# Java threads and its use in GUI program.

# Outlines

- Part 1 Thread
  - Introduction
  - Creating & Starting the Thread
  - More thread methods
    - getPriority(), setPriority()
    - sleep(), interrupt(), isInterrupted(), isAlive()
    - join(), yield()
  - Thread Pools (Executor)
- Part 2 Thread in JavaFX

# Part 1 : Thread

# Introduction

- Multitasking refers to a computer's ability to perform multiple jobs concurrently
  - more than one program are running concurrently, e.g., UNIX
- A process (job) is a program in execution.
- A thread is a single sequence of execution within program.
- Each thread works independently except:
  - Two threads share the same code when they execute from instances of the same process.
  - Two threads share the same data when they share access to a common object.

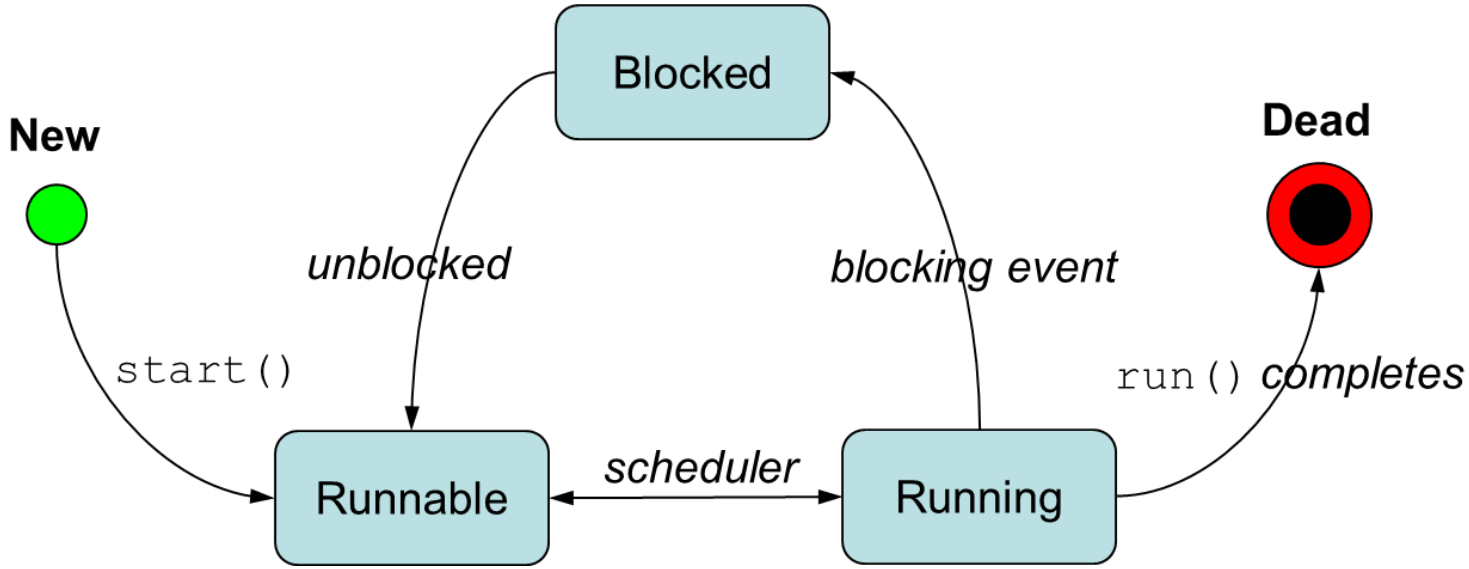
# Introduction (cont.):

- Multithreading refers to multiple threads of control within a single program
  - Each program can run multiple threads of control within it
  - Microsoft Office: “spell checking thread” and “auto-saving thread” are working simultaneously.
  - YouTube: “downloading thread” and “playing thread” are working concurrently.

- Why threads?
  - To maintain responsiveness of an application during a long running task.
  - Some problems are intrinsically parallel.
  - To monitor status of some resource (DB).
  - Some APIs and systems demand it: Swing.

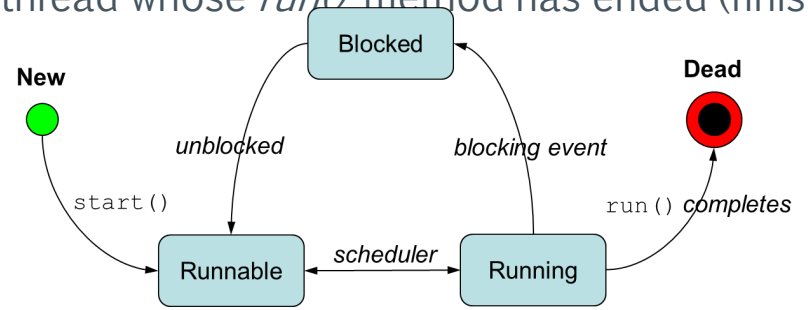
# Introduction (cont.): Simple Life Cycle

- States:
  - NEW, RUNNABLE, RUNNING, BLOCKED, WAITING, TIMED_WAITING, TERMINATED



# Introduction (cont.): Simple Thread States

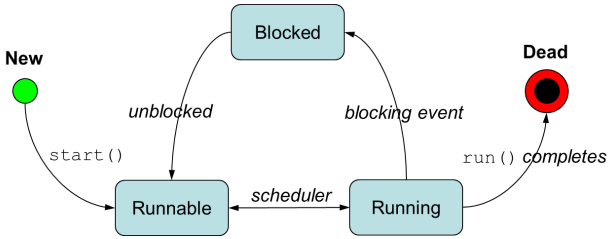
- NEW : A Fresh thread that has not yet started to execute.
- RUNNING : A thread that is actually executing in the Java virtual machine.
- RUNNABLE : A thread that is ready to run but has not been selected by a scheduler.
- BLOCKED : A thread that is blocked waiting for a monitor lock.
- WAITING : A thread that is waiting to be notified by another thread.
- TIMED_WAITING : A thread that is waiting to be notified by another thread for a specific amount of time.
- TERMINATED : A thread whose *run()* method has ended (finished).



