Java Threading

Threads are objects, too

Two Ways to Thread

- Two approaches
- java.lang.Thread
    - https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.html
- java.lang.Runnable
    - https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Runnable.html

Thread Example

```
public class HelloThread extends Thread {

        private int val;

        public HelloThread(int val) {
                this.val = val;
        }

        public void run() {
                System.out.println("Hello from thread " + this.getName() +
                                ". My val is " + val);
        }

        public static void main(String[] args) {
                for (int i = 0; i < 5; i++) {
                        HelloThread hello = new HelloThread(i);
                        hello.start();
                }
        }
}
```

Thread Example Output

- The threads are started sequentially
- Execution is inter-leaved

```
david.apostal@MacBook-Pro tmp % java HelloThread
Hello from thread Thread-1. My val is 1
Hello from thread Thread-0. My val is 0
Hello from thread Thread-4. My val is 4
Hello from thread Thread-3. My val is 3
Hello from thread Thread-2. My val is 2
david.apostal@MacBook-Pro tmp %
```

- The run() method has one instruction. How many steps?
- Operating system only allows one thread to access stdout at a time

Why not call run()?

```
public static void main(String[] args) {
    for (int i = 0; i < 5; i++) {
        HelloThread hello = new HelloThread(i);
        hello.run();
    }
}
```

- DO NOT Call Thread.run() — No concurrent execution
- Thread.start() does more than JUST call Thread.run()

```
Hello from thread Thread-0. My value is 0
Hello from thread Thread-1. My value is 1
Hello from thread Thread-2. My value is 2
Hello from thread Thread-3. My value is 3
Hello from thread Thread-4. My value is 4
```

Runnable Example

```
public class HelloRunnable implements Runnable {

    private int val;

    public HelloRunnable(int val) {
        this.val = val;
    }

    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println("Hello from thread " + name +
                ". My val is " + val);
    }

    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            Runnable runnable = new HelloRunnable(i);
            Thread thread = new Thread(runnable);
            thread.start();
        }
    }
}
```

Thread vs Runnable

- Extend Thread class
  - Simpler.
- Implement Runnable
  - More flexible.
- General rule: Program to Interfaces

Multi-threaded Server

```
while (true) {
    accept a connection
    create thread for client
}
```

Advantages

- Less work for main thread
- Multiple requests handled simultaneously
- Better responsiveness than single-threaded server

Caution

- Make sure tasks are thread-safe
- Does not limit the number of threads created

Multi-threaded Server

```
public class MultiEchoServer implements Runnable {
    Socket clientSocket;

    public MultiEchoServer(Socket client) {
        clientSocket = client;
    }

    public void run() {
        try {
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                    new InputStreamReader(clientSocket.getInputStream()));

            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                System.out.println(inputLine);
                out.println(inputLine);
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }

    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: java EchoServer <port number>");
            System.exit(1);
        }

        int portNumber = Integer.parseInt(args[0]);

        try {
            ServerSocket serverSocket = new ServerSocket(Integer.parseInt(args[0]));
            System.out.println("The server is listening at: " +
                    serverSocket.getInetAddress() + " on port " +
                    serverSocket.getLocalPort());

            while (true) {
                Socket clientSocket = serverSocket.accept();
                MultiEchoServer mes = new MultiEchoServer(clientSocket);
                new Thread(mes).start();
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Multi-threaded Server (1/3)

```
public class MultiEchoServer implements Runnable {
   Socket clientSocket;

   public MultiEchoServer(Socket client) {
      clientSocket = client;
   }
```

Multi-threaded Server (2/3)

```
public void run() {
   try {
      PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
      BufferedReader in = new BufferedReader(
            new InputStreamReader(clientSocket.getInputStream()));
      String inputLine;
      while ((inputLine = in.readLine()) != null) {
         System.out.println(inputLine);
         out.println(inputLine);
      }
   } catch (IOException e) {
      System.out.println(e.getMessage());
   }
}
```

Multi-threaded Server (3/3)

```
public static void main(String[] args) {
   if (args.length != 1) {
      System.err.println("Usage: java EchoServer <port number>");
      System.exit(1);
   }

   int portNumber = Integer.parseInt(args[0]);

   try {
      ServerSocket serverSocket = new ServerSocket(portNumber);
      System.out.println("The server is listening at: " +
            serverSocket.getInetAddress() + " on port " +
            serverSocket.getLocalPort());

      while (true) {
         Socket clientSocket = serverSocket.accept();
         MultiEchoServer mes = new MultiEchoServer(clientSocket);
         new Thread(mes).start();
      }
   } catch (IOException e) {
      System.out.println(e.getMessage());
   }
  }
}
```

Online Reference

- http://docs.oracle.com/javase/tutorial/essential/concurrency/

Pausing a Thread

