12 Multi-Threads

Object-Oriented Programming

페이지 1

멀티 쓰레드 강의 시작 하겠습니다.

Program, Process, Multiprocessing

- Program
  - 실행되지 않은 상태의 소프트웨어
  - Disk에 파일형태로 존재, Memory에 load 후 실행
- Process
  - 실행 중인 program의 instance (OS에 의해 생성)
  - 자신만의 독립적 memory space (code, data, heap, stack)
- Multiprocessing
  - 여러 개의 process를 동시에 실행하여, parallel로 작업 수행
  - ex) 웹 서버를 multiprocessing하면 각 사용자의 요청을 동시에 서비스

페이지 2

먼저 program, process, multiprocessing 이라는
용어들에 대해 살펴 보겠습니다.
프로그램은 아직 실행되지 않은 소프트웨어로,
디스크에 파일로 저장되어 있습니다.
일단 memory에 load하면 실행할 준비가 된 것입니다.
process는 프로그램이 실행되기 시작할 때 나타나는 것으로,
OS에서 관리하는 active instance입니다.
각 process에는 code, data, heap, stack을 위한 섹션이
포함된
별도의 memory 공간이 있습니다.
이러한 분리는 체계적이고 safe하게
프로그램 실행의 상태를 유지하는 데 도움이 됩니다.
multiprocessing이란 여러 process를 동시에 실행하여
작업을 parallel로 처리하는 것을 말합니다.
예를 들어 웹 서버에서 multiprocessing을 사용하면

각 사용자 요청을 동시에 처리할 수 있으므로
서버가 대기 시간 없이 여러 사용자를 한 번에 처리할 수 있습니다.

Thread and Multithreading

- Thread
  - Process 내에서 실행되는 작업의 단위
  - 하나의 process는 여러 개의 thread를 가질 수 있음
  - ex) Video Game
    - Main thread: game의 main loop 처리, game logic, input과 event 처리
    - Rendering thread: 3D graphics or 2D graphics를 display에 rendering
    - AI thread: NPC (Non-Player Character)의 AI 행동 계산
- Multithreading
  - 하나의 process 내에서 여러 thread를 생성, 동시에 여러 작업을 수행
  - thread간 memory 공유
  - Synchronization, deadlock, race condition 등 위험 존재
  - ex) Text Editor: UI, auto-completion, syntax analysis, file backup, ...

페이지 3

Thread는 하나의 Process 내에서 실행되는 작업의 단위입니다.
하나의 process 안에는 여러 개의 thread가 있을 수 있습니다.
예를 들어, Video Game에서는
다음과 같이 다양한 thread들이 작업을 나눠서 처리합니다:

먼저 Main thread는 game의 메인 루프를 처리하고,

게임 logic, 입력 및 이벤트를 처리합니다.

Rendering thread는 3D 그래픽이나 2D 그래픽을 display에 렌더링합니다.

AI thread는 NPC, 즉, Non-Player Character의 AI 행동을 계산하는 역할을 합니다.
Multithreading은 하나의 process 내에서 여러 thread를 생성하여
동시에 여러 작업을 수행하는 방식입니다.

여러 thread가 memory를 공유하기 때문에 작업을 효율적으로 처리할 수 있지만,
Synchronization 문제나 Deadlock, Race condition과 같은 위험이 존재할 수 있습니다.
예를 들어, Text Editor에서는
UI, 자동 완성, 구문 분석, 파일 백업 등의 작업을 각각의 thread로 처리하여
더 효율적인 작업 수행을 가능하게 합니다.

Task and Multitasking

- Task
  - 수행해야 할 작업이나 명령어의 단위를 의미
  - thread 또는 process에 의해 실행 됨
  - OS에서 scheduling 단위로 사용
  - Process나 Thread 보다 추상적인 개념
- Multitasking
  - OS가 여러 task (process 또는 thread) 를 동시에 실행
  - CPU resource의 효율적 사용을 위해 task가 번갈아 실행됨
  - 종류
    - Process-based multitasking (= multiprocessing)
    - Thread-based multitasking (= multithreading)

페이지 4

Task는 수행해야 할 작업이나 명령어의 단위를 의미합니다.
이는 thread 또는 process에 의해 실행되며, OS에서 scheduling의 단위로 사용됩니다.
Process나 Thread보다 더 추상적인 개념입니다.
Multitasking은 OS가 여러 task, 즉, process 또는 thread를 동시에 실행하는 것을 말합니다.
OS는 CPU resource을 효율적으로 사용하기 위해 여러 task가 번갈아 가며 실행되도록 합니다.
Multitasking의 종류는 다음과 같습니다:

Process-based multitasking 또는 multiprocessing은

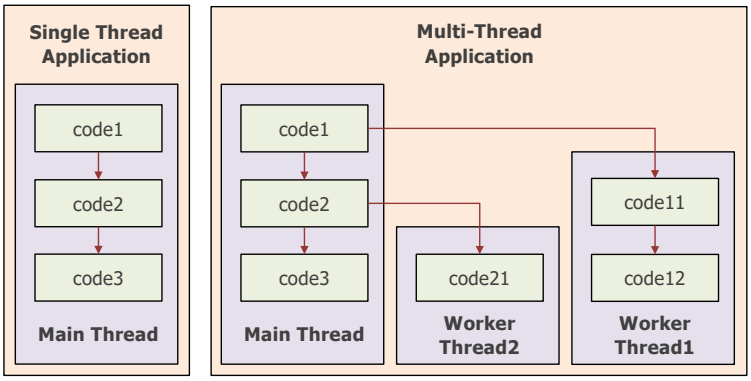
여러 개의 process를 사용하여 multitasking을 수행하는 방식입니다.

반면에, Thread-based multitasking 또는 multithreading은

하나의 process 내에서 여러 thread를 사용하여
multitasking을 수행하는 방식입니다.

Main Thread and Worker Threads

- Single Thread Application
  - Main Thread 하나만 존재
    - code1, code2, code3
- Multi Thread Application
  - Main Thread
    - code1 (Worker Thread1 분기)
    - code2 (Worker Thread2 분기)
    - code3
  - Worker Thread1
    - code11
    - code12
  - Worker Thread2
    - code21



페이지 5

Single Thread Application에서는 Main Thread 하나만 존재하여
code1, code2, code3 순서로 code가 순차적으로 실행됩니다.
반면, Multi Thread Application에서는
Main Thread와 Worker Thread들이 함께 작업을 수행합니다.
Main Thread는 code1을 실행하면서 Worker Thread1을 분기하여
code11과 code12를 처리하도록 합니다.
이어서 Main Thread는 code2를 실행하면서
Worker Thread2를 분기하여 code21을 처리하게 합니다.
마지막으로 Main Thread는 code3을 실행합니다.
이처럼 Multi Thread Application에서는
Main Thread가 실행 중 특정 지점에서 Worker Thread를 생성하여,
동시에 다양한 작업을 parallel하게 수행할 수 있습니다.

Example: main thread only

```java
import java.awt.Toolkit;

public class BeepPrintExample {
    public static void main(String[] args) {
        Toolkit toolkit = Toolkit.getDefaultToolkit();
        for(int i=0; i<5; i++) {
            toolkit.beep(); // beep음 발생
            try { Thread.sleep(500); } // 0.5초 일시 정지
            catch(Exception e) {}
        }
        for(int i=0; i<5; i++) {
            System.out.println("띵");
            try { Thread.sleep(500); }
            catch(Exception e) {}
        }
    }
}
```

OUTPUT:
(0.5초 간격으로) beep 5번
(0.5초 간격으로) 아래 print
띵
띵
띵
띵
띵

페이지 6

이 프로그램은 Java의 Toolkit class를 사용하여
일정한 간격으로 beep 소리를 발생시키고,
그 후에 콘솔에 "띵"이라는 메시지를 출력하는 예제입니다.
BeepPrintExample class의 main method에서 프로그램이
시작됩니다.
먼저, Toolkit 객체를 생성하여 시스템의 기본 도구 모음을
얻습니다.
그런 다음, 첫 번째 for 반복문에서 toolkit.beep() method를
call하여
beep 소리를 다섯 번 발생시킵니다.
각각의 beep 소리 사이에는 Thread.sleep(500); method를
통해
0.5초 동안 일시 정지하여 일정한 간격을 유지합니다.
exception이 발생할 경우 catch block에서 처리되지만,
이 예제에서는 아무 작업도 수행하지 않습니다.

beep 소리가 끝나면 두 번째 for 반복문으로 넘어가
콘솔에 "띵"이라는 문자열을 다섯 번 출력합니다.
이 반복문 역시 Thread.sleep(500); method를 사용하여
출력 간에 0.5초 간격을 둡니다.
이 프로그램은 Single Thread Application으로
Main Thread 하나만 존재하여
beep 소리 출력 작업과 "띵" 메시지 출력 작업을 순차적으로
수행합니다.

