Q1. Explain all compile-time dependencies found between the interfaces / classes of the program below: (from Topic 05 Page 6)

```
class Student {
    private String name;
    public Student(String n) {name=n;}
    public void doExercise(IReadWrite x, int ans) {
        x.writeAnswer(ans);
    }
}
```

```
class Grader {
    private String name;
    public Grader(String n) {name=n;}
    public void gradeExercise(IGrade x) {
        x.grade();
    }
}
```

```
interface IReadWrite {
    void readAnswer();
    void writeAnswer(int anAnswer);
}
```

```
interface IGrade {
    void readAnswer();
    void grade();
}
```

```
class Exercise implements IGrade, IReadWrite {
    private int studentAnswer;
    private char grade;
    private final String question;
    private final int modelAnswer;

    public Exercise(String q, int a) {
        question = q; modelAnswer=a;
    }

    public void writeAnswer(int anAnswer) {
        studentAnswer=anAnswer;
    }

    public void readAnswer() {
        System.out.println(
            "Student's answer is "+ studentAnswer);
    }

    public void grade() {
        if (studentAnswer==modelAnswer) grade='A';
        else grade='F';
    }

    public void displayResult() {
        System.out.println(
            "Student's answer is "+studentAnswer+
            ", grade is: "+grade);
    }
}
```

```
public static void main(String[] args) {
    Exercise ex = new Exercise("What is 4!", 24);
    Student m = new Student("Mary");
    Grader h = new Grader("Helena");
    m.doExercise(ex,24);
    h.gradeExercise(ex);
    ex.displayResult();
}
```

Output:
Student's answer is 24, grade is: A

Q2. Complete the method below which checks whether an array of employee contains sorted elements.

Assume that the Employee class has implemented the Comparable interface

(Topic 05 Page 7)

```
private static boolean areSorted(Employee[] arr) {









}
public static void main(String[] args) {
    Employee[] arr = new Employee[3];
    arr[0] = new Employee(..);
    arr[1] = new Employee(..);
    arr[2] = new Employee(..);
    if (areSorted(arr))
        System.out.println("Sorted");
    else
        System.out.println("Not Sorted");
}
```

Q3 Suppose we have the classes: **`Book`**, **`Member`**, **`Library`**; where the Library instance, **`lib`**, has an array of books and and array members, we need to work out the *borrow book* operation.

Compare the following approaches:

Approach 1:

```
Book b = lib.getBook(bookID);
Member m = lib.getMember(memberID);
lib.addBorrowRecord(b,m);
```

Check for problems like `b`/`m` are `null`, book has been borrowed, member quota exceeded etc..

If okay, change the book's status as borrowed, and store the loan details.

Approach 2:

```
Member m = lib.getMember(memberID);

if (m is not null) //or use try-catch
    m.borrowBook(bookID);
```

`m.borrowBook(bookID)` checks the member itself's quota.

If OK, lookup for the book b and call `b.borrowBy(m)`.

In `b.borrowBy(m)`, check whether the book is available. If OK, change its status as borrowed and store the loan details.

Exception handling may be used quite a lot.

Approach 2':

```
Book b = lib.getBook(bookID);
Member m = lib.getMember(memberID);
if (m is not null) //or use try-catch
    m.borrowBook(b):
```

Your task: choose the correct answers below:

- Approach 1 / Approach 2 is a top-down approach
- Approach 1 is poor/good OO design
- Approach 2 is poor/good OO design

Reference:

[Notes writen by Dr. W.K. Chan / CS2312 [2013-2014 Sem A]

- Executing the typical sequence of statements within a method *M* of an object may not always be compatible to the current attribute values of the same object at any time.

  In **poor** OO programming, an object O_1 checks the attribute values of another object O_2 *before* O_1 invokes the method *M* of O_2. (or else, whenever O_2 change its internal representation, we need to change the coding in O_1; otherwise, we may inject bugs into the program!)

  In **good** OO programming, an object O_1 simply invokes the method *M* of O_2, and let *M* both determine how to check the attribute values of O_2 and determine whether *M* should affect the current attribute values of O_2 (or invoke other methods of this object or some other object)

Q4 [Lab08] How to write the Fire command class

```
public class Fire extends RecordedCommand {









    @Override
    public void execute(String[] cmdParts) {









        addUndoCommand(this);
        clearRedoList();

        System.out.println("Done.");

    }

    @Override
    public void undoMe() {



        addRedoCommand(this);
    }

    @Override
    public void redoMe() {



        addUndoCommand(this);

    }
}
```

Given:

```
public class Company {
    ..

    // getInstance,
    // addEmployee, findEmployee, etc..

    public void removeEmployee(Employee e) {
        allEmployees.remove(e);
    }
}
```

```
main():
    if (cmdParts[0].equals("fire"))
        (new fire()).execute(cmdParts);
```

Q5. Advantages of Using Exceptions [Contents extracted from http://docs.oracle.com/javase/tutorial/essential/exceptions/advantages.html]

Your task: Choose the correct heading for each *Advantage* below.

Grouping and Differentiating Errors | Propagating Errors Up Call Stack | Separating Error-Handling from "Regular" Code

Advantage 1: _____

Suppose we are to read a file into memory,

the code

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

has potential errors:

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

Traditional solution:

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Is the code doing the right thing? Is the file really being closed when an error happens?

Using Exception Handling:

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Exceptions help you organize the work more effectively.

