CSE3040 Java Language

Lecture 26: Multithreaded Programming with Java (2)

Dept. of Computer Engineering,
Sogang University

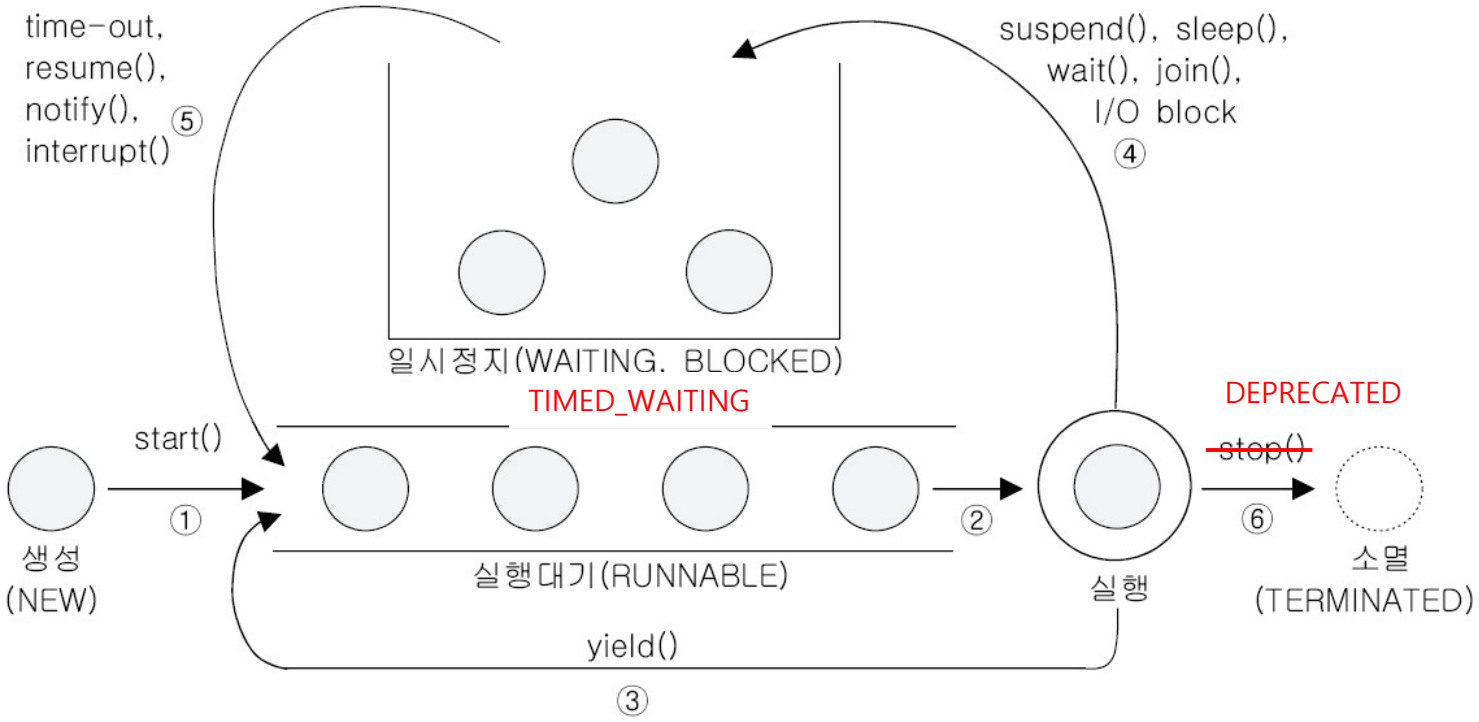
Thread States

- A thread is in one of the following states.

state	description
NEW	When a new thread is created, it is in this state. The thread has not yet started to run.
RUNNABLE	A thread that is ready to run is moved to this state. In this state, a thread might actually be running or it might be ready to run at any instant of time.
BLOCKED	A thread is in this state if it is waiting to enter a synchronized section. In other words, it is waiting to obtain a lock, which is currently held by another thread.
WAITING	A thread is in this state if it is waiting for another thread to call notify() or notifyAll(). A thread enters this state by calling wait() or join().
TIMED_WAITING	A thread is in this state if it is waiting but with a timeout. A thread enters this state by calling sleep(). Also, method wait can be called with a specified timeout. In this case, the thread enters this state.
TERMINATED	A thread enters this state when it returns from run().

