Threads 1
Creating a thread 2
Implement Runnable interface 2
Extend the Thread class 6
Useful Thread Methods: 7
Lab 7

Threads

Computers generally execute code instructions one line at a time. A PC is generally infinitely faster than a human. Some of the wasted CPU cycles can be put to use by starting a secondary activity concurrently as the task the user is running. This is all under the program's control. At other times, the task to be accomplished is long lived eg downloading a large file to disk. So rather than waiting for that activity to be completed before moving on to the next, a thread can be employed to take care of the slower process of downloading the file.

A thread is a light weigh process within a program. It executes on its own in parallel with the main body of code. It makes the program appear to do more than on thing at a time. Examples of these include starting a thread to print a document while the program is engaged accepting user input or perhaps even doing a grammar and spell check. A thread can be employed to download files over a slow network connection while the program is doing something else.

A program can have multiple threads running concurrently. Threads can be running, suspended, resumed, blocked (waiting for response) and multiple threads can have different priorities. A higher priority thread will use more CPU time and will execute more instructions in unit time that a low priority thread.

The main body of code is itself one thread. This thread can be made to rest (sleep) for a given amount of time. By simply calling Thread.sleep(milliTime), the program will wait until the time indicated by milliTime (expressed as an integer value of milliseconds) has elapsed. This function throws an InterruptedException exception and it must be caught, or declared to be thrown. In the following code snippet, the program pauses for 1.5 seconds after printing statement 1, and then it proceeds to print statement 2.

```
try
{
      System.out.println("statement 1");
      Thread.sleep(1500);
      System.out.println("statement 2");


} catch (InterruptedException e) {}
```

Note that Thread.sleep() is not needed in a thread. It in fact slows the thread down. It is used to introduce delays. It has been used in subsequent examples so that the progress of the independent threads can be observed. Do not include it in threads, unless you expressly want to slow the thread down for good reason.

Creating a thread

There are two main ways to crate a thread. Option one is to implement a runnable interface and the alternative is to extend the Thread object.

Implement Runnable interface

Using the runnable interface is particularly convenient if the object already extends another class. Java allows inheriting from only one class at a time, thus leaving the runnable interface as the only other option. The run() method in the interface has to be implemented. The code in the run() method will serve as the "thread" of code that will be executed when the thread is started. The interface is defined as follows:

```
public interface Runnable{
      public void run();
}
```

To use the interface, the following steps are followed:

1. Import the java.util package.
2. Your class should implement the Runnable interface.
3. Provide a body for the run() method.
4. Instantiate a (local) Thread object and pass the object that implements the Runnable interface as a parameter to the thread object. Eg

   ```
   Thread t= new Thread (Runnable threadObj, String threadName)
   ```

   The second parameter is just a string to uniquely identify this thread, in case multiple threads are created.
5. Call the start() method of local thread object. Note that the run() method is never called by the body of code. Instead a call is made to the start() which in turn makes a call back to the run() automatically.

Note: simply providing a method named run() in a piece of code, without implementing the Runnable interface does not result in a thread.

The following example shows a very simple case of running a thread. The CharPrinter class implements the Runnable interface. The run method has been provided. In this case, it simply prints out the characters from a to f. That will be the thread that runs in parallel with the main program. The main program instantiates a local thread object, passing an instance of the CharPrinter object. Next, the thread is asked to start(). The program then proceeds to print out 5 numbers. The output given in Listing 2 deserves some comment.

```
import java.util.*;
class Tester0
{
      public void doOperations(){

            Thread t1= new Thread(new CharPrinter());
            t1.start();
            for (int i = 0; i < 5; i++)
```

```
                System.out.println("Main Program " + "\t" + i);
        }

        public static void main(String args[])  throws Exception
        {
                Tester0 prog= new Tester0();
                prog.doOperations();
                System.out.println("Main Program Has Ended!");
        }


        class CharPrinter implements Runnable{
                private int ID;
                public void run() {
                        for (char i = 'a'; i <= 'f'; i++) {
                                System.out.println("\t CharPrinter \t" + i);
                                try {
                                        Thread.sleep((long)(0.5 * 1000));
                                } catch (InterruptedException e) {}
                        }
                        System.out.println("Char Printing Thread has
finished");
                }
        }
}
```

Listing 1 Simple thread using the Runnable Interface

Notice in the output that when the thread is started, the program does not wait for it to finish executing. It rather moves on directly to start with its main business of printing 5 numbers. The thread is long lived (in this case, ***it has been artificially slowed*** down by making it sleep for 5 seconds after printing each character. ***You do not have to call the sleep method in a thread***. We usually want a thread to finish execution quickly anyway. The output shows that the last line of the main program finishes long before the thread completes. It should be very clear that when a thread is called, the main program cannot expect a result from that thread right away before proceeding. This is especially important when the program starts a thread to read some value from a networked resource. The code for processing the result should be part of the thread. This cannot be emphasized enough!

