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:

```
public class HelloRunnable implements Runnable {

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

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

}
```

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:

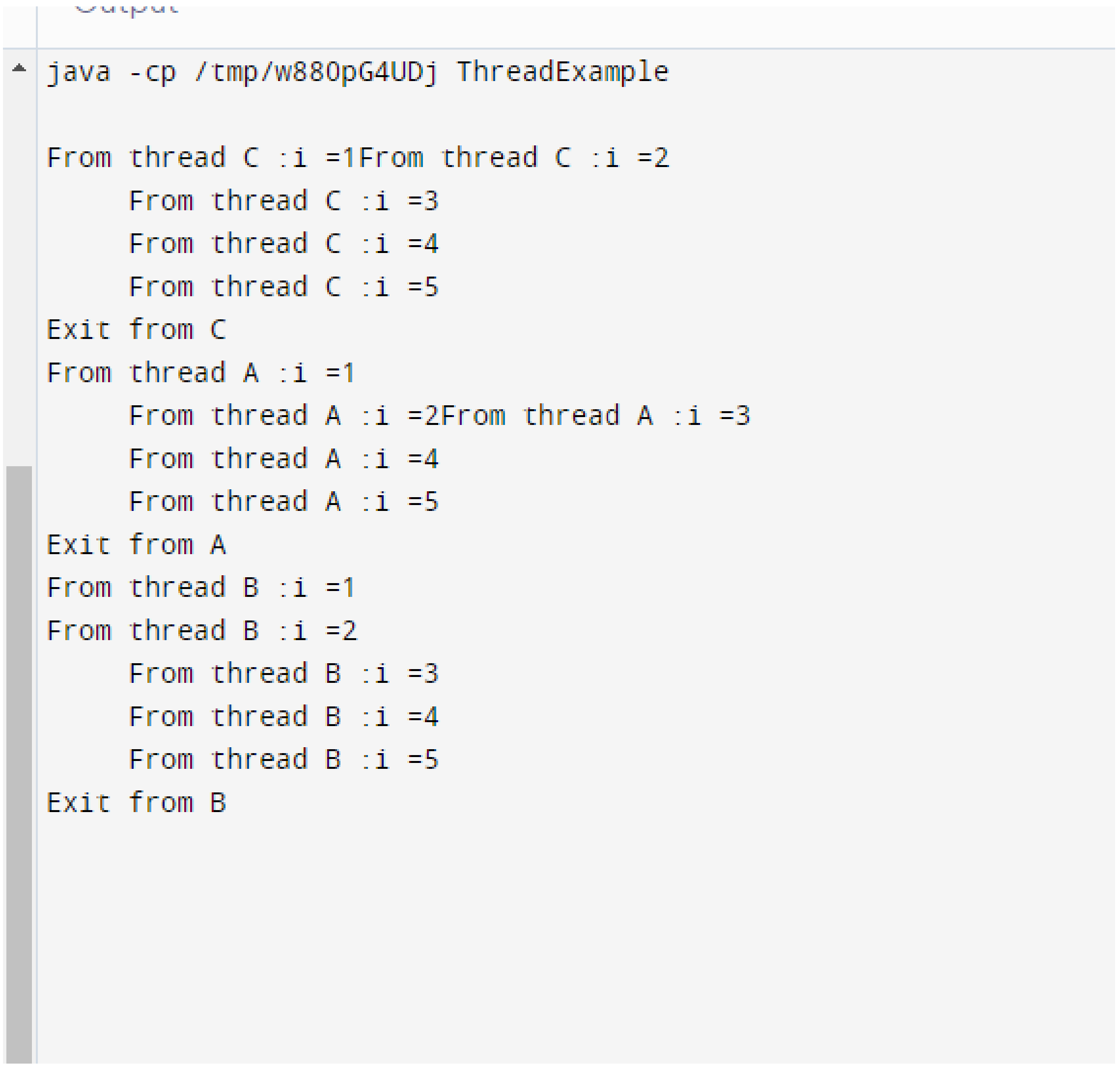
```
public class HelloThread extends Thread {

