Concepts of Object-Oriented Programming
AS 2020

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 14

Self-Study Exercise Sheet

NOTE: This exercise sheet will not be discussed in an exercise session. We publish it together with the solution to help you better prepare for the exam. If you have any questions, please submit them for the Q&A session.

Subtyping and Behavioral Subtyping

Task 1

Assume the following class definitions in a nominally typed language:

```
class A {...}
class B extends A {...}
```

Consider now the following two classes:

```
class Super {
  B foo(B x) { return x; }
}

class Sub extends Super {
  A foo(A x) { return x; }
}
```

According to the rules presented in the lecture, this subtyping is illegal. Briefly explain why this is the case. However, considering the substitution principle, this subtyping is *safe*. Why?

solution

According to the rules from the lecture, overriding methods should have covariant results; `Sub.foo()` returns an `A` and $A \not<: B$.

However, an object of `Sub` has a wider interface (it applies to arguments of more types). When it is applied to an object of type `B`, it returns a `B`, which is exactly the behavior expected by the client from an object of `Super`.

Task 2

Consider the class `X` and its only method `foo`, where `ZZZ` is a placeholder for a class name:

```
class X {
   /// requires x > 0 ∧ (¬∃i,j: int | 2 ≤ i,j ≤ x ∧ i * j = x)
   /// ensures result > 0 ∧ result % 2 = 0
   int foo(final int x){ return (new ZZZ()).foo(x); }
}
```

Which of the four classes below could be substituted for `ZZZ` such that no contracts will be violated?

(a)
```
class A {
    /// requires x ≥ 0
    /// ensures result = x + 1
    int foo(final int x) {...}    }
```

(b)
```
class B {
    /// requires true
    /// ensures result % 2 = 0
    int foo(final int x) {...}    }
```

(c)
```
class C {
    /// requires x % 2 = 1
    /// ensures result = x + 1
    int foo(final int x) {...}    }
```

(d) **CORRECT:**

```
class D {
    /// requires true
    /// ensures result = x * (x + 1)
    int foo(final int x) {...}    }
```

solution

Choice `(a)` is not valid since 2 is a valid input to `X::foo()`, but breaks the postcondition if `result = x + 1`.

Choice `(b)` is not valid as it has a weaker postcondition, namely the result is not guaranteed to be larger than 0.

Choice `(c)` is not valid as it does not have a weaker precondition. Note that `X::foo()` accepts 1 and all prime numbers. However, this includes 2, which is even and thus not allowed by the precondition of `C::foo()`.

Choice `(d)` has a weaker precondition. Moreover, on strictly positive inputs, it guarantees strictly positive even outputs. Therefore it has a stronger postcondition.

Inheritance, Dynamic Method Binding, Multiple Inheritance, and Linearization

Task 3

Consider the following Java classes and interfaces:

```
public interface IA { IA g(IA x); }

public interface IB extends IA { IB g(IA x); IA g(IB x); }

public interface IC extends IA { IC g(IB x); }

class B implements IB {
   public IB g(IA x) { System.out.print("B1 "); return null; }
   public IC g(IB x) { System.out.print("B2 "); return null; }
}

class C implements IC {
   public IC g(IA x) { System.out.print("C1 "); return null; }
   public  C g(IB x) { System.out.print("C2 "); return null; }
}
```

```
class Main{
   public static void main(String[] args) {
      B  b  = new B();
      C  c  = new C();
      IA a1 = b;
      IA a2 = c;

      IA r1 = a1.g(a2);
      IA r2 = a2.g(b);
      IC r3 = b.g(b);
      IA r4 = c.g(a2);
      C  r5 = c.g(b);
   }
}
```

What is the output of the execution of the `Main.main` method? Explain your answer.

solution

The code will print `B1 C1 B2 C1 C2`:

`a1` has static type `IA`, `a2` has static type `IA`: the statically selected method for the first call is `IA.g(IA)`. This method is overriden in `B` (the dynamic type of `a1`) by `B.g(IA)`.

`a2` has static type `IA`, `b` has static type `B`: the statically selected method for the second call is `IA.g(IA)`. This method is overriden in `C` (the dynamic type of `a2`) by `C.g(IA)`.

`b` has static and dynamic type `B`: the statically selected method for the third call is `B.g(IB)` (the most specific method according to the overloading resolution). This method will be executed at runtime.

`c` has static and dynamic type `C`, `a2` has static type `IA`: the statically selected method for the fourth call is `C.g(IA)`. This method will be executed at runtime.

`c` has static and dynamic type `C`, `b` has static type `B`: the statically selected method for the last call is `C.g(IB)` (the most specific method according to the overloading resolution). This method will be executed at runtime.

