CS 213

Sesh Venugopal

Multithreaded Programming I

Prime Numbers Counter

```
package primes;

import java.util.Scanner;

public class Primes {
   static int countPrimes(int n) {
     int count=0, p=2;
     while (p <= n) {
        int d;
        for (d=2; d <= p/2; d++) {
            if ((p % d) == 0) {
               break;
            }
        }
        if (d > p/2) {
            count++;
        }
        p++;
     }
     return count;
   }
   public static void main(String[] args) throws IOException {
     Scanner sc = new Scanner(System.in);
     System.out.print("Enter n: ");
     int n = Integer.parseInt(sc.nextLine());
     System.out.println("Number of primes <= " + n + " : " +
                        countPrimes(n));
   }
}
```

```
> java Primes
Enter n: 10000
Number of primes <= 10000 : 1229
```

What if we wanted the user to be able to watch progress by interrupting the program, and seeing how many primes have been computed up to that point?

Prime Numbers: Watching Progress

- There are two ways to address this:
  - **Program-controlled interrupts**:
    - Have the program break at regular intervals
    - Divide range 2..n into k intervals: k is determined by program
    - After number of primes for an interval have been found, interrupt prime computation and print
  - **User-driven interrupts**:
    - Have the user interrupt the program when needed
    - How to record status at every interrupt so computation can be resumed correctly?
    - Solution: On every interrupt, the program keeps churning out the primes, even as it is interacting with the user. That is, the (time intensive) I/O with user should not stop the program from its main work, of counting primes. Question is: *how to have two independent executions at the same time*:
      - One that interacts with user
      - Another that keeps counting primes

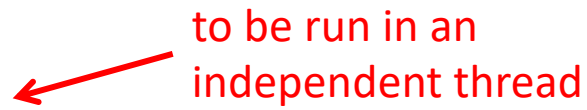
Multithreading I/O with Computation

- The answer is to run two independent *threads* in the program: one that interacts with user, and another that computes number of primes
- Here's a recipe to take the first version of Primes and make it multithreaded:
  - **Step 1**: Extend the **`java.lang.Thread`** class:

```
public class PrimesThread extends Thread {
```

Multithreading I/O with Computation

- Recipe for conversion to multithreading (continued) :
  - **Step 2**: Place the primes counting code in a method called `run` that is specifically defined by the `Thread` class (and is overridden by `PrimesThread`) so it can be executed independently:




```
public void run() {
   count=0,p=2;
   while (p <= n) {
      int d;
      for (d=2; d <= p/2; d++) {
         if ((p % d) == 0) {
          break;
         }
      }
      if (d > p/2) {
         count++;
      }
      p++;
   }
}
```

Multithreading I/O with Computation

- Recipe for conversion to multithreading (continued) :
  - **Step 3**: Since the `run` method is defined not to return values, we need to make `count` and `p` static fields that can be shared by the main method, to report progress on demand
  - **Step 4**: Define a constructor that starts up an independent thread for run:

```
public PrimesThread(int n) {
    this.n = n;
    start();
}
```

The start method is defined by the `Thread` class – calling it does the following:

- Set up the necessary resources to run an independent thread
- Start up the thread to execute the `run` method

Calling run directly (instead of calling start) will not start an independent thread!!!

Multithreading I/O with Computation

- Recipe for conversion to multithreading (continued) :
  - **Step 5**: Change the main method to:
    - Set up an independent thread to count primes
    - On every user interruption, report current number of primes computed

```
Scanner sc = new Scanner(System.in);
System.out.print("Enter n: ");
n = Integer.parseInt(sc.nextLine());

new PrimesThread(n);
```

independent thread to count primes

```
while (true) {
   System.out.print(" ? ");
   String line = sc.nextLine();
   if (line.equals("quit")) {
       break;
   }
   System.out.println("At " + (p-1) +
                      ", number of primes = " + count);
}
System.out.println("At " + (p-1) +
                      ", number of primes = " + count);
```

Multithreading I/O with Computation

- Two threads are running simultaneously

main thread

```
public static void
main(String[] args)
throws IOException {
  Scanner sc =
    new Scanner(System.in);

  System.out.print(
    "Enter integer bound => ");
  n = Integer
      .parseInt(sc.nextLine());

  new PrimesThread(n);

  while (true) {
    ...
  }
  ...
}
```

prime thread

```
public void run() {
  count=0; p=2;
  while (p <= n) {
    int d;
    for (d=2;
         d <= p/2;
         d++) {
      if ((p%d) == 0) {
        break;
      }
    }
    if (d > p/2) {
      count++;
    }
    p++;
  }
}
```

```
> java PrimesThread
Enter n: 100000
 ?
 At 73740, number of primes = 7254
 ?
 At 100000, number of primes = 9592
```

Multithreading I/O with Computation

- Every time the user hits enter, the main thread fetches the current status of count and prints it out
- In the meantime, the prime thread continues with its computation
- If the user types "quit", the prime thread continues independently until it runs through all p's up to to n

Having the prime thread keep doing stuff past the time when the user hits "quit" is pointless!!

