CS 213 Spring 2016

Lecture 25: April 19
Multithreaded Programming I

Prime Numbers Counter

```
package primeui;

import java.io.*;

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 {
     BufferedReader br = new BufferedReader(
                            new InputStreamReader(System.in));
     System.out.print("Enter integer bound => ");
     int n = Integer.parseInt(br.readLine());
     System.out.println("Number of primes <= " + n + " : " +
                         countPrimes(n));
   }
}
```

```
> java Primes
Enter integer bound => 10
Number of primes <= 10 : 4
```

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 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 PrimeThread extends Thread {
```

  - **Step 2**: Place the primes counting code in a method called **`run`** that is specifically defined by the Thread class (and will be overridden by PrimeThread) 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++;
   }
}
```

to be run in an independent thread

  - **Step 3**: Since the run method is defined not to accept any parameters, we need to make **`n`** a (static) field. Also **`count`** and **`p`** will be fields that can be accessed by the main method as well, to report progress on demand

Multithreading I/O with Computation

- Recipe for conversion to multithreading (continued) :
  - **Step 4**: Define a constructor that starts up an independent thread for run:

```
public PrimeThread() { 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

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

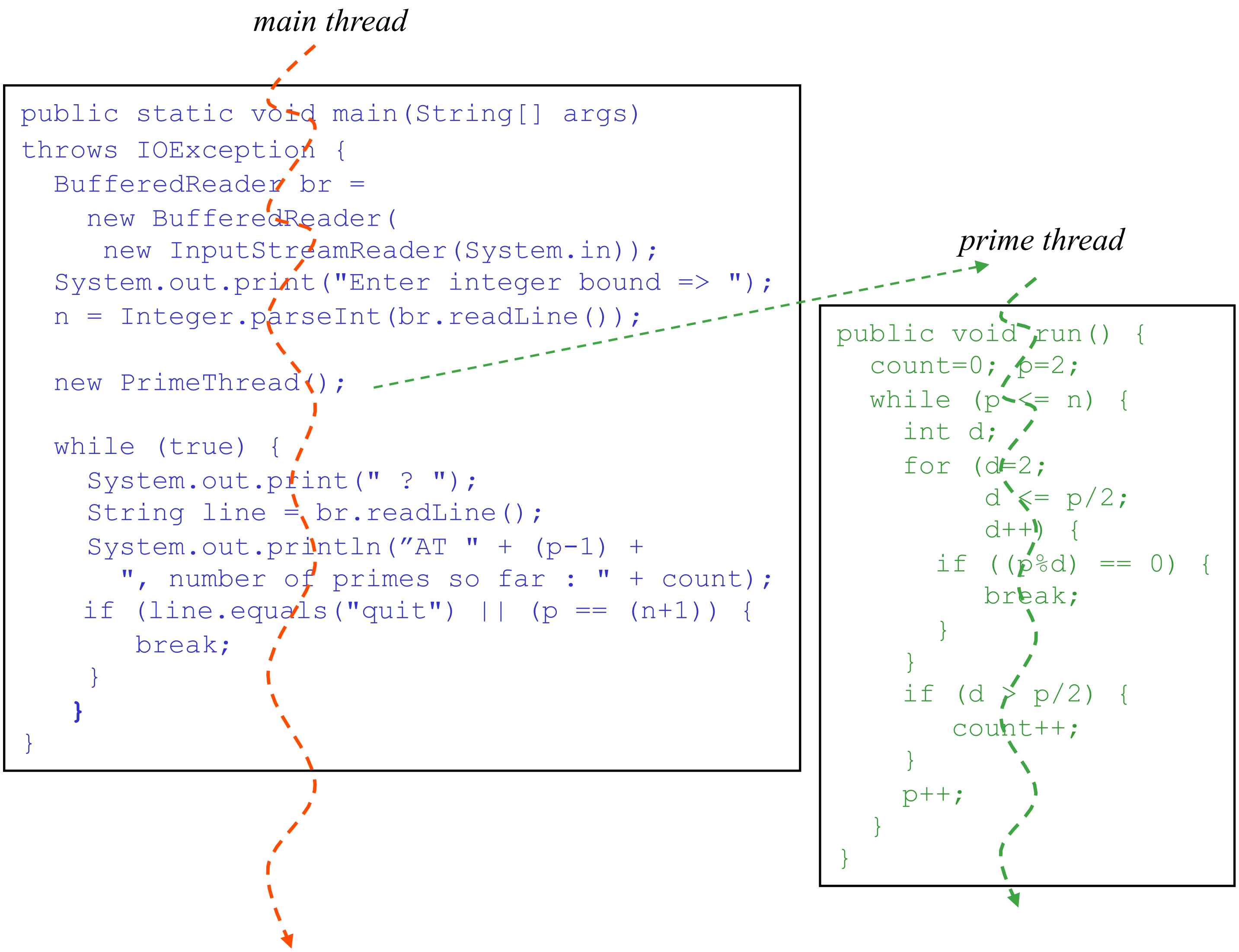
Multithreading I/O with Computation

- 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

```
BufferedReader br = new BufferedReader(
                       new InputStreamReader(System.in));
System.out.print("Enter integer bound => ");
n = Integer.parseInt(br.readLine());

new PrimeThread();

while (true) {
   System.out.print(" ? ");
   String line = br.readLine();
   System.out.println(”AT " + (p-1) +
                      ", number of primes so far : " + count);
   if (line.equals("quit") || (p == (n+1)) {
       break;
   }
}
```

Multithreading I/O with Computation

- Two threads are running simultaneously



