CSE3040 Java Language

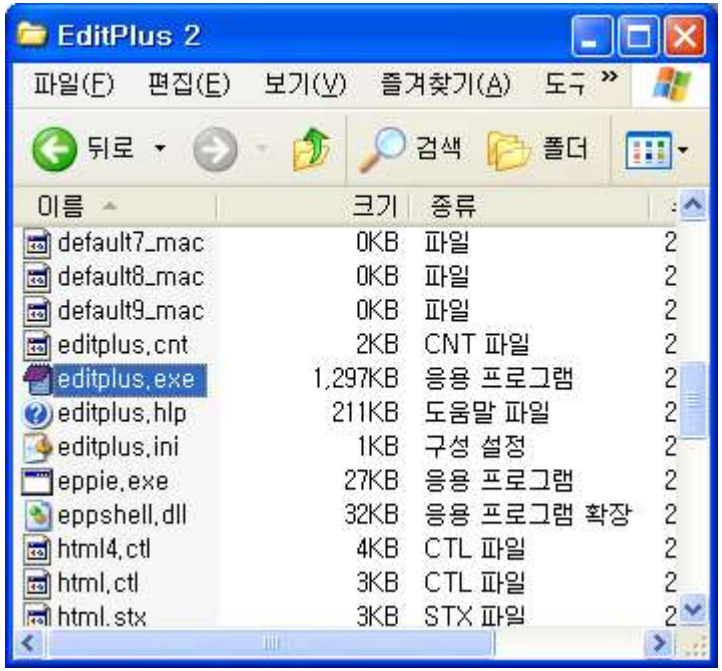
Lecture 23: Multithreaded Programming with Java (1)

Dept. of Computer Engineering,
Sogang University

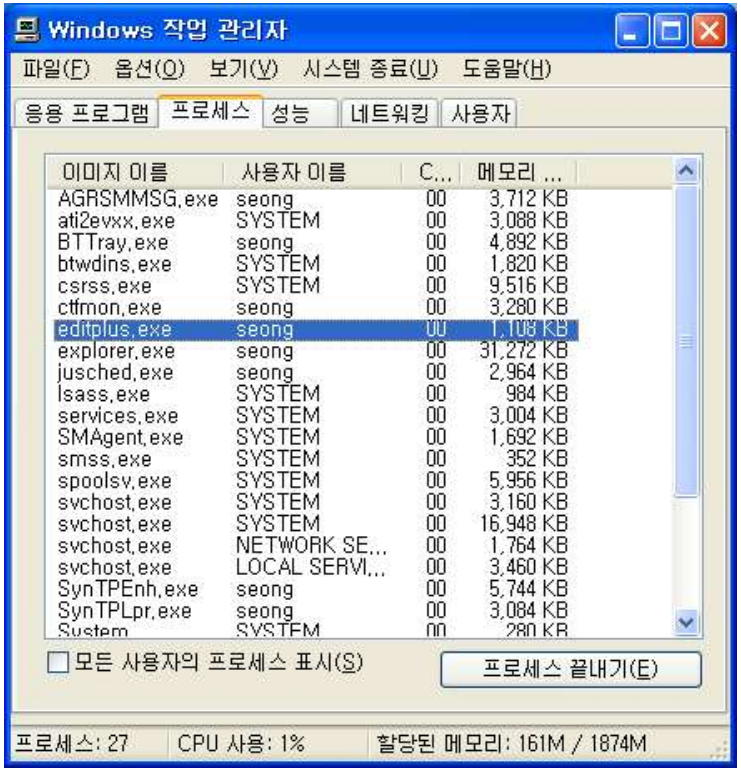
Background: processes and threads

- Process: a program in execution
  - When we execute our Java program, OS creates a process run the program code.
  - Necessary amount of memory is allocated to the process.