As soon as the user hits "quit" the prime thread must be terminated

Prime Numbers Counter: Version 3

- Before we fix this glitch, there is another Java-specific issue we need to deal with: a class may support multithreading by extending **`Thread`**, but what if it already extends some other class?

- The solution is to have the class in question implement the `java.lang.Runnable` interface instead of extending the `Thread` class

- This interface prescribes a single method:

```
void run( )
```

  that must be implemented. The `Thread` class itself implements the `Runnable` interface—we have already seen the `run` method

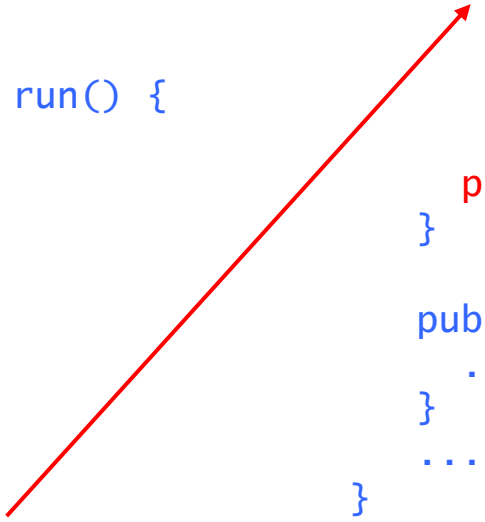
- In general, it is preferable to design a multithreading supporting class to implement the `Runnable` interface even if the class does not extend another, in order to provide for future extensibility

Prime Numbers Counter: Version 3

- Converting from extending `Thread` to implementing `Runnable` is done as follows:

```
public class PrimesThread
    extends Thread {

   int n;
   static int p,count;

   public PrimesThread(int n)
   {
      this.n = n;
      start();
   }

   public void run() {
     ...
   }
   ...
}
```

```
public class PrimesRunnable
    implements Runnable {

   static int count,p; int n;
   static Thread primesThread;

   public PrimesRunnable(int n)
   {
     this.n = n;
     primesThread = new
           Thread(this);

     primesThread.start();
   }

   public void run() {
     ...
   }
   ...
}
```



"target" of the thread

- Since `Runnable` is only an interface, `PrimesRunnable` is not a `Thread`—a new `Thread` must be created explicitly

Prime Numbers Counter: Version 3

- If the prime thread is finished, the main thread should be terminated, i.e. break out of the main while loop

```
public static void main(String[] args)
throws IOException {
  . . .
  new PrimesRunnable(n);
  while (true) {
  while (
    primesThread.getState() != Thread.State.TERMINATED) {

    System.out.print("? ");
    String line = sc.nextLine();
    if (line.equals("quit")) {


       break;
    }
    System.out.println("At " + (p-1) +
                ", number of primes: " + count);

  }
  System.out.println("At " + (p-1) +
              ", number of primes: " + count);
}
```

Prime Numbers Counter: Version 3

- If the main thread is finished, the prime thread should be notified to terminate

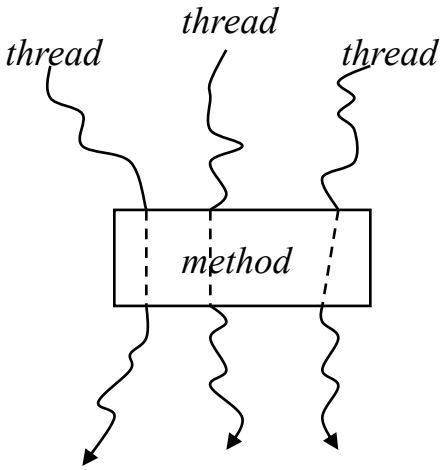
```
public static void main(String[] args)
throws IOException {
  . . .
  new PrimesRunnable(n);

  while (
    primesThread.getState() != Thread.State.TERMINATED) {
    ...
    if (line.equals("quit")) {
       primesThread.interrupt();
       break;
    }
    ...
  }
  ...
}
```

