Final Exam – Object Oriented Programming (236703)

Spring Semester 2015, Moed A

Lecturer: Eran Gilad

TAs: Nurit Moscovici, Helal Assi, Eliran Weiss

Instructions:

1. The exam contains 5 equally graded questions. You must answer all of them.
2. In case you don’t know the answer to a question or sub-question, you may write “I don’t know” and receive 20% of the points.
3. Unless specified otherwise, in questions for which you are required to provide the output of given code, the code might not compile or run to completion. In such cases you should describe the reason for the compilation or run-time error.
4. During the course you saw some specific implementations of subject matter (dynamic binding, the object model, mixins, etc.), for which there are alternative implementations. In relevant questions you may describe any implementations learned in class except for those which were given as an example of a bad solution.
5. Answer to the point. Adding irrelevant information (even if correct) may cause a deduction of points.
6. On the other hand, a mostly correct explanation to a wrong answer may be awarded a few point. You are advised to explain your answers even if not explicitly requested, but please avoid long-winded answers.
7. Make sure your handwriting is legible. Illegible answers cannot be checked. No cursive, please.
8. Answer each question on a separate page.
9. No additional material is allowed.
10. The exam contains 11 pages (including this page). Make sure your form includes all of them.
11. The duration of the exam is 3 hours.

Good Luck!
Question 1: Multiple Inheritance in C++

```cpp
struct A {
    int x;
    A() : x(1) { cout << "A" << endl; }
    A(int i) : x(2) { cout << "A int " << i << endl; }
};

struct B : virtual public A {
    int x;
    B() : A(), x(3) { cout << "B" << endl; }
    B(int i) : A(), x(4) { cout << "B int " << i << endl; }
};

struct C1 : public B {
    int x;
    C1(int i) : B(i), x(5) { cout << "C1 " << endl; }
    virtual void foo() { cout << "foo " << x << endl; }
};

struct C2 : virtual public B {
    int x;
    C2() : B(), x(6) { cout << "C2" << endl; }
    virtual void bar() { cout << "bar " << x << endl; }
};

struct D : public C1, public C2 {
    int x;
    D(): C1(7), C2(), x(8) {
        cout << "D" << endl; }
};

D* d = new D();
C1* c1 = d;
A* a = d;
```

a. What is the output of the program?

b. Explain the differences between the following C++ casts:
   - static_cast
   - dynamic_cast
   - reinterpret_cast

c. For each of the following lines of code, determine whether or not the line will compile. If not, explain.
   1. static_cast<D*>(a);
   2. dynamic_cast<D*>(a);
   3. static_cast<B*>(d);
   4. static_cast<B*>(reinterpret_cast<C2*>(d));
   5. dynamic_cast<A*>(d);

The question is continued on the following page
Continuation of Question 1

d. Does the execution of the following two lines succeed? If so, what is the output? If not, explain.

C2* c2 = dynamic_cast<C2*>(c1);
c2->bar();

e. Consider the following function:

```cpp
int offset(D* d) {
    return reinterpret_cast<int>(static_cast<A*>(d)) -
        reinterpret_cast<int>(d);
}
```

Assume that there are no errors in casting the pointer to int. Will a call to offset always return the same value? If so, what is the value? If not, explain.
Question 2: Squeak

A new feature is to be added to the Squeak language: the ability to retrieve from any class the number of instances of that class that were created. In order to do this, the following will be added to the class Behavior:

- The field `count`, which will hold the number of instances that were created
- The method `getInstanceNum`, which will return the value of `count`
- The method `initCount`, which will initialize the counter to zero

For your convenience, on the next page you will find a partial diagram of the Squeak object model.

a. How can you compute the number of classes (including metaclasses) in the system using the new feature? For simplicity, the result can be close to the actual number; without taking into account special classes that are created before initialization of the mechanism, or that aren’t part of the model as we learned it in class. Write a Squeak expression calculating this number, and explain your answer. Full points will be given only for solutions that do not use loops or recursion.

b. Explain where the `count` field should be initialized and how this should be done. For simplicity, handle only initialization for classes that are not metaclasses.

c. Explain where the `count` field should be updated and how this should be done. For simplicity, handle only initialization for classes that are not metaclasses.