▶ Program



▶ Process

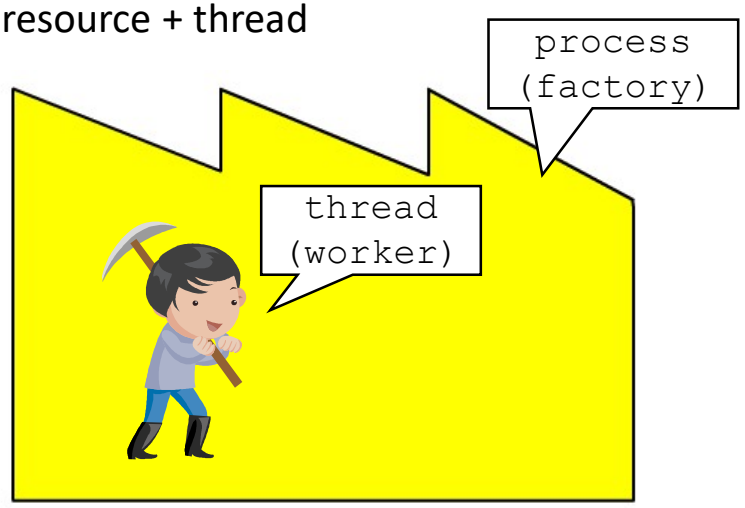


Background: processes and threads

- Thread: a smallest sequence of programmed instructions that can be managed independently by a scheduler.
  - A process has at least one thread.
  - A process may have multiple threads.
  - When a process has more than two threads, it is called a multi-threaded process.

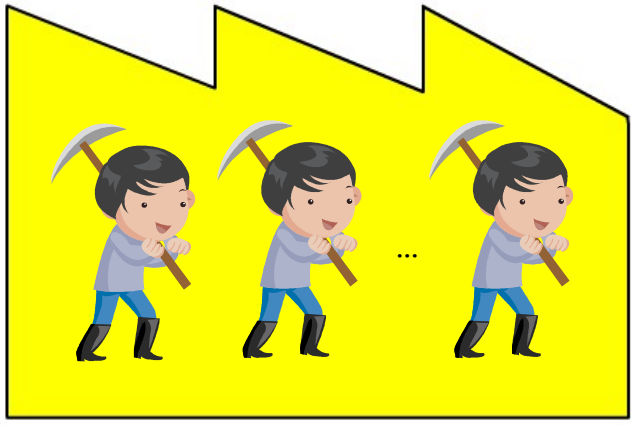
▶ A single-threaded process

= resource + thread



▶ A multi-threaded process

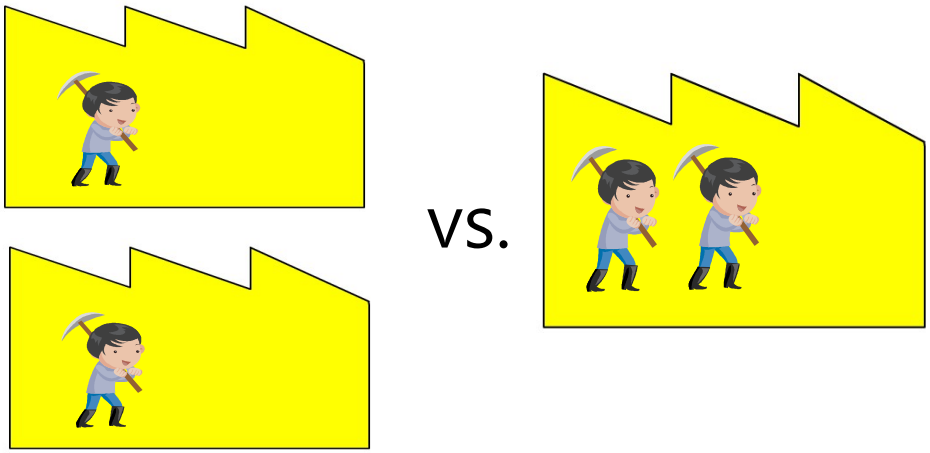
= resource + N x threads



Background: multitasking vs. multithreading

- Multitasking
  - An operating system runs multiple processes (tasks) concurrently.
  - Most modern OSes support multitasking. (Windows, Linux, etc.)

