


# 11장. 스레드

한림대학교 소프트웨어융합대학 양은샘.

	<b>이것이 자바다</b> 신용권의 Java 프로그래밍 정복	<b>혼자 공부하는 자바</b> JAVA 8 & 11 지원/무료 동영상 강의 제공	
저자	신용권	저자	신용권
출판	한빛미디어   2015.1.5	출판	한빛미디어   2019.6.10.
페이지수	1,224	페이지수	708
사이즈	183*235mm	사이즈	188*257mm
판매가	서적 27,000원	판매가	서적 21,600원
구매이벤트	IT독자 설문이벤트 외 6건		

# 11장. 스레드(Thread)

---

- ❖ 안녕하세요? 여러분!
- ❖ 오늘은 자바의 스레드 단원을 학습 합니다.
- ❖ 이번 장에서는
  - 스레드의 생성과 실행 및 스레드를 이용한 여러 가지 유용한 작업환경을 만드는 방법에 대해 알아보도록 하겠습니다.
  - 스레드를 이용한 작업환경 구성은 동시에 다양한 처리를 할 수 있는 장점을 제공합니다.
- ❖ 지난 시간에 학습한 내용을 리뷰한 후 학습을 시작하도록 하겠습니다.

# 지난 시간 Review

---

1절. 자바 API 문서

2절. java.lang & java.util 패키지

3절. Object 클래스 / java.lang 패키지

4절. System 클래스 / java.lang 패키지

5절. Class 클래스 / java.lang 패키지

6절. String 클래스 / java.lang 패키지

7절. StringBuffer, StringBuilder 클래스 / java.lang 패키지

8절. Wrapper 관련 클래스 / java.lang 패키지

9절. Math 클래스 / java.lang 패키지

10절. Random 클래스 / java.util 패키지

11절. Arrays 클래스 / java.util 패키지

12절. Objects 클래스 / java.util 패키지

13절. StringTokenizer 클래스 / java.util 패키지

14절. Pattern 클래스 / java.util 패키지

15절. Date 클래스 / java.util 패키지

16절. Calendar 클래스 / java.util 패키지

17절. Format 클래스 / java.text 패키지

18절. java.time 패키지

# 학습 목차

---

1절. Process와 Thread

2절. 작업 스레드 생성과 실행

3절. 스레드 스케줄링

4절. 스레드 동기화

5절. 스레드 상태

# 학습 목표

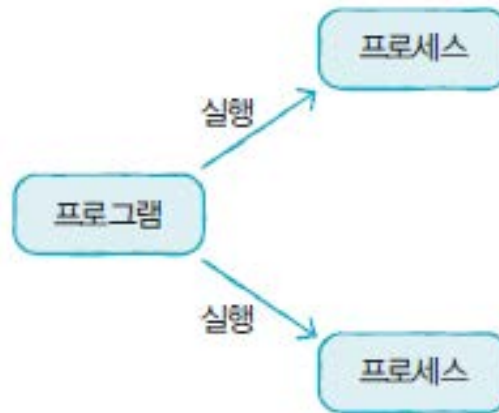
---

- ❖ Process와 Thread의 차이점을 안다.
- ❖ 작업 스레드를 생성하고 실행하는 방법을 안다.
- ❖ 스레드 스케줄링을 이용하여 스레드에 우선순위를 부여할 수 있다.
- ❖ 스레드 동기화 작업을 통해 여러 스레드가 공유하는 객체나 메소드의 임계영역을 관리할 수 있다.
- ❖ 스레드의 상태를 이용하여 다중 스레드 상황에서의 문제를 해결할 수 있다.

# 1절. Process와 Thread

## ❖ 프로세스(process)

- 실행 중인 하나의 애플리케이션 프로그램
- 애플리케이션이 실행되면 운영체제로부터 실행에 필요한 메모리 할당 받아 코드를 실행 함
- 하나의 프로그램이 다중 프로세스를 만들기도 함



이름	상태	9% CPU	24% 메모리
애플리케이션 (3)			
> 메모장		0.4%	2.2MB
> 메모장		0.1%	2.2MB
> 작업 관리자		1.3%	34.2MB
백그라운드 프로세스 (84)			
AcroTray(32비트)		0%	1.3MB
Activation Licensing Service(32...		0%	1.6MB
Adobe Acrobat Update Service(...		0%	0.9MB

## ❖ 스레드(thread)

- 한 가지 작업을 실행하기 위해 순차적으로 실행할 코드를 이어놓은 것
- 하나의 스레드는 하나의 코드 실행 이름

# multi tasking

## ❖ 멀티 태스킹(multi tasking)

- 두 가지 이상의 작업을 동시에 처리하는 것

## ❖ 멀티 프로세스(multi process)

- 독립적으로 프로그램들을 실행하고 여러 가지 작업 처리

## ❖ 멀티 스레드(multi thread)

- 하나의 프로세스로 두 가지 이상의 작업을 처리
- 데이터를 분할하여 병렬로 처리하거나, 다수의 클라이언트 요청을 처리하는 서버 등의 용도로 사용
- 한 스레드가 예외 발생시킬 경우 프로세스 자체가 종료될 수 있음



# main 스레드

## ❖ main 스레드

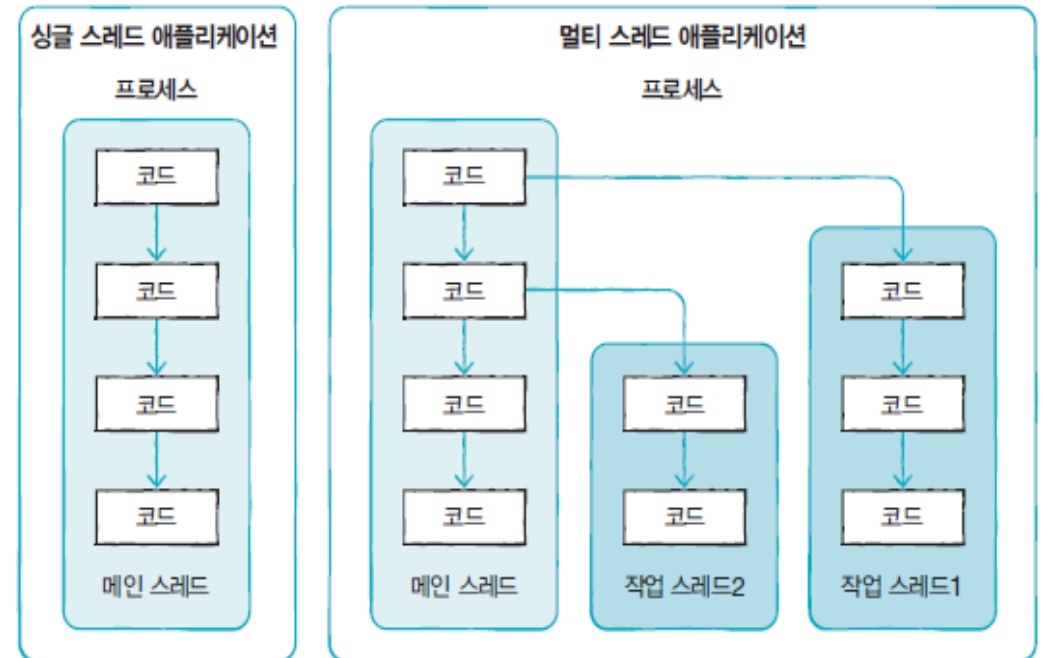
- 모든 자바 애플리케이션은 메인 스레드가 main() 메소드를 실행하면서 시작됨
- main() 메소드의 첫 코드부터 아래로 순차적으로 실행
- 필요에 따라 작업 스레드들 만들어 병렬로 코드 실행 가능(멀티 스레드를 생성해 멀티 태스킹 수행)
- 멀티 스레드 애플리케이션에서는 실행 중인 스레드가 하나라도 있으면 프로세스는 종료되지 않음

```
public static void main(String[] args) {  
    String data = null;  
    if(...) {  
    }  
    while(...) {  
    }  
    System.out.println("...");  
}
```