```
public class SleepMessages {
    public static void main(String args[]) throws InterruptedException {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };

        for (int i = 0; i < importantInfo.length; i++) {
            //Pause for 4 seconds
            Thread.sleep(4000);
            //Print a message
            System.out.println(importantInfo[i]);
        }
    }
}
```

- The `sleep` method pauses a thread for (roughly -- OS dependent) that many milliseconds

- If another thread interrupts a sleeping thread, the sleep method will throw an `InterruptedException`

Interrupts

- An interrupt is a signal that a thread should stop what it is doing and so that other work can be done.
    - An event has occurred in another thread.

- Programmers can anticipate certain interrupts and code how a thread should respond.

Handling an Interrupt

```
public void run() {
    for (int i = 0; i < importantInfo.length; i++) {
        //Pause for 4 seconds
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            //We've been interrupted: no more messages.
            return;
        }
        //Print a message
        System.out.println(importantInfo[i]);
    }
}
```

- Will print a message every four seconds until interrupted or there are no more messages

Handling an Interrupt 2

```
public void run() {
    for (int i = 0; i < inputs.length; i++) {
        heavyCrunch(inputs[i]);
        if (Thread.interrupted()) {
            //We've been interrupted: no more crunching.
            return;
        }
    }
}
```

- What if your methods don't throw `InterruptedException`?
- `Thread.interrupted()` returns `true` if the current thread has been interrupted. A subsequent call to `Thread.interrupted()` will return `false` unless the thread was interrupted again.
- It may be better to throw a new `InterruptedException` instead of returning.

Interrupting a Thread

```
public class HelloThread extends Thread {

        …

        public void run() {
                if (Thread.interrupted()) {
                        System.out.println(“interrupted”);
                        return;
                }
                System.out.println("Hello from thread " + this.getName() +
                                ". My val is " + val);
        }

        public static void main(String[] args) {
                for (int i = 0; i < 5; i++) {
                        HelloThread hello = new HelloThread(i);
                        hello.start();

                        if (i % 2 == 0) {
                                hello.interrupt();
                        }
                }
        }
}
```

Output

```
interrupted
interrupted
interrupted
Hello from thread Thread-3. My value is 3
Hello from thread Thread-1. My value is 1
```

Similar but Different

- public static boolean interrupted();
  - Clears the *interrupt status flag*
    - Must be called with class name: *Thread.interrupted();*
- public boolean isInterrupted();
  - Does not clear the *interrupt status flag*
  - Must be called with an object ref: *t.isInterrupted();*

“Joining” a Thread

- `t.join();` will wait for the thread `t` to complete
- `t.join(millis);` will wait at most `millis` ms (again roughly) for `t` to complete
- if interrupted, will throw an `InterruptedException`

HelloThread with join

```
public class HelloThread extends Thread {

        private int val;

        public HelloThread(int val) {
                this.val = val;
        }

        public void run() {
                System.out.println("Hello from thread " + this.getName() +
                                ". My val is " + val);
        }

        public static void main(String[] args) {
                for (int i = 0; i < 5; i++) {
                        HelloThread hello = new HelloThread(i);
                        hello.start();
                        try {
                            hello.join();
                        } catch (InterruptedException ie) {
                            System.out.println(“interrupted”);
                        }
                }
        }
}
```

Synchronization

- Threads communicate by sharing access to fields and methods of objects they reference
  - More efficient that network communication
- This can lead to errors that are difficult to resolve
- Two kinds of errors:
  - Thread interference
  - Memory consistency
- Synchronization prevents these errors

Thread Interference

```
class Counter {
    private int c = 0;

    public void increment() { c++; }
    public void decrement() { c--; }

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

the c++ statement:

retrieve c
increment c
store value

the c-- statement:

retrieve c
decrement c
store value

Thread Interference 2

What if two threads use the same `Counter`?

Thread A calls increment(), Thread B calls decrement()

1. Thread A: retrieve c (A's c == 0)
2. Thread B: retrieve c (B's c == 0)
3. Thread A: increment c (A's c = 1)
4. Thread B: decrement c (B's c = -1)
5. Thread A: store c (stores 1)
6. Thread B: store c (stores -1)

Thread Interference 3

- Take away
  - Performing operations on the same memory with multiple threads at the same time can cause errors that are difficult to find
  - Use synchronization techniques to prevent *data races*. Ensure that your code is *deterministic*.

Memory Consistency

Memory consistency errors occur when threads have inconsistent view of data

Thread A and B share a reference to counter:

```
int counter = 0;
```

Thread A increments counter:

```
counter++;
```

Also, Thread B prints counter:

```
System.out.println(counter);
```

Thread B may print 0 or 1. We cannot tell which instruction happens before the other.

What if the two statements (increment, then print) were in the same thread?

The value printed would be 1. Increment happens before print.

Happens-Before

Happens-Before relationships guarantee some statements happen before others

Example: `Thread.start()` and `Thread.join()`

Synchronization ensures a happens-before relationship

More Reading:

See Chapter 17 of the Java Language Specification