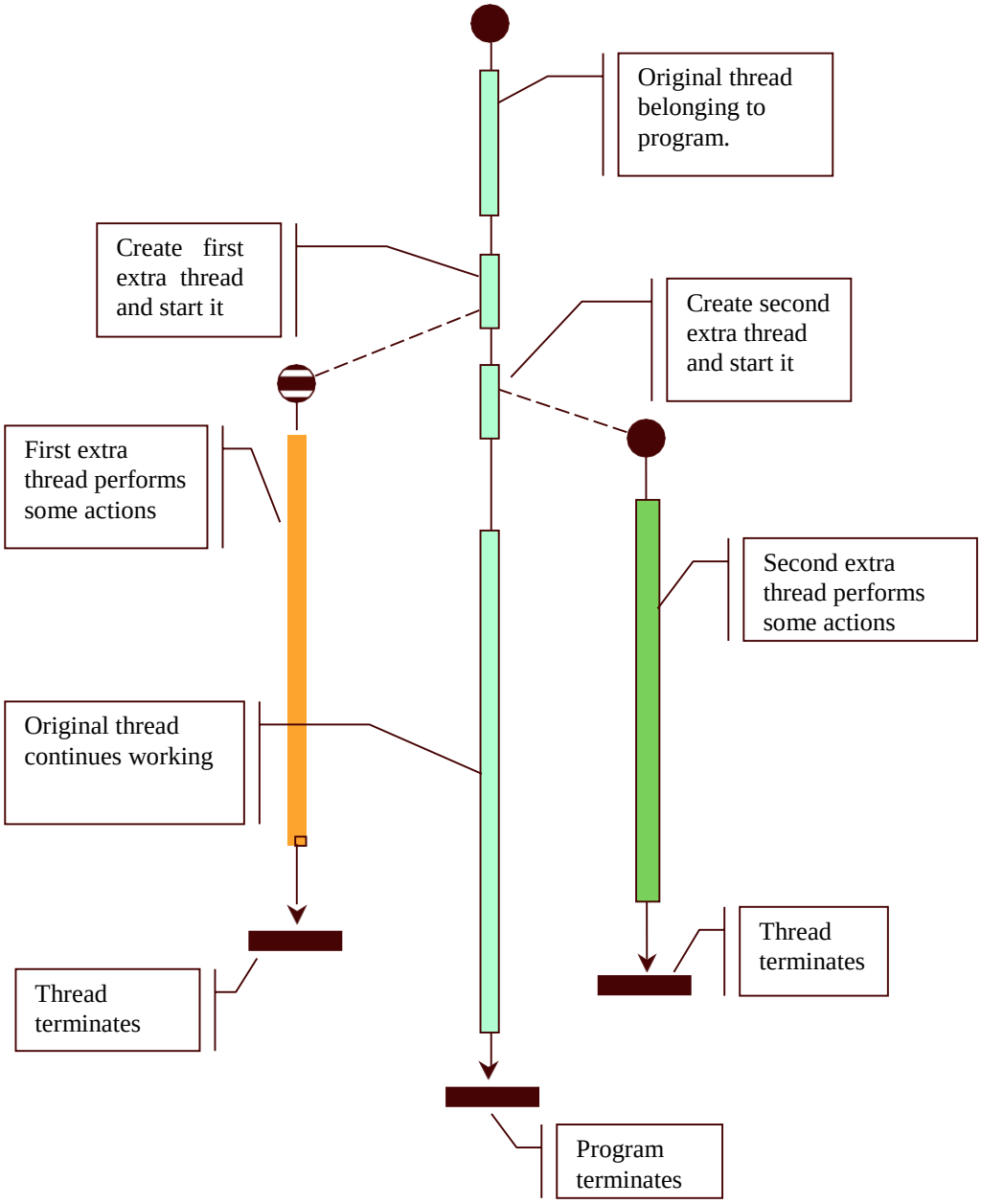
Programming using Threads

A Thread is a single sequential flow of control within a process. This simply means that while executing within a program, each thread has a beginning, a sequence, a point of execution occurring at any time during runtime of the thread and of course, an ending. Thread objects are the basis for multi-threaded programming. Multi-threaded programming allows a single program to conduct concurrently running threads that perform different tasks.