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

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

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

Program1:

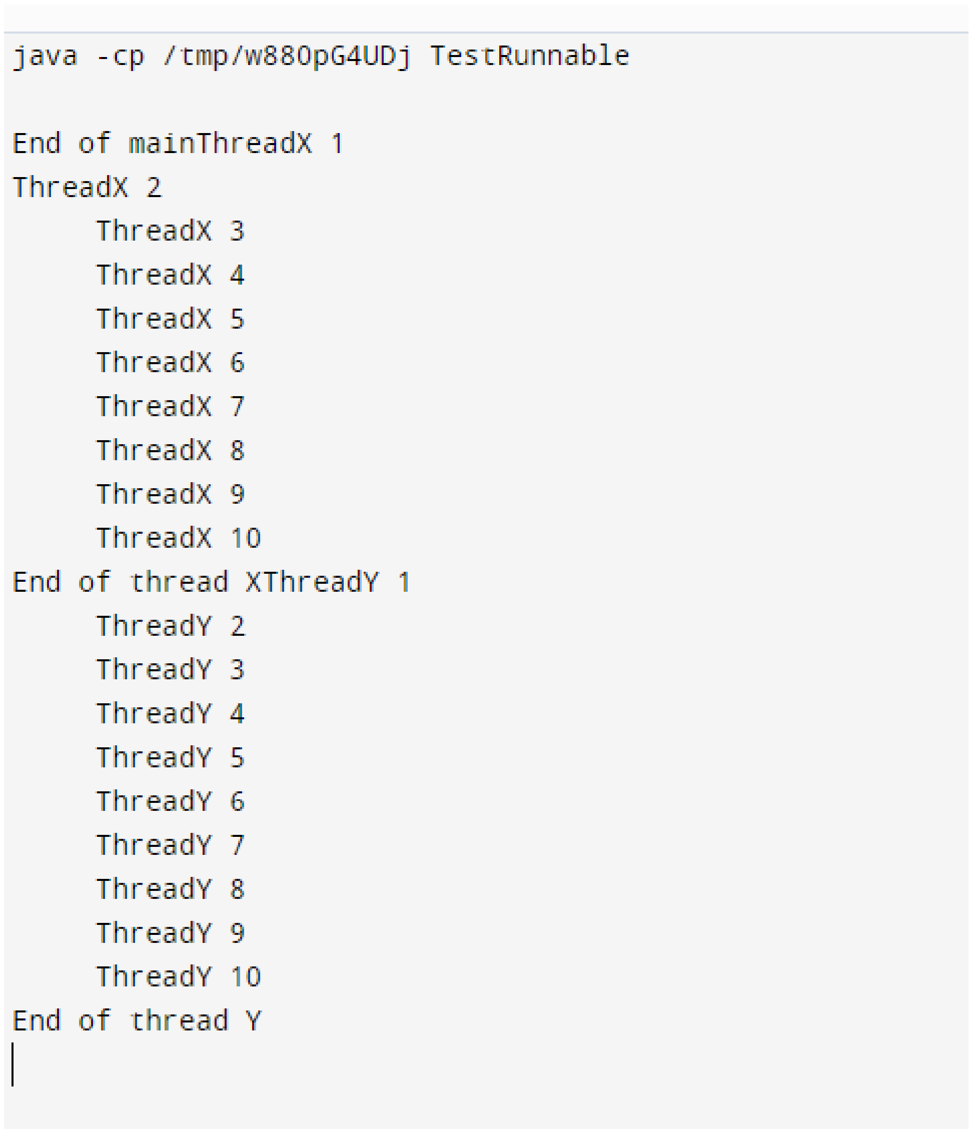
```
class A extends Thread
{
 public void run()
 {
   for(int i=1;i<=5;i++)
           { System.out.println("\t From thread A :i =" +i); }
           System.out.println("Exit from A");
 }
}
```

