ECE 462 Exam 3

08:10-09:20AM, November 20, 2009

I certify that I will not receive nor provide aid to any other student for this exam.

Signature:

You must sign here. Otherwise, the exam is not graded.

This exam is printed **double sides**.

Write your answers next to the questions. If you need more space, you can use the two blank pages.

This is an *open-book, open-note* exam. You can use any book or note or program printouts.

Please turn off your cellular phone **now** .

Two outcomes are tested in this exam. To pass each outcome, you must receive 50% of the points.

- Outcome 7: exception handling
- Outcome 8: multiple threads and synchronization

There are 9 questions in this exam. If a program has a syntax error, **point out** which line causes the error. If there are multiple errors, you need to write only one of them.

Contents

Total Score:

Passed Outcomes:

This page is blank. You can write answers here.

This page is blank. You can write answers here.

1 Java Exception (outcome 7)

Write function `f`

- If the input argument `i` is larger than zero, throw an exception object of class `Exception`.
- Otherwise, return 1.

Answer:

```
    public int f(int i) throws Exception {
        if (i > 0) {
            throw new Exception();
        }
        return 1;
    }
```

```
import java.io.*;

class X {
    public int f(int i)
    //
    // *** FIX ME ***
    //
}
class outcome71 {
    public static void main( String[] args )
    {
        X xobj = new X();
        try {
            System.out.println("calling f(-3)");
            xobj.f(-3);
        } catch(Exception e) {
            System.out.println("Exception caught" );
        }

        try {
            System.out.println("calling f(5)");
            xobj.f(5);
        } catch(Exception e) {
            System.out.println("Exception caught" );
        }
    }
}
```

2 Java Exception (outcome 7)

Briefly explain the difference between `final` and `finally` in Java. In Java, `final` has multiple usages. You need to explain **at least** two of them.

Answer:

`finally`*: after* `try-catch` *block. The code inside* `finally` *will be executed regardless whether an exception has been caught.*

`final`*: several meanings in Java. If a class is* `final`*, this class cannot have derived classes. If an argument is* `final`*, the argument cannot be changed inside the function. If a function is* `final`*, this function cannot be overridden in derived classes.*

3 Java and C++ Exception (outcome 7)

Briefly explain the differences of exception handling in C++ and Java.

Answer:

	Java	C++
What type can be thrown	**must** be an object of `Exception` or its derived classes	primitive types such as `int` as well as objects
Function declaration	If a function may throw an exception, the function **must** declare so before {	not required
If a function may throw an exception	**must** be enclosed in a `try-catch` block	not required
Inside `catch (ExceptionType obj)`	`obj` is required	not required
Inside a `catch` block, an exception **can** be thrown	allowed	allowed
After one `try` block, **multiple** `catch` blocks for different types of exceptions	allowed	allowed

	Java	C++
What type can be thrown		primitive types such as `int` as well as objects
Function declaration	If a function may throw an exception, the function **must** declare so before {	
If a function may throw an exception	**must** be enclosed in a `try-catch` block	
Inside `catch (ExceptionType obj)`	`obj` is required	
Inside a `catch` block, an exception **can** be thrown		allowed
After one `try` block, **multiple** `catch` blocks for different types of exceptions	allowed	

4 C++ Exception Handling (outcome 7)

Write the `catch` block so that

- If `f` throws an exception of `ExceptionType1`, the program prints "caught ExceptionType1".
- If `f` throws an exception of `ExceptionType2`, the program prints "caught ExceptionType2".

Answer:

```
catch (ExceptionType2 et2 ) {
    cout << "caught Type 2 " << endl;
  } catch (ExceptionType1 et1 ) {
    cout << "caught Type 1 " << endl;
  }
```

"Type2" must be caught first.

```
#include <iostream>
#include <string>
using namespace std;

class ExceptionType1
{
};

class ExceptionType2: public ExceptionType1
{
  // *****
  // ATTENTION: Type2 is a derived class of Type1
  // *****
};

void f (int i) throw (ExceptionType1, ExceptionType2)
{
  switch (i)
    {
    case 1:
      throw ExceptionType1();
      break;
    case 2:
      throw ExceptionType2();
      break;
    default:
      cout << "no exception" << endl;
    }
}

int main()
{
  cout << "Enter a number : ";
  int v;
  cin >> v;
  try {
    cout << "f(" << v << ")" << endl;
    f(v);
  }
  // *****
  // FIX ME
  // *****
  return 0;
}
```