Example: main + worker threads

```
import java.awt.Toolkit;

public class BeepPrintExample2 {
    public static void main(String[] args) {
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                Toolkit toolkit = Toolkit.getDefaultToolkit();
                for(int i=0; i<5; i++) {
                    toolkit.beep();
                    try { Thread.sleep(500); } catch(Exception e) {}
                }
            }
        });
        thread.start(); // worker thread 실행
        for(int i=0; i<5; i++) {
            System.out.println("띵");
            try { Thread.sleep(500); } catch(Exception e) {}
        }
    }
}
```

worker thread 생성

worker thread code

main thread code

페이지 7

이 프로그램 BeepPrintExample2는 main thread와 worker thread를 사용하여
동시에 beep 소리와 "띵" 메시지를 출력하는 예제입니다.
main method에서 프로그램이 시작되며, 먼저 worker thread를 생성합니다.
이 worker thread는 Runnable interface를 구현한 anonymous class를 통해 정의됩니다.
worker thread 안에서는 Toolkit 객체를 생성하여 beep 소리를 다섯 번 발생시키며,
각각의 beep 사이에는 0.5초의 간격을 두기 위해 Thread.sleep(500);이 사용됩니다.
예외가 발생할 경우 catch block에서 처리되지만, 여기서는 아무 작업도 하지 않습니다.
worker thread는 thread.start();를 call함으로써 비로소 실행됩니다.

Main Thread에서는 worker thread가 실행되는 동안
System.out.println("띵");을 통해 "띵"이라는 메시지를 다섯 번 출력합니다.
이 역시 Thread.sleep(500);을 사용하여 출력 간격을 0.5초로 맞춥니다.
이 프로그램은 Multi Thread Application으로,
Main Thread가 "띵" 메시지를 출력하는 동안
Worker Thread는 beep 소리를 동시에 발생시킵니다.
이전의 Single Thread Application에서는 Main Thread 하나만 사용하여
beep 소리 출력과 "띵" 메시지 출력을 순차적으로 수행했습니다.
즉, beep 소리가 다 끝난 후에 "띵" 메시지가 출력되었습니다.
반면, 이 Multi Thread Application에서는
Main Thread와 Worker Thread가 동시에 실행됩니다.
Main Thread는 "띵" 메시지를 출력하고,
Worker Thread는 beep 소리를 발생시켜 두 작업이 parallel로 수행됩니다.
이로 인해 beep 소리와 "띵" 메시지가 동시에 나타나는 차이가 있습니다.

Creating Thread

1) Thread class로 직접 생성
   - java.lang.Thread class 에서 worker class를 직접 생성
   - Runnable implements object를 parameter로 하는 constructor를 call

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

2) Thread Child Class로 생성
   - Thread class inherit, run() method를 override, 실행 code를 run()에 작성

```
public class WorkerThread extends Thread {
    @Override
    public void run() {
        // thread가 실행할 code
    }
}
// thread object 생성
Thread thread = new WorkerThread();
```

```
// anonymous class 이용
Thread thread = new Thread() {
    @Override
    public void run() {
        // thread가 실행할 code
    }
};
thread.start();
```


페이지 8

Java에는 Thread를 생성하는 두 가지 방법이 있습니다.
첫 번째 방법은 Thread class를 직접 생성하는 방식입니다.
java.lang.Thread class를 사용하여 worker class를 직접 생성하며,
이때 Runnable interface을 구현한 객체를 생성자의 파라미터로 전달합니다.
예를 들어, Thread thread = new Thread(Runnable target);과 같이
Runnable 객체를 전달할 수 있습니다.
참고로 Runnable interface는
void run() 이라는 abstract method 하나만을 가지고 있습니다.

두 번째 방법은 Thread의 Child Class를 생성하는 방식입니다.
이 방법에서는 Thread class를 상속하여 run() method를 override하고,

해당 method에 실행할 code를 작성합니다.
아래와 같이 두 가지 방식으로 작성할 수 있습니다.

첫번째는 일반 class 방식으로, WorkerThread라는 class를 만들고

Thread class를 상속받아 run() method를 override합니다.

그런 다음, Thread thread = new WorkerThread()와 같이

thread object를 생성하여 실행할 수 있습니다.

두번째는 anonymous class 방식으로,

new Thread() { public void run() { /* 실행할 code */ } } 형태로
작성하여

바로 run() method를 override하고,

thread.start();를 call하여 thread를 실행할 수 있습니다.

Thread's Name

- Worker thead의 default name: Thread-n
- 다른 name으로 설정: Thread class의 setName() method

```
thread.setName("myThread");
```

- Naming: Debugging할 때 어떤 thread가 작업을 하는지 조사하는데 유용
- Thread.currentThread(): 현재 작업하는 thread의 reference return
- getName(): thread의 name return

```
Thread thread = Thread.currentThread();
System.out.println(thread.getName());
```


페이지 9

이 슬라이드는 Thread's Name에 대해 설명하고 있습니다.
기본적으로 Worker thread는 Thread-n 형태의 기본 이름을 가지며,
n은 정수로 0부터 Worker thread가 하나 create될 때마다 자동 증가합니다.
그러나 Thread class의 setName() method를 사용하여 다른 이름으로 설정할 수 있습니다.
예를 들어, thread.setName("myThread");와 같이 작성하면 thread의 이름이 "myThread"로 설정됩니다.
이름을 설정하는 것은 디버깅할 때 유용합니다.
이를 통해 어떤 thread가 특정 작업을 수행 중인지 쉽게 확인할 수 있습니다.
현재 실행 중인 thread를 참조하려면 Thread.currentThread()를 사용하고,
이 thread의 이름을 얻으려면 getName() method를 call할 수

있습니다.
예를 들어, Thread thread = Thread.currentThread();
System.out.println(thread.getName());와 같이 작성하면
현재 thread의 이름이 출력됩니다.

ThreadNameExample

```java
public class ThreadNameExample {
    public static void main(String[] args) {
        Thread mainThread = Thread.currentThread();
        System.out.println(mainThread.getName() + " 실행");
        for(int i=0; i<3; i++) {
            Thread threadA = new Thread() {
                @Override
                public void run() {
                    System.out.println(getName() + " 실행");
                }
            };
            threadA.start();
        }
        Thread chatThread = new Thread() {
            @Override
            public void run() {
                System.out.println(getName() + " 실행");
            }
        };
        chatThread.setName("chat-thread");
        chatThread.start();
    }
}
```

