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?

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())?

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

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?

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#.)

Task 2

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

```
1  class Animal {}
2  class Mammal extends Animal {}
3  class Tiger extends Mammal {}
4
5  class Ship<T extends Animal> {
6     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.

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;
}
```

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
```

B) Describe how the instantiation above will look like.

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

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));
      }
}
```

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.

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`.

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.

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.

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.