# Introduction (cont.): Thread Method Overview

## SIMPLE METHODS

- void setName()
- String getName()
- String getState()
- Thread currentThread()
- void run()
- void start()



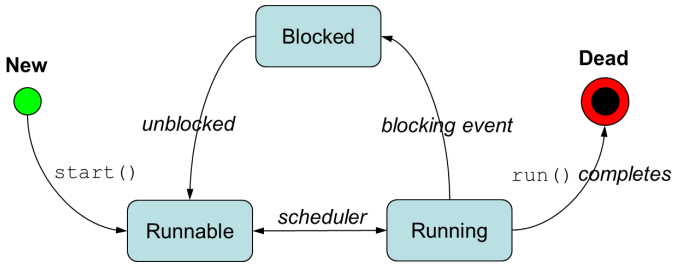

## ADVANCED METHODS

- void wait()
- void notify()
- void notifyAll()
- static void sleep(long mills) throws InterruptedException
- void yield

# Introduction (cont.): OS Scheduling

- The "Running" state depends on the type of Operating System (OS).
- Considering time-slicing, there are two kinds of OS:
- Preemptive OS (e.g., Windows):
  - Each process can run (in the Running state) only a certain time (called time-slice) and, then, it will be moved to the Runnable state to allow other processes to run. This can maximize throughputs.
- Non-preemptive OS (e.g., Solaris):
  - There is no time-slice, so each process can run until it finishes.

# Check States




StateTest.java

```
public class StateTest {

    public static void main(String[] args) {

        Thread t = new Thread();

        System.out.println(t.getState());

        t.start();

        Thread.State s;

        do{

            s = t.getState();

            System.out.println(s);

        }while(s != Thread.State.TERMINATED);

    }

}
```

Result1

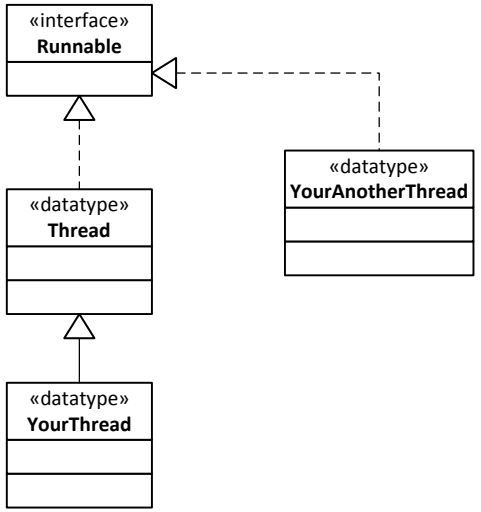
```
NEW

RUNNABLE

RUNNABLE

TERMINATED
```

Result2

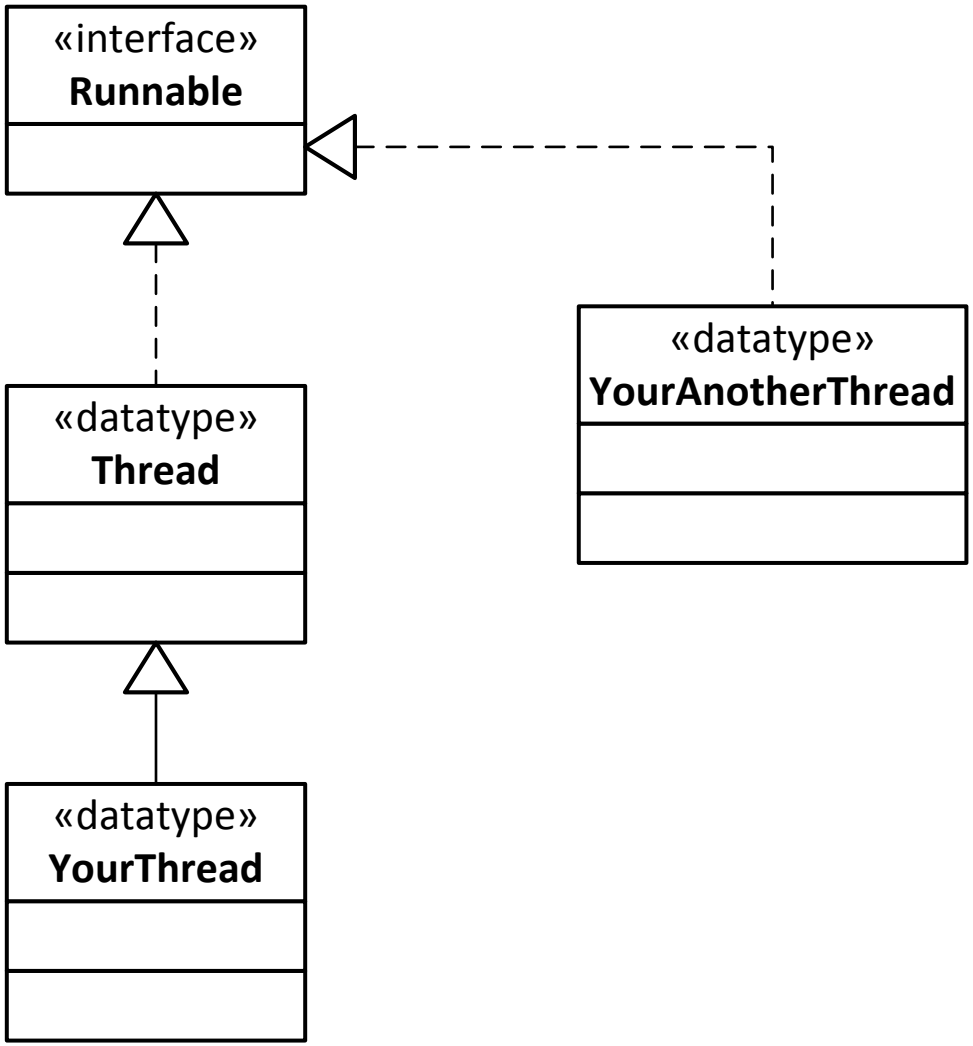
```
NEW
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
RUNNABLE
TERMINATED
```

# Creating & Starting the Thread

- New class **`extends Thread`**
  - simple
  - cannot extend from other class
- Creating a new class that **`implements Runnable`** interface (preferred)
  - better OOD
  - single inheritance
  - consistency
- Overriding **`run()`** method
- Using the **`start`** method to run thread