OUTPUT:
main 실행
Thread-1 실행
Thread-2 실행
Thread-0 실행
chat-thread 실행

OUTPUT:
main 실행
Thread-0 실행
Thread-1 실행
Thread-2 실행
chat-thread 실행

OUTPUT:
main 실행
Thread-1 실행
Thread-0 실행
Thread-2 실행
chat-thread 실행

OUTPUT:
main 실행
Thread-1 실행
chat-thread 실행
Thread-0 실행
Thread-2 실행

OUTPUT:
main 실행
Thread-1 실행
Thread-0 실행
chat-thread 실행
Thread-2 실행

OUTPUT:
main 실행
Thread-0 실행
Thread-2 실행
chat-thread 실행
Thread-1 실행

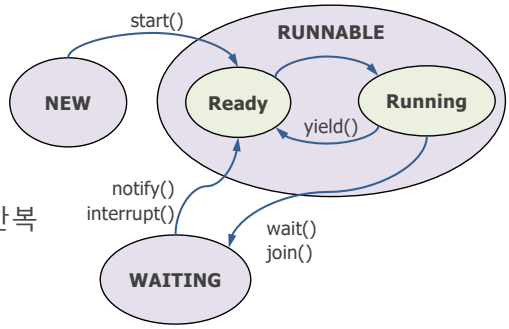
페이지 10

여기에서는 ThreadNameExample이라는 class를 통해
Java Thread의 이름 설정과 실행 순서를 설명하고 있습니다.
이 예제에서, main method가 시작되면
Main Thread의 이름을 가져와 main 실행이라는 메시지를 출력합니다.
그 후, for 반복문을 통해 세 개의 Thread 객체 threadA를 생성하고 각각 실행합니다.
threadA는 기본 이름인 Thread-0, Thread-1, Thread-2로 순차적으로 할당되며,
getName()을 통해 각 thread 이름과 함께 "실행" 메시지를 출력합니다.
반복문이 끝난 후, chatThread라는 이름의 새로운 Thread가 생성됩니다.
chatThread는 setName("chat-thread"); method를 사용하여 이름을 "chat-thread"로 설정하고,

start()를 call하여 "chat-thread 실행" 메시지를 출력합니다.
오른쪽에 있는 여러 출력 예시는
Thread의 실행 순서가 항상 일정하지 않음을 보여줍니다.
Java에서 Thread는 parallel로 실행되므로,
각 Thread가 실행되는 순서는 다를 수 있습니다.
이를 통해 Thread의 실행 순서는
OS scheduler에 따라 결정된다는 점을 알 수 있습니다.

Thread's States (1/2)

- NEW
  - Thread 생성됨. 아직 start 되지 않은 상태
  - start() method에 의해 RUNNABLE의 Ready로 전환
- RUNNABLE: CPU scheduling에 의해 아래 두 state를 반복
  - Ready: 실행을 기다리고 있는 상태.
  - Running: CPU를 점유, run() method를 실행하는 상태
    - Scheduling 또는 yield()로 Ready로 전환
- WAITING: Thread가 다른 thread의 작업 종료를 무한히 대기
  - wait(), join() 에 의해 진입
  - notify(), interrupt()에 의해 해제, RUNNABLE로 돌아감



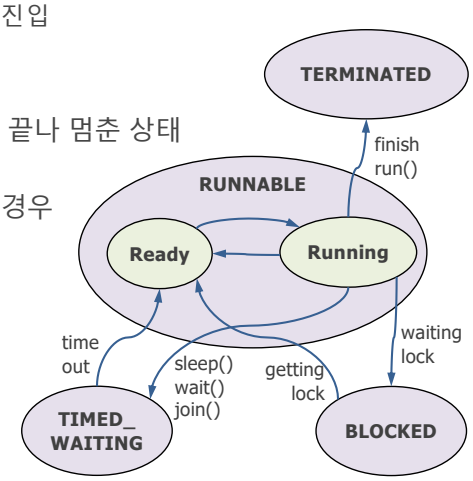
페이지 11

Java의 Thread는 실행 중에 상황에 따라
여러가지 state를 가지고 있습니다.
먼저, NEW 상태는 Thread가 생성된 직후의 상태를
의미합니다.
이때는 start() method를 call하기 전이므로
Thread가 아직 실행되지 않은 상태입니다.
start() method를 call하면 Thread는 RUNNABLE state로
전환됩니다.
RUNNABLE state는 CPU scheduling에 의해 실행될 준비가
되었음을 나타내며,
이 state에서는 다시 두 가지 하위 state를 오가게 됩니다.
첫 번째는 Ready state로, 이 state에서는 실행을 기다리고
있으며,
CPU resource을 사용할 준비가 되어 있습니다.
두 번째는 Running state로,

CPU resource을 얻어 run() method를 실제로 실행하는 state입니다.
OS는 CPU를 여러 thread에게 공유하게 하기 위해
적절한 scheduling으로 thread들을
Ready와 Running을 오가게 합니다.
OS의 CPU scheduling에 의하지 않더라도
만약 Running상태의 thread가 yield()를 call하면
자신은 Running state에서 다시 Ready state로 돌아가게 되며
다음 scheduling 될 thread에게 CPU를 양보하게 됩니다.
그 다음, WAITING state는 다른 Thread의 작업이 끝날 때까지
대기하는 state입니다.
예를 들어, wait()나 join() method를 call하면
Thread는 WAITING state로 들어가게 되어
CPU resource을 사용하지 않고 무한히 대기합니다.
WAITING 상태에 있는 Thread는
다른 thread가 call하는 notify()나 interrupt() method에 의해
다시 activate될 수 있으며,
이 경우 RUNNABLE의 Ready state로 돌아갑니다.

Thread's States (2/2)

- TIMED_WAITING: Thread가 정해진 시간동안 대기
  - sleep(), wait() (시간 지정시), join() (시간 지정시) 에 의해 진입
  - 정해진 시간 후에 해제, RUNNABLE로 돌아감
- TERMINATED
  - Running에서 run() method가 종료. Thread의 실행이 다 끝나 멈춘 상태
- BLOCKED
  - Synchronization method (또는 block) 으로 Lock이 걸린 경우
  - Lock과 Unlock으로 진입과 해제



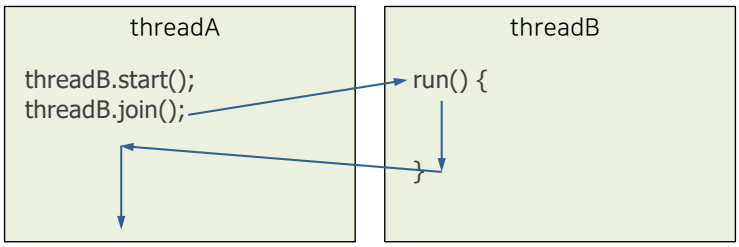
페이지 12

TIMED_WAITING state는 Thread가 정해진 시간 동안
대기하는 상태입니다.
sleep() 또는 시간이 지정된 wait()나 join() method를
call하면서
이 state에 진입하게 됩니다.
지정된 시간이 지나면 thread는 자동으로 해제되어
RUNNABLE의 Ready state로 돌아가게 됩니다.
다음으로, TERMINATED state는 Thread의 실행이 모두
완료되어
더 이상 실행되지 않는 상태입니다.
이는 run() method가 종료될 때 도달하는 상태로,
Thread의 수명이 끝났음을 의미합니다.
마지막으로, BLOCKED state는
Thread가 다른 Thread에 의해 lock되어 대기 중인 state입니다.
Thread가 synchronized block이나 synchronized method에

진입하려고 할 때
해당 block이나 method에 다른 Thread가 lock을 걸어놓은 경우
BLOCKED state가 됩니다.
다른 Thread가 lock을 해제, 즉, unlock하면
BLOCKED Thread는 해당 block이나 method를 획득하고
RUNNABLE의 Ready state로 돌아갑니다.

Waiting for Other Thread

- 다른 thread의 종료를 기다렸다가 그 결과를 받아 처리하는 경우
- threadA
  - threadB.start()
  - threadB.join() // threadA는 일시 정지 (WAIT 상태), threadB의 종료를 기다림
  - threadB 종료 후 계속 진행
- threadB
  - run() 실행 후 종료



페이지 13

다른 Thread의 종료를 기다렸다가 그 결과를 처리하는 상황에 대해 설명하고 있습니다.
여기서 threadA는 threadB가 완료될 때까지 기다리는 역할을 합니다.
먼저 threadA에서 threadB를 시작하기 위해 threadB.start()를 call합니다.
그런 다음, threadB.join() method를 call하여 threadA는 WAITING 상태로 들어가
threadB의 종료를 기다립니다.
threadB는 run() method를 실행한 후 종료됩니다.
이때 threadA는 threadB가 종료될 때까지 대기하므로,
threadB가 종료되면 threadA는 WAITING 상태에서 빠져나와
다시 실행을 이어갈 수 있게 되고
이후의 작업을 계속 진행하게 됩니다.

