Thread Management

1. Defining and Starting a Thread

An application that creates an instance of `Thread` must provide the code that will run in that thread. There are two ways to do this:

- *Provide a `Runnable` object.* The **`Runnable`** interface defines a single method, `run`, meant to contain the code executed in the thread. The `Runnable` object is passed to the `Thread`constructor, as in the `HelloRunnable` example:

```
package hellorunnable;
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        HelloRunnable r1=new HelloRunnable();
        Thread t1=new  Thread(r1);
        t1.start();
    }
}
```

`// or in the main section:` **`(new Thread(new HelloRunnable())).start();`** `in this way, we can not control thread sleeping and stoping … etc`

- *Subclass `Thread`.* The `Thread` class itself implements `Runnable`, though its `run` method does nothing. An application can subclass `Thread`, providing its own implementation of `run`, as in the **`HelloThread`** example:

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        HelloThread t1=new HelloThread();
        t1.start();
    }
}
```

The code of the main method can be as following as well:

```
 (new HelloThread()).start();
// in this way, we do not have a thread full control to sleep or stop
Or :
 Thread t1=new Thread(new HelloRunnable());
 t1.start();
```

another example:

```
package threadexample;

public class ThreadExample extends Thread {
  public void run() {
    for (int i = 0; i < 5; i++) {
      printMyName();
    }
  }

  public void printMyName() {
      System.out.println("The Thread name is " + Thread.currentThread().getName());
  }

  public static void main(String[] args) {
    ThreadExample ttsn = new ThreadExample();
    ttsn.setName("Created One");
    ttsn.start();

    Thread t2 = currentThread();
    t2.setName("Main One");

    for (int i = 0; i < 5; i++) {
      ttsn.printMyName();
    }
  }
}
```

Notice that both examples invoke `Thread.start` in order to start the new thread.

Which of these idioms should you use? The first idiom, which employs a `Runnable` object, is more general, because the `Runnable` object can subclass a class other than `Thread`. The second idiom is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of `Thread`.

Why Thread Class ?

The `Thread` class defines a number of methods useful for thread management. These include `static` methods, which provide information about, or affect the status of, the thread invoking the method.

Thread Management

1- Sleep

`Thread.sleep` causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system.

Notice that `Thread` declares that it `throws InterruptedException (in case of main thread) or try-catch (anywhere else)`. This is an exception that `sleep` throws when another thread interrupts the current thread while `sleep` is active.

```
package hellorunnable;
import java.util.logging.Level;
import java.util.logging.Logger;
public class HelloRunnable implements Runnable {
    public void run() {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };
        for (int i = 0;i < importantInfo.length;i++) {
            try {
                //Pause for 4 seconds
                Thread.sleep(4000);
            } catch (InterruptedException e) {}
            //Print a message
            System.out.println(importantInfo[i]);
        }
    }
    public static void main(String args[])
        throws InterruptedException {
```

```
        Thread t1=new Thread(new HelloRunnable());
        t1.start();
    }
}
```

Another example showing that even if the main thread finished execution, the child threads still running until they finished their task:

```
package hellorunnable;



class HelloRunnable implements Runnable {

  public static void main(String args[]) {
    Thread t = Thread.currentThread();
    Thread t1=new Thread(new HelloRunnable());
    t1.start();
    Thread t2=new Thread(new HelloRunnable());
    t2.start();
    Thread t3=new Thread(new HelloRunnable());
    t3.start();

    System.out.println("Current thread: " + t);

    t.setName("My Thread: Main");
    t1.setName("My Thread: Created t1");
    t2.setName("My Thread: Created t2");
    t3.setName("My Thread: Created t3");
    System.out.println("After name change of current thread: " + t);

    for (int n = 5; n > 0; n--)
        System.out.println(n);
    System.out.println("main thread finished execution");
  }
```

```
    public void run() {
     for (int i=0;i<15;i++)
       System.out.println("name of the created thread: " + Thread.currentThread());
    }
}
```

2- Stop

This instruction focuses the running thread to stop what is doing, there is no need for try-catch block.