A program always has at least one thread - the one created when the program begins. When your program creates a thread, it is in addition to the threat of execution that created it.

Each additional thread is represented by an object of the class **Thread** or of a subclass of Thread. If your program is to have two additional threads, you will need to create two such objects.



The Thread Class

To start the execution of the thread you call the **start()** method for the thread object. The code that executes in a new thread is always a method called **run()** which is public accepts no arguments and doesn't return a value.

To create a new thread of execution, declare a new class which is a subclass of Thread and then override the **run()** method with the code that you want executed in this Thread. An instance of the Thread subclass should be created next with a call to the **start()** method following the instance. The **start()** method will start the thread and execute the **run()** method.

The program below, i.e., ex10, creates two threads and runs them concurrently along with the main program. The first thread will print out just even numbers whilst the second thread will print out just odd numbers. After the two thread objects, have been created they are started and the main program executes a loop to print out a string. Implement ex10.java program.

```
public class ex10 {
  public static void main(String[] args){
       DisplayEven t1 = new DisplayEven(2,100);
       DisplayOdd t2 = new DisplayOdd(1,100);
       t1.start();
       t2.start();
       for(int i=0; i < 10; i++){
              try{
                 System.out.println("Main thread");
                 Thread.sleep(80);
              }catch(InterruptedException e){}
       }
   }
}
```

A section of the Resulting it printout is shown opposite. In this example the two threads terminate before the main program. But this is not always the case. Normally a thread and once started will continue executing even if its parent program has terminated.

```
Main thread
2
1
3
4
Main thread
5
6
7
Main thread
9
10
11
```

The default thread object has an empty **run()** function and therefore it does nothing. To create a useable, thread the programmer must create his or her own **Thread** class which extend the default **Thread** class by adding data perhaps and also a run function which contains some actual called to be executed. Using Java's form of inheritance does this. The two thread classes used in ex10 are shown below. Each class calls the **sleep()** function whose parameter is a sleep time in milliseconds. You need to enclose this call in a try...catch block to handle the **InterruptedException** which may be generated if for some reason the operating system needs to interrupt the thread.

```
class DisplayEven extends Thread {
    int from;
    int to;

    public DisplayEven(int _from, int _to){
         from = _from;
         to = _to;
    }

    public void run(){
         for(int i = from; i < to; i+=2){
            try {
                  System.out.println(i);
                  Thread.sleep(70);
            }catch(InterruptedException e) {}
         }
    }
}
```

```
class DisplayOdd extends Thread {
    int from;
    int to;

    public DisplayOdd(int _from, int _to){
         from = _from;
         to = _to;
    }

    public void run(){
         for(int i = from; i < to; i+=2){
            try {
                  System.out.println(i);
                  Thread.sleep(50);
            }catch(InterruptedException e) {}
         }
    }
}
```

The Runnable Interface

Any class that is not a Thread class may be run within a thread so long as it implements the **Runnable** interface. An interface contains a list of functions that must be implemented by any class wishing to implement the interface (It is a specification that says "in order to implement this type you must include at least these functions"). The **Runnable** interface has one function called **run()** thus any class may implement the **Runnable** interface so long as it has a function called **run()**.

The thread class has a constructor that allows a Runnable class object to be passed to the Thread object as it is created. The thread object then forms a wrapper for the Runnable object and executes it. The code to initialize a thread is of the form.

```
Thread thread = new Thread( new <Runnable Class object> );
```

ex10 is repeated below in **ex10A** using Runnable objects passed to threads. The difference with ex10 above are highlighted in red. The main class and DisplayEvens and DisplayOdds classes become:

```
public class ex10A {
  public static void main(String[] args){
       Thread t1 = new Thread(new DisplayEven(2,100));
       Thread t2 = new Thread(new DisplayOdd(1,100));
       t1.start();
       t2.start();
       for(int i=0; i < 10; i++){
              try{
                 System.out.println("Main thread");
                 Thread.sleep(80);
              }catch(InterruptedException e){}
       }
   }
}
```

```
class DisplayEven implements Runnable {
    int from;
    int to;

    public DisplayEven(int _from, int _to){
         from = _from;
         to = _to;
    }

    public void run(){
         for(int i = from; i < to; i+=2){
            try {
                 System.out.println(i);
                 Thread.sleep(70);
            }catch(InterruptedException e) {}
         }
    }
}
```