Thread States

- Thread states and methods that change the states



Controlling Threads

- Methods related to thread control

Method	Description
static void sleep(long millis) static void sleep(long millis, int nanos)	Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds (plus the specified number of nanoseconds).
void join()	Waits for this thread to die.
void join(long millis) void join(long millis, int nanos)	Waits at most *millis* milliseconds (plus *nanos* nanoseconds) for this thread to die.
void interrupt()	Interrupts this thread.
boolean isInterrupted()	Tests whether this thread has been interrupted.
static boolean interrupted()	Tests whether the current thread has been interrupted.
static void yield()	A hint to the scheduler that the current thread is willing to yield its current use of a processor.

sleep()

- Sleep() ceases execution of a thread for a specified amount of time.

```
static void sleep(long millis)
static void sleep(long millis, int nanos)
```

- This code causes the current thread to sleep for 0.0015 seconds.

```
try {
    Thread.sleep(1, 500000);
} catch(InterruptedException e) {}
```

- When sleep() is called, the thread enters TIMED_WAITING state.
  - The thread becomes RUNNABLE when the specified time runs out.

- While sleeping, if interrupt() is called on this thread, then an InterruptedException occurs, and the thread returns to the RUNNABLE state.

sleep()

- An example program using sleep().

```
class ThreadEx12_1 extends Thread {
    public void run() {
        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) {}
        for(int i=0; i<300; i++) {
            System.out.print("-");
        }
        System.out.print("<<end of th1>>");
    }
}

class ThreadEx12_2 extends Thread {
    public void run() {
        try {
            Thread.sleep(2000);
        } catch(InterruptedException e) {}

        for(int i=0; i<300; i++) {
            System.out.print("|");
        }
        System.out.print("<<end of th2>>");
    }
}
```

sleep()

- An example program using sleep(). (cont.)
  - the main method

```
public class Lecture {
    public static void main(String args[]) {
        ThreadEx12_1 th1 = new ThreadEx12_1();
        ThreadEx12_2 th2 = new ThreadEx12_2();
        th1.start();
        th2.start();
        try {
            Thread.sleep(3000);
        } catch(InterruptedException e) {}

        System.out.print("<<end of main>>");
    }
}
```

- th1 waits for 1 second before printing characters
- th2 waits for 2 seconds before printing characters
- main method waits for 3 seconds before printing the ending message.

interrupt() and interrupted()

- interrupt() interrupts another thread.
- isInterrupted() is an instance method that checks whether a thread is in the interrupted state.
- interrupted() is a static method that returns the interrupted state of the current thread and sets the state to false.

```
void interrupt()
boolean isInterrupted()
static boolean interrupted()
```

interrupt() and interrupted()

- Every thread has an instance variable which indicates if a thread is interrupted.
  - If another thread calls th1.interrupt(), the thread th1 is interrupted.
  - Specifically, the instance variable of thread th1 indicates that it is interrupted.
  - We can check whether a thread is interrupted or not by calling interrupted().

```
class ThreadEx13_1 extends Thread {
    public void run() {
        int i = 10;
        while(i != 0 && !isInterrupted()) {
            System.out.println(i--);
            for(long x = 0; x<2500000000L; x++);  // waste time
        }
    }
}

public class Lecture {
    public static void main(String[] args) {
        ThreadEx13_1 th1 = new ThreadEx13_1();
        th1.start();
        String input = JOptionPane.showInputDialog("Enter any string.");
        System.out.println("You entered " + input);
        th1.interrupt();
        System.out.println("isInterrupted(): " + th1.isInterrupted());
    }
}
```

interrupt() and interrupted()