The loop condition checks whether the the thread was interrupted, and if so, stops safely, before it enters an iteration, and not in the middle

```
public void run() {
  while (!Thread.interrupted()
          && p <= n) {
    int d;
    for (d=2; d <= p/2; d++) {
      if ((p%d) == 0) {break;}
    }
    if (d > p/2) {count++;}
    p++;
  }
}
```

Being Executed in a Thread

- When working with multi-threaded programs it is important to see that the code within a method may be executed by any number of threads, even simultaneously (same runnable target for several threads)



- Thus, "currently executing thread" means the thread that is currently executing the statement in question:

```
Thread.currentThread();
```

- So, all methods in **`Thread`** that are static are invoked on the currently executing thread:

```
Thread.interrupted();
```

- Or:

```
Thread.sleep(1000);
```

- The name of the thread that is currently executing may be obtained by using the construct:

```
Thread.currentThread().getName()
```

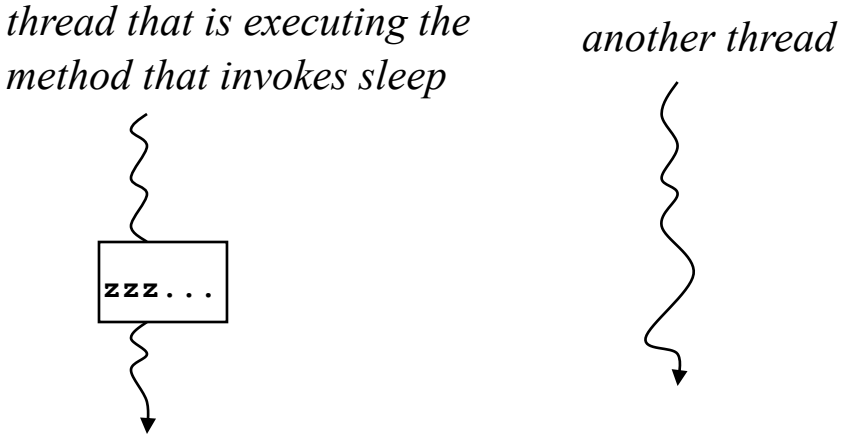
Putting a Thread To Sleep

- A thread may be put to sleep for a fixed amount of time by invoking the static sleep method:

```
public static void sleep(long millis)
throws InterruptedException
```

This causes the *currently executing* thread to sleep for the given milliseconds:

- It remains in an active state, but is not scheduled to run until the sleep period has expired
- It can be interrupted from its sleep by another thread



- Another version of **`sleep`** allows the specification of an additional nanoseconds longer for which the thread sleeps:

```
public static void sleep(long millis, int nanos)
throws InterruptedException
```

value 0..999999

Multiple Threads Through Same Code

```
public class Interleave implements Runnable {

   public Interleave(String name) {
       new Thread(this, name).start();
   }

   public void run() {
      for (int i=0; i < 4; i++) {
          System.out.println(
              Thread.currentThread().getName());
          try {
             Thread.sleep((int)Math.random()*1000);
          } catch (InterruptedException e) { }
      }
   }

   public static void main(String[] args) {
      new Interleave("Java");
      new Interleave("Sumatra");
   }
}
```

a `Thread` *constructor that accepts runnable target as well as name for thread*

Run 1

```
Java
Sumatra
Java
Java
Java
Sumatra
Sumatra
Sumatra
```

Run 2

```
Java
Sumatra
Java
Suamtra
Java
Sumatra
Java
Sumatra
```

Each thread executes the body of the `for` loop in `run` four times, in random interleaved sequence – the sequence may be different for different runs

Why Threads

- A thread runs asynchronously, independent of the thread that created it

- A Java application or applet itself runs as a thread, and can spin off as many other threads as needed

- A collection of asynchronously running threads may communicate with each other either indirectly via a buffer, or directly by invoking methods on each other

- Asynchronous computing allows several tasks to be performed in parallel, resulting in:
  - **improved execution time** for the application as a whole
  - **improved turn-around time** seen by the user - for instance a consumer thread displays data on the fly as it comes from the producer, instead of blocking until all data is available

- Asynchronous computing places more onus on the programmer to insure that the program:
  - **avoids race conditions** e.g. two threads trying to update a variable at the exact same time
  - **maintains consistency of data** e.g. two transactions both deposit money into an account, but because of an unlucky interleaving of executions, only one of the deposits is recorded.