- Multithreading
  - A process creates multiple threads and run them concurrently.
  - Two single-threaded processes vs. One two-threaded process
    - Creating a thread is more light-weight than creating a process.



Background: multithreading

- Advantages
  - Efficient use of resources
    - While one thread is reading a file from a disk, another thread sends messages on the network.
  - Fast response to users
    - If the server has a single thread, all other users must wait while one user is communicating with the server.
  - Improved modularity of codes by separating tasks to threads
    - e.g. a chat server: one thread for receiving requests (receptioninst thread), N threads for communicating with N users (one server thread per user).

- Disadvantages
  - Programmer should put extra effort to avoid problems
    - Synchronization: A thread reading from memory while another thread writing to the memory.
    - Dead-lock: Thread A is holding resource C and requesting resource D, while thread B is holding resource D and requesting resource C.
    - Inefficiency: A thread may be waiting in idle state although it can work on other tasks.

Implementing threads in Java

- In order to create and run a thread, we either need to:
  - extend class Thread, or
  - implement interface Runnable.
    - When implementing interface Runnable, we should implement method run.
    - When extending class Thread, it is typical that we override method run.

```
class MyThread extends Thread {
    public void run() {   // overriding
        /* tasks to run on a thread */
    }
}
```

```
class MyThread implements Runnable {
    public void run() {
        /* tasks to run on a thread */
    }
}
```

Implementing threads in Java: Example

- Extending class Thread vs. implementing interface Runnable
  - When class ThreadEx1_2 implements Runnable, we need to create a Thread type instance and give the ThreadEx1_2 instance as an argument to the constructor.

```
public class Lecture {
    public static void main(String[] args) {
        ThreadEx1_1 t1 = new ThreadEx1_1();
        Runnable r = new ThreadEx1_2();
        Thread t2 = new Thread(r);
        t1.start();
        t2.start();
    }
}
class ThreadEx1_1 extends Thread {
    public void run() {
        for(int i=0; i<5; i++) System.out.println(getName());
    }
}
class ThreadEx1_2 implements Runnable {
    public void run() {
        for(int i=0; i<5; i++) System.out.println(Thread.currentThread().getName());
    }
}
```

Implementing threads in Java: Example

- Extending class Thread vs. implementing interface Runnable
  - When class ThreadEx1_2 implements Runnable, it cannot call getName() which is defined in class Thread. Instead, we should use Thread.currentThread().

```
public class Lecture {
    public static void main(String[] args) {
        ThreadEx1_1 t1 = new ThreadEx1_1();
        Runnable r = new ThreadEx1_2();
        Thread t2 = new Thread(r);
        t1.start();
        t2.start();
    }
}
class ThreadEx1_1 extends Thread {
    public void run() {
        for(int i=0; i<5; i++) System.out.println(getName());
    }
}
class ThreadEx1_2 implements Runnable {
    public void run() {
        for(int i=0; i<5; i++) System.out.println(Thread.currentThread().getName());
    }
}
```

Implementing threads in Java: Example

- A thread starts running when method start is called.
  - The thread will go into the scheduler, and is executed on the CPU when the scheduler dispatches the thread.
  - Calling start twice on the same thread creates an exception.
    - IllegalThreadStateException (a descendant of RuntimeException)

```
public class Lecture {
    public static void main(String[] args) {
        ThreadEx1_1 t1 = new ThreadEx1_1();
        Runnable r = new ThreadEx1_2();
        Thread t2 = new Thread(r);
        t1.start();
        t2.start();
    }
}
```