It was suggested that this design is not optimal in terms of space complexity, as the counter field does not need to be declared in all of the classes. Describe the changes needed to reach the optimal implementation:

d. Where should the `count` field be defined? Specify all of the possible places, and explain your answer.

e. What further changes must be implemented for this new design? Explain.
Question 2 reference – a (partial) view of the Squeak object model

[Diagram of the Squeak object model with the following classes:
- nil
- ProtoObject
- Object
- Class
- ClassDescription
- Behavior
- Metaclass
- Metaclass class
- ProtoObject class
- Object class]
Question 3 – Exceptions and Annotations

It was suggested that a feature similar to Java’s `finally` should be added to C++. Consider the following code:

```c++
1. int* arr = new int[10];
2. try {
3.     foo(arr);
4. } catch (const exception& e) {
5.     cout << “D’oh!” << endl;
6. }
```

a. Define a class called `Finally` and use it to release the allocation of `arr` whether or not `foo` throws an exception. That is to say, it must provide functionality similar to that of `finally` in Java (You may assume that releasing memory does not cause an exception to be thrown). Describe the changes that must be made to the code above.

In the future version 1k of Java, developers wish to remove the requirement that all exceptions a function may throw must be declared explicitly. The new rules are to be similar to those in previous versions of C++:

- An exception may be declared using the keyword `throws`, but doesn’t have to be.
- Exceptions may be caught, but don’t have to be.
- If an exception was thrown in contradiction to an existing declaration, the program exits immediately, before the return from the function that threw the exception. (Assome that if no exceptions are declared then any exception may be thrown, without causing the program to immediately exit).
- The distinction between the two kinds of exceptions is eliminated, and all exceptions have the same status.

In order to allow backward compatibility, a program should be able to run even if some of its class files were compiled in Java 1k and some in older versions of Java.

b. Consider the method A.caller, that calls the method B.callee. Each of these classes was compiled separately. Decide whether each of the following situations may result in conformance issues. That is to say, is the code in A.caller “surprised” by the behavior of B.callee? If so, describe the problem in detail:

1. Class A was compiled in version 7 of Java, and class B in version 1k.
2. Class A was compiled in version 1k of Java, and class B in version 7.

The question is continued on the following page
Continuation of Question 3

As you no doubt remember, annotations have three kinds of retention policies: Source, Class and Runtime. The retention level determines whether the annotation exists only in the code itself, in the class file, or up until (and including) runtime.

c. For each of the following components of the Java exceptions mechanism, state and explain what the necessary retention type is. For example, if a component must exist during run time then the necessary retention type is Runtime. (Of course, exceptions are not based on annotations; the question is simply based on retention scopes, which are well defined for annotations):
   1. The try-catch structure
   2. The throw of the exception
   3. The throws declaration

d. The definition of retention for an annotation is achieved by using the meta-annotation Retention. What should be the retention scope of the meta-annotation Retention? Explain.
Question 4 – HW 5 (Advanced C++)

In the homework assignment you saw how it is possible to partially implement the `static_cast` function, which works during compile time, and the `dynamic_cast` function, which performs type checks during run time.

a. As you know, one of these cast types allows downcast when the source type is a virtual parent of the destination type, and the other does not allow such a downcast. Which of the casts allows this downcast? Explain why this works.

Yossi, a student in OOP, heard in a lecture that there may be problems with downcasting in the case of a complex inheritance graph. He found a function `can_cast()`, which receives as input an object of a polymorphic type. The function returns true if the object’s dynamic type does not contain "diamond inheritance", i.e. no base class in the inheritance hierarchy is inherited twice. Otherwise, the function returns false. He then wrote the following function for casting between types:

```cpp
template<typename Dst, typename Src>
Dst do_cast(Src src) {
    if (!can_cast(src))
        throw std::bad_cast();
    return (Dst)src;
}
```

You are given the following classes:

```cpp
class A { public: virtual ~A() {} };
class B : public A { public: virtual ~B() {} };
class C : public A { public: virtual ~C() {} };
class D : public B, public C { public: virtual ~D() {} };
```

You are also given the following code from Yossi’s main program:

```cpp
try {
    do_cast<D>(A());
}
catch (const std::bad_cast& e) {
    cout << "bad cast!" << endl;
}
catch (...) {
    cout << "unexpected error!" << endl;
}
```

b. Does the code compile? If so, describe the output and explain. Else, explain why not and suggest a fix for the code (in words or in code) that compiles and runs as intended.