- In this example, the countdown continues until the end.
  - If th1.interrupt() is called while th1 is in sleep, an InterruptedException occurs. In this case, the interrupted state of th1 becomes false.

```
class ThreadEx14_1 extends Thread {
    public void run() {
        int i=10;
        while(i!=0 && !isInterrupted()) {
            System.out.println(i--);
            try {
                Thread.sleep(1000);
            } catch(InterruptedException e) {}
        }
        System.out.println("countdown complete.");
    }
}
public class Lecture {
    public static void main(String[] args) throws Exception {
        ThreadEx14_1 th1 = new ThreadEx14_1();
        th1.start();
        String input = JOptionPane.showInputDialog("Enter any string.");
        System.out.println("You entered " + input);
        th1.interrupt();
        System.out.println("isInterrupted(): " + th1.isInterrupted());
    }
}
```

yield()

- Suppose threads th1, th2, and th3 were all in RUNNABLE state. They are placed in the run queue.
- Assuming there is a single CPU core, the scheduler determines which thread to execute on the CPU.
- For example, th1 runs for 1 second, then th2 runs for 1 second, and then th3 runs for 1 second, and so on.
- While th1 is running, if yield() is called, then it gives up its remaining CPU time and returns to the run queue.

- Once a thread calls yield(), it is up to the scheduler to decide which thread should be executed on the yielded CPU.

join()

- When join() is called, the current thread ceases execution, waiting for the other thread to finish execution.
- If join is called with an argument, the current thread waits for the specified amount of time for the other thread to finish execution.

```
void join()
void join(long millis)
void join(long millis, int nanos)
```

- join() is used when another thread must be executed first.

```
try {
    th1.join();  // current thread waits until thread th1 finishes execution (and dies).
} catch(InterruptedExecution e) {}
```

join()

- The main method waits for other threads to finish execution.

```
class ThreadEx19_1 extends Thread {
    public void run() {
        for(int i=0; i<300; i++) { System.out.print(new String("-")); }
    }
}
class ThreadEx19_2 extends Thread {
    public void run() {
        for(int i=0; i<300; i++) { System.out.print(new String("|")); }
    }
}
public class Lecture {
    static long startTime = 0;
    public static void main(String args[]) {
        ThreadEx19_1 th1 = new ThreadEx19_1();
        ThreadEx19_2 th2 = new ThreadEx19_2();
        th1.start();
        th2.start();
        startTime = System.currentTimeMillis();
        try {
            th1.join();
            th2.join();
        } catch(InterruptedException e) { }
        System.out.print("elapsed time: " + (System.currentTimeMillis() - Lecture.startTime));
    }
}
```

join()

- This example simulates a garbage collector without the use of join().
  - Why does usedMemory go over 1000?

```
class ThreadEx20_1 extends Thread {
    final static int MAX_MEMORY = 1000;
    int usedMemory = 0;

    public void run() {
        while(true) {
            try {
                Thread.sleep(10*1000);   // sleep for 10 seconds
            } catch(InterruptedException e) {
                System.out.println("Awaken by interrupt()");
            }
            gc();
            System.out.println("Garbage Collected. Free Memory: " + freeMemory());
        }
    }

    public void gc() {
        usedMemory -= 300;
        if(usedMemory < 0) usedMemory = 0;
    }
    public int totalMemory() { return MAX_MEMORY; }
    public int freeMemory() { return MAX_MEMORY - usedMemory; }
}
```

join()

