

Concurrency in Java

Sergio Luján Mora



Departamento de Lenguajes y
Sistemas Informáticos



Universitat d'Alacant
Universidad de Alicante

Contents

- Introduction
- Thread class
- Sleep
- Join
- Daemon threads

Introduction

- In concurrent programming, there are two basic units of execution: *processes* and *threads*
- In the Java programming language, concurrent programming is mostly concerned with **threads**

Introduction

- A **process** has a self-contained execution environment
- A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space

Introduction

- Threads are sometimes called *lightweight processes*
- Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process
- Threads exist within a process: every process has at least one
- Threads share the process's resources, including memory and open files
 - This makes for efficient, but potentially problematic, communication

Thread class

- An application that creates an instance of `Thread` must provide the code that will run in that thread
- There are two ways to do this:

Thread class

- Provide a Runnable object
 - The Runnable interface defines a single method, run, meant to contain the code executed in the thread
 - The Runnable object is passed to the Thread constructor, as in the HelloRunnable example:

```
public class HelloRunnable implements Runnable {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new Thread(new HelloRunnable())).start();  
    }  
  
}
```

Thread class

- Subclass Thread
 - The Thread class itself implements Runnable, though its run method does nothing
 - An application can subclass Thread, providing its own implementation of run, as in the HelloThread example:

```
public class HelloThread extends Thread {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
  
}
```

Thread class

- The first idiom, which employs a `Runnable` object, is more general, because the `Runnable` object can subclass a class other than `Thread`
- The second idiom is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of `Thread`

```
public class CountThreads extends Thread {

    private int countDown = 5;
    private static int threadCount = 0;
    private int threadNumber = ++threadCount;

    public CountThreads() {
        System.out.println("Making " + threadNumber);
    }

    public void run() {
        while(true) {
            System.out.println("Thread " +
                threadNumber + "(" + countDown + ")");
            if(--countDown == 0) return;
        }
    }

    public static void main(String[] args) {
        for(int i = 0; i < 5; i++)
            new CountThreads().start();
        System.out.println("All Threads Started");
    }
}
```

Sleep

- `Thread.sleep()` causes the current thread to suspend execution for a specified period
- This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system
- The `sleep` method can also be used for pacing, as shown in the example that follows, and waiting for another thread with duties that are understood to have time requirements

Sleep

```
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]);
        }
    }
}
```

Join

- The `join()` method allows one thread to wait for the completion of another
- If `t` is a `Thread` object whose thread is currently executing, `t.join();` causes the current thread to pause execution until `t`'s thread terminates

```
public class SimpleThreads {

    //Display a message, preceded by the name of the current thread
    static void threadMessage(String message) {
        String threadName = Thread.currentThread().getName();
        System.out.format("%s: %s\n", threadName, message);
    }

    private static class MessageLoop implements Runnable {
        public void run() {
            String importantInfo[] = {
                "Mares eat oats",
                "Does eat oats",
                "Little lambs eat ivy",
                "A kid will eat ivy too"
            };
            try {
                for (int i = 0; i < importantInfo.length; i++) {
                    //Pause for 4 seconds
                    Thread.sleep(4000);
                    //Print a message
                    threadMessage(importantInfo[i]);
                }
            } catch (InterruptedException e) {
                threadMessage("I wasn't done!");
            }
        }
    }
}
```

```

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

    //Delay, in milliseconds before we interrupt MessageLoop
    //thread (default one hour).
    long patience = 1000 * 60 * 60;

    //If command line argument present, gives patience in seconds.
    if (args.length > 0) {
        try {
            patience = Long.parseLong(args[0]) * 1000;
        } catch (NumberFormatException e) {
            System.err.println("Argument must be an integer.");
            System.exit(1);
        }
    }

    threadMessage("Starting MessageLoop thread");
    long startTime = System.currentTimeMillis();
    Thread t = new Thread(new MessageLoop());
    t.start();

```

```

    threadMessage("Waiting for MessageLoop thread to finish");
    //loop until MessageLoop thread exits
    while (t.isAlive()) {
        threadMessage("Still waiting...");
        //Wait maximum of 1 second for MessageLoop thread to
        //finish.
        t.join(1000);
        if (((System.currentTimeMillis() - startTime) > patience) &&
            t.isAlive()) {
            threadMessage("Tired of waiting!");
            t.interrupt();
            //Shouldn't be long now -- wait indefinitely
            t.join();
        }
    }
    threadMessage("Finally!");
}

```

Daemon threads

- A daemon thread is a thread that runs for the benefit of other threads
- Unlike conventional user threads, daemon threads do not prevent a program from terminating
- We designate a thread as a daemon with the method call:
`setDaemon(true);`
- A false argument means that the thread is not a daemon thread
- A program can include a mixture of daemon threads and nondaemon threads
- When only daemon threads remain in a program, the program exits

```
class Daemon extends Thread {
    private static final int SIZE = 10;
    private Thread[] t = new Thread[SIZE];

    public Daemon() {
        setDaemon(false);
        start();
    }

    public void run() {
        for(int i = 0; i < SIZE; i++)
            t[i] = new DaemonSpawn(i);
        for(int i = 0; i < SIZE; i++)
            System.out.println(
                "t[" + i + "].isDaemon() = "
                + t[i].isDaemon());
        while(true)
            yield();
    }
}
```

```
class DaemonSpawn extends Thread {
public DaemonSpawn(int i) {
    System.out.println(
        "DaemonSpawn " + i + " started");
    start();
}

public void run() {
    while(true)
        yield();
}
}

public class Daemons {
public static void main(String[] args)
throws IOException {
    Thread d = new Daemon();
    System.out.println(
        "d.isDaemon() = " + d.isDaemon());
    // Allow the daemon threads to
    // finish their startup processes:
    System.out.println("Press any key");
    System.in.read();
}
}
```