Example: Waiting for Other Thread

```
public class SumThread extends Thread {
    private long sum;
    public long getSum() { // accessor
        return sum;
    }
    @Override
    public void run() {
        for(int i=1; i<=100; i++) {
            sum += i;
        }
    }
}
```

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

OUTPUT:
1~100 합: 5050

페이지 14

여기에서는 Thread의 종료를 기다리는 예제를 보여주고 있습니다.
SumThread class와 SumThreadJoin class가 이를 구현합니다.
SumThread class는 Thread를 상속받아 sum이라는 변수를 통해 1부터 100까지의 합을 계산하는 역할을 합니다.
getSum() method는 sum 값을 반환하는 접근자 method입니다.
run() method에서는 for 반복문을 통해 sum에 1부터 100까지의 값을 더해 나갑니다.
SumThreadJoin class의 main method에서는 SumThread 객체를 생성하고 sumThread.start()로 sumThread를 시작합니다.
이 후 sumThread.join()을 call하여 main thread가 WAITING 상태에 들어가

sumThread가 작업을 끝낼 때까지 대기합니다.
sumThread가 완료되면 main thread는 대기 상태에서 벗어나게 되며,
sumThread.getSum()을 call하여 1부터 100까지의 합계를 출력합니다.
이 예제의 출력 결과는 1~100 합: 5050으로,
sumThread가 완료된 후 main thread가 sum 값을 출력하는 것을 확인할 수 있습니다.

Yielding Execution to Another Thread

```
public void run() {
    while(true) {
        if (work)
            System.out.println("threadA 작업");
    }
}
```

- work가 false인 경우
- while문은 의미 없는 반복
- 이 thread는 아무 것도 하지 않고 있음

```
public void run() {
    while(true) {
        if (work)
            System.out.println("threadA 작업");
        else
            Thread.yield();
    }
}
```

- 의미 없는 반복 없이
- work가 false인 경우 다른 thread에게 yield (양보)

페이지 15

여기에서는 Thread 실행을 다른 Thread에 양보하는 방법을 보여줍니다.
위쪽 code에서는 run() method가 무한 반복을 수행하는 동안 if 문으로 work 조건을 검사합니다.
만약 work가 true이면 "threadA 작업"이라는 메시지를 출력하지만,
work가 false이면 아무 작업도 수행하지 않고
계속해서 의미 없는 반복을 하게 됩니다.
이 경우 Thread는 아무런 작업을 하지 않으면서도
계속해서 CPU resource을 차지하고 있습니다.
아래쪽 code에서는 Thread.yield() method를 사용하여,
work가 false일 경우 현재 Thread가 실행을 멈추고
다른 Thread에게 CPU resource을 양보하도록 합니다.
이렇게 하면 다른 Thread들이 CPU resource을 사용할 수 있으므로,

불필요한 CPU resource 낭비를 줄일 수 있습니다.
요약하자면, work가 false일 때 Thread.yield()를 call함으로써
의미 없는 반복을 줄이고
다른 Thread에게 CPU resource을 양보하는 효과를 얻을 수
있습니다.

Example: YieldExample (1/2)

```
public class YieldExample {
    public static void main(String[] args) {
        WorkThread workThreadA = new WorkThread("workThreadA");
        WorkThread workThreadB = new WorkThread("workThreadB");
        workThreadA.start();
        workThreadB.start();


        // 5초 후 workThreadA가 yield 시작, workThreadB가 더 많이 실행 됨
        try { Thread.sleep(5000); } catch (InterruptedException e) {}
        workThreadA.work = false;

        // 10초 후 workThreadA, workThreadB가 비슷한 횟수로 실행 됨
        try { Thread.sleep(10000); } catch (InterruptedException e) {}
        workThreadA.work = true;
    }
}
```


페이지 16

이 code는 YieldExample class에서 WorkThread라는
두 개의 thread를 생성하여 실행하는 예제입니다.
WorkThread class는 이름을 통해 식별되며,
내부적으로 work라는 flag를 통해
특정 상황에서 yield()를 call할지 여부를 결정합니다.
main method가 나타내는 main thread의 실행 흐름을
살펴보면
먼저, workThreadA와 workThreadB라는 두 개의 WorkThread
객체를 생성하고
start() method를 call하여 실행합니다.
각 thread는 자신의 작업을 parallel로 수행하기 시작합니다.

Thread.sleep(5000);을 통해 5초 동안 main thread를 일시
중지시킵니다.

5초가 지난 후, workThreadA의 work flag를 false로 설정하여,

workThreadA가 yield()를 call하도록 합니다.
이로 인해 workThreadA는 더 이상 CPU resource을 독점하지 않고
다른 thread (여기서는 workThreadB) 에게 CPU를 양보하게 됩니다.
결과적으로, workThreadB가 CPU를 더 많이 사용하게 되어
상대적으로 더 자주 실행됩니다.
이후 Thread.sleep(10000);을 통해
다시 10초 동안 main thread를 일시 중지합니다.
10초가 지난 후 workThreadA의 work flag를 true로 설정하여,
workThreadA가 yield()를 더 이상 call하지 않고
workThreadB와 비슷한 빈도로 실행될 수 있도록 합니다.
이 프로그램의 목적은 yield()를 사용하여
특정 thread가 다른 thread에게 CPU를 양보하도록 제어함으로써
두 thread의 실행 빈도를 조정하는 방법을 보여주는 것입니다.
5초 후에는 workThreadB가 더 많이 실행되고,
10초 후에는 두 thread가 비슷한 횟수로 실행되도록 설정됩니다.

Example: YieldExample (2/2)

```
class WorkThread extends Thread {
    public boolean work = true; // 초기 work = true
    public WorkThread(String name) {
        setName(name);
    }
    @Override
    public void run() {
        while(true) {
            if(work) {
                System.out.println(getName() + ": 작업처리");
            } else {
                Thread.yield();
            }
        }
    }
}
```

OUTPUT:
workThreadB: 작업처리
...
workThreadA: 작업처리
...
workThreadB: 작업처리
...
workThreadA: 작업처리
...
workThreadA: 작업처리

페이지 17

이 code는 이전 슬라이드의 YieldExample class에서 사용하는 WorkThread class의 정의를 포함하고 있습니다.
WorkThread class는 Thread를 상속하며,
work라는 flag를 통해 특정 조건에서 yield()를 call하여 CPU resource을 다른 Thread에게 양보할지 여부를 제어합니다.
WorkThread class의 주요 구성 요소로는

먼저 work flag는 public boolean work = true;로 초기화되고,

Thread가 작업을 계속 수행할지,

아니면 yield()를 통해 CPU resource을 양보할지를 결정합니다.

생성자인 WorkThread(String name) 생성자는

Thread의 이름을 설정하기 위해 setName(name);을 call합니다.

이를 통해 각 Thread를 식별할 수 있게 합니다.

run() method는 Thread의 작업을 정의합니다.

무한 while 루프 안에서 work flag의 값에 따라

두 가지 동작을 수행합니다:

work가 true일 때, getName() method를 call하여 Thread의 이름을 가져오고,

System.out.println(getName() + ": 작업처리");을 통해

"작업처리" 메시지를 출력합니다.

즉, Thread가 실제 작업을 수행하는 상태입니다.

work가 false일 때는 Thread.yield();를 call하여

CPU resource을 다른 Thread에게 양보합니다.

이때 Thread는 WAITING 상태가 되어

다른 Thread가 CPU resource을 사용할 수 있게 됩니다.

이 class와 이전의 YieldExample code를 결합하여 실행하면

처음에는 workThreadA와 workThreadB 모두 work가 true이므로

두 Thread가 번갈아 가며 "작업처리" 메시지를 출력합니다.

5초 후, workThreadA의 work가 false로 설정되면서

workThreadA는 yield()를 call하여 CPU resource을 양보하게 됩니다.

이로 인해 workThreadB가 상대적으로 더 자주 실행되며

"workThreadB: 작업처리"가 더 많이 출력됩니다.

10초 후, workThreadA의 work가 다시 true로 설정되면서

yield()를 call하지 않게 되고,

workThreadA와 workThreadB가 비슷한 빈도로 "작업처리"를

출력합니다.

이 code의 목적은 yield() method를 사용하여

특정 Thread가 CPU resource을 양보하도록 제어함으로써

thread 간의 실행 빈도를 조절하는 방법을 보여주는 것입니다.

