쓰레드 생성과 실행

소유자	종수 김
태그	

쓰레드 시작 1

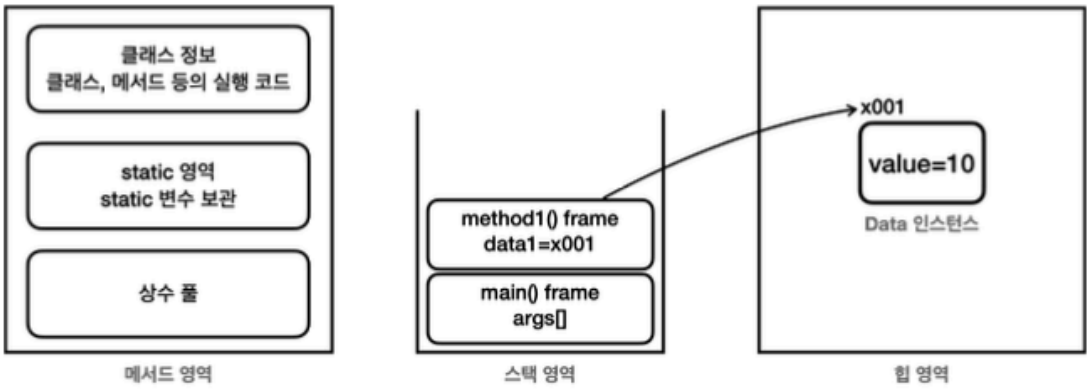
유사성

OS 프로세스 메모리구조 ↔ JVM의 메모리 구조

관점	OS 프로세스	JVM
코드/클래스	코드 영역	메서드 영역
전역/정적 변수	데이터 영역	메서드 영역 (static)
동적 메모리	힙	힙
쓰레드별 실행 컨텍스트	스택	JVM 스택

> JVM도 결국 운영체제 프로세스 안에서 실행되기 때문에, 자연스럽게 OS 프로세스의 메모리 구조를 따라가며 자바 특성(클래스 로딩, 가비지 컬렉션, 바이트코드 실행 등)에 맞게 추상화 한 것.

자바 메모리 구조



- **메서드 영역(Method Area)**: 메서드 영역은 프로그램을 실행하는데 필요한 공통 데이터를 관리한다. 이 영역은 프로그램의 모든 영역에서 공유한다.
  - 클래스 정보: 클래스의 실행 코드(바이트 코드), 필드, 메서드와 생성자 코드등 모든 실행 코드가 존재한다.
  - static 영역: `static` 변수들을 보관한다.
  - 런타임 상수 풀: 프로그램을 실행하는데 필요한 공통 리터럴 상수를 보관한다.
- **스택 영역(Stack Area)**: 자바 실행 시, 하나의 실행 스택이 생성된다. 각 스택 프레임은 지역 변수, 중간 연산 결과, 메서드 호출 정보 등을 포함한다.
  - 스택 프레임: 스택 영역에 쌓이는 네모 박스가 하나의 스택 프레임이다. 메서드를 호출할 때 마다 하나의 스택 프레임이 쌓이고, 메서드가 종료되면 해당 스택 프레임이 제거된다.
- **힙 영역(Heap Area)**: 객체(인스턴스)와 배열이 생성되는 영역이다. 가비지 컬렉션(GC)이 이루어지는 주요 영역이며, 더 이상 참조되지 않는 객체는 GC에 의해 제거된다.

> **참고**: 스택 영역은 더 정확히는 각 스레드별로 하나의 실행 스택이 생성된다. 따라서 스레드 수 만큼 스택이 생성된다. 지금은 스레드를 1개만 사용하므로 스택도 하나이다. 이후 스레드를 추가할 것인데, 그러면 스택도 스레드 수 만큼 증가한다.

쓰레드 생성

> 자바를 시작하면 main이라는 쓰레드를 자바가 만들어서 main 쓰레드 내에서 코드를 실행.
> - 프로세스가 작동하려면 최소 하나 이상의 쓰레드가 있어야 하기 때문.
>
> 1. java MyApp 실행 → OS가 JVM 프로세스를 실행시킴.
> 2. JVM이 클래스 로더로 MyApp.class를 메모리에 로드하고, main(String[] args) 메서드를 찾음.
> 3. JVM이 **“main 쓰레드”**(user thread, non-daemon)를 하나 생성.
> 4. main 쓰레드가 main() 메서드부터 프로그램 코드를 실행하기 시작.
> 5. 필요하면 main 쓰레드가 다른 작업 쓰레드(new Thread, ExecutorService 등)를 만들어서 병렬로 실행.

