**자바프로그래밍2 실습과제**

**제12주 동시실행프로그램**

프로젝트 아래에 w12 패키지를 만들고 그 아래에서 작업하세요.

1. [TwoThreads.java](src\TwoThreads.java) 프로그램을 읽어보고 실행해 보세요.

이 프로그램의 경우 main 메소드가 끝나면서 “main 메소드 끝.”을 출력합니다. 콘솔창 첫 줄에 이 문장이 출력됩니다. 그런데 main 메소드가 끝나도 애플리케이션(프로그램)이 종료되지 않습니다. main에서 start시킨 두 개의 스레드가 아직 돌고 있기 때문입니다. 스레드들이 모두 끝나야 비로소 애플리케이션이 끝납니다. 애플리케이션을 강제로 끝내려면 이클립스 콘솔 창 오른쪽 위에 있는 빨간 네모 버튼을 누릅니다.

이 프로그램은 Runnable을 구현한 클래스를 선언하고 그 객체를 task로 삼아 쓰레드를 구동했습니다. 클래스를 만들지 않고 람다식을 task로 삼아 쓰레드를 구동하도록 프로그램을 수정하세요. 새 프로그램의 이름은 TwoThreadsLambda로 지으세요. 프로그램 실행 결과는 같아야 합니다.

1. 그래픽프로그램들은 대부분 멀티쓰레딩 방식으로 작동합니다. [CircleComponent.java](src\CircleComponent.java)를 실행해 보고 프로그램을 살펴보세요(main 메소드는 여러 가지 메소드 호출을 하고 마지막에 “main 끝.”을 출력하면서 끝납니다.). 이 프로그램의 경우에 main은 금방 끝납니다. 그렇지만 화면에 그림을 그리는 쓰레드가 계속 살아 있습니다. (이 프로그램에는 그림 그리는 별도의 쓰레드를 만들어 start시키는 코드가 멍시적으로 나타나지는 않지만 무대 뒤에서 별도의 쓰레드가 만들어집니다.)

그림이 그려진 창의 오른쪽 위에 x 버튼을 클릭하면 그림을 그려주는 쓰레드가 끝나고 애플리케이션이 종료됩니다. 콘솔 창의 빨간 버튼을 누르면 강제로 애플리케이션이 종료됩니다.

1. 스레드에 인터럽트 걸기
   1. 인터럽트 기초

 Thread.sleep 메소드는 InterruptedException을 던지는 수가 있습니다. 위 1번의 TwoTreads 프로그램에서는 InterruptedException을 처리하는 예외처리기가 간단한 출력만 했습니다. 스레드가 잠들어 있는 동안 스레드에 인터럽트(끼어들다. 중단시키다, 말을 끊다 등의 의미)가 걸리면 쓰레드가 잠자던 문장을 빠져나와 이벤트핸들러, 즉 캐치 블록으로 이동합니다. 이벤트핸들러에서 간단한 출력을 하고 그 다음 문장으로 넘어가는데 TwoThreads 프로그램의 경우 run 메소드가 끝나게 됩니다. 그러면 run 메소드를 실행하던 스레드도 끝납니다.

[TwoThreads2.java](src\TwoThreads2.java) 프로그램을 읽고 실행해 보세요. Timing 태스크는 100번 출력을 하는 놈인데 중간에 인터럽트를 걸어서 중단시켰습니다. Timing은 위 1번의 TwoThreads.java 파일에 package access로 선언(class 선언 앞에 public이나 private이 붙지 않음)되어 있는 클래스입니다. 같은 디렉토리(패키지) 안에 이미 컴파일되어 있으므로 패키지 내의 다른 클래스에서 자유롭게 사용할 수 있습니다.

* 1. 사용자 입력에 의한 인터럽트

[TwoThreads3.java](src\TwoThread3.java) 프로그램을 읽고 실행해 보세요.

1. 경쟁 조건 (Race Condition)

 여러 스레드가 데이터를 공유하면서 공유데이터를 수정할 때 그 데이터가 훼손되는 현상을 실습합니다.