```
public static void main(String[] args)
throws IOException {
  BufferedReader br =
    new BufferedReader(
     new InputStreamReader(System.in));
  System.out.print("Enter integer bound => ");
  n = Integer.parseInt(br.readLine());

  new PrimeThread();

  while (true) {
    System.out.print(" ? ");
    String line = br.readLine();
    System.out.println(”AT " + (p-1) +
      ", number of primes so far : " + count);
    if (line.equals("quit") || (p == (n+1)) {
       break;
    }
   }
}
```

```
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 PrimeThread
Enter integer bound => 100000
 ?
 AT 73740, number of primes so far : 7254
 ?
 AT 100000, number of primes so far : 9592
```

hit Enter

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 meanwhile, 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

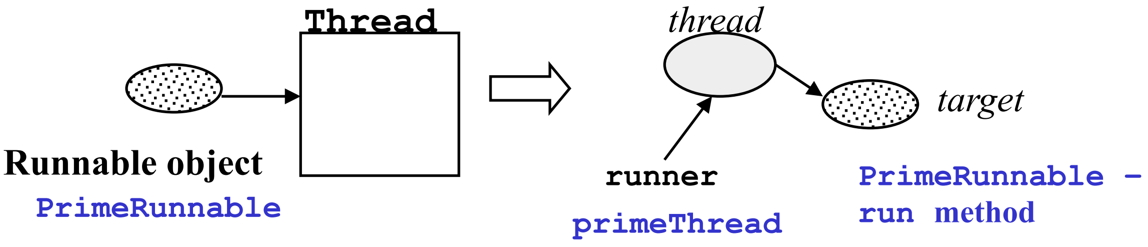
- Converting from extending **Thread** to implementing **Runnable** is done as follows:

```
public class PrimeThread
    extends Thread {

   static int n,p,count;

   public PrimeThread() {
     start();
   }

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

```
public class PrimeRunnable
    implements Runnable {

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


   public PrimeRunnable() {
     primeThread = new Thread(this);
     primeThread.start();
   }

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

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

- The **Thread** constructor accepts a **Runnable** object and creates a **Thread** object with this **Runnable** object as the *target:*



Prime Numbers Counter: Version 3

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

```
public static void main(String[] args) throws IOException {
  . . .
  new PrimeRunnable();
  while (true) {
    if (primeThread.getState() == Thread.State.TERMINATED) {
       System.out.println(“Number of primes <= “ +(p-1) + “:”+count);
       break;
    }
    System.out.print("? ");
    String line = br.readLine();
    if (line.equals("quit")) {
       primeThread.interrupt();  // interrupt prime thread
       System.out.println(”AT" + (p-1) +
                ", number of primes so far : " + count
           break;
    }
    System.out.println(”AT" + (p-1) +
                ", number of primes so far : " + count);
  }
}
```

- The state of a thread can be examined – if the state is TERMINATED, that means the thread has finished executing its target code

If the user hits quit, the prime thread is <u>interrupted</u>.

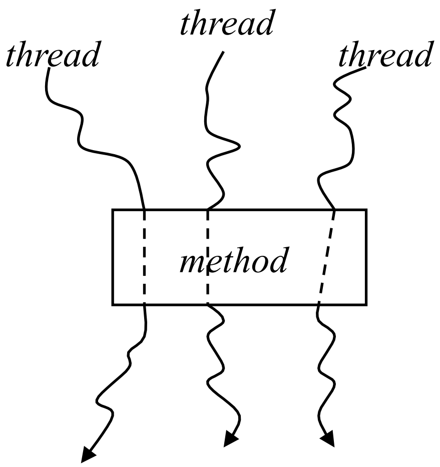
The loop condition checks whether the the thread was interrupted, and if so, stops safely, before it enters an iteration, and not in the middle

primeThread

```
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, the phrase "currently executing thread" means the thread that is currently executing the statement in question:

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

- Thus, also, the methods in **`Thread`** that are static are invoked on the currently executing thread

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

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

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

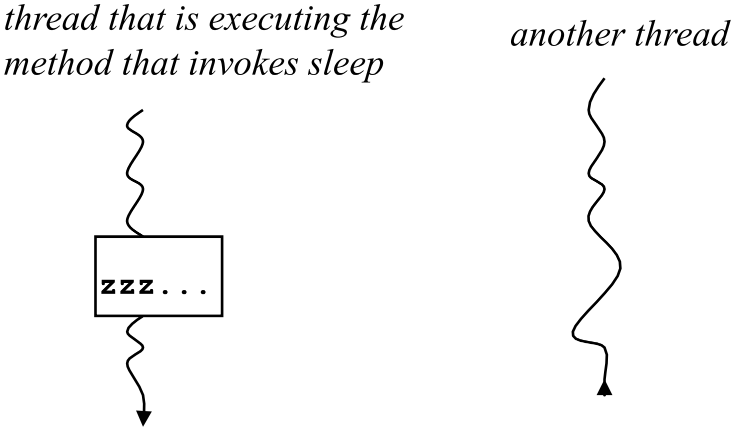
Putting a Thread To Sleep

- 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



thread that is executing the method that invokes sleep

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
```

The value of **nanos** can be between 0 and 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*

- Sample output(s):

```
Java          Java
Sumatra                Sumatra
Java          Java
Java          Sumatra
Java          Java
Sumatra                Sumatra
Sumatra                Java
Sumatra                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 the consumer thread displays data on the fly as it comes from the server, 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 withdraw money from an account, but the second does not see the withdrawal made by the first