```
package stopex;


class MyThread implements Runnable{

    private Boolean stop = false;

    public void run(){

        while(!stop){

          System.out.println("continue processing");
        }
    }
    public Boolean getStop() {
        return stop;
    }

    public void setStop(Boolean stop) {
        this.stop = stop;
    }

}

 public class StopEx {

    public static void main(String[] args){
        //new StopEx();
        MyThread myThread=new MyThread();
        Thread th = new Thread(myThread);
        th.start();
        for(int i=0;i<10;i++)
          try {
                Thread.sleep(1000);
             } catch (InterruptedException e) {}


        //This will compell the thread to stop
        myThread.setStop(true);
    }

}
```

3- Join and Alive instructions

Sometimes one thread needs to know when another thread is ending. In java, **isAlive()** and **join()** are two different methods to check whether a thread has finished its execution.

The **isAlive()** methods return **true** if the thread upon which it is called is still running otherwise it return **false**.

```
final boolean isAlive()
```

But, **join()** method is used more commonly than **isAlive()**. This method waits until the thread on which it is called terminates.

```
final void join() throws InterruptedException
```

Using **join()** method, we tell our thread to wait until the specifid thread completes its execution. There are overloaded versions of **join()** method, which allows us to specify time for which you want to wait for the specified thread to terminate.

```
final void join(long milliseconds) throws InterruptedException
```

```
public class MyThread extends Thread
{
public void run()
   {
       System.out.println("r1 ");
       try{
        Thread.sleep(500);
    }catch(InterruptedException ie){}

       System.out.println("r2 ");
  }
public static void main(String[] args)
{
MyThread t1=new MyThread();
MyThread t2=new MyThread();
t1.start();
t2.start();
System.out.println(t1.isAlive());
System.out.println(t2.isAlive());
}
}
```

Output

```
r1
true
true
r1
r2
r2
```

example without join:

```
public class MyThread extends Thread
{
public void run()
   {
       System.out.println("r1 ");
       try{
        Thread.sleep(500);
    }catch(InterruptedException ie){}
```

```
       System.out.println("r2 ");
  }
public static void main(String[] args)
{
MyThread t1=new MyThread();
MyThread t2=new MyThread();
t1.start();
t2.start();
}
}
```

Output

```
r1
r1
r2
r2
```

In this above program two thread t1 and t2 are created. t1 starts first and after printing "r1" on console thread t1 goes to sleep for 500 mls.At the same time Thread t2 will start its process and print "r1" on console and then goes into sleep for 500 mls. Thread t1 will wake up from sleep and print "r2" on console similarly thread t2 will wake up from sleep and print "r2" on console. So you will get output like `r1 r1 r2 r2`

example with join:

```
public class MyThread extends Thread
{
public void run()
   {
       System.out.println("r1 ");
       try{
        Thread.sleep(500);
    }catch(InterruptedException ie){}

       System.out.println("r2 ");
  }
public static void main(String[] args)
{
MyThread t1=new MyThread();
MyThread t2=new MyThread();
t1.start();

try{
  t1.join();            //Waiting for t1 to finish
}catch(InterruptedException ie){}

t2.start();
}
}
```

Output

```
r1
r2
r1
r2
```

full example:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package joinex;

class NewThread implements Runnable {

   String name; // name of thread
   Thread t;

   NewThread(String threadname) {
      name = threadname;
      t = new Thread(this, name);
      System.out.println("New thread: " + t);
      t.start(); // Start the thread
   }
// This is the entry point for thread.

   public void run() {
      try {
         for (int i = 5; i > 0; i--) {
            System.out.println(name + ": " + i);
            Thread.sleep(1000);
         }
      } catch (InterruptedException e) {
            System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
    }
}

public class JoinEx {

    public static void main(String args[]) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        NewThread ob3 = new NewThread("Three");
        System.out.println("Thread One is alive: "
                + ob1.t.isAlive());
        System.out.println("Thread Two is alive: "
                + ob2.t.isAlive());
        System.out.println("Thread Three is alive: "
                + ob3.t.isAlive());
// wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Thread One is alive: "
                + ob1.t.isAlive());
        System.out.println("Thread Two is alive: "
                + ob2.t.isAlive());
        System.out.println("Thread Three is alive: "
                + ob3.t.isAlive());
        System.out.println("Main thread exiting.");
    }
}
```

Output

```
run:
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Thread One is alive: true
Thread Two is alive: true
Two: 5
Thread Three is alive: true
Waiting for threads to finish.
Three: 5
One: 4
Two: 4
Three: 4
Three: 3
One: 3
Two: 3
Three: 2
Two: 2
One: 2
Two: 1
One: 1
Three: 1
Three exiting.
One exiting.
Two exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

Example:
Write a java project that has four threads, at the beginning of the execution all of them are alive, thread3 should not start its operation until both of thread 1 and 2 finish their execution, similarly, thread 4 cannot start until thread 3 finishes execution, assume the code of the thread is printing any message of your choice.