Task 4

Consider the following C++ code:

```
class Person {
   bool likesDiamonds;

   public: Person (bool l) { likesDiamonds = l; }
};

class Programmer : virtual public Person {
   public: Programmer () : Person (false) {}
           // diamonds are a programmer's worst enemy
};
```

It is expected that `!likesDiamonds` is an invariant for the class `Programmer`. Use inheritance to break this invariant, without altering the above code.

solution

The following C++ code breaks the invariant:

```
class Jeweler : virtual public Person {
   public: Jeweler () : Person (true) {}
```

```
           // diamonds are a jeweler's best friend
};

class JewelerProgrammer : public Jeweler, public Programmer {
   public: JewelerProgrammer () :
           Person (true), Jeweler (), Programmer () {}
};

void break () {
   JewelerProgrammer* programmer = new JewelerProgrammer();
}
```

The call of the constructor `Person(true)` in class `JewelerProgrammer` bypasses the corresponding call `Person(false)` in class `Programmer`, breaking the invariant.

Task 5

Write three C++ classes:

- A class `Queue` that represents a queue of integers and has an `enqueue` and a `dequeue` method
- A subclass of `Queue` that also stores (and allows clients to retrieve) the current sum of all items in the queue, using the `enqueue` and `dequeue` methods
- A subclass of `Queue` that also stores (and allows clients to retrieve) the current product of all items in the queue, using the `enqueue` and `dequeue` methods

We would now like to have a class that supports both functionalities (i.e., stores and allows clients to retrieve both the sum and the product of all the items in the queue).

- Suppose that we use multiple inheritance and override the `enqueue` and `dequeue` methods of the new class, such that the new methods call the `enqueue` and `dequeue` methods of both of the old classes. Are there any problems with this approach?
- How can you solve this problem in Scala, using traits? Does this fix the above-mentioned problems from C++?

solution

Here are the three requested classes:

```
class Queue {
   int[] contents;
   int size;

   public: Queue() { contents = new int[100]; size = 0; }
           void enqueue(int x) {...}
           int dequeue() {...}
           int getSize() { return size; }
};

class SumQueue : virtual public Queue {
   int sum;

   public: SumQueue() : Queue() { sum = 0; }
           void enqueue(int x) {
             sum += x;
             Queue::enqueue(x);
           }
```

```
            int dequeue() {
              int r = Queue::dequeue();
              sum -= r;
              return r;
            }

            int getSum() { return sum; }
};

class ProductQueue : virtual public Queue { ... };

class SuperQueue : public ProductQueue, SumQueue {
   public: SuperQueue() : Queue(), ProductQueue(), SumQueue() {}
            void enqueue(int x) {
              ProductQueue::enqueue(x);
              SumQueue::enqueue(x);
            }

            int dequeue() {
              int r = ProductQueue::dequeue();
              SumQueue::dequeue();
              return r;
            }
};
```

One problem is that the enqueue and dequeue methods of the superclass are called twice. An item is enqueued and dequeued twice. Interestingly, this behaves exactly like a queue, but the capacity is half of the capacity of the original and the getSize method reports the correct size multiplied by 2.

We can use traits and linearization to ensure that the enqueue/dequeue methods are called only once. Here is the relevant Scala code:

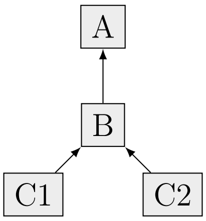
```
class Queue {
   ...
   def enqueue(x: int) = {...}
   def dequeue(): int = {...}
}

trait Sum extends Queue {
   var sum: int = 0
   override def enqueue(x:int) = { sum += x; super.enqueue(x); }
   override def dequeue(): int = {
        var x = super.dequeue();
        sum = sum - x;
        return x;
   }
}

trait Prod extends Queue {
   var count: int = 1
   override def enqueue(x: int) = { prod *= x; super.enqueue(x); }
   override def dequeue(): int = {
        var x = super.dequeue();
        prod = prod / x;  // this assumes no zeros in the queue
        return x;
   }
}
```

Now, an object of type `Queue with Sum with Prod` has both functionalities, but calls each underlying `enqueue`/`dequeue` method only once. The problems of the multiple inheritance solution do not appear here.

Bytecode Verification

Task 6

Consider the following type hierarchy:



Suppose that the method `f` of class `E` has the following signature:

```
A f(boolean b1, boolean b2);
```

and there are three local variables `x`, `y`, `z`. The maximal stack size is equal to 1.

The method `f` contains the following code snippet:

```
 0: iload 1
 1: ifeq  22
 4: iload 2
 5: ifeq  12
 8: aload 3
 9: goto 14
12: aload 4
14: astore 3
15: aload 5
17: astore 4
19: goto 0
22: aload 3
23: areturn
```

It is known that the state at the beginning of the snippet is:

```
([], [E,boolean,boolean,C1,C2,A])
```

Note: In this example, `ifeq x` pops an integer from the stack and jumps to line `x` if the integer is equal to zero.

