12장 멀티 스레드

12.1 멀티 스레드 개념

12.1.1 프로세스와 스레드

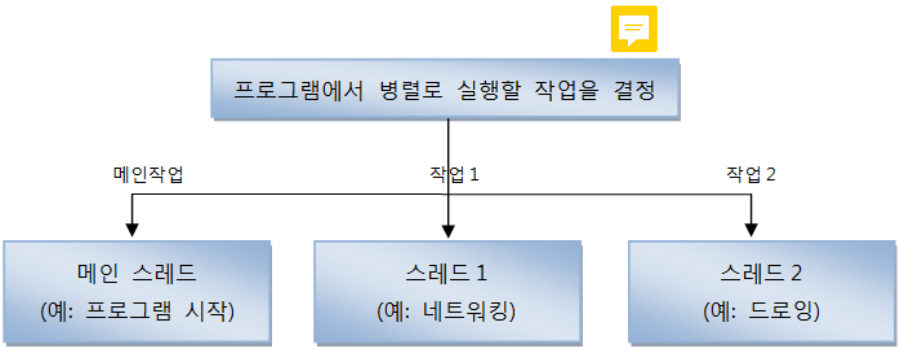
- 프로세스(process)
  - 실행 중인 하나의 프로그램
  - 하나의 프로그램이 다중 프로세스 만들기도 하는데, 예를 들어 Chrome 브라우저를 두 개 실행했다면 두 개의 Chrome 프로세스가 생성된 것이다.
- 멀티 태스킹(multi tasking)
  - 두 가지 이상의 작업을 동시에 처리하는 것
- 멀티 프로세스
  - 독립적으로 프로그램들을 실행하고 여러 가지 작업 처리
- 멀티 스레드
  - 한 개의 프로그램을 실행하고 내부적으로 여러 가지 작업 처리

12.1.2 메인 스레드

- 모든 자바 프로그램은 메인 스레드가 main() 메소드 실행하며 시작
- main() 메소드의 첫 코드부터 아래로 순차적으로 실행
- 실행 종료 조건
  - 마지막 코드 실행
  - return 문을 만나면
- main 스레드는 작업 스레드들을 만들어 병렬로 코드들 실행
  - 멀티 스레드 생성해 멀티 태스킹 수행
- 프로세스의 종료
  - 싱글 스레드: 메인 스레드가 종료하면 프로세스도 종료
  - 멀티 스레드: 실행 중인 스레드가 하나라도 있다면, 프로세스 미종료

12.2 작업 스레드 생성과 실행

- 멀티 스레드로 실행하는 애플리케이션을 개발하려면 먼저 몇 개의 작업을 병렬로 실행할지 결정하고 각 작업별로 스레드를 생성해야 한다.



■ 작업 스레드는 객체로 생성되어야 한다. -> java.lang.Thread 클래스를 직접 객체화해서 생성하거나, Thread를 상속해서 하위 클래스를 만들어 생성할 수 도 있다.

12.2.1 Thread 클래스로부터 직접 생성

■ Runnable을 매개값으로 갖는 생성자를 호출해야 한다.

```
//기본형
Thread thread = new Thread(Runnable target);

// 방법1: Runnable 구현 객체
class Task implements Runnable {
   public void run() {
      스레드가 실행할 코드;
   }
}
Runnable task = new Task();  //Task task = new Task();
Thread thread = new Thread(task);

// 방법2: 익명 구현 객체
Thread thread = new Thread( new Runnable() {
   public void run() {
      스레드가 실행할 코드;
   }
} );

// 방법3: 람다식
Thread thread = new Thread( () -> {
   스레드가 실행할 코드;
} );

// 실행
thread.start();
```

[ThreadLife.java] Runnable 구현 객체

```
01   package sec02.exam01_createthread;
02
03   public class ThreadLife implements Runnable {
04
05            @Override
06            public void run() {
07                     for (int i = 1; i < 21; i++) {
08                              // thread의 이름과 정수 출력
09                              System.out.println(Thread.currentThread().getName() + " number = " + i);
10                     }
11            }
```

```
12
13          public static void main(String[] args) {
14                  ThreadLife tl = new ThreadLife();
15
16                  // 첫 번째 Thread 생성
17                  Thread first = new Thread(tl, "first1");
18                  // 두 번째 Thread 생성
19                  Thread second = new Thread(tl, "second1");
20                  // 세 번째 Thread 생성
21                  Thread third = new Thread(tl, "third1");
22
23                  second.start();
24                  first.start();
25                  third.start();
26          }
27
28  }
29
```

12.2.2 Thread 하위 클래스로부터 생성

■ Thread 클래스 상속 후 run 메소드 재정의해서 스레드가 실행할 코드 작성한다.

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

// 방법1: WorkerThread 자식 객체 이용
Thread thread = new WorkerThread();

// 방법2: 익명 자식 객체 이용
Thread thread = new Thread() {
   @Override
   public void run() {
      //스레드가 실행할 코드
   }
};

thread.start();
```

[ThreadEnd.java] extend Thread

```
01  package sec02.exam01_createthread;
02
03  public class ThreadEnd extends Thread {
04
05          @Override
06          public void run() {
07                  // thread가 시작되면 실행되는 문장
08                  for (int i = 1; i <= 20; i++) {
09                          System.out.println("run number = " + i);
10                  }
11          }
12
13          public static void main(String[] args) {
14                  ThreadEnd te = new ThreadEnd();
15                  // thread를 실행시킴
```

```java
                te.start();

                // main()내에서 화면에 101부터 120까지 출력
                for (int i = 101; i <= 120; i++) {
                        System.out.println("-------> main number = " + i);
                }
        }

}
```

[과제] 쓰레드 사용하기

■ 쓰레드(Thread) 두 개를 만들어 실행시켜 보자. (힌트: 인터페이스 Runnable를 구현하거나 Thread 클래스를 상속한다.)

[MyRun.java] 인터페이스 Runnable를 구현

```java
package verify.exam00;

...
```

[MyThread.java] Thread 클래스를 상속

```java
package verify.exam00;

...
```

[MyRunMain.java]

```java
package verify.exam00;

public class MyRunMain {
        public static void main(String[] args) {
                MyRun mr1 = new MyRun();
                Thread t1 = new Thread(mr1);
                Thread t2 = new MyThread();
                t1.start();
                t2.start();
                for (int i = 0; i < 500; i++) {
                        System.out.print("M");
                }
        }
}
```

[실행 결과]

```
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTSSMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
SSSSSSSSSSSSSSSSSSSSSSSSSSSSTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTMMMMMMMMMMMMMMMMMMMMMM
```

```
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
```

12.2.3 스레드의 이름

- 메인 스레드 이름: main
- 작업 스레드 이름(자동 설정): thread.getName();
- 작업 스레드 이름 변경: thread.setName("스레드 이름");
- 코드 실행하는 현재 스레드 객체의 참조 얻기: Thread thread = Thread.currentThread();

[ThreadNameExample.java] MainThread 이름 출력과 UserThread 생성 및 시작

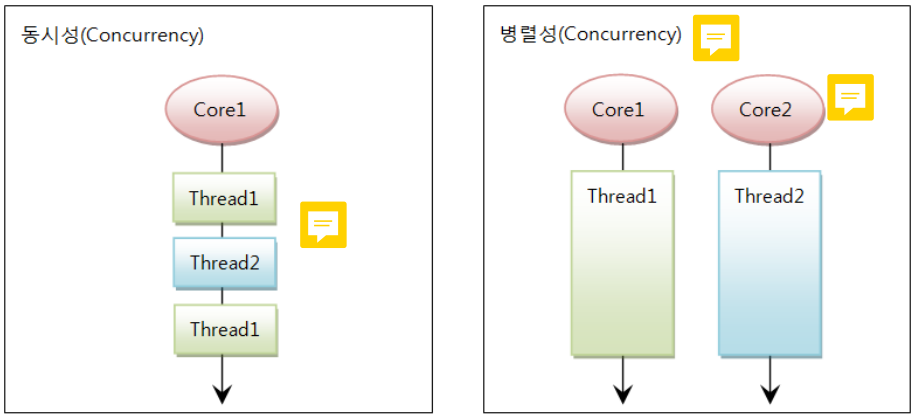
```java
package sec02.exam02_threadname;

public class ThreadNameExample {
		public static void main(String[] args) {
				Thread mainThread = Thread.currentThread();
				System.out.println("프로그램 시작 스레드 이름: " + mainThread.getName());

				ThreadA threadA = new ThreadA();
				System.out.println("작업 스레드 이름: " + threadA.getName());
				threadA.start();