Threads: start and run

- When we want to run a task on a thread, we define a class that extends Thread or implements Runnable.
- Inside the class, we override or implement method run().
- To make thread running, we create an instance of the class and call method start().
- Why do we call start() instead of run()?

```
class MyThread extends Thread {
    public void run() {   // overriding
        /* tasks to run on a thread */
    }
}
```

```
public class ThreadTest {
    public static void main() {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

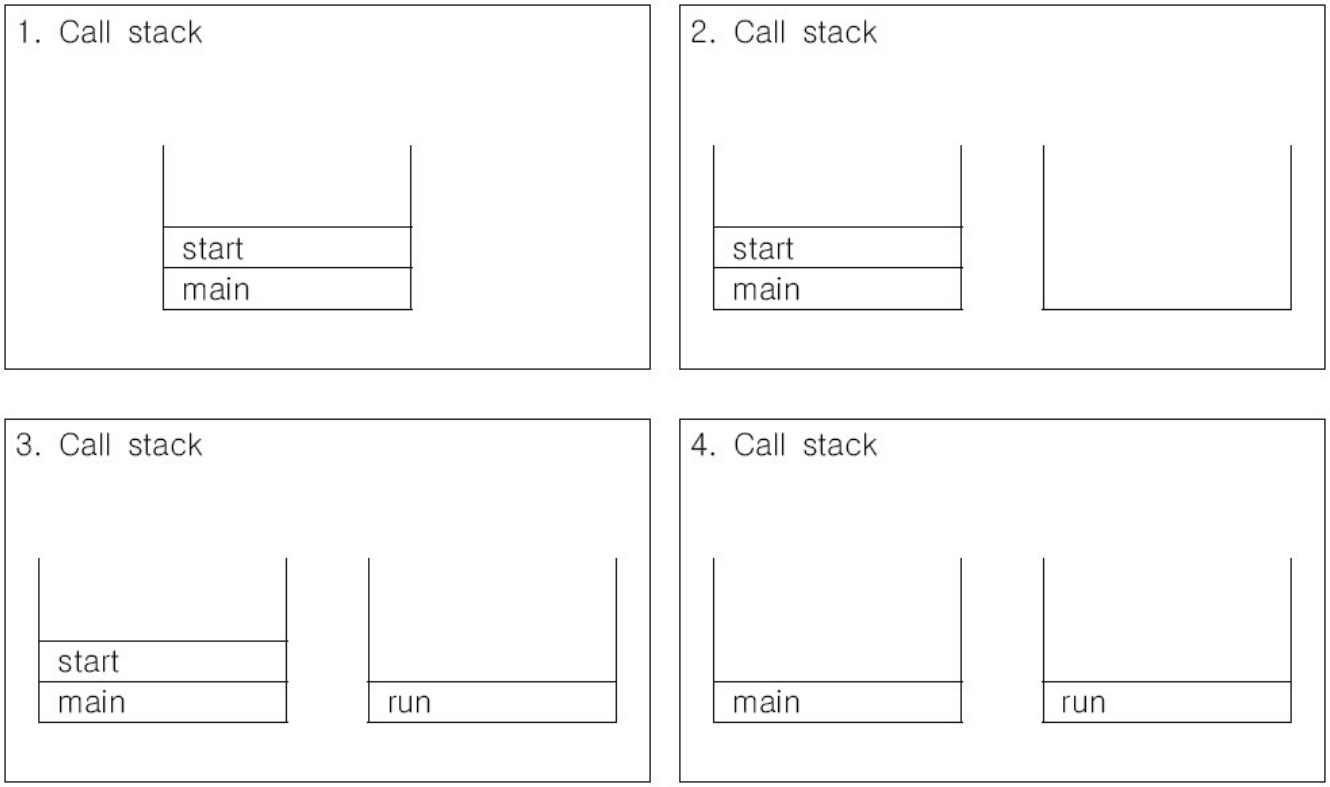
Threads: start and run

- If method run() is called inside main(), it just calls the method, instead of starting a new thread.
- The call stack will look like this.

- In the call stack, the one at the top is the method that is being executed, and all the other methods are caller methods that are waiting for the called method to return.

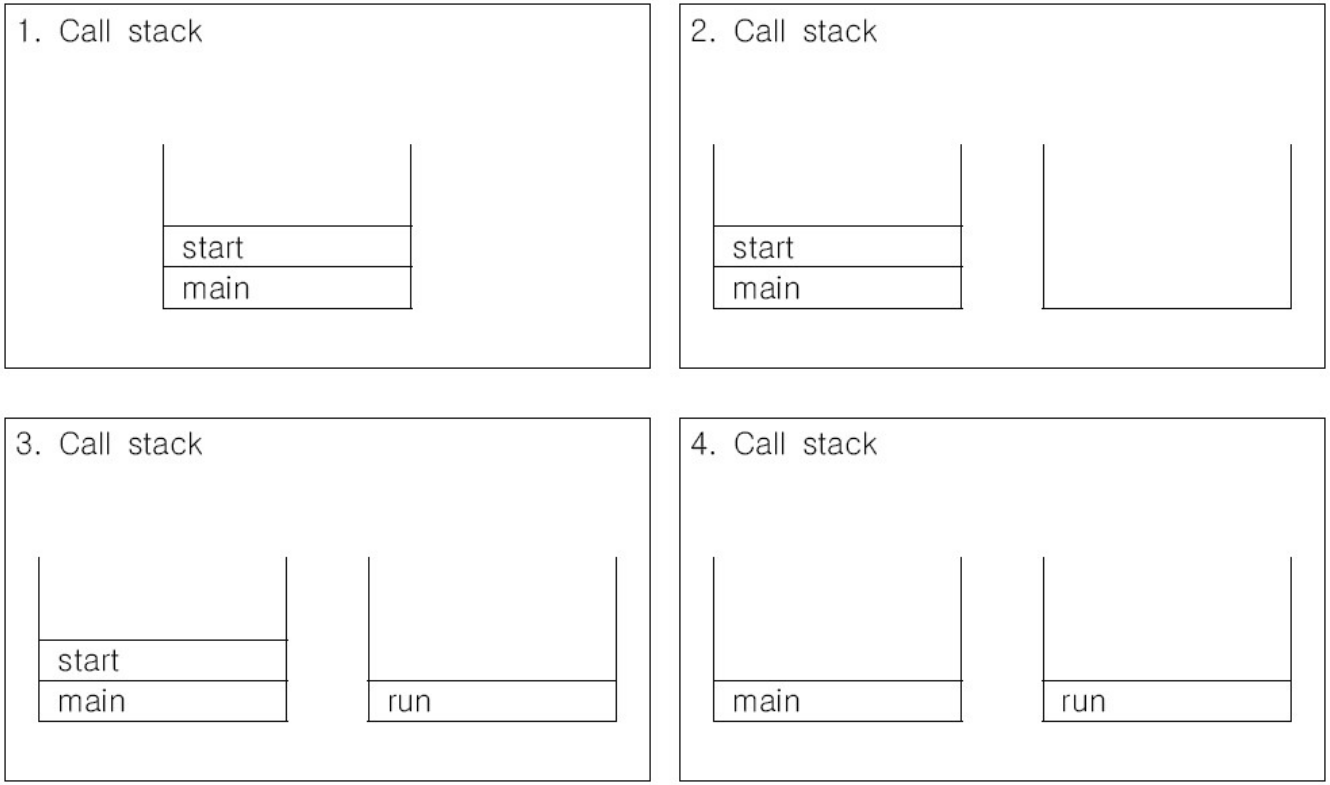
Threads: start and run

- If we call start(), it will create a new call stack.
- In the new call stack, the first method is run().
- Then, start() is returned.



Threads: start and run

- After calling start(), we have two threads.
  - One thread is executing main(), and the other thread is executing run().
  - If the computer has two cores, they can run concurrently.
  - If the computer has a single core, the scheduler decides which thread will use the CPU when.



Threads: start and run

- Since two threads are running concurrently, we do not know which thread will finish earlier.

- If the thread running main() finishes before the thread running run(), its call stack is destroyed but the program does not end yet.

- The program ends when all the threads are finished.

Threads: Example 2

- The purpose of this example is to see the call stack of the new thread.
  - The first (bottom) method in the call stack is run(), not main().

