# JAVA Programming

*Practice 7: Multi-thread*

BeomSeok Kim
Department of Computer Engineering
KyungHee University
passion0822@khu.ac.kr

# [실습 7-1] 다중 쓰레드

- 두개의 쓰레드가 동작하는 프로그램 작성
  - ✓ 첫 번째 쓰레드: Hello를 출력
  - ✓ 두 번째 쓰레드: Goodbye를 출력

  - ✓ 요구사항
    - ➢ Runnable 인터페이스를 상속받는 GreetingRunable 클래스를 구현
      - 출력할 문자열을 저장할 greeting 변수
      - 생성자에서는 출력할 문구를 입력값으로 가져와 greeting에 저장
      - Run() 메소드 오버라이딩
        - » 현재시간 + greeting을 화면에 출력 후, 1000 millisecond 만큼 지연
        - » 오버라이딩 구현 시 try-catch를 활용하여 예외처리를 할 수 있는 틀을 만들어 둘 것.

# [실습 7-1] 다중 쓰레드

- GreetingRunnable.java

```
import java.util.Date;

/**
A runnable that repeatedly prints a greeting.
*/
public class GreetingRunnable implements Runnable {
    private static final int REPETITIONS = 10;
    private static final int DELAY = 1000;

    private String greeting;

    /**
    Constructs the runnable object.
    @param aGreeting the greeting to display */
    public GreetingRunnable(String aGreeting) {
        greeting = aGreeting;
    }

    public void run() {
        try {
            for (int i = 1; i <= REPETITIONS; i++) {
                Date now = new Date();
                System.out.println(now + " " + greeting);
                Thread.sleep(DELAY);
            }
        }
        catch (InterruptedException exception) {
        }
    }
}
```

# [실습 7-1] 다중 쓰레드

- GreetingThreadRunner.java

```
/**
This program runs two greeting threads in parallel.
*/

public class GreetingThreadRunner {
    public static void main(String[] args) {
        GreetingRunnable r1 = new GreetingRunnable("Hello");
        GreetingRunnable r2 = new GreetingRunnable("Goodbye");
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
}
```

# [실습 7-2] 동기화문제: Race condition

- 강의자료에서 나오는 Bank account 문제의 구현
  - ✓ 클래스 구조
    - ➢ BankAccountRunner
      - – 뼈대 클래스
    - ➢ DepositRunnable
      - – 입금 thread를 구현
    - ➢ WithdrawRunnable
      - – 출금 thread를 구현
    - ➢ BankAccount
      - – 은행계좌 클래스

# [실습 7-2] 동기화문제: Race condition

- BankAccountRunner.java

```
/**
This program runs threads that deposit and withdraw
Money from the same bank account
*/

public class BankAccountThreadRunner
{
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        final double AMOUNT = 100;
        final int REPETITIONS = 100;
        final int THREADS = 100;

        for (int i = 1; i <= THREADS; i++) {
            DepositRunnable d = new DepositRunnable( account, AMOUNT, REPETITIONS);
            WithdrawRunnable w = new WithdrawRunnable( account, AMOUNT, REPETITIONS);
            Thread dt = new Thread(d);
            Thread wt = new Thread(w);

            dt.start();
            wt.start();
        }
    }
}
```

# [실습 7-2] 동기화문제: Race condition

- DepositRunner.java

```
/**
A deposit runnable makes periodic deposits to a bank account.
*/

public class DepositRunnable implements Runnable {
    private static final int DELAY = 1;
    private BankAccount account;
    private double amount;
    private int count;

    /**
    Constructs a deposit runnable.
    @param anAccount the account into which to deposit money
    @param anAmount the amount to deposit in each repetition
    @param aCount the number of repetitions
    */
    public DepositRunnable(BankAccount anAccount, double anAmount, int aCount) {
        account = anAccount; amount = anAmount; count = aCount;
    }
    public void run() {
        try {
            for (int i = 1; i <= count; i++) {
                account.deposit(amount);
                Thread.sleep(DELAY);
            }
        }
        catch (InterruptedException exception) {
        }
    }
}
```

# [실습 7-2] 동기화문제: Race condition

- WithdrawRunner.java

```
/**
A withdraw runnable makes periodic withdrawals from a bank account.
*/

public class WithdrawRunnable implements Runnable {
    private static final int DELAY = 1;
    private BankAccount account;
    private double amount;
    private int count;

/**
Constructs a withdraw runnable.
@param anAccount the account from which to withdraw money
@param anAmount the amount to withdraw in each repetition
@param aCount the number of repetitions
*/
    public WithdrawRunnable(BankAccount anAccount, double anAmount, int aCount) {
        account = anAccount;
        amount = anAmount;
        count = aCount;
    }

    public void run() {
        try {
            for (int i = 1; i <= count; i++) {
                account.withdraw(amount);
                Thread.sleep(DELAY);
            }
        }
        catch (InterruptedException exception) {
        }
    }
}
```

# [실습 7-2] 동기화문제: Race condition

- BankAccount.java

```
/**
A bank account has a balance that can be changed by deposits and withdrawals.
*/
public class BankAccount {
    private double balance;

    /**
    Constructs a bank account with a zero balance.
    */
    public BankAccount() {
        balance = 0;
    }

    /**
    Deposits money into the bank account.
    @param amount the amount to deposit
    */
    public void deposit(double amount) {
        System.out.print("Depositing " + amount);
        double newBalance = balance + amount;
        System.out.println(", new balance is " + newBalance);
        balance = newBalance;
    }
```