				ThreadB threadB = new ThreadB();
				System.out.println("작업 스레드 이름: " + threadB.getName());
				threadB.start();
		}
}
```

[ThreadA.java] ThreadA 클래스

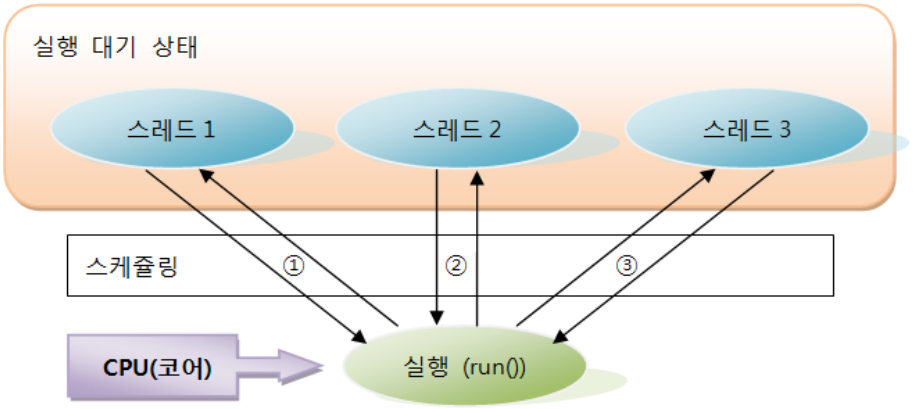
```java
package sec02.exam02_threadname;

public class ThreadA extends Thread {
		public ThreadA() {
				setName("ThreadA");
		}

		public void run() {
				for(int i=0; i<2; i++) {
						System.out.println(getName() + "가 출력한 내용");
				}
		}
}
```

12.3 스레드 우선순위

- 동시성: 멀티 작업 위해 하나의 코어에서 멀티 스레드가 번갈아 가며 실행하는 성질
- 병렬성: 멀티 작업을 위해 멀티 코어에서 개별 스레드를 동시에 실행하는 성질



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



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

```
// 기본형
thread.setPriority(우선순위);

thread.setPriority(Thread.MAX_PRIORITY);
thread.setPriority(Thread.NORM_PRIORITY);
thread.setPriority(Thread.MIN_PRIORITY);
```

[PriorityExample.java] 우선순위를 설정해서 스레드 실행

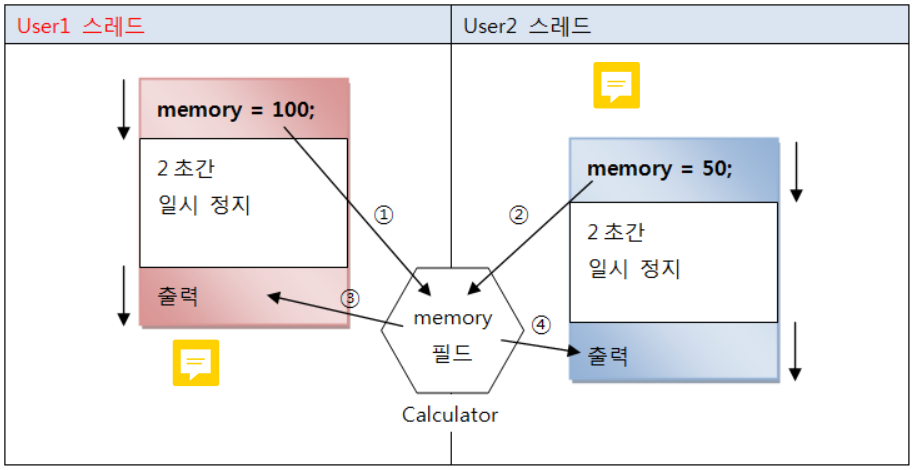
```
01  package sec03.exam01_priority;
02
03  public class PriorityExample {
04          public static void main(String[] args) {
05                  for(int i=1; i<=10; i++) {
06                          Thread thread = new CalcThread("thread" + i);
07                          if(i != 10) {
08                                  thread.setPriority(Thread.MIN_PRIORITY);
```

```
09                                  } else {
10                                          thread.setPriority(Thread.MAX_PRIORITY);
11                                  }
12                                  thread.start();
13                          }
14                  }
15          }
```

12.4 동기화 메소드와 동기화 블록

12.4.1 공유 객체를 사용할 때의 주의할 점

■ 멀티 스레드가 하나의 객체를 공유해서 생기는 오류



[실습] 쓰레드와 자원 공유 - 멤버 필드

■ 세 개의 쓰레드가 멤버 필드 하나를 공유하게 만들어 보자. (힌트: 같은 자원(같은 객체의 멤버 필드)을 여러 개의 쓰레드가 공유할 수 있다.)

[MemberPrint.java]

```
01  package verify.exam00;
02
03  public class MemberPrint implements Runnable {
04          private int i = 0; //
05
06          public void run() {
07                  show();
08          }
09
10          public void show() {
11                  for (; i < 500; i++) {
12                          if (((Thread.currentThread()).getName()).equals("a")) {
13                                  System.out.print("a");
14                          } else if (((Thread.currentThread()).getName()).equals("b")) {
15                                  System.out.print("b");
16                          } else if (((Thread.currentThread()).getName()).equals("c")) {
17                                  System.out.print("c");
18                          }
19                  }
20          }
```

```
}
```

[MemberPrintMain.java]

```
package verify.exam00;

public class MemberPrintMain {
        public static void main(String[] args) {
                MemberPrint mp = new MemberPrint();
                Thread t1 = new Thread(mp, "a");
                Thread t2 = new Thread(mp, "b");
                Thread t3 = new Thread(mp, "c");
                t1.start();
                t2.start();
                t3.start();
        }
}
```

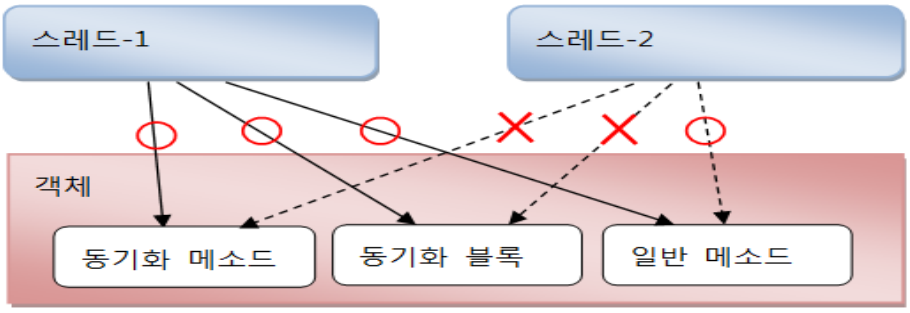
[실행 결과]

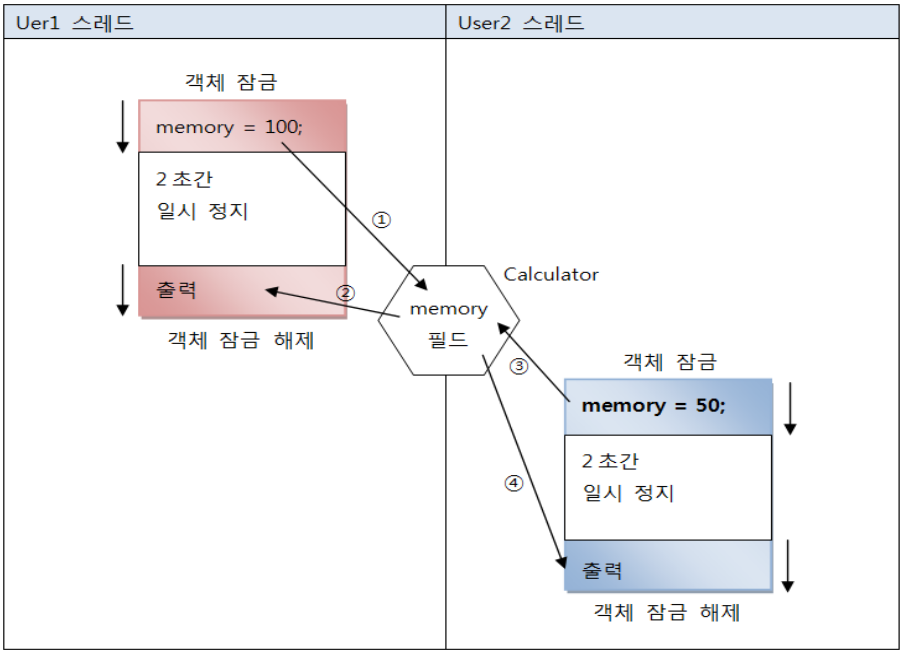
```
bbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaacb
```

12.4.2 동기화 메소드 및 동기화 블록

- 동기화 메소드 및 동기화 블록 synchronized
  - 임계 영역(critical section): 단 하나의 스레드만 실행할 수 있는 코드 영역
  - 다른 스레드는 메소드나 블록이 실행이 끝날 때까지 대기해야 한다.