> 6. 모든 **non-daemon 쓰레드**가 종료되면 JVM 프로세스도 종료됨.
>
> 👉 즉, 자바 프로그램 시작 = JVM 프로세스 실행 + main 쓰레드 생성 → main() 실행.

1. Thread 클래스를 상속 받는 방법

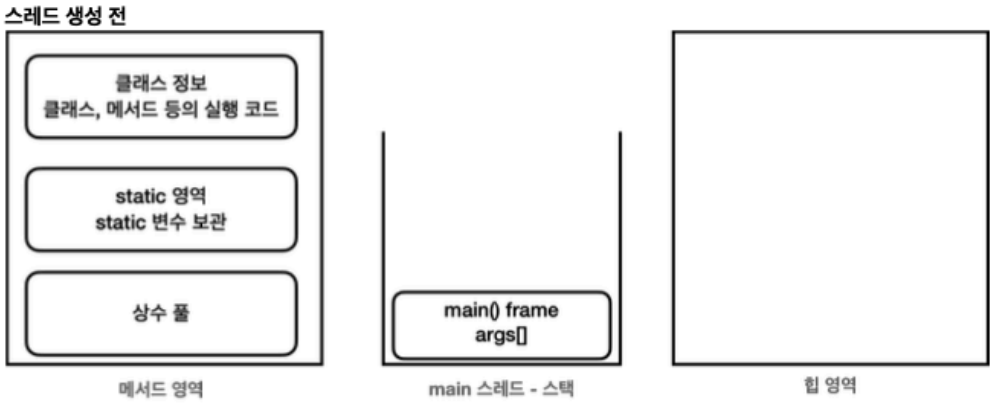
```
package thread.start;

public class HelloThread extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + ": run()");
    }
}
```

Thread 클래스를 상속하고, 쓰레드가 실행할 코드를 run() 메서드에 재정의.

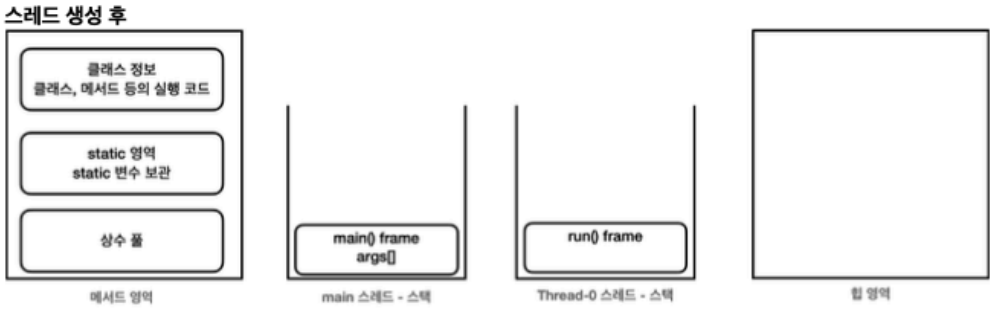
```
package thread.start;

public class HelloThreadMain {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + ": main Th
read() start");

        HelloThread helloThread = new HelloThread();
//        helloThread.run(); // 절대 아님.
        System.out.println(Thread.currentThread().getName()+": start() 호
출 전");
        helloThread.start();
        System.out.println(Thread.currentThread().getName()+": start() 호
출 후");

        System.out.println(Thread.currentThread().getName() + ": main Th
read() end");
    }
}

main: main Thread() start
main: start() 호출 전
```

```
main: start() 호출 후
main: main Thread() end
Thread-0: run()
```