# [실습 7-2] 동기화문제: Race condition

- BankAccount.java (cont'd)

```
    /**
    Withdraws money from the bank account.
    @param amount the amount to withdraw
    */
    public void withdraw(double amount) {
        System.out.print("Withdrawing " + amount);
        double newBalance = balance - amount;
        System.out.println(", new balance is " + newBalance);
        balance = newBalance;
    }

    /**
    Gets the current balance of the bank account.
    @return the current balance
    */
    public double getBalance() {
        return balance;
    }
}
```

# [실습 7-3] 동기화문제: Race condition

- 실습 7-2 문제에서 발견할 수 있는 동기화문제를 해결하시오
  - ✓ Hint:
    - ➢ Lock(), unlock()

# [실습 7-4] 동기화문제: Avoiding deadlocks

- Deadlock
  - ✓ 다수의 쓰레드가 2개 이상의 공유자원에 접근할 때 발생하는 문제
    - ➢ 자원 사용의 권한을 상호 기다림으로써 전체 프로그램이 의도치 않게 정지되는 현상
  - ✓ 해결방안
    - ➢ 논리적인 문제로 개발자가 프로그램 및 공유자원을 접근하는 알고리즘을 설계 시 사려깊게 접근해야함

# [실습 7-2] 동기화문제: Race condition

- BankAccount.java

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReenterantLock;

/**
A bank account has a balance that can be changed by deposits and withdrawals.
*/
public class BankAccount {
    private double balance;
    private Lock balanceChangeLock;
    private Condition sufficientFundsCondition;
    /**
    Constructs a bank account with a zero balance.
    */
    public BankAccount() {
        balance = 0;
        balanceChangeLock = new ReentrantLock();
        sufficientFundsCondition = balanceChangeLock.newCondition()
    }

    /**
    Deposits money into the bank account.
    @param amount the amount to deposit
    */
    public void deposit(double amount) {
        System.out.print("Depositing " + amount);
        double newBalance = balance + amount;
        System.out.println(", new balance is " + newBalance);
        balance = newBalance;
    }
```

# [실습 7-2] 동기화문제: Race condition

- BankAccount.java

```
/**
Deposits money into the bank account.
@param amount the amount to deposit
*/
public void deposit(double amount) {
    balanceChangeLock.lock();
    try {
        System.out.print("Depositing " + amount);
        double newBalance = balance + amount;
        System.out.println(", new balance is " + newBalance);
        balance = newBalance;
        sufficientFundsCondition.signalAll();
    }
    finally {
        balanceChangeLock.unlock();
}
```

# [실습 7-2] 동기화문제: Race condition

- BankAccount.java (cont'd)

```
    /**
    Withdraws money from the bank account.
    @param amount the amount to withdraw
    */
    public void withdraw(double amount)
          throws InterruptedException {
        try {
            while (balance < amount) {
                sufficientFundsCondition.await();
            }
            System.out.print("Withdrawing " + amount);
            double newBalance = balance - amount;
            System.out.println(", new balance is " + newBalance);
            balance = newBalance;
        }
        finally {
            balanceChangeLock.unlock();
    }

    /**
    Gets the current balance of the bank account.
    @return the current balance
    */
    public double getBalance() {
        return balance;
    }
}
```

# [실습 7-4] 동기화문제: Avoiding deadlocks

- BankAccountThreadRunner.java

```
/**
This program runs threads that deposit and withdraw
Money from the same bank account
*/

public class BankAccountThreadRunner
{
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        final double AMOUNT = 100;
        final int REPETITIONS = 100;
        final int THREADS = 100;

        for (int i = 1; i <= THREADS; i++) {
            DepositRunnable d = new DepositRunnable( account, AMOUNT, REPETITIONS);
            WithdrawRunnable w = new WithdrawRunnable( account, AMOUNT, REPETITIONS);
            Thread dt = new Thread(d);
            Thread wt = new Thread(w);

            dt.start();
            wt.start();
        }
    }
}
```

# [실습 7-2] 동기화문제: Race condition

- DepositRunner.java

```
/**
A deposit runnable makes periodic deposits to a bank account.
*/

public class DepositRunnable implements Runnable {
    private static final int DELAY = 1;
    private BankAccount account;
    private double amount;
    private int count;

    /**
    Constructs a deposit runnable.
    @param anAccount the account into which to deposit money
    @param anAmount the amount to deposit in each repetition
    @param aCount the number of repetitions
    */
    public DepositRunnable(BankAccount anAccount, double anAmount, int aCount) {
        account = anAccount; amount = anAmount; count = aCount;
    }
    public void run() {
        try {
            for (int i = 1; i <= count; i++) {
                account.deposit(amount);
                Thread.sleep(DELAY);
            }
        }
        catch (InterruptedException exception) {
        }
    }
}
```

# [실습 7-2] 동기화문제: Race condition

- WithdrawRunner.java

```
/**
A withdraw runnable makes periodic withdrawals from a bank account.
*/

public class WithdrawRunnable implements Runnable {
    private static final int DELAY = 1;
    private BankAccount account;
    private double amount;
    private int count;

/**
Constructs a withdraw runnable.
@param anAccount the account from which to withdraw money
@param anAmount the amount to withdraw in each repetition
@param aCount the number of repetitions
*/
    public WithdrawRunnable(BankAccount anAccount, double anAmount, int aCount) {
        account = anAccount;
        amount = anAmount;
        count = aCount;
    }

    public void run() {
        try {
            for (int i = 1; i <= count; i++) {
                account.withdraw(amount);
                Thread.sleep(DELAY);
            }
        }
        catch (InterruptedException exception) {
        }
    }
}
```

# Thank You!
# Q&A