```
package threadexample;


class NewThread extends Thread {

   String name; // name of thread
   Thread t;
   boolean stop;

   NewThread(String threadname, boolean status) {
      name = threadname;
      t=new Thread(this, name);
      System.out.println("New thread: " + t);
      t.start(); // Start the thread
      stop = status;
   }
// This is the entry point for thread.
   public void setStop(boolean status){
      stop=status;
   }

   public void run() {

      while(stop) ;

      System.out.println(name + " is start working...");

      try {
            Thread.sleep(1000);
      } catch (InterruptedException e) {}
```

```
        System.out.println(name + " is finished working.");
    }
}

class NewThread3 extends Thread {

    String name; // name of thread
    Thread t;
    boolean stop;

    NewThread3(String threadname, boolean status) {

        name = threadname;
        t=new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
        stop=status;
    }

// This is the entry point for thread.
    public void setStop(boolean status){
        stop=status;
    }


    public void run() {
        while(stop)
        try {
                Thread.sleep(1000);
        } catch (InterruptedException e) {};


        System.out.println(name + " is start working...");

        try {
                Thread.sleep(1000);
        } catch (InterruptedException e) {}
        System.out.println(name + " is finished working.");
    }
}

class NewThread4 extends Thread {

    String name; // name of thread
    boolean stop;
    Thread t;

    public void setStop(boolean status){
        stop=status;
    }



    NewThread4(String threadname, boolean status) {
        name = threadname;
        t=new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
        stop=status;
    }
// This is the entry point for thread.

    public void run() {
        while(stop)
        try {
                Thread.sleep(1000);
        } catch (InterruptedException e) {};

        System.out.println(name + " is start working...");

        try {
                Thread.sleep(1000);
        } catch (InterruptedException e) {}

        System.out.println(name + " is finished working.");
    }
}

public class ThreadExample {

    @SuppressWarnings("empty-statement")
    public static void main(String args[]) {
        NewThread th1 = new NewThread("One",false);
        NewThread th2 = new NewThread("Two",false);
        NewThread3 th3 = new NewThread3("Three",true);
        NewThread4 th4 = new NewThread4("Four",true);
        System.out.println("Thread One is alive: "+ th1.t.isAlive());
        System.out.println("Thread Two is alive: "+ th2.t.isAlive());
        System.out.println("Thread Three is alive: "+th3.t.isAlive());
        System.out.println("Thread Four is alive: "+ th4.t.isAlive());

        //controlling the thread exeution
        while (th1.t.isAlive() || th2.t.isAlive());
        th3.setStop(false);
        while (th3.t.isAlive() );
          //System.out.println("status of thread 3 is : "+ th3.t.isAlive());

        th4.setStop(false);
```

// wait for threads to finish; below is the output in case of not existence and existence of this code.

```
      try {
        System.out.println("Waiting for threads to finish.");
        th1.t.join();
        th2.t.join();
        th3.t.join();
        th4.t.join();
        System.out.println("all threads are finished execution...");

      } catch (InterruptedException e) {
        System.out.println("Main thread Interrupted");
      }
      System.out.println("Thread One is alive: "
          + th1.t.isAlive());
        System.out.println("Thread Two is alive: "
              + th2.t.isAlive());
        System.out.println("Thread Three is alive: "
              + th3.t.isAlive());
        System.out.println("Thread Four is alive: "
              + th4.t.isAlive());
        System.out.println("Main thread exiting.");
    }
}
```

Output in case of **no join** of threads at end:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One is start working...
Two is start working...
New thread: Thread[Three,5,main]
New thread: Thread[Four,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Thread Four is alive: true
One is finished working.
Two is finished working.
Three is start working...
Three is finished working.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Thread Four is alive: true
Main thread exiting.
Four is start working...
Four is finished working.
```

(case 1)