강의자료 12~17쪽의 내용을 먼저 잘 읽어 보세요.

* 1. 두 개의 스레드, 두 개의 Runnable

[NumberPrinter](src\NumberPrinter.java)와 [NumberPrinterRunner1](src\NumberPrinterRunner1.java) 클래스를 이클립스에 넣고 코드를 읽어 보고 어떤 출력이 나올지 예측해 보세요. 또, 프로그램을 실행하여 예측이 맞는지 확인하세요. 이 프로그램은 두 개의 스레드가 각각의 Runnable 객체를 실행하는 예입니다. 데이터가 두 스레드 사이에 공유되지 않습니다.

* 1. 두 개의 스레드, 한 개의 Runnable

 NumberPrinterRunner1을 복사하여 NumberPrinterRunner2로 저장하고 NumberPrinterRunner2를 수정하여 두 스레드가 하나의 Runnable을 실행하게 하세요. 이 경우 두 스레드가 number라는 이름의 데이터를 공유하면서 각각 number의 값을 갱신합니다. number의 값이 올바르지 않은 값으로 훼손되는지 프로그램을 출력을 관찰하세요.

* 1. 두 개의 스레드, 한 개의 Runnable, 지역변수 출력

 NumberPrinter를 복사하여 IPrinter로 저장하고 number를 출력하는 대신 반복문의 인덱스인 i를 출력하도록 수정하세요. 또, NumberPrinterRunner2를 복사하여 NumberPrinterRunner3로 저장하고 NumberPrinterRunner3를 수정하여 두 스레드가 하나의 IPrinter를 실행하게 하세요. 이 경우 출력이 어떻게 될지 예측하고 프로그램을 실행하여 확인하세요.

1. 경쟁 조건 (Race Condition) 제어 (Control)

 경쟁 조건에서 공유 데이터가 훼손되지 않도록 제어하는 방법을 공부합니다.

* 1. 경쟁 조건 (Race Condition) - BankAccount

 하나의 BankAccount에 여러 스레드가 입금하고 여러 스레드가 출금하는 상황을 만들어 봅니다. w12.bankaccount 패키지를 만들고 그 아래에 아래 프로그램들을 넣은 후 코드를 잘 읽어 보세요.

[bankaccount\BankAccount.java](src\bankaccount\BankAccount.java)

[bankaccount\DepositRunnable.java](src\bankaccount\DepositRunnable.java)

[bankaccount\WithdrawRunnable.java](src\bankaccount\WithdrawRunnable.java)

[bankaccount\BankAccountThreadRunner.java](src\bankaccount\BankAccountThreadRunner.java)

이 프로그램에서는 10개의 입금 스레드 각각이 한 개의 BankAccount에 100원씩 10번 입금을 하고, 두 개의 출금 스레드 각각이 같은 BankAccount에서 100원씩 50번 출금을 합니다. 결국 입금 금액 전체는 10,000원이고 출금 금액 전체도 10,000원이므로 일을 마치면 잔액이 0원이어야 합니다. 12개 스레드가 입금과 출금을 하는 일은 동시에 진행됩니다.

 프로그램을 여러 번 실행해 보면 그 때마다 최종 잔액이 달라지며 그 값은 0이 아닌 경우가 많다는 것을 관찰할 수 있습니다. 경쟁 조건이 만들어져서 공유 데이터가 훼손되기 때문입니다. 그런데 잘 보면 최종 잔액이 음수인 경우가 양수인 경우보다 많이 관찰됩니다. 그 이유가 무엇일까요?

모두 12개의 스레드가 실행됩니다. 데이터는 balance 한 개입니다. 몹시도 무질서하고 정신없는 상황이군요. (경쟁 조건!)

12개 중 10개는 입금스레드이고 나머지 2개는 출금스레드입니다.

각 입금스레드는 1ms 간격으로 한 번에 100원씩 10번 입금합니다.

각 출금스레드는 1ms 간격으로 한 번에 100원씩 50번 출금합니다.

입금 총액 = 10 스레드 \* 100원 \* 10번 = 10,000원.