Thread Synchronization

- User1Thread와 User2Thread가 하나의 object (Calculator)를 공유하며 작업
- User1Thread
  - Calculator's memory = 100으로 set
  - 2초간 일시 정지
  - print Calculator's memory: 50으로 바뀌어 있음 (User2Thread가 변경)
- User2Thread
  - Calculator's memory = 50으로 set
  - 2초간 일시 정지

페이지 18

이 내용은 Thread Synchronization의 필요성을 보여주는 예제입니다.
User1Thread와 User2Thread가
하나의 객체인 Calculator를 공유하며 작업을 수행할 때
발생하는 문제를 설명합니다.

먼저 User1Thread이 Calculator 객체의 memory 값을 100으로 설정합니다.

이후 2초간 일시 정지합니다.

일시 정지 후 Calculator의 memory 값을 출력할 때,

기대했던 100이 아닌 50으로 변경되어 있는 것을 확인합니다.

이는 User2Thread가 memory 값을 변경했기 때문입니다.

한편 User2Thread는 User1Thread와

동일한 Calculator 객체의 memory 값을 50으로 설정합니다.

이후 2초간 일시 정지합니다.

두 Thread가 동일한 Calculator 객체를 공유하면서

memory 값을 설정하고 있기 때문에,

한 Thread가 memory 값을 변경하면

다른 Thread의 작업 결과에도 영향을 미칩니다.

이러한 경우 Thread Synchronization이 필요합니다.

Thread Synchronization을 통해

두 Thread가 memory 값을 safe하게 수정하고 읽을 수 있도록 하면

이러한 문제를 방지할 수 있습니다.
결국 이 예제는 Thread들이 공유 resource에 동시에 접근할 때
발생하는
충돌을 방지하기 위해
Thread Synchronization이 중요함을 보여줍니다.

Example: Non-SynchronizedExample1 (1/4)

```
public class SynchronizedExample1 {
    public static void main(String[] args) {
        Calculator1 calculator = new Calculator1();

        User1Thread1 user1Thread = new User1Thread1();
        user1Thread.setCalculator(calculator);
        user1Thread.start();

        User2Thread1 user2Thread = new User2Thread1();
        user2Thread.setCalculator(calculator);
        user2Thread.start();
    }
}
```


페이지 19

앞 슬라이드에서의 설명을 code로 옮긴 프로그램이 되겠습니다.
이 프로그램은 Thread Synchronization의 필요성을 보여주는 예제입니다.
여기서 두 개의 Thread인 User1Thread1과 User2Thread1가 하나의 Calculator1 객체를 공유하며 작업을 수행합니다.
Calculator1 class는 memory(memory) 값을 설정하고 출력하는 method를 가지고 있습니다.
main method에서 Calculator1 객체인 calculator를 생성합니다.
이 객체는 두 개의 Thread, 즉, user1Thread와 user2Thread가 공유하여 사용하게 됩니다.
user1Thread를 생성하고 setCalculator() method를 call하여 calculator 객체를 전달합니다.
그리고 user1Thread.start()를 call하여 실행시킵니다.
마찬가지로 user2Thread를 생성하고

setCalculator() method를 call하여
calculator 객체를 전달합니다.
그리고 user2Thread.start()를 call하여 실행시킵니다.

Example: Non-SynchronizedExample1 (2/4)

```
class User1Thread1 extends Thread {
    private Calculator1 calculator;

    public User1Thread1() {
        setName("User1Thread");
    }

    public void setCalculator(Calculator1 calculator) {
        this.calculator = calculator;
    }

    @Override
    public void run() {
        calculator.setMemory(100);
    }
}
```


페이지 20

User1Thread1 class는 Thread의 child로 정의됩니다.
생성자에서 setName을 이용하여
"User1Thread"로 이름을 바꿉니다.
private instance variable인 calculator의 값을
setting하기 위한
mutator setCalculator method를 가지고 있고
Overriding된 run에서는 calculator의 memory를
100으로 set하였습니다.

Example: Non-SynchronizedExample1 (3/4)

```
class User2Thread1 extends Thread {
    private Calculator1 calculator;

    public User2Thread1() {
        setName("User2Thread");
    }

    public void setCalculator(Calculator1 calculator) {
        this.calculator = calculator;
    }

    @Override
    public void run() {
        calculator.setMemory(50);
    }
}
```


페이지 21

User2Thread1 class도 앞의 User1Thread1 class와 거의 유사하게 정의됩니다.
다만 여기서는 Overriding된 run에서 calculator의 memory를 50으로 set하였습니다.

Example: Non-SynchronizedExample1 (4/4)

```
class Calculator1 {
    private int memory;

    public int getMemory() {
        return memory;
    }

    public void setMemory(int memory) {
        this.memory = memory;
        try {
            Thread.sleep(2000);
        } catch(InterruptedException e) {}
        System.out.println(Thread.currentThread().getName() + ": " +  this.memory);
    }
}
```

OUTPUT:
User2Thread: 50
User1Thread: 50

페이지 22

Calculator1 class는
정수인 memory를 instance variable로 가지고 있습니다.
setMemory(int memory) method에서는
우선 2초 정도 sleep 하여 기다리다가
현재 thread의 이름을 얻어 프린트하고
calculator의 memory 값을 프린트 합니다.
이 프로그램은 Thread Synchronization을 사용하지 않기 때문에
User1Thread1과 User2Thread1이 서로 간섭하게 됩니다.
예를 들어, User1Thread1이 memory를 100으로 설정하고 2초 대기하는 동안
User2Thread1이 memory를 50으로 변경할 수 있습니다.
그 결과 User1Thread1이 memory 값을 출력할 때 기대와 달리
100이 아닌 50이 출력될 수 있습니다.
마찬가지로 User2Thread1도 영향을 받게 됩니다.

이 문제를 해결하려면 setMemory() method에 synchronized 키워드를 추가하여
두 Thread가 동시에 setMemory() method에 접근하지 못하도록 해야 합니다.
synchronized를 사용하면 하나의 Thread가 setMemory() method를 실행하는 동안
다른 Thread는 대기 상태로 기다리게 되므로, memory 값이 safe하게 유지됩니다.
이 예제는 공유 resource에 대한 Thread Synchronization의 중요성을 강조합니다.

Synchronized Method and Block

- Method나 Block을 한번에 하나의 thread만 실행할 수 있게 lock을 건다
- 그 thread가 synchronized method나 block의 실행을 끝내면 lock을 푼다
- 다른 thread가 lock을 얻어 걸고 synchronized method나 block을 실행

```
public synchronized void method() {
    // 하나의 thread만 실행하는 영역
}


public void method() {
    // 여러 thread가 실행할 수 있는 영역
    synchronized(this) {
        // 하나의 thread만 실행할 수 있는 영역
    }
    // 여러 thread가 실행할 수 있는 영역
}
```


페이지 23

Synchronized Method와 Block에 대해 설명하겠습니다.
synchronized 키워드를 사용하면 Method나 Block을
한 번에 하나의 Thread만 실행할 수 있도록 lock을 걸 수
있습니다.
이 lock은 현재 Thread가 그 synchronized method나 block을
모두 실행할 때까지 유지됩니다.
실행이 끝나면 lock이 풀려서 다른 Thread가 이 영역에 들어올
수 있게 됩니다.
예를 들어, public synchronized void method()처럼
Method에 synchronized 키워드를 붙이면,
이 method는 한 번에 하나의 Thread만 실행할 수 있습니다.
즉, 이 method를 실행하는 동안에는 다른 Thread가
이 method에 접근하지 못하도록 잠겨 있는 것입니다.
다른 방식으로는, method 전체가 아닌 특정한 code 영역에만
lock을 걸 수도 있습니다.

이때는 synchronized(this)와 같이 사용하여
특정 block만 한 번에 하나의 Thread만 실행할 수 있게 할 수 있습니다.
예를 들어, public void method() 내부에서 특정 영역만 synchronized(this)로 감싸서,
그 부분만 한 번에 하나의 Thread만 실행하도록 제한할 수 있습니다.
그 외의 부분은 여러 Thread가 동시에 실행할 수 있습니다.
즉, synchronized를 method에 붙이면 전체 method가 잠기고,
synchronized block을 사용하면 method의 일부 code만 잠그는 것입니다.
이를 통해 필요한 부분만 lock 처리를 해서 성능을 최적화할 수 있습니다.