Output in case of join of threads at end:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One is start working...
Two is start working...
New thread: Thread[Three,5,main]
New thread: Thread[Four,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Thread Four is alive: true
Two is finished working.
One is finished working.
Three is start working...
Three is finished working.
Waiting for threads to finish.
Four is start working...
Four is finished working.
all threads are finished execution...
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Thread Four is alive: false
Main thread exiting.
```

(case 2)

We can notice that, in (case 1), thread four is still running although main is existing, and the output just came behind the main output, whereas, in (case 2), all threads are finished exeution and after that the main just prints out its message and exits.

Example:

Write a program that has four threads with tasks distributed as following table:

Thread Name	Function
Reader	Creates a vector with 50 random integer numbers (0..100)
Average	Finds the average of all numbers
Pass	Finds the count of numbers above 49
Fail	Finds the count of numbers below 50

Sol:

```
package threadexample;


class ReaderTh extends Thread {

   String name; // name of thread
   Thread t;
   boolean stop;
   int[] A;

   ReaderTh(String threadname, boolean status) {
      name = threadname;
      t=new Thread(this, name);
      System.out.println("New thread: " + t);
      t.start(); // Start the thread
      stop = status;
      A=new int[100];
   }
// This is the entry point for thread.
   public void setStop(boolean status){
      stop=status;
   }

   public int[] getArray() {
      return A;
   }
```

```
        public void run() {

            while(stop)
            try {
                    Thread.sleep(1000);
            } catch (InterruptedException e) {};

            System.out.println(name + " is start working...");

            for(int i=0;i<100;i++)
                A[i]=(int) (Math.random()*100);

            for(int i=0;i<100;i++)
                System.out.println("next number is: "+A[i]);

            System.out.println(name + " is finished working.");
        }
    }

    //...........................................................................................

    class AverageTh extends Thread {

        String name; // name of thread
        Thread t;
        boolean stop;
        int[] A;

        AverageTh(String threadname, boolean status) {

            name = threadname;
            t=new Thread(this, name);
            System.out.println("New thread: " + t);
            t.start(); // Start the thread
            stop=status;
        }
        public void setArray(int[] A) {
            this.A=A;
        }

        public int[] getArray() {
            return A;
        }

    // This is the entry point for thread.
        public void setStop(boolean status){
            stop=status;
        }


        public void run() {
            while(stop)
            try {
                    Thread.sleep(1000);
            } catch (InterruptedException e) {};


            System.out.println(name + " is start working...");
            int av=0;
            for(int i=0;i<100;i++)
                av=av+A[i];
            av=av/100;

            System.out.println(" the average is :"+ av);
            System.out.println(name + " is finished working.");

        }
    }


    //..........................................................................................

    class PassTh extends Thread {
       String name; // name of thread
       Thread t;
       boolean stop;
       int[] A;

       PassTh(String threadname, boolean status) {

          name = threadname;
          t=new Thread(this, name);
          System.out.println("New thread: " + t);
          t.start(); // Start the thread
          stop=status;
       }

       public void setArray(int[] A) {
          this.A=A;
       }

       public int[] getArray() {
          return A;
       }

    // This is the entry point for thread.
       public void setStop(boolean status){
          stop=status;
       }


       public void run() {
          while(stop)
          try {
                Thread.sleep(1000);
          } catch (InterruptedException e) {};


          System.out.println(name + " is start working...");
          int count=0;
          for(int i=0;i<100;i++)
```

```
          if(A[i]>=50) count++;

        System.out.println(" the count of passed is :"+ count);
        System.out.println(name + " is finished working.");

    }
  }


  //...............................................................................


  class FailTh extends Thread {

    String name; // name of thread
    Thread t;
    boolean stop;
    int[] A;

    FailTh(String threadname, boolean status) {

        name = threadname;
        t=new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
        stop=status;
    }

    public void setArray(int[] A) {
        this.A=A;
    }

    public int[] getArray() {
        return A;
    }

  // This is the entry point for thread.
    public void setStop(boolean status){
        stop=status;
    }


    public void run() {
        while(stop)
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {};


        System.out.println(name + " is start working...");
        int count=0;
        for(int i=0;i<100;i++)
            if(A[i]<50) count++;

        System.out.println(" the count of fails is :"+ count);
        System.out.println(name + " is finished working.");