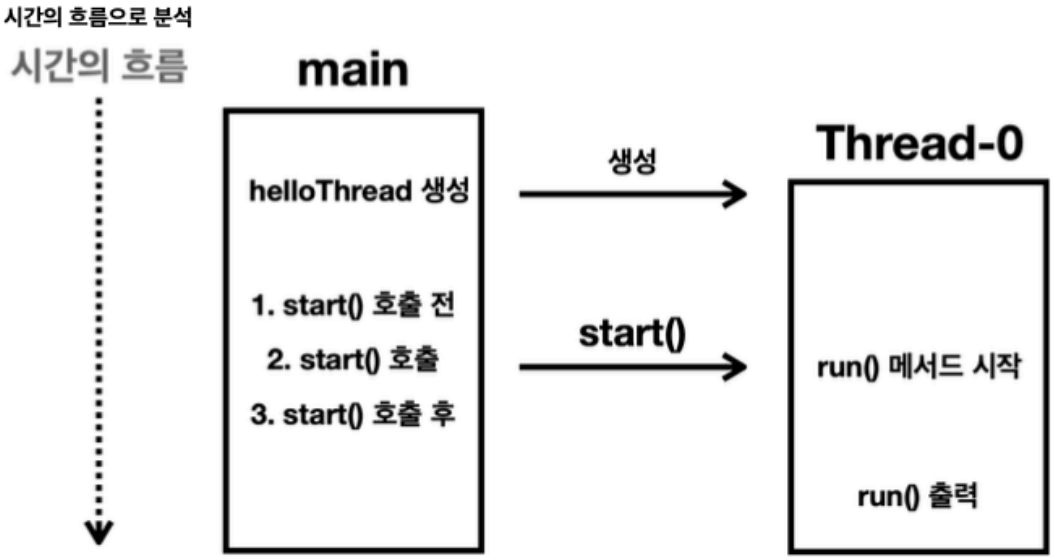
- run() 메서드를 재정의했지만, start() 메서드를 호출해줘야 함.



실행 결과를 보면 `main()` 메서드는 `main`이라는 이름의 스레드가 실행하는 것을 확인할 수 있다. 프로세스가 작동하려면 스레드가 최소한 하나는 있어야 한다. 그래야 코드를 실행할 수 있다. 자바는 실행 시점에 `main`이라는 이름의 스레드를 만들고 프로그램의 시작점인 `main()` 메서드를 실행한다.



1. HelloThread 쓰레드 객체를 생성한 다음 start()를 호출하면, 자바는 '**쓰레드를 위한 별도의 스택 공간을 할당**'
   - 쓰레드에 이름을 주지 않으면 Thread-0, Thread-1 과 같은 임의의 이름 부여
2. 새로운 Thread-0 쓰레드가 사용할 전용 스택 공간이 마련됨.
3. Thread-0 쓰레드는 run() 메서드의 스택 프레임을 스택에 올리면서 run() 메서드 시작.



- main Thread가 run() 메서드를 실행하는 것이 아님.
  - **Thread-0 쓰레드가 run() 메서드를 실행함!**

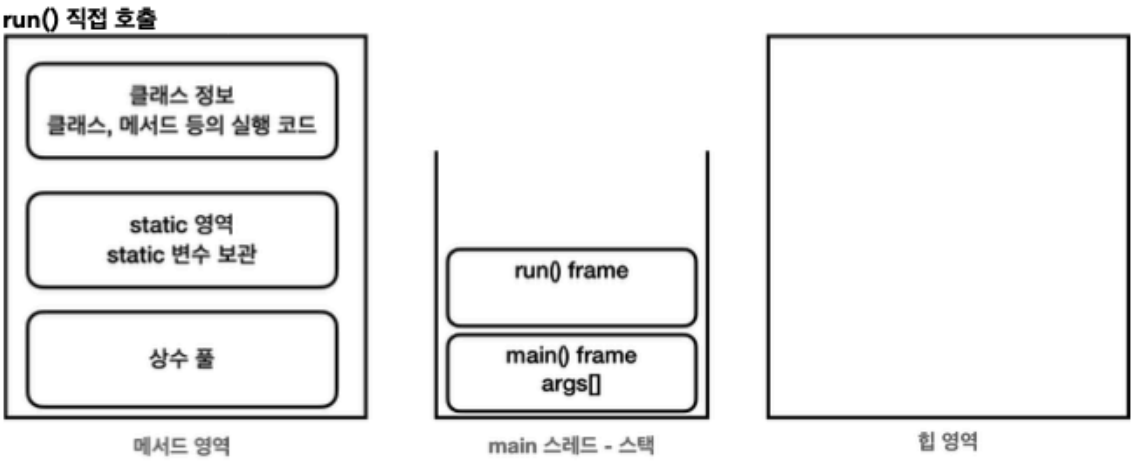
> main 쓰레드는 단지, start() 메서드를 통해 Thread-0 쓰레드에게 실행을 '**지시만 함**'
> 이 후, main 쓰레드는 그 다음 코드를 멈추지 않고 계속 수행하며 Thread-0는 CPU 코어에 스케줄링 되면 그 때 로직을 동작.
> 때문에, main 쓰레드와, Thread-0가 동시에 실행되는 것.