Example: SynchronizedExample2 (1/2)

```
class Calculator2 {
    private int memory;

    public int getMemory() {
        return memory;
    }

    public synchronized void setMemory1(int memory) { // synchronized method
        this.memory = memory;
        try {
            Thread.sleep(2000);
        } catch(InterruptedException e) {}
        System.out.println(Thread.currentThread().getName() + ": " +  this.memory);
    }
```


페이지 24

여기서는 Calculator2 class의
synchronization 기능에 대해 설명하겠습니다.
이 프로그램은 여러 thread가 동시에
`memory`라는 필드에 접근할 때 발생할 수 있는 문제를
해결하기 위해
자바의 synchronized method와 synchronized block을
사용하는 예제입니다.
이전에 synchronized 기능이 구현되지 않은
SynchronizedExample1 example과
다른 부분은 거의 비슷하기 때문에 설명을 생략하고,
synchronized 기능이 구현된 Calculator2 class부터
설명하겠습니다.
`Calculator2` class의 구조를 살펴보면,
`memory`라는 정수형 필드를 가지고 있습니다.
memory 값을 읽어오는 accessor로는

getMemory() method가 정의되어 있습니다.
memory 값을 바꾸는데 이용할 수 있는 두 가지 method가 있습니다:
synchronized method인 `setMemory1`과
synchronized block을 사용하는 `setMemory2`입니다.
두 method는 `memory` 필드에 접근할 때
한 번에 하나의 thread만 접근할 수 있도록 제한합니다.
먼저 `setMemory1` method를 보겠습니다.
이 method는 `synchronized` 키워드를 사용하여 synchronized되어 있습니다.
즉, 한 thread가 `setMemory1` method를 실행하는 동안,
다른 thread는 이 method에 접근할 수 없습니다.
이 method가 실행될 때 현재 실행 중인 thread는 `Calculator2` 객체에 lock을 걸어 다른 thread의 접근을 차단합니다.
method 내부에서 `Thread.sleep(2000);`을 call하여 2초간 대기한 후,
현재 thread 이름과 memory 값을 출력합니다.

Example: SynchronizedExample2 (2/2)

```
    public void setMemory2(int memory) {
        synchronized(this) { // synchronized block
            this.memory = memory;
            try {
                Thread.sleep(2000);
            } catch(InterruptedException e) {}
            System.out.println(Thread.currentThread().getName() + ": " +  this.memory);
        }
    }
}
```

```
OUTPUT:
User1Thread: 100
User2Thread: 50
```


페이지 25

다음으로 `setMemory2` method를 보겠습니다.
이 method는 method 전체가 아닌
특정 code block만 synchronized하는 방식으로 구현되어 있습니다.
`setMemory2` method는
`synchronized(this)` 구문을 통해 synchronized block을 설정하고,
이 block 안에서만 `memory` 필드에 접근합니다.
따라서 `Calculator2` 객체에 대해
한 번에 하나의 thread만 이 block을 실행할 수 있습니다.
이후 `Thread.sleep(2000);`으로 2초간 대기한 후,
thread 이름과 memory 값을 출력합니다.
이 프로그램의 출력은 User1Thread는 100,
User2Thread는 50이 되어 원래의 의도가 잘 반영되었습니다.
이 출력은 두 개의 thread가 각각

`setMemory1`과 `setMemory2` method를 call하여
각기 다른 값인 100과 50을 `memory`에 설정했음을 보여줍니다.
각 thread는 2초의 대기 시간을 거친 후,
최종적으로 설정한 memory 값을 출력합니다.
이처럼 `synchronized`를 사용하면
여러 thread가 `memory` 필드에 접근하더라도
동시에 값이 변경되는 것을 방지할 수 있습니다.

Thread Control Using wait() and notify()

- Waiting pool에 BLOCKED thread들이 synchronized method (block)을 wait 중
- 한 thread가 작업을 완료하면 notify() 또는 notifyAll() 을 call하여 다른 thread (또는 wait pool의 모든 thread들)을 Ready 상태로 만들고, 자신은 wait()하여 BLOCKED 상태로 만듦
- wait()와 notify()는 synchronized method (block) 내에서만 실행 가능
- 이를 통해 thread들이 번갈아서 synchronized method (block) 을 실행 가능

페이지 26

wait()와 notify() method를 사용하여
thread를 제어하는 방법에 대해 설명 하겠습니다.
먼저, Waiting pool은 BLOCKED 상태에 있는 thread들이
synchronized method 또는 synchronized block을 기다리는
곳입니다.
즉, 특정 작업을 위해 대기 중인 thread들이
이 Waiting pool에서 wait 상태로 대기하게 됩니다.

한 thread가 synchronized method 또는 synchronized
block에서 작업을 완료하면,

notify() 또는 notifyAll() method를 call하여 다른 thread들을
깨울 수 있습니다.

notify()는 Waiting pool에 있는 하나의 thread를 Ready 상태로
만들어

대기열에서 빠져나오게 하고,

notifyAll()은 Waiting pool에 있는 모든 thread를 Ready 상태로 만들어줍니다.

이 과정을 통해, 다른 thread들이 다시 실행될 수 있도록 준비되며,

현재 thread는 wait() method를 call하여 BLOCKED 상태로 들어가 대기할 수 있습니다.

다음은 wait()와 notify() method의 제약사항입니다.

이 두 method는 반드시 synchronized method 또는

synchronized block 내에서만 call될 수 있습니다.

그렇지 않으면 IllegalMonitorStateException이라는 예외가 발생합니다.

이는 synchronized block 내에서만

공유 resource에 대한 제어권을 얻을 수 있기 때문에 필수적인 제약입니다.

이 메커니즘을 통한 효과를 정리하면,

wait()와 notify()를 통해 여러 thread가 번갈아 가며

synchronized method 또는 synchronized block을 실행할 수 있게 됩니다.

즉, 한 thread가 resource을 다 사용한 후, 다른 thread가 resource을 사용할 수 있도록 제어할 수 있는 것입니다.

Example: WaitNotifyExample (1/4)

```
public class WaitNotifyExample {
    public static void main(String[] args) {
        WorkObject workObject = new WorkObject();

        ThreadA threadA = new ThreadA(workObject);
        ThreadB threadB = new ThreadB(workObject);

        threadA.start();
        threadB.start();
    }
}
```


페이지 27

여기에서는 `WaitNotifyExample` 프로그램을 통해
`wait()`와 `notify()` method를 사용한
thread 간 통신 방법을 설명드리겠습니다.
이 프로그램은 두 개의 thread가 번갈아 가며
`methodA`와 `methodB` 작업을 수행하는 구조로 되어
있습니다.
이를 통해 자바에서 멀티스레딩 환경에서의
thread 제어와 synchronization 방법을 배워보겠습니다.

먼저 `WaitNotifyExample` class의 `main` method를
살펴보겠습니다.
이 method는 `WorkObject`라는 공유 객체를 생성하고,
두 개의 thread `ThreadA`와 `ThreadB`를 생성합니다.
이 두 thread는 `WorkObject`의 `methodA`와 `methodB`를
번갈아 가며 call하도록 설계되어 있습니다.

마지막으로 `threadA.start()`와 `threadB.start()`를 통해
두 thread를 실행합니다.

Example: WaitNotifyExample (2/4)

```
class ThreadA extends Thread {
    private WorkObject workObject;

    public ThreadA(WorkObject workObject) {
        setName("ThreadA");
        this.workObject = workObject;
    }

    @Override
    public void run() {
        for(int i=0; i<10; i++) {
            workObject.methodA();
        }
    }
}
```


페이지 28

`ThreadA` class는 `Thread` class를 상속하여 만들어졌으며,
`WorkObject`를 instance 변수로 가집니다.
생성자에서는 thread 이름을 "ThreadA"로 설정하고,
`WorkObject` 객체를 받아서 저장합니다.
`run()` method에서는 반복문을 통해
`workObject.methodA()`를 10번 call합니다.
이 thread는 `methodA` method를 실행하며
`notify()`와 `wait()` method를 이용해 다른 thread와 교대로
실행됩니다.