    }
}

//.........................................................................................
public class ThreadExample {
    public static void main(String args[]) {
        ReaderTh th1 = new ReaderTh("Reader",true);
        AverageTh th2 = new AverageTh("Average",true);
        PassTh th3 = new PassTh("Pass",true);
        FailTh th4 = new FailTh("Fail",true);


        th1.setStop(false);
        while (th1.t.isAlive());

        int[] A=th1.getArray();
        th2.setArray(A); th3.setArray(A); th4.setArray(A);

        th2.setStop(false);
        th3.setStop(false);
        th4.setStop(false);

        try {
           System.out.println("Waiting for threads to finish.");
           th2.t.join();
           th3.t.join();
           th4.t.join();
           System.out.println("all threads are fuinished execution...");

        } catch (InterruptedException e) {}

        System.out.println("Main thread exiting.");
    }
}
```

```
run:
New thread: Thread[Reader,5,main]
New thread: Thread[Average,5,main]
New thread: Thread[Pass,5,main]
New thread: Thread[Fail,5,main]
Reader is start working...
next number is: 79
next number is: 96
next number is: 20
next number is: 48
next number is: 18 …. (printing all generated numbers)
Reader is finished working.
Waiting for threads to finish.
Average is start working...
the average is :46
Average is finished working.
Pass is start working...
 the count of passed is :45
Pass is finished working.
Fail is start working...
 the count of fails is :55
```

```
Fail is finished working.
all threads are finished execution...
Main thread exiting.
```

4- Synchronization in Java

The **synchronized** keyword is all about different threads reading and writing to the same variables, objects and resources. This is not a trivial topic in Java, but here is a quote from Sun:

synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods.

The synchronization is mainly used for the following two reasons:

A- To prevent thread interference.

B- To prevent consistency problem (memory consistency error).

As follows:

A- Thread Interderence

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

`Counter` is designed so that each invocation of `increment` will add 1 to `c`, and each invocation of `decrement` will subtract 1 from `c`. However, if a `Counter` object is referenced from **multiple threads**, interference between threads may prevent this from happening as expected.

Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

B- To prevent consistency problem (memory consistency error)

Memory consistency errors occur when different threads have inconsistent views of what should be the same data.

The key to avoiding memory consistency errors is understanding the *happens-before* relationship. This relationship is simply a guarantee that memory writes by one specific statement are visible to another specific statement. To see this, consider the following example. Suppose a simple `int` field is defined and initialized:

```
int counter = 0;
```

The `counter` field is shared between two threads, A and B. Suppose thread A increments `counter`:

```
counter++;
```

Then, shortly afterwards, thread B prints out `counter`:

```
System.out.println(counter);
```

If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be "1". But if the two statements are executed in separate threads, the value printed out might well be "0", because there's no guarantee that thread A's change to `counter` will be visible to thread B — unless the programmer has established a **happens-before** relationship between these two statements.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
   Mutual Exclusive helps keep threads from interfering with one another while sharing data.
   a) Synchronized method.
   b) Synchronized block.
   c) Static synchronization.
2. Cooperation (Inter-thread communication in java)

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

```
package threadexample;
class Table{

void printTable(int n){//method not synchronized
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
     Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }

 }
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
```

```
this.t=t;
}
public void run(){
t.printTable(100);
}
}

class TestSynchronization1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output

```
 5
100
10
200
15
300
20
400
25
500
```

As seen, the problem is that the two threads are running interference with each another and we cannot split the output of each thread in this case.

a) Synchronized method.

If you declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
//example of java synchronized method
class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
     Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }

 }
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
Output: 5
        10
        15
        20
        25
        100
        200
        300
        400
        500
```

b) Synchronized block.

Synchronized block can be used to perform synchronization on any specific resource of the method. Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block. If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

