Q) Can we override start() method?
Yes we can override as it is a non-final method. Below is the valid code No CE, and No RE.

```
class MyThread extends Thread{
        public void run(){
                System.out.println("run");
        }
        public void start(){
                System.out.println("start");
        }
}
```

Q) If we override start() method is custom thread created?
No, custom thread is not created.

Q) Then why Thread class developer not declared start() method as final?
It is project requirement: before starting this thread execution if we want do some validations and calculations to update current custom thread object state, we should override start method in subclass with this validation logic and then we should start custom thread.

This is the reason Thread class developer leaving start as non-final method

Q) How can we start custom thread from overriding method?
We must place *super.start()* at end of overriding start() method.

Given:

```
class MyThread extends Thread{
        public void run(){
                System.out.println("run");
        }
        public void start(){
                System.out.println("start");
        }
        public static void main(String[] args){
                MyThread mt = new MyThread();
                mt.start();
                System.out.println("main");
        }
}
```

Q1) From above program is custom thread created and what is the output?
A) Custom thread is not created. So run() is not executed.
Output is:

```
start
main
```

Q2) If we call run() from overriding start(), in which thread run() is executed and what is the output?

add below start() method in above code

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

A) custom thread is not created, run() is executed in main thread as start() method is executed in main thread.

Output:

```
start
run
main
```

Q3) If we call super.start() in the above updated program, is custom thread created, where run() method is executed, how many times?

add below start method in above code

```
public void start(){
        System.out.println("start");
        run();
        super.start();
}
```

A)

Yes custom is created with name Thread-0.

run() is executed 2 times

1. in main thread because we call it explicity from overriding start method
2. in custom thread because it is implicitly called by JVM becuase of super.start() call

Output:

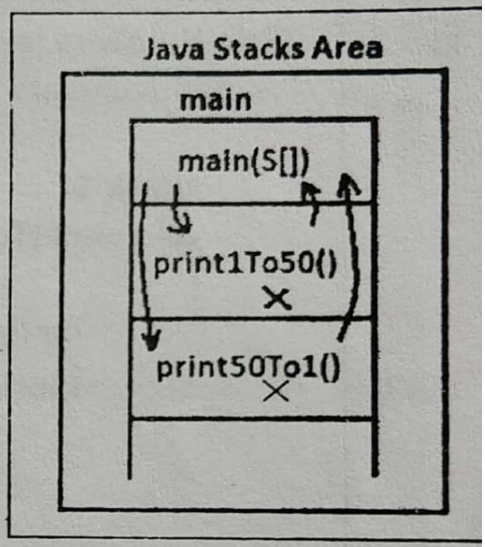
```
start
run
main
run
```

Q4) When should we override start() method in projects as it is given as non-final method?

A) If we want do some validation and calculations to update current custom thread object state before starting this thread execution, we should override start method in subclass with this validation logic and at end of this overriding start() method we must place super.start() to create custom thread.

```
//PrintNumbers.java
class PrintNumbers
{
	//task 1
	void print1To50()
	{
		for (int i = 1; i <= 50 ; i++)
		{

			System.out.print(i + "\t");

			try{ Thread.sleep(100); }
			catch(InterruptedException ie){ ie.printStackTrace(); }

		}
	}

	//task 2
	void print50To1()
	{
		for (int i = 50; i >= 1 ; i--)
		{

			System.out.print(i + "\t");

			try{ Thread.sleep(100); }
			catch(InterruptedException ie){ ie.printStackTrace(); }

		}
	}
}
```