쓰레드 시작2

만약 start()가 아닌 run()을 통해 실행한다면 ?

```
package thread.start;

public class BadThreadMain {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + ": main Threa
d() start");

        HelloThread helloThread = new HelloThread();
        System.out.println(Thread.currentThread().getName()+": start() 호출
전");
        helloThread.run(); // 절대 아님.
        System.out.println(Thread.currentThread().getName()+": start() 호출
```

```
후");

        System.out.println(Thread.currentThread().getName() + ": main Threa
d() end");
    }
}

main: main Thread() start
main: start() 호출 전
main: run()
main: start() 호출 후
main: main Thread() end
```



- 실행 결과를 잘 보면 별도의 스레드가 `run()`을 실행하는 것이 아니라, `main` 스레드가 `run()` 메서드를 호출할 것을 확인할 수 있다.
- 자바를 처음 실행하면 `main` 스레드가 `main()` 메서드를 호출하면서 시작한다.
- `main` 스레드는 `HelloThread` 인스턴스에 있는 `run()` 이라는 메서드를 호출한다.
- `main` 스레드가 `run()` 메서드를 실행했기 때문에 `main` 스레드가 사용하는 스택위에 `run()` 스택 프레임이 올라간다.

- **start() 메서드는 쓰레드에 스택 공간을 할당하면서 쓰레드를 시작하는 특별한 메서드인 것.**
- 쓰레드를 생성한다 = 단순히 객체만 new 하는 게 아니라, OS 차원에서 실행 컨텍스트를 부여하는 것
    - **OS 차원에서 새로운 실행 단위를 만들고, 그에 필요한 스택 메모리와 컨텍스트를 할당한다**는 뜻

- new Thread() : 단순히 자바 객체 (Thread 인스턴스)를 만드는 것.
  - 아직 OS 레벨의 쓰레드는 만들어지지 않음.
- start() 호출 : JVM이 네이티브 메서드를 통해 OS에 요청
  - JVM은 운영체제 위에서 하나의 프로세스로 실행되며, '내 프로세스 안에서 실행할 새로운 쓰레드를 하나 만들고, 스택 공간과 실행 컨텍스트를 초기화 해줘' 라고 요청하는 것.
  - OS가 새로운 커널 쓰레드를 생성
  - 해당 쓰레드를 위한 독립적인 스택 메모리 공간이 할당.
  - PC(Program Counter), 레지스터, 스택 포인터 등 실행 컨텍스트 초기화
- run() 호출 : 자바 레벨의 '실행 흐름' 생성.
  - 각 쓰레드는 자기만의 스택을 가지며, 힙과 메서드 영역은 공유

> **run() 직접 호출**
>
> - 단순히 일반 메서드 호출일 뿐.
> - 새로운 쓰레드가 만들어지지 않고 **현재 쓰레드(main 쓰레드)**에서 실행됨.
>
> **start() 호출**
>
> - JVM이 OS에 요청해서 **새로운 스레드(예: Thread-0)를 생성**.
> - 새로 만들어진 스레드 내부에서 run()이 실행됨.
> - 따라서 병렬 실행이 가능.
>
> 👉 핵심 차이점:
>
> - start() → **새로운 실행 흐름을 만든다 (스레드 생성 + run() 실행)**
> - run() → **그냥 일반 메서드 실행, 현재 쓰레드에서 실행**
>
> start() 메서드는 쓰레드에 스택 공간을 할당하면서 쓰레드를 시작하는 특별한 메서드인 것.

데몬 쓰레드

> 쓰레드는 사용자(user) 쓰레드와 데몬(daemon) 쓰레드 두 가지 종류로 구분 가능.
> main Thread 또한 user 쓰레드.