코드의 실행 흐름 → 스레드

## ❖ 작업 스레드

- 작업 스레드 역시 객체로 생성되므로 클래스 필요





## 2절. 작업 스레드 생성과 실행

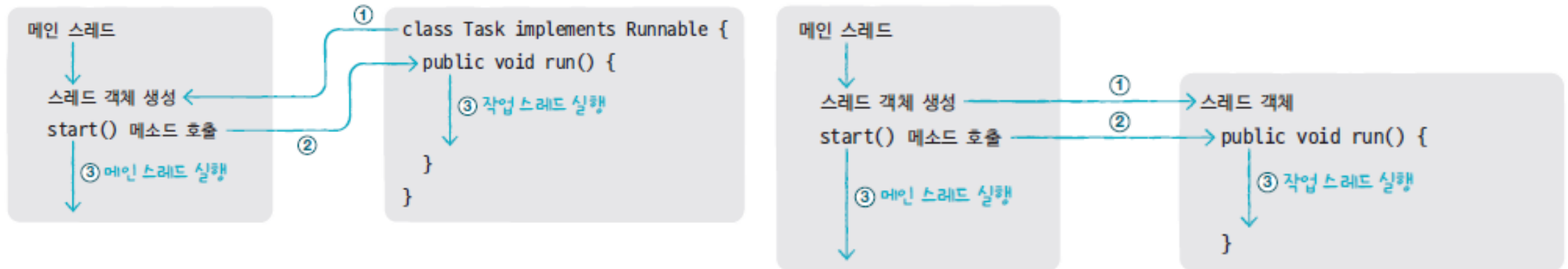
### ❖ Thread 클래스로부터 직접 생성

- 방법 1) Runnable 인터페이스를 구현한 객체를 매개 값으로 Thread 생성자 호출
- 방법 2) Thread 클래스를 상속 받은 하위 클래스의 생성자 호출

### ❖ 스레드 실행

- 작업 스레드 객체 생성 후 start() 메소드를 호출하면 run() 메소드가 실행 됨

```
thread.start();
```



# 스레드 생성 : Runnable 구현 이용

- ❖ Runnable 인터페이스를 구현한 클래스 작성

```
class Task implements Runnable {  
    public void run() {  
        스레드가 실행할 코드;  
    }  
}
```

- ❖ 구현 객체를 매개 값으로 Thread 생성자를 호출하면 작업 스레드가 생성됨

```
Runnable task= new Task();  
Thread thread = new Thread(task);
```

- ❖ Runnable 익명 객체를 매개 값으로 사용하여 Thread 생성자를 호출할 수도 있음

```
Thread thread = new Thread( new Runnable() {  
    public void run() {  
        스레드가 실행할 코드;  
    }  
});
```

← 익명 구현 객체

# 스레드 생성 : Thread 상속 이용

- ❖ Thread의 하위 클래스로 작업 스레드를 정의

```
public class WorkerThread extends Thread {  
    @Override  
    public void run() {  
        스레드가 실행할 코드;  
    }  
}
```

← run() 메소드 재정의

- ❖ 정의된 작업 스레드의 생성자를 호출하면 작업 스레드가 생성됨

```
Thread thread = new WorkerThread();
```

- ❖ 익명 하위 객체를 사용하여 작업 스레드를 정의할 수도 있음

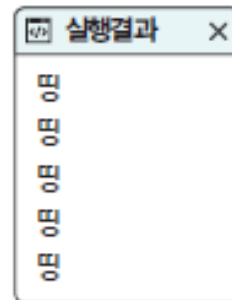
```
Thread thread = new Thread() {  
    public void run() {  
        스레드가 실행할 코드;  
    }  
};
```

← 익명 자식 객체

# 스레드 예 : beep()

## ❖ 메인 스레드만 이용한 경우

```
01 package sec01.exam01;
02
03 import java.awt.Toolkit;
04
05 public class BeepPrintExample1 {
06     public static void main(String[] args) {
07         Toolkit toolkit = Toolkit.getDefaultToolkit(); ← Toolkit 객체 얻기
08         for(int i=0; i<5; i++) {
09             toolkit.beep(); ← 비프음 발생
10             try { Thread.sleep(500); } catch(Exception e) {}
11         } ↑ 0.5초간 일시 정지
12
13         for(int i=0; i<5; i++) {
14             System.out.println("땡");
15             try { Thread.sleep(500); } catch(Exception e) {}
16         } ↑ 0.5초간 일시 정지
17     }
18 }
```



# Runnable 구현 스레드 예 : beep()

- ❖ main 스레드와 작업 스레드가 동시에 실행
  - Runnable 구현 클래스 : 비프음을 들려주는 작업 정의

```
01 package sec01.exam02;
02
03 public class BeepPrintExample2 {
04     public static void main(String[] args) {
05         Runnable beepTask = new BeepTask();
06         Thread thread = new Thread(beepTask);
07         thread.start();
08
09         for(int i=0; i<5; i++) {
10             System.out.println("땡");
11             try { Thread.sleep(500); }
12                 catch(Exception e) {}
13         }
14     }
15 }
```

```
01 package sec01.exam02;
02
03 import java.awt.Toolkit;
04
05 public class BeepTask implements Runnable {
06     public void run() {
07         Toolkit toolkit = Toolkit.getDefaultToolkit();
08         for(int i=0; i<5; i++) {
09             toolkit.beep();
10             try { Thread.sleep(500); } catch(Exception e) {}
11         }
12     }
13 }
```