5 Java Exception (outcome 7)

What is the output of this program? **Explain your answer.**

Answer:

0

`xobj.f(0);` *throws* `Exception` *and it is caught in the last* `catch`*.* `val` *is never changed.*

```
class ExceptionDerived1 extends Exception {
}

class ExceptionDerived2 extends ExceptionDerived1 {
}

class ExceptionDerived3 extends ExceptionDerived2 {
}

class X {
    private int val = 0;
    public void f(int i) throws Exception,
                                ExceptionDerived1,
                                ExceptionDerived2,
                                ExceptionDerived3 {
        if (i == 0) {
            throw new Exception();
        }
        if (i == 1) {
            val ++;
            throw new ExceptionDerived1();
        }
        if (i == 2) {
            val += 2;
            throw new ExceptionDerived2();
        }
        if (i == 3) {
            val += 3;
            throw new ExceptionDerived3();
        }
        System.out.println("no exception");
    }
    public int getVal() {
        return val;
    }
}
```

```
class outcome75
{
    public static void main(String[] args)
    {
        X xobj = new X();
        try {
            xobj.f(0);
            xobj.f(1);
            xobj.f(2);
            xobj.f(3);
        } catch (ExceptionDerived2 e1) {
            try {
                xobj.f(2);
            } catch (Exception e) {

            }
        } catch (ExceptionDerived1 e3) {
            try {
                xobj.f(2);
            } catch (Exception e2) {

            }
        } catch (Exception e) {
        }
        System.out.println(xobj.getVal());
    }
}
```

6 Timer using Thread (outcome 8)

Create a timer class so that it periodically calls the `update` function of an `Actuator` object.

Answer:

```
    public Outcome8Timer (Actuator a, int d, int r)
    {
        t_act = a;
        t_delay = d;
        t_repeat = r;
    }
    public void run()
    {
        for (int r = 0; r < t_repeat; r++)
```

```
            {
                try
                    {
                        t_act.update();
                        sleep(t_delay);
                    }
                catch (Exception ept)
                    {
                        System.out.println("caught exception");
                    }
            }
    }

class Actuator {
    public void update()
    {
        System.out.println("update called at " +
                            System.currentTimeMillis());
    }
}

class Outcome8Timer extends Thread {
    private Actuator t_act;
    private int t_delay;
    private int t_repeat;
    // >>>>>
    // The constructor initializes the three attributes
    // <<<<<





    // >>>>>
    // when start is called, call t_act.update() once every
    // t_delay millisecond.  This repeats t_repeat times.
    // If you use "sleep", please remember to enclose it inside
    // a try block.
    // <<<<<




}
public class outcome81 {
```

```
    public static void main( String[] args ) {
        Actuator act = new Actuator();
        Outcome8Timer t = new Outcome8Timer(act, 1500, 20);
        // call act.update every 1500 millisecond for 20 times
        t.start();
        try
            {
                t.join();
            }
        catch (Exception ept)
            {
                System.out.println("caught exception");
            }
    }
}
```

7 Java Thread (outcome 8)

This program tries accumulate numbers. The correct answer is 800000. Does the program **always** print 800000 in the last statement? **Explain your answer.**

Answer:

No. The answer is not always 800000. There are multiple thread objects so there are multiple locks. The static variable `total` *has to be protected by a single lock.*

```
import java.util.*;

class Accumulator extends Thread
{
    public static long iteration; // ATTENTION: static
    public static long total = 0;

    public Accumulator(long i) { iteration = i;  }

    synchronized public void incrTotal() // ATTENTION: synchronized
    { total ++; }

    public static long getTotal() { return total; }

    public void run()
    {
        for (long iter = 0; iter < iteration; iter ++)
            { incrTotal(); }
```

```
    }
}

class outcome82
{
    public static void main( String[] args )
    {
        long iteration = 1000;
        int numThread = 800;
        System.out.println("Iteration = " + iteration +
                           " Thread = " + numThread);
        System.out.println("Correct Answer = " +
                           iteration * numThread);
        Accumulator [] accum = new Accumulator[numThread];
        for (int tcnt = 0; tcnt < numThread; tcnt ++)
            { accum[tcnt] = new Accumulator(iteration); }

        for (int tcnt = 0; tcnt < numThread; tcnt ++)
            { accum[tcnt].start(); }

        try {
            for (int tcnt = 0; tcnt < numThread; tcnt ++)
                { accum[tcnt].join(); }
        } catch (InterruptedException ie) {
            System.out.println("exception caught");
        }
        System.out.println("Total = " + Accumulator.getTotal());
    }
}
```

