- **Multithreading in Java**

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms:
1. Extending the Thread class
2. Implementing the Runnable Interface

Thread creation by extending the Thread class

We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

```
// Java code for thread creation by extending
// the Thread class
class MultithreadingDemo extends Thread
{
    public void run()
    {
        try
        {
            // Displaying the thread that is running
            System.out.println ("Thread " +
                  Thread.currentThread().getId() +
                  " is running");

        }
        catch (Exception e)
        {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}

// Main Class
public class Multithread
{
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i=0; i<8; i++)
        {
            MultithreadingDemo object = new MultithreadingDemo();
            object.start();
```

```
        }
    }
}
```

Output :

```
Thread 8 is running
Thread 9 is running
Thread 10 is running
Thread 11 is running
Thread 12 is running
Thread 13 is running
Thread 14 is running
Thread 15 is running
```

Thread creation by implementing the Runnable Interface

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

```
// Java code for thread creation by implementing
// the Runnable Interface
class MultithreadingDemo implements Runnable
{
    public void run()
    {
        try
        {
            // Displaying the thread that is running
            System.out.println ("Thread " +
                                Thread.currentThread().getId() +
                                " is running");

        }
        catch (Exception e)
        {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}

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

```
        int n = 8; // Number of threads
        for (int i=0; i<8; i++)
        {
            Thread object = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

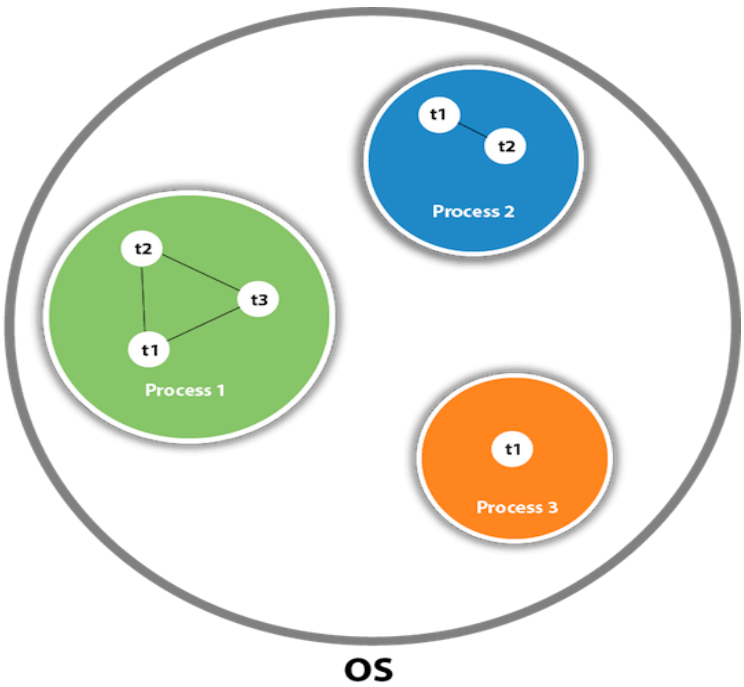
Output :

```
Thread 8 is running
Thread 9 is running
Thread 10 is running
Thread 11 is running
Thread 12 is running
Thread 13 is running
Thread 14 is running
Thread 15 is running
```

What is Thread in java?