- Example continued.

```
public class Lecture {
    public static void main(String args[]) {
        ThreadEx20_1 gc = new ThreadEx20_1();
        gc.setDaemon(true);  // set this thread as a daemon thread.
        gc.start();

        int requiredMemory = 0;
        for(int i=0; i<20; i++) {
            requiredMemory = (int)(Math.random()*10) * 20;
            if(gc.freeMemory() < requiredMemory || gc.freeMemory() < gc.totalMemory() * 0.4) {
                gc.interrupt();
            }
            gc.usedMemory += requiredMemory;
            System.out.println("usedMemory: " + gc.usedMemory);
        }
    }
}
```

join()

- From the previous example, we insert join() so that the garbage collector can execute for enough amount of time.
  - Why do we use timeout here? (gc.join(100);)

```
public class Lecture {
    public static void main(String args[]) {
        ThreadEx20_1 gc = new ThreadEx20_1();
        gc.setDaemon(true);  // set this thread as a daemon thread.
        gc.start();

        int requiredMemory = 0;
        for(int i=0; i<20; i++) {
            requiredMemory = (int)(Math.random()*10) * 20;
            if(gc.freeMemory() < requiredMemory || gc.freeMemory() < gc.totalMemory() * 0.4) {
                gc.interrupt();
                try {
                    gc.join(100);
                } catch(InterruptedException e) {}
            }
            gc.usedMemory += requiredMemory;
            System.out.println("usedMemory: " + gc.usedMemory);
        }
    }
}
```

Daemon Thread

- A daemon thread is a thread that automatically dies when all other non-daemon threads finish execution.
- Typically, daemon threads use infinite loops to do background work.
- To set a thread to be a daemon thread, we call setDaemon(true), before calling start().

```
public class Lecture implements Runnable {
    static boolean autoSave = false;
    public static void main(String[] args) {
        Thread t = new Thread(new Lecture());
        t.setDaemon(true);
        t.start();
        for (int i = 1; i <= 10; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {}
            System.out.println(i);
            if (i == 5)
                autoSave = true;
        }

        System.out.println("Terminating program.");
    }
```

Daemon Thread

- In this example, the daemon thread saves work to a file every 3 seconds, after 5 seconds from the start. (Once *autoSave* becomes true.)

```
    public void run() {
        while (true) {
            try {
                Thread.sleep(3 * 1000);
            } catch (InterruptedException e) {}
            if (autoSave) autoSave();
        }
    }

    public void autoSave() {
        System.out.println("Your work is saved to a file.");
    }
}
```

Thread Synchronization

- Critical Section
  - It is possible that multiple threads access a shared variable.
    - Multiple threads may read from and write to the variable.
  - It is possible that if multiple threads execute a code section concurrently, problems could occur.
    - Thus, while one thread is executing the code section, other threads must not enter the section.
    - This kind of code section is called a Critical Section.

Thread Synchronization

- An example where multiple threads access a shared memory.
  - We have a class Account which keeps a balance.
  - It has a method withdraw() which withdraws money from the account.

```
class Account {
    private int balance = 1000;
    public int getBalance() {
        return balance;
    }
    public void withdraw(int money) {
        if(balance >= money) {
            try { Thread.sleep(1000); } catch(InterruptedException e) {}
            balance -= money;
        }
    }
}
```

Thread Synchronization

- An example where multiple threads access a shared memory.
  - The thread goes through an infinite loop.
  - In each loop, the thread checks the balance, and if the balance is positive, then it withdraws 200 from the balance.
  - What is the problem with the following code?

```
class RunnableEx21 implements Runnable {
    Account acc = new Account();
    public void run() {
        while(acc.getBalance() >= 200) {
            int money = 200;
            acc.withdraw(money);
            System.out.println("balance: " + acc.getBalance());
        }
    }
}

public class Lecture {
    public static void main(String[] args) {
        Runnable r = new RunnableEx21();
        new Thread(r).start();
        new Thread(r).start();
    }
}
```

Thread Synchronization

- What happens in the program
  - Suppose the balance is 200.
  - Thread t1 calls withdraw(). Since balance >= 200, t1 goes into sleep for 1 second.
  - While t1 is asleep, thread t2 calls withdraw(). Since balance is still 200, t2 goes into sleep.
  - t1 wakes up and reduces balance to 0.
  - t2 wakes up and reduces balance to -200.

