Outline

• Synchronization
Motivation for Synchronization

- Thread synchronization is used to coordinate the execution of dependent threads
  - Sometimes we want to have some control over when threads execute or what code they are able to simultaneously access

- **Shared resources** pose potential problems to multi-threaded code since we don’t typically have control over when a thread will be switched into and out of the CPU
Synchronization Program Analogy

▪ Assume I brought a piggy bank to class (shared resource)
▪ 100 students in the class have to deposit a penny into the piggy bank (independent threads)
▪ How many pennies should be in the bank at the end?
Synchronization Example #1

```java
public class AddAPenny implements Runnable {
    private static PiggyBank piggy = new PiggyBank();

    public void run() {
        piggy.deposit(1);
    }

    public static void main(String[] args) {
        for (int i=0; i < 100; i++) {
            Thread t = new Thread(new AddAPenny());
            t.start();
        }
        System.out.println("Balance = " + piggy.getBalance());
    }
}

class PiggyBank {
    private int balance = 0;
    public int getBalance() {
        return balance;
    }
    public void deposit(int amount) {
        int newBalance = balance + amount;
        Thread.yield();
        balance = newBalance;
    }
}
```

---

4 Executions

- Balance = 9
- Balance = 6
- Balance = 12
- Balance = 10
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class AddAPenny implements Runnable {
    private static PiggyBank piggy = new PiggyBank();

    public void run() {
        piggy.deposit(1);
    }

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        for (int i = 0; i < 100; i++) {
            executor.execute(new AddAPenny());
        }
        executor.shutdown();
        // wait until all tasks are finished
        while (!executor.isTerminated()) {
            Thread.yield();
        }
        System.out.println("Balance = " + piggy.getBalance());
    }
}

class PiggyBank {
    private int balance = 0;
    public int getBalance() {
        return balance;
    }
    public void deposit(int amount) {
        int newBalance = balance + amount;
        Thread.yield();
        balance = newBalance;
    }
}
Synchronization Example #3

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

public class AddAPenny implements Runnable {
    private static PiggyBank piggy = new PiggyBank();

    public void run() {
        piggy.deposit(1);
    }

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        for (int i=0; i < 100; i++) {
            executor.execute(new AddAPenny());
        }
        executor.shutdown();
        while(!executor.isTerminated()) {
            Thread.yield();
        }
        System.out.println("Balance = " + piggy.getBalance());
    }
}

class PiggyBank {
    private int balance = 0;
    public int getBalance() {
        return balance;
    }
    public void deposit(int amount) {
        balance += amount;
    }
}
Synchronization Overview

- The problems in the previous examples are based on the use of a shared variable across multiple threads (the PiggyBank object piggy)
  - The OS switches threads out of the CPU at times unknown to us
  - This seems to be happening inside the `deposit(int)` method after `newBalance` is set but before it is assigned back to `balance`
  - This causes the old value of `balance` to be used in subsequent threads

- When we reduce the `deposit(int)` method to a single statement, we get higher values for `balance` because there are fewer locations at which the OS can switch the thread out of the CPU before setting the value of `balance`
Race Conditions

- A race condition is a state where more than one thread is accessing the same variable and overwriting the other thread’s updates
  - This is a common problem in multi-threaded programming

- A class is thread-safe if an object of the class does not cause a race condition in the presence of multiple threads
  - The AddAPenny class in the previous examples is not thread-safe
To avoid race conditions, it is necessary to prevent more than one thread from simultaneously entering a certain part of the program. This part of the program is called a critical section or critical region.

In the previous examples, the critical section would be the `deposit(int)` method in the `PiggyBank` class.

If only one thread at a time was able to access the `deposit(int)` method, we would not have a race condition on the shared variable `balance`.

- Note that `balance` is shared in the static instance of `PiggyBank`.

```java
public void deposit(int amount) {
    int newBalance = balance + amount;
    Thread.yield();
    balance = newBalance;
}
```