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

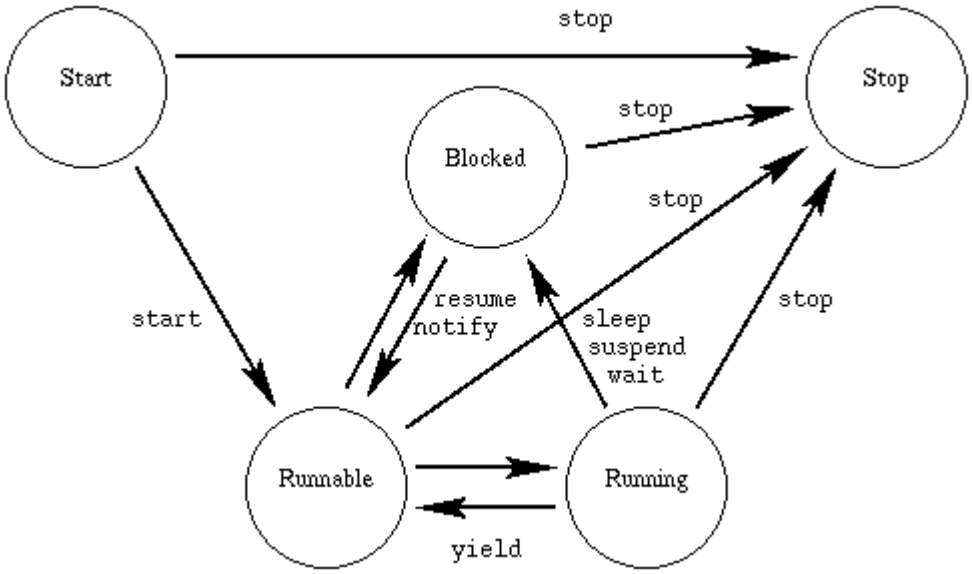


As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

Q2. • **Life Cycle of a Thread:**

Thread States

The thread scheduler's job is to move threads in and out of the therunning state. While the thread scheduler can move a thread from the running state back to runnable, other factors can cause a thread to move out of running, but not back to runnable. One of these is when the thread's run()method completes, in which case the thread moves from the running state directly to the dead state.



New/Start:

This is the state the thread is in after the Thread instance has been created, but the start() method has not been invoked on the thread. It is a live Thread object, but not yet a thread of execution. At this point, the thread is considered not alive.

Runnable:

This means that a thread can be run when the time-slicing mechanism has CPU cycles available for the thread. Thus, the thread might or might not be running at any moment, but there's nothing to prevent it from being run if the scheduler can arrange it. That is, it's not dead or blocked.

Running:

This state is important state where the action is. This is the state a thread is in when the thread scheduler selects it (from the runnable pool) to be the currently executing process. A thread can transition out of a running state for several reasons, including because "the thread scheduler felt like it". There are several ways to get to the runnable state, but only one way to get to the running state: the scheduler chooses a thread from the runnable pool of thread.

Blocked:

The thread can be run, but something prevents it. While a thread is in the blocked state, the scheduler will simply skip it and not give it any CPU time. Until a thread reenters the runnable state, it won't perform any operations. Blocked state has some sub-states as below,

- **Blocked on I/O:** The thread waits for completion of blocking operation. A thread can enter this state because of waiting I/O resource. In that case, the thread sends back to runnable state after the availability of resources.
- **Blocked for join completion:** The thread can come in this state because of waiting for the completion of another thread.
- **Blocked for lock acquisition:** The thread can come in this state because of waiting for acquire the lock of an object.

Dead:

A thread in the dead or terminated state is no longer schedulable and will not receive any CPU time. Its task is completed, and it is no longer runnable. One way for a task to die is by returning from its run( ) method, but a task's thread can also be interrupted, as you'll see shortly.

We have various methods which can be called on Thread class object. These methods are very useful when writing a multithreaded application. Thread class has following important methods. We will understand various thread states as well later in this tutorial.

- **Thread Methods**

Method Signature	Description
String getName()	Retrieves the name of running thread in the current context in String format
void start()	This method will start a new thread of execution by calling run() method of Thread/runnable object.
void run()	This method is the entry point of the thread. Execution of thread starts from this method.
void sleep(int sleeptime)	This method suspend the thread for mentioned time duration in argument (sleeptime in ms)
void yield()	By invoking this method the current thread pause its execution temporarily and allow other threads to execute.
void join()	This method used to queue up a thread in execution. Once called on thread, current thread will wait till calling thread completes its execution
boolean isAlive()	This method will check if thread is alive or dead

- **Thread Scheduling**

As mentioned briefly in the previous section, most computer configurations have a single CPU. Hence, threads run one at a time in such a way as to provide an illusion of concurrency. Execution of multiple threads on a single CPU in some order is called *scheduling*. The Java runtime environment supports a very simple, deterministic scheduling algorithm called *fixed-priority scheduling*. This algorithm schedules threads on the basis of their priority relative to other Runnable threads.