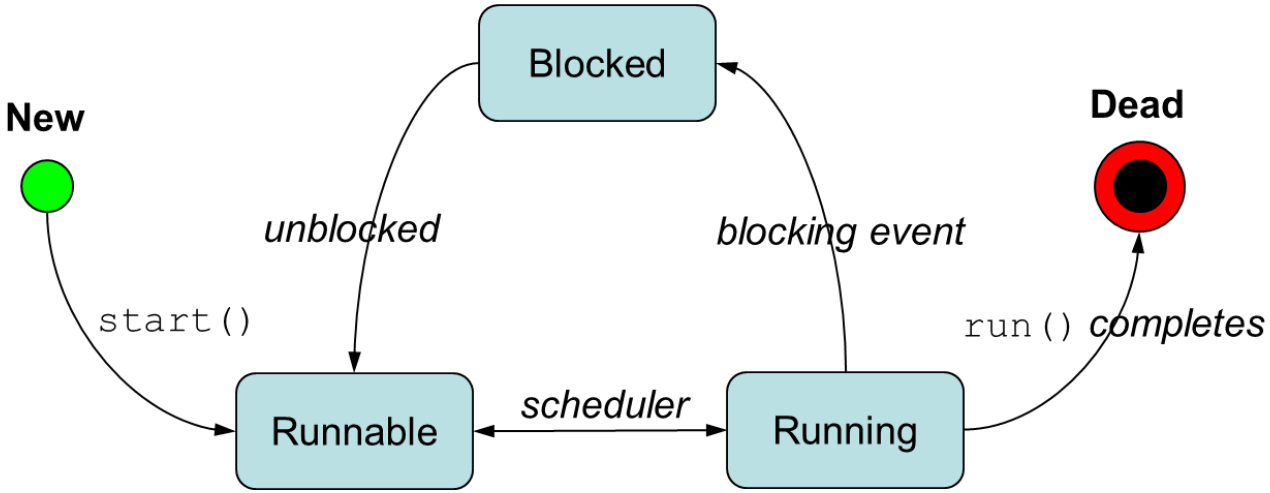

Based on Aj. Chate's slide


«interface»
Runnable
«datatype»
Thread
«datatype»
YourThread
«datatype»
YourAnotherThread

# Creating & Starting the Thread

› Using the **`start`** method to run thread

```
Thread t = new Thread();
t.start();
```




Based on Aj. Chate's slide

# Subclass of Thread

```
public class SomeThread extends Thread{
  public void run() {
    // code for thread execution
  }
}
```

```
public class ThreadTester {
  public static void main(String[] args) {
    // creating a thread
    SomeThread t = new SomeThread();

    // start the thread
    t.start();
  }
}
```

Based on Aj. Chate's slide

# Implementing Runnable

```
public class RunningClass [extends XXX] implements Runnable {
  public void run() { // must be overridden
    // code for thread execution
  }
}
```

```
public class ThreadTester {
  public static void main(String[] args) {
    // creating an instance of a Runnable
    RunningClass rc = new RunningClass();

    // creating a new thread for the Runnable instance
    Thread t = new Thread(rc);

    // starting the thread
    t.start();
  }
}
```

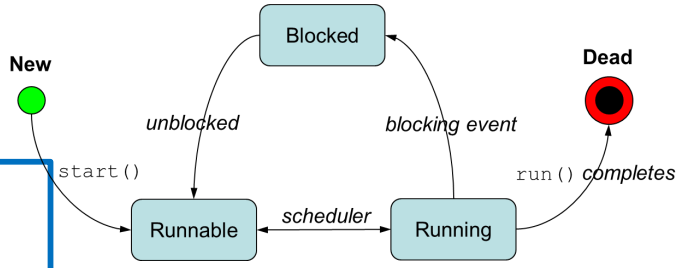


Based on Aj. Chate's slide

# Implementing Runnable as Anonymous (Preferred)

```
public class ThreadTester {
  public static void main(String[] args) {
    // creating a new thread for the Runnable instance
    Thread t = new Thread(new Runnable()
                    public void run(){ /* fill code */ }
                });

    // starting the thread
    t.start();
  }
}
```

# Subclass of Thread (Demo)




ThreadTest.java

```
class MyThread extends Thread{
    public MyThread(String n) {
        super(n);
    }

    public void run(){
        for(int i=0; i<100; ++i)
            System.out.print(getName());
    }
}

public class ThreadTest {
    public static void main(String[] args){
        new MyThread("A").start();
        new MyThread("B").start();
    }
}
```

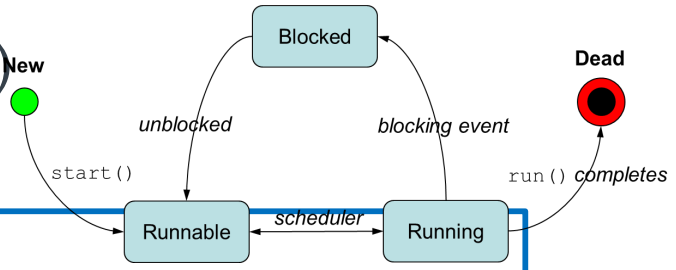
Result1

```
AAAAAAAAAAAAAAAAAAAABBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBAB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBAAAABBBBBBA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA
```

Result2

```
ABBBBBBBBBBBBBBBBBBBBBBBAAAAAA
AAAAAAAAAAAABBBBBBBBBBBBBBBBB
BBBBBBBAAAAAAABBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAABAAAAAAAAAAAA
AAAAAAAAAA
```

# Implementing Runnable (Demo)




RunnableTest.java

```
class MyRunnable implements Runnable{
   public void run(){
      for(int i=0; i<100; ++i)

   System.out.print(Thread.currentThread().getName());
   }
}

public class RunnableTest {
   public static void main(String[] args) {
      Runnable r = new MyRunnable();
      new Thread(r, "A").start();
      new Thread(r, "B").start();
   }
}
```

Result1

```
ABBBBBBBBBBBBBBBAAAAAAAAABBBBB
BBBBBBBBBBBBBBBBBBBAAAAAAAAAB
BBABBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBABBBBBBBBBBBBBBBBAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA
```