사용자 쓰레드(non-daemon 쓰레드)

- 프로그램의 주요 작업을 수행
- 작업이 완료될 때 까지 실행
- **모든 user 쓰레드가 종료되면 JVM도 종료**
    - main Thread 또한 user 쓰레드.

데몬 쓰레드

사용자에게 직접적으로 보이지 않으면서, 시스템의 백그라운드에서 작업을 수행하는 것 = 데몬 쓰레드, 데몬 프로세스 Ex) 사용하지 않는 파일이나, 메모리 정리 등.

- 백그라운드에서 보조적인 작업을 수행
- 모든 user 쓰레드가 종료되면, 데몬 쓰레드는 자동으로 종료

> JVM은 데몬 쓰레드의 실행 완료를 기다리지 않고 종료됨. 즉, **데몬 쓰레드가 아닌 모든 쓰레드가 종료되면, 자바 프로그램도 종료.**
> 즉, JVM의 종료는 main 메써드의 종료가 아닌, '모든' user 쓰레드의 종료 시점.

1. **데몬 스레드의 주 역할**
    - GC, JIT 컴파일, 모니터링 등 **백그라운드 보조 작업**.
    - 이들은 애플리케이션 로직을 직접 실행하는 게 아니라, **유저 스레드가 잘 실행되도록 돕는 역할**.
2. **JVM 종료 시 동작**
    - 모든 **User Thread**가 끝났다는 건 → "프로그램의 주요 로직이 모두 종료됨"을 의미.
    - 이 시점에서는 더 이상 GC나 모니터링 같은 작업을 할 필요가 없음. (메모리도 반납될 거라서)
    - 그래서 데몬 스레드만 남아 있어도 JVM은 강제로 종료.
3. **정리**
    - User Thread: **프로그램의 본질적인 작업 담당 → 종료 시점까지 보장 필요**
    - Daemon Thread: **보조적 성격 → User Thread 종료 후엔 의미 없음 → JVM도 종료 가능**

```
package thread.start;
```

```
public class DaemonThreadMain {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + ": main() star
t");
        DaemonThread daemonThread = new DaemonThread();
        daemonThread.setDaemon(true); // 데몬 쓰레드 여부
        daemonThread.start();

        System.out.println(Thread.currentThread().getName() + ": main() en
d");
    }

    static class DaemonThread extends Thread {
        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName() + ": run()");
            try {
                Thread.sleep(10000); // 10초간 실행되는 어떤 쓰레드
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

프로그램 실행 시 바로 종료.

JVM의 종료 시점은, 모든 ‘user Thread’의 종료 시점이기 때문

> Springboot와 같은 웹서버는 톰캣이 User Thread풀을 만들어서 요청 대기 상태로 유지하고 있음.
> 즉, User Thread Pool이 JVM 프로세스를 종료하지 못하도록 막고 있는 것.

쓰레드 생성 - Runnable

쓰레드를 만들 때는 Thread 클래스를 상속받는 것과 Runnable 인터페이스를 구현하는 방법이 있음.

```
package thread.start;
```

```
public class HelloRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + ": run()");
    }
}

package thread.start;

public class HelloRunnableMain {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + ": main() star
t");

        HelloRunnable runnable = new HelloRunnable();
        Thread thread = new Thread(runnable);
//        Thread thread = new Thread(() -> System.out.println(Thread.current
Thread().getName()));
        thread.start();