← 스레드 실행 내용

# Thread 상속 예 : beep()

- ❖ main 스레드와 작업 스레드가 동시에 실행
  - Thread 상속 클래스 : 비프음을 들려주는 작업 정의

```
01 package sec01.exam04;
02
03 public class BeepPrintExample4 {
04     public static void main(String[] args) {
05         Thread thread = new BeepThread();
06         thread.start();
07
08         for(int i=0; i<5; i++) {
09             System.out.println("땡");
10             try { Thread.sleep(500); }
11                 catch(Exception e) {}
12         }
13     }
14 }
```

메인 스레드

BeepThread

```
01 package sec01.exam04;
02
03 import java.awt.Toolkit;
04
05 public class BeepThread extends Thread {
06     @Override
07     public void run() {
08         Toolkit toolkit = Toolkit.getDefaultToolkit();
09         for(int i=0; i<5; i++) {
10             toolkit.beep();
11             try { Thread.sleep(500); } catch(Exception e) {}
12         }
13     }
14 }
```

스레드 실행 내용

# 스레드 이름

---

- ❖ 메인 스레드 이름 : main
- ❖ 작업 스레드 이름 (자동 설정)
  - Thread-n
- ❖ 작업 스레드 이름 가져오기

```
thread.getName();
```
- ❖ 작업 스레드 이름 변경

```
thread.setName("스레드 이름");
```
- ❖ 코드를 실행하는 현재 스레드 객체의 참조 얻기

```
Thread thread = Thread.currentThread();
```

# 스레드 이름 예

```
01 package sec01.exam06;
02
03 public class ThreadA extends Thread {
04     public ThreadA() {
05         setName("ThreadA"); ← 스레드 이름 설정
06     }
07
08     public void run() {
09         for(int i=0; i<2; i++) {
10             System.out.println(getName() + "가 출력한 내용"); ← ThreadA 실행 내용
11         }
12     }
13 }
```

스레드 이름 얻기

```
01 package sec01.exam06;
02
03 public class ThreadB extends Thread {
04     public void run() {
05         for(int i=0; i<2; i++) {
06             System.out.println(getName() + "가 출력한 내용"); ← ThreadB 실행 내용
07         }
08     }
09 }
```

스레드 이름 얻기

```
01 package sec01.exam06;
02
03 public class ThreadNameExample {
04     public static void main(String[] args) {
05         Thread mainThread = Thread.currentThread(); ← 이 코드를 실행하는 스레드 객체 얻기
06         System.out.println("프로그램 시작 스레드 이름: " + mainThread.getName());
07
08         ThreadA threadA = new ThreadA(); ← ThreadA 생성
09         System.out.println("작업 스레드 이름: " + threadA.getName()); ← 스레드 이름 얻기
10         threadA.start(); ← ThreadA 시작
11
12         ThreadB threadB = new ThreadB(); ← ThreadB 생성
13         System.out.println("작업 스레드 이름: " + threadB.getName()); ← 스레드 이름 얻기
14         threadB.start(); ← ThreadB 시작
15     }
16 }
```

실행결과

```
프로그램 시작 스레드 이름: main
작업 스레드 이름: ThreadA
ThreadA가 출력한 내용
ThreadA가 출력한 내용
작업 스레드 이름: Thread-1
Thread-1가 출력한 내용
Thread-1가 출력한 내용
```

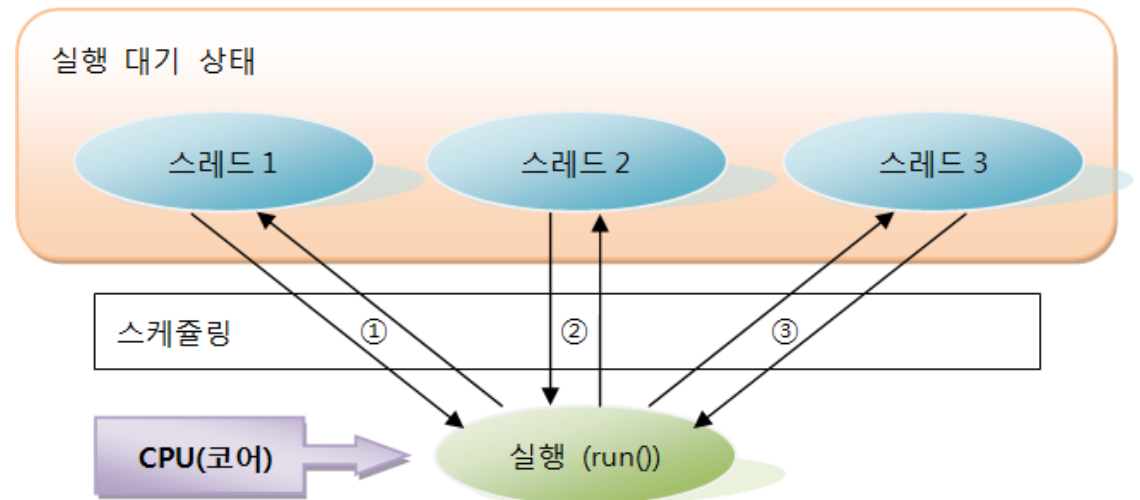


# 3절. 스레드 스케줄링

- ❖ 스레드의 개수가 코어의 수보다 많을 경우
  - 스레드를 어떤 순서로 동시성으로 실행할 것인가 결정 : 스레드 스케줄링
  - 스케줄링 의해 스레드들은 번갈아 가며 run() 메소드를 조금씩 실행

- ❖ 스레드 스케줄링

- 우선 순위 방식 (코드로 제어 가능)
  - 우선 순위는 1~10까지 값을 가질 수 있으며 기본은 5
  - 우선 순위가 높은 스레드가 실행 상태를 더 많이 가지도록 스케줄링
- 순환 할당 방식 (코드로 제어할 수 없음)
  - 시간 할당량(Time Slice)을 정해서 하나의 스레드를 정해진 시간만큼 실행