```
//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
   ...
}
```





[Family.java]

```
01  package sec04.exam02_synchronized;
02
03  // Thread 클래스를 상속받아 Thread를 정의함.
04
05  public class Family extends Thread {
06            Toilet toilet;
07            String who;
08            boolean key; // 초기값: false
09
10            // 생성자
11            public Family(String name, Toilet t) {
12                      who = name;
13                      toilet = t;
14            }
15
16            public void run() {
17                      toilet.openDoor(who, key);
18            }
19  }
```

[Toilet.java]

```
01  package sec04.exam02_synchronized;
02
```

```
03  // key를 이용해서 Thread간에 충돌이 일어나지 않도록 실행순서를 맞추는 것을
04  // Thread의 동기화(Synchronization)라고 한다.
05
06  public class Toilet { // 화장실을 사용하는 과정을 보여주는 클래스
07
08            // 메소드의 동기화 방법
09            // synchronized로 선언된 openDoor() 메소드는 한번 실행이 끝나야 다음 실행이 가능함.
10            // 다른 Thread들은 한개의 Thread가 이 메소드의 실행을 끝낼때 까지 대기함.
11            public synchronized void openDoor(String name, boolean b) {
12                      // public void openDoor( String name, boolean b ) {
13                      if (b == false) {
14                                System.out.println(name);
15                                usingTime();
16                                System.out.println("아~~~~! 시원해");
17                      } else {
18                                System.out.println("사용중");
19                      }
20            }// openDoor() end
21
22            public void usingTime() { // 화장실 사용하는 시간
23                      for (int i = 0; i < 100000000; i++) {
24                                if (i == 10000) {
25                                          System.out.println("끄으응");
26                                }
27                      }
28            }// usingTime() end
29
30  }
```

[ManageToilet.java]

```
01  package sec04.exam02_synchronized;
02
03  // 5개의 Thread를 만들어 실행 시키는 클래스
04
05  public class ManageToilet {
06
07      public static void main( String[] args ) {
08                      Toilet t = new Toilet();
09
10                      // thread 생성
11                      Family father = new Family("아버지", t );
12                      Family mother = new Family("어머니", t );
13                      Family sister = new Family("누나", t );
14                      Family brother = new Family("형", t );
15                      Family me = new Family("나", t );
16
17  /*** 우선 순위 적용안됨
18                      father.setPriority(10);
19                      mother.setPriority(7);
20                      sister.setPriority(5);
21                      brother.setPriority(3);
22                      me.setPriority(1);
23  */
24
25                      // 각 Thread는 Runnable 상태에 들어감
26                      father.start();
27                      mother.start();
28                      sister.start();
29                      brother.start();
30                      me.start();
31      }
32  }
```

[실습] 쓰레드와 자원 공유 - 동기화

■ 여러 개의 쓰레드에서 같은 자원을 동시에 사용할 수 없게 만들어 보자. (힌트: synchronized 키워드를 이용한다.)

[MemberLockPrint.java]

```
package verify.exam00;

public class MemberLockPrint implements Runnable {
	private int i = 0; //

	public void run() {
		show();
	}

	public synchronized void show() {
		for (; i < 500; i++) {
			if (((Thread.currentThread()).getName()).equals("a")) {
				System.out.print("a");
			} else if (((Thread.currentThread()).getName()).equals("b")) {
				System.out.print("b");
			} else if (((Thread.currentThread()).getName()).equals("c")) {
				System.out.print("c");
			}
		}
	}
}
```

[MemberLockPrintMain.java]

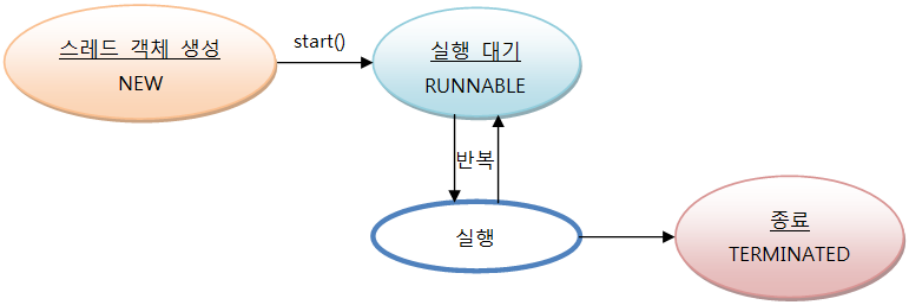
```
package verify.exam00;

public class MemberLockPrintMain {
	public static void main(String[] args) {
		MemberLockPrint mp = new MemberLockPrint();
		Thread t1 = new Thread(mp, "a");
		Thread t2 = new Thread(mp, "b");
		Thread t3 = new Thread(mp, "c");
		t1.start();
		t2.start();
		t3.start();
	}
}
```

[실행 결과]

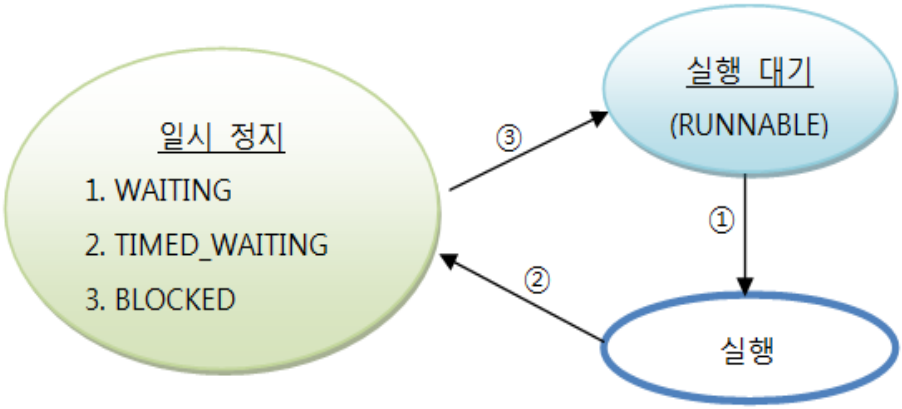
```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

12.5 스레드 상태

■ 스레드의 일반적인 상태



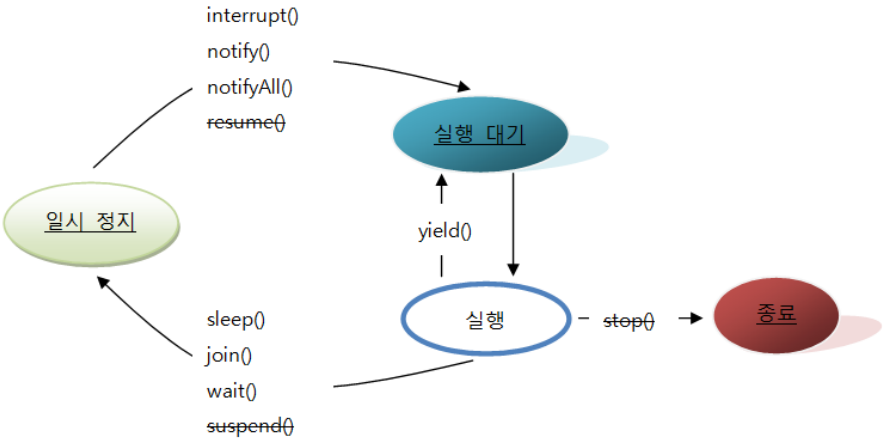
■ 스레드에 일시 정지 상태 도입한 경우



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 때가지 기다리는 상태, 다른 스레드가 동기화 메소드를 호출한 경우
	WAITING	다른 스레드가 통지할 때까지 기다리는 상태, wait() 메소드를 호출한 경우
	TIMED_WAITING	주어진 시간 동안 기다리는 상태, sleep() 메소드를 호출한 경우
종료	TERMINATED	실행을 마친 상태

12.6 스레드 상태 제어

■ 실행 중인 스레드의 상태를 변경하는 것
■ 상태 변화를 가져오는 메소드의 종류



■ 위 그림에서 취소선을 가진 메소드는 스레드의 안전성을 헤친다고 하여 더 이상 사용하지 않도록 권장된 Deprecated 메소드들이다.

12.6.1 주어진 시간동안 일시 정지(sleep())

■ 얼마 동안 일시 정지 상태로 있을 것인지 밀리 세컨드(1/1000) 단위로 지정
■ 일시 정지 상태에서 interrupt() 메소드 호출 -> InterruptedException 발생

```
try {
   Thread.sleep(1000);
} catch(InterruptedException e) {
   // interrup() 메소드가 호출되면 실행
}
```

[실습] 쓰레드와 sleep 메서드

■ 50밀리세컨드마다 쓰레드 이름을 출력하게 만들어 보자. (힌트: Thread.sleep 메서드를 사용하고 예외처리를 해준다.)

