Concepts of Object-Oriented Programming
AS 2022

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 9

Information Hiding, Encapsulation, and Aliasing

December 2, 2022

Task 1

Suppose that we have a language with the information hiding rules of Java, but with structural subtyping. What should the subtyping relations between the following three classes be?

```
class A { int foo(); }
class B { protected int foo(); }
class C { public int foo(); }
```

Task 2

From a previous exam

Consider the following Java program consisting of two packages:

```
package A;

public abstract class Person {
    _______ int tickets = 0;
    _______ final int maxTickets = 3;

    _______ abstract void buy(int t);
}

public class Buyer extends Person {
    _______ void inc(int t) {
        if (this.tickets + t <= this.maxTickets) this.tickets += t;
    }
    _______ void buy(int t) { if (t >= 0) inc(t); }
}



package B;
import A.*;

public class SmartBuyer extends Buyer {
    _______ void inc(int t) { this.tickets += t; }
}

public class Main {
    public static void main(String args[]) {
        Buyer b = new SmartBuyer();
        b.buy(9);
    }
}
```

Provide the *most restrictive* access modifiers for the fields `tickets` and `maxTickets` and the methods `inc()` and `buy()` such that the program is still accepted by the compiler.

Task 3

Consider the following Java programs:

Program 1

```
package A1;
public class X {
 int x;
}
```

```
package A2;
import A1.X;
class Y extends X
{
   int f(X v) {
      return v.x;
   }
}
```

Program 2

```
package A1;
public class X {
 protected int x;
}
```

```
package A2;
import A1.X;
class Y extends X
{
   int f(X v) {
      return v.x;
   }
}
```

Program 3

```
package A1;
public class X {
 private int x;
}
```

```
package A2;
import A1.X;
class Y extends X
{
   int f(X v) {
      return v.x;
   }
}
```

Program 4

```
package A1;
public class X {
 protected int x;
}
```

```
package A2;
import A1.X;
class Y extends X
{
  int f() {
   return this.x;
  }
}
```

Only one of these programs compiles. Which one? Why are the other programs rejected?

You can refer to the Java Language Specification rule 6.6.2.1 for more detailed information about the `protected` access modifier.

Task 4

Suppose that the following Java classes are part of a package, to which an external user cannot add classes.

```
public abstract class BankAccount {
   ... boolean importantCustomer = false;
   ... int amount = 0;
   ... final int maxDebit = 1000;

   /// invariant amount >= -maxDebit &&
   ///   !importantCustomer => amount >= 0 &&
   ///   importantCustomer <=> this instanceof RichCustomer

   ... void deposit(int amount);
   ... void withdraw(int amount);
}

public final class PoorCustomer extends BankAccount {
   ... void deposit(int amount) {
      if(amount >= 0)
         this.amount += amount;
   }
   ... void withdraw(int amount) {
      if(amount <= this.amount)
         this.amount -= amount;
   }
}

public final class RichCustomer extends BankAccount {
   public RichCustomer() { importantCustomer = true; }
   ... void deposit(int amount) {
      if(this.amount + amount >= -maxDebit)
         this.amount += amount;
   }
   ... void withdraw(int amount) {
         if(-maxDebit <= this.amount - amount)
            this.amount -= amount;
   }
}
```

Provide the most permissive access modifiers for each field and method, such that the class invariant cannot be broken from outside the package. Assume that no integer over/underflow occurs.

In Scala, a class can be declared as sealed. That means that the class can be extended only by classes written in the same `.scala` file. Suppose that the class `BankAccount` is declared as sealed, and `PoorCustomer` and `RichCustomer` are part of the same `.scala` file. Does this allow you to choose more permissive access modifiers? Note that `PoorCustomer` and `RichCustomer` are still declared as final.

Task 5

Consider the following Java code:

```java
package p;

public final class List {
   /// invariant 1: The list starting at head is acyclic
   /// invariant 2: The list starting at head is non-decreasing

   public void prepend(int x){
      if (head == null || x <= head.getValue())
         head = new Node(x, head);
   }

   public Node getHead() { return head; }
   public Node head = null;
}

public final class Node {
   private Node next;
   private int value;

   Node(int x, Node n) {
      value = x;
      next = n;
   }

   public Node getNext() { return next; }
   public int getValue() { return value; }
}
```

Assuming that we cannot modify the classes `List` and `Node`, we would like to see whether or not the invariants can be broken, either by adding classes to package `p`, or by clients outside of package `p`. Assume reflection is not used at all.