# 우선 순위 방식 예

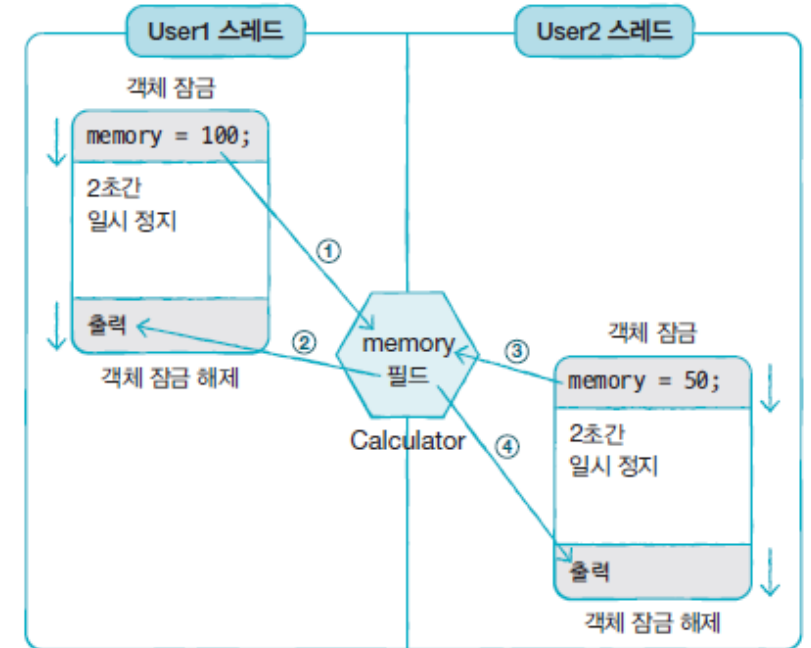
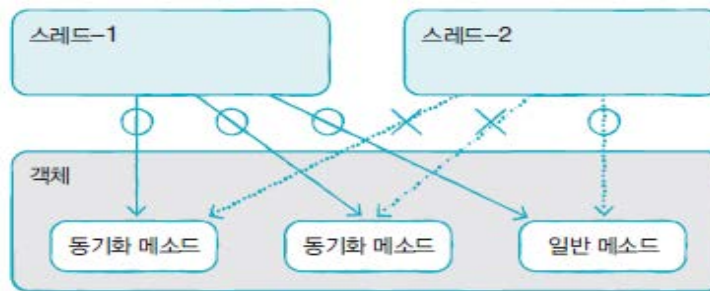
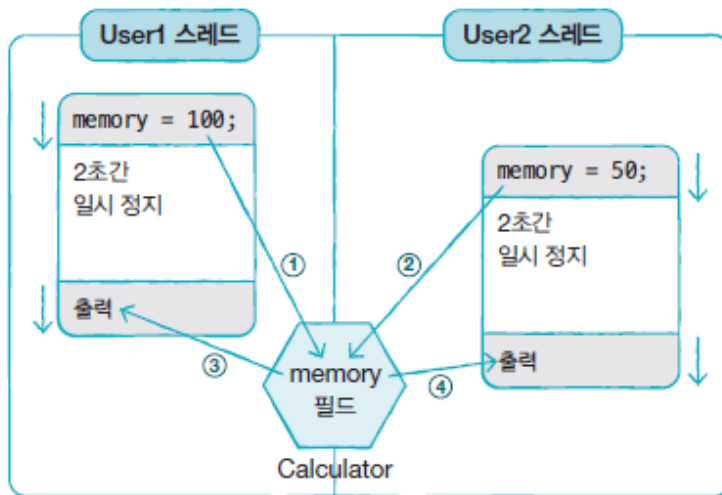
```
public class CalcThread extends Thread {  
    public CalcThread(String name) {  
        setName(name);  
    }  
    public void run() {  
        for(int i=0; i<2000000000; i++) { }  
        System.out.println(getName());  
    }  
}
```

```
public class PriorityExample {  
    public static void main(String[] args) {  
        for(int i=1; i<=10; i++) {  
            Thread thread = new CalcThread("thread" + i);  
  
            if(i != 10) {  
                thread.setPriority(Thread.MIN_PRIORITY);  
            } else {  
                thread.setPriority(Thread.MAX_PRIORITY);  
            }  
            thread.start();  
        }  
    }  
}
```

# 4절. 스레드 동기화

## ❖ 공유 객체를 사용할 때 주의할 점

- 멀티 스레드 프로그램에서 스레드들이 특정 영역을 공유해서 작업해야 하는 경우 의도했던 것과 다른 결과가 나올 수 있음



# 동기화 메소드와 동기화 블록

## ❖ 동기화 메소드 및 동기화 블록

- 임계 영역 (critical section) : 단 하나의 스레드만 실행할 수 있는 메소드 또는 블록
- synchronized 설정
  - 스레드가 객체 내부의 동기화 메소드를 실행하면 즉시 객체에 잠금이 걸림
  - 다른 스레드는 메소드나 블록의 실행이 끝날 때까지 대기해야 함

## ❖ 동기화 메소드

```
public synchronized void method() {  
    임계 영역; //단 하나의 스레드만 실행  
}
```

## 동기화 블록

```
public void method () {  
    //여러 스레드가 실행 가능 영역  
    ...  
    synchronized(공유객체) {  
        임계 영역 //단 하나의 스레드만 실행  
    }  
    //여러 스레드가 실행 가능 영역  
    ...  
}
```

# 공유 객체 예

```
03 public class MainThreadExample {
04     public static void main(String[] args) {
05         Calculator calculator = new Calculator();
06
07         User1 user1 = new User1(); ← User1 스레드 생성
08         user1.setCalculator(calculator); ← 공유 객체 설정
09         user1.start(); ← User1 스레드 시작
10
11         User2 user2 = new User2(); ← User2 스레드 생성
12         user2.setCalculator(calculator); ← 공유 객체 설정
13         user2.start(); ← User2 스레드 시작
14     }
15 }
```

실행결과

User1: 50
User2: 50

```
03 public class Calculator {
04     private int memory;
05
06     public int getMemory() {
07         return memory;
08     }
09
10     public void setMemory(int memory) { ← 계산기 메모리에 값을 저장하는 메소드
11         this.memory = memory; ← 매개값을 memory 필드에 저장
12         try {
13             Thread.sleep(2000); ← 스레드를 2초간 일시 정지시킴
14         } catch (InterruptedException e) {}
15         System.out.println(Thread.currentThread().getName() + ": " + this.memory);
16     }
17 }
```

```
03 public class User1 extends Thread {
04     private Calculator calculator;
05
06     public void setCalculator(Calculator calculator) {
07         this.setName("User1"); ← 스레드 이름을 User1로 설정
08         this.calculator = calculator; ← 공유 객체인 Calculator를 필드에 저장
09     }
10
11     public void run() {
12         calculator.setMemory(100); ← 공유 객체인 Calculator의
13     }                                     메모리에 100을 저장
14 }
```

```
03 public class User2 extends Thread {
04     private Calculator calculator;
05
06     public void setCalculator(Calculator calculator) {
07         this.setName("User2"); ← 스레드 이름을 User2로 설정
08         this.calculator = calculator; ← 공유 객체인 Calculator를 필드에 저장
09     }
10
11     public void run() {
12         calculator.setMemory(50); ← 공유 객체인 Calculator의
13     }                                     메모리에 50을 저장
14 }
```