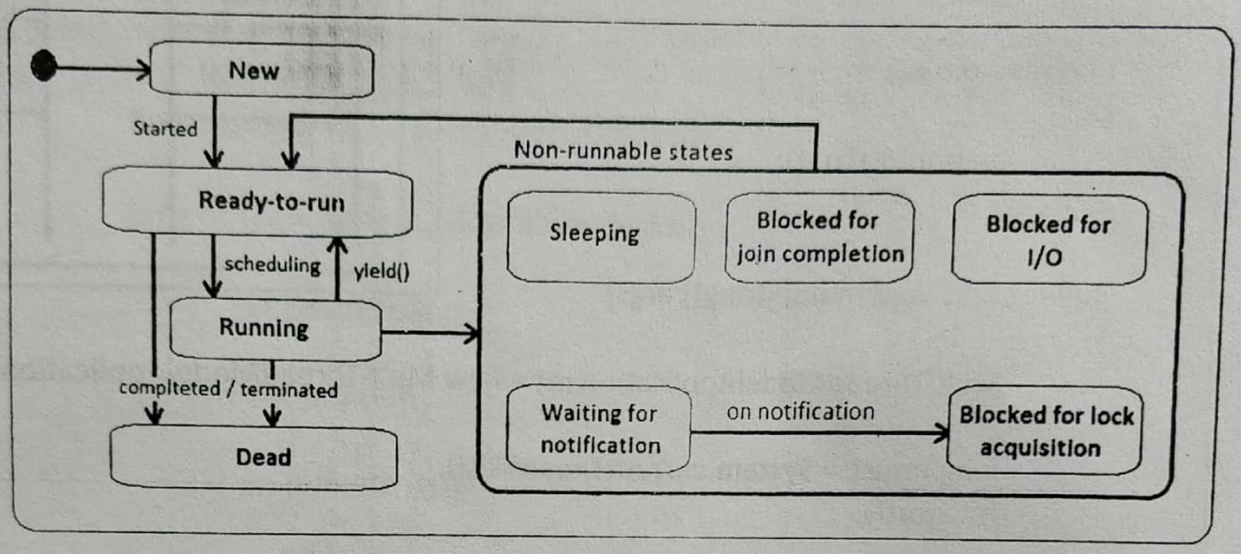
```
//SingleThreadModelApplication.java

class SingleThreadModelApplication
{
    public static void main(String[] args)
    {
        PrintNumbers pn = new PrintNumbers();

        long time1 = System.currentTimeMillis();

            pn.print1To50();

        System.out.println();

            pn.print50To1();

        long time2 = System.currentTimeMillis();

        Sopln("Time taken to complete both tasks: "+((time2- time1) / 1000) +" secs");
    }
}
```



```
//MultiThreadModelApplication.java
class MultiThreadModelApplication extends Thread
{
        static PrintNumbers pn = new PrintNumbers();

        public void run()
        {
                pn.print50To1();
        }

        public static void main(String[] args)
        {
                MultiThreadModelApplication mt = new

                long time1 = System.currentTimeMillis();
                mt.start();

                        pn.print1To50();

                long time2 = System.currentTimeMillis();

                SopIn("Time taken to complete both task
        }
}
```

Threads transition diagram

Understanding the life cycle of a thread is valuable when programming with threads. Threads can exist in different states. Just because of a thread's start() method has been called, it does not mean that the thread has access to the CPU and can start executing straight away. Several factors determine how it will proceed. Once thread is created it is available in any one the below five states. Below Figure shows the states and the transitions in the life cycle of a thread.



- **New state**

  A thread has been created, but it has not yet started. A thread is started by calling its start() method.

- **Ready-to-run state**

  This start is also called Runnable state, also called Queue. A thread starts life in the Ready-to-run state by calling start method and waits for its turn. The thread scheduler decides which thread runs and for how long.

- **Running state**

  If a thread is in the Running state, it means that the thread is currently executing

- **Dead state**

  Once in this state, the thread cannot ever run again.

- **Non-runnable states**

  A running thread can transit to one of the non-runnable states, depending on the circumstances. A thread remains in a non-runnable state until a special transition occurs. A thread does not go directly to the Running state from a non-runnable state, but transits first to the Ready-to-run state.

  The non-runnable states can be characterized as follows:

  - *Sleeping*: The thread sleeps for a specified amount of time.
  - *Blocked for I/O*: The thread waits for a blocking operation to complete
  - *Blocked for join completion*: The thread awaits completion of another thread
  - *Waiting for notification*: The thread awaits notification from another thread
  - *Blocked for lock acquisition*: The thread waits to acquire the lock of an object

Threads execution procedure -> How threads are executed in JVM?
JVM executes Threads based on their *priority* and *scheduling*.

Thread Scheduler
Schedulers in JVM implementations usually employ one of the two following strategies:

- Preemptive scheduling.
  If a thread with a higher priority than the current running thread moves to the Ready-to-run state, the current running thread can be preempted (moved to the Ready-to-run state) to let the higher priority thread execute.

- Time-Sliced or Round-Robin scheduling.
  A running thread is allowed to execute for a fixed length of time, after which it moves to the Ready-to-run state to await its turn to run again.

Thread schedulers are implementation and platform dependent; therefore, how threads will be scheduled is unpredictable.

Thread Priority
Every thread created in JVM is assigned with a priority. The priority range is between 1 and 10.

- 1 is called minimum priority
- 5 is called normal priority
- 10 is called maximum priority.