A) Verify that the code snippet is type safe.

solution

Here the initial state is `([], [E,b,b,C1,C2,A])`. We denote the type `boolean` as `b` for convenience (in reality the Java bytecode verifier views it as an integer). We show the solution following the convention from the lecture. To each command we dedicate an input and an output column. A command may have multiple inputs and outputs, corresponding to the different iterations of the algorithm. You may also want to see an animated solution of this task, published separately.

		IN	OUT
0	`iload 1`	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	([b], [E,b,b,C1,C2,A]) ([b], [E,b,b,B,A,A]) ([b], [E,b,b,A,A,A])
1	`ifeq 22`	([b], [E,b,b,C1,C2,A]) ([b], [E,b,b,B,A,A]) ([b], [E,b,b,A,A,A])	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])
4	`iload 2`	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	([b], [E,b,b,C1,C2,A]) ([b], [E,b,b,B,A,A]) ([b], [E,b,b,A,A,A])
5	`ifeq 12`	([b], [E,b,b,C1,C2,A]) ([b], [E,b,b,B,A,A]) ([b], [E,b,b,A,A,A])	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])
8	`aload 3`	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	([C1], [E,b,b,C1,C2,A]) ([B], [E,b,b,B,A,A]) ([A], [E,b,b,A,A,A])
9	`goto 14`	([C1], [E,b,b,C1,C2,A]) ([B], [E,b,b,B,A,A]) ([A], [E,b,b,A,A,A])	([C1], [E,b,b,C1,C2,A]) ([B], [E,b,b,B,A,A]) ([A], [E,b,b,A,A,A])
12	`aload 4`	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	([C2], [E,b,b,C1,C2,A]) ([A], [E,b,b,B,A,A]) ([A], [E,b,b,A,A,A])
14	`astore 3`	([C1], [E,b,b,C1,C2,A]) ([B], [E,b,b,C1,C2,A]) ([B], [E,b,b,B,A,A]) ([A], [E,b,b,B,A,A]) ([A], [E,b,b,A,A,A])	- ([], [E,b,b,B,C2,A]) - ([], [E,b,b,A,A,A]) ([], [E,b,b,A,A,A])
15	`aload 5`	([], [E,b,b,B,C2,A]) ([], [E,b,b,A,A,A])	([A], [E,b,b,B,C2,A]) ([A], [E,b,b,A,A,A])
17	`astore 4`	([A], [E,b,b,B,C2,A]) ([A], [E,b,b,A,A,A])	([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])
19	`goto 0`	([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])
22	`aload 3`	([], [E,b,b,C1,C2,A]) ([], [E,b,b,B,A,A]) ([], [E,b,b,A,A,A])	- - ([A], [E,b,b,A,A,A])
23	`areturn`	- - ([A], [E,b,b,A,A,A])	- - ([], [E,b,b,A,A,A])

B) Provide the minimal type information that enables the type checking algorithm (i.e., the algorithm that does not perform a fixpoint computation) to verify the bytecode.

solution

In the following code, we show the types that are given by the user, and those inferred by the type checker.

```
  // given: ([],[E,b,b,A,A,A])
0: iload 1
  // ([b], E,b,b,A,A,A])
1: ifeq 22
```

```
  // ([], [E,b,b,A,A,A])
4: iload 2
  // [b], [E,b,b,A,A,A]
5: ifeq 12
  // ([], [E,b,b,A,A,A])
8: aload 3
  // ([A], [E,b,b,A,A,A])
9: goto 14


  // ([], [E,b,b,A,A,A])
12: aload 4
  // given: ([A], [E,b,b,A,A,A])
14: astore 3
  // ([], [E,b,b,A,A,A])
15: aload 5
  // ([A], [E,b,b,A,A,A])
17: astore 4
  // ([], [E,b,b,A,A,A])
19: goto 0
  // ([], [E,b,b,A,A,A])
22: aload 3
  // ([A], [E,b,b,A,A,A])
23: areturn
  // ([], [E,b,b,A,A,A])
```

The requirement to have type information at all basic blocks is a simplification that makes it easier to determine where the compiler should output the information. Note that some basic blocks have only a single preceding instruction, but determining this statically could be hard. Such basic blocks, in theory, do not need type information. Only basic blocks that are also join points definitely need type information. In our example, the instructions 4, 8, 12 and 22 are indeed the beginnings of basic blocks, but there is exactly one path to enter these blocks and therefore type information is not really needed since this information will be identical to the *out*-state of the single preceding instruction.

Task 7

The bytecode type inference algorithm rejects a verified program if there are different stack sizes for input values of a join point.

A) Provide a bytecode program that is rejected because of this limitation but that does not cause runtime errors.

solution

```
0 : aload 0
1 : iconst 1
2 : ifne 4
3 : aload 0
4 : astore 1
```

Note: `ifne` jumps to the given index if the integer value at the top of the stack is not equal to zero. It pops the value at the top of the stack.

There are two possibilities for the stack size after executing this program. In any of the two cases, the height of the stack at point 4 is at least 1, and there will be surely a reference value at the top of the stack.