# 동기화 메소드 예

```
03 public class Calculator {
04     private int memory;
05
06     public int getMemory() {
07         return memory;
08     }
09
10     public synchronized void setMemory(int memory) {
11         this.memory = memory;
12         try {
13             Thread.sleep(2000);
14         } catch (InterruptedException e) {}
15         System.out.println(Thread.currentThread().getName() + ": " + this.memory);
16     }
17 }
```

```
03 public class MainThreadExample {
04     public static void main(String[] args) {
05         Calculator calculator = new Calculator();
06
07         User1 user1 = new User1(); ← User1 스레드 생성
08         user1.setCalculator(calculator); ← 공유 객체 설정
09         user1.start(); ← User1 스레드 시작
10
11         User2 user2 = new User2(); ← User2 스레드 생성
12         user2.setCalculator(calculator); ← 공유 객체 설정
13         user2.start(); ← User2 스레드 시작
14     }
15 }
```

실행결과

User1: 100  
User2: 50

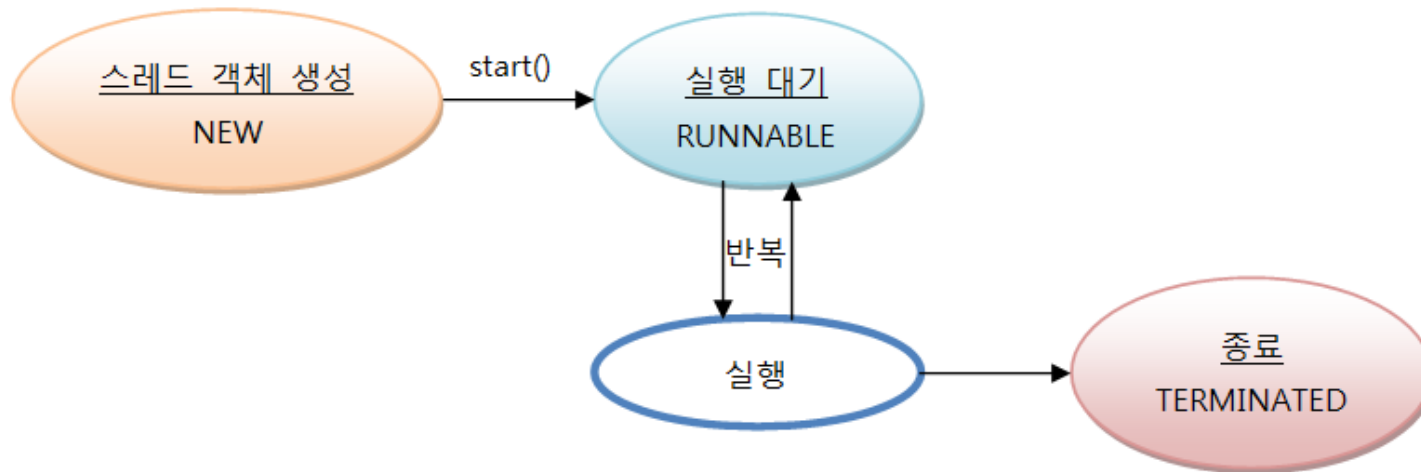
```
03 public class User1 extends Thread {
04     private Calculator calculator;
05
06     public void setCalculator(Calculator calculator) {
07         this.setName("User1"); ← 스레드 이름을 User1로 설정
08         this.calculator = calculator; ← 공유 객체인 Calculator를 필드에 저장
09     }
10
11     public void run() {
12         calculator.setMemory(100); ← 공유 객체인 Calculator의
13                                     메모리에 100을 저장
14     }
```

```
03 public class User2 extends Thread {
04     private Calculator calculator;
05
06     public void setCalculator(Calculator calculator) {
07         this.setName("User2"); ← 스레드 이름을 User2로 설정
08         this.calculator = calculator; ← 공유 객체인 Calculator를 필드에 저장
09     }
10
11     public void run() {
12         calculator.setMemory(50); ← 공유 객체인 Calculator의
13                                     메모리에 50을 저장
14     }
```

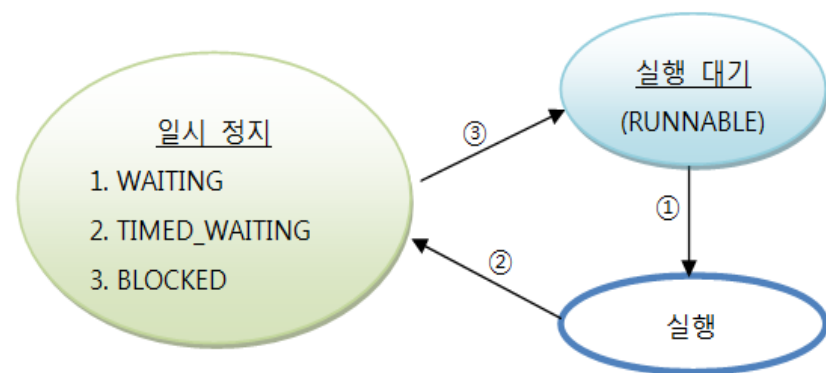
# 5절. 스레드 상태

## ❖ 스레드의 일반적인 상태

- 스레드 객체를 생성하고 start() 메소드를 호출하면 바로 실행되는 것이 아니라 실행 대기 상태가 됨
- 실행 상태의 스레드는 run() 메소드를 모두 실행하기 전 다시 실행 대기 상태로 돌아갈 수 있음
- 실행 대기 상태에 있던 다른 스레드가 선택되어 그 스레드가 실행 상태가 되기도 함
- 실행 상태에서 run() 메소드의 내용이 모두 실행되면 스레드의 실행이 멈추고 종료 상태가 됨