        System.out.println(Thread.currentThread().getName() + ": main() en
d");
    }
}
```

실행 결과는 기존과 같으나, 쓰레드와 해당 쓰레드가 실행할 작업이 분리되어 있음.

- 쓰레드 객체를 생성할 때, 실행할 작업을 생성자로 전달하면 됨.

Thread 상속 vs Runnable 구현

쓰레드 사용시에는 Thread 상속보다, **Runnable 인터페이스를 구현하는 방식을 사용하는 게 좋음.**

Thread 클래스 상속 방식

장점

간단한 구현: `Thread` 클래스를 상속받아 `run()` 메서드만 재정의하면 된다.
단점

상속의 제한: 자바는 단일 상속만을 허용하므로 이미 다른 클래스를 상속받고 있는 경우 `Thread` 클래스를 상속
받을 수 없다.
유연성 부족: 인터페이스를 사용하는 방법에 비해 유연성이 떨어진다.

Runnable 인터페이스를 구현 방식
장점

상속의 자유로움: `Runnable` 인터페이스 방식은 다른 클래스를 상속받아도 문제없이 구현할 수 있다.
코드의 분리: 스레드와 실행할 작업을 분리하여 코드의 가독성을 높일 수 있다.
여러 스레드가 동일한 `Runnable` 객체를 공유할 수 있어 자원 관리를 효율적으로 할 수 있다.

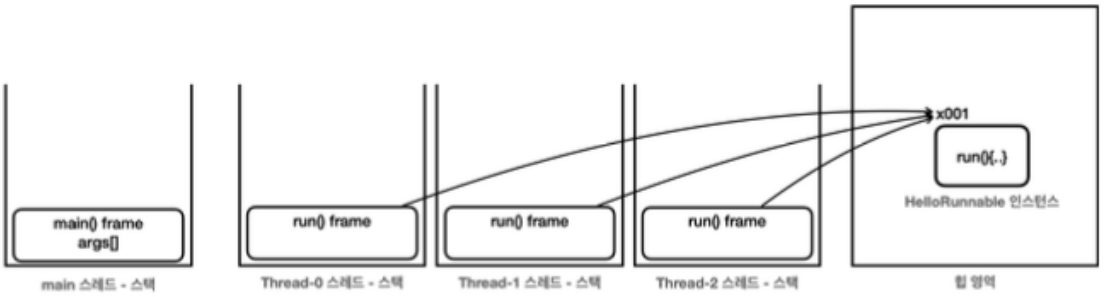
단점
코드가 약간 복잡해질 수 있다. `Runnable` 객체를 생성하고 이를 `Thread` 에 전달하는 과정이 추가된다.

여러 쓰레드 만들기

많은 쓰레드를 한 번에 만드는 방법

```
package thread.start;

import util.MyLoggerThread;

import static util.MyLoggerThread.*;

public class ManyThreadMainV1 {
    public static void main(String[] args) {
        log("main() start");

        HelloRunnable runnable = new HelloRunnable();
        Thread thread1 = new Thread(runnable);
        thread1.start();
        Thread thread2 = new Thread(runnable);
        thread2.start();
        Thread thread3 = new Thread(runnable);
        thread3.start();

        log("main() end");
```

```
    }
}
```



- 스레드3개를 생성할 때 모두 같은 `HelloRunnable` 인스턴스(`x001`)를 스레드의 실행 작업으로 전달했다.
- `Thread-0`, `Thread-1`, `Thread-2`는 모두 `HelloRunnable` 인스턴스에 있는 `run()` 메서드를 실행한다.

Runnable을 만드는 다양한 방법

1. 중첩 클래스 사용

```
package thread.start;

import util.MyLoggerThread;

import static util.MyLoggerThread.*;

public class InnerRunnableMainV1 {
    public static void main(String[] args) {
        log("main() start");

        MyRunnable runnable = new MyRunnable();
        Thread thread = new Thread(runnable);
        thread.start();

        log("main() end");
    }

    static class MyRunnable implements Runnable {
        @Override
        public void run() {
```

```
            log("run()");
        }
    }
}
```

2. 익명 클래스 사용

```
package thread.start;

import static util.MyLoggerThread.log;

public class InnerRunnableMainV2 {
    public static void main(String[] args) {
        log("main() start");

        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                log("run()");
            }
        };
        Thread thread = new Thread(runnable);
        thread.start();

        log("main() end");
    }
}
```

3. 익명 클래스 변수 없이 직접 전달

```
package thread.start;

import static util.MyLoggerThread.log;

public class InnerRunnableMainV3 {
    public static void main(String[] args) {
```

```
        log("main() start");

        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                log("run()");
            }
        });
        thread.start();

        log("main() end");
    }
}
```

4. 람다

```
package thread.start;

import static util.MyLoggerThread.log;

public class InnerRunnableMainV3 {
    public static void main(String[] args) {
        log("main() start");

        Thread thread = new Thread(() -> log("run()"));
        thread.start();

        log("main() end");
    }
}
```