Output:

```
$ java Tester0
        CharPrinter      a
Main Program      0
Main Program      1
Main Program      2
Main Program      3
Main Program      4
Main Program Has Ended!
        CharPrinter      b
        CharPrinter      c
        CharPrinter      d
        CharPrinter      e
        CharPrinter      f
Char Printing Thread has finished
```

Listing 2 Output of running thread

The following example show how an application can start multiple threads of the same object or of different object. One new feature is forcing the main thread to wait for a particular thread before proceeding. There may a rare need for this. (eg to properly deallocate resources in one place before shutting down an application). The output in this indicates that the final statement of the program is not executed until Thread3 has completed. The join() method forces this behavior. The main thread will wait at the point where the join() method is called, until that thread completes and then joins the main program.

```
import java.util.*;

class Tester1
{
static Thread t3;

public void doOperations(){

	Thread t1= new Thread(new NumberPrinter(1));
	Thread t2= new Thread(new NumberPrinter(2));
	t1.start();
	t2.start();
	for (int i = 0; i < 5; i++)
		System.out.println("Main Program " + "\t" + i);
	t3= new Thread(new CharPrinter());
	t3.start();
}

public static void main(String args[])  throws Exception
{
	Tester1 prog= new Tester1();
	prog.doOperations();
	t3.join();
	System.out.println("Main Program Has Ended!");
}

class NumberPrinter implements Runnable{
	private int ID;
	public void run() {
		for (int i = 0; i < 5; i++) {
			System.out.println("ThreadID="+this.ID + "\t" + i);
			try {
				Thread.sleep((long)(0.5 * 1000));
			} catch (InterruptedException e) {}
		}
		System.out.println("Thread "+this.ID+" has finished");

	}

	NumberPrinter(int id){
		ID=id;
	}
}

class CharPrinter implements Runnable{
	private int ID;
	public void run() {
		for (char i = 'a'; i <= 'f'; i++) {
```

```
                    System.out.println("\t CharPrinter \t" + i);
                    try {
                          Thread.sleep((long)(0.5 * 1000));
                    } catch (InterruptedException e) {}
              }
              System.out.println("Char Printing Thread has finished");
        }

}
}
```

Listing 3 Multiple Threads, Different Thread types, and join()

Output:

```
$ java Tester1
ThreadID=1  0
ThreadID=2  0
Main Program      0
Main Program      1
Main Program      2
Main Program      3
Main Program      4
       CharPrinter      a
       CharPrinter      b
ThreadID=1  1
ThreadID=2  1
ThreadID=1  2
ThreadID=2  2
       CharPrinter      c
       CharPrinter      d
ThreadID=2  3
ThreadID=1  3
       CharPrinter      e
ThreadID=1  4
ThreadID=2  4
       CharPrinter      f
Thread 1 has finished
Thread 2 has finished
Char Printing Thread has finished
Main Program Has Ended!
```

There are many strategies for starting the thread. These code snippets have illustrated one way only. Other strategies include adding a local Thread Object to the class that implements the Runnable interface, and calling the start() method directly after the thread is crated. The Thread object

Eg

```
class CharPrinter implements Runnable{
      Thread t;
      public void run() {
            //do semething
      }
      CharPrinter(){
            t=new Thread(this,"AName");
            t.start();
      }

}
```

The object can simply be instantiated and run automatically by simply calling `new CharPrinter()`.

Extend the Thread class

The second way of creating a thread is to

1. Extend the Thread Object
2. Call super("name_of_thread") in the constructor
3. Provide a run method
4. Call start()

The following example illustrates this operation. By using a Thread object, one can pass a unique identifying name to the thread. The thread can be identified later by calling the getName() method on the thread. The thread can then be stopped, killed etc. In the example, the thread priorities have been altered making thread 2 complete generally ahead of thread 1. (1 means higher priority than 9)

```
import java.util.*;
//simple class that extends Thread Object
class Tester4
{

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



        CharPrinter t1=new CharPrinter("Thread1");
       CharPrinter t2=new CharPrinter("Thread2");
       t1.start();
       t2.start();
       t1.setPriority(9);
       t2.setPriority(1);

       for (int i = 0; i < 5; i++)
        System.out.println("Main Program " + "\t" + i);
    System.out.println("The program has executed its last
statement!");

}

}


class CharPrinter extends Thread{
    public CharPrinter(String str) {
        super(str);
    }
    public void run() {
        for (char i = 'a'; i <= 'f'; i++) {
            System.out.println("\t CharPrinter \t" + i);
            try {
                Thread.sleep((long)(0.5 * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("Char Printing Thread named "+getName()+"
has finished");
```

```
    }
}
```

Listing 4 Example class that extends Thread Object

It is relatively simple to extend the Thread Object in order to create a thread. The disadvantage is that in Java, one can only extend one class. Thus if a class already extends another class, then it is impossible to extend the Thread class. If the Thread object is extended, it will also be impossible to extend any other class.

Useful Thread Methods:

To suspend, resume or stop a thread t, the following are called on the thread object respectively:

```
t.suspend();
t.resume();
t.stop();
```

Use t.join() to make the main thread of a program pause at that point until the running thread t has completed before proceeding. The `setPriority()` methods can be used to change the priority of a thread compared to others.

Lab

Lab 1

Create a program that does three things in parallel:

1. Print the numbers one to 1 million on the screen.
2. Save key value pairs in a hashMap. The keys are Integers one to 1000. The values are the corresponding square roots (saved as doubles). After saving them in the hash map, print out the key value pairs
3. Write into a file all the multiples of 4 between 5000 and 20,000 and their corresponding cube root in a file. The file will be formatted as follows;

Number	Square
5000	17.100
5004	17.104
…..	

For each thread, print out when the thread has completed. Also print out when the main thread completes its last statement.

Lab 2.

Write a program that connects to up to 5 URLs and copies the web resources at those locations into local files. Use a thread for each different connection.