# 스레드 상태 변경

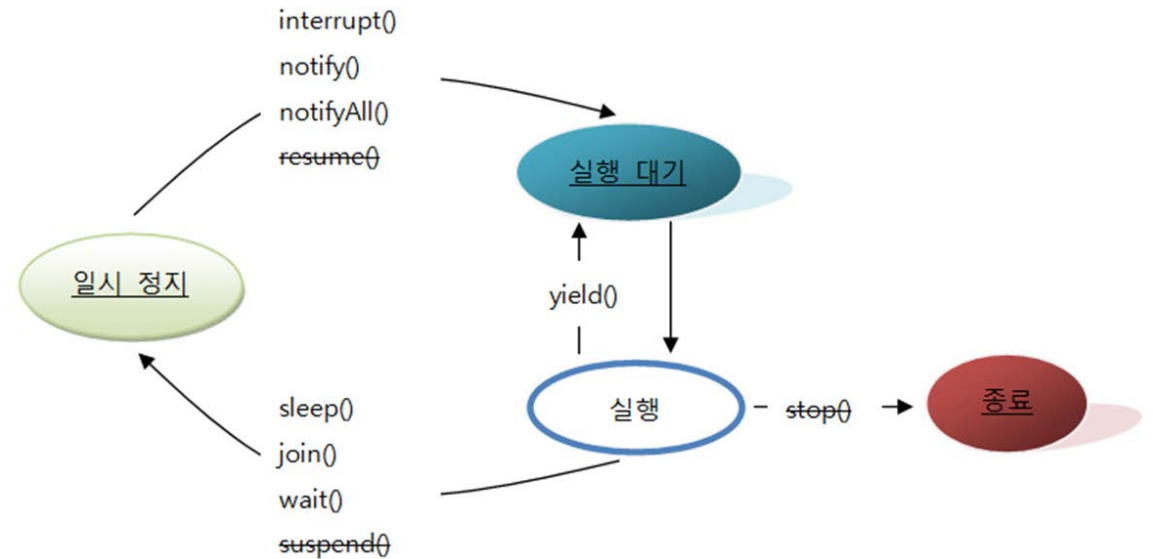


상태	열거 상수	설명
객체 생성	NEW	스레드 객체가 생성, 아직 start() 메소드가 호출되지 않은 상태
실행 대기	RUNNABLE	실행 상태로 언제든지 갈 수 있는 상태
일시 정지	BLOCKED	사용코저하는 객체의 락이 풀릴 때까지 기다리는 상태
	WAITING	다른 스레드가 통지할 때까지 기다리는 상태
	TIMED_WAITING	주어진 시간 동안 기다리는 상태
종료	TERMINATED	실행을 마친 상태



# 스레드 상태 제어 : sleep()

- ❖ 실행 중인 스레드의 상태를 변경하는 것
- ❖ 상태 변화를 가져오는 메소드의 종류 (취소선을 가진 메소드는 사용하지 않음)

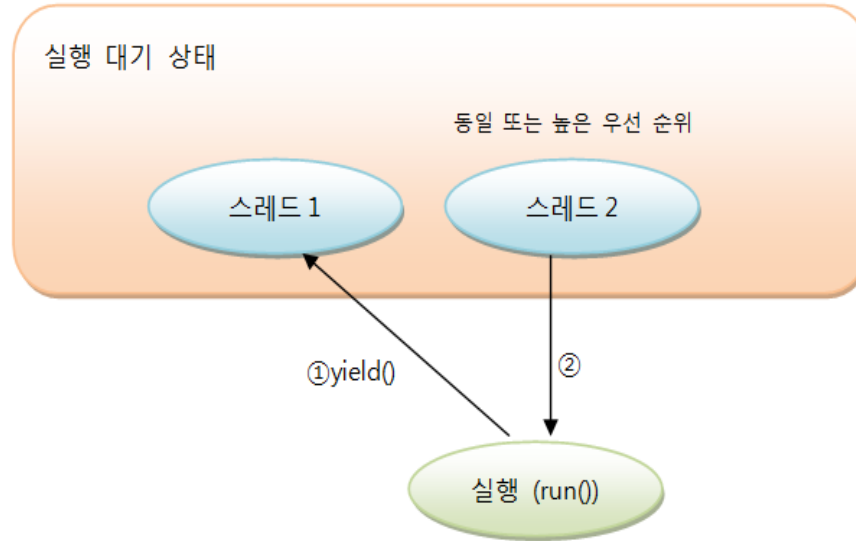


- ❖ sleep()
  - 주어진 시간 동안 일시 정지
  - 얼마 동안 일시 정지 상태로 있을 것인지 밀리 세컨드(1/1000) 단위로 지정
  - 일시 정지 상태에서 interrupt() 메소드를 호출하면 InterruptedException 발생

# 스레드 상태 제어 : yield(), join()

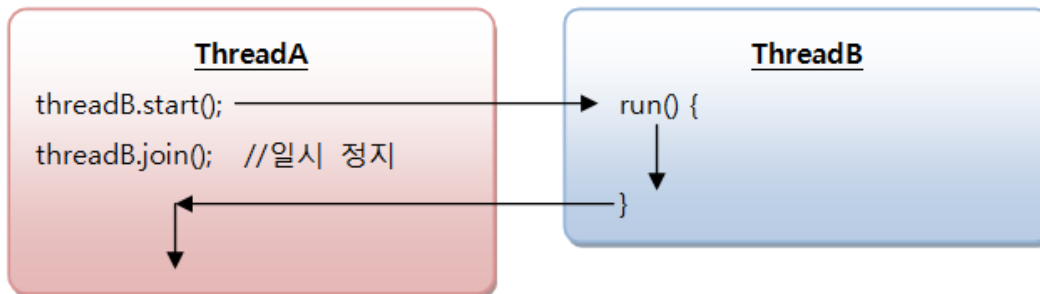
## ❖ yield()

- 다른 스레드에게 실행을 양보



## ❖ join()

- 다른 스레드의 종료를 기다림
- 계산 작업을 하는 스레드가 모든 계산 작업 마쳤을 때, 결과값을 받아 이용하는 경우 주로 사용



# yield() 예

```
//
public class ThreadA extends Thread {
    public boolean stop = false;
    public boolean work = true;

    public void run() {
        while(!stop) {
            if(work) { System.out.println("ThreadA 작업 내용"); }
            else { Thread.yield(); }
        }
        System.out.println("ThreadA 종료");
    }
}

//
public class ThreadB extends Thread {
    public boolean stop = false;
    public boolean work = true;

    public void run() {
        while(!stop) {
            if(work) { System.out.println("ThreadB 작업 내용"); }
            else { Thread.yield(); }
        }
        System.out.println("ThreadB 종료");
    }
}
```

```
//
public class YieldExample {
    public static void main(String[] args) {
        ThreadA threadA = new ThreadA();
        ThreadB threadB = new ThreadB();
        threadA.start();
        threadB.start();

        try { Thread.sleep(3000); } catch (InterruptedException e) {}
        threadA.work = false;

        try { Thread.sleep(3000); } catch (InterruptedException e) {}
        threadA.work = true;

        try { Thread.sleep(3000); } catch (InterruptedException e) {}
        threadA.stop = true;
        threadB.stop = true;
    }
}
```