When a thread is created, it inherits its priority from the thread that created it. You also can modify a thread's priority at any time after its creation by using the setPriority method. Thread priorities are integers ranging

between MIN_PRIORITY and MAX_PRIORITY (constants defined in the Thread class). The higher the integer, the higher the priority. At any given time, when multiple threads are ready to be executed, the runtime system chooses for execution the Runnable thread that has the highest priority. Only when that thread stops, yields, or becomes Not Runnable will a lower-priority thread start executing. If two threads of the same priority are waiting for the CPU, the scheduler arbitrarily chooses one of them to run. The chosen thread runs until one of the following conditions is true:

- A higher priority thread becomes runnable.
- It yields, or its run method exits.
- On systems that support time-slicing, its time allotment has expired.

Then the second thread is given a chance to run, and so on, until the interpreter exits.

The Java runtime system's thread scheduling algorithm is also preemptive. If at any time a thread with a higher priority than all other Runnable threads becomes Runnable, the runtime system chooses the new higher-priority thread for execution. The new thread is said to *preempt* the other threads.

A Thread Race

RaceApplet is an applet that animates a race between two "runner" threads of different priorities. Clicking the mouse on the applet starts the two runners. Runner 2 has a priority of 2; runner 3 has a priority of 3.

The runners are implemented by a Thread subclass called Runner. Here is the run method for the Runner class, which simply counts from 1 to 10,000,000:

```
public int tick = 1;
public void run() {
   while (tick < 10000000) {
      tick++;
   }
}
```

Selfish Threads

The Runner class used in the previous races implements impaired thread behavior. Recall the run method from the Runner class used in the races:

```
public int tick = 1;
public void run() {
   while (tick < 10000000) {
      tick++;
   }
}
```

The while loop in the run method is in a tight loop. Once the scheduler chooses a thread with this thread body for execution, the thread never voluntarily relinquishes control of the CPU; it just continues to run until the while loop terminates naturally or until the thread is preempted by a higher-priority thread. This thread is called a *selfish thread*.

Time-Slicing

Some systems limit selfish-thread behavior with a strategy known as time slicing. Time slicing comes into play when multiple Runnable threads of equal priority are the highest-priority threads competing for the CPU. For example, a standalone program, stand-alone Java program based on RaceApplet creates two equal priority selfish threads that have this run method:

```
public void run() {
   while (tick < 400000) {
      tick++;
      if ((tick % 50000) == 0) {
         System.out.println("Thread #" + num + ", tick = " +
                                              tick);
      }
   }
}
```

This run method contains a tight loop that increments the integer tick. Every 50,000 ticks prints out the thread's identifier and its tick count.

- **Thread Priorities in Multithreading:**

In a Multi threading environment, thread scheduler assigns processor to a thread based on priority of thread. Whenever we create a thread in Java, it always has some priority assigned to it. Priority can either be given by JVM while creating the thread or it can be given by programmer explicitly.
Accepted value of priority for a thread is in range of 1 to 10. There are 3 static variables defined in Thread class for priority.

public static int MIN_PRIORITY: This is minimum priority that a thread can have. Value for this is 1.
public static int NORM_PRIORITY: This is default priority of a thread if do not explicitly define it. Value for this is 5.
public static int MAX_PRIORITY: This is maximum priority of a thread. Value for this is 10.
Get and Set Thread Priority:

1. **public final int getPriority():** java.lang.Thread.getPriority() method returns priority of given thread.
2. **public final void setPriority(int newPriority):** java.lang.Thread.setPriority() method changes the priority of thread to the value newPriority. This method throws IllegalArgumentException if value of parameter newPriority goes beyond minimum(1) and maximum(10) limit.