B) Is it possible to construct a bytecode verification algorithm that avoids this limitation? If yes, then provide an updated algorithm. If no, then show that it cannot be done.

solution

We distinguish between two different cases:

1. If the stack sizes are statically known we can construct such an algorithm. The update is as follows: when joining stacks of different sizes, pick the smallest one, but carry as extra information the size of the largest one to be used when checking for overflow. Note that if we just picked the smaller one and used that, we would not prevent stack overflows at runtime. If we just picked the largest one and made the "extra" values into dummy values by giving them the "top" type, we might not prevent underflows when using instructions such as `pop()`.
2. In general it is not possible to implement an algorithm that can deal with stack sizes which could vary at runtime. For example, if we push elements on top of the stack in a loop, then the verifier will have no way of deciding what an upper bound for the size is. Conversely for loops which pop elements from the stack, the verifier will not be able to deduce a lower bound for the stack size. These situations can easily result in over/underflows and should be rejected.

C) How serious is this restriction from a pragmatic perspective?

solution

This limitation is not essential. If there are two states `{[head1, x], [head2]}` where `head1` and `head2` are stacks of the same size, then any following code cannot access `x` and it would have been possible to remove `x` already during bytecode generation. This is indeed what the Java compiler does. Consider the following Java code:

```java
public int bar() { return 42; }

public int foo(int x) {
  if (x == 0) { bar(); }
  return x;
}
```

If `bar` is called then it will put 42 on the stack, but this value is not actually needed for the final `return` instruction. The Java compiler would emit as many `pop` instructions as necessary to remove unneeded stack elements and make sure that all the paths that reach `return` have the same stack length. Here is the bytecode that corresponds to the `foo` method:

```
 0: iload_1
 1: ifne          9
 4: aload_0
 5: invokevirtual bar // Call to bar(), puts an int on the stack
 8: pop               // Pop the stack to remove the unnecssary int
 9: iload_1           // Here we get equal stack sizes from both paths
10: ireturn
```

Parametric Polymorphism

Task 8

Consider the following Java code:

```
class Box<T extends Number> {
   private T t;

   public void set(T t) { this.t = t; }
   public T get() { return t; }
}

class Main {
   public static void main(String[] args) {
      Box<Number> b = new Box<________>();
      b.set(new ________);
      ________ c = b.get();
      System.out.println( c );
   }
}
```

and recall that `Integer <: Number <: Object`. How can you fill in the blanks in the `Main.main` method so that the code compiles and executes successfully?

(a) `Integer, Integer(9), Integer`

(b) `Integer, Integer(9), Object`

(c) `Number, Integer(9), Integer`

(d) **CORRECT:** `Number, Integer(9), Object`

(e) None of the above

solution

Choices `(a)` and `(b)` are not valid as generic types are invariant in Java. Therefore, assigning a `Box<Integer>` to a `Box<Number>` is illegal.

Choice `(c)` is not valid since `b.get()` would return a `Number`, hence dissallowing the assignment `Integer c = b.get()`.

Choice `(d)` is valid. In the first gap, `Number` is clearly a valid option. In the second, by the substitution principle, we can pass an `Integer` as it is a subtype of `Number`. Finally, the assignment `Object c = b.get()` simply adds an implicit upcast from `Number` to `Object`, which is valid as `Number` is a subtype of `Object`.

Choice `(e)` is not valid as choice `(d)` is valid.

Task 9 (extended version of a previous exam question)

Consider the following Java code:

```
interface Food {}
interface Grass extends Food {}
interface Meat extends Food {}

abstract class Animal<F extends Food> implements Meat {
   abstract void eat(F food);
   F getLunchBag(){ return lunchBag; };
   F lunchBag;
}

final class Sheep extends Animal<Grass> { void eat(Grass f) {} }
final class Wolf extends Animal<Meat>   { void eat(Meat  f) {} }

class Cage { //You are allowed to modify this class
   Cage(Animal<?> animal){ this.animal = animal; }
   Animal<?> getAnimal() { return animal; }
   Animal<?> animal;
}

class Zoo {
   void feedAnimal(Cage cage){ /*code given in each section*/ }

   <F extends Food> void feed(F food, Animal<F> animal) {
      animal.eat(food);
   }

   void manage(){ /*your code here*/ }
}
```

Clearly a `Wolf` can eat a `Sheep` but not the other way around. In the following subtasks we explore if relaxing some of the Java type rules can lead to a situation where a `Sheep` can eat a `Wolf` - that is, the method `eat` is called on an object of the dynamic type `Sheep` with an argument object of the dynamic type `Wolf`. All the code you give in the answers to the following sections is in the same package as the code above, and must type check in standard Java. The top method that is called in all sections is `Zoo.manage`. You can assume that method call arguments are evaluated left to right, and that the program is sequential. You are not allowed to use reflection, raw types, or type-casts.

