.println("ConsumeItem method invoked by: " + Thread.currentThread().getName());

        //using while loop to avoid "spurious wake-up", sometimes because of system noise
        while (!itemAvailable){
            try {
                System.out.println("Thread " + Thread.currentThread().getName() + " is waiting now");
                wait(); //it releases the monitor lock
            }catch (Exception e){
                // handle exception here
            }
        }

        System.out.println("Item Consumed by: " + Thread.currentThread().getName());
        itemAvailable = false;
    }
}
```

```java
public class ProduceTask implements Runnable {

    SharedResource sharedResource;

    ProduceTask(SharedResource resource){
        this.sharedResource = resource;
    }

    @Override
    public void run() {
        System.out.println("Producer thread: " + Thread.currentThread().getName());
        try {
            Thread.sleep( millis: 50001);
        }catch (Exception e){
            //handle any exception here
        }
        sharedResource.addItem();
    }
}
```

```java
public class ConsumeTask implements Runnable{

    SharedResource sharedResource;

    ConsumeTask(SharedResource resource){
        this.sharedResource = resource;
    }

    @Override
    public void run() {
        System.out.println("Consumer thread: " + Thread.currentThread().getName());
        sharedResource.consumeItem();
    }
}
```

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

        System.out.println("Main method start");

        SharedResource sharedResource = new SharedResource();

        // producer thread
        Thread producerThread = new Thread(new ProduceTask(sharedResource));
        // consumer thread
        Thread consumerThread = new Thread(new ConsumeTask(sharedResource));

        //thread is in "RUNNABLE state"
        producerThread.start();
        consumerThread.start();

        System.out.println("Main method end");

    }
}
```

Or use lambda expression, instead of creating ProduceTask and ConsumeTask class

```java
Thread consumerThread = new Thread(() -> {
    System.out.println("Consumer Thread: " + Thread.currentThread().getName());
    sharedResource.consumeItem();
});
```

Assignment: Implement PRODUCER CONSUMER Problem

Question:

Two threads, a producer and a consumer, share a common, fixed-size buffer as a queue.
The producer's job is to generate data and put it into the buffer, while the consumer's job is to consume the data from the buffer.
The problem is to make sure that the producer won't produce data if the buffer is full, and the consumer won't consume data if the buffer is empty.