[SleepThread.java]

```
01   package verify.exam00;
02
03   public class SleepThread extends Thread {
04            public SleepThread(String name) {
05                     setName(name);
06            }
07
08            public void run() {
09                     show();
10            }
11
12            public void show() {
13                     for (int i = 0; i < 50; i++) {
14                              print();
15                              try {
16                                       Thread.sleep(50);// 50/1000 초
17                              } catch (InterruptedException ite) {
18                              }
```

```
                }
        }

        public void print() {
                System.out.print(getName());// Thread에서
        }
}
```

[SleepThreadMain.java]

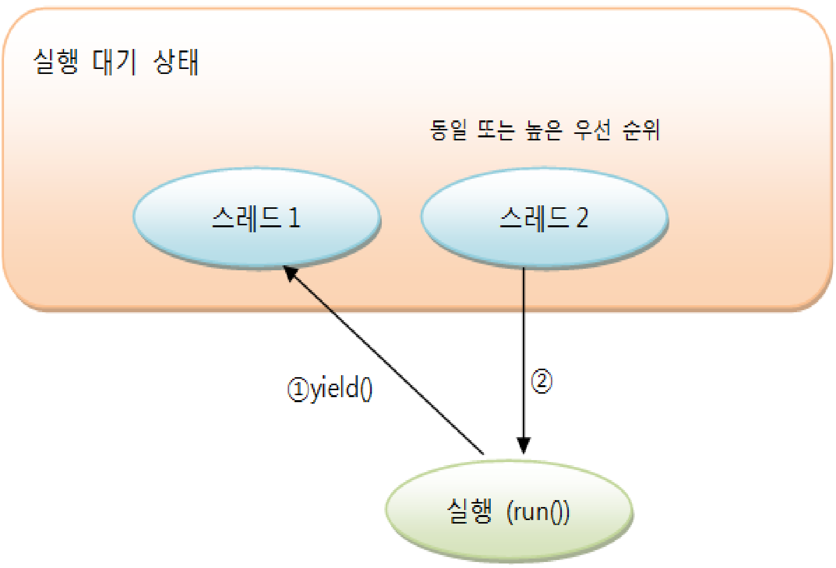
```
package verify.exam00;

public class SleepThreadMain {
        public static void main(String[] args) {
                SleepThread t1 = new SleepThread("a");
                SleepThread t2 = new SleepThread("b");
                SleepThread t3 = new SleepThread("c");

                t2.setPriority(7);// 1~10 클수록 우서순의
                t1.start();// t2가 t1보다 우선이지만
                try {
                        t1.join();// t1을 끝낸후 t2, t3를 실행한다.
                } catch (InterruptedException ite) {
                }
                t2.start();
                t3.start();
        }
}
```

[실행 결과]

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabcbcbccbcbbccbcbbccbcbbccbcbbccbbcbccbbcbcbccbbcbccbcbbcc
bbcbccbbcbcbccbbccbcbbcbcbccbcbcbbcbcbc
```

12.6.2 다른 스레드에게 실행 양보(yield())

■ 실행 중에 우선순위가 동일한 다른 스레드에게 실행을 양보하고 실행 대기 상태가 된다.

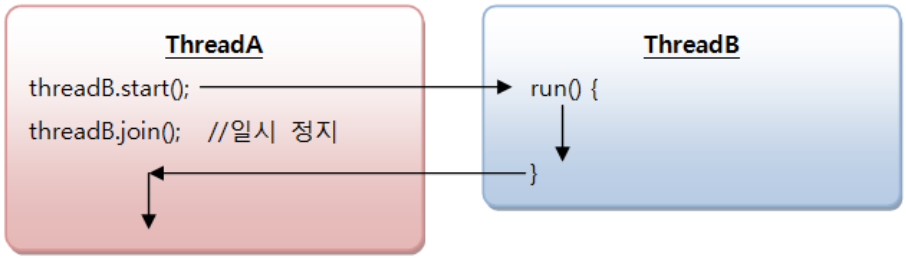


[ThreadA.java] 스레드 실행 양보 예제

```
01  package sec06.exam02_yield;
02
03  public class ThreadA extends Thread {
04          public boolean stop = false;
05          public boolean work = true;
06
07          public void run() {
08                  while (!stop) {
09                          if (work) {
10                                  System.out.println("ThreadA 작업 내용");
11                          } else {
12                                  Thread.yield();
13                          }
14                  }
15                  System.out.println("ThreadA 종료");
16          }
17  }
```

12.6.3 다른 스레드의 종료를 기다림(join())

■ 계산 작업을 하는 스레드가 모든 계산 작업 마쳤을 때, 결과값을 받아 이용하는 경우 주로 사용

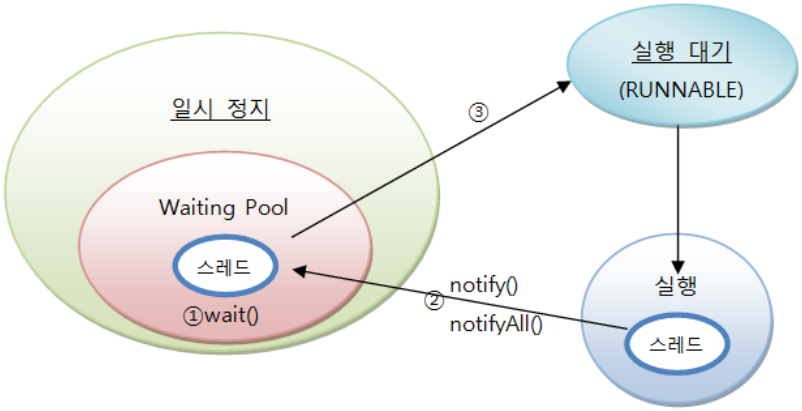


[JoinExample.java] 다른 스레드가 종료될 때까지 일시 정지 상태 유지

```
01  package sec06.exam03_join;
02
03  public class JoinExample {
04          public static void main(String[] args) {
05                  SumThread sumThread = new SumThread();
06                  sumThread.start();
07                  try {
08                          sumThread.join();  //sumThread가 종료될 때까지 메인 스레드를 일지 정지
09                  } catch (InterruptedException e) {
10                  }
11                  System.out.println("1~100 합: " + sumThread.getSum());
12          }
13  }
```

12.6.4 스레드 간 협업(wait(), notify(), notifyAll())

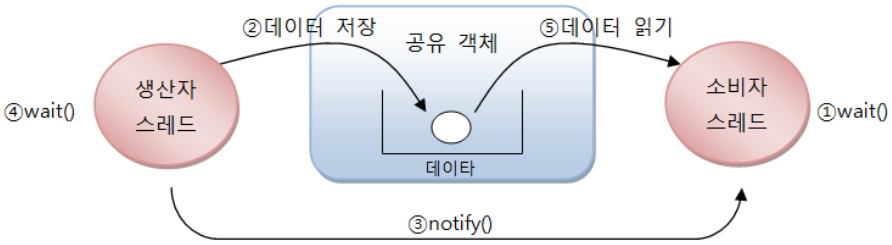
■ 동기화 메소드 또는 블록에서만 호출 가능한 Object의 메소드



[WorkObject.java] 두 스레드의 작업 내용을 동기화 메소드로 작성한 공유 객체

```
01  package sec06.exam04_wait_notify;
02
03  public class WorkObject {
04          public synchronized void methodA() {
05                  System.out.println("ThreadA의 methodA() 작업 실행");
06                  notify(); /* 일시 정지 상태에 있는 ThreadB를 실행 대기 상태로 만듬 */
07                  try {
08                          wait(); /* ThreadA를 일시 정지 상태로 만듬 */
09                  } catch (InterruptedException e) {
10                  }
11          }
12
13          public synchronized void methodB() {
14                  System.out.println("ThreadB의 methodB() 작업 실행");
15                  notify(); /* 일시 정지 상태에 있는 ThreadA를 실행 대기 상태로 만듬 */
16                  try {
17                          wait(); /* ThreadB를 일시 정지 상태로 만듬 */
18                  } catch (InterruptedException e) {
19                  }
20          }
21  }
```

■ 두 개의 스레드가 교대로 번갈아 가며 실행해야 할 경우 주로 사용



[DataBox.java] 두 스레드의 작업 내용을 동기화 메소드로 작성한 공유 객체

```
01  package sec06.exam05_wait_notify;
02
03  public class DataBox {
04          private String data;
05
06          public synchronized String getData() {
07                  if (this.data == null) { // data 필드가 null이면 소비자 스레드를 일시 정지 상태로
08  만듬
```

```
09                              try {
10                                      wait();
11                              } catch (InterruptedException e) {
12                              }
13                      }
14                      String returnValue = data;
15                      System.out.println("ConsummerThread가 읽은 데이터: " + returnValue);
16                      data = null; // data 필드를 null로 만들고 생상자 스레드를 실행 대기 상태로 만듬
17                      notify();
18                      return returnValue;
19              }
20
21              public synchronized void setData(String data) {
22                      if (this.data != null) { // data 필드가 null이 아니면 생산자 스레드를 일시 정지
23      상태로 만듬
24                              try {
25                                      wait();
26                              } catch (InterruptedException e) {
27                              }
28                      }
29                      this.data = data; // data 필드에 값을 저장하고 소비자 스레드를 실행 대기 상태로 만듬
30                      System.out.println("ProducerThread가 생성한 데이터: " + data);
31                      notify();
32              }
33      }
34
```

[실습] 쓰레드와 wait, notifyAll 메서도 이해하기

■ 필요한 자원이 준비될 때까지 쓰레드 작업을 지연시켜 보자. (힌트: 동기화된 메서드에 대해 wait와 notifyAll 메서드를 사용한다.)

[CakePlate.java]

