Concepts of Object-Oriented Programming
AS 2018

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 9

Parametric Polymorphism and Information Hiding

November 23, 2018

Task 1

Consider the following Java method:

```java
String concatenate(List<?> list) {
   String  result="";
   String separator="";
   if(list instanceof List<String>) {
      result="String:";
      separator=" ";
   }
   else if(list instanceof List<Integer>) {
      result="Integers:";
      separator="+";
   }
   for(Object el : list)
      result=result+separator+el.toString();
   return result;
}
```

A) This program is rejected by the Java compiler. Why?

solution

The Oracle and the Open JDK compilers both produce these short errors:

```
illegal generic type for instanceof
illegal generic type for instanceof
```

The Eclipse compiler tries to be more helpful:

```
Cannot perform instanceof check against parameterized type
List<String>. Use the form List<?> instead since further
generic type information will be erased at runtime

Cannot perform instanceof check against parameterized type
List<Integer>. Use the form List<?> instead since further
generic type information will be erased at runtime
```

This happens because of type erasure in Java.

B) Using the advice given by the Eclipse Java compiler (replace `List<...>` with `List<?>`), rewrite and compile the program. What are the results of executing the method passing each of the following:

- A list of strings containing only one element "word"?
- A list of Integers containing only one element Integer(1)?
- A list of Objects containing only one element (initialized by new Object())?

solution

First of all, we follow the output of the compiler, and so we rewrite the method to:

```
String concatenate(List<?> list) {
   String  result="";
   String separator="";
   if(list instanceof List<?>) {
      result="String:";
      separator=" ";
   }
   else if(list instanceof List<?>) {
      result="Integers:";
      separator="+";
   }
   for(Object el : list)
      result=result+separator+el.toString();
   return result;
}
```

The Java compiler will compile this program without any warning. The output of the method is obviously:

```
String: word
String: 1
String: java.lang.Object@3e25a5
```

C) Is this behaviour consistent with what you would expect from the initial program? If not, how can you fix it?

solution

No, in the original program we expected:

```
String: word
Integers:+1
java.lang.Object@3e25a5
```

We can fix it in the following way:

```
String concatenate(List<?> list) {
   String  result="";
   String separator="";
   if(list.size() >= 1 )
      if(list.get(0) instanceof String) {
         result="Strings:";
         separator=" ";
      }
      else if(list.get(0) instanceof Integer) {
            result="Integers:";
            separator="+";
        }
    for(Object el : list)
        result=result+separator+el.toString();
    return result;
}
```

But this requires to have at least one element in the list. Moreover, there is no guarantee that if the first element is, for example, a string, that this is not a list of Objects.

D) What would happen if you tried to implement the different cases using method overloading instead of just one method. Why is this the case?

solution

If we introduce separate methods which differ only by the generic types of their arguments, we get compile-time errors such as:

```
Method concatenate(List<? extends Object>) has the same
erasure concatenate(List<E>) as another method in type C
```

This restriction is imposed to ensure that when choosing which of the overloaded method definitions to call, we always have a "best fit". Java class files do however include generic versions of the method signatures in the class (to enable separate compilation and type-checking of generic code). For this reason, it might seem surprising that we cannot disambiguate between these different overloaded methods, since at compile-time the type information is all available. However, Java also supports raw types - versions of generic classes in which no type parameter is provided (e.g., `List` for a `List<X>` class). These are supported for backwards compatibility with pre-generics Java code. For this reason, we need to consider the possibility that a client calling our method provides an argument of raw type `List`. In this case, we would not be able to choose between our different method overloads.

E) What happens if you compile and execute the initial program in C# ? Why?

Assume that we replace the wildcard by a method type parameter `T` to make it work in C#.

solution

The program is compiled and we obtain the expected results ("String: word", "Integers:+1", "..."), since in C# there is no type erasure and the information about generics is preserved at runtime.

Task 2

A C++ template class can inherit from its template argument:

```
template <typename T>
class SomeClass : public T { ... }
```

A) Using this technique and given the following class definition

```
class Cell {
public:
   virtual void setVal(int x) { x_ = x; }
   virtual int value() { return x_; }
private:
   int x_;
}
```

write two template classes that can be used as “mixins” for class `Cell`

- `Doubling` - doubles the value stored in the cell.
- `Counting` - counts the number of times the value of the cell was read.