# Basic Control of Threads

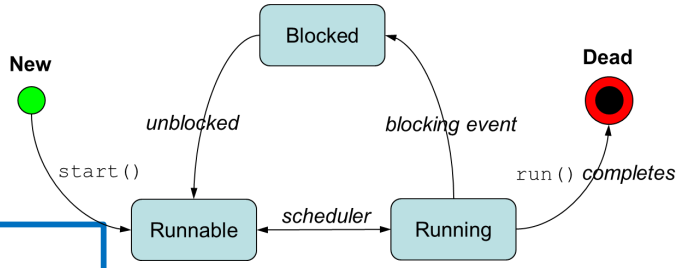
- Testing threads:
  - `isAlive()`
- Accessing thread priority:
  - `getPriority()`
  - `setPriority()`
- Putting threads on hold:
  - `Thread.sleep()`
  - `join()`
  - `Thread.yield()`

Based on Aj. Chate's slide

# Thread Priority

- There are 3 default thread priorities:
  - public static final int MIN_PRIORITY; (1)
  - public static final int NORM_PRIORITY; (5)
  - public static final int MAX_PRIORITY; (10)

# Thread Priority (Demo)




ThreadPriority.java

```
class MyThread extends Thread{
    public MyThread(String n) {
        super(n);
    }
    public void run(){
        for(int i=0; i<100; ++i)
            System.out.print(getName());
    }
}
public class ThreadPriority {
    public static void main(String[] args) {
        Thread a = new MyThread("A");
        Thread b = new MyThread("B");
        a.setPriority(Thread.MAX_PRIORITY);
        a.start();
        b.start();
    }
}
```

Result1

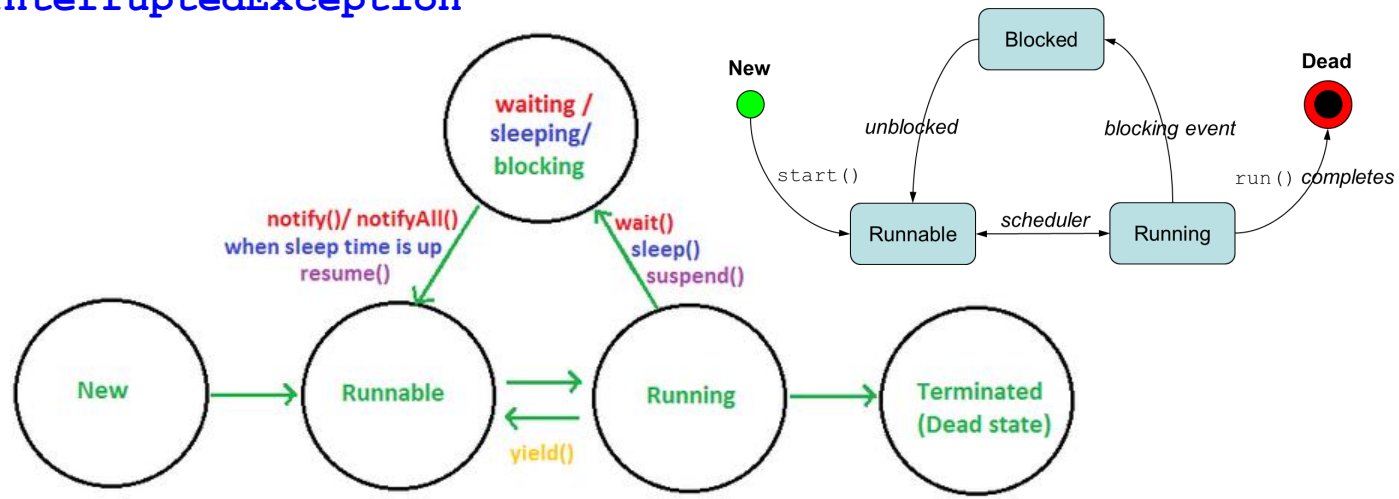
```
ABBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBAAAAAAAAAA
AAAAAAAABAAAAAAABBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA
```

Result2

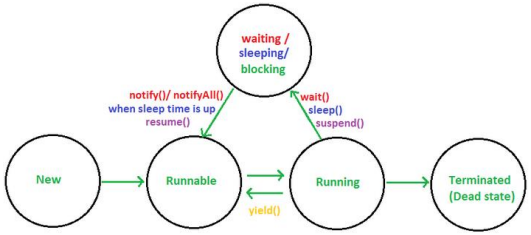
```
AAAAAAAAAAAAAAAAAAABAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAABBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBAAAAAAAAAA
ABBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBB
AAAAAAAAAA
```

# Thread.sleep()

- Allow other threads a chance to execute
- Change from the Running state to the TIMED_WAITING state
  - TIMED_WAITING: A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state
- *sleep* is a *static* method in the **`Thread`** class
- throws **`InterruptedException`**



# Thread.sleep() (Demo)




SleepState.java

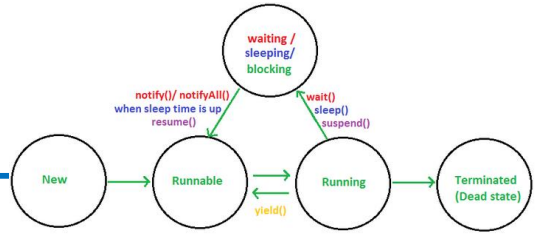
```
class SleepThread extends Thread{
    public void run(){
        try{
            Thread.sleep(1);
        }
        catch(InterruptedException e){}
    }
}
public class SleepState {
    public static void main(String[] args) {
        SleepThread t = new SleepThread();
        System.out.println(t.getState());
        t.start();
        Thread.State s;
        do{
            s = t.getState();
            System.out.println(s);
        }while(s != Thread.State.TERMINATED);
    }
}
```

Result

```
NEW
RUNNABLE
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
TIMED_WAITING
RUNNABLE
TERMINATED
```

Let's change to sleep(1000)

# Interrupt (Demo)




InterruptSleep.java

```
class Sleep10Thread extends Thread{
    public void run(){
        try{
            System.out.println("Sleep10Thread sleep");
            Thread.sleep(10000);
            System.out.println("Sleep10Thread wake up");
        }
        catch(InterruptedException e){
            System.out.println("Sleep10Thread interrupted");
        }
    }
}
```