```
01      package verify.exam00;
02
03      public class CakePlate {
04              private int breadCount = 0;
05
06              public CakePlate() {
07              }
08
09              public synchronized void makeBread() {
10                      if (breadCount >= 10) {
11                              try {
12                                      System.out.println("빵이 남는다.");
13                                      wait();
14                              } catch (InterruptedException ire) {
15                              }
16                      }
17                      breadCount++;// 빵이 10개가 안되면 더 만들자.
18                      System.out.println("빵을 1개 더 만듬  총 : " + breadCount + "개");
19                      this.notifyAll();
20              }
21
22              public synchronized void eatBread() {
23                      if (breadCount < 1) {
24                              try {
25                                      System.out.println("빵이 모자라 기다림");
26                                      wait();
27                              } catch (InterruptedException ire) {
28                              }
```

```
29                      }
30                      breadCount--;// 빵이 있으니 먹자.
31                      System.out.println("빵을 1개 먹음  총 : " + breadCount + "개");
32                      this.notifyAll();
33              }
34      }
```

[CakeMaker.java]

```
01      package verify.exam00;
02
03      public class CakeMaker extends Thread {
04              private CakePlate cake;
05
06              public CakeMaker(CakePlate cake) {
07                      setCakePlate(cake);
08              }
09
10              public void setCakePlate(CakePlate cake) {
11                      this.cake = cake;
12              }
13
14              public CakePlate getCakePlate() {
15                      return cake;
16              }
17
18              public void run() {
19                      for (int i = 0; i < 30; i++) {
20                              cake.makeBread();
21                      }
22              }
23      }
```

[CakeEater.java]

```
01      package verify.exam00;
02
03      public class CakeEater extends Thread {
04              private CakePlate cake;
05
06              public CakeEater(CakePlate cake) {
07                      setCakePlate(cake);
08              }
09
10              public void setCakePlate(CakePlate cake) {
11                      this.cake = cake;
12              }
13
14              public CakePlate getCakePlate() {
15                      return cake;
16              }
17
18              public void run() {
19                      for (int i = 0; i < 30; i++) {
20                              cake.eatBread();
21                      }
22              }
23      }
```

[CakeEatings.java]

```
01  package verify.exam00;
02
03  public class CakeEatings {
04
05          public static void main(String[] args) {
06                  CakePlate cake = new CakePlate();// Cake 접시 준비
07                  CakeEater eater = new CakeEater(cake);// cake 접시 공유
08                  CakeMaker baker = new CakeMaker(cake);// cake 접시 공유
09
10                  // baker.setPriority(6);//우선순위--먼저 채워 넣고 시작하자.
11                  baker.start();// 먼저 채워 넣고 시작하자.
12                  eater.start();
13          }
14  }
```

[실행 결과]

```
빵을 1개 더 만듬  총 : 1개
빵을 1개 더 만듬  총 : 2개
빵을 1개 더 만듬  총 : 3개
...(생략)
빵을 1개 먹음  총 : 3개
빵을 1개 먹음  총 : 2개
빵을 1개 먹음  총 : 1개
빵을 1개 먹음  총 : 0개
```

12.6.5 스레드의 안전한 종료(stop 플래그, interrupt())

(1) stop 플래그를 이용하는 방법

- 경우에 따라 실행 중인 스레드 즉시 종료해야 할 필요 있을 때 사용
- stop() 메소드 사용시
  - 스레드 즉시 종료 되는 편리함
  - Deprecated : 사용 중이던 자원들이 불안전한 상태로 남겨짐
- 안전한 종료 위해 stop 플래그 이용하는 방법
  - stop 플래그로 메소드의 정상 종료 유도

[PrintThread1.java] 무한 반복해서 출력하는 스레드

```
01  package sec06.exam06_stop;
02
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  }
```

(2) interrupt() 메소드를 이용하는 방법

- 스레드가 일시 정지 상태일 경우, InterruptedException 발생 시킨다.
- 실행대기 또는 실행상태에서는 InterruptedException 발생하지 않는다.
- 일시 정지 상태로 만들지 않고 while문 빠져 나오는 방법으로도 쓰인다.
- interrupt() 메소드가 호출되었다면 스레드의 interrupted()와 isInterrupted() 메소드는 true를 리턴한다.

[InterruptExample.java] 1초 후 출력 스레드를 중지시킴

```
01  package sec06.exam06_stop;
02
03  public class InterruptExample {
04          public static void main(String[] args) {
05                  Thread thread = new PrintThread2();
06                  thread.start();
07
08                  try {
09                          Thread.sleep(1000);
10                  } catch (InterruptedException e) {
11                  }
12
13                  thread.interrupt();
14          }
15  }
```

[PrintThread2.java] 무한 반복해서 출력하는 스레드

```
01  package sec06.exam06_stop;
02
03  public class PrintThread2 extends Thread {
04          public void run() {
05                  // how1
06                  /*
07                   * try { while(true) { System.out.println("실행 중"); Thread.sleep(1); } }
08                   * catch(InterruptedException e) { }
09                   */
10
11                  // how2
12                  while (true) {
13                          System.out.println("실행 중");
14                          if (Thread.interrupted()) {
15                                  break;
16                          }
17                  }
18
19                  System.out.println("자원 정리");
20                  System.out.println("실행 종료");
21          }
22  }
```

12.7 데몬 스레드

- 주 스레드의 작업 돕는 보조적인 역할 수행하는 스레드
- 주 스레드가 종료되면 데몬 스레드는 강제적으로 자동 종료

- 워드프로세서의 자동저장, 미디어플레이어의 동영상 및 음악 재생

■ 스레드를 데몬 스레드로 만들기
- 주 스레드가 데몬이 될 스레드의 setDaemon(true) 호출
- 반드시 start() 메소드 호출 전에 setDaemon(true) 호출 -> 그렇지 않으면 IllegalThreadStateException이 발생

■ 현재 실행중인 스레드가 데몬 스레드인지 구별법
- isDaemon() 메소드의 리턴값 조사 -> true면 데몬 스레드

[DaemonExample.java] 메인 스레드가 실행하는 코드

```
01  package sec07.exam01_daemon;
02
03  public class DaemonExample {
04          public static void main(String[] args) {
05                  AutoSaveThread autoSaveThread = new AutoSaveThread();
06                  autoSaveThread.setDaemon(true);  //AutoSaveThread를 데몬 스레드로 만듬
07                  autoSaveThread.start();
08
09                  try {
10                          Thread.sleep(3000);
11                  } catch (InterruptedException e) {
12                  }
13
14                  System.out.println("메인 스레드 종료");
15          }
16  }
```

12.8 스레드 그룹

■ 관련된 스레드 묶어 관리 목적으로 이용
■ 스레드 그룹은 계층적으로 하위 스레드 그룹 가질 수 있음
■ 자동 생성되는 스레드 그룹
- system 그룹: JVM 운영에 필요한 스레드들 포함
- system/main 그룹: 메인 스레드 포함

■ 스레드는 반드시 하나의 스레드 그룹에 포함
- 기본적으로 자신을 생성한 스레드와 같은 스레드 그룹
- 스레드 그룹에 포함시키지 않으면 기본적으로 system/main 그룹

12.8.1 스레드 그룹 이름 얻기

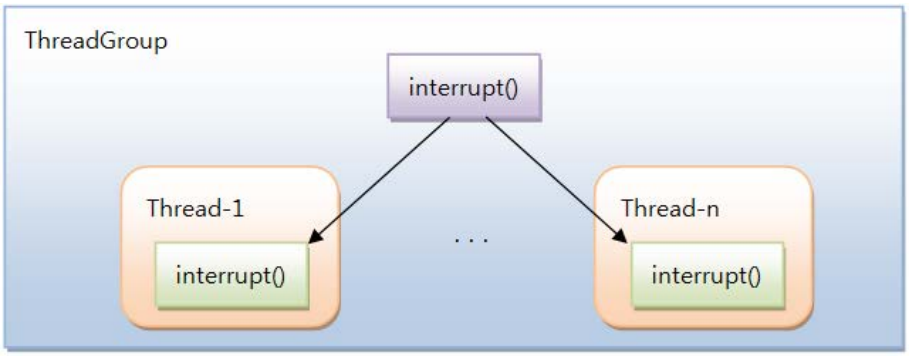
```
ThreadGroup group = Thread.currentThread().getThreadGroup();
String groupName = group.getName();