Advantage 2: _____

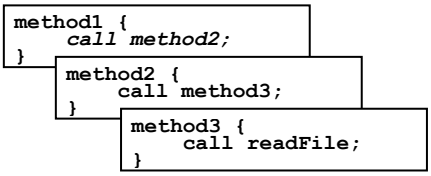
Using exception handling, it gives us a natural way group or classify exceptions. E.g. Java's IOException (most general for IO errors) and its subclasses (more specific).

- java.io.**IOException**
  - java.io.**CharConversionException**
  - java.io.**EOFException**
  - java.io.**FileNotFoundException**
  - java.io.**InterruptedIOException**
  - java.io.**ObjectStreamException**
    - java.io.**InvalidClassException**
    - java.io.**InvalidObjectException**
    - java.io.**NotActiveException**
    - java.io.**NotSerializableException**
    - java.io.**OptionalDataException**
    - java.io.**StreamCorruptedException**
    - java.io.**WriteAbortedException**
  - java.io.**SyncFailedException**
  - java.io.**UnsupportedEncodingException**
  - java.io.**UTFDataFormatException**

- java.lang.**RuntimeException**
  - java.lang.**ArithmeticException**
  - java.lang.**ArrayStoreException**
  - java.lang.**ClassCastException**
  - java.lang.**EnumConstantNotPresentException**
  - java.lang.**IllegalArgumentException**
    - java.lang.**IllegalThreadStateException**
    - java.lang.**NumberFormatException**
  - java.lang.**IllegalMonitorStateException**
  - java.lang.**IllegalStateException**
  - java.lang.**IndexOutOfBoundsException**
    - java.lang.**ArrayIndexOutOfBoundsException**
    - java.lang.**StringIndexOutOfBoundsException**
  - java.lang.**NegativeArraySizeException**
  - java.lang.**NullPointerException**
  - java.lang.**SecurityException**
  - java.lang.**TypeNotPresentException**
  - java.lang.**UnsupportedOperationException**

Advantage 3: _____

Suppose that the readFile method is the fourth method in a series of nested method calls made by the main program: method1 calls method2, which calls method3, which finally calls readFile.,



Traditional code:

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}

errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}

errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

Traditional error-notification techniques force method2 and method3 to propagate the error codes returned by readFile up the call stack until the error codes finally reach method1—the only method that is interested in them.

Using Exception Handling:

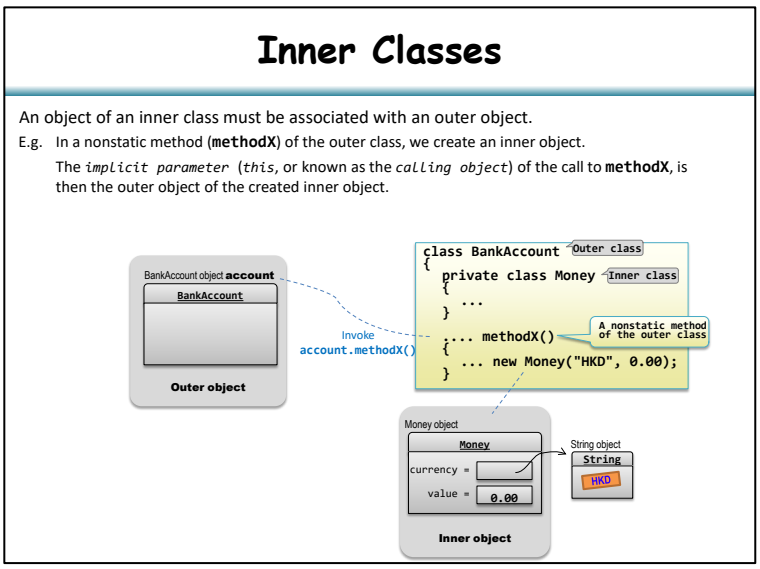
```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

A method can duck any exceptions thrown within it, thereby allowing a method farther up the call stack to catch it. Hence, only the methods that care about errors have to worry about detecting errors.

Q6. Recall Topic05 p.12



(i) What is the output of the given program? How many BankAccount and Money objects are created?

```
class BankAccount {
    private class Money {
        private String currency;
        private double value;
        public Money(String c, double b) {currency=c; value=b;}
        public String getOwner() {return owner;}
    }

    private Money balance; private String owner;

    public BankAccount(String currency, String ow) {
        owner = ow;
        balance = new Money(currency, 0.00);
    }

    public void methodX() {
        BankAccount another = new BankAccount("RMB","Tom");
        Money m1 = new Money("HKD", 88);
        Money m2 = another.new Money("JPY",123);
        System.out.printf("%.2f(%s) %.2f(%s) %.2f(%s) %.2f(%s)",
                this.balance.value, this.balance.getOwner(),
                another.balance.value, another.balance.getOwner(),
                m1.value, m1.getOwner(),
                m2.value, m2.getOwner());
    }

    //  public static void methodY() {
    //      new Money("JPY", 0.00); //Compilation error
    //  }
}
```

Error Message:

No enclosing instance of type BankAccount is accessible. Must qualify the allocation with an enclosing instance of type BankAccount (e.g. x.new A() where x is an instance of BankAccount).

```
public static void main(String[] args) {
    BankAccount account = new BankAccount("HKD", "Helena");
    account.methodX();
}
```

Totally
___ BankAccount object(s),
___ Money object(s)

Output:

____(_____) ____(_____) ____(_____) ____(_____)

(ii) Can we create an inner object in a static method?