출금 총액 = 2 스레드 \* 100원 \* 50번 = 10,000원.

입출금 총액 = 10,000 - 10,000 = 0원

10개의 입금 스레드는 10ms 동안 지속됩니다. 2개의 출금 스레드는 50ms 동안 지속됩니다.

처음 10ms 동안에는 12개의 쓰레드들 간에 경쟁조건이 심하게 형성되어 입출금이 제대로 되지 않는 수가 많습니다. 그 후 40ms 동안에는 두 개의 쓰레드만 돌게 되므로 경쟁조건이 덜합니다. 이 시간 동안 출금은 비교적 원활하게 이루어지기 때문에 최종적으로 음수 잔액이 되는 수가 많습니다.

* 1. 경쟁 조건 (Race Condition) 제어 – 동기화 (Synchronizing)

 경쟁 조건을 제어하는 한 가지 방법은 공유 데이터에 접근하는 코드가 실행되는 동안 스레드 스위칭이 일어나지 않게 하는 것입니다. 이렇게 하는 데 사용하는 방법이 바로 메소드나 코드 블록을 synchronized로 선언하는 것입니다. Synchronized 메소드나 코드 블록이 하나의 객체에게 호출될 때는 그 실행 중간에 스레드 스위칭이 되지 않습니다.

<“하나의 객체에게 호출될 때”라는 말의 의미>

BankAccount의 withdraw 메소드가 synchronized 메소드로 선언되어 있고 아래와 같이 BankAccount 인스턴스 하나를 만들었다고 합시다.

BankAccount b1 = BankAccount();

두 개의 쓰레드들이 b1.withdraw()를 동시에 호출하면, 두 쓰레드 중 어느 하나만 withdraw를 실행하게 되고 나머지 쓰레드는 Blocked 상태가 됩니다. Withdraw 메소드 실행이 끝나야 다른 쓰레드가 Blocked 상태에서 다시 Runnable 상태로 오면서 withdraw를 실행하게 됩니다.

그러나, 아래와 같이 두 개의 BankAccount 인스턴스를 만들고, 한 쓰레드는 b1.withdraw를 다른 쓰레드는 b2.withdraw를 호출하는 경우에는 두 쓰레드가 서로 간섭 없이 동시에 withdraw를 실행합니다.

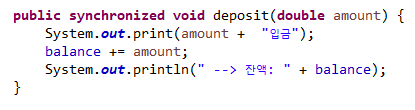
BankAccount b1 = BankAccount();

BankAccount b2 = BankAccount();

w12.bankaccount 패키지를 w12.bankaccount\_synch라는 이름의 패키지로 복사하고 새 패키지에서 작업하세요. BankAccount 클래스의 이름을 SynchronizedBankAccount로 바꾸세요.

이 클래스에는 공유 데이터인 balance 값을 갱신하는 메소드가 두 개 있습니다. deposit과 widthdraw입니다. getBalance도 balance에 접근하지만 읽기만 할 뿐 쓰기는 하지 않으므로 balance 데이터를 훼손하지는 않습니다. (1) deposit과 widthdraw 메소드를 synchronized로 선언하거나, (2) 그 안의 balance를 갱신하는 코드를 synchronized block으로 선언하세요.

 (1)   의 예는 아래 그림과 같습니다.



 (2)   의 예는 아래 그림과 같습니다. 메소드 전체를 synchronized로 선언하지 않고 꼭 필요한 코드만 선택하여 synchronized block으로 선언합니다.



 (2)의 방법에서 synchronized(this)라고 적었는데 this를 생략하면 안 됩니다. BankAccount 객체가 여러 개 있을 때 그 중 어느 것에 대한 synchronization인지를 지정해 주는 것입니다. 현재 코드를 실행하고 있는 객체에 대해 synch를 해 주라는 의미입니다. BankAccount가 두 개 있는 경우, 그 중 하나에 입금을 하고 있는 중에 다른 BankAccount에서 출금을 하지 못하게 할 필요는 없기 때문입니다. (1)의 방법에서는 this를 적어주지 않습니다. 인스턴스 메소드 자체가 특정 객체를 대상으로 하여 실행되는 것이기 때문입니다.