Thread t = new Thread(ThreadGroup group, Runnable target);
```

12.8.2 스레드 그룹 생성

```
ThreadGroup tg = new ThreadGroup(String name);
ThreadGroup tg = new ThreadGroup(ThreadGroup parent, String name);
```

12.8.3 스레드 그룹의 일괄 interrupt()

■ 스레드 그룹의 interrupt() 호출 시 소속된 모든 스레드의 interrupt() 호출

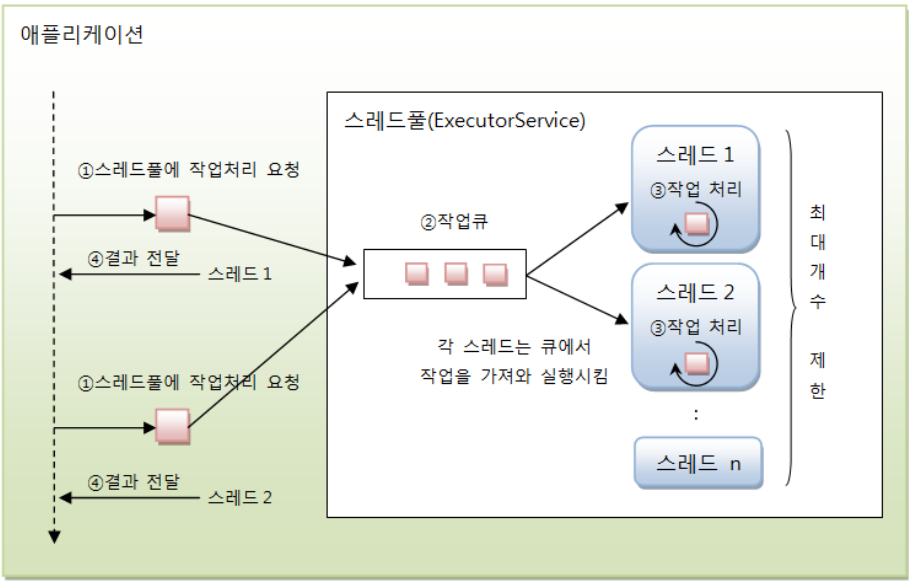


12.9 스레드풀

■ 스레드 폭증으로 일어나는 현상
- 병렬 작업 처리가 많아지면 스레드 개수 증가
- 스레드 생성과 스케줄링으로 인해 CPU가 바빠짐
- 메모리 사용량이 늘어남
- 애플리케이션의 성능 급격히 저하

■ 스레드 풀(Thread Pool)
- 작업 처리에 사용되는 스레드를 제한된 개수만큼 미리 생성
- 작업 큐(Queue)에 들어오는 작업들을 하나씩 스레드가 맡아 처리
- 작업 처리가 끝난 스레드는 작업 결과를 애플리케이션으로 전달
- 스레드는 다시 작업 큐에서 새로운 작업을 가져와 처리

- ExecutorService 인터페이스와 Executors 클래스
  - 스레드 풀 생성, 사용 - java.util.concurrent 패키지에서 제공
  - Executors의 정적 메소드 이용 - ExecutorService 구현 객체 생성
  - 스레드 풀 = ExecutorService 객체
  - ExecutorService가 동작하는 방식



12.9.1 스레드풀 생성 및 종료

(1) 스레드풀 생성

- 다음 두 가지 메소드 중 하나로 간편 생성

메소드명(매개변수)	초기 스레드수	코어 스레드수	최대 스레드수
newCachedThreadPool()	0	0	Integer.MAX_VALUE
newFixedThreadPool(int nThreads)	0	nThreads	nThreads

- newCachedThreadPool()
  - int 값이 가질 수 있는 최대값만큼 스레드가 추가되지만, 운영체제의 성능과 상황에 따라 달라진다.
  - 1개 이상의 스레드가 추가되었을 경우 60초 동안 추가된 스레드가 아무 작업을 하지 않으면 추가된 스레드를 종료하고 풀에서 제거한다.

- newFixedThreadPool(int nThreads)
  - 코어 스레드 개수와 최대 스레드 개수가 매개값으로 준 nThreads이다.
  - 스레드가 작업 처리하지 않고 놀고 있더라도 스레드 개수가 줄지 않는다.

- 위 두 가지 메소드도 내부적으로 ThreadPoolExecutor 객체를 생성해서 리턴한다.

```
// 1. Executors 클래스의 두 가지 메소드
ExecutorService executorService = Executors.newCachedThreadPool();
ExecutorService executorService = Executors.newFixedThreadPool(
   Runtime.getRuntime().availableProcessors()
);
```

```
// 2. 위 메소드를 사용하지 않고 ThreadPoolExecutor 객체를 생성
ExecutorService threadPool = new ThreadPoolExecutor(
   3,     //코어 스레드 개수
   100,   //최대 스레드 개수
   120L,  //놀고 있는 시간
   TimeUnit.SECONDS,  //놀고 있는 시간 단위
   new SynchronousQueue<Runnable>();  //작업 큐
);
```

(2) 스레드풀 종료

- 스레드 풀의 스레드는 기본적으로 데몬 스레드가 아님
  - main 스레드 종료되더라도 스레드 풀 스레드는 작업 처리 위해 계속 실행 -> 애플리케이션은 종료되지 않는다.
  - 애플리케이션을 종료하려면 스레드풀을 종료시켜 스레드들이 종료상태가 되도록 처리해 주어야 한다.
- 스레드 풀 종료 메소드

리턴타입	메소드명(매개변수)	설명
void	shutdown()	현재 처리 중인 작업뿐만 아니라 작업큐에 대기하고 있는 모든 작업을 처리한 뒤에 스레드풀을 종료시킨다.
List<Runnable>	shutdownNow()	현재 작업 처리 중인 스레드를 interrupt 해서 작업 중지를 시도하고 스레드풀을 종료시킨다. 리턴값은 작업큐에있는 미처리된 작업(Runnable)의 목록이다.
boolean	awaitTermination(long timeout, TimeUnit unit)	shutdown() 메소드 호출 이후, 모든 작업 처리를 timeout 시간 내에 완료하면 true 를 리턴하고, 완료하지 못하면 작업 처리 중인 스레드를 interrupt 하고 false 를 리턴한다.

```
executorService.shutdown();
executorService.shutdownNow();
```

12.9.2 작업 생성과 처리 요청

(1) 작업 생성

- 하나의 작업은 Runnable 또는 Callable 객체로 표현
- Runnable과 Callable의 차이점: 작업 처리 완료 후 리턴값이 있느냐 없느냐

Runnable 구현 클래스	Callable 구현 클래스

Runnable task = new Runnable() {
  @Override
  public void run() {
  //스레드가 처리할 작업 내용
  }
}
	Callable<T> task = new Callable<T> {
	  @Override
	  public T call() throws Exception {
	  //스레드가 처리할 작업 내용
	  return T;
	  }
	}

(2) 작업 처리 요청

- ExecutorService의 작업 큐에 Runnable 나 Callable 객체 넣음
- 작업 처리 요청 위해 ExecutorService는 두 가지 메소드 제공

리턴타입	메소드명(매개변수)	설명
void	execute(Runnable command)	- Runnable을 작업큐에 저장 - 작업 처리 결과를 받지 못함
Future<?> Future<V> Future<V>	submit(Runnable task) submit(Runnable task,V result) submit(Callable<V> task)	- Runnable 또는 Callable을 작업큐에 저장 - 리턴된 Future를 통해 작업 처리 결과를 얻을 수 있음

- 작업 처리 도중 예외 발생할 경우
  - execute(): 스레드 종료 후 해당 스레드 제거, 스레드 풀은 다른 작업 처리를 위해 새로운 스레드 생성
  - submit(): 스레드가 종료되지 않고 다음 작업 위해 재사용

[ExecuteVsSubmitExample.java] execute() 메소드로 작업 처리 요청한 경우

```
01  package sec09.exam01_execute_submit;
02
03  import java.util.concurrent.ExecutorService;
04  import java.util.concurrent.Executors;
05  import java.util.concurrent.ThreadPoolExecutor;
06
07  public class ExecuteVsSubmitExample {
08          public static void main(String[] args) throws Exception {
09                  ExecutorService executorService = Executors.newFixedThreadPool(2);
10
11                  for (int i = 0; i < 10; i++) {
12                          Runnable runnable = new Runnable() {
13                                  @Override
14                                  public void run() {
15                                          // 스레드 총 개수 및 작업 스레드 이름 출력
16                                          ThreadPoolExecutor threadPoolExecutor =
17  (ThreadPoolExecutor) executorService;
18                                          int poolSize = threadPoolExecutor.getPoolSize();
19                                          String threadName = Thread.currentThread().getName();
20                                          System.out.println("[총 스레드 개수: " + poolSize + "]
21  작업 스레드 이름: " + threadName);
22                                          // 예외 발생 시킴
23                                          int value = Integer.parseInt("삼");
24                                  }
25                          };
26
27                          executorService.execute(runnable);  //작업 처리 요청
28                          // executorService.submit(runnable);
29
30                          Thread.sleep(10);
31                  }
32
33                  executorService.shutdown();  //스레드풀 종료
34          }
35  }
36
```

12.9.3 블로킹 방식의 작업 완료 통보

- 작업이 완료될 때까지 기다렸다가 (지연 되었다가) 메소드 실행

리턴타입	메소드명(매개변수)	설명
Future<?>	submit(Runnable task)	- Runnable 또는 Callable 을 작업큐에 저장 - 리턴된 Future 를 통해 작업 처리 결과 얻음
Future<V>	submit(Runnable task, V result)	
Future<V>	submit(Callable<V> task)	

■ Future 객체

- 작업 결과가 아니라 작업이 완료될 때까지 기다렸다가 최종 결과를 얻는데 사용하는 지연 완료(pending completion) 객체이다.
- Future의 get() 메소드를 호출하면 스레드가 작업을 완료할 때까지 블로킹되었다가 작업을 완료하면 처리 결과를 리턴한다. 즉, 스레드가 작업을 완료하기 전까지는 get() 메소드가 블로킹되므로 다른 코드를 실행할 수 없다.

리턴타입	메소드명(매개변수)	설명
V	get()	작업이 완료될 때까지 블로킹되었다가 처리 결과 V 를 리턴
V	get(long timeout, TimeUnit unit)	timeout 시간동안 작업이 완료되면 결과 V 를 리턴하지만, 작업이 완료되지 않으면 TimeoutException 을 발생시킴

