Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 9

Information hiding, encapsulation and object structures

November 21, 2014

Task 1

Java allows an object of a class `C` to access the private fields of other objects declared in `C`. Discuss the resulting level of information hiding, its advantages and limitations, and provide some examples.

What is the policy concerning the visibility of protected fields of other objects?

solution

Limitation: we cannot check the consistency of an object considering only the current instance.

```
public class Foo {
   private int a=0; /// invariant a>=0
   public Foo broken() {
      Foo result=new Foo();
      result.a=-1;
      return result;
   }
}
```

Advantage: we can access the internal structure of other objects of the same class. Note that we already know their internal structure, since it is exactly the same as that of the current object.

```
public class List {
   private Object el;
   private List next;

   public removeSecondElement() {this.next=this.next.next;}
}
```

Similarly to private fields, an object can access protected fields of other objects of the same type. However there are some restrictions. Consider the code below:

```
public class A {
   protected int x;
}
public class B extends A {
   public void foo() { this.x = 2; } // OK
   public void bar(B b) { b.x = 2; } // OK
   public void foobar(A a) { a.x = 2; } // Does not compile
}
```

The `foo` method compiles without errors - an object has access to its own protected fields. `bar` also compiles - an object has access to private and protected fields of other objects of

the *same* type. However, as we see from `foobar`, an object does not have access to the protected fields of another object of a supertype. If this were allowed it would be possible to break the internal state of an arbitrary class by simply inheriting from it:

```
public class A {
   protected int x;
}
public final class B extends A {
   /// invariant x==0
   public B() { this.x = 0 };
}

public class MaliciousClass extends A {
   public void foo(A a) {a.x = 42;}
   public void evil() {
      B b = new B();
      foo(b);
      // The invariant of b would now be broken if
      // method foo could be compiled.
   }
}
```

Note that in the examples above we assume that all classes are in different packages.

Task 2

Consider the following Java code:

```
file A.java:
package p;
public class A {}

file B.java:
package p;
class B extends A {public int x;}

file C.java:
package p;
public class C extends B {          //1
  public B get(){...}               //2
  public void set(B d){...}         //3
}

file client.java:
package client;
import p.*;
class Client
  void f(){
    C c = new C();

    c.set(c.get());     //4
    c.set(c);           //5

    A a = c.get();      //6
    a = c;              //7
    c.x=1;              //8
  }
  void g(C c){
    c.get().x=1;        //9
  }
```

The code of `f` is accepted by the Java compiler, while `g` is rejected.

A) What seems inconsistent with information hiding in this code? Which lines would you expect the compiler to reject? Could allowing this still be useful in some cases?

solution

The code has a package access class `B` in the public interface. We would expect the compiler to complain about that. This could, however, prove useful, for example (disregarding the inheritance) if `C` was a `LinkedList` class and `B` a `ListNode` - we might want to let the client get Nodes, but not be able to do anything with them except pass them directly back to the list (e.g. as a search result used to remove an element). Also, we might use class `B` as an abstract implementation class inherited by several public classes, that is not exposed to the client. We would expect the compiler to reject all numbered lines except numbers 1, and perhaps also number 7 - as explained in the last section.

B) Does this problem exist in C++? C#? Scala? If not explain which line the respective compiler would complain about.

solution

- C++ does not have access modifiers for classes, and private inheritance does not allow upcasting, so this problem does not directly exist.

  However, if we look at inner classes, C++ behaves likes Java except that the method `g` is accepted - this lets us access a public field of a private class. Not that line 8 accesses a public field of a public class - class `C` exposes `x` to clients.

- C# requires all types mentioned in a method signature to have at least the visibility of the method (for access modifiers for classes they would have to be inner classes, as namespaces only support public members), and similarly for inheritance - so the problem is prevented

- Scala allows the `set` method but not the `get` method and not the inheritance - so `set` would only be callable from inside the package (or with `null`)- not very useful

C) Suggest a rule the Java compiler could enforce in order to be more consistent with information hiding - the rule should reject all marked lines.

solution

We could require that:

- For inheritance, the superclass (and interfaces) must be at least as visible as the inheriting class
- For method signatures, all types mentioned in the signature must be at least as visible as the method being declared

D) Now suppose we wanted to support also lines 1 and 7 (could you suggest why this might be useful?) how can we relax our new rule to allow this?

solution

We could relax the inheritance rule, and allow an annotation of the sort `C extends* A` that would mean to the client that `C` can be assigned to `A` but does not mention `B` and does not state the length of the inheritance path. This could be useful, for example, if `B` was an abstract private class with some implemented methods, inherited by several public classes (that should be subtypes of `A`) that use its method implementations, but we didn't want to expose `B` as it is an implementation detail.

Task 3

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.

solution

The body of the `malicious` method could look like:

```
void malicious(Login l) {
    User u = new User("user","pass");
    l.registerUser(u);
    u.name = null;

    try {
        l.login(u);
    }
    catch(LoginException e) {
        boolean success = l.login(e.problemUser);
        //Logged in as the user that was registered before user u
    }
}
```

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?

solution

- We could make both fields of `User` have the default (package) access:

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

Therefore, code outside the package will not be able to change existing `User` objects and the `malicious` method could not cause the exception as before.

- only modifying the `LoginException` class?

solution

The `LoginException` class currently captures the value of the problematic user. Instead it could create a new user that has the same name as `problemUser` but hides

the password.

```
public class LoginException extends RuntimeException {
   public User problemUser;
   public LoginException(String message, User problemUser) {
      super(message);
      this.problemUser = new User(problemUser.name, "****");
    }
}
```

This way, even if an exception is thrown, that refers to the wrong user name, the user's password will not be leaked.