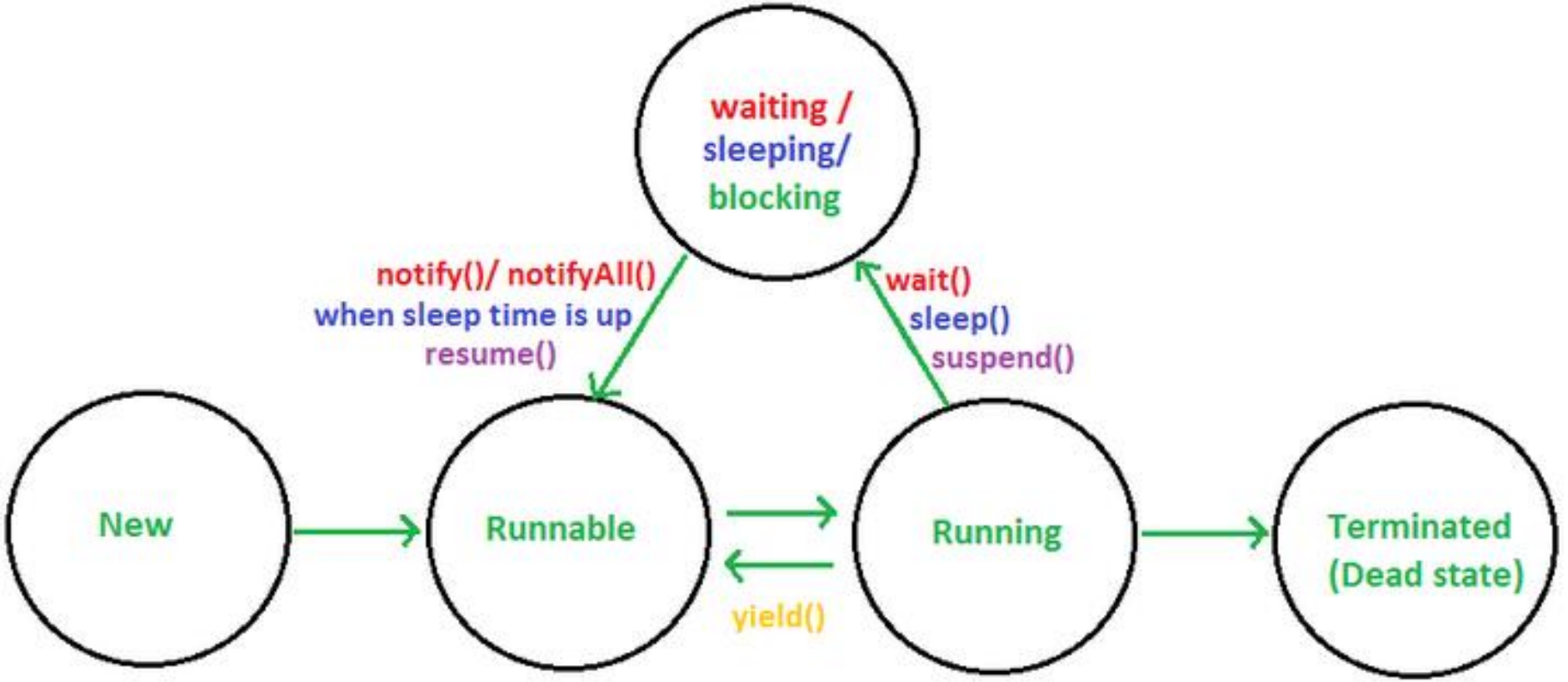
Result

```
Main sleep

Sleep10Thread sleep

Main wake up

Sleep10Thread interrupted
```

```
public class InterruptSleep {
    public static void main(String[] args) {
        Sleep10Thread t = new Sleep10Thread();
        t.start();
        try{
            System.out.println("Main sleep");
            Thread.sleep(3000);
            System.out.println("Main wake up");
        }
        catch(InterruptedException e){}
        t.interrupt();
    }
}
```



# Thread.yield()



# Thread.yield() (Demo1)

YieldExample.java

```
public class YieldExample {

     public static void main(String[] args) {

          Thread producer = new Producer2();
          Thread consumer = new Consumer2();

          producer.setPriority(Thread.MIN_PRIORITY);
          consumer.setPriority(Thread.MAX_PRIORITY);

          producer.start();
          consumer.start();

     }
```

Result

```
I am Producer : Produced Item 0
I am Producer : Produced Item 1
I am Consumer : Consumed Item 0
I am Consumer : Consumed Item 1
I am Consumer : Consumed Item 2
I am Consumer : Consumed Item 3
I am Producer : Produced Item 2
I am Consumer : Consumed Item 4
I am Producer : Produced Item 3
I am Producer : Produced Item 4
```

```
class Producer2 extends Thread {

  public void run() {

    for (int i = 0; i < 5; i++) {

      System.out.println("I am Producer : Produced Item " + i);

      // Thread.yield();

    }

  }

}
```

```
class Consumer2 extends Thread {

  public void run() {

    for (int i = 0; i < 5; i++) {

      System.out.println("I am Consumer : Consumed Item " + i);

      // Thread.yield();

    }

  }

}
```

# Thread.yield() (Demo2)

Not always!!