```
class B extends Thread
{
 public void run()
 {
   for(int i=1;i<=5;i++)
		{ System.out.println("\t From thread B :i =" +i); }
		System.out.println("Exit from B");
 }
}

class C extends Thread
{
 public void run()
 {
   for(int i=1;i<=5;i++)
		{ System.out.println("\t From thread C :i =" +i); }
		System.out.println("Exit from C");
 }
}

class ThreadExample
{
		public static void main(String a[])
		{ A obj =new A();

				new A().start();
				new B().start();
				new C().start();
		}
}
```



```
java -cp /tmp/w88OpG4UDj ThreadExample

From thread C :i =1From thread C :i =2
     From thread C :i =3
     From thread C :i =4
     From thread C :i =5
Exit from C
From thread A :i =1
     From thread A :i =2From thread A :i =3
     From thread A :i =4
     From thread A :i =5
Exit from A
From thread B :i =1
From thread B :i =2
     From thread B :i =3
     From thread B :i =4
     From thread B :i =5
Exit from B
```

Program2:
```
class X implements Runnable
{
 public void run()
{
 for (int i=1;i<=10;i++)
        {System.out.println("\t ThreadX "+ i);}
        System.out.println("End of thread X");
}

}

class Y implements Runnable
{
 public void run()
{
 for (int i=1;i<=10;i++)
        {System.out.println("\t ThreadY "+ i);}
        System.out.println("End of thread Y");
}

}

class TestRunnable
{
public static void main(String a[])
        { X o1= new X();

        Thread tx =new Thread(o1);
        tx.start();

        Y o2= new Y();
        Thread tx1 =new Thread(o2);
        tx1.start();

        System.out.println("End of main");
        }
}
```



```
java -cp /tmp/w88OpG4UDj TestRunnable

End of mainThreadX 1
ThreadX 2
	ThreadX 3
	ThreadX 4
	ThreadX 5
	ThreadX 6
	ThreadX 7
	ThreadX 8
	ThreadX 9
	ThreadX 10
End of thread XThreadY 1
	ThreadY 2
	ThreadY 3
	ThreadY 4
	ThreadY 5
	ThreadY 6
	ThreadY 7
	ThreadY 8
	ThreadY 9
	ThreadY 10
End of thread Y
```

The SimpleThreads Example

The following example brings together some of the concepts of this section. `SimpleThreads` consists of two threads. The first is the main thread that every Java application has. The main thread creates a new thread from the `Runnable` object, `MessageLoop`, and waits for it to finish. If the `MessageLoop` thread takes too long to finish, the main thread interrupts it.

The `MessageLoop` thread prints out a series of messages. If interrupted before it has printed all its messages, the `MessageLoop` thread prints a message and exits.

```
public class SimpleThreads {

   // Display a message, preceded by
   // the name of the current thread
```

```
static void threadMessage(String message) {
   String threadName =
      Thread.currentThread().getName();
   System.out.format("%s: %s%n",
                 threadName,
                 message);
}

private static class MessageLoop
   implements Runnable {
   public void run() {
      String importantInfo[] = {
         "Mares eat oats",
         "Does eat oats",
         "Little lambs eat ivy",
         "A kid will eat ivy too"
      };
      try {
         for (int i = 0;
             i < importantInfo.length;
             i++) {
            // Pause for 4 seconds
            Thread.sleep(4000);
            // Print a message
            threadMessage(importantInfo[i]);
         }
      } catch (InterruptedException e) {
         threadMessage("I wasn't done!");
      }
   }
}

public static void main(String args[])
   throws InterruptedException {

   // Delay, in milliseconds before
   // we interrupt MessageLoop
   // thread (default one hour).
   long patience = 1000 * 60 * 60;

   // If command line argument
   // present, gives patience
   // in seconds.
   if (args.length > 0) {
      try {
         patience = Long.parseLong(args[0]) * 1000;
      } catch (NumberFormatException e) {
         System.err.println("Argument must be an integer.");
```