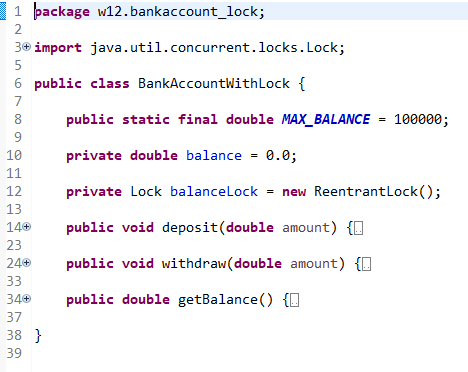
BankAccount를 SynchronizedBankAccount로 바꾼 후 프로그램을 실행해보세요. 경쟁 조건이 통제되어 프로그램 실행이 끝나면 언제나 잔액이 정확히 0이 됨을 확인하세요.

* 1. 경쟁 조건 (Race Condition) 제어 – 자물쇠 (Lock)

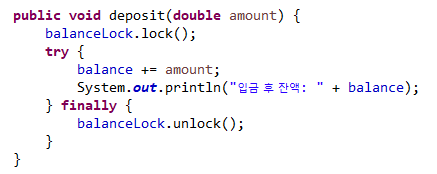
 경쟁 조건을 제어하는 두 번째 방법은 공유 데이터를 갖고 있는 객체에 자물쇠를 설치하는 것입니다. 스레드가 객체의 자물쇠를 점유하고자 할 때 (lock() 호출), 다른 쓰레드가 자물쇠를 이미 점유하고 있다면, 자물쇠를 점유하지 못하고 차단된 (Blocked) 상태로 가게 됩니다. (차단된 쓰레드에게는 쓰레드 스위칭에 의한 코드 실행 기회가 주어지지 않습니다.) 자물쇠를 점유하고 있던 쓰레드가 자물쇠를 놓으면 (unlock() 호출) 그 때 비로서 살아나 (Runnable 상태) 다시 자물쇠 점유를 시도하게 됩니다. 쓰레드가 자물쇠를 점유하고 나면 객체의 공유 데이터에 접근하는 코드를 실행할 수 있습니다. 코드 실행을 마칠 때는 다시 자물쇠를 열어 놓고 (unlock() 호출) 나옵니다. ([쓰레드가 가질 수 있는 상태](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.State.html) 참고)

w12.bankaccount 패키지를 w12.bankaccount\_lock라는 이름의 패키지로 복사하고 새 패키지에서 작업하세요. BankAccount 클래스의 이름을 BankAccountWithLock로 바꾸고 아래와 같이 수정하세요.

l  아래 12번 줄과 같이 Lock을 인스턴스 필드로 추가한다. Lock은 인터페이스이고 ReentrantLock은 그 인터페이스를 구현한 클래스입니다.



 l  Deposit 메소드를 실행할 때면 Lock을 점유하도록 합니다. 만약, Lock이 이미 다른 스레드에 의해 점유되어 있다면, 이 스레드는 Lock을 점유하지 못하고 기다리게 됩니다(Blocked state). Lock을 점유하고 있는 스레드가 자기 일을 마치고 Lock을 놓으면 그 때 비로소 멈추어 있던 스레드가 Lock을 점유하고 코드를 실행하게 됩니다. 코드 실행을 마치면 Lock을 풀고 나와야 합니다. 이렇게 해 주는 코드는 아래와 같습니다.



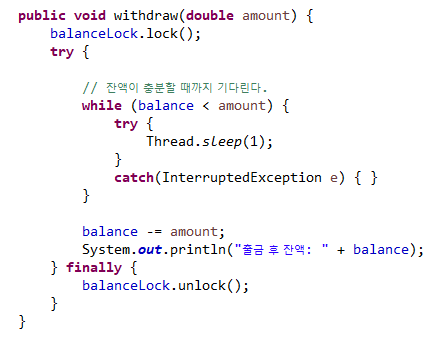
 l  withdraw 메소드도 위 그림처럼 바꾸어 줍니다.

