Midterm Examination Problem Sheet

TIME: 04/15/2013, 10:20–12:20

This is a open-book exam. You can use any printed materials as your reference during the exam. Any electronic devices are not allowed.
Any form of cheating, lying or plagiarism will not be tolerated. Students can get zero scores and/or get negative scores and/or fail the class and/or be kicked out of school and/or receive other punishments for those kinds of misconducts.
Both English and Chinese (if suited) are allowed for answering the questions. We do not accept any other languages.

There are 6 problems in the exam, each worth 35 points—the full credit is 210 points. Some problems would be divided to two or three sub-problems. There is another bonus problem with 10 points. The midterm would be weighted as 2–3 homework sets, but the exact weight would depend on your performance today. Good luck!

1 Java and C

(1) (20%+15%) Describe to someone who knows C about **two** differences between C and Java. The difference can be a big one or a small one, and it is the **correctness** and **clarity** of your description that the TAs will grade on. There is no need to be overly long in your description—an ideal answer would be concise. Also, we want the differences to be "different" between the two by themselves. The first difference you list would be worth 20 points, the second would be worth 15 points.

A sample answer would look like (and no, you cannot use these two):

```
[0] Java uses a "Garbage Collector" to manage memory automatically,
    while C assumes that the programmers have the power and the
    responsibility to allocate/de-allocate memory slots on their own.
[1] C allows flexible "goto" statements, while Java does not.
```

2 Constructors

```
class A{ }
class B extends A {
  int u; C other;
  B(int u){ this.u = u; other = new C(u); }
  B(){ this(20); }
}
class C extends A {
  int v;
  C(int v){ this.v = v; }
}
class D extends B {
  double x;
  D(int u, double x){ super(u); this.x = x; }
}
```

(1) (15%) Line 1 of the code below would compile normally while line 2 results in **HAHAHA** (compile error). Why?

```
1 A ref = new A();
2 A ref = new C();
```

(2) (20%) List the order of constructors that are called after executing `new D(1, 2.0)`. You need to list all the constructors, including the constructor of `java.lang.Object` and the constructor(s) automatically added by `javac`.

3 Mutable versus Immutable

We know that `java.lang.String` is an immutable class, which makes it "safer" to use the instances of the class. The following class `Color`, however, is mutable.

(1) (20%) Rewrite `Color` such that it is immutable. Your re-written `Color` must be fully compatible with the original one in its **methods**. You can just explain the "differences" of your new class to the TAs instead of listing every line (unless you really want to).

(2) (15%) Can your re-written `Color` still be immutable if some other programmers write a `ColorChild` that extends your `Color` class? That is, can someone use a `Color`-type reference with a `ColorChild`-type instance to change some contents of the `Color` part? If so, how could you prevent such from happening? If not, why is your `Color` class safe from such?

```java
public class Color {
  public int red;
  public int green;
  public int blue;

  public Color(int _red, int _green, int _blue){
    red = _red; green = _green; blue = _blue;
  }

  public int getColor() { return ((red << 16) | (green << 8) | blue); }
  public Color invert() {
    red = 255 - red;
    green = 255 - green;
    blue = 255 - blue;
    return this;
  }
}
```

4 Lucky-tons

(1) (35%) Consider writing a class `License` that only allows at most 1126 licenses to be used at the same time. Each licenses needs to be **created when needed for the first time**, but can be re-used (after being released) afterwards. Please complete the following code that does the job.

```java
public class License{
  private static License[] pool = new License[1126];
  private static boolean[] used = new boolean[1126];

  /* (1) (10%) You need a constructor to allocate your instances,
      but at the same time preventing external code from directly
      using it. */

  /* if there is an unused license, return that;
      if not, return null */
  public static License getLicense(){
    /* (2) (15%) You need code here */
  }

  /* if there is a matching license being used, release it;
      if not, do nothing */
  public static void releaseLicense(License toRelease){
    /* (3) (10%) You need code here */
  }
}
```

5 Encapsulation

(1) (20%) Assume that the class `ntu.csie.YoungProfessor` extends class `ntu.Professor`, and the class `ntu.csie.SeniorProfessor` extends `ntu.Professor` as well. Which of the following can be invoked within an instance method of `ntu.csie.YoungProfessor`?

- (a) a call to a public method of `ntu.Professor`
- (b) a call to a protected method of `ntu.Professor`
- (c) a call to a default method of `ntu.Professor`
- (d) a call to a private method of `ntu.Professor`
- (e) a call to a public method of `ntu.csie.SeniorProfessor`
- (f) a call to a protected method of `ntu.csie.SeniorProfessor`
- (g) a call to a default method of `ntu.csie.SeniorProfessor`
- (h) a call to a private method of `ntu.csie.SeniorProfessor`
- (i) a call to a protected method of `ntu.Student`
- (j) a call to a default method of `ntu.Student`

(2) (10%) The "finalizer" we discussed in class was actually declared in `java.lang.Object` with the `protected` access modifier, and can be overridden with the finalizing steps of different classes. Which of the following classes (for simplicity, please just consider regular classes, not the array types) can call `java.lang.Object.finalize()`?

- (a) classes that are descendants of `java.lang.Object`
- (b) classes that are of package `java.lang`

(3) (5%) If your answer above include "classes that are descendants of `java.lang.Object`", then you agree that every class in Java shall be able to call the finalizer. Then, guess why the finalizer shouldn't just come with the `public` access modifier (what's the difference between using `public` and `protected` here?). If your answer above does not include "classes that are descendants of `java.lang.Object`", then describe what classes cannot call `java.lang.Object.finalize()` in their methods.

You are almost done with the exam. Only two more problems on the next page. Hang in there!

6 References and Instances

I know that you guys studied hard for Java night. What did you drink? Java coffee, of course! Let's play with Java coffee a bit.

```java
class Drink{
  public double amount;
}
class Java extends Drink{
  public double caffeine;
  public Java(double a, double c){
    amount = a;
    caffeine = c;
  }
  public Java(){ this(1.0, 2.0); }
}
class Milk extends Drink{ }
class FatlessMilk extends Milk{ public double fat; }
class BlendedJava extends Java{
  public Milk milk;
  public double ratio;
}

public class JavaDemo{
  public static void main(String[] argv){
    Milk m1 = new Milk();
    Milk m2 = new Milk();
    Java j1 = new Java(3.0, 2.0);
    BlendedJava b1 = new BlendedJava();
    BlendedJava b2 = new BlendedJava();
    b1.milk = m1;
    j1 = b1;
    b1.milk = new FatlessMilk();
    Milk[] marr = new Milk[3];
    marr[0] = b1.milk;
    marr[1] = m2;
  }
}
```

(1) (10%) Assume that you want a line of

```java
System.out.println(j1);
```

around the end of `JavaDemo.main(String[])` to print out a line of `Java coffee of caffeine ratio 2/3`. How would you do that?

(2) (25%) Illustrate the memory layout of JVM at the end of `JavaDemo.main(String[])`. You need to show all the instances and self-defined local variables that are "alive" from the stack frame of `JavaDemo.main(String[])`, their connections (references) and states (primitive-variable values).

7 `java.lang.Object`

(1) (Bonus 10%) In class, we discussed about why `java.lang.Object` shouldn't contain abstract methods — if so, there is a huge burden on programmers when designing their own classes. On the other hand, the remaining question is why `java.lang.Object` shouldn't be an abstract class. Is there really a need of constructing an instance of `java.lang.Object`? Please reason why the designers of Java may have chosen to keep `java.lang.Object` concrete.