```
public class Lecture {
    public static void main(String args[]) throws Exception {
        ThreadEx2_1 t1 = new ThreadEx2_1();
        t1.start();
    }
}

class ThreadEx2_1 extends Thread {
    public void run() {
        throwException();
    }
    public void throwException() {
        try {
            throw new Exception();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Threads: Example 3

- Instead of calling start(), we call run() method of the thread instance.
- See how the printed call stack is different.

```
public class Lecture {
    public static void main(String args[]) throws Exception {
        ThreadEx3_1 t1 = new ThreadEx3_1();
        t1.run();
    }
}

class ThreadEx3_1 extends Thread {
    public void run() {
        throwException();
    }
    public void throwException() {
        try {
            throw new Exception();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Single thread vs. multiple threads: experiment

- A program that prints -"s and "|"s using a single thread.
  - The two tasks are done sequentially.

```
public class Lecture {
    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();

        for(int i=0; i<300; i++) {
            System.out.printf("%s",  new String("-"));
        }

        System.out.print("elapsed time: " + (System.currentTimeMillis() - startTime));

        for(int i=0; i<300; i++) {
            System.out.printf("%s",  new String("|"));
        }

        System.out.print("elapsed time: " + (System.currentTimeMillis() - startTime));
    }
}
```

Single thread vs. multiple threads: experiment

- A program that prints -"s and "|"s using two threads.
  - The two tasks are done concurrently, and their finish times are similar.

```
public class Lecture {
    static long startTime = 0;

    public static void main(String[] args) {
        ThreadEx5_1 th1 = new ThreadEx5_1();
        th1.start();
        startTime = System.currentTimeMillis();
        for(int i=0; i<300; i++) System.out.printf("%s", new String("-"));
        System.out.print("elapsed time 1: " + (System.currentTimeMillis() - Lecture.startTime));
    }
}

class ThreadEx5_1 extends Thread {
    public void run() {
        for(int i=0; i<300; i++) System.out.printf("%s", new String("|"));
        System.out.print("elapsed time 2: " + (System.currentTimeMillis() - Lecture.startTime));
    }
}
```

Single thread vs. multiple threads: discussion

- In the previous examples, we printed 300 "-"s and "|"s using a single thread and multiple threads.

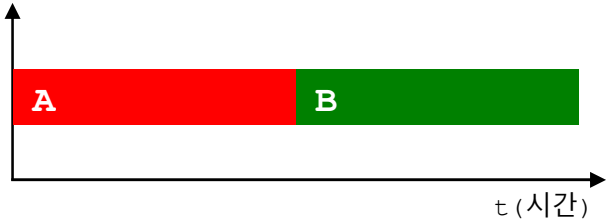
- What can we observe from the running time of two configurations?

- What happens when we print 3,000 or 30,000 letters instead of 300?

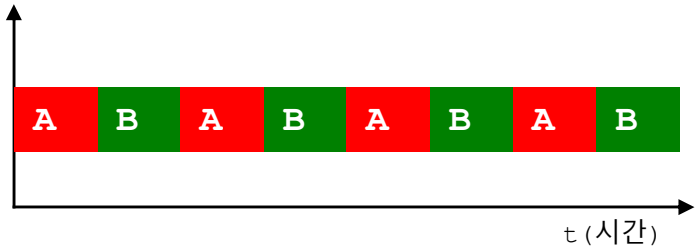
- What if we run tasks other than printing characters on the screen?

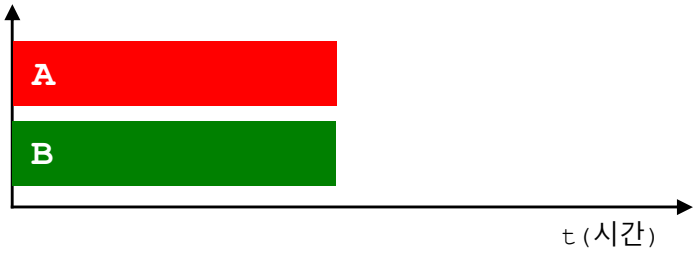
Single thread vs. multiple threads

- With a single thread, the two tasks are executed sequentially.



- If tasks A and B run on a different thread, there are two cases.
  - If there is only one CPU core, then the two tasks cannot run simultaneously.
    - The scheduler toggles between the two threads, so that the user feels like the two tasks are running simultaneously.
    - In this case, the total running time will be the same as a single-thread case (disregarding the overhead for creating threads, etc.)
  - If there are multiple CPU cores, the two tasks can actually run simultaneously.
    - Total running time can be reduced.
      - (Actual running time depends on how threads access shared resources.)





Benefits of multithreading

- The program asks user to input a string, and also counts down 10 seconds.
- Single-threaded version