Examples of getPriority() and set

```
// Java program to demonstrate getPriority() and setPriority()
import java.lang.*;

class ThreadDemo extends Thread
{
```

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

    public static void main(String[]args)
    {
        ThreadDemo t1 = new ThreadDemo();
        ThreadDemo t2 = new ThreadDemo();
        ThreadDemo t3 = new ThreadDemo();

        System.out.println("t1 thread priority : " +
                           t1.getPriority()); // Default 5
        System.out.println("t2 thread priority : " +
                           t2.getPriority()); // Default 5
        System.out.println("t3 thread priority : " +
                           t3.getPriority()); // Default 5

        t1.setPriority(2);
        t2.setPriority(5);
        t3.setPriority(8);

        // t3.setPriority(21); will throw IllegalArgumentException
        System.out.println("t1 thread priority : " +
                           t1.getPriority());  //2
        System.out.println("t2 thread priority : " +
                           t2.getPriority()); //5
        System.out.println("t3 thread priority : " +
                           t3.getPriority());//8

        // Main thread
        System.out.print(Thread.currentThread().getName());
        System.out.println("Main thread priority : "
                    + Thread.currentThread().getPriority());

        // Main thread priority is set to 10
        Thread.currentThread().setPriority(10);
        System.out.println("Main thread priority : "
                    + Thread.currentThread().getPriority());
    }
}
```

Output:

```
t1 thread priority : 5
t2 thread priority : 5
t3 thread priority : 5
t1 thread priority : 2
t2 thread priority : 5
```

```
t3 thread priority : 8
Main thread priority : 5
```

Main thread priority : 10

Note:

- Thread with highest priority will get execution chance prior to other threads. Suppose there are 3 threads t1, t2 and t3 with priorities 4, 6 and 1. So, thread t2 will execute first based on maximum priority 6 after that t1 will execute and then t3.
- Default priority for main thread is always 5, it can be changed later. Default priority for all other threads depends on the priority of parent thread.
  Example:

```
// Java program to demonstrat ethat a child thread
// gets same priority as parent
import java.lang.*;

class ThreadDemo extends Thread
{
   public void run()
   {
      System.out.println("Inside run method");
   }

   public static void main(String[]args)
   {
      // main thread priority is 6 now
      Thread.currentThread().setPriority(6);

      System.out.println("main thread priority : " +
            Thread.currentThread().getPriority());

      ThreadDemo t1 = new ThreadDemo();

      // t1 thread is child of main thread
      // so t1 thread will also have priority 6.
      System.out.println("t1 thread priority : "
                        + t1.getPriority());
   }
}
```

```
Output:
Main thread priority : 6
```

t1 thread priority : 6

- If two threads have same priority then we can't expect which thread will execute first. It depends on thread scheduler's algorithm(Round-Robin, First Come First Serve, etc)
- If we are using thread priority for thread scheduling then we should always keep in mind that underlying platform should provide support for scheduling based on thread priority.

- **Creating and Executing threads:**
  [Refer page no. 1,2 for creating threads.]
  by
  1) Extending thread class and
  2) Implementing runnable interface.

What is Single Thread?

A single thread is basically a lightweight and the smallest unit of processing. Java uses threads by using a "Thread Class".

There are two types of thread – **user thread and daemon thread** (daemon threads are used when we want to clean the application and are used in the background).

When an application first begins, user thread is created. Post that, we can create many user threads and daemon threads.

Single Thread Example:

```
package demotest;

public class GuruThread
{
    public static void main(String[] args) {
        System.out.println("Single Thread");
    }
}
```

Advantages of single thread:

- Reduces overhead in the application as single thread execute in the system
- Also, it reduces the maintenance cost of the application.

Q1. **What is Multithreading?**

Multithreading in java is a process of executing two or more threads simultaneously to maximum utilization of CPU.

Multithreaded applications are where two or more threads run concurrently; hence it is also known as **Concurrency** in Java. This multitasking is done, when multiple processes share common resources like CPU, memory, etc.

Each thread runs parallel to each other. Threads don't allocate separate memory area; hence it saves memory. Also, context switching between threads takes less time.

Example of Multi thread:

```
package demotest;

public class GuruThread1 implements Runnable
{
     public static void main(String[] args) {
      Thread guruThread1 = new Thread("Guru1");
      Thread guruThread2 = new Thread("Guru2");
      guruThread1.start();
      guruThread2.start();
      System.out.println("Thread names are following:");
      System.out.println(guruThread1.getName());
      System.out.println(guruThread2.getName());
   }
   @Override
   public void run() {
   }

}
```

Advantages of multithread:

- The users are not blocked because threads are independent, and we can perform multiple operations at times
- As such the threads are independent, the other threads won't get affected if one thread meets an exception.

- **Thread Synchronization**:
- When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.
- So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.
- Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block. Following is the general form of the synchronized statement −
- Syntax

```
synchronized(objectidentifier) {
  // Access shared variables and other shared resources
}
```

- Here, the **objectidentifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents. Now we are going to see two examples, where we will print a counter using two different threads. When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

- Multithreading Example without Synchronization

- Here is a simple example which may or may not print counter value in sequence and every time we run it, it produces a different result based on CPU availability to a thread.

Example

```
class PrintDemo {
  public void printCount() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Counter   ---   "  + i );
      }
    } catch (Exception e) {
      System.out.println("Thread  interrupted.");
    }
  }
}

class ThreadDemo extends Thread {
  private Thread t;
  private String threadName;
  PrintDemo  PD;

  ThreadDemo( String name,  PrintDemo pd) {
    threadName = name;
    PD = pd;
  }

  public void run() {
    PD.printCount();
    System.out.println("Thread " +  threadName + " exiting.");
  }

  public void start () {
    System.out.println("Starting " +  threadName );
    if (t == null) {
      t = new Thread (this, threadName);
      t.start ();
    }
```

```
   }
}

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

    PrintDemo PD = new PrintDemo();

    ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
    ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );

    T1.start();
    T2.start();

    // wait for threads to end
      try {
      T1.join();
      T2.join();
    } catch ( Exception e) {
      System.out.println("Interrupted");
    }
  }
}
```

This produces a different result every time you run this program −

Output

```
Starting Thread - 1
Starting Thread - 2
Counter   ---   5
Counter   ---   4
Counter   ---   3
Counter   ---   5
Counter   ---   2
Counter   ---   1
Counter   ---   4
Thread Thread - 1  exiting.
Counter   ---   3
Counter   ---   2
Counter   ---   1
Thread Thread - 2  exiting.
```

- Multithreading Example with Synchronization
- Here is the same example which prints counter value in sequence and every time we run it, it produces the same result.

Example

```
class PrintDemo {
  public void printCount() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Counter   ---   "  + i );
      }
    } catch (Exception e) {
      System.out.println("Thread  interrupted.");
    }
  }
}

class ThreadDemo extends Thread {
  private Thread t;
  private String threadName;
  PrintDemo  PD;

  ThreadDemo( String name,  PrintDemo pd) {
    threadName = name;
    PD = pd;
  }

  public void run() {
    synchronized(PD) {
      PD.printCount();
    }
    System.out.println("Thread " +  threadName + " exiting.");
  }

  public void start () {
    System.out.println("Starting " +  threadName );
    if (t == null) {
      t = new Thread (this, threadName);
      t.start ();
    }
  }
}

public class TestThread {

  public static void main(String args[]) {
    PrintDemo PD = new PrintDemo();

    ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
    ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );

    T1.start();
    T2.start();
```

```
        // wait for threads to end
        try {
          T1.join();
          T2.join();
        } catch ( Exception e) {
          System.out.println("Interrupted");
        }
      }
    }
```

- This produces the same result every time you run this program −

  Output
  Starting Thread - 1
  Starting Thread - 2
  Counter --- 5
  Counter --- 4
  Counter --- 3
  Counter --- 2
  Counter --- 1
  Thread Thread - 1 exiting.
  Counter --- 5
  Counter --- 4
  Counter --- 3
  Counter --- 2
  Counter --- 1
  Thread Thread - 2 exiting.

- **Runnable interface:**

java.lang.Runnable is an interface that is to be implemented by a class whose instances are intended to be executed by a thread. There are two ways to start a new Thread – Subclass Thread and implement Runnable. There is no need of subclassing Thread when a task can be done by overriding only run() method of Runnable.

Steps to create a new Thread using Runnable :

1. Create a Runnable implementer and implement run() method.

2. Instantiate Thread class and pass the implementer to the Thread, Thread has a constructor which accepts Runnable instance.

3. Invoke start() of Thread instance, start internally calls run() of the implementer.

Invoking start(), creates a new Thread which executes the code written in run().

Calling run() directly doesn't create and start a new Thread, it will run in the same thread. To start a new line of execution, call start() on the thread.

Example,

```
public class RunnableDemo {
```

```
    public static void main(String[] args)
    {
        System.out.println("Main thread is- "
                    + Thread.currentThread().getName());
        Thread t1 = new Thread(new RunnableDemo().new RunnableImpl());
        t1.start();
    }

    private class RunnableImpl implements Runnable {

        public void run()
        {
            System.out.println(Thread.currentThread().getName()
                        + ", executing run() method!");
        }
    }
}
```

Output:

```
Main thread is- main
Thread-0, executing run() method!
```

Output shows two active threads in the program – main thread and Thread-0, main method is executed by the Main thread but invoking start on RunnableImpl creates and starts a new thread – Thread-0.

What happens when Runnable encounters an exception ?
Runnable can't throw checked exception but RuntimeException can be thrown from run(). Uncaught exceptions are handled by exception handler of the thread, if JVM can't handle or catch exceptions, it prints the stack trace and terminates the flow.
Example,

```
import java.io.FileNotFoundException;
```

```
public class RunnableDemo {


   public static void main(String[] args)
   {
      System.out.println("Main thread is- " +
                        Thread.currentThread().getName());
      Thread t1 = new Thread(new RunnableDemo().new RunnableImpl());
      t1.start();
   }


   private class RunnableImpl implements Runnable {


      public void run()
      {
         System.out.println(Thread.currentThread().getName()
                           + ", executing run() method!");
         /**
          * Checked exception can't be thrown, Runnable must
          * handle checked exception itself.
          */
         try {
            throw new FileNotFoundException();
         }
         catch (FileNotFoundException e) {
            System.out.println("Must catch here!");
            e.printStackTrace();
         }
```

```
            int r = 1 / 0;
            /*
             * Below commented line is an example
             * of thrown RuntimeException.
             */
            // throw new NullPointerException();
        }
    }
}
```

Output:

```
java.io.FileNotFoundException
    at RunnableDemo$RunnableImpl.run(RunnableDemo.java:25)
    at java.lang.Thread.run(Thread.java:745)
Exception in thread "Thread-0" java.lang.ArithmeticException: / by zero
    at RunnableDemo$RunnableImpl.run(RunnableDemo.java:31)
    at java.lang.Thread.run(Thread.java:745)
```

Output shows that Runnable can't throw checked exceptions, FileNotFoundException in this case, to the callers, it must handle checked exceptions in the run() but RuntimeExceptions (thrown or auto-generated) are handled by the JVM automatically.