A) Assume the following body of `Zoo.feedAnimal(Cage cage)`, which is rejected by the Java type checker:

```
{ feed(cage.getAnimal().lunchBag, cage.getAnimal()); }
```

Make a `Sheep` eat a `Wolf` assuming the body of `feedAnimal` is exempted from the type checker. Show all necessary code. You are only allowed to change the `Cage` class and provide the body of the `Zoo.manage` method.

solution

Note that in order to have a `Sheep` eat a `Wolf`, `Cage.getAnimal().lunchbag` needs to return a `Wolf`, and the second call to `Cage.getAnimal()` to return a `Sheep`. This is not possible with the current implementation of `Cage.getAnimal()`. Therefore the solution is to change the implementation of said function to return different objects for different calls. This can be done in several ways.

The following code uses an implicit flag (`null` field), but using an explicit flag is also possible. Another possible solution is to count the number of times the function was called and alternate the objects that are returned.

```
class Cage {
   ...
   Animal<?> getAnimal() {
      if (animal != null)  return animal;
      else {
         animal = new Sheep();
         Wolf wolf = new Wolf();
         wolf.lunchBag = wolf;
         return wolf;
      }
   }
}
```

```
class Zoo {
   ...
   void manage(){
      feedAnimal(new Cage(null));
   }
}
```

B) Assume the following body of `Zoo.feedAnimal(Cage cage)`, which is rejected by the Java type checker:

`{ feed(cage.animal.getLunchBag(),cage.animal); }`

Can you make a `Sheep` eat a `Wolf` if the body of `feedAnimal` is exempted from the type checker? If so, show all necessary code, otherwise explain why not. You are only allowed to change the `Cage` class, provide the body of the `Zoo.manage` method, and add new classes.

solution

The solution here is to realize that the only code that can be modified that will run between the two accesses of `cage.animal` is in the call to `cage.animal.getLunchbag()`. Somehow the code will need to change the animal contained in the cage so that the access of `cage.animal` returns a `Sheep`. Clearly the animal cannot be a `Sheep` all along, as this would disallow returning a `Wolf` from `getLunchbag()`. The idea is to have an animal capable of eating a `Wolf` (a subtype of `Animal<Meat>` that contains a reference to its cage, in order to change the contents of its own cage to a `Sheep` during the call to `getLunchbag()`:

```
class Fox extends Animal<Meat> {
   Fox() {}
   void eat(Meat m) {}
   Wolf getLunchBox(){ cage.animal = new Sheep(); return new Wolf(); }
   Cage cage;
}

class Zoo{
   ...
   void manage(){
      Fox fox = new Fox();
      Cage cage = new Cage(fox);
      fox.cage = cage;
      feedAnimal(cage);
   }
}
```

C) Answer the question in the previous section, assuming the field `Cage.animal` is `final`. Explain your answer. Reminder: Java's `final` fields can be assigned to only in the constructor of the class that declares them.

solution

Here we cannot make a sheep eat a wolf.

The reason is that `cage.animal` evaluates to the same value in both expressions `cage .animal` and `cage.animal.getLunchBox()` and so type safety is not broken and the `Sheep` can only be fed with `Grass`, which the `Wolf` is not.

D) Assume the following body of `Zoo.feedAnimal(Cage cage)`, which is rejected by the Java type checker:

```
{ feed(cage.animal.lunchBag, cage.animal); }
```

Can you make a `Sheep` eat a `Wolf` if the body of `feedAnimal` is exempted from the type-checker? If so, show all necessary code, otherwise explain why not.

solution

This is safe as no methods are called during the evaluation of the arguments, so `cage.animal` cannot change.

E) Which of the above cases, that is safe in the sequential case, is unsafe in a multithreaded program? For each such case, explain what can happen in the multithreaded case that cannot happen in the sequential case.

solution

The version of `feedAnimal` in section D is unsafe as another thread might modify `Cage.animal` between the evaluation of the two expressions. The version in section C is safe.

Information Hiding and Encapsulation

Task 10

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

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

solution

The subtyping relations are as follows: `C <: B <: A`

In a language with structural subtyping, methods and fields of subclasses should be more accessible than those of the superclasses. For access modifiers, this means that methods with more permissive modifiers may override methods with less permissive modifiers.