```
public class Lecture {
    public static void main(String[] args) throws Exception {
        String input = JOptionPane.showInputDialog("Enter any string.");
        System.out.println("You have entered: " + input);

        for(int i=10; i>0; i--) {
            System.out.println(i);
            try {
                Thread.sleep(1000);
            } catch(Exception e) { /* do nothing */ }
        }
    }
}
```

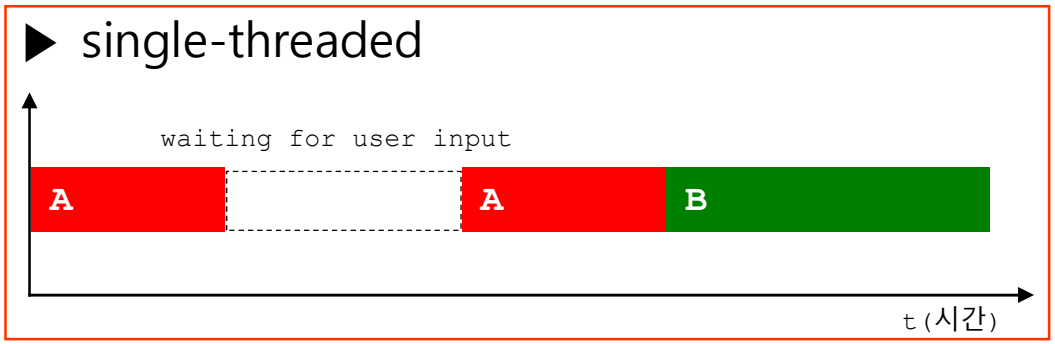
Benefits of multithreading

- The program asks user to input a string, and also counts down 10 seconds.
- Multi-threaded version

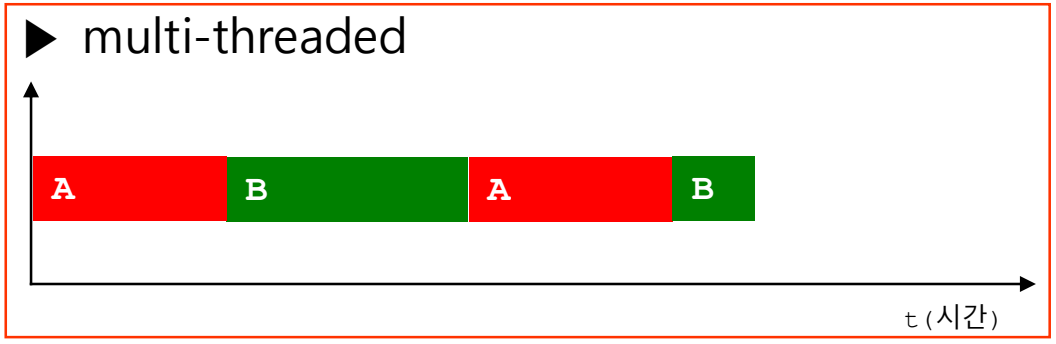
```
public class Lecture {
    public static void main(String[] args) throws Exception {
        ThreadEx7_1 th1 = new ThreadEx7_1();
        th1.start();

        String input = JOptionPane.showInputDialog("Enter any string.");
        System.out.println("You have entered: " + input);
    }
}

class ThreadEx7_1 extends Thread {
    public void run() {
        for(int i=10; i>0; i--) {
            System.out.println(i);
            try {
                sleep(1000);
            } catch(Exception e) { }
        }
    }
}
```

Benefits of multithreading

- If we use a single thread, the thread must wait while waiting for the user input.



- If we use multiple threads, thread 2 can take over the CPU while thread 1 is waiting for user input.



Programming Lab #23

23-01. Implementing threads

- Execute the following code and understand the results.
- Understand the difference between extending Thread and implementing Runnable.