- The problem
  - Before t1 subtracts money from balance, t2 may check the balance.
  - Then, t1 subtracts money from balance, which becomes zero.
  - While t1 is inside withdraw(), t2 should not enter the method.

```
class Account {
    ...
    public void withdraw(int money) {
        if(balance >= money) {
            try { Thread.sleep(1000); } catch(InterruptedException e) {}
            balance -= money;
        }
    }
}
```

Thread Synchronization

- Solution
  - Make the code block a critical section using synchronized.
  - The critical section could be a whole method or a code block.

```
class Account {
    ...
    public synchronized void withdraw(int money) {
        if(balance >= money) {
            try { Thread.sleep(1000); } catch(InterruptedException e) {}
            balance -= money;
        }
    }
}
```

```
class Account {
    ...
    public void withdraw(int money) {
        synchronized(this) {
            if(balance >= money) {
                try { Thread.sleep(1000); } catch(InterruptedException e) {}
                balance -= money;
            }
        }
    }
}
```

Programming Lab #26

26-01. Using Thread.sleep()

- Execute the following code and understand the results.
- Try changing the amount of sleep for the three threads.

```
class ThreadEx12_1 extends Thread {
    public void run() {
        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) {}
        for(int i=0; i<300; i++) {
            System.out.print("-");
        }
        System.out.print("<<end of th1>>");
    }
}

class ThreadEx12_2 extends Thread {
    public void run() {
        try {
            Thread.sleep(2000);
        } catch(InterruptedException e) {}

        for(int i=0; i<300; i++) {
            System.out.print("|");
        }
        System.out.print("<<end of th2>>");
    }
}
```

26-01. Using Thread.sleep()

- (cont.)

```
public class Ex26_01 {
     public static void main(String args[]) {
        ThreadEx12_1 th1 = new ThreadEx12_1();
        ThreadEx12_2 th2 = new ThreadEx12_2();
        th1.start();
        th2.start();
        try {
            Thread.sleep(3000);
        } catch(InterruptedException e) {}

        System.out.print("<<end of main>>");
    }
}
```

26-02. Using interrupt() (1)

- Execute the following code and understand the results.
- Why does the count down stop?

```
class ThreadEx13_1 extends Thread {
    public void run() {
        int i = 10;
        while(i != 0 && !isInterrupted()) {
            System.out.println(i--);
            for(long x = 0; x<2500000000L; x++);  // waste time
        }
    }
}

public class Ex26_02 {
    public static void main(String[] args) {
        ThreadEx13_1 th1 = new ThreadEx13_1();
        th1.start();
        String input = JOptionPane.showInputDialog("Enter any string.");
        System.out.println("You entered " + input);
        th1.interrupt();
        System.out.println("isInterrupted(): " + th1.isInterrupted());
    }
}
```

26-03. Using interrupt() (2)

- Execute the following code and understand the results.
- Why does the count down continue?

```
class ThreadEx14_1 extends Thread {
    public void run() {
        int i=10;
        while(i!=0 && !isInterrupted()) {
            System.out.println(i--);
            try {
                Thread.sleep(1000);
            } catch(InterruptedException e) {}
        }
        System.out.println("countdown complete.");
    }
}

public class Ex26_03 {
    public static void main(String[] args) throws Exception {
        ThreadEx14_1 th1 = new ThreadEx14_1();
        th1.start();
        String input = JOptionPane.showInputDialog("Enter any string.");
        System.out.println("You entered " + input);
        th1.interrupt();
        System.out.println("isInterrupted(): " + th1.isInterrupted());
    }
}
```

26-04. Using join() (1)

