ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Exercise 12
## Initialization
## December 13, 2013

### Task 1

Consider the following abstract class, representing a node of a singly-linked list:

```
public abstract class ListNode<X> {
   public abstract void setItem(X x);
   public abstract X getItem();
   public abstract ListNode<X> getNext();
}
```

Consider now the following implementation using a simple (acyclic) list:

```
public class AcyclicListNode<X> extends ListNode<X> {
   protected X item;
   protected AcyclicListNode<X> next;

   public AcyclicListNode<X> (X item) {
      this.item = item;
      this.next = null;
   }

   public void setItem(X x) { item = x; }
   public X getItem() { return item; }
   public AcyclicListNode<X> getNext() { return next; }
}
```

In this implementation, suppose that an empty list is represented simply by a `null` reference. Suppose that a further design intention of this implementation is that each node is guaranteed to store an `X` object in its item field.

- Annotate the class `AcyclicListNode<X>` with appropriate non-null type annotations to express these design intentions as far as possible. You do not need any annotations from the Construction Types system (`free` or `unc` annotations).

Now consider an alternative implementation using a cyclic list:

```
public class CyclicListNode<X> extends ListNode<X> {
   protected X item;
   protected CyclicListNode<X> next;

   public CyclicListNode<X> (X item) {
      this.item = item;
      this.next = this;
   }

   public void setItem(X x) { item = x; }
   public X getItem() { return item; }
   public CyclicListNode getNext() { return next; }
}
```

In this implementation, the design intention is that every node will always have a next object in the list (sometimes itself). In this design, we choose to represent an empty list by a single node whose `next` field points to itself, but whose `item` field is `null`. All non-empty lists will be represented using only nodes whose `item` fields are non-null.

- Annotate the class `CyclicListNode<X>` with appropriate non-null type annotations to express these design intentions as far as possible. You do not need any annotations from the Construction Types system (`free` or `unc` annotations).
- Now consider how to annotate the method signatures in `ListNode<X>` so that both implementations can be accommodated. Your solution should be compatible with the usual co/contra-variance rules for subclass method signatures.

**Solution**

(Side note: the interaction of generic types and non-null types, e.g., the interpretation of a type `X!` if `X` can be instantiated with types that themselves include non-nullity expectations, is beyond the scope of the course, but in case you are worried, you can assume that the explicitly visible annotation `!` overrides any annotation in the instantiation for `X`, i.e., `X!` can still be safely assumed to always store a non-null value)

- The following class definitions express the design expectations:

```
public class AcyclicListNode<X> extends ListNode<X> {
   protected X! item;
   protected AcyclicListNode<X>? next;

   public AcyclicListNode<X> (X! item) {
      this.item = item;
   }

   public void setItem(X! x) { item = x; }
   public X! getItem() { return item; }
   public AcyclicListNode<X>? getNext() { return next; }
}
```

-

```
public class CyclicListNode<X> extends ListNode<X> {
   protected X? item;
   protected CyclicListNode<X>! next;

   public CyclicListNode<X> (X? item) {
      this.item = item;
      this.next = this; // default - maybe changed later
   }

   public void setItem(X? x) { item = x; }
   public X? getItem() { return item; }
   public CyclicListNode! getNext() { return next; }
}
```

  Note that we may decide to pass a non-null reference to `setItem`.

- We have to pick suitable method signatures so that the implementing methods have valid overriding signatures in both classes above. This typically means strengthening the argument types and weakening the return types:

```
public abstract class ListNode<X> {
   public abstract void setItem(X! x);
   public abstract X? getItem();
   public abstract ListNode<X>? getNext();
}
```

## Task 2

Some languages like C++ provide numerous features that enable programmers to implement advanced types with very customizable behavior. In this question we explore how to implement a universal non-null pointer template class. Your task is to write the body of the `NN` class such that the `main` function below works as expected. Incorrect code should not compile if uncommented.

After you write the code, discuss how implementing non-null types in this way is different from built-in support for non-null types.

Hint: To fully answer this question you'll need to get familiar with the following:

- Defining templates.
- Defining copy and conversion constructors.
- Defining and overloading operators (=, *, ->, type conversions).
- Defining friend classes.

```
class BadNonNullCast{}; // Throw when casting null to a non-null pointer

template <class T>
class NN {

...

};

// Some classes
class A{ public: int v{}; };
class B : public A{};
class C : public B{};
class X {};

#include <iostream>
using namespace std;
int main()
{
   try
   {
      // Standard C++ pointers
      A* a = new A(); a->v = 1;
      B* b = new B(); b->v = 2;

      // Non-null pointers can be initialized from standard pointers.
      // Non-nullnes is checked at run-time.
      NN<A> nna = a;
      NN<B> nnb = b;
      NN<C> nnc = new C(); nnc->v = 3;

      // Subtyping works as expected
      C* c = nnc;
      nnb = nnc;
      nnb = c;
      b = nnc;

      // NN<T> behaves just like a standard pointer
      cout << (*a).v << (*nna).v << endl;
      cout << c->v << nnc->v << endl;

      // Non-null checks are performed at run time.
      try{ nnb = nullptr; }
```

```
        catch (BadNonNullCast&) {cout << "Expected exception"<<endl;}

        C* nullC = nullptr;
        try{ nnb = nullC; }
        catch (BadNonNullCast&) {cout << "Expected exception"<<endl;}

        X* x = new X();
        // Each of the following will cause a compile time error
        //NN<A> uninitialized;
        //nna = x;
        //nnc = nna;
        //nnc = a;
    }
    catch (BadNonNullCast&)
    {
        cout << "Unexpected Exception" << endl;
    }

    return 0;
}
```

### Solution

Here is the body of the NN class:

```
template <class T>
class NN {

    // The non-null type
    // Note that any reasonable compiler will inline all of the methods
    // below. Furthermore since this class has no virtual members, it
    // does not have a virtual table or any other class data. Thus
    // at run-time there is no overhead of using NN<T> instead of T*
    // directly and NN<T> has the same memory footprint as T*.

public:
    // Constructors also serve as "Casts". Note that these are implicit
    // without using an explicit casting operator. It is possible to
    // make these more explicit by using the 'explicit' keyword.
    NN(T* pointer) : p( pointer ? pointer : throw BadNonNullCast() ) {};
    NN<T>& operator=( T* rhs )
    {
        p = rhs ? rhs : throw BadNonNullCast();
        return *this;
    }

    // Copy constructor and assignment operator for NN<T>
    NN(const NN<T>& other ) : p(other.p){};
    NN<T>& operator=( const NN<T>& rhs )
    {
        p=rhs.p;
        return *this;
    }

    // Copy constructor and assignment operators for subtypes of NN<T>
    template<class S> NN<T>(const NN<S>& other) : p(other.p){}
    template<class S> NN<T>& operator=(const NN<S>& other)
    {
        p=other.p;
        return *this;
    }
```

```
    // Conversion to regular pointer
    operator T*() const { return p; }

    // Dereferencing operators
    T* operator->() const {return p;}
    T& operator*() const {return *p;}

    // Get access
    template<class U> friend class NN;

private:
    // The stored pointer
    T* p;
};
```

A built-in non-null type system has many advantages:

- Its syntax is easier to write and read.
- It can be better integrated with the compiler to perform additional analysis, e.g. dataflow analysis.
- Error messages from the compiler show the problem precisely and are much easier to understand than error messages about templates.
- A built-in type system is omni-present in the entire program and easily enforcible. The template solution only works if programmers always use the `NN` class, but there is no way to enforce this. On the other hand this could also be considered a weakness - the C++ solution gives the programmer flexibility in choosing when to work with non-null types and when not to.

A benefit of our C++ implementation of non-null types is that we can do it in an existing mainstream language and we can use it with minimal effort, without switching languages or tools.

## Task 3

Consider the following three classes (declared in the same package):

```
public class Person {
    Dog? dog; // people might have a dog

    public Person() { }
}

public class Dog {
    Person! owner; // Dogs must have an owner
    Bone! bone;    // Dogs must have a bone
    String! breed; // Dogs must have a breed

    public Dog(Person owner, String breed) {
        this.owner = owner;
        this.bone = new Bone(this);
        this.breed = breed;
    }
}

public class Bone {
    Dog! dog;         // Bones must belong to a dog..

    public Bone(Dog toOwn) {
        this.dog = toOwn;
```

```
    }
}
```

- Annotate the code with non-null and Construction Type annotations where they are necessary. Explain why the code now type-checks according to Construction Types.
- Could we provide constructors for classes `Dog` and `Bone` with no parameters?

Now, suppose a (possibly mad) scientist wants to extend the implementations of these classes with some genetic engineering. Firstly, we want to be able to "clone" a bone. We can add the following method to class `Bone` to make a copy of an existing bone, and assign it to another `Dog`:

```
public Bone clone(Dog toOwn) {
      return new Bone(toOwn);
    }
```

However, our scientist would like to go further, and be able to clone dogs. A cloned `Dog` should also have its bone cloned along with it, but may be assigned to a new owner: we add the following extra constructor and method to class `Dog`:

```
Dog(Dog toClone, Person newOwner) {
    this.owner = newOwner;
    this.breed = toClone.breed;
    this.bone = new Bone(this);
}

public Dog clone(Person toOwn) {
    return new Dog(this, toOwn);
}
```

However, our scientist would like to go