In Thread class below three variables are defined to represent above three values.

- public static final int **MIN_PRIORITY**;
- public static final int **NORM_PRIORITY**;
- public static final int **MAX_PRIORITY**;

Threads are assigned priorities, based on that the thread scheduler can use to determine how the threads will be scheduled. The thread scheduler can use thread priorities to determine which thread gets to run. The thread scheduler favors giving CPU time to the thread with the highest priority in the Ready-to-run state. This is not necessarily the thread that has been the longest time in the Ready-to-run state.

A thread inherits the priority from its parent thread. The default priority of every thread is normal priority 5, because *main thread* priority is 5.

The priority of a thread can be set using the setPriority() method and read using the getPriority() method, both of which are defined in the Thread class with the below prototype.

```
public final void setPriority(int newPriority)
public final int getPriority()
```

Rule: newPriority value range should between 1 to 10, else it leads to exception

```
// ThreadNameAndPriority.java
class MyThread extends Thread
{
        MyThread()
        {
                super();
        }

        MyThread(String name)
        {
                super(name);
        }
        public void run()
        {
                for (int i = 0; i < 10 ; i++)
                {
                        System.out.println(getName() + " i: "+i);

                }

        }
}
```

```
class ThreadNameAndPriority
{
	public static void main(String[] args)
	{
		MyThread mt1 = new MyThread();
		MyThread mt2 = new MyThread("child2");

		System.out.println("mt1 Thread's initial name and priority");
		System.out.println("mt1 name: "+mt1.getName());
		System.out.println("mt1 priority: "+mt1.getPriority());

		System.out.println();

		System.out.println("mt2 Thread's initial name and priority");
		System.out.println("mt2 name: "+mt2.getName());
		System.out.println("mt2 priority: "+mt2.getPriority());

		mt1.setName("child1");

		mt1.setPriority(6);
		mt2.setPriority(9);

		System.out.println("mt1 Thread's changed name and priority");
		System.out.println("mt1 name: "+mt1.getName());
		System.out.println("mt1 priority: "+mt1.getPriority());

		System.out.println();

		System.out.println("mt2 Thread's changed name and priority");
		System.out.println("mt2 name: "+mt2.getName());
		System.out.println("mt2 priority: "+mt2.getPriority());

		mt1.start();
		mt2.start();

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

```
// CurrentThreadDemo.java
class CurrentThreadDemo
{
	static
	{
		System.out.println("In SB");

		//retrieving currently executing thread reference
		Thread th = Thread.currentThread();
		System.out.println("SB is  exeucting in \""+ th.getName() + "\" thread\n");
	}

	public static void main(String[] args)
	{
		System.out.println("\nIn main method");

		//retrieving currently executing thread reference
		Thread th = Thread.currentThread();

		System.out.println("Original name and priority of main thread");
		System.out.println("current thread name: "+th.getName());
		System.out.println("current thread priority: "+th.getPriority());

		th.setName("xxyy");
		th.setPriority(7);

		System.out.println("\nmodified name and priority of main thread");
		System.out.println("current thread name: "+th.getName());
		System.out.println("current thread priority: "+th.getPriority());

	}
}
```

ThreadGroup

Every thread is created with a thread group. The default thread group name is "***main***". We can also create user defined thread groups using *ThreadGroup* class.

In Thread class we have below method to retrieve current thread's ThreadGroup object reference

public final ThreadGroup getThreadGroup()

In ThreadGroup class we have below method to retrieve the ThreadGroup name

public final String getName()

So in the program we must write below statement to get thread's ThreadGroup name

String groupName = th.getThreadGroup().getName();

For more details on creating ThreadGroup object and placing threads in user defined thread groups check *Application #16*.

---

toString() method in Thread class

In Thread class toString() method is overridden to return thread object information as like below: *Thread[thread name, thread priority, thread group name]*

So its logic in Thread class should be as like below

```
public String toString(){
    return "Thread["+getName() +"," +getPriority() +","+getThreadGroup().getName() +"]";
}
```

Application #10: This application shows Thread class's toString() method functionality.

```
// ToStringDemo.java
class ToStringDemo{
        public static void main(String[] args){
                Thread th1 = new Thread();
                System.out.println(th1);

                Thread th2 = new Thread("child1");
                System.out.println(th2);

                Thread th3 = Thread.currentThread();
                System.out.println(th3);

                th3.setPriority(7);

                Thread th4 = new Thread();
                System.out.println(th4);
        }
}
```