- only modifying the `registerUser` method?

solution

We can change the `registerUser` method so that it does not capture its argument:

```
public void registerUser(User u) {
   if (u == null || u.name == null || u.password == null
         || u.name.isEmpty() || u.password.isEmpty()) return;
   users.add(new User(u.name, u.password));
}
```

Now we would not be able to modify the internal structure of the `Login` class by modifying the user we just registered in the `malicious` method.

- only modifying the body of the `for` loop inside the `login` method?

solution

This for loop actually contains a bug which allows the exploit to work. To fix it we must move the assignment to the `current` variable to the beginning of the loop:

```
for(User registered : users) {
   current = registered;
   boolean nameEqual = registered.name.equals(u.name);

   ...
}
```

In the original code we were able to cause an exception regarding a particular user, but report the previous user as an invalid, since `current` was not updated yet. This is no longer the case.

Task 4

In answering this task, do not use reflection, inheritance, and static fields or methods.

This task is concerned with reasoning about *non-modification* in a modular setting in the presence of aliasing.

Consider the following code:

```
package cell;
class Cell {
   ///ensures get()==newValue
   public Cell(int newValue){value=newValue;}

   ///ensures get()==newValue
```

```
    public void set(int newValue){value=newValue;}
    ///pure
    public int get(){return value;}
    private int value;
}

package client;
import cell.*;
class Client{
    ///requires c1!=null
    ///requires c2!=null
    void setCells(Cell c1, Cell c2) {
        c1.set(1);
        c2.set(2);
        assert(c1.get() == 1);
    }

    void setCellsClient() {
        Cell c1 = new Cell(5);
        Cell c2 = new Cell(5);
        setCells(c1,c2);
    }
}
```

The objective of this task is to make sure that the assertion in the method `setCells` does not fail, using modular reasoning. The potential problem is that of determining whether the call `c2.set(2)` can affect the return value of `c1.get()`.

A) Modify the second line in method `setCellsClient` (the initialization of `c2`) so that the assertion in method `setCells` fails. The precondition of `setCells` must still be satisfied by the modified version.

solution

```
void setCellsClient() {
    Cell c1 = new Cell(5);
    Cell c2 = c1;
    setCells(c1,c2);
}
```

B) Add a precondition to `setCells` that will make the call from your version of `setCellsClient` illegal. The precondition should be such that the original call is legal. Remember that the precondition can only refer to the arguments of the method and to public fields and methods.

solution

```
///requires c1!=c2;
void setCells(Cell c1, Cell c2)
...
```

C) We now add a `clone` method to the `Cell` class:

```
    ///ensures result != null
    ///ensures result != this
    ///ensures result.get()==get()
    ///ensures get()==old(get())
    public Cell clone() { return new Cell(value); }
```

We also add to the client the methods `left` and `right`, which use the `clone` method:

```
void left() {
   Cell c1 = new Cell(5);
   Cell c2 = c1.clone();
   setCells(c1,c2);
}
```

```
void right() {
   Cell c1 = new Cell(5);
   Cell c2a = new Cell(5);
   Cell c2 = c2a.clone();
   setCells(c1,c2);
}
```

Modify only the `Cell` class so that a call to `left` causes the assertion in `setCells` to fail, while a call to `right` does not cause the assertion to fail. You can add private and default access members and methods to the `Cell` class and add private classes to the `cell` package, and also modify the implementation of existing methods, but not change the public interface in any way. Your implementations must satisfy the existing contracts, including the one from task B.

solution

```
package cell;
class Cell{
   ///ensures get()==newValue
   public Cell(int newValue){value = new CellInt(newValue);}

   ///ensures result != null
   ///ensures result != this
   ///ensures result.get()==get()
   ///ensures get()==old(get())
   public Cell clone(){return new Cell(value);}

   ///ensures get()==newValue
   public void set(int newValue){value.set(newValue);}
   ///pure
   public int get(){return value.get();};

   private Cell(CellInt ci){value = ci;}

   private CellInt value;
}

private class CellInt{
   CellInt(int newValue){ value = newValue;}
   int get(){ return value; }
   void set(int newValue){ value = newValue; }
   private int value;
}
```

The `clone` method now creates a new `Cell` that shares the representation (the `CellInt`), and so modifying the cloned or original `Cell` also modifies the other.

D) Strengthen the precondition of the method `setCells` so that, with your modified `Cell`, the call from `left` would fail the precondition check, while the call from the method `right` would satisfy the precondition.

You can use the concept of the reach of an object, where, for an object `x`, `reach(x)` is defined as the the set of objects which are reachable from `x` — the set of objects which can be described by an access path `x.f1.f2. ... .fn` for some n and some sequence of field names `f1..fn` (we do not consider arrays in this task). All fields are considered, regardless whether they are public or private. You can also use set operations in your precondition.

Remember that the precondition of a method can refer only to the `this` object and the method's arguments, dereferencing of public fields, and call public pure methods.

solution

```
///requires reach(c1) disjoint reach(c2);
void setCells(Cell c1, Cell c2)
...
```

Now the reach of the arguments `c1` and `c2` are disjoint, so modifying one cannot affect the other in any way.

E) In order to prove the correctness of the body of the methods `left` and `right`, when `setCells` has the stronger precondition from section D, we would have to strengthen the postcondition of the `clone` method of class `Cell`. Write a stronger postcondition to the method `Cell.clone` so that the bodies of the methods `left` and `right` can be proven modularly — i.e., without knowing the implementation of the `clone` method and other private details of the class `Cell`.

solution

```
///ensures result != null
///ensures reach(result) disjoint reach(this)
///ensures result.get()==get()
///ensures get()==old(get())
public Cell clone(){return new Cell(value);}
```