```
// 새로운 스레드를 생성해서 호출
new Thread(new Runnable() {
   @Override
   public void run() {
      try {
         future.get();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}

// 스레드풀의 스레드가 호출
executorService.submit(new Runnable() {
   @Override
   public void run() {
      try {
         future.get();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

- Future 객체에 속한 다른 메소드

리턴타입	메소드명(매개변수)	설명
boolean	cancel(boolean mayInterruptIfRunning)	작업 처리가 진행중일 경우 취소 시킴
boolean	isCancelled()	작업이 취소되었는지 여부
boolean	isDone()	작업 처리가 완료되었는지 여부

(1) 리턴값이 없는 작업 완료 통보

■ Runnable 객체로 생성해 처리

■ 결과값이 없음에도 Future 객체를 리턴하는데, 이것은 스레드가 작업 처리를 정상적으로 완료했는지, 아니면 작업 처리 도중에 예외가 발생했는지 확인하기 위해서이다.

```
Runnable task = new Runnable() {
   @Override
   public void run() {
      //스레드가 처리할 작업 내용
   }
};

Future future = executorService.submit(task);

try {
   future.get();
} catch (InterruptedException e) {
   //작업 처리 도중 스레드가 interrupt 될 경우 실행할 코드
} catch (ExecutionException e) {
   //작업 처리 도중 예외가 발생된 경우 실행할 코드
}
```

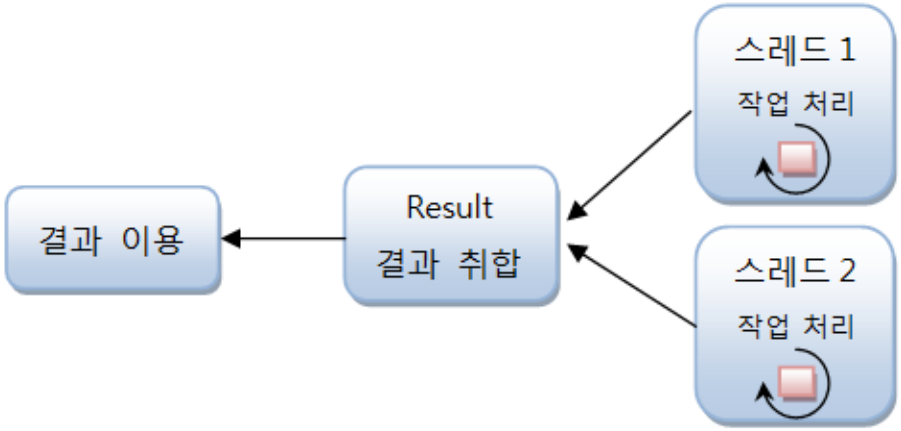
(2) 리턴값이 있는 작업 완료 통보

■ 작업 객체를 Callable 로 생성

```
Callable<T> task = new Callable<T>() {
   @Override
   public T call() throws Exception {
      //스레드가 처리할 작업 내용
      return T;
   }
}

Future<T> future = executorService.submit(task);

try {
   T result = future.get();
} catch (InterruptedException e) {
   //작업 처리 도중 스레드가 interrupt 될 경우 실행할 코드
} catch (ExecutionException e) {
   //작업 처리 도중 예외가 발생된 경우 실행할 코드
}
```

(3) 작업 처리 결과를 외부 객체에 저장

■ 보통은 두 개 이상의 스레드 작업을 취합할 목적으로 사용




```
Result result = ...;
Runnable task = new Task(result);
Future<Result> future = executorService.submit(task, result);
result = future.get();
```

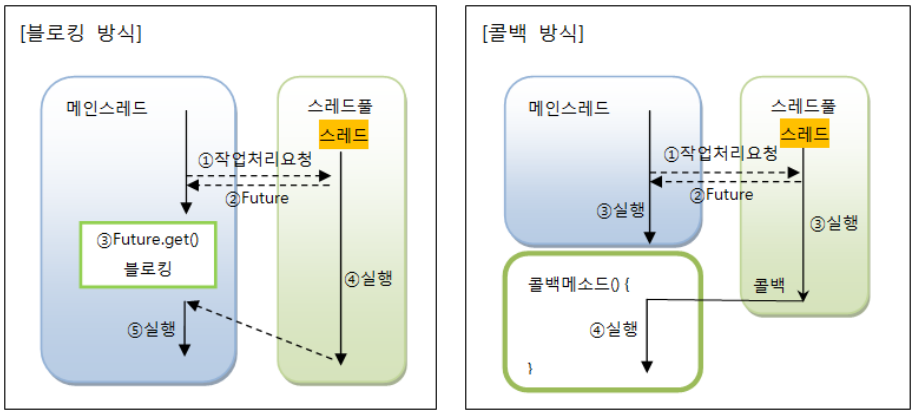
(4) 작업 완료 순으로 통보

- 작업 요청 순서대로 작업 처리가 완료되는 것은 아님.
- 작업 처리가 완료된 것부터 결과를 얻어 이용하는 것이 좋음.
- 스레드풀에서 작업 처리가 완료된 것만 통보받는 방법 --> CompletionService를 이용

```
ExecutorService executorService = Executors.newFixedThreadPool( Runtime.getRuntime().availableProcessors() );
CompletionService<V> completionService = new ExecutorCompletionService<V>( executorService );

completionService.submit(Callable<V> task);
completionService.submit(Runnable task, V result);
```

12.9.4 콜백 방식의 작업 완료 통보

- 콜백의 개념
  - 애플리케이션이 스레드에게 작업 처리를 요청한 후, 다른 기능 수행할 동안 스레드가 작업을 완료하면 애플리케이션의 메소드를 자동 실행하는 기법
  - 이때 자동 실행되는 메소드를 콜백 메소드
- 작업 완료 통보 얻기
  - 블로킹 방식: 작업 처리를 요청한 후 작업이 완료될 때까지 블로킹된다.
  - 콜백 방식: 결과를 기다릴 필요 없이 다른 기능을 수행한다. 작업 처리가 완료되면 자동적으로 콜백 메소드가 실행된다.




```
// 콜백 메소드를 가진 콜백 객체
CompletionHandler<V, A> callback = new CompletionHandler<V, A>() {
   @Override
   public void completed(V result, A attachment) {}
   @
   public void failed(Throwable exc, A attachment) {}
};

// 콜백 메소드를 호출하는 Runnable 객체
Runnable task = new Runnable() {
   @Override
   public void run() {
      try {
         //작업 처리
         v result = ...;
         callback.completed(result, null);  // 작업을 정상 처리했을 경우 호출
      } catch(Exception e) {
         callback.failed(e, null);          // 에러가 발생 했을 경우 호출
      }
   }
```

[CallbackExample.java] 콜백 방식의 작업 완료 통보받기

```
01    package sec09.exam03_callback;
02
03    import java.nio.channels.CompletionHandler;
04    import java.util.concurrent.ExecutorService;
05    import java.util.concurrent.Executors;
06
07    public class CallbackExample {
08              private ExecutorService executorService;
09
10              public CallbackExample() {
11                        executorService = Executors.newFixedThreadPool(
12                                  Runtime.getRuntime().availableProcessors()
13                        );
14              }
15
16              // 콜백 메소드를 가진 CompletionHandler 객체 생성
17              private CompletionHandler<Integer, Void> callback = new CompletionHandler<Integer, Void>() {
18                        @Override
19                        public void completed(Integer result, Void attachment) {
20                                  System.out.println("completed() 실행: " + result);
21                        }
22
23                        @Override
```

```
24                      public void failed(Throwable exc, Void attachment) {
25                              System.out.println("failed() 실행: " + exc.toString());
26                      }
27              };
28
29              public void doWork(final String x, final String y) {
30                      Runnable task = new Runnable() {
31                              @Override
32                              public void run() {
33                                      try {
34                                              int intX = Integer.parseInt(x);
35                                              int intY = Integer.parseInt(y);
36                                              int result = intX + intY;
37                                              callback.completed(result, null);
38                                      } catch(NumberFormatException e) {
39                                              callback.failed(e, null);
40                                      }
41                              }
42                      };
43                      executorService.submit(task);  // 스레드풀에게 작업 처리 요청
44              }
45
46              public void finish() {
47                      executorService.shutdown();  // 스레드풀 종료
48              }
49
50              public static void main(String[] args) {
51                      CallbackExample example = new CallbackExample();
52                      example.doWork("3", "3");
53                      example.doWork("3", "삼");
54                      example.finish();
55              }
56      }
```

[과제] 확인문제

1. 스레드에 대한 설명 중 틀린 것은 무엇입니까?