```
synchronized (object reference expression) {
  //code block
}
```

As follows:

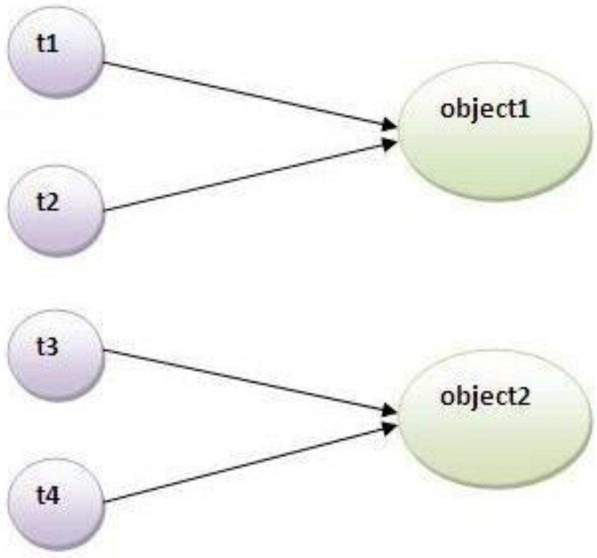
```
class Table{

 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
     System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
```

```
public class TestSynchronizedBlock1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
Output:5
       10
       15
       20
       25
       100
       200
       300
       400
       500
```

c) Static synchronization.

If you make any static method as synchronized, the lock will be on the class not on object.



Problem without static synchronization

Suppose there are two objects of a shared class(e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock.But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

Example of static synchronization

In this example we are applying synchronized keyword on the static method to perform static synchronization.

```
class Table{

 synchronized static void printTable(int n){
   for(int  i=1;i<=10;i++){
     System.out.println(n*i);
     try{
       Thread.sleep(400);
     }catch(Exception e){}
   }
 }
}

class MyThread1 extends Thread{
public void run(){
Table.printTable(1);
}
}

class MyThread2 extends Thread{
public void run(){
Table.printTable(10);
}
}

class MyThread3 extends Thread{
public void run(){
```

```
Table.printTable(100);
}
}

 class MyThread4 extends Thread{
public void run(){
Table.printTable(1000);
}
}

public class TestSynchronization4{
public static void main(String t[]){
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();
MyThread3 t3=new MyThread3();
MyThread4 t4=new MyThread4();
t1.start();
t2.start();
t3.start();
t4.start();
}
}
```

```
Output: 1
        2
        3
        4
        5
        6
        7
        8
        9
        10
        10
        20
        30
        40
        50
        60
        70
        80
        90
        100
        100
```

```
200
300
400
500
600
700
800
900
1000
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
```

Another example using synchronized method:

```
package threadexample;


    class  sharedData {
        public static int counter;

        sharedData() {
            counter=0;
        }

       public void printOut(String name) {
        counter++;
        System.out.println("Current Counter is: " + counter + ", updated by: "
+ name);
        }
    }
//-------------------------------------------------------------------
    class Thread1  extends Thread {

        sharedData d;
        Thread t;

        Thread1 (sharedData data, String name) {
            t=new Thread(this, name);
            this.d=data;
            t.start();
        }

    public void run() {
                for(int i=0; i<5; i++){
                    d.printOut("t1");
                    try{
                        Thread.sleep(1000);
                    }
                    catch (InterruptedException e) {}

                }

        }
    }
//-------------------------------------------------------------

    class Thread2  extends Thread {

        sharedData d;
        Thread t;

        Thread2 (sharedData data, String name) {
            t=new Thread(this, name);
            this.d=data;
            t.start();
        }

    public void run() {
                for(int i=0; i<5; i++){
                    d.printOut("t2");
                    try{
                        Thread.sleep(1000);
                    }
                    catch (InterruptedException e) {}

                }

    }

    }
//------------------------------------------------------------

public class ThreadExample {

    public static void main(String[] args) {
        sharedData data=new sharedData();

        Thread1 t1=new Thread1(data, "one");
        Thread2 t2=new Thread2(data, "two");


    }
}
```

Output

```
run:
Current Counter is: 1, updated by: t1
Current Counter is: 2, updated by: t2
Current Counter is: 3, updated by: t2
Current Counter is: 3, updated by: t1
Current Counter is: 5, updated by: t2
Current Counter is: 5, updated by: t1
Current Counter is: 7, updated by: t1
Current Counter is: 7, updated by: t2
Current Counter is: 9, updated by: t1
Current Counter is: 9, updated by: t2
```

But by changing the method `countMe()` to:

```
private synchronized static void printOut(){
        counter++;
        System.out.println("Current Counter is: " + counter + ", updated by: "
+ name);
}
```

We can get this output:

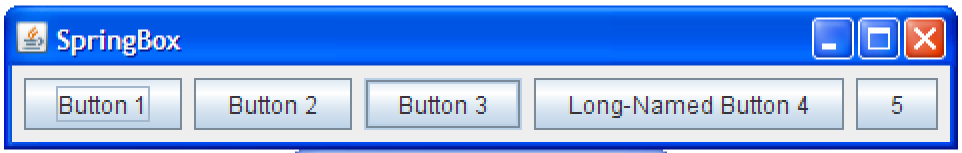
```
run:
Current Counter is: 1, updated by: t1
Current Counter is: 2, updated by: t2
Current Counter is: 3, updated by: t2
Current Counter is: 4, updated by: t1
Current Counter is: 5, updated by: t2
Current Counter is: 6, updated by: t1
Current Counter is: 7, updated by: t1
Current Counter is: 8, updated by: t2
Current Counter is: 9, updated by: t2
Current Counter is: 10, updated by: t1
```

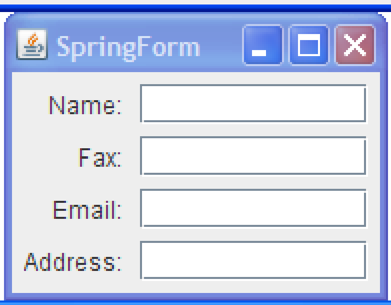
So, we can see that the data are updated successfully without any errors.

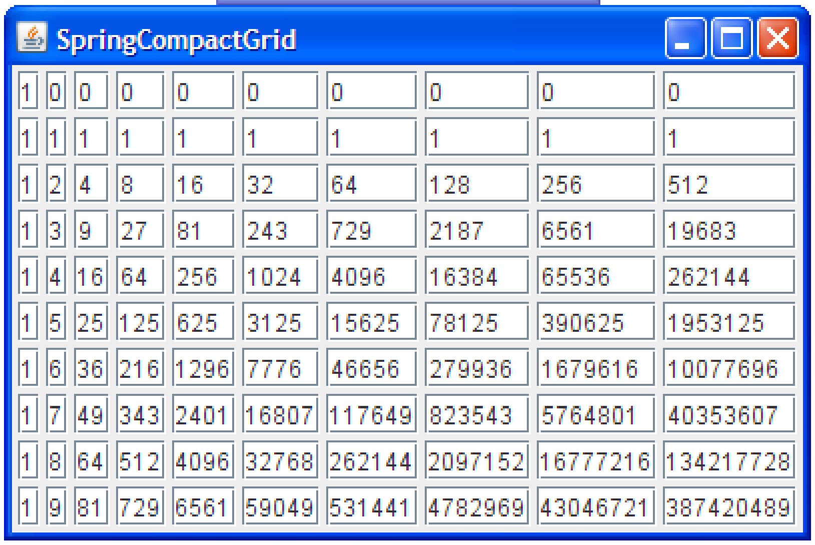
SpringLayout

The SpringLayout class was added in JDK version 1.4 to support layout in GUI builders. SpringLayout is a very flexible layout manager that can emulate many of the features of other layout managers. SpringLayout is, however, very low-level, and as such, you really should only use it with a GUI builder rather than attempting to code a spring layout manager by hand. This section begins with a simple example showing all the things you need to remember to create your first spring layout — and what happens when you forget them! Later it presents utility methods that let you lay out components in a couple of different types of grids.

Here are pictures of some of the layouts that can be applied using spring layout in this course we use only the simple design of it:







1	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	2	4	8	16	32	64	128	256	512
1	3	9	27	81	243	729	2187	6561	19683
1	4	16	64	256	1024	4096	16384	65536	262144
1	5	25	125	625	3125	15625	78125	390625	1953125
1	6	36	216	1296	7776	46656	279936	1679616	10077696
1	7	49	343	2401	16807	117649	823543	5764801	40353607
1	8	64	512	4096	32768	262144	2097152	16777216	134217728
1	9	81	729	6561	59049	531441	4782969	43046721	387420489

How Spring Layouts Work

Spring layouts do their job by defining directional relationships, or *constraints*, between the edges of components. For example, you might define that the left edge of one component is a fixed distance (5 pixels, say) from the right edge of another component.

In a SpringLayout, the position of each edge is dependent on the position of just one other edge. If a constraint is subsequently added to create a new binding for an edge, the previous binding is discarded and the edge remains dependent on a single edge.
Unlike many layout managers, SpringLayout does not automatically set the location of the components it manages. If you hand-code a GUI that uses SpringLayout, remember to initialize component locations by constraining the west/east and north/south locations. Depending on your constraints, you may also need to set the size of the container explicitly.
Components define *edge* properties, which are connected by Spring instances. Each spring has four properties — its *minimum*, *preferred*, and *maximum* values and its actual (current) *value*. The springs associated with each component are collected into a SpringLayout.Constraints object.
An instance of the Spring class holds three properties that characterize its behaviour: the minimum, preferred, and maximum values. Each of these properties may be involved in defining its fourth value, property based on a series of rules.
An instance of the Spring class can be visualized as a mechanical spring that provides a corrective force as the spring is compressed or stretched away from its preferred value. This force is modelled as a linear function of the distance from the preferred value but with two constants -- one for the compressional force and one for the tensional one. Those constants are specified by the minimum and maximum values of the spring such that a spring at its minimum value produces an equal and opposite force to that created at its maximum value. The difference between the preferred and minimum values, therefore, represents the ease with which the spring can be compressed. The difference between its maximum and preferred values indicates the ease with which the Spring can be extended. Based on this, a SpringLayout can be visualized as a set of objects connected by a set of springs on their edges.

Example: Spring layout

