Class 27: Exam 2

Fall 2010
UVA
David Evans

- Exam 2
- Parenthesizing the Expression

(a) (Average 3.9/5) Describe one clear advantage of Design A over Design B.
(b) (Average 4.5/5) Describe one clear advantage of Design B over Design A.

(c) (Average 7.7/10) Draw a better design than either Design A or Design B, and explain clearly why your design is better.
2(a) (4.2 / 5) Does the implementation of the Annihilator `destroy` method satisfy its specification? Your answer should either explain clearly why it does not, or use precise reasoning to argue why it does.

```java
public class Annihilator {
    // OVERVIEW: An Annihilator is a mutable object that can kill and be killed by other objects. A typical Annihilator is state where state is either Alive, Dying, or Dead.
    private int state;

    // A.F.(c): if c.state = 2, Alive; if c.state = 1, Dying; if c.state = 0, Dead.
    // RepInvariant(c): 0 <= c <= 2

    public void destroy(Annihilator a) {
        // REQUIRES: a == Dying
        // MODIFIES: a
        // EFFECTS: a_post = Dead
        a.state = a.state - 1;
    }
}
```

2(b) (3.9 / 5) Does the Obliterator `destroy` method satisfy the substitution principle? A good answer will include a clear and convincing argument supporting your answer.

```java
public class Annihilator {
    // OVERVIEW: An Annihilator is a mutable object that can kill and be killed by other objects. A typical Annihilator is state where state is either Alive, Dying, or Dead.
    public void destroy(Annihilator a) {
        // REQUIRES: a == Dying
        // MODIFIES: a
        // EFFECTS: a_post = Dead
    }
}
```

2(c) (2.5 / 5) Suppose an Obliterator is always stronger than an Annihilator, so we override the Obliterator `pickStronger` method as:

```java
public Obliterator pickStronger(Annihilator a) {
    // EFFECTS: a_pickStronger = this if a is not dead; otherwise, returns the stronger of this and a (or either one if they are equally strong).
    // where Alive is stronger than Dying which is stronger than Dead.
    if (a instanceof Obliterator) {
        if (isAlive() || (isDying() && !a.isAlive())) { return this; } else { return (Obliterator) a; } } else { return this; } return this;
}
```

Note that we have not provided a specification for the Annihilator `pickStronger` method. Could the Obliterator `pickStronger` method satisfy the substitution principle?

2(d) (challenge bonus) (+2; max +8) Note that our `pickStronger` comparisons are now not symmetric: `a.pickStronger(b)` is not necessarily equal to `b.pickStronger(a)`. Explain a general solution to this problem. For maximum bonus, your answer should include correct code for your solution and an argument why it satisfies the symmetry property, clearly stating any assumptions on which that argument relies.

```java
public Annihilator pickStronger(Annihilator a) {
    if (?) { // something that always is symmetric!
        return a.pickStronger(this);
    } else { // normal body of pickStronger
        return this;
    }
}
```

Best Idea (Jiamin Chen)

```java
public Annihilator pickStronger(Annihilator a) {
    if (?) { // something that always is symmetric!
        return a.pickStronger(this);
    } else {
        // normal body of pickStronger
    }
}
```
Almost Works

public class Annihilator {
    public Annihilator pickStronger(Annihilator a) {
        if (hashCode() < a.hashCode()) {
            return a.pickStronger(this);
        } else {
            // normal body of pickStronger
        }
    }
}

java.lang.Object public int hashCode()

Returns a hash code value for the object. This method is supported for the benefit of
hashtables such as those provided by java.util.Hashtable.

The general contract of hashCode is:
- Whenever it is invoked on the same object more than once during an execution of a Java
application, the hashCode method must consistently return the same integer, provided no
information used in equals comparisons on the object is modified. This integer need not
remain consistent from one execution of an application to another execution of the same
application.
- If two objects are equal according to the equals(Object) method, then calling the hashCode
method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(java.lang.Object)
method, then calling the hashCode method on each of the two objects must produce
distinct integer results. However, the programmer should be aware that producing distinct
integer results for unequal objects may improve the performance of hashtables.

As much as is reasonably practical, the hashCode method defined by class Object does
return distinct integers for distinct objects. (This is typically implemented by converting the
internal address of the object into an integer, but this implementation technique is not
required by the JavaTM programming language.)

3. Our philosophers from PS5 have decided that it is not natural or efficient to only
argue with one other philosopher at a time. Instead, they should be able to argue
with as many other philosophers as they want at once. The philosophers sit around a
table (represented by the Table class), and take turns making their argument. It has
the problem though, that philosophers will interrupt each other’s argument. ...

(a) (6.7 / 10) Explain how to modify the code to prevent this race condition.
Your solution to not introduce any deadlocks in the code. For this part, ignore what
happens when the philosopher wins enough points to leave the game.

public boolean philosophize () {
    say("My turn!");
    for (Philosopher p : colleagues) {
        say(p.getName() + "", you are wrong! " + quote);
        points++;
        if (points > 20) {
            ...
        }
        say("Okay, I'm done.");
        return true;
    }
}

Locks, Threads, and Objects

Every Object has an associated lock
A lock is held by a Thread
(not by an Object or Class!)

