Outline

• Locks and Conditions
The `synchronized` keyword puts a restriction on a block of code that only one thread can be inside at a time.

- This is accomplished using a monitor.
- No other thread will be able to enter that block of code if another thread is currently executing inside of it, regardless of whether it is in the CPU currently or not.
Explicitly Acquiring Locks

- A synchronized method or block of code implicitly acquires a lock on the instance before executing.
- Java gives us the ability to explicitly acquire locks using the `java.util.concurrent.locks.Lock` interface and `java.util.concurrent.locks.ReentrantLock` class.

```java
public class MyClass {
    private final ReentrantLock lock = new ReentrantLock();

    public synchronized void doSomething() {
        // Acquire the lock
        lock.lock();
        try {
            // Perform operation
        } finally {
            // Release the lock
            lock.unlock();
        }
    }
}
```
Reentrant Locks

- A reentrant lock is a lock that does not need to be acquired more than once if entering a second critical section on the same lock
  - Monitors use reentrant locks

```java
1  class Reentrant {
2    private synchronized void foo() {
3        System.out.println("1");
4        bar();
5        System.out.println("3");
6    }
7    private synchronized void bar() {
8        System.out.println("2");
9    }
10   public static void main(String[] args) {
11      Reentrant r = new Reentrant();
12      r.foo();
13   }
14 }
```

Once a thread enters `foo()`, it has obtained the lock on the object

Since the thread already has the lock, it does not need to acquire it again to get into `bar()`
```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) {
        int newBalance = balance + amount;
        Thread.yield();
        balance = newBalance;
    }
}
```

### 4 Executions

- **Balance = 4**
- **Balance = 6**
- **Balance = 7**
- **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 synchronized void deposit(int amount) {
        int newBalance = balance + amount;
        Thread.yield();
        balance = newBalance;
    }
}
Explicitly Acquiring Locks Example

```java
import java.util.concurrent.*; // import * to save space
import java.util.concurrent.locks.*; // import * to save space

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 synchronized void deposit(int amount) {
    int newBalance = balance + amount;
    Thread.yield();
    balance = newBalance;
  }
}
```
Let’s modify our **AddAPenny** program to also allow a thread to remove a penny

Create a `withdraw(int)` method in the **PiggyBank** class

Have half of the threads deposit a penny and half of the threads withdraw a penny
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class AddAndRemoveAPenny implements Runnable{
  private static PiggyBank piggy = new PiggyBank();
  private boolean isWithdrawal;

  public void run() {
    if (isWithdrawal) {
      piggy.withdraw(1);
    } else {
      piggy.deposit(1);
    }
  }

  public static void main(String [] args) {
    ExecutorService executor = Executors.newCachedThreadPool();
    for (int i=0; i < 100; i++) {
      AddAndRemoveAPenny penny = new AddAndRemoveAPenny();
      if (i < 50) { // exactly 50 threads will deposit
        penny.isWithdrawal = false;
      } else { // and exactly 50 threads will withdraw
        penny.isWithdrawal = true;
      }
      executor.execute(penny);
    }
    executor.shutdown();
    while(!executor.isTerminated()) {
      Thread.yield();
    }
    System.out.println("balance = "+piggy.getBalance());
  }
}

class PiggyBank {
  private int balance = 0;
  private Lock lock = new ReentrantLock();
  public int getBalance() {
    return balance;
  }
}

public void withdraw(int amount) {
  lock.lock();
  try {
    while (balance < amount) {
      System.out.print("Waiting for deposit...");
      System.out.print(" to withdraw "+amount);
      System.out.println(" from balance of "+balance);
    }
    balance -= amount;
    System.out.print("$" + amount + " withdrawn,");
    System.out.println(" leaving balance of "$ + balance);
  } finally {
    lock.unlock();
  }
}

public void deposit(int amount) {
  lock.lock();
  try {
    System.out.print("Depositing "$ + amount + ", ");
    int newBalance = balance + amount;
    Thread.yield();
    balance = newBalance;
    System.out.println("giving balance of "$ + balance);
  } finally {
    lock.unlock();
  }
}
In the previous example, sometimes everything works properly.

And other times we end up in an endless loop because we are stuck in the `withdraw` method's `while` loop, holding onto the lock, which does not allow any deposits.
Thread Conditions

- Threads are able to communicate with each other based on specific conditions.
- Threads can perform the following actions using the `java.util.concurrent.Condition` interface:
  - `await()`: Wait for the condition to be signaled.
  - `signal()`: Wake up one thread waiting on the condition.
  - `signalAll()`: Wake up all of the threads waiting on the condition.
- A `Condition` is created from a `Lock` object by calling the `newCondition()` method.
Thread Condition Issues

- Once a thread invokes `await()` on a condition, the thread will move to the waiting state until it is signaled
  - If `signal()` or `signalAll()` is never called on the condition, the thread will wait forever

- You must first obtain the lock on the object before you are able to invoke a method on its `Condition`
  - When a thread calls `await()`, it will release its lock
    - It will not be able to start executing again, even after it is signaled, if it is not able to obtain the lock again
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class AddAndRemoveAPenny implements Runnable{
    private static PiggyBank piggy = new PiggyBank();
    private boolean isWithdrawal;
    public void run() {
        if (isWithdrawal) {
            piggy.withdraw(1);
        }
        else {
            piggy.deposit(1);
        }
    }
    public static void main(String [] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        for (int i=0; i < 100; i++) {
            AddAndRemoveAPenny penny = new AddAndRemoveAPenny();
            if (i < 50) { // exactly 50 threads will deposit
                penny.isWithdrawal = false;
            } else { // and exactly 50 threads will withdraw
                penny.isWithdrawal = true;
            }
            executor.execute(penny);
        }
        executor.shutdown();
        while(!executor.isTerminated()) {
            Thread.yield();
        }
        System.out.println("balance = " + piggy.getBalance());
    }
}

class PiggyBank {
    private int balance = 0;
    private Lock lock = new ReentrantLock();
    private Condition depositMade = lock.newCondition();
    public int getBalance() {
        return balance;
    }
    public void withdraw(int amount) {
        lock.lock();
        try {
            while (balance < amount) {
                System.out.print("Waiting for deposit...");
                System.out.print(" to withdraw " + amount);
                System.out.println(" from balance of "+ balance);
                depositMade.await();
            }
            balance -= amount;
            System.out.print("$" + amount + " withdrawn,",
            System.out.println("giving balance of "+ balance);
            depositMade.signalAll();
        }finally {
            lock.unlock();
        }
    }
    public void deposit(int amount) {
        lock.lock();
        try {
            System.out.print("Depositing "+ amount + ", ");
            int newBalance = balance + amount;
            Thread.yield();
            balance = newBalance;
            System.out.println("giving balance of "+ balance);
            depositMade.signalAll();
        }finally {
            lock.unlock();
            System.out.println("deposit Made.signalAll();
        }finally {
            lock.unlock();
        }
    }
}