YieldExample.java

```
public class YieldExample {

     public static void main(String[] args) {

          Thread producer = new Producer2();
          Thread consumer = new Consumer2();

          producer.setPriority(Thread.MIN_PRIORITY);
          consumer.setPriority(Thread.MAX_PRIORITY);

          producer.start();
          consumer.start();

     }
```

Result

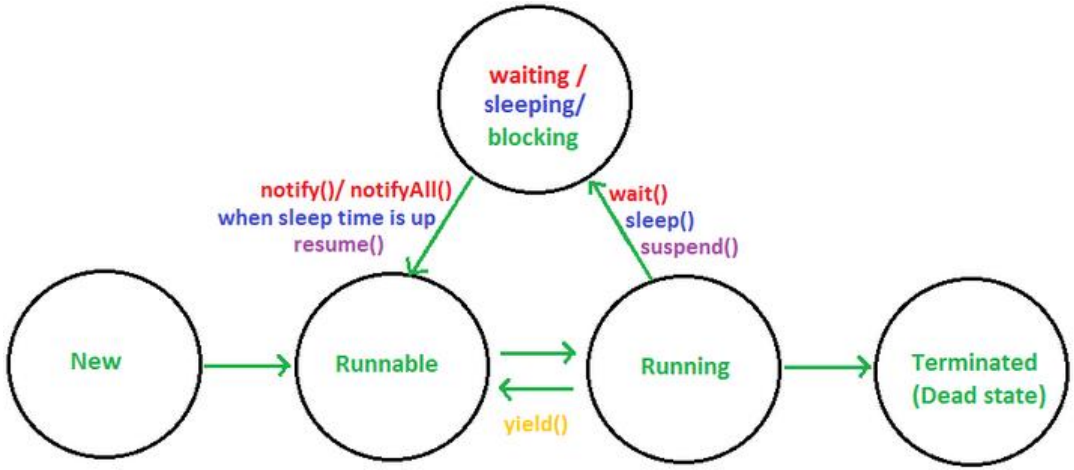
```
I am Producer : Produced Item 0
I am Consumer : Consumed Item 0
I am Producer : Produced Item 1
I am Consumer : Consumed Item 1
I am Producer : Produced Item 2
I am Consumer : Consumed Item 2
I am Producer : Produced Item 3
I am Consumer : Consumed Item 3
I am Producer : Produced Item 4
I am Consumer : Consumed Item 4
```

```
class Producer2 extends Thread {

  public void run() {

    for (int i = 0; i < 5; i++) {

      System.out.println("I am Producer : Produced Item " + i);

      Thread.yield();

    }

  }
}
```

```
class Consumer2 extends Thread {

  public void run() {

    for (int i = 0; i < 5; i++) {

      System.out.println("I am Consumer : Consumed Item " + i);

      Thread.yield();

    }

  }
}
```

# Thread.join()

- wait until the thread on which the **`join`** method is called terminates
- If join() is called on a Thread instance, the currently running thread will block until the Thread instance has finished executing.
- throws **`InterruptedException`**



# Thread.join() (Demo)

Joining.java

```
class Sleeper extends Thread {
  private int duration;

  public Sleeper(String name, int sleepTime) {
    super(name);
    duration = sleepTime;
    start();
  }

  public void run() {
    try {
      Thread.sleep(duration);
    } catch (InterruptedException e) {
      System.out.println(getName() + " was interrupted.“
        + "isInterrupted(): " + isInterrupted());
    }
    System.out.println(getName() + " has awakened");
  }
```

```
class Joiner extends Thread {
  private Sleeper sleeper;

  public Joiner(String name, Sleeper sleeper) {
    super(name);
    this.sleeper = sleeper;
    start();
  }

  public void run() {
    try {
      sleeper.join();
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }
    System.out.println(getName() + " join completed");
  }
}
```

```
public class Joining {
  public static void main(String[] args) {
    Sleeper sleepy = new Sleeper("Sleepy", 1500);
    Sleeper grumpy = new Sleeper("Grumpy", 1500);
    Joiner dopey = new Joiner("Dopey", sleepy);
    Joiner doc = new Joiner("Doc",grumpy);
    grumpy.interrupt();
  }

}
```

Result

```
Grumpy was interrupted.isInterrupted(): false

Grumpy has awakened

Doc join completed

Sleepy has awakened

Dopey join completed
```

# Thread Pools

- It is recommended that you do not explicitly create and use Thread objects to implement concurrency.

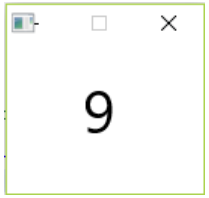
- In Java 1.5+, it provides a better way to manage a pool of threads
  - Class ThreadPoolExecutor
  - Interface Executor
- Using an Executor has many advantages over creating threads yourself.
  - It can reuse existing threads to eliminate the overhead of creating a new thread for each task;
  - It can optimize the number threads to ensure that the processor stays busy, without creating so many threads that the application runs out of resources.

Based on Aj. Chate's slide

# Part 2 : Thread in JavaFX

# TimerWithThread.java

```
public class TimerWithThread extends Application {
    private Canvas canvas;
    private int currentTime;
    private Thread timerThread;
    @Override
    public void start(Stage primaryStage) throws Exception {
        ...
        Initialize UI then display
        ...

        this.currentTime = 0;
        GraphicsContext gc = canvas.getGraphicsContext2D();
        this.timerThread = new Thread(() -> {
            while(true){
                try {
                    Thread.sleep(1000);
                    currentTime++;
                    drawCurrentTimeString(gc);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                    System.out.println("Stop Timer Thread");
                    break;
                }
            }
        });
        this.timerThread.start();
    }

    @Override
    public void stop() throws Exception {
        // TODO Auto-generated method stub
        this.timerThread.interrupt();
    }
}
```




This is an example of a Timer program, which displays total seconds passed since the program has started, utilizing a Thread

- Note that when a JavaFX Application is terminated, any running background threads won't stop alongside it
  - You have to make it stop by yourself

Interrupt the timer thread to make it stop running when a JavaFX application is terminated

## TimerWithAnimationTimer.java

```
public class TimerWithAnimationTimer extends Application {
     private Canvas canvas;
     private int currentTime;
     private Thread timerThread;
     @Override
     public void start(Stage primaryStage) throws Exception {
          ...
          Initialize UI then display
          ...

          this.currentTime = 0;
          this.lastTimeTriggered = -1;
          GraphicsContext gc = canvas.getGraphicsContext2D();
          this.animationTimer = new AnimationTimer() {
               @Override
               public void handle(long now) {
                    lastTimeTriggered = (lastTimeTriggered < 0 ?
                    now : lastTimeTriggered);
                    if (now - lastTimeTriggered >= 1000000000){
                         currentTime++;
                         drawCurrentTimeString(gc);
                         lastTimeTriggered = now;
                    }
               }
          };
          this.animationTimer.start();
     }
}
```

› You might notice that a Timer program can also be implemented by utilizing an AnimationTimer

What is difference between Thread and AnimationTimer?

