CSE-2102
Object Oriented Programming

Dr. Muhammad Ibrahim
Assistant Professor
Dept. of Computer Science and Engineering
University of Dhaka

Topics

Multithreading

- Motivation
- Main thread

Two ways to create a thread

Multiple threads

Passing parameters to threads

Returning results from threads

Multithreading

- In a normal program, there is a single sequence of execution of instructions.
  - From top to bottom, unless specified by the code such as loop, break/continue, try-catch etc.
- Oftentimes we feel that we could execute different parts of a single program simultaneously.
  - Benefit? Obvious, the total execution time of the program will decrease.

```
int feet=12, inch, kilometer = 11, meter;
inch = feet * 12;
meter = kilometer * 1000; //this line can be run in parallel with the previous
line
System.out.println("Feet to inch: " + inch);
System.out.println("Kilometer to meter: " + meter); //this line can be run in
//parallel with the previous line
```

Multithreading

- Multithreading is a mechanism to execute different parts of the same program simultaneously.
- Each part of such a program is called a thread, and each thread defines a separate path of execution.
    - A text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.
- To execute different parts of a program simultaneously, it is better for the host machine to have multiple CPU/cores. (Very commonplace nowadays.)
- In a single core system, multithreading may reduce idle CPU time by switching between threads (although more than one thread does not actually run simultaneously).
    - A thread may take user input which is slow, while another thread can proceed its own execution (of different parts).
- So what is the difference between multitasking and multithreading?
    - Multitasking is about executing different programs simultaneously, whereas multithreading is about executing different parts of the same program simultaneously.
    - A program incurs some overhead to be executed, also interprocess communication is computationally costly, so oftentime programmers prefer multithreading over multitasking.
    - Both multitasking and multithreading can be executed simultaneously, as done in modern day systems.

Importance of Multithreading

Maximum use of the processing power available in the system.

One important way multithreading achieves this is by keeping idle time to a minimum.

This is especially important for the interactive, networked environment in which Java operates because idle time is common.

For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU. And, of course, user input is much slower than the computer. In a single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one — even though most of the time the program is idle, waiting for input. Multithreading helps you reduce this idle time because another thread can run when one is waiting.

Multithreading

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind.

So as a programmer you may not use multithreading very often, but the built-in classes you use may often use multithreading.

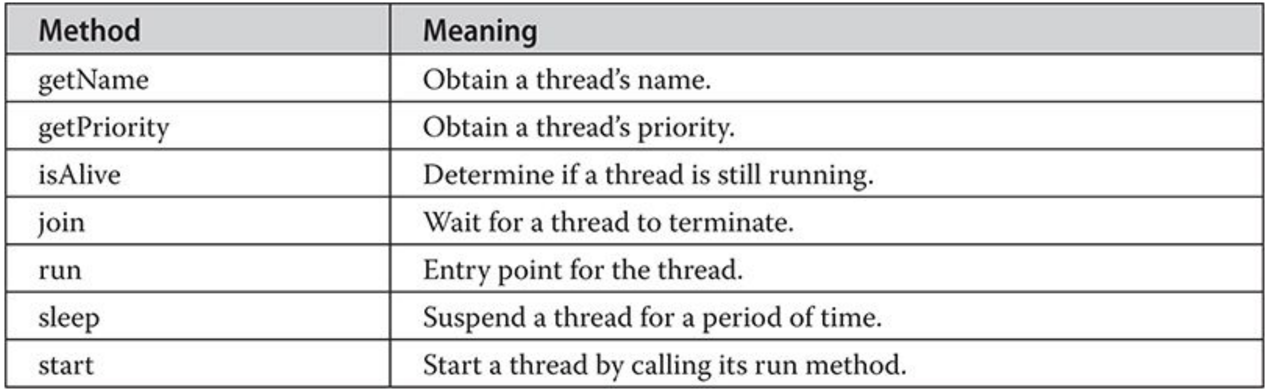
Example

States of Threads

- A thread can be running.
- It can be ready to run as soon as it gets CPU time.
- A running thread can be suspended, which temporarily halts its activity.
- A suspended thread can then be resumed, allowing it to pick up where it left off.
- A thread can be blocked when waiting for a resource.
- At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed

The Thread Class and Runnable Interface

- By now you know that a thread is a piece of code of a program.
- Java's multithreading system is built upon
  - the `Thread` class, its methods,
  - and its companion interface, `Runnable`.
- Any Java program has one thread by default, which is the main thread.
- To create a new thread, a program either extends `Thread` class or implements `Runnable` interface.
- The `Thread` class defines several methods:

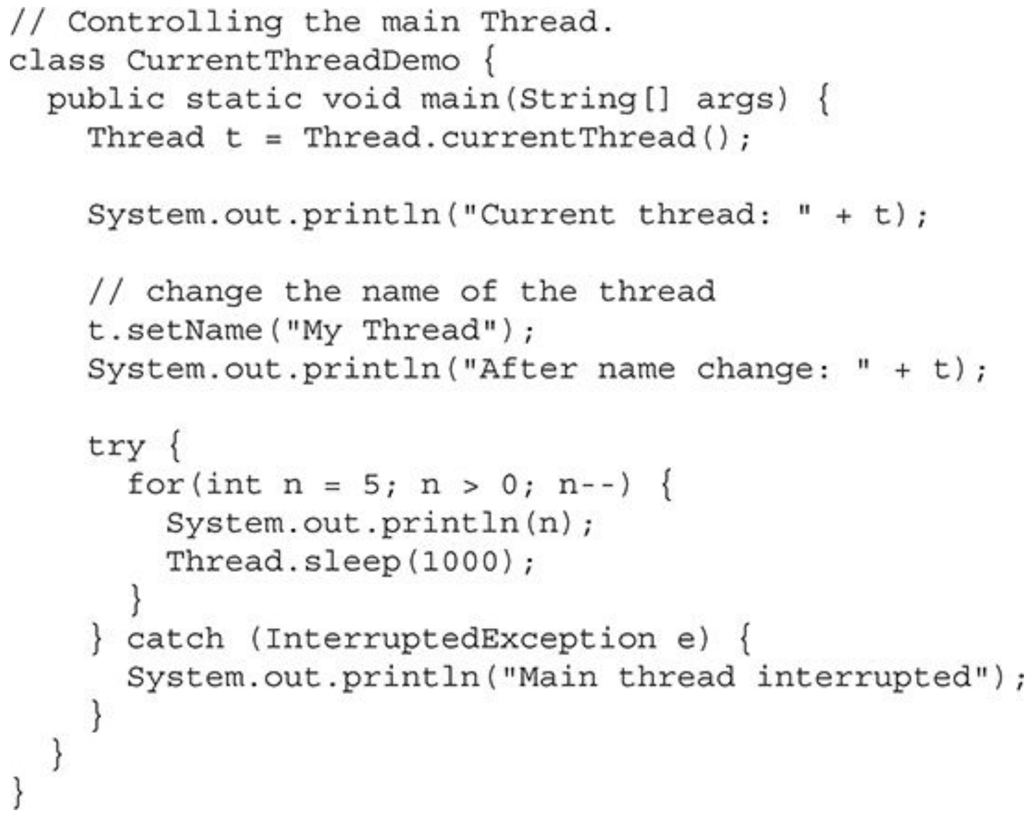


Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

The Main Thread

- As mentioned earlier, the main thread is automatically created.
- Other threads are spawned (explicitly by the programmer) from the main thread.
- Oftentimes the main thread is the last thread to terminate as it performs some shutdown operations such as closing files.
- Main thread can be controlled using a thread object:
    - `Thread t = Thread.currentThread();`
    - `Thread` is a member of `java.lang package` which is automatically imported in any Java program.
    - `currentThread` is a public static member of Thread class.
- Lets see some more methods of `Thread` class: `sleep(), setName(), getName()`
    - `static void sleep(long milliseconds) throws InterruptedException`
    - `final void setName(String threadName) //cannot be overridden due to being “final”`
    - `final String getName()`

Example



```
// Controlling the main Thread.
class CurrentThreadDemo {
  public static void main(String[] args) {
    Thread t = Thread.currentThread();

    System.out.println("Current thread: " + t);

    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);

    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }
  }
}
```

This displays, in order: the *name* of the thread, its *priority*, and the name of its *group*.