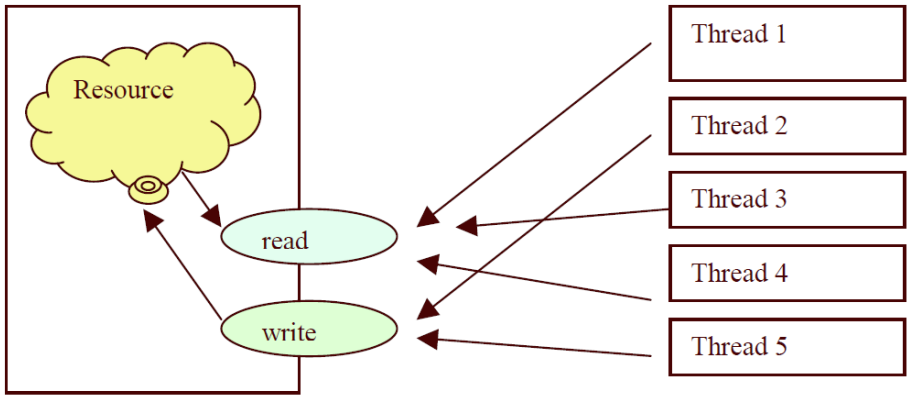
```
class DisplayOdd implements Runnable {
    int from;
    int to;

    public DisplayOdd(int _from, int _to){
         from = _from;
         to = _to;
    }

    public void run(){
         for(int i = from; i < to; i+=2){
            try {
                 System.out.println(i);
                 Thread.sleep(50);
            }catch(InterruptedException e) {}
         }
    }
}
```

Thread Synchronization

Multiple threads accessing the same resource can cause data problems unless the system is managed so as to give each thread exclusive access to the resource. Generally, in an object-oriented language such as Java the resource will be wrapped inside some class which has several functions giving access in some way to the resource encapsulated in the class.



In these circumstances one may use the keyword **synchronized** to allow the system to automatically enforce mutual exclusive access to the class functions which may be concurrently accessed.

Modify ex11.java Java Program

Consider the following Java program that decrements an integer until zero. The ex11.java program is NOT multi-threaded. Hence your task is to convert it to a multi-threaded Java program that will use **two** threads to decrement the `int number`.

Hint: in addition to the modified ex11.java class you should create another class that **implements Runnable or extends Thread.** In that class you should include the task of the Threads, e.g., to call the **decrement()** method until the global variable, e.g., **number,** becomes zero. Also, set your threads to **sleep** for 50 milliseconds (for context switch). Make sure that your output does not contain any "duplicated decrements" from a **Race Condition**.

```
class ex11 {

    public int number = 100;

    public void decrement(){
         number--;
         System.out.println("Decrement by one = " + number);
    }

    public static void main(String[] args) {
         ex11 e = new ex11();
         while(e.number > 0){
            e.decrement();
         }
    }
}
```

Implement ex12.java Java Program

Convert your Java Program ex8server.java from Practical 4 from single threaded to multithreaded. You should implement the new server, e.g., ex12server and a new Java class, e.g., ex12handler.java, to handle multiple clients. The ex8client.java may remain the same and rename it to ex12client.java.

Hint: In case your ex12client.java implementation does not prompt the user to enter data from the keyboard, you should add the following lines (used in Practical 3 ex7) in your ex12client.java:

```
BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
String sendMessage = inFromUser.readLine();
```

and send to the server the `sendMessage` string.

In this way, the client will remain connected to the server until the user enters something from the keyboard in order to test multiple client connections to your multithreaded server. After the client inputs something from keyboard its connection will be closed.