# join() 예

```
//
public class SumThread extends Thread {
    private long sum;

    public long getSum() { return sum; }

    public void setSum(long sum) { this.sum = sum; }

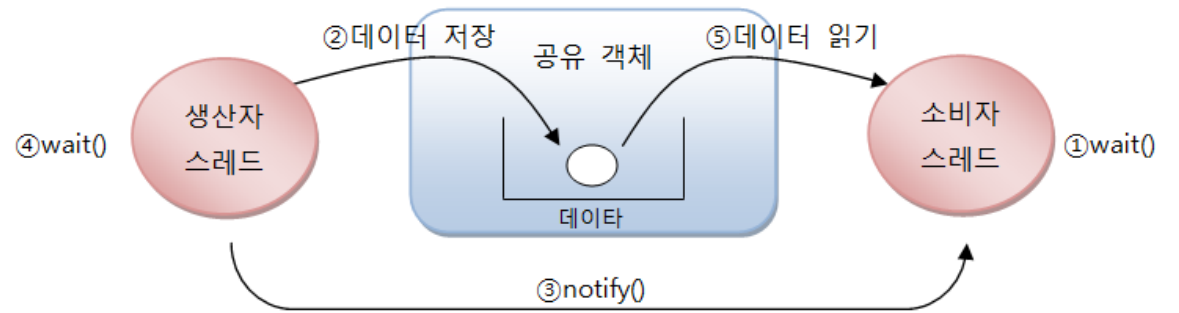
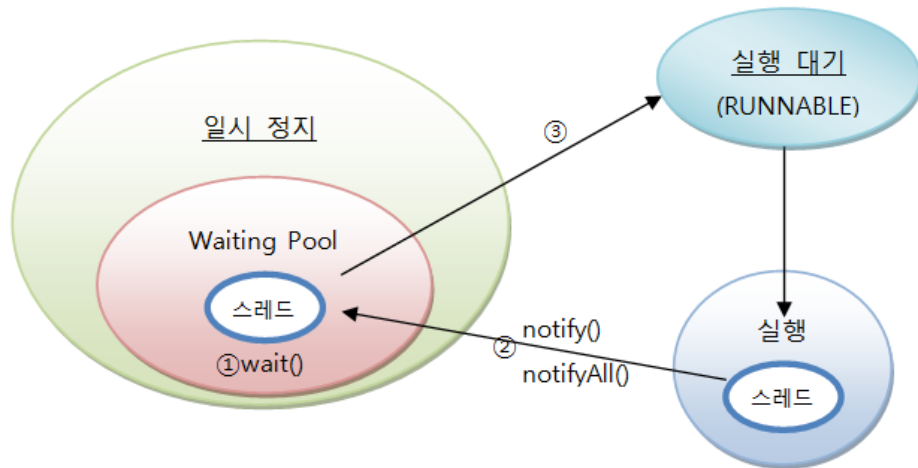
    public void run() {
        for(int i=1; i<=100; i++) { sum+=i; }
    }
}
```

```
//
public class JoinExample {
    public static void main(String[] args) {
        SumThread sumThread = new SumThread();
        sumThread.start();
        try {
            sumThread.join();
        } catch (InterruptedException e) {
        }
        System.out.println("1~100 합: " + sumThread.getSum());
    }
}
```

# 스레드 상태 제어 : wait(), notify(), notifyAll()

## ❖ wait(), notify(), notifyAll()

- 스레드 간 협업에 관련 됨
- 동기화 메소드 또는 블록에서만 호출 가능한 Object의 메소드
- 두 개의 스레드가 교대로 번갈아 가며 실행해야 할 경우 주로 사용



# wait(), notify() 예

```
//
public class DataBox {
    private String data;

    public synchronized String getData() {
        if(this.data == null) {
            try { wait(); } catch(InterruptedException e) {}
        }
        String returnValue = data;
        System.out.println("ConsumerThread가 읽은 데이터: " + returnValue);
        data = null;
        notify();
        return returnValue;
    }

    public synchronized void setData(String data) {
        if(this.data != null) {
            try { wait(); } catch(InterruptedException e) {}
        }
        this.data = data;
        System.out.println("ProducerThread가 생성한 데이터: " + data);
        notify();
    }
}

//
public class ProducerThread extends Thread {
    private DataBox dataBox;

    public ProducerThread(DataBox dataBox) { this.dataBox = dataBox; }

    @Override
    public void run() {
        for(int i=1; i<=3; i++) {
            String data = "Data-" + i;
            dataBox.setData(data);
        }
    }
}
```

```
//
public class ConsumerThread extends Thread {
    private DataBox dataBox;

    public ConsumerThread(DataBox dataBox) { this.dataBox = dataBox; }

    @Override
    public void run() {
        for(int i=1; i<=3; i++) {
            String data = dataBox.getData();
        }
    }
}

//
public class WaitNotifyExample {
    public static void main(String[] args) {
        DataBox dataBox = new DataBox();

        ProducerThread producerThread = new ProducerThread(dataBox);
        ConsumerThread consumerThread = new ConsumerThread(dataBox);
        producerThread.start();
        consumerThread.start();
    }
}
```

# 스레드 상태 제어 : stop()

## ❖ stop()

- 스레드가 즉시 종료 되는 편리함
- 사용 중이던 자원들이 불안정한 상태로 남겨 짐

## ❖ stop 플래그, interrupt()

- 실행 중인 스레드 즉시 종료해야 할 필요 있을 때 사용
- 스레드의 안전한 종료를 위해 stop 플래그, interrupt() 사용
- stop 플래그를 사용법

```
03 public class PrintThread1 extends Thread {  
04     private boolean stop;  
05  
06     public void setStop(boolean stop) {  
07         this.stop = stop;  
08     }  
09  
10     public void run() {  
11         while(!stop) {  
12             System.out.println("실행 중");  
13         }  
14         System.out.println("자원 정리");  
15         System.out.println("실행 종료");  
16     }  
17 }
```

stop이 true가 될 때

# 스레드 상태 제어 : interrupt()

- ❖ 스레드의 안전한 종료를 위해 사용
  - 실행대기 또는 실행상태에서는 InterruptedException을 발생하지 않음
  - 스레드가 미래에 일시정지 상태가 되면 InterruptedException을 발생 시킴
  - 일시 정지 상태로 만들지 않고 while문 빠져 나오는 방법으로도 쓰임