Example: WaitNotifyExample (3/4)

```
class ThreadB extends Thread {
    private WorkObject workObject;

    public ThreadB(WorkObject workObject) {
        setName("ThreadB");
        this.workObject = workObject;
    }

    @Override
    public void run() {
        for(int i=0; i<10; i++) {
            workObject.methodB();
        }
    }
}
```


페이지 29

`ThreadB` class는 `ThreadA`와 유사하게 `Thread` class를 상속받았으며,
`WorkObject`를 instance 변수로 가집니다.
생성자에서는 thread 이름을 "ThreadB"로 설정하고,
`WorkObject` 객체를 받아서 저장합니다.
`run()` method에서는 반복문을 통해
`workObject.methodB()`를 열번 call합니다.
이 thread는 `methodB` method를 실행하며,
역시 `notify()`와 `wait()` method를 통해 `ThreadA`와 번갈아 가며 실행됩니다.

Example: WaitNotifyExample (4/4)

```
class WorkObject {
    public synchronized void methodA() {
        Thread thread = Thread.currentThread();
        System.out.println(thread.getName() + ": methodA 작업 실행");
        notify(); // ThreadB를 Ready로
        try {
            wait(); // 자신(ThreadA)을 BLOCKED state로
        } catch (InterruptedException e) { }
    }
    public synchronized void methodB() {
        Thread thread = Thread.currentThread();
        System.out.println(thread.getName() + ": methodB 작업 실행");
        notify(); // ThreadA를 Ready로
        try {
            wait(); // 자신(ThreadB)를 BLOCKED state로
        } catch (InterruptedException e) { }
    }
}
```

OUTPUT:
ThreadA: methodA 작업 실행
ThreadB: methodB 작업 실행
ThreadA: methodA 작업 실행
ThreadB: methodB 작업 실행
ThreadA: methodA 작업 실행
...

페이지 30

`WorkObject` class는 `methodA`와 `methodB` 두 개의 method를 가지고 있으며,
이 두 method는 모두 `synchronized`로 선언되었습니다.
이 class는 `ThreadA`와 `ThreadB` 간의 공유 객체로서,
두 thread가 교대로 이 method를 call합니다.
각 method는 현재 실행 중인 thread의 이름을 출력하고,
`notify()`로 상대 thread를 깨운 뒤,
`wait()`로 자신을 `BLOCKED` 상태로 만듭니다.
이를 통해 두 thread는 서로 번갈아 가며 작업을 수행하게 됩니다.
`methodA`와 `methodB` method 내부에서
`wait()`와 `notify()` method를 사용하는 동작을 설명하겠습니다.
`methodA`는 `notify()`를 call해 `ThreadB`를 깨우고,
`ThreadA`는 `wait()`를 call해 대기 상태로 전환됩니다.

`methodB`는 `notify()`를 call해 `ThreadA`를 깨우고,
`ThreadB`는 `wait()`를 call해 대기 상태로 전환됩니다.
이렇게 `wait()`와 `notify()`를 통해 두 thread는 번갈아 가며
`methodA`와 `methodB`를 교대로 실행하게 됩니다.
이 방식은 공유 resource을 synchronize하고,
교대로 사용해야 하는 멀티thread 환경에서 유용합니다.
프로그램 출력에서는 `ThreadA`와 `ThreadB`가 번갈아 가며
`methodA`와 `methodB`를 실행합니다.
`wait()`와 `notify()` method를 통해
두 thread가 서로 순차적으로 실행되도록 제어했기 때문에
이와 같은 결과가 나옵니다.

Safe Termination of Thread

- Thread가 예상치 않은 상태나 부작용 없이, 모든 resource를 적절히 정리하고 작업을 마무리하는 것
  - Resource Release (resource 해제) – 파일, 네트워크 소켓 등 thread 사용resource 해제
  - State Cleanup (상태 정리) – thread 작업을 적절히 마무리
  - Consistent Behavior (일관된 동작) – 예측 가능한 종료 과정

페이지 31

여기에서는 thread의 safe한 종료
(즉, Safe Termination of Thread)에 대해 설명 드리겠습니다.

thread의 safe한 종료란,

thread가 예기치 않은 상태나 부작용 없이 모든 resource를 적절히 정리하고

작업을 마무리하는 것을 의미합니다.

safe한 종료는 멀티thread 환경에서 중요한 개념으로,

이를 통해 시스템의 안정성과 예측 가능성을 높일 수 있습니다.

safe termination시 행해지는 작업들을 살펴보겠습니다.

첫번째로 Resource Release 입니다.

thread가 종료되기 전에 모든 resource을 해제하는 것이 중요합니다.

여기에는 파일, 네트워크 소켓 등 thread가 사용한 resource이 포함됩니다.

만약 resource을 제대로 해제하지 않으면, 리소스가 낭비되거나

시스템에 부정적인 영향을 줄 수 있습니다.

두 번째는 State Cleanup입니다.

State Cleanup은 thread가 작업을 종료하기 전에

그 상태를 정리하는 것을 의미합니다.

thread가 도중에 중단될 수 있는 상황에서 그 상태를 정리해 주어야,

다른 thread나 process가 혼란 없이 이어서 작업을 수행할 수 있습니다.

마지막으로 Consistent Behavior입니다.

thread가 일관된 동작을 유지하면서

예측 가능한 종료 과정을 거치는 것이 중요합니다.

이 과정을 통해 시스템의 안정성을 보장하고,

사용자가 thread의 종료를 예상할 수 있도록 해줍니다.

결론적으로, thread의 safe한 종료는 리소스를 효율적으로 관리하고

시스템의 예측 가능성을 높이는 중요한 요소입니다.

멀티thread 환경에서는 특히 이러한 정리가 필수적입니다.

Safe Termination Method

1) Volatile variable을 사용한 flag 방식
   - volatile variable: value가 변경될 때마다 모든 thread에서 즉시 반영 보장
2) Interrupt() method
   - thread의 WAIT, BLOCKED state 해제, 작업 중단 가능
3) ExecutorService 사용
   - thread pool 관리
   - work submission (thread pool의 thread에게 작업 의뢰)
   - termination control
   - Using shutdown() and shutdownNow()

페이지 32

thread의 safe한 종료를 위한 방법들에 대해 설명하겠습니다.

첫 번째 방법은 Volatile variable을 사용하는 것입니다.

volatile variable은 thread 간에 공유되는 변수로,

값이 변경될 때마다 모든 thread에서 즉시 반영되는 것을 보장합니다.

이를 통해 flag(flag) 변수를 사용하여

thread 종료 조건을 설정할 수 있습니다.

예를 들어, 특정 조건에서 volatile flag 값을 false로 설정하여

모든 thread가 종료되도록 제어할 수 있습니다.

두 번째 방법은 Interrupt() method를 사용하는 것입니다.

interrupt() method는 특정 thread가 WAIT 상태나 BLOCKED

상태에 있을 때

해당 상태를 해제하고, 작업을 중단할 수 있도록 합니다.

예를 들어, thread가 대기 중일 때 interrupt()를 call하면

InterruptedException이 발생하고, thread가 작업을 중단할 수 있게 됩니다.

세 번째 방법은 ExecutorService를 사용하는 것입니다.

ExecutorService는 thread 풀(thread pool)을 관리하는 유용한 도구입니다.

이를 통해 여러 작업을 thread에 효과적으로 분배하고,

종료 시점까지 제어할 수 있습니다.

작업을 thread 풀에 제출하고,

shutdown()이나 shutdownNow() method를 통해 safe하게 thread를 종료할 수 있습니다.

shutdown()은 이미 제출된 작업이 모두 완료될 때까지 기다린 후 종료하고,

shutdownNow()는 현재 진행 중인 작업을 즉시 중단하도록 요청합니다.

이와 같은 세 가지 방법을 사용하여 thread를 safe하게 종료할 수 있습니다.

각 방법은 상황에 따라 유용하게 활용될 수 있으며,

이를 통해 멀티thread 환경에서 안정성과 예측 가능성을 높일 수 있습니다.