```
class ThreadEx1_1 extends Thread {
	public void run() {
		for(int i=0; i<5; i++) System.out.println(getName());
	}
}

class ThreadEx1_2 implements Runnable {
	public void run() {
		for(int i=0; i<5; i++) System.out.println(Thread.currentThread().getName());
	}
}

public class Ex23_01 {
	public static void main(String[] args) {
		ThreadEx1_1 t1 = new ThreadEx1_1();
		Runnable r = new ThreadEx1_2();
		Thread t2 = new Thread(r);
		t1.start();
		t2.start();
	}
}
```

23-02. Thread.start() vs. Thread.run()

- Execute the following code and understand the results.
- Understand the difference between calling start() and calling run() of a thread object.
  - You should call start() to create a new thread.

```
class ThreadEx2_1 extends Thread {
	public void throwException() {
		try {
		    throw new Exception();
		}
		catch(Exception e) {
		    e.printStackTrace();
		}
	}
	public void run() {
		throwException();
	}
}

public class Ex23_02 {
	public static void main(String[] args) throws Exception {
		ThreadEx2_1 t1 = new ThreadEx2_1();
		t1.start();
	}
}
```

23-03. Single thread vs. multiple threads

- Execute the following code and understand the results.
- This is a single-threaded version of printing characters '-' and '|'.
  - Note the finishing times of the two tasks.

```
public class Ex23_03 {

     public static void main(String[] args) {

        long startTime = System.currentTimeMillis();

        for(int i=0; i<300; i++) {
            System.out.printf("%s",  new String("-"));
        }

        System.out.print("elapsed time: " + (System.currentTimeMillis() - startTime));

        for(int i=0; i<300; i++) {
            System.out.printf("%s",  new String("|"));
        }

        System.out.print("elapsed time: " + (System.currentTimeMillis() - startTime));

    }
}
```

23-04. Single thread vs. multiple threads

- Execute the following code and understand the results.
- This is a multi-threaded version of printing characters '-' and '|'.
  - Note the finishing times of the two tasks.

```
public class Ex23_04 {
    static long startTime = 0;

    public static void main(String[] args) {
        ThreadEx5_1 th1 = new ThreadEx5_1();
        th1.start();
        startTime = System.currentTimeMillis();
        for(int i=0; i<300; i++) System.out.printf("%s", new String("-"));
        System.out.print("elapsed time 1: " + (System.currentTimeMillis() - Ex23_04.startTime));
    }
}

class ThreadEx5_1 extends Thread {
    public void run() {
        for(int i=0; i<300; i++) System.out.printf("%s", new String("|"));
        System.out.print("elapsed time 2: " + (System.currentTimeMillis() - Ex23_04.startTime));
    }
}
```

23-05. Benefit of using multiple threads

- Execute the following code and understand the results.
- This is a single-threaded version
  - Counting down can start after the user inputs a string at the window.

```
public class Ex23_05 {
     public static void main(String[] args) throws Exception {
        String input = JOptionPane.showInputDialog("Enter any string.");
        System.out.println("You have entered: " + input);

        for(int i=10; i>0; i--) {
            System.out.println(i);
            try {
                Thread.sleep(1000);
            } catch(Exception e) { /* do nothing */ }
        }
    }
}
```

23-06. Benefit of using multiple threads

- Execute the following code and understand the results.
- This is a multi-threaded version
  - Counting down can take place concurrently with the window.

```
public class Ex23_06 {
     public static void main(String[] args) throws Exception {
        ThreadEx7_1 th1 = new ThreadEx7_1();
        th1.start();

        String input = JOptionPane.showInputDialog("Enter any string.");
        System.out.println("You have entered: " + input);
    }
}

class ThreadEx7_1 extends Thread {
    public void run() {
        for(int i=10; i>0; i--) {
            System.out.println(i);
            try {
                sleep(1000);
            } catch(Exception e) { }
        }
    }
}
```

End of Class

Instructor office: AS818A

Email: jso1@sogang.ac.kr