A) Can invariant 1 be broken by adding clients outside of package `p`? If yes, show code that when run, ends in a state in which the invariant is broken; if not, explain why.

B) Can invariant 1 be broken by adding classes to package `p`? If yes, show code that when run, ends in a state in which the invariant is broken; if not, explain why.

C) Can invariant 2 be broken by adding clients outside of package `p`? If yes, show code that when run, ends in a state in which the invariant is broken; if not, explain why.

D) Can invariant 2 be broken by adding classes to package `p`? If yes, show code that when run, ends in a state in which the invariant is broken; if not, explain why.

Task 6

Consider the following Java code:

```
public class Hour {
   public int h = 0;
}

public class Time {
   private Hour hour = new Hour();
   /// invariant hour.h >= 0 && hour.h < 24

   public void setHour(int h) {
      if (h >= 0 && h < 24) this.hour.h = h;
   }

   public Hour getHour() { return hour; }
}
```

A) Provide an example that breaks the invariant of `Time` without changing the code above and without using reflection.

B) There are two immediate ways to fix the problem. In one of them, signatures of methods are modified, while in the other they are not. What are these ways of fixing the problem?

C) Clearly, we would prefer to keep the signatures the same as before. Are there any drawbacks to this approach?

D) Would it be possible to introduce an interface with no mutator methods and use it to solve the problem? Explain how this approach would look and whether there would still be a way to break the invariant.

Task 7

The following Java classes, all part of the `security` package, were written by an unexperienced programmer and contain a number of issues:

```
package security;

public class User {
   public String name;
   public String password;
   public User(String name, String password) {
      this.name = name;
      this.password = password;
   }
}

public class LoginException extends RuntimeException {
   public User problemUser;
   public LoginException(String message, User problemUser) {
      super(message);
      this.problemUser = problemUser;
   }
}

public class Login {
   private List<User> users = new LinkedList<User>();
   public void registerUser(User u) {
      if (u == null || u.name == null || u.password == null
         || u.name.isEmpty() || u.password.isEmpty()) return;
      users.add(u);
   }

   // Returns true if the user ’u’ was successfully logged in.
   // Otherwise returns false or throws an exception.
   public boolean login(User u) throws LoginException {
      if (u == null) return false;
      User current = null;
      try {
         for(User registered : users) {
            boolean nameEqual = registered.name.equals(u.name);
            current = registered;

            if (nameEqual) {
               if (registered.password.equals(u.password))
                  return true;
            }

            if (nameEqual)
               throw new LoginException("Invalid password for user", u);
         }

         return false;
      }
      catch(Exception e) {
         throw new LoginException("Invalid user", current);
      }
   }
}
```

The `malicious` method is in a different package:

```
void malicious(Login l) { ... }
```

Assume the `Login` object that is passed into the method already has registered users.

A) Complete the body of the `malicious` method so that you manage to log-in as an already existing user. You do not know any names or passwords of existing users. Do not use reflection. You are not allowed to call `login` more than a constant number of times.

B) Is it possible to fix the problem under the following restrictions? In each of these cases, explain how you can prevent the malicious login or why it is not possible.

- only modifying the `User` class?
- only modifying the `LoginException` class?
- only modifying the `registerUser` method?
- only modifying the body of the `for` loop inside the `login` method?

Task 8

Data structures often intentionally share aliases. For instance, consider the following Java class:

```
class ArrayList<T> {
   private T[] elements = ...;
   private int lastEl = 0;
   public T get(int i) { return elements[i]; }
   public int size() { return lastEl; }
   public void add(T el) { elements[lastEl++] = el; }
}
```

Imagine that this class is extended as follows

```
class Coordinates {
   int x, y;
   public Coordinates(int xx, int yy) { x = xx; y = yy; }
}

class CList extends ArrayList<Coordinates> {
   /// invariant ∀ i:int | 0 ≤ i ∧ i < size() ⇒ get(i).x > get(i).y
   public void add(Coordinates el) {
      if (el.x > el.y) super.add(el);
   }
}
```

A) Write a program that breaks the invariant of `CList`.

B) How can we fix this problem?

C) Is it possible to fix it without allocating new objects (either directly or indirectly), that is, without consuming additional memory? What new problems might arise from your changes?

D) Discuss the benefits and the drawbacks of using alias sharing in data structures.