Example: SafeStopVolatile

```
class VolatileThread extends Thread {
    private volatile boolean running = true;
    public void run() {
        while (running) {
            // 작업 수행
        }
    }
    public void stopThread() {
        running = false;
    }
}
public class SafeStopVolatile {
    public static void main(String[] args) throws InterruptedException {
        VolatileThread thread = new VolatileThread();
        thread.start();
        Thread.sleep(1000); // 1초 후에 종료
        thread.stopThread();
    }
}
```


페이지 33

이 슬라이드는 volatile 변수를 사용하여 thread를 safe하게 종료하는 예제를 보여줍니다.
이 예제에서는 SafeStopVolatile class와 VolatileThread class를 사용하여
volatile 키워드가 어떻게 thread의 safe한 종료를 보장하는지 설명합니다.

먼저 VolatileThread class입니다.

이 class는 Thread class를 상속하여 새 thread를 정의합니다.

running이라는 volatile 변수를 선언하고 초기값을 true로 설정합니다.

volatile 키워드를 사용하면 이 변수의 값이 변경될 때

모든 thread에 즉시 반영되므로,

값이 변경될 때마다 thread가 변수의 최신 상태를 확인할 수 있게 됩니다.

run() method에서는 running이 true일 동안 반복해서 작업을 수행합니다.

while (running) 루프는 running이 false로 변경되면 자동으로 종료됩니다.

이 반복문 안에서 실제 작업이 수행되며, 작업 수행 부분은 주석으로 표시되어 있습니다.

stopThread() method는 running을 false로 설정하여

run() method의 while 루프가 종료되도록 합니다.

이 method를 call함으로써 thread가 safe하게 종료될 수 있습니다.

마지막으로 SafeStopVolatile class입니다.

main() method에서 VolatileThread instance를 생성하고

start() method를 call하여 thread를 시작합니다.

thread가 실행된 후 Thread.sleep(1000);을 통해 1초 동안 대기하고,

그 후 stopThread()를 call하여 thread를 중지합니다.

이로 인해 running 변수가 false로 설정되고,

VolatileThread의 run() method는 루프를 빠져나와 종료됩니다.

이 예제를 통해 volatile 변수를 사용하여

thread를 safe하게 종료하는 방법을 확인할 수 있습니다.

volatile 키워드를 사용하면

thread가 변수의 최신 상태를 항상 확인할 수 있어,

running flag를 통해 thread를 예측 가능하게 종료할 수 있습니다.

Example: SafeStopInterrupt

```java
class InterruptibleThread extends Thread {
    public void run() {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                Thread.sleep(100); // 작업 수행 중 차단 상태일 수 있음
            }
        } catch (InterruptedException e) { // interrupt가 발생했을 때 처리
            System.out.println("Thread interrupted");
        }
    }
}
public class SafeStopInterrupt {
    public static void main(String[] args) throws InterruptedException {
        InterruptibleThread thread = new InterruptibleThread();
        thread.start();
        Thread.sleep(1000); // 1초 후에 인터럽트
        thread.interrupt();
    }
}
```

OUTPUT:
Thread interrupted

페이지 34

여기에서는 interrupt() method를 사용하여
thread를 safe하게 종료하는 예제를 설명하겠습니다.

먼저 InterruptibleThread class는

Thread class를 상속하여 정의된 thread class입니다.

run() method에서 while 루프를 사용하여 반복 작업을
수행합니다.

이 루프는 현재 thread가 interrupted 상태가 아닐 때까지
계속됩니다.

Thread.sleep(100);을 call하여

100밀리초 동안 thread를 잠시 멈추게 합니다.

이렇게 짧은 대기 시간을 설정해 두면 interrupt()가 발생했을
때 즉시 중단될 수 있습니다.

run() method는 try-catch block으로 감싸져 있습니다.

sleep() method가 실행 중일 때 interrupt()가 call되면

InterruptedException이 발생합니다.

이 예외가 발생하면 catch block에서

"Thread interrupted"라는 메시지를 출력하며

thread를 safe하게 종료할 수 있습니다.

SafeStopInterrupt class의 main() method에서

InterruptibleThread instance를 생성하고

start() method를 call하여 thread를 시작합니다.

thread가 시작된 후, Thread.sleep(1000);을 통해 1초 동안 대기한 후

interrupt() method를 call하여 thread에 인터럽트를 발생시킵니다.

이로 인해 InterruptibleThread의 run() method에서

InterruptedException이 발생하고,

thread는 "Thread interrupted" 메시지를 출력하며 종료됩니다.

이 code의 실행 결과는 OUTPUT: Thread interrupted입니다.

이 메시지는 interrupt() call로 인해 thread가 정상적으로
중단되었음을 나타냅니다.

이 예제를 통해 interrupt() method를 사용하여

thread를 safe하게 중단하는 방법을 이해할 수 있습니다.

interrupt() method는 thread가 sleep()이나 wait() 상태일 때 중단을
요청할 수 있어,

멀티thread 환경에서 유용하게 활용됩니다.

Example: SafeStopExecutorService

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class SafeStopExecutorService {
    public static void main(String[] args) throws InterruptedException {
        // thread pool 생성 (2개 thread)
        ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.submit(() -> { // pool의 1st thread에게 작업 의뢰
            while (!Thread.currentThread().isInterrupted()) {
                System.out.println(Thread.currentThread().getName() + " running");
            }
        });
        executor.submit(() -> { // pool의 2nd thread에게 작업 의뢰
            while (!Thread.currentThread().isInterrupted()) {
                System.out.println(Thread.currentThread().getName() + " running");
            }
        });
        Thread.sleep(10); // 10 miliseconds (0.001초) 후에 종료
        executor.shutdown(); // ExecutorService 종료
        if (!executor.awaitTermination(10, TimeUnit.MILLISECONDS)) {
            executor.shutdownNow(); // 강제 종료 (interrupt 발생)
        }
    }
}
```

OUTPUT:
pool-1-thread-1 running
pool-1-thread-1 running
pool-1-thread-2 running
pool-1-thread-1 running
pool-1-thread-2 running
...

페이지 35

여기에서는 ExecutorService를 사용하여
thread를 safe하게 종료하는 방법을 설명하겠습니다.

이 예제에서는 ExecutorService, Executors, TimeUnit class를 사용합니다.

ExecutorService는 thread 풀을 관리하며,

이를 통해 thread의 실행과 종료를 효율적으로 제어할 수 있습니다.

Executors.newFixedThreadPool(2)를 call하여

2개의 thread로 구성된 thread 풀을 생성합니다.

이 thread 풀은 두 개의 작업을 동시에 처리할 수 있습니다.

첫 번째 executor.submit()에서 첫 번째 thread가

무한 루프에서 isInterrupted() method를 통해 중단 상태를 확인하며

"running" 메시지를 출력하도록 합니다.

두 번째 executor.submit()도 동일하게 무한 루프에서

중단 상태를 확인하며 "running" 메시지를 출력합니다.

이 두 작업은 thread가 중단되지 않는 한 계속 실행됩니다.

Thread.sleep(10);을 통해 메인 thread가 10밀리초 대기한 후,

executor.shutdown()을 call하여 ExecutorService의 종료를 요청합니다.

shutdown()은 이미 제출된 작업을 완료하지만 새로운 작업은 받지 않게 합니다.

이후, awaitTermination(10, TimeUnit.MILLISECONDS) method를 사용하여

최대 10밀리초 동안 thread의 종료를 기다립니다.

만약 지정된 시간 내에 모든 작업이 종료되지 않으면,

shutdownNow()를 call하여 강제 종료(interrupt)를 발생시킵니다.

출력 결과는 각 thread가 "running" 메시지를 지속적으로 출력하다가,

shutdownNow() call 이후에 강제로 중단되는 모습을 보여줍니다.

출력은 pool-1-thread-1 running,

pool-1-thread-2 running과 같은 메시지가 출력됩니다.

이 예제를 통해 ExecutorService를 사용하여

thread 풀 내의 thread를 safe하게 종료하는 방법을 이해할 수 있습니다.

shutdown()과 shutdownNow()를 통해

thread를 순차적으로 종료하거나 강제적으로 중단할 수 있어,

멀티thread 환경에서 resource 관리를 효율적으로 수행할 수 있습니다.