Concepts of Object-Oriented Programming
AS 2020

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 11

Type Erasure, Templates and Information Hiding

December 4, 2020

Task 1

Consider the following Java method:

```
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 try to 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. Therefore, an improved solution would be to iterate over all the elements of the list and to compute their smallest common supertype.

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

Consider the following Java program, which compiles correctly and makes use of generics:

```
class Animal {}
class Mammal extends Animal {}
class Tiger extends Mammal {}

class Ship<T extends Animal> {
  public T content;
```

```
 7 }
 8
 9 class Cage<T extends Mammal> {
10    public T content;
11
12    void takeFromShip(Ship<T> other) {
13       this.content = other.content;
14       other.content = null;
15    }
16 }
17
18 class Zoo<T extends Mammal> {
19    void swapTigers(Ship<T> mammalShip, Cage<Tiger> tigerCage) {
20       Tiger tiger = tigerCage.content;
21       Cage<Tiger> tmpCage = new Cage<Tiger>();
22       tmpCage.takeFromShip((Ship<Tiger>) mammalShip);
23       mammalShip.content = (T) tiger;
24       tigerCage.content = tmpCage.content;
25    }
26 }
```

A) List all the typecasts that the virtual machine will perform at runtime, when executing the methods `takeFromShip` and `swapTigers`. For each cast, write at which line number in the original program it is performed, and what expression is cast to which type. Do not optimize away casts that are statically known to succeed.

solution

- At line 13: `(Mammal) other.content`
- At line 20: `(Tiger) tigerCage.content`
- At line 22 `(Ship) mammalShip`
- At line 23: `(Mammal) tiger`

B) For each of the following two methods (from **B.1** and **B.2**), write if they would compile without errors if added to the class `Cage`. If they do not compile, briefly explain why.

B.1

```
Cage<Tiger>[] getTigers(int number) {
   Cage<Tiger>[] cages = new Cage<Tiger>[number];
   for (int i = 0; i < number; i++) {
      cages[i] = new Cage<Tiger>();
      cages[i].content = new Tiger();
   }
   return cages;
}
```

B.2

```
int numCageFields() {
   Class cl = Cage<Animal>.class;
   return cl.getFields().length;
}
```

solution

1. Compiler error: array of generic types not allowed.

2. Compiler error: class object of generic types not available, or the type parameter does not respect the constraint of class Cage.

Task 3

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 4

The type correctness of a C++ template class is checked only when the template is instantiated. This makes it difficult to develop templates modularly. We can try to make templates more modular by extending C++ with a new way to declare type arguments:

```
template<T s_extends SomeClass>
class TemplateClass {...}
```

Here `T` is the template argument and `SomeClass` is the name of a class which is an upper type bound for `T`. A template defined in this way may only be instantiated with a class `T` that is a ***structural*** subtype of `SomeClass`. Assume that the type checker checks such a template *definition* without having any concrete instantiation, under the assumption that `T` is a structural subtype of `SomeClass`.

This new feature is the only place where we introduce structural subtyping in C++, all other subtype relations in the language remain nominal as usual. Assume in general for any subtyping mode that method argument types are contravariant and method return types are covariant. Also, assume that all the methods are `public` and `virtual`.

A) Provide a declaration of the `Operation` class such that the class `Compose` can be type-checked before it is instantiated.

```
template<T s_extends Operation , U s_extends Operation>
class Compose : public Operation {
  public:
     T* t;
     U* u;
     int compute(int x) {
        return t->compute(u->compute(x));
     }
}
```

solution

```
class Operation {
   int compute(int x);
}
```

B) We also allow template parameters to occur as type arguments in upper bounds of the same template:

```
template<T s_extends Bound<T>>
class TemplateClass{...}
```

The above limits the possibilities for `T` to only structural subtypes of `Bound<T>`.

Consider the classes below:

```
class A :             { void foo(A* a); };
class B : public A    { B*   bar();     };
class C : public B      {};

template <class T>
class FOO {
  void foo(T* t){...}
};

template <T s_extends FOO<T>>
class X { ... };

template <class T>
class BAR {
   T* bar(){...}
};

template <T s_extends BAR<T>>
class Y { ... };
```

Which of the following instantiations typecheck:

```
X<B>
X<C>
Y<B>
Y<C>
```

Explain why each combination does or does not typecheck.

solution

`X` can be instantiated with both classes:

- `B`: `B.foo(A*)` overrides `FOO<B>.foo(B*)`
- `C`: `C.foo(A*)` overrides `FOO<C>.foo(C*)`

`Y` can be instantiated only with `B`:

- `B`: `B* B.bar()` overrides `B* BAR<B>.bar()`
- `C`: `B* C.bar()` does not override `C* BAR<C>.bar()`, therefore `C` is not a structural subtype of `BAR<C>`

C) As a bound we also allow the template that is being declared:

```
template <T s_extends X<T>>
class X {
   int foo(T* t) {...}
}
```

Let the class `A` be:

```
class A {};
```

- Write an implementation of the body of the `foo` method of `X` such that `X` typechecks with the bound above (`T s_extends X<T>`) and also typechecks if the bound is changed to `T s_extends A`.
- Write an implementation of the body of the `foo` method of `X` such that `X` typechecks with the bound above (`T s_extends X<T>`), but does not typecheck if the bound is changed to `T s_extends A`.
- Write a class `B` that can be used to instantiate `X`.

solution

- `int foo(T* t) { return t ? 1 : 0; }`
- `int foo(T* t) { return t->foo(t); }`
- `class B { int foo(B* b) { return 0; } }`

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

```
template <class T>
class Mixin : public T { ... }
```

Such a template is called a mixin. We want to use the newly introduced template bound feature `<T s_extends ...>` in order to create a mixin that is guaranteed only to override existing methods but not introduce new ones. Show how this can be done.

solution

```
template <T s_extends Mixin<T>>
class Mixin : public T
```

Here we restrict the base class `T` to be a structural subtype of `Mixin<T>`. Thus all admissible base classes `T`, have at least the methods that `Mixin<T>` defines in its body.

If the mixin tried to add a new method, this would fail. For example:

```
template <T s_extends Mixin<T>>
class Mixin : public T {
  public: int foo(int x) { return x + T::foo(x); } // overriden method
          int bar(int y) { return 2 + y; } // newly-added method
}

class A { public: int foo(int x) { return 2 * x; } }
```

The instantiation `Mixin<A>` fails, because `A` is not a structural subtype of `Mixin<A>`, since `A` does not have the `bar` method.

Task 5

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.

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`).

Task 6

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`.