8 Java Deadlock (outcome 8)

Is it **possible** that the following program encounters a deadlock? **Explain your answer.** Consider the four necessary conditions for deadlock. If it is possible, explain how the necessary conditions are met. If it is not possible, explain which condition (or conditions) is not met.

Answer:

No. There is no circular wait.

```
class Account {
    synchronized void trans1 () { // ATTENTION: synchronized
        trans2(); // ATTENTION: calls another synchronized method
    }
```

```
    synchronized void trans2 () {
    }
}

class TransThread extends Thread {
    Account acnt;
    TransThread(Account a) {
        acnt = a;
    }
    public void run() {
        acnt.trans1();
    }
}

public class outcome83 {
    public static void main( String[] args ) {
        Account act1 = new Account();
        Account act2 = new Account();
        TransThread tt1 = new TransThread(act1);
        TransThread tt2 = new TransThread(act2);
        tt1.start();
        tt2.start();
        try {
            tt1.join();
            tt2.join();
        } catch (Exception je) {
            System.out.println("join exception " + je);
        }
        System.out.println("all transactions completed");
    }
}
```

9 C++ Thread Conditions (outcome 8, 2 points)

Please add code in appropriate locations so that the account balance is **never** negative. You need to modify **multiple** locations.

Answer:

```
class Account {
private:
  QMutex mutex;
  QWaitCondition cond;
  int balance;
```

```
public:
  Account() { balance = 0; }
  void deposit( int dep ) {
    mutex.lock();
    balance += dep;
    cond.wakeAll();
    mutex.unlock();
  }
  void withdraw( int draw ) {
    mutex.lock();
    while ( balance < draw ) {
      cond.wait( & mutex );
    }
    balance -= draw;
    mutex.unlock();
  }
};
```

```
#include <QtCore>
#include <cstdlib>
#include <iostream>
using namespace std;

class Account {
private:
  // >>>>>
  // add needed attributes
  // <<<<<



  int balance;
public:
  Account() { balance = 0; }
  void deposit( int dep ) {
    // >>>>>
    // add needed code
    // <<<<<


    balance += dep;



  }
  void withdraw( int draw ) {
```

```
    // >>>>>
    // add needed code
    // <<<<<



    while ( balance < draw ) {



    }



    balance -= draw;




  }
// ----------
// Do not modify anything below this line
// ----------
  void getBalance() {
    mutex.lock();
    cout << "balance: " << balance << endl;
    mutex.unlock();
  }
};
class Depositor : public QThread {
  Account * act;
public:
  Depositor (Account * a) { act = a; }
  void run() {
    int i = 0;
    while ( true ) {
      int x = (int) ( rand() % 10 );
      act -> deposit( x );
      if ( i++ % 100 == 0 )
        { act -> getBalance(); }
    }
  }
};
class Withdrawer : public QThread {
  Account * act;
```

```
public:
  Withdrawer(Account * a) { act = a; }
  void run() {
    int i = 0;
    while ( true ) {
      int x = (int) ( rand() % 100 );
      act -> withdraw( x );
      if ( i++ % 100 == 0 )
        { act -> getBalance(); }
    }
  }
};
int main()
{
  Account act;
  Depositor* depositors[5];
  Withdrawer* withdrawers[5];

  for ( int i=0; i < 5; i++ ) {
    depositors[ i ] = new Depositor(& act);
    withdrawers[ i ] = new Withdrawer(& act);
    depositors[ i ]->start();
    withdrawers[ i ]->start();
  }
  for ( int i=0; i < 5; i++ ) {
    depositors[ i ]->wait();
    withdrawers[ i ]->wait();
  }
}
```