Possible Answer #1

public boolean philosophize () {
    synchronize (this) {
        say("My turn!");
        for (Philosopher p : colleagues) {
            say(p.getName() + "", you are wrong! " + quote);
            points++;
            if (points > 20) {
                ...
            }
            say("Okay, I'm done.");
            return true;
        }
    }
}

Wrong Answer #2

public boolean philosophize () {
    say("My turn!");
    for (Philosopher p : colleagues) {
        synchronize (p) {
            say(p.getName() + "", you are wrong! " + quote);
            points++;
            if (points > 20) {
                ...
            }
        }
        say("Okay, I'm done.");
        return true;
    }
}
Wrong Answer #3

```
public class Philosopher {
    private ArrayList<Philosopher> colleagues;
    ... 
    public boolean philosophize () {
        synchronize (colleagues) {
            say("My turn!");
            for (Philosopher p : colleagues) {
                say(p.getName() + ", you are wrong! " + quote);
                points++;
                if (points > 20) {
                    ... 
                }
                say("Okay, I'm done.");
                return true;
            }
        }
    }
    ...
}
```

`Is it possible to lock all the colleagues?` Left as a challenge question... solution will give +30 points on Exam 2

(Alpmost) Correct Answer

```
public class Philosopher {
    private ArrayList<Philosopher> colleagues;
    ...
    public boolean philosophize () {
        synchronize (this) {
            synchronize (table) {
                say("My turn!");
                for (Philosopher p : colleagues) {
                    say(p.getName() + ", you are wrong! " + quote);
                    points++;
                    if (points > 20) {
                        ... 
                    }
                    say("Okay, I'm done.");
                    return true;
                }
            }
        }
    }
    ...
}
```

Correct Answer?

```
public class Philosopher {
    private Table table;
    ...
    public synchronized void addColleague(Philosopher p) { ... }
    public boolean philosophize () {
        synchronize (table) {
            say("My turn!");
            for (Philosopher p : colleagues) {
                say(p.getName() + ", you are wrong! " + quote);
                points++;
                if (points > 20) {
                    ... 
                }
                say("Okay, I'm done.");
                return true;
            }
        }
    }
    ...
}
```

Table

```
// OVERVIEW: A Table is a place
//    where philosophers argue.
```

Fully Correct Answer

```
public class Philosopher {
    private Table table;
    ...
    public synchronized void addColleague(Philosopher p) { ... }
    public boolean philosophize () {
        synchronize (table) {
            synchronize (this) {
                say("My turn!");
                for (Philosopher p : colleagues) {
                    say(p.getName() + ", you are wrong! " + quote);
                    points++;
                    if (points > 20) {
                        ... 
                    }
                    say("Okay, I'm done.");
                    return true;
                }
            }
        }
    }
    ...
}
```

3(b) (6.8 / 10) Are there any deadlocks or race conditions in leaveTable? If so, explain how to fix them. If not, explain why not.

```
public void goodbye(Philosopher p) {
    say("Goodbye " + p.getName());
    colleagues.remove(p);
}
public void leaveTable() {
    for (Philosopher p : colleagues) {
        p.goodbye(this);
    }
    colleagues = null;
    table = null;
}
```

4. As mentioned in Class 17, Barbara Liskov identified four main problems with Simula that motivated the design of CLU (from Barbara Liskov, A History of CLU, 1992). For each item below, explain how well the design of Java addresses the identified problem.

   Especially good answers will use concrete examples to show how the problem she identified either still exists in Java or has been avoided by Java.

(a) (4.3 / 5) "Simula did not support encapsulation, so its classes could be used as a data abstraction mechanism only if programmers obeyed rules not enforced by the language."

Encapsulation

```
Encapsulation The hiding of implementation details so that they are inaccessible outside of the module providing the implementation. (Liskov's definition from textbook)
```

Encapsulation Packaging state with procedures. (roughly, cs1120 definition)
4(b). (3.4 / 5) Simula associated operations with objects, not with types.

**Dyadic operators:** 3 + 4
- a equals b

**Smalltalk:**
- 3.add(4)
- a.equals(b)

**Java:**
- 3 + 4 (operators for **primitive types** only)
- a.equals(b)

**CLU:**
- math$add(a, b)
- Object$equals(a, b)

---

**Java (Sort-Of) Solution**

**Static methods:**
- method associated with Class, not Object

```
static public Annihilator pickStronger(Annihilator a, Annihilator b) {
    // something that always is symmetric!
    return a.pickStronger(b);
}
```

Does this solve the problem with subtyping?

```java
Annihilator pickStronger(Annihilator a, Annihilator b) {
    a.pickStronger(a, b);
    b.pickStronger(a, b);
}
```

---

2(d) (challenge bonus) (+2; max +8) Note that our pickStronger comparisons are now not symmetric: a.pickStronger(b) is not necessarily equal to b.pickStronger(a).

Explain a general solution to this problem. For maximum bonus, your answer should include correct code for your solution and an argument why it satisfies the symmetry property, clearly stating any assumptions on which that argument relies.

Note: general solution means it needs to work for **all possible subtypes**!

```
public Annihilator pickStronger(Annihilator a) {
    if (?) { // something that always is symmetric!
        return a.pickStronger(this);
    } else {
        ... // normal body of pickStronger
    }
}
```

---

4(c). (3.7 / 5) Simula "treated built-in and user-defined types non-uniformly. Objects of user-defined types had to reside in the heap, but objects of built-in type could be in either the stack or the heap."

Java still treats primitive types and Object types differently!

---

**Exam Recap**

- Exam questions meant to test:
  - Do you understand **specifications** and how to reason about procedures and data types
  - Do you understand **subtyping** and **inheritance**
  - Do you understand **concurrency** (locks, race conditions, deadlocks)
  - Do you understand tradeoffs between **expressiveness** and **truthiness**

Your oral final exam will give you one last chance to convince me the answer to all these questions is “Yes”. Some of the questions will probably be based directly on things from this exam.

---

**Parenthesizing Question**

Given an arithmetic expression involving addition, subtraction, and multiplication of natural numbers, add parentheses to maximize the **value** of the expression.