프로그램을 실행하여 경쟁 조건이 통제되어 데이터가 훼손되지 않음을 확인하세요.

* 1. 교착 상태 (Deadlock)

 위에서는 BankAccount의 balance가 음수가 될 수 있게 허용했습니다. 이 문제에서는 balance가 음수가 될 수 없도록 제한합니다. 사용자가 돈을 찾으려고 하는데 찾고자 하는 금액보다 잔액이 적다면 찾을 수 없습니다. 알바 가게 사장님이나 부모님께서 돈을 입금해 주셔야 필요한 만큼의 돈을 찾을 수 있게 됩니다. 돈이 언제 입금될지 확실히 알 수 없으니 일정 시간 간격으로 ATM 기계에 가서 출금을 시도해 봅니다.

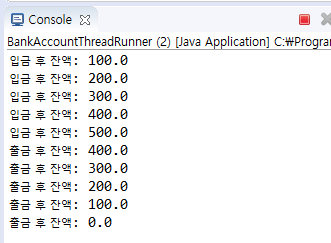
w12.bankaccount\_lock 패키지를 w12.bankaccount\_waiting이라는 이름의 패키지로 복사하고 새 패키지에서 작업하세요. BankAccountWithLock 클래스의 이름을 BankAccountWaiting으로 바꾸고 withdraw 메소드를 아래와 같이 수정하세요. deposit 메소드는 바꾸지 않아도 됩니다.



 아래 그림과 같이 입금 스레드 갯수가 출금 스레드 갯수에 비해 훨씬 많으면, 출금할 때 돈이 모자라는 경우가 잘 발생하지 않으므로, 입금 스레드 갯수와 출금 스레드 갯수를 같은 수(가령 5)로 설정합니다.



 프로그램을 실행하면 프로그램이 조금 진행하다가 중지되는 것을 볼 수 있습니다. 교착 상태에 빠진 것입니다. 출금 스레드가 withdraw 메소드에 진입하하면서 BankAccount 객체의 자물쇠를 차지했는데, 막상 돈을 찾으려고 보니 잔액이 부족하게 되자 잔액이 충분해질 때까지 1초간 잠들었다가 깨어나서 돈의 들어왔나 확인하고 다시 잠들기를 반복하게 됩니다. 자물쇠는 채워져 있는 상태입니다. 입금 스레드가 입금을 하려고 해도 자물쇠가 채워져 있어 deposit 메소드에 진입할 수가 없습니다. 출금 스레드는 자물쇠를 차지한 채로 돈이 들어 오기만을 기다리고, 입금 스레드는 자물쇠가 풀리기 만을 기다리는 상태로서 교착 상태(deadlock)입니다. 한 번 이 상태로 빠지면 아무리 시간이 가도 해소되지 않습니다. 아래 그림의 빨간 네모는 프로그램이 끝나지 않았음을 나타냅니다. 그렇지만 아무런 출력도 더 이상 나오지 않습니다. 빨간 네모를 클릭하여 강제로 프로그램을 끝내는 수밖에 없습니다.



* 1. 교착 (Deadlock) 방지

 프로그램이 교착 상태에 빠지지 않게 하기 위해 아래 그림과 같이 자물쇠를 채우는 시점을 늦추는 방법을 생각해 볼 수 있을 것입니다. 그러나 이렇게 하면 안 됩니다.



출금 스레드가 (28번 줄에서) 잠들기를 반복하면서 기다리다가 잔액이 충분해졌다고 판단하면, 35번 줄로 가서 자물쇠를 채우려고 할 것입니다. 그런데 28번 줄에서 35번 줄로 넘어가는 사이에 스레드 스위칭이 일어나 다른 출금 스레드가 돈을 먼저 빼 갈 수가 있습니다. 그러면 36번 줄에서 balance가 음수가 됩니다. 이것은 허용되지 않는 것으로 전제했기 때문에 에러입니다. 경쟁 조건을 완벽하게 제어하여 공유 데이터가 훼손되지 않게 해야 하는데 그렇게 못하는 것이지요. 즉, 경쟁 조건을 제대로 통제하지 못하게 되는 것입니다. 그러니까 자물쇠를 채우는 문장을 위 그림처럼 35번줄로 이동하면 안 됩니다.