Output:

```
Current thread:
Thread[main,5,main]
After name change:
Thread[My Thread,5,main]
5
4
3
2
1
```

Creating a New Thread

Two ways to create a new thread: by

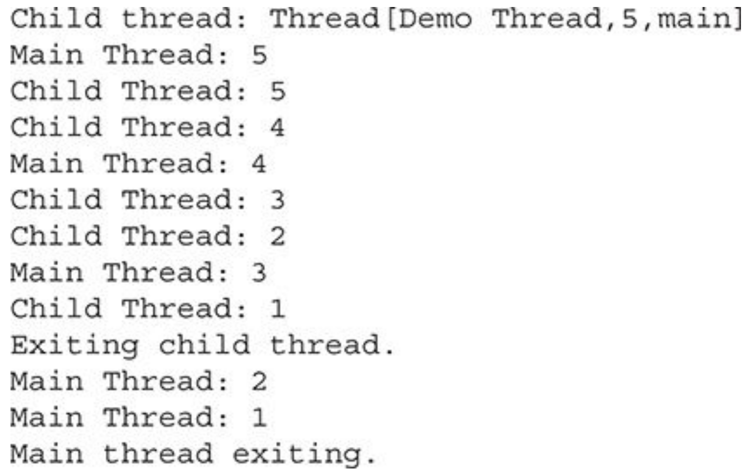
a. implementing `Runnable` interface and
b. extending `Thread` class

1st Way: Implementing the Runnable Interface

1. You can construct a thread on any object that implements `Runnable`.
2. Create a class that implements the `Runnable` interface.
3. Define the `run()` method declared in `Runnable` interface.
    a. In this method, put the code to be executed for this thread.
    b. So `run()` is the entry point of a concurrent thread.
        i. This thread will terminate when `run()` will return.
4. In the constructor of your class:
    a. create an object of `Thread` class.
    b. Call the `start()` method of `Thread` class - the `start()` method calls the `run()` method.
5. In the place where you want to run your thread, simply create an object of your class just written in accordance with the above.
    a. The constructor will call `start()` which begins the execution of the thread (i.e., the `run()` method).

```
class new_thread implements Runnable {
    Thread t;
    new_thread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread, i.e., runs run() method.
    }
    // This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);//may throw a checked exception.
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
    System.out.println("Exiting child thread.");
    }
}
```

```
class thread_demo {
    public static void main(String args[ ] ) {
        new new_thread(); // create a new thread
        try {
            for(int i = 5; i > 0; i--) {
            System.out.println("Main Thread: " + i);
            Thread.sleep(1000);
        }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```



```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

2nd Way: Extending Thread Class

```
class NewThread extends Thread {
     NewThread() {
     // Create a new, second thread
          super("Demo Thread");
          System.out.println("Child thread: " + this);
          start(); // Start the thread
     }
     // This is the entry point for the second thread.
     public void run() {
          try {
               for(int i = 5; i > 0; i--) {
               System.out.println("Child Thread: " + i);
               Thread.sleep(500);
          }
          } catch (InterruptedException e) {
               System.out.println("Child interrupted.");
          }
          System.out.println("Exiting child thread.");
     }
}
```

```
class ExtendThread {
     public static void main(String args[]) {
          new NewThread(); // create a new thread
          try {
               for(int i = 5; i > 0; i--) {
               System.out.println("Main Thread: " + i);
               Thread.sleep(1000);
          }
          } catch (InterruptedException e) {
               System.out.println("Main thread interrupted.");
          }
          System.out.println("Main thread exiting.");
     }
}
```

Why Two Ways?