```
                System.exit(1);
            }
        }

        threadMessage("Starting MessageLoop thread");
        long startTime = System.currentTimeMillis();
        Thread t = new Thread(new MessageLoop());
        t.start();

        threadMessage("Waiting for MessageLoop thread to finish");
        // loop until MessageLoop
        // thread exits
        while (t.isAlive()) {
            threadMessage("Still waiting...");
            // Wait maximum of 1 second
            // for MessageLoop thread
            // to finish.
            t.join(1000);
            if (((System.currentTimeMillis() - startTime) > patience)
                  && t.isAlive()) {
                threadMessage("Tired of waiting!");
                t.interrupt();
                // Shouldn't be long now
                // -- wait indefinitely
                t.join();
            }
        }
        threadMessage("Finally!");
    }
}
```
OUTPUT:
```
main: Starting MessageLoop thread
main: Waiting for MessageLoop thread to finishmain: Still waiting...
main: Still waiting...main: Still waiting...
main: Still waiting...
Thread-0: Mares eat oatsmain: Still waiting...
main: Still waiting...
main: Still waiting...
main: Still waiting...
Thread-0: Does eat oats
main: Still waiting...
main: Still waiting...
main: Still waiting...
main: Still waiting...
Thread-0: Little lambs eat ivy
main: Still waiting...
main: Still waiting...
main: Still waiting...main: Still waiting...
```

```
Thread-0: A kid will eat ivy too
main: Finally!
```

Synchronization

Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: *thread interference* and *memory consistency errors*. The tool needed to prevent these errors is *synchronization*.

However, synchronization can introduce *thread contention*, which occurs when two or more threads try to access the same resource simultaneously *and* cause the Java runtime to execute one or more threads more slowly, or even suspend their execution. Starvation and livelock are forms of thread contention. Which is also measure as liveliness of thread.

This section covers the following topics:

- Thread Interference describes how errors are introduced when multiple threads access shared data.

  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.

- Memory Consistency Errors describes errors that result from inconsistent views of shared memory.

  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. (In java)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.

There are several actions that create happens-before relationships. One of them is **synchronization**,

- Synchronized Methods describes a simple idiom that can effectively prevent thread interference and memory consistency errors.

To make a method synchronized, simply add the `synchronized` keyword to its declaration:

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

If `count` is an instance of `SynchronizedCounter`, then making these methods synchronized has two effects:

- First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
- Second, when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Note that constructors cannot be synchronized — using the `synchronized` keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

- Implicit Locks and Synchronization describes a more general synchronization idiom, and describes how synchronization is based on implicit locks.

Synchronized Statements

Another way to create synchronized code is with *synchronized statements*. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

In this example, the `addName` method needs to synchronize changes to `lastName` and `nameCount`, but also needs to avoid synchronizing invocations of other objects' methods. (Invoking other objects' methods from synchronized code can create problems that are described in the section on Liveness.) Without synchronized statements, there would have to be a separate, unsynchronized method for the sole purpose of invoking `nameList.add`.

Synchronized statements are also useful for improving concurrency with fine-grained synchronization. Suppose, for example, class `MsLunch` has two instance fields, `c1` and `c2`, that are never used together. All updates of these fields must be synchronized, but there's no reason to prevent an update of c1 from being interleaved with an update of c2 — and doing so reduces concurrency by creating unnecessary blocking.

Instead of using synchronized methods or otherwise using the lock associated with `this`, we create two objects solely to provide locks.

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Use this idiom with extreme care. You must be absolutely sure that it really is safe to interleave access of the affected fields.

Reentrant Synchronization

Recall that a thread cannot acquire a lock owned by another thread. But a thread *can* acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables *reentrant synchronization*. This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock. Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

https://javaconceptoftheday.com/print-odd-and-even-numbers-by-two-threads-in-java/

Java Program To Print Odd And Even Numbers By Two Threads :

```
//OddThread to print odd numbers
//Calls printOdd() method of SharedPrinter class until limit is exceeded.


class OddThread extends Thread
{
    int limit;

    sharedPrinter printer;


    public OddThread(int limit, sharedPrinter printer)
    {
        this.limit = limit;

        this.printer = printer;
    }

    @Override
    public void run()
    {
        int oddNumber = 1;        //First odd number is 1
        while (oddNumber <= limit)
        {
            printer.printOdd(oddNumber);       //Calling printOdd() method of
SharedPrinter class

            oddNumber = oddNumber + 2;         //Incrementing to next odd
number
        }
    }
}

//EvenThread to print even numbers.
//Calls printEven() method of SharedPrinter class until limit is exceeded.