## TimerWithAnimationTimer.java

```
public class TimerWithAnimationTimer extends Application {
     private Canvas canvas;
     private int currentTime;
     private Thread timerThread;
     @Override
     public void start(Stage primaryStage) throws Exception {
          ...
          Initialize UI then display
          ...

          this.currentTime = 0;
          this.lastTimeTriggered = -1;
          GraphicsContext gc = canvas.getGraphicsContext2D();
          this.animationTimer = new AnimationTimer() {
               @Override
               public void handle(long now) {
                    lastTimeTriggered = (lastTimeTriggered < 0 ?
                    now : lastTimeTriggered);
                    if (now - lastTimeTriggered >= 1000000000){
                         currentTime++;
                         drawCurrentTimeString(gc);
                         lastTimeTriggered = now;
                    }
               }
          };
          this.animationTimer.start();
     }
}
```

› You might notice that a Timer program can also be implemented by utilizing an AnimationTimer

First, you should know what is JavaFX Application Thread.

# JavaFX Application Thread

- JavaFX scene graph, which represents the graphical user interface of a JavaFX application, is not thread-safe
  - It can only be accessed and modified from the UI thread also known as the JavaFX Application Thread
- JavaFX Application Thread starts when start() method on an Application instance finishes its execution
- Every JavaFX event, such as ActionEvent, MouseEvent or KeyEvent, is also handled by JavaFX Application Thread
  - If you implement a long-running task on the JavaFX Application Thread, it will make your application unresponsive until the task are finished

You should use Thread to handle these long-running tasks

# JavaFX Application Thread

- JavaFX scene graph, which represents the graphical user interface of a JavaFX
- JavaFX Application Thread starts when start() method on an Application instan
- Every vent or KeyEvent, is also hand
  - If y ation Thread, it will make your application unresponsive until the task are finished

JavaFX scene graph, which represents the graphical user interface of a JavaFX application, is not thread-safe

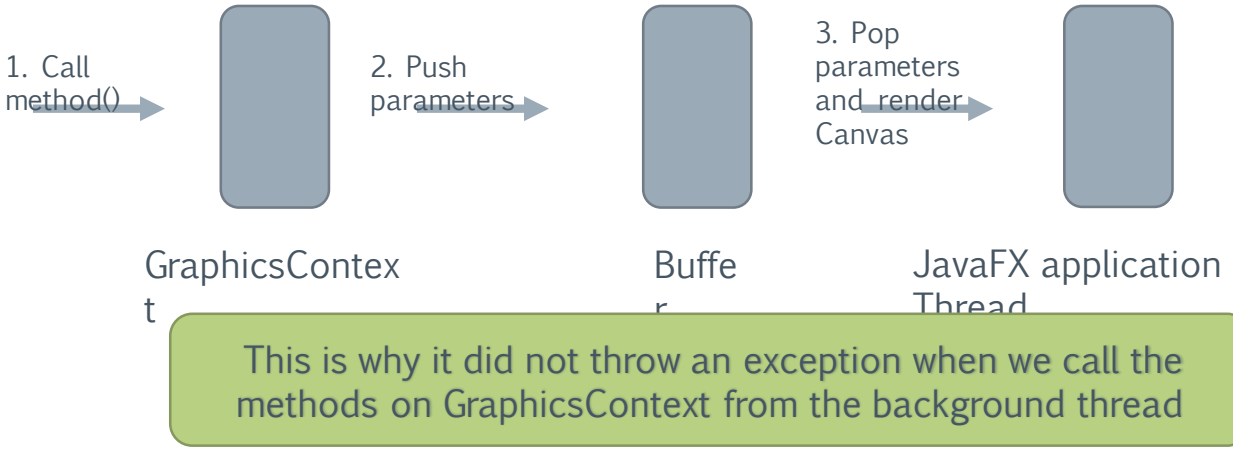
Isn't there something incorrect about this statement and the Timer program?

Then why the Timer Program didn't throw any exception when we call the methods on GraphicsContext from the background thread

You should use long-running tasks

# How GraphicsContext work with JavaFX Application Thread

- When we call the methods on GraphicsContext, the rendering process of the Canvas node will not be executed immediately
- Each call will push the necessary parameters onto the buffer until JavaFX Application Thread is ready
- Then pop these parameters from the buffer and execute the rendering process of the Canvas node on JavaFX Application Thread



# AnimationTimer vs Thread

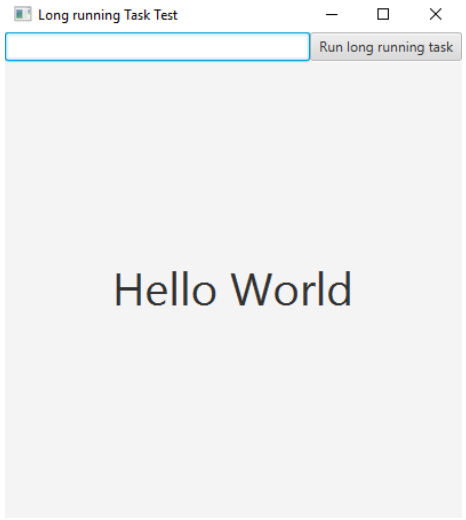
- AnimationTimer is a background thread which run a task, implemented inside handle() method, in each frame
  - The task is executed on the JavaFX Application Thread so you can directly access and modify JavaFX scene graph safely
  - It should only be used when you need some tasks to run in each frame and those tasks are not long-running task
- Thread is a base level of AnimationTimer so you can implement to do something more complex than AnimationTimer
  - Unlike AnimationTimer, you cannot directly access and modify JavaFX scene graph safely
  - However you can still do it by communicating with the JavaFX Application Thread

In most cases, you should use Thread rather than AnimationTimer

## TaskOnJavaFXThread.java

```
public class TaskOnJavaFXThread extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        ...
        Initialize UI then display
        ...

        button.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                try {
                    Thread.sleep(5000);
                displayLabel.setText(textField.getText());
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

    }
}
```