Do not use multiple inheritance. It should be possible to use the classes like this:

```
auto c = new Doubling<Counting<Cell>>(); // instantiation
c->setVal(5);
c->value(); // returns 10
c->numRead(); // returns 1
```

solution

```
template <typename T>
class Doubling : public T {
public:
    virtual void setVal(int x) override {
        T::setVal(x * 2);
    }
}

template <typename T>
class Counting : public T {
public:
    virtual int value() override {
        ++numRead_;
        return T::value();
    }
    int numRead() { return numRead_; }
private:
    int numRead_;
}
```

B) Describe how the instantiation above will look like.

solution

When the mix-ins are instantiated the following two classes will be generated:

```
class Counting$_{Cell}$ : public Cell {
public:
    virtual int value() override {
        ++numRead_;
        return Cell::value();
    }
    int numRead() { return numRead_; }
private:
    int numRead_;
}

class Doubling$_{Counting_{Cell}}$ : public Counting$_{Cell}$ {
public:
    virtual void setVal(int x) override {
        Counting$_{Cell}$::setVal(x * 2);
    }
}
```

C) How does this concept of mixins in C++ differ from Scala traits?

solution

While this concept is similar to Scala traits there are some notable differences. In Scala it is possible to mix any number of traits in a class and use this in any location of the code that requires the same class and a subset of the traits:

```
var x = new X with A with B with C with D
var x1: (X) = x // OK
var x2: (X with A) = x // OK
var x3: (X with B) = x // OK
var x4: (X with A with D with C) = x // OK
```

Using the proposed solution in C++ however is more restrictive, as there is no way to refer to the class `X` with arbitrary mix-ins:

```
auto x = new D<C<B<A<X>>>>();
X* x1 = x; // OK
A<X>* x2 = x; // OK
B<X>* x3 = x; // Does not compile
C<D<A<X>>>* x4 = x; // Does not compile
```

This is particularly important for traits that introduce new methods like `Counting.numRead()` since any client code that uses this new behavior would have to know exactly how the trait was mixed-in.

Another problem of the C++ solution is object construction. If the base class does not have a default constructor then the mix-ins should know to call the correct constructor and provide appropriate parameters. An alternative here is for the mixin to just inherit the base class constructors: `using T::T;` which will allow clients of the mixin to use all constructor available in the base class. This works fine if the state of the mixin can be initialized with default values.

A further difference to Scala is that in the C++ solution it is possible to include the same “trait” more than once:

```
auto x = new Doubling<Doubling<X>>();
x->setVal(5);
x->value(); // returns 20
```

An advantage of the C++ solution is that we do not need to declare the base class that the mix-ins extend. Thus it is possible to use them with different base classes as long they have matching virtual methods.

Task 3

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.

solution

For the fields of class `BankAccount`, the most permissive access modifiers are:

`importantCustomer`: default modifier. In this way, it would be accessible by other classes in the same package but not by subclasses. Otherwise, we may have a class that extends `BankAccount` and sets to true `importantCustomer` without being a `RichCustomer`.

`maxDebit`: public, since it is final and it cannot be modified by other classes.

`amount`: default, since we need to access it from the other classes of this package (e.g. `PoorCustomer` and `RichCustomer`), but we must prevent external attackers from modifying it.

Methods `withdraw` and `deposit` can be declared public, since they preserve the invariants.

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?

solution

If class `BankAccount` had been declared as sealed, we could choose protected as the access modifier of the `amount` and `importantCustomer` fields, since external classes would not be allowed to extend it and so would not be able to gain access to these fields. More generally, if a class is sealed, the default and protected levels are equivalent, since it is not possible to extend the current class outside the current package.

Task 4

From a previous exam

Consider the following Java program consisting of two packages:

```java
package A;

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

        /// invariant 0 <= tickets <= maxTickets

    ________ 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);
    }
}
```

A) 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.

solution

The field `tickets` must be `protected` (since we need to access it from the class `SmartBuyer` which belongs to another package). The field `maxTickets` must have a `default` access modifier (because we need to access it from the class `Buyer` which belongs to the same package). The method `inc()` can be declared `private` in both `Buyer` and `SmartBuyer`. The method `buy()` in class `Person` must have a `default` access modifier (because abstract methods cannot be private), while the method `buy()` in class `Buyer` must be `public` (because we need to access it from the class `Main` which belongs to another package and is not a subclass of `Buyer`).