(1) 자바 애플리케이션은 메인(main) 스레드가 main() 메소드를 실행시킨다.
(2) 작업 스레드 클래스는 Thread 클래스를 상속해서 만들 수 있다.
(3) Runnable 객체는 스레드가 실행해야 할 코드를 가지고 있는 객체라고 볼 수 있다.
(4) 스레드 실행을 시작하려면 run() 메소드를 호출해야 한다.

2. 동영상과 음악을 재생하기 위해 두 가지 스레드를 실행하려고 합니다. 비어 있는 부분에 적당한 코드를 넣어 보세요.

[ThreadExample.java]

```
01  package verify.exam02;
02
03  public class ThreadExample {
04          public static void main(String[] args) {
05                  Thread thread1 = new MovieThread();
06                  thread1.start();
07
08                  Thread thread2 = new Thread( ____#1____ );
09                  thread2.start();
```

```
10            }
11    }
```

[MovieThread.java]

```
01    package verify.exam02;
02
03    public class MovieThread ________#2_______ {
04            @Override
05            public void run() {
06                    for(int i=0;i<3;i++) {
07                            System.out.println("동영상을 재생합니다.");
08                            try {
09                                    Thread.sleep(1000);
10                            } catch (InterruptedException e) {
11                            }
12                    }
13            }
14    }
```

[MusicRunnable.java]

```
01    package verify.exam02;
02
03    public class MusicRunnable ________#3_______ {
04            @Override
05            public void run() {
06                    for(int i=0;i<3;i++) {
07                            System.out.println("음악을 재생합니다.");
08                            try {
09                                    Thread.sleep(1000);
10                            } catch (InterruptedException e) {
11                            }
12                    }
13            }
14    }
```

3. 스레드의 우선순위에 대한 설명 중 틀린 것은 무엇입니까?

(1) 우선순위가 높은 스레드가 실행 기회를 더 많이 가질 수 있다.
(2) 우선순위는 1부터 10까지 줄 수 있는데, 디폴트는 5이다.
(3) Thread 클래스는 NORM_PRIORITY, MIN_PRIORITY, MAX_PRIORITY 상수를 제공한다.
(4) 1은 가장 높은 우선순위이기 때문에 다른 스레드보다 실행 기회를 더 많이 갖는다.

4. 동기화 메소드와 동기화 블록에 대한 설명 중 틀린 것은 무엇입니까?

(1) 동기화 메소드와 동기화 블록은 싱글(단일) 스레드 환경에서는 필요 없다.
(2) 스레드가 동기화 메소드를 실행할 때 다른 스레드는 일반 메소드를 호출할 수 없다.
(3) 스레드가 동기화 메소드를 실행할 때 다른 스레드는 다른 동기화 메소드를 호출할 수 없다.
(4) 스레드가 동기화 블록을 실행할 때 다른 스레드는 다른 동기화 메소드를 호출할 수 없다.

5. 스레드 일시 정지 상태에 대한 설명 중 틀린 것은 무엇입니까?

(1) 일시 정지 상태는 BLOCKED, WAITING, TIMED_WAITING이 있다.
(2) 스레드가 동기화 메소드를 실행할 때 다른 스레드가 동기화 메소드를 호출하게 되면 BLOCKED 일시정지상태가 된다.
(3) 동기화 메소드 내에서 wait()를 호출하면 WAITING 일시 정지 상태가 된다.
(4) yield() 메소드를 호출하면 TIMED_WAITING 일시 정지 상태가 된다.

6. 스레드 상태 제어를 하는 메소드에 대한 설명 중 틀린 것은 무엇입니까?

(1) yield() 메소드를 호출한 스레드는 동일한 우선순위나 높은 우선순위의 스레드에게 실행 기회를 양보하고 자신은 실행 대기 상태가 된다.
(2) sleep() 메소드를 호출한 스레드는 주어진 시간 동안 일시 정지 상태가 된다.
(3) stop() 메소드는 스레드를 즉시 종료시키기 때문에 스레드 안전성에 좋지 못하다.
(4) join() 메소드를 호출한 스레드가 종료할 때까지 join() 메소드를 멤버로 가지는 스레드는 일시 정지 상태가 된다.

7. interrupt() 메소드를 호출한 효과에 대한 설명 중 틀린 것은 무엇입니까?

(1) 일시 정지 상태에서 InterruptedException를 발생시킨다.
(2) 스레드는 즉시 종료한다.
(3) 실행 대기 상테에서 호출되면 일시 정지 상태가 될 때까지 InterruptedException이 발생하지 않는다.
(4) 아직 InterruptedException이 발생하지 않았다면 interrupted(), isInterrupted() 메소드는 true를 리턴한다.

8. 메인 스레드에서 1초 후 MovieThread의 interrupt() 메소드를 호출해서 MovieThread를 안전하게 종료하고 싶습니다. 비어 있는 부분에 적당한 코드를 작성해보세요.

[MovieThread.java]

```
01  package verify.exam08;
02
03  public class MovieThread extends Thread {
04          @Override
05          public void run() {
06                  while(true) {
07                          System.out.println("동영상을 재생합니다.");
08                          // 작성 위치
09
10                  }
11          }
12  }
```

9. wait()와 notify() 메소드에 대한 설명 중 틀린 것은 무엇입니까?

(1) 스레드가 wait()를 호출하면 일시 정지 상태가 된다.
(2) 스레드가 notify()를 호출하면 wait()로 일시 정지 상태에 있던 다른 스레드가 실행 대기 상태가 된다.
(3) wait()와 notify()는 동기화 메소드 또는 블록에서 호출할 필요가 없다.
(4) 스레드가 wait(long millis)를 호출하면 notify()가 호출되지 않아도 주어진 시간이 지나면 자동으로 실행 대기 상태가 된다.

10. 메인 스레드가 종료하면 MovieThread도 같이 종료되게 만들고 싶습니다. 비어 있는 부분에 적당한 코드를 넣어 보세요.

[ThreadExample.java]

```
01  package verify.exam10;
02
03  public class ThreadExample {
04          public static void main(String[] args) {
05                  Thread thread = new MovieThread();
06                  (           #1          )
07                  thread.start();
08
09                  try { Thread.sleep(3000); } catch (InterruptedException e) {}
10          }
11  }
```

[MovieThread.java]

```
01  package verify.exam10;
02
03  public class MovieThread extends Thread {
04          @Override
05          public void run() {
06                  while(true) {
07                          System.out.println("동영상을 재생합니다.");
08                          try { Thread.sleep(1000); } catch (InterruptedException e) {}
09                  }
10          }
11  }
```

11. while문으로 반복적인 작업을 하는 스레드를 종료시키는 방법에 대한 설명 중 최선의 방법이 아닌 것은?

(1) stop() 메소드를 호출해서 즉시 종료시킨다.
(2) 조건식에 boolean 타입의 stop 플래그를 이용해서 while문을 빠져나가게 한다.
(3) 스레드가 반복적으로 일시 정지 상태가 된다면 InterruptedException을 발생시켜 예외 처리 코드에서 break문으로 while문을 빠져나가게 한다.
(4) 스레드가 일시 정지 상태로 가지 않는다면 isInterrupted()나 interrupted() 메소드의 리턴 값을 조사해서 true인 경우 break문으로 while문을 빠져나가게 한다.

12. 스레드풀에 대한 설명 중 틀린 것은 무엇입니까?

(1) 갑작스러운 작업의 증가로 스레드의 폭증을 막기 위해 사용된다.
(2) ExecutorService 객체가 스레드풀이며 newFixedThreadPool() 메소드로 얻을 수 있다.
(3) 작업은 Runnable 또는 Callable 인터페이스를 구현해서 정의한다.
(4) submit() 메소드로 작업 처리 요청을 하면 작업이 완료될 때까지 블로킹된다.

13. Future 객체에 대한 설명 중 틀린 것은 무엇입니까?

(1) Future는 스레드가 처리한 작업의 결과값을 가지고 있는 객체이다.
(2) submit() 메소드를 호출하면 즉시 리턴되는 객체이다.
(3) Future의 get() 메소드는 스레드가 작업을 완료하기 전까지 블로킹된다.
(4) CompletionService를 이용하면 작업 완료된 순으로 Future를 얻을 수 있다.