class EvenThread extends Thread
{
    int limit;

    sharedPrinter printer;

    public EvenThread(int limit, sharedPrinter printer)
    {
        this.limit = limit;

        this.printer = printer;
    }

    @Override
    public void run()
    {
        int evenNumber = 2;           //First even number is 2

        while (evenNumber <= limit)
        {
            printer.printEven(evenNumber);          //Calling printEven()
method of SharedPrinter class

            evenNumber = evenNumber + 2;           //Incrementing to next even
number
        }
    }
}

class sharedPrinter
{
    //A boolean flag variable to check whether odd number is printed or not
    //Initially it is false.

    boolean isOddPrinted = false;

    //synchronized printOdd() method to print odd numbers. It is executed by
OddThread.
```

```
    //First checks isOddPrinted,
    //if isOddPrinted is true then it waits until next even number is printed
by EvenThread
    //If isOddPrinted is false then prints next odd number, sets isOddPrinted
to true
    //sleeps for 1 second before notifying EvenThread

    synchronized void printOdd(int number)
    {
        while (isOddPrinted)
        {
            try
            {
                wait();
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }

        System.out.println(Thread.currentThread().getName()+" : "+number);

        isOddPrinted = true;

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

        notify();
    }

    //Synchronized printEven() method to print even numbers. It is executed by
EvenThread.
    //First checks isOddPrinted,
    //if isOddPrinted is false then it waits until next odd number is printed
by OddThread
    //If isOddPrinted is true then it prints next even number, sets
isOddPrinted to false
    //sleeps for 1 second before notifying OddThread

    synchronized void printEven(int number)
    {
        while (! isOddPrinted)
        {
            try
            {
                wait();
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }

        System.out.println(Thread.currentThread().getName()+" : "+number);

        isOddPrinted = false;

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

        notify();
    }
}

//Main Class

public class MainClass
{
    public static void main(String[] args)
    {
        sharedPrinter printer = new sharedPrinter();

        OddThread oddThread = new OddThread(20, printer);

        oddThread.setName("Odd-Thread");

        EvenThread evenThread = new EvenThread(20, printer);

        evenThread.setName("Even-Thread");

        oddThread.start();

        evenThread.start();
    }
}
```

Output :

```
Odd-Thread : 1
Even-Thread : 2
Odd-Thread : 3
Even-Thread : 4
Odd-Thread : 5
```

```
Even-Thread : 6
Odd-Thread : 7
Even-Thread : 8
Odd-Thread : 9
Even-Thread : 10
Odd-Thread : 11
Even-Thread : 12
Odd-Thread : 13
Even-Thread : 14
Odd-Thread : 15
Even-Thread : 16
Odd-Thread : 17
Even-Thread : 18
Odd-Thread : 19
Even-Thread : 20
```

```
/ CPP program to demonstrate multithreading

// using three different callables.

#include <iostream>

#include <thread>

using namespace std;



// A dummy function

void foo(int Z)

{

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

        cout << "Thread using function"

               " pointer as callable\n";
    }
}

// A callable object
class thread_obj {
public:
    void operator()(int x)
    {
        for (int i = 0; i < x; i++)
            cout << "Thread using function"
                  " object as  callable\n";
    }
};

int main()
{
    cout << "Threads 1 and 2 and 3 "
         "operating independently" << endl;
```

```
// This thread is launched by using

// function pointer as callable

thread th1(foo, 3);



// This thread is launched by using

// function object as callable

thread th2(thread_obj(), 3);



// Define a Lambda Expression

auto f = [](int x) {

    for (int i = 0; i < x; i++)

        cout << "Thread using lambda"

         " expression as callable\n";

};



// This thread is launched by using

// lamda expression as callable

thread th3(f, 3);
```

```
    // Wait for the threads to finish

    // Wait for thread t1 to finish

    th1.join();



    // Wait for thread t2 to finish

    th2.join();



    // Wait for thread t3 to finish

    th3.join();



    return 0;

}
```

Output (Machine Dependent)

```
Threads 1 and 2 and 3 operating independently
Thread using function pointer as callable
Thread using lambda expression as callable
Thread using function pointer as callable
Thread using lambda expression as callable
Thread using function object as  callable
Thread using lambda expression as callable
Thread using function pointer as callable
Thread using function object as  callable
Thread using function object as  callable
```