Task 11

Consider the class `Hour`, defined as follows:

```
public class Hour {
   protected int h = 0;
   /// invariant h >= 0 && h < 24

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

What is the external interface of `Hour`?

solution

The external interface is composed only of the method `public set(int)` since this is the only public element of class `Hour`.

Can we extend the code, without changing the class, so that the invariant is broken? If yes, provide an example, and propose how to fix the class.

solution

The invariant can be broken easily by extending class `Hour`, and accessing the field `h` directly. For instance:

```java
public WrongHour extends Hour {
    public WrongHour() { super.h = -1; }
}
```

This can be prevented by making the field `h` private.

Task 12 (from a previous exam)

Consider the following Java program consisting of two packages `BTC` and `B2X`:

```java
package BTC;

public class Chain {

    /// ensures result <= 2
    ________ int max_size() {
        return 2;
    }
}

package B2X;
import BTC.*;

public class Chain2x extends Chain {

    /// ensures result <= 4
    protected int max_size() {
        return 4;
    }
}
```

A) What is the *most permissive* access modifier for the method `max_size()` in class `Chain` such that class `Chain2x` is a *behavioral subtype* of `Chain`? Assume that we *do not use* specification inheritance. **Fill the blank above with your answer**. Explicitly write `default` for a default access modifier. Write `none` if no choice of access modifier allows `Chain2x` to be a behavioral subtype of `Chain`.

solution

The method `max_size()` in class `Chain` should have a `default` access modifier, so that method `max_size()` in class `Chain2x` does not override it but only hides it. In this way, even if method `max_size()` in class `Chain2x` has a weaker postcondition than method `max_size()` in class `Chain`, we still vacuously have behavioral subtyping.

B) We now add a class `Block` and a subclass `Block2x` to package `BTC`:

```java
package BTC;

public class Block {

    protected int num;
    /// invariant: 1 <= num

    public Block(int n) {
        num = (n < 1 ? 1 : n);
    }

}

public class Block2x extends Block {

    /// invariant: 2 <= num
    protected Block pred;
    /// invariant: pred != null ==> pred.num < num

    public Block2x(int n, Block b) {
        super(n < 1 ? 2 : 2*n);
        pred = (b != null && 2 <= b.num && b.num < num ? b : null);
    }

}
```

B.1) Do the invariants in `Block` and `Block2x` satisfy the requirements of *behavioral subtyping*? Assume that we *do not use* specification inheritance. **Briefly explain your answer**.

solution

Yes, the invariants satisfy the requirements of behavioral subtyping because the invariants in class `Block2x` are stronger than the invariants in class `Block`.

B.2) A class `C` is *correct* with respect to its invariants if all constructors of `C` establish the invariants *of the new object* and all exported methods `m` of `C` preserve the invariants *of the receiver object*, that is, the invariant holds in the poststate of `m` provided that it held in the prestate of `m`. Are classes `Block` and `Block2x` correct with respect to their invariants? **Briefly explain your answer**.

solution

Yes, classes `Block` and `Block2x` are correct with respect to their invariants because their constructors establish the invariants of the newly created objects (and there are no methods in the two classes).

C) We now want to extend the code in part **B** with *methods that preserve the invariants of the class in which they are declared* but that make it possible to violate the invariants of `Block2x` from client code in another package.

C.1) Extend the code in part **B** with a method that preserves the invariants of the class in which it is declared but makes it possible to violate the *first invariant* of class `Block2x` (i.e., `2 <= num`) from client code in package `B2X`. **Specify in which class you want to declare the method, write the method, and write the client code that violates the invariant.**

solution

It is possible to break the invariant by adding the following method to class `Block`:

```
public void reset() { num = 1; }
```

The client code that breaks the invariant is the following:

```
class Client {
    public static void main(String[] args) {
        Block2x b2x = new Block2x(1, null);
        b2x.reset();
    }
}
```

C.2) How can you prevent the code that you wrote in part **C.1** from violating the invariant by further extending the code in part **B**? *You are not allowed to modify existing code.* **Write the code that fixes the specific problem you exploited in part C.1.**

solution

It is possible to prevent the above problem by overriding the newly added method `reset` in class `Block2x`:

```
public void reset() { num = 2; }
```

C.3) Extend the code in part **B** with a method that preserves the invariants of the class in which it is declared but makes it possible to violate the *second invariant* of class `Block2x` (i.e., `pred != null ==> pred.num < num`) from client code in package `B2X` *in a way that cannot be prevented by further extending the code in part* ***B***. **Specify in which class you want to declare the method, write the method, and write the client code that violates the invariant.**

solution

It is possible to break the invariant by adding the following method to class `Block`:

```
public void incr() { num = num + 1; }
```

The client code that breaks the invariant is the following:

```
class Client {
    public static void main(String[] args) {
        Block b = new Block(2);
        Block2x c = new Block2x(2, b);
        b.incr();
        b.incr();
    }
}
```

Aliasing, Readonly Types, and Ownership Types

Task 13

Consider the following C++ class:

```
class Person {
   int money;
   Person *spouse;

   public: Person (int m, Person *s) {
               if (!s) { spouse = NULL; }
               else { spouse = s; s->spouse = this; }
               money = m;
            }
            void f () const;
};
```

The method `f` promises not to make any changes to its receiver object. Provide an implementation for `f` that violates this claim. You are not allowed to use casts, nor to introduce any local variables.

solution

We can violate the claim by changing the receiver object `this` through the field `spouse`, for instance : `spouse->spouse->money = 0;`

This would only work if the `s` passed in the constructor is non-null.

Task 14

Consider the following class definitions in the context of the read-only type system taught in the course:

```
class C {
   public D f;
   void foo(readonly C other) {...}
}

class D { E g; }

class E {}
```

Let `a` and `b` be non-null references of type `C`. Which of the following statements is true:

(a) The call `a.foo(b)` is guaranteed not to change the value of `b.f`, but may change the value of `b.f.g`

(b) The call `a.foo(b)` is guaranteed not to change the value of `b.f` and neither the value of `b.f.g`

(c) The assignment `other.f.g = new E();` may appear in the code of `foo`

(d) **CORRECT:** None of the above is correct

solution

Choice `(a)` and `(b)` are not true since we can have aliasing (`a` and `b` point to the same object) and `foo()` has no restriction on modifying its receiver, therefore it might modify the value of `b.f` via the alias `a`.

Choice `(c)` is not true since readonly types are transitive, meaning that `other.f.g` is readonly since `other` is readonly. Therefore the assignment is not allowed.

Task 15

Annotate the following program with appropriate ownership type modifiers (according to the topological ownership system) in order to maximize the buffer, the producer, and the consumer *encapsulation*. This means that the modifiers you choose should increase the depth of nested ownership context and reduce the number of (non-rep) edges/pointers between different contexts.

```
class Producer {
 int[] buf;
 int n;
 Consumer con;

 Producer() {
  buf = new int[10];
 }

 void produce(int x) {
  buf[n] = x;
  n = (n+1)
   % buf.length;
 }
}
```

```
class Consumer {
 int[] buf;
 int n;
 Producer pro;

 Consumer(Producer p) {
  buf = p.buf;
  pro = p;
  p.con = this;
 }

 int consume() {
  n = (n+1)
   % buf.length;
  return buf[n];
 }
}
```

```
class Context {
 Producer p;
 Consumer c;

 Context() {
  p = new Producer();
  c = new Consumer(p);
 }

 public void run() {
  for(int i=-5; i<=5;
      ++i) {
   p.produce(i);
   if(i%2 == 0)
    c.consume();
  }
 }
}
```

solution

```
class Producer {
 rep int[] buf;
 int n;
 peer Consumer con;

 Producer() {
  buf = new rep int
      [10];
 }

 void produce(int x) {
  buf[n] = x;
  n = (n+1)
   % buf.length;
 }
}
```

```
class Consumer {
 any int[] buf;
 int n;
 peer Producer pro;

 Consumer(peer
    Producer p) {
  buf = p.buf;
  pro = p;
  p.con = this;
 }

 int consume() {
  n = (n+1)
   % buf.length;
  return buf[n];
 }
}
```

```
class Context {
 rep Producer p;
 rep Consumer c;

 Context() {
  p = new rep Producer
      ();
  c = new rep Consumer
      (p);
 }

 public void run() {
  for(int i=-5; i<=5;
      ++i) {
   p.produce(i);
   if(i%2 == 0)
    c.consume();
  }
 }
}
```

We do not have to add ownership modifiers to primitive types. We could have annotated `con` in `Producer` and `pro` in `Consumer` as `any` — in general, this would even allow one modification less (in the topological system): of an `any` receiver, only an `any` field can be modified, whereas of a `peer` receiver, both a `peer` and an `any` field can be modified. However, our goal is to maximize encapsulation, and therefore `peer` is the best choice here.

Task 16

In the following question we do not consider the owners-as-modifiers discipline. We are only concerned with the topology of the ownership type system.

Consider the assignment:

```
o.f = p.g;
```

and assume that `o.f` and `p.g` have the same static type.

A) The assignment is forbidden if `o.f` has ownership modifier `lost`. Show an example to demonstrate why we need this rule to preserve the topological invariant.

solution

The following code breaks the acyclicity requirement for the topology:

```
class C {
  rep C down;

  void foo() {
    down.down = this;
  }
}
```

B) If the ownership modifier of `o.f` is `any`, then what are the requirements for the assignment to be legal?

solution

None. The assignment is always legal.

C) If `o.f` has ownership modifier `lost` can we upcast `o.f` to an `any` reference and make the assignment legal? Why (not)?

solution

We cannot upcast a reference that is being assigned to. This is illegal according to the subtyping rules.

Task 17 (from a previous exam)

Consider the following declarations:

```
class A {
   rep B first;
   rep B second;
}

class B {
   any A obj;
   peer B sibling;
}
```

Which of the following programs are allowed in the topological ownership system? For any program that is accepted in the topological system, is it also accepted in the owner-as-modifier system? Assume that none of the objects involved are `null`. Briefly explain each of your answers.

Program 1	**Program 2**	**Program 3**	**Program 4**

`rep B b;`
`...`
`b = b.sibling;`
	`peer A a; rep B b;`
	`...`
	`a = b.obj;`
		`any A a;`
		`...`
		`a.first.obj = a;`
			`peer A a;`
			`...`
			`a.first = a.first;`

solution

- **Program 1** is accepted in both systems.
- **Program 2** is not accepted in the topological system (and neither in the owner-as-modifier system). It attempts the assignment of an `any` reference to a `peer` reference. `peer` is not a super-type of `any`.
- **Program 3** is accepted in the topological system (it assigns `any` to `any`). However, it assigns to the field of a `lost` reference, which means that it is not accepted in the owner-as-modifier system.
- **Program 4** is not accepted in the topological system (and neither in the owner-as-modifier system), because it assigns to a `lost` location.

Task 18

Assume the topological ownership framework taught in the course. Suppose that we want to create a class `SortedLinkedList`, with the internal invariant that the values stored in the nodes are sorted in ascending order.

```
package SortedLinkedList;
public class SortedLinkedList {
  private rep Node head;

  /// invariant head != null ==> head.sorted()
  ...
}
private class Node {
  protected peer Node next;
  protected int value;

  /// pure
  boolean sorted() {
    return next != null ==> value < next.value && next.sorted()
  }
}
```

Suppose that all the methods in `SortedLinkedList` are guaranteed to preserve the invariant of the class. Furthermore, suppose that we want to create iterators for such lists (defined in the same package):

```
public class LinkedListIterator { private any Node current_item; ... }
```

A) Why is the field `current_item` annotated as `any`? What drawbacks would the other possible annotations have?

solution

If `current_item` were annotated as `rep`, then the owner of the node it refers to is the iterator itself. In this case, the iterator cannot iterate over a `SortedLinkedList` object l, because l also owns its nodes. The ownership topology allows at most one owner per object.

If `current_item` were annotated as `peer`, then, assuming that `current_item` has a list owner l, the owner of the iterator must also be l. This may be OK in topological ownership. However, if we add "owners as modifiers", the iterator's methods that traverse l cannot be called directly from an object outside l, which defeats the purpose of iterators.

B) We would like to have the following features:

(i) the invariant of a `SortedLinkedList` object is guaranteed to hold in any program, except when one of its methods executes

(ii) `LinkedListIterator` is a *modifying* iterator, i.e., it may change the value of the object it is pointing to

We can't have both features. Depending on whether or not we impose the "owners as modifiers" discipline, we can have either (i) or (ii). Explain why this is the case.

solution

If we don't have "owners as modifiers", an object may get/hold an `any` reference to a node of the list, modify its `value` field, and break the invariant: (i) is not achieved.

If we do have "owners as modifiers", then the iterator may not modify the value of the node it is pointing at, because it holds an `any` reference to it: (ii) is not achieved.

C) The fact that (i) and (ii) cannot both hold together is not surprising. A modifying iterator can break the invariant of the list it is iterating over. However, the "owners as modifiers" discipline may disallow harmless designs. Write a benign class (perhaps a restricted modifying iterator), which would not break the invariant of any object of `SortedLinkedList`, but still does not compile under "owners-as modifiers".

solution

We could have an iterator that performs the requested modification iff this does not violate the invariant:

```
public class LinkedListIterator {
  private any Node f;

  ... // some non-modifying methods

  public void modifyCarefully(int x) {
    if (f.value <= x && (f.next == null || x < f.next.value))
      f.value = x;
    // benign but does not type check under "owners as modifiers"
  }
}
```

Reflection

Task 19

Which of the following is the *defining* characteristic of reflection?

(a) It allows for much simpler code

(b) It enables more flexibility

(c) **CORRECT:** It allows a program to observe and modify its own structure and behavior

(d) It is not statically safe

(e) It may hurt performance

(f) None of the above

Task 20

Consider the following Java code:

```java
void foo() throws java.lang.Exception {
  LinkedList<String> xs = new LinkedList<String>();
  xs.add("A"); xs.add("B"); xs.add("C");

  Class<?> c = xs.getClass();
  Method remove = c.getMethod("remove");
  xs.add(remove.invoke(xs));
}
```

which uses the following methods of class `LinkedList<E>`

```java
  public E remove()
  public boolean add(E e)
```

Which of the following statements is true? The invocation of . . .

(a) `c.getMethod("remove")` is rejected by the compiler

(b) `c.getMethod("remove")` raises an exception (at runtime)

(c) `remove.invoke(xs)` is rejected by the compiler

(d) `remove.invoke(xs)` raises an exception (at runtime)

(e) **CORRECT:** `xs.add(...)` is rejected by the compiler

(f) `xs.add(...)` raises an exception (at runtime)

solution

This code snippet aims to create a `LinkedList` of `String`, add three elements to it, recover class and method information via reflection, remove an element from the list and add it to the list again.

The issue with this code is that the return type of `Method.invoke(...)` is `Object`. Therefore, the compiler complains that there is no suitable method `xs.add()` that takes an `Object` parameter returned from `remove.invoke(xs)`.