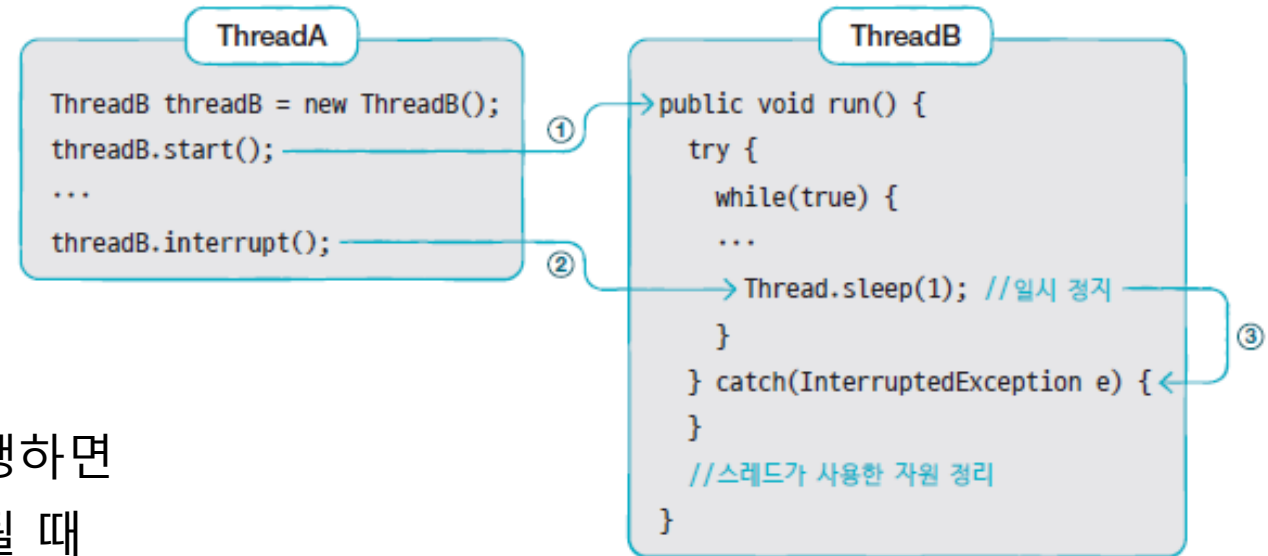
- ❖ 일시정지를 만들지 않는 경우 사용법

```
boolean status = Thread.interrupted();  
boolean status = objThread.isInterrupted();
```

- interrupt() 메소드가 호출된 경우 true 리턴

- ❖ interrupt() 메소드 사용법

- ThreadA가 ThreadB의 interrupt() 메소드를 실행하면
- ThreadB가 sleep() 메소드로 일시정지 상태가 될 때
- ThreadB에서 InterruptedException 발생, 예외 처리 블록으로 이동





# interrupt() 예

```
03 public class InterruptExample {
04     public static void main(String[] args) {
05         Thread thread = new PrintThread2();
06         thread.start();
07
08         try { Thread.sleep(1000); catch (InterruptedException e) {}
09
10         thread.interrupt(); ← 스레드를 종료하기 위해
                               InterruptedException을 발생시킴
11     }
12 }
```

실행결과

실행 중
실행 중
실행 중
자원 정리
실행 종료

```
03 public class PrintThread2 extends Thread {
04     public void run() {
05         try {
06             while(true) {
07                 System.out.println("실행 중");
08                 Thread.sleep(1); ← InterruptedException 발생
09             }
10         } catch(InterruptedException e) {} ←
11
12         System.out.println("자원 정리");
13         System.out.println("실행 종료");
14     }
15 }
```

# 학습 정리 1

---

- ❖ 프로세스 : 애플리케이션이 실행하면 운영체제로부터 실행에 필요한 메모리 할당 받아 실행됨
- ❖ 멀티 스레드 : 하나의 프로세스 내에 동시 실행하는 스레드가 두 개 이상인 경우
- ❖ 메인 스레드
  - 자바의 모든 어플리케이션은 main 스레드가 main() 메소드 실행하면서 시작
  - main() 메소드의 첫 코드부터 아래로 순차 실행
  - main() 메소드의 마지막 코드 실행하거나 return을 만나면 실행 종료
- ❖ 작업 스레드
  - 메인 작업 이외에 병렬 작업의 수만큼 생성하는 스레드
  - 객체로 생성되기 때문에 클래스 필요
  - Runnable 인터페이스를 구현한 스레드를 직접 객체화해서 생성할 수도 있고,
  - Thread 클래스를 상속해서 하위 클래스 만들어 생성할 수도 있음

# 학습 정리 2

---

## ❖ 동기화 메소드

- 멀티 스레드 프로그램에서 단 하나의 스레드만 실행할 수 있는 코드 영역을 임계영역이라 함
- 임계영역을 지정하기 위해 synchronized가 설정된 동기화 블록 또는 메소드가 제공됨
- 임계영역의 블록 또는 메소드를 실행하면, 즉시 잠금이 걸려 다른 스레드가 임계영역을 실행하지 못함

## ❖ 스레드 상태

- 스레드는 다양한 상태를 가지게 되며, 이는 자동으로 혹은 코드에 의해 변경될 수 있음

## ❖ 일시 정지

- 실행 중인 스레드를 일정 시간 멈추게 하는 경우 Thread 클래스의 정적 메소드인 sleep() 사용
- Thread.sleep() 메소드를 호출한 스레드는 주어진 시간 동안 일시정지 상태가 되고 다시 실행 대기 상태로 돌아감

## ❖ 안전한 종료

- 스레드를 안전하게 종료하기 위해 stop 플래그나 interrupt() 메소드를 이용할 수 있음

# 학습 정리 3

---

## ❖ 스레드에 대한 설명 중 틀린 것은 무엇입니까?

- 자바 애플리케이션은 메인 스레드가 `main()` 메소드를 실행한다.
- 작업 스레드 클래스는 `Thread` 클래스를 상속해서 만들 수 있다.
- `Runnable` 객체는 스레드가 실행해야 할 코드를 가지고 있는 객체라 볼 수 있다.
- 스레드 실행을 시작하려면 `run()` 메소드를 호출해야 한다.

## ❖ 동기화 메소드에 대한 설명 중 틀린 것은 무엇입니까?

- 동기화 메소드는 싱글 스레드 환경에서는 필요하지 않다.
- 스레드가 동기화 메소드를 실행할 때 다른 스레드는 일반 메소드를 호출할 수 없다.
- 스레드가 동기화 메소드를 실행할 때 다른 스레드는 다른 동기화 메소드를 호출할 수 없다.
- 동기화 메소드 선언부에는 `synchronized` 키워드가 필요하다.

# 적용 확인 학습 & 응용 프로그래밍

---

- ❖ 다음 파일에 있는 문제들의 해답을 스스로 작성 해 보신 후 개념 & 적용 확인 학습 영상을 학습 하시기 바랍니다.
  - java\_11장\_스레드\_ex.pdf
- ❖ 퀴즈와 과제가 출제되었습니다.
  - 영상 수업을 학습하신 후 과제와 퀴즈를 수행 하시기 바랍니다.

# Q & A

- ❖ “스레드”에 대한 학습이 모두 끝났습니다.
- ❖ 모든 내용을 이해 하셨나요?
- ❖ 아직 이해가 안되는 내용이 있다면 다시 한번 복습하시기 바랍니다.
- ❖ 질문은 한림 SmartLEAD 쪽지 또는 e-mail 또는 전화상담을 이용하시기 바랍니다.



- ❖ 퀴즈와 과제가 출제되었습니다. 마감시간에 늦지 않도록 주의해 주세요.
- ❖ 다음 시간에는 “제네릭과 컬렉션”을 공부하도록 하겠습니다.
- ❖ 수고하셨습니다.^^