프로그램이 교착 상태에 빠지지 않게 하려면 출금 스레드가 입금을 기다리는 동안, 입금 스레드가 deposit 메소드에 진입할 수 있게 해야 합니다. 이럴 때 사용하는 도구가 바로 Condition입니다. Condition은 Lock과 연결되어 있습니다. 하나의 Lock에는 하나의 Condition이 있을 수 있습니다. Condition은 아래 그림 1처럼 사용합니다.

Text

Description automatically generated with medium confidence

그림 1 Condition 사용 예

Condition 객체는 Lock으로부터 얻습니다(14번 줄). 잔액이 모자라 출금 스레드가 입금을 기다려야 할 때는 Thread.sleep(1)을 호출하는 대신 condition.await()를 호출합니다. Thread.sleep(1)은 1ms 동안 잠들었다가 스스로 깨어나는 반면, condition.await()를 호출하면 스레드가 ‘대기 (Waiting) 상태’로 가게 되며, 다른 스레드가 도와주지 않으면 영원히 복귀하지 못합니다. 또, condition.await()를 호출하면 스레드가 이미 점유하고 있는 Lock을 놓게 됩니다. 그러면 입금 스레드가 통장에 돈을 넣을 수 있게 됩니다. 입금 스레드는 돈을 입금한 후에 condition.signalAll( ) 메소드를 호출합니다. 그러면 이 condition으로 ‘대기 (Waiting)” 상태에 있는 모든 스레드가 대기 상태에서 빠져나오면서 자물쇠가 풀리기를 기다립니다. 입금 스레드가 자물쇠를 놓으먼 대기 상태에서 빠져나온 출금 스레드들 중 하나가 Lock을 갖고 출금 작업을 하고 Lock을 놓습니다. 만약, 대기 상태에서 빠져나온 출금 스레드가 더 있다면 그 놈에게 Lock이 돌아갑니다.

 w12.bankaccount\_waiting 패키지를 w12.bankaccount\_condition이라는 이름의 패키지로 복사하고 새 패키지에서 작업하세요. BankAccountWaiting 클래스의 이름을 BankAccountWithCondition으로 바꾸고 위 그림과 같이 condition을 사용하도록 코드를 수정하고 프로그램을 실행해보세요. 여러 개의 출금 스레드와 여러 개의 입금 스레드가 동시에 입출금을 반복해도 교착 상태에 빠지지 않고 프로그램이 끝난 후 balance가 정확히 0이 됨을 확인하세요. 프로그램을 여러 번 실행해도 단 한 번도 문제가 발생하지 않고 확실하게 작동해야 합니다.

1. Thread를 사용할 때 지금까지 학습한 것과 같이 Thread 생성자를 이용하여 Thread를 생성할 수도 있지만, 실제로 응용프로그램을 작성할 때는 표준라이브러리가 제공하는 추가적인 api(클래스와 인터페이스)를 사용하면 편리하기도 하고 에러도 줄일 수 있습니다. 이 문제에서는 Thread pool을 사용하는 방법을 공부합니다.

아래 [ThreadPool.java](src\ThreadPool.java) 코드를 잘 읽어 보세요. 주석을 잘 보고 프로그램의 동작을 이해하도록 하세요.

A screenshot of a computer code

Description automatically generated

A screenshot of a computer code

Description automatically generated

위 프로그램을 실행하면 아래와 같은 출력이 나옵니다. 출력이 나타난 시간을 프로그램 코드와 함께 주의 깊게 보세요.

A screenshot of a computer program

Description automatically generated

1, 2번 줄은 main 쓰레드에서 **Runnable 태스크**를 제출(execute 또는 submit)하고 나서 출력한 것입니다.