B) With respect to the access modifiers that you provided in part **A**, add code to the class `SmartBuyer` such that the execution of the `main()` method of the class `Main` breaks the invariant of the class `Person`.

solution

One possible solution consists in overriding the method `buy()` in `SmartBuyer` as follows:

```
public void buy(int t) { inc(t); }
```

Alternatively, the implementation of the new `buy()` method can also directly modify `this.tickets` in a way that breaks the invariant.

Task 5

Consider the following Java code:

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

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

solution

Invariant 1 cannot be broken by clients outside `p` because the field `Node.next` is private and can only be set in the constructor to an argument of the constructor, which must point to an already existing list that does not include the object currently being created.

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.

solution

Invariant 1 cannot be broken from inside `p` for the same reasons as above.

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.

solution

Invariant 2 cannot be broken from outside `p` because:
The invariant depends only on the fields `Node.next`, `Node.value`, and `List.head`.
Both `Node` fields are only written to in the constructor of `Node` and cannot be modified later as they are private.
The constructor of `Node` is of package access and so cannot be called directly by the client. The only public method that calls it is `List.prepend`, which ensures invariant 2 - hence no decreasing list of nodes (whether or not attached to a `List`) can be created by clients of the package. So, although we can assign `List.head` any value, we cannot obtain a value (a `Node`) that would break the invariant.

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.

solution

Invariant 2 can be broken as follows (all code inside `p`):

```
class Client{
   void client(){
      List l = new List();
      l.prepend(0);
      Node n = new Node(1,l.getHead());
      l.head = n;
   }
}
```

Task 6

Consider the following Java code:

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

public class Time {
   private Hour hour=new Hour();
   private int m=0;
   /// 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.

solution

We can easily break the invariants through alias leaking. For instance, the following code breaks the invariant of class `Time`:

```
Time t=new Time();
Hour h=t.getHour();
h.h=-1;
```

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?

solution

We can fix this in two ways. We have to avoid the alias leaking. We can reach this goal returning an integer value instead of an object, or a copy of the `Hour` object stored in the current `Time` object.

```
public int getHour() {return hour.h;}
public Hour getHour() {return (Hour) hour.clone();}
```

In general, it is simpler for reasoning, if possible, to return only primitive values, or to avoid exposing aliases of the local state of the object, by instead returning copies of the stored objects. In this way, we can avoid alias leaking, thus no external code can modify the values contained in the current object.

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

solution

The drawback of the second approach is that we are creating a new object and thus are using more memory. Additionally, client code that uses reference equality to check if the `Hour` object returned by `getHour()` is equal to another `Hour` object breaks if `getHour()` returns a new object on every call.

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.

solution

We could hide the `h` field of `Hour` by making `Hour` implement an interface `IMHour` that has no mutator methods. `Time.getHour()` could then return this interface.

The client could still downcast from `IMHour` to `Hour` and break the invariant but aside from that the invariant is protected. This could be prevented by making `Hour` a private inner class of `IMHour`.

Task 7

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.

solution

Here is a recap of the meaning of the Java access modifiers:

- `public`: every class can access the element
- `protected`: only subclasses and classes in the same package can access the element
- *default*: only classes in the same package can access the element
- `private`: only this class can access the element

The detailed semantics of the `protected` modifier are available in the Java Language Specification linked above.

Explanation:

- *Program 1* does not compile because method `f` of class `Y` tries to access a field of the superclass with default access modifier (that is, it can be accessed only by classes in the same package) from an external package.
- *Program 2* does not compile because method `f` of class `Y` tries to access a protected field of an object instance of the superclass, but from a different package (`A2`, while the superclass belongs to `A1`). Note that Java does not allow subclasses to access protected fields of other objects instance of the superclass if they belong to a different package.

  In order to make this program compile and run, we could define class `X` and class `Y` in the same package. Alternatively, if the parameter `v` was of type `Y` (or any subclass of it, defined in any package), the program would also be accepted.
- *Program 3* does not compile because method `f` of class `Y` tries to access a private field of the superclass.
- *Program 4* compiles. In fact, method `f` of class `Y` is allowed to access `this.x` since it is a protected field of class `X`.