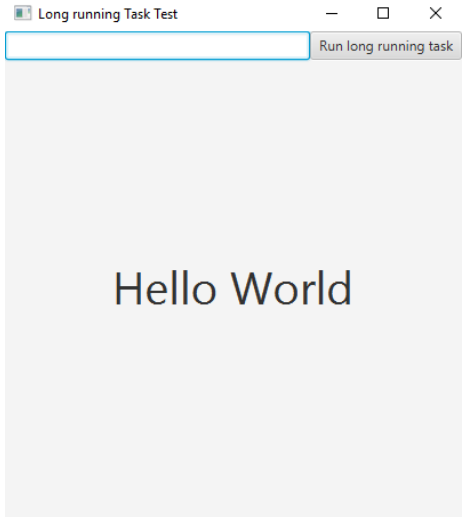
After the button is clicked, user is not able to interact with the application for ~5 seconds




An example program

## TaskOnBackgroundThreadWithException.java

```
public class TaskOnBackgroundThreadWithException extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        ...
        Initialize UI then display
        ...

        button.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                Thread thread = new Thread(() -> {
                    try {
                        Thread.sleep(5000);
                        displayLabel.setText(textField.getText());
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                });
                thread.start();
            }
        });

    }
}
```



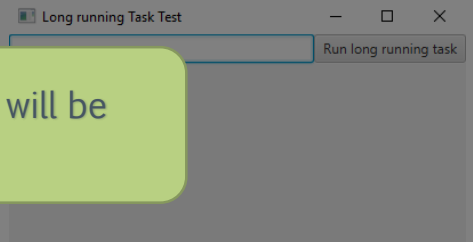

An example program

User is able to interact with the application even after the button is clicked But ....



TaskOnBackgroundThreadWithException.java

```
public class TaskOnBackgroundThreadWithException extends Application {
    @Override
    public void start(Stage pr
        ...
        Initialize UI then di
        ...

        button.setOnAction(new EventHandler<ActionEvent>() {
    }
}
```




After ~5 seconds, this exception will be thrown

```
Exception in thread "Thread-4" java.lang.IllegalStateException: Not on FX application thread; currentThread = Thread-4
        at com.sun.javafx.tk.Toolkit.checkFxUserThread(Toolkit.java:236)
        at com.sun.javafx.tk.quantum.QuantumToolkit.checkFxUserThread(QuantumToolkit.java:423)
        at javafx.scene.Parent$2.onProposedChange(Parent.java:367)
        at com.sun.javafx.collections.VetoableListDecorator.setAll(VetoableListDecorator.java:113)
        at com.sun.javafx.collections.VetoableListDecorator.setAll(VetoableListDecorator.java:108)
        at com.sun.javafx.scene.control.skin.LabeledSkinBase.updateChildren(LabeledSkinBase.java:575)
        at com.sun.javafx.scene.control.skin.LabeledSkinBase.handleControlPropertyChanged(LabeledSkinBase.java:204)
        at com.sun.javafx.scene.control.skin.LabelSkin.handleControlPropertyChanged(LabelSkin.java:49)
        at com.sun.javafx.scene.control.skin.BehaviorSkin
        at com.sun.javafx.scene.control.MultiplePropertyC
        at javafx.beans.value.WeakChangeListener.changed(
        at com.sun.javafx.binding.ExpressionHelper$Single
        at com.sun.javafx.binding.ExpressionHelper.fireValueChangedEvent(ExpressionHelper.java:81)
        at javafx.beans.property.StringPropertyBase.fireValueChangedEvent(StringPropertyBase.java:103)
        at javafx.beans.property.StringPropertyBase.markInvalid(StringPropertyBase.java:110)
        at javafx.beans.property.StringPropertyBase.set(StringPropertyBase.java:144)
        at javafx.beans.property.StringPropertyBase.set(StringPropertyBase.java:49)
        at javafx.beans.property.StringProperty.setValue(StringProperty.java:65)
        at javafx.scene.control.Labeled.setText(Labeled.java:145)
        at LongRunningTaskOnBackgroundThread$1.lambda$0(LongRunningTaskOnBackgroundThread.java:50)
        at java.lang.Thread.run(Thread.java:745)
```

Why ?

User is able to interact with the application even after the button is clicked But ....

## TaskOnBackgroundThreadWithException.java

```
public class TaskOnBackgroundThreadWithException extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        ...
        Initialize UI then display
        ...

        button.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                Thread thread = new Thread(() -> {
                    try {
                        Thread.sleep(5000);
                        displayLabel.setText(textField.getText());
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                });
                thread.start();
            }
        });

    }
}
```

- An exception is thrown because there is an attempt to access and modify JavaFX scene graph from our thread

To make this possible, you have to communicate with JavaFX Application Thread

# Communication between JavaFX Application Thread and Background Thread

- The communication between UI Thread and Background Thread is a common thing, you have to deal with, when you develop a GUI application which connect to the internet nowadays
  - In this case, the UI Thread is JavaFX Application Thread
- Generally, there are 2 ways to communicate between JavaFX Application Thread and Background Thread
  - Approach 1 : use Platform.runLater() (Simple) (Teach in this course)
  - Approach 2 : use javafx.concurrent.Task and javafx.concurrent.Service (Best)
    - http://docs.oracle.com/javafx/2/threads/jfxpub-threads.htm

# Approach 1 : use Platform.runLater()

- Platform.runLater() simply receive an instance of Runnable Object as a parameter then execute it later on the JavaFX Application Thread

```
Platform.runLater(new Runnable(){
    @Override
    public void run() {
        ..................................
        ..................................
        Access and Modify JavaFX Scene Graph
        ..................................
        ..................................
    }
});
```

Inside a run() method, should contain only the codes with access and modify JavaFX Scene Graph

## TaskOnBackgroundThreadWithRunLater.java

```
public class TaskOnBackgroundThreadWithRunLater extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        ...
        Initialize UI then display
        ...

        button.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                Thread thread = new Thread(() -> {
                    try {
                        Thread.sleep(5000);
                        Platform.runLater(new Runnable(){
                            @Override
                            public void run() {
                                displayLabel.setText(textField.getText());
                            }
                        });
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                });
                thread.start();
            }
        });

    }
}
```

- separate the part of codes with access and modify JavaFX Scene Graph then put it into Platform.runlater()

Now, the program can run successfully without an exception