3, 4번 줄은 thread pool의 두 쓰레드(pool-1-thread-1, pool-1-thread-2)에서 각각 task를 끝내면서 출력한 것입니다.

3, 4번 줄에 나타난 시간이 1, 2번 줄의 시간보다 나중임에 유의하세요. es.execute(Runnable), es.submit(Runnable) 호출은 즉시 반환하기 때문입니다.

6번 줄은 main 쓰레드에서 **Callable 태스크**를 제출(submit)하고 나서 출력한 것입니다. es.submit(Callable) 호출도 즉시 Future<String>을 반환합니다.

7번 줄은 thread pool의 한 쓰레드(pool-1-thread-2)에서 task를 끝내면서 출력한 것입니다.

8번 줄은 main 쓰레드가 future.get()으로 result를 얻은 직후에 출력한 것입니다. 시간을 보면 7번 줄의 시간과 거의 같습니다. 그러니까 소스코드 65번 줄의 future.get() 메소드가 즉시 반환하지 않고 block되어 있다가 task가 끝난 후에 반환된다는 것을 확인할 수 있습니다.

12번 줄은 main 쓰레드에서 여러 Callable 태스크들을 한꺼번에 제출(invokeAll)하기 직전에 출력한 것입니다.

13-15번 줄은 Callable 태스크를 실행하는 세 개의 쓰레드 각각이 태스크를 끝내면서 출력한 것입니다.

16번 줄은 invokeAll 메소드가 반환된 직후에 main 쓰레드가 출력한 것입니다.

16번 줄에 나타난 시간이 13-15번 줄의 시간보다 나중임을 볼 수 있고, invokeAll 메소드는 task들이 모두 끝난 후에야 비로소 결과를 반환한다는 것을 확인할 수 있습니다.

Executors.newCachedThreadPool()은 필요한 개수의 쓰레드들을 갖는 쓰레드풀을 만들어주는 static factory 메소드입니다. 이 메소드가 만들어 주는 객체는 단순한 쓰레드풀이 아니고 여러가지 기능을 추가로 갖는 ExecutorService 타입 객체입니다.

1. 이 문제에서는 Thread pool을 사용하는 조금 더 실용적인 예를 보겠습니다. 이 예에서는 배열에 들어 있는 정수들의 합을 구합니다. 배열이 매우 큰 경우, 단순 반복문을 이용하여 순차적으로 계산을 하면 시간이 많이 걸리므로, 여러 쓰레드들에게 덧셈 작업을 나누어 시키고 그 결과들을 모아 최종 덧셈을 하면 계산 시간을 많이 줄일 수 있습니다. 아래 [ThreadPool2.java](src\ThreadPool2.java) 코드와 주석을 잘 읽어보고 실행해 보세요.

A screenshot of a computer code

Description automatically generated

A screen shot of a computer

Description automatically generated

쓰레드풀을 이용한 인위적인 병렬 계산에 비해 스트림을 이용한 컴퓨터 자율 병렬 계산이 더 빠르게 실행되었습니다.

또, 단순반복을 이용한 직렬 계산이 스트림 방법에 비해서는 느리지만 쓰레드풀 방법에 비해서는 오히려 빠르게 실행되었습니다. 작은 문제이므로 계산 시간에 비해 쓰레드풀을 조작하는 오버헤드가 크기 때문입니다. 문제(배열)가 더 커지면 직렬계산보다는 병렬계산이 빠르게 실행될 것입니다.

이 문제는 성격상 큰 문제를 작은 문제로 쪼개도 작은 문제 서로 간 간섭이 없습니다. 즉, 스레드 간 공유데이터가 없습니다. 그래서 병렬처리의 이득을 충분히 누릴 수 있습니다. 스레드 간 공유데이터가 있을 수밖에 없는 문제의 경우, 큰 문제를 작은 문제로 나누어 병렬로 계산을 한다고 하더라도 스레드간 동기화를 위해 쓰레드들이 대기하는 시간이 있습니다. 그런 경우에는 병렬처리의 이득이 크지 않습니다.