- Execute the following code and understand the results.

```
class ThreadEx19_1 extends Thread {
    public void run() {
        for(int i=0; i<300; i++) { System.out.print(new String("-")); }
    }
}
class ThreadEx19_2 extends Thread {
    public void run() {
        for(int i=0; i<300; i++) { System.out.print(new String("|")); }
    }
}
public class Ex26_04 {
    static long startTime = 0;
    public static void main(String args[]) {
        ThreadEx19_1 th1 = new ThreadEx19_1();
        ThreadEx19_2 th2 = new ThreadEx19_2();
        th1.start();
        th2.start();
        startTime = System.currentTimeMillis();
        try {
            th1.join();
            th2.join();
        } catch(InterruptedException e) { }
        System.out.print("elapsed time: " + (System.currentTimeMillis() - Ex26_04.startTime));
    }
}
```

26-05. Using join() (2)

- Why does usedMemory go over 1000?
- Modify the code so that usedMemory will not go over 1000.

```
class ThreadEx20_1 extends Thread {
    final static int MAX_MEMORY = 1000;
    int usedMemory = 0;

    public void run() {
        while(true) {
            try {
                Thread.sleep(10*1000);   // sleep for 10 seconds
            } catch(InterruptedException e) {
                System.out.println("Awaken by interrupt()");
            }
            gc();
            System.out.println("Garbage Collected. Free Memory: " + freeMemory());
        }
    }

    public void gc() {
        usedMemory -= 300;
        if(usedMemory < 0) usedMemory = 0;
    }
    public int totalMemory() { return MAX_MEMORY; }
    public int freeMemory() { return MAX_MEMORY - usedMemory; }
}
```

26-05. Using join() (2)

- (cont.)

```
public class Ex26_05 {
     public static void main(String args[]) {
        ThreadEx20_1 gc = new ThreadEx20_1();
        gc.setDaemon(true);  // set this thread as a daemon thread.
        gc.start();

        int requiredMemory = 0;
        for(int i=0; i<20; i++) {
            requiredMemory = (int)(Math.random()*10) * 20;
            if(gc.freeMemory() < requiredMemory || gc.freeMemory() < gc.totalMemory() * 0.4) {
                gc.interrupt();
            }
            gc.usedMemory += requiredMemory;
            System.out.println("usedMemory: " + gc.usedMemory);
        }
    }
}
```

26-06. Daemon Thread

- Execute the following code and understand the results.

```
public class Ex26_06 implements Runnable {
    static boolean autoSave = false;
    public static void main(String[] args) {
        Thread t = new Thread(new Ex26_06());
        t.setDaemon(true);
        t.start();
        for (int i = 1; i <= 10; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {}
            System.out.println(i);
            if (i == 5)
                autoSave = true;
        }
        System.out.println("Terminating program.");
    }
    public void run() {
        while (true) {
            try {
                Thread.sleep(3 * 1000);
            } catch (InterruptedException e) {}
            if (autoSave) autoSave();
        }
    }
    public void autoSave() {
        System.out.println("Your work is saved to a file.");
    }
}
```

26-07. Critical Section

- Execute the following code and understand the results.
- What is the problem with this code? Why does the balance go below zero?
- Modify the code to correct the problem.

```
class Account {
    private int balance = 1000;
    public int getBalance() {
        return balance;
    }
    public void withdraw(int money) {
        if(balance >= money) {
            try { Thread.sleep(1000); } catch(InterruptedException e) {}
            balance -= money;
        }
    }
}

class RunnableEx21 implements Runnable {
    Account acc = new Account();
    public void run() {
        while(acc.getBalance() >= 200) {
            int money = 200;
            acc.withdraw(money);
            System.out.println("balance: " + acc.getBalance());
        }
    }
}
```

26-07. Critical Section

- (cont.)

```
public class Ex26_07 {
    public static void main(String[] args) {
        Runnable r = new RunnableEx21();
        new Thread(r).start();
        new Thread(r).start();
    }
}
```

End of Class

Instructor office: AS818A

Email: jso1@sogang.ac.kr