Another student, Gil, realized that a special version of `do_cast` is needed for pointers, which will return NULL instead of throwing an exception. In order to allow this functionality Gil decided to use the ‘partial specialization’ feature in C++.

c. Explain how partial specialization can be used to implement the change suggested by Gil, and provide a code example if necessary. You may change the existing code, but must explain how the cast function is to be used in your new design.

The question is continued on the following page
d. Gil is no longer interested in using a C-Style cast for the actual conversion in his dynamic casting mechanism. Instead, he wants to implement the cast himself. What additional information must be saved for every class in order to support the cast during runtime?

In order to check whether all objects using the cast save the additional information, Gil decided that every class that uses the additional information will implement the function void hasSpecialFeature(), which will be used solely as an indicator: the function implementation may be empty. He wrote the following code to perform the check:

```cpp
template<typename T>
class FeatureChecker  {
public:
    T& obj;
    FeatureChecker(T& t) : obj(t) {}  
private:
    void CheckFeature()  {
        obj.hasSpecialFeature();
    }
};
```

Gil intends to use this function in the following fashion:

```cpp
MyClass cls;
FeatureChecker<MyClass> chkr(cls);
```

The goal is that if a type does not contain the additional information, the code will fail at compile time.

e. Will Gil's code fulfill this goal correctly? If not, explain and fix the code accordingly.
Question 5 – Java, Reflection and Advanced Features

This question appeared in the Moed A of Spring 2012

Preface: In order for a Java class to support the clone method, it must satisfy the following rules:

- The class must implement the Cloneable interface. The interface has no methods.
- The class must override the method clone() such that the method will be public and the return type will be the type of the class.
- The class must call Object.clone(), either directly or indirectly, in order to create a new instance of the class.

A class that wishes to prevent use of clone (for example, if the class inherits from a class that supports clone) must override the method and throw CloneNotSupportedException.

Answer the following:

a. The following method claims to do the same as Object.clone. Which rules does it break?

```java
public static Object myClone(Object toClone) throws CloneNotSupportedException {
    try {
        Class<?> cls = toClone.getClass();
        Object clone = cls.newInstance();
        for (Field field : cls.getDeclaredFields()) {
            field.setAccessible(true);
            field.set(clone, field.get(toClone));
        }
        return clone;
    } catch (Exception e) {
        throw new CloneNotSupportedException();
    }
}
```

b. Assume that the problems from the former sub-question were fixed (if there were problems). Is it possible to change the method so that it performs deep copy of all of the fields in an object? (You may ignore the possibility of circular referencing)

The question is continued on the following page.
Continuation of Question 5

The following class is given for sub-questions c and d. The class overrides Object’s clone:

class CanBeCloned implements Cloneable {
    public CanBeCloned clone() {
        try {
            return (CanBeCloned)super.clone();
        } catch (CloneNotSupportedException ex) {
            return null; // never happen – we support cloning!
        }
    }
}

c. The class is correct according to the language rules. For each of the following, state whether implementing it in the class will break the rules of conformance:

1. Defining the method clone as public (In Object it is defined as protected)
2. Defining the method to return CanBeCloned instead of Object.
3. Removing the declaration of CloneNotSupportedException, which is a checked exception.

d. For each of the following, state whether it might cause problems in the implementation of classes that inherit from CanBeCloned:

1. Defining the method clone as public (In Object it is defined as protected)
2. Defining the method to return CanBeCloned instead of Object.
3. Removing the declaration of CloneNotSupportedException, which is a checked exception.

The following code is given for sub-questions e and f (assume that the method in which the code exists declares that it throws CloneNotSupportedException).

Cloneable[] cloneables = getCanBeClonedArray(); // no problem here!

For (Cloneable c : cloneables) {
    Object cclone = c.clone();
}

e. Why does the code not compile?

f. The way in which Java enables support of clone is considered to be problematic. In Squeak there is a similar implementation in Object, but this is not considered problematic. Why is this?