- Recall that in Java a class cannot inherit more than one superclass.
    - So if a class needs to inherit some other class than `Thread`, it must implement `Runnable` interface to create a thread.
- So why does the method of inheriting `Thread` class exist?
    - `Thread` class defines some other useful methods (such as isAlive and join) that are inherited by a subclass.
        - Of these methods, `run()` is mandatory to implement.
- In a nutshell, it is up to the programmer to decide which way to use.
- We'll be using both the methods.

Creating Multiple Threads

- Similar to the previous code, let’s see an example….

```java
// Create multiple threads.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
  }

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + "Interrupted");
    }
    System.out.println(name + " exiting.");
  }
}
```

,

```
class MultiThreadDemo {
  public static void main(String[] args) {
    NewThread nt1 = new NewThread("One");
    NewThread nt2 = new NewThread("Two");
    NewThread nt3 = new NewThread("Three");

    // Start the threads.
    nt1.t.start();
    nt2.t.start();
    nt3.t.start();

    try {
      // wait for other threads to end
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    System.out.println("Main thread exiting.");
  }
}
```

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

isAlive(), join() and getState() Methods

- As mentioned earlier, oftentimes we want the main thread to finish only after being certain that all other threads are finished.
- So is there any way to be certain about that?
- `final boolean isAlive()` is a method defined in `Thread` class.
  - Returns true if the thread which calls it is still running, false otherwise
- `final void join()` is a method that waits until the thread joins the current thread.
- `Thread.State getState()` method returns the current state of a thread.
  - `Thread.State` is an enumeration which can take one of six possible values.

```
// Suspending and resuming a thread the modern way.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;
  boolean suspendFlag;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    suspendFlag = false;
  }

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 15; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(200);
        synchronized(this) {
          while(suspendFlag) {
            wait();
          }
        }
      }
    } catch (InterruptedException e) {
      System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
  }

  synchronized void mysuspend() {
    suspendFlag = true;
  }

  synchronized void myresume() {
    suspendFlag = false;
    notify();
  }
}

class SuspendResume {
  public static void main(String[] args) {
    NewThread ob1 = new NewThread("One");
    NewThread ob2 = new NewThread("Two");

    ob1.t.start(); // Start the thread
    ob2.t.start(); // Start the thread

    try {
      Thread.sleep(1000);
      ob1.mysuspend();
      System.out.println("Suspending thread One");
      Thread.sleep(1000);
      ob1.myresume();
      System.out.println("Resuming thread One");
      ob2.mysuspend();
      System.out.println("Suspending thread Two");
      Thread.sleep(1000);
```

Passing Parameter to Thread

- We've seen that `start()` method is the starting point for us to begin execution of a thread.
  - We can call `run()` method directly, but that doesn't create a new thread.
  - Note: we can call `start`() method outside our thread class only if our class extends `Thread` class - that is, if we implement Runnable interface, `start`() method cannot be called outside our thread class.
- Common practice to pass parameter to a thread is to use parameterized constructors.
- Let's see an example of displaying the contents of an array….

Returning Result from Thread

Two common ways to return results:

1. Pass reference variable as parameter
2. Write a separate method for returning result

Let’s see examples…

Final Word on Multithreading

- When used properly, multithreading can greatly increase the efficiency of a program.
- Creating unnecessary threads may harm the efficiency, however.
    - Switching between threads incur some overhead.
- Experience in programming will guide you when and how to use multithreading.
- When you feel that a large task can be decomposed into several mutually. exclusive smaller tasks, using multithreading will drastically reduce the execution time (provided your machine has multiple CPUs/cores).
    - To be specific, if you have a quad-core machine, the execution time of the program will roughly be reduced by 1/4th (assuming you're dividing the tasks among four threads).

More Interested?

In addition to the multithreading features described in this chapter, you will also want to explore the Fork/Join Framework. It provides a powerful means of creating multithreaded applications that automatically scale to make best use of multicore environments. The Fork/Join Framework is part of Java’s support for parallel programming, which is the name commonly given to the techniques that optimize some types of algorithms for parallel execution in systems that have more than one CPU. For a discussion of the Fork/Join Framework and other concurrency utilities, see Chapter 29.

End of Lectures 19, 20, 21.

Note: Some text of these slides are taken verbatim from your textbook.