```
package javaapplication1;

/**
 *
 * @author safa
 */
import javax.swing.*;
import java.awt.*;
public class JavaApplication1 {

   /**
    * @param args the command line arguments
    */
 public static void main(String[] args) {
 // TODO code application logic here is the main window Method for setting the default look and feel decorated status of the JFrame.
JFrame.setDefaultLookAndFeelDecorated(true);
// Creating an object of the "JFrame" class
JFrame f = new JFrame("Spring Layout Example");
```

```
// Function to set the default close operation status of JFrame
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// method to determine the size status of the JFrame
f.setSize(310, 210);
// to get the content pane
Container cntt = f.getContentPane();
// Creating Object of "SpringLayout" class
SpringLayout sprLayout = new SpringLayout();
// for setting the layout class
f.setLayout(sprLayout);
// Initializing the object "btn1" of the JButton class.
Component btn1 = new JButton("C++");
// Initializing the object "btn2" of the JButton class.
Component btn2 = new JButton("Python");
// Initializing the object "btn3" the JButton class.
Component btn3 = new JButton("JAVA");
// Initializing the object "btn4" of the JButton class.
Component btn4 = new JButton("NETWORKING");
// Adding the JButton "btn1" on the frame f
f.add(btn1);
// Adding the JButton "btn2" on the frame f
f.add(btn2);
// Adding the JButton "btn3" on the frame f
f.add(btn3);
// Adding the JButton "btn4" on the frame f
f.add(btn4);
// It is used for inserting the layout constraint in the JFrame by using the springLayout class on
the btn1 JButton
sprLayout.putConstraint(SpringLayout.WEST, btn1, 24, SpringLayout.WEST, cntt);
 sprLayout.putConstraint(SpringLayout.NORTH, btn1, 9, SpringLayout.NORTH, cntt);
// It is used for inserting the layout constraint in the JFrame using the springLayout class on the
btn2 JButton
sprLayout.putConstraint(SpringLayout.WEST, btn2, 49, SpringLayout.WEST, cntt);
sprLayout.putConstraint(SpringLayout.NORTH, btn2,10, SpringLayout.SOUTH, btn1);
// It is used for inserting the layout constraint in the JFrame using springLayout class on the btn3
JButton
sprLayout.putConstraint(SpringLayout.WEST, btn3,74, SpringLayout.WEST, cntt);

sprLayout.putConstraint(SpringLayout.NORTH, btn3, 9, SpringLayout.SOUTH, btn2);
// It is used for inserting the layout constraint in the JFrame using sprLayout class on the btn4
JButton
sprLayout.putConstraint(SpringLayout.WEST, btn4, 20, SpringLayout.EAST, btn1);
sprLayout.putConstraint(SpringLayout.NORTH, btn4,9, SpringLayout.NORTH, cntt);
```

```
// method for setting the visibility status of the JFrame
f.setVisible(true);
}  }
```