100\_000과 같이 숫자 속에 “\_”를 넣은 것은 사람이 읽기 쉽게 하기 위함입니다. 컴파일러는 이 글자를 무시합니다. 프로그래머는 원하는 위치에 필요한 개수의 “\_” 글자를 삽입할 수 있습니다.

1. 이 문제에서는 Thread-safe 객체에 대해 설명합니다. 라이브러리 클래스나 이런 저런 설명서를 읽다보면 “thread-safe”라는 단어가 가끔씩 나옵니다. 아래 [ThreadSafe.java](src\ThreadSafe.java) 프로그램에서 이에 대해 설명합니다. 코드와 주석을 잘 읽어보고 프로그램을 실행해 보세요. 실행할 때마다 출력이 달라집니다.

A white background with text

Description automatically generated

이 프로그램의 출력 예는 아래와 같습니다. 물론 실행할 때마다 달라집니다.

A screenshot of a computer

Description automatically generated

Thread-safe한 객체에는 여러 쓰레드가 경쟁적으로 접근하여 값을 변경하더라도 데이터가 깨지지 않음을 확인할 수 있습니다.

1. 이 문제에서는 비동기식 (asynchronous) 계산에 대해 설명합니다. 비동기식 계산이란 main 쓰레드가 하는 작업과 다른 쓰레드가 하는 작업이 시간적으로 서로 얽히지 않는다는 말입니다. 비동기식 계산을 위해서는 CompletableFuture를 사용하면 편리합니다. 아래 [SimpleDemo.java](src\SimpleDemo.java) 프로그램에서 이에 대해 설명합니다. 코드와 주석을 잘 읽어보고 프로그램을 실행해 보세요. 실행할 때마다 출력이 달라집니다.

A computer screen shot of a program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer code

Description automatically generated

출력의 첫 번째 예는 아래 그림과 같습니다.

A screenshot of a computer program

Description automatically generated

Chain으로 엮여 있는 비동기 작업들이 순차적으로 ForkJoinPool의 쓰레드에 의해 실행되었습니다. main 메소드의 작업들은 모두 main 쓰레드에 의해 실행되었고, 비동기 작업들이 끝나기 전에 main 쓰레드가 자기 할 일들을 다 하고 먼저 끝났습니다.

출력의 다른 예를 봅니다.

A screenshot of a computer

Description automatically generated

Chain으로 엮여 있는 비동기 작업들이 다섯 개 있는데 그 중 첫 두 개는 ForkJoinPool의 쓰레드에 의해 실행되었지만 나머지 세 작업은 main 쓰레드에 의해 실행되었습니다. VM(Virtual Machine)은 시스템 효율을 높이기 위해 형편이 되는대로 작업들을 임의의 쓰레드에 할당한다는 점을 확인할 수 있습니다.

또 다른 출력 예입니다.

A screenshot of a computer

Description automatically generated

프로그램 실행은 정상적으로 끝난 상태입니다. 이클립스 콘솔 창에 프로그램 행 중임을 나타내는 빨간 네모 불이 들어와 있지 않고 에러 메세지도 없으므로 정상적으로 프로그램이 종료된 것인데 출력을 보면 마치 일을 하다 만 것처럼 보입니다.

이것은 위에서 공부한 경쟁조건(race condition)이 만들어졌기 때문입니다. 출력을 담당하는 System.out 객체를 여러 쓰레드가 경쟁적으로 사용하기 때문에 출력데이터가 깨져서 제대로 출력이 되지 않은 것뿐입니다.

이 출력 문제를 해결하는 여러 방법이 있지만 이번 주 주제와는 다소 거리가 있기 때문에 다루지 않습니다만, System.out 객체에 대한 쓰레드 접근을 동기화하거나, System.out 대신 시스템과 프로그램의 작업 경과를 기록하는데 사용하는 Logger를 사용하는 수가 있습니다. 단, Logger은 main 쓰레스에 의해서만 작동하기 때문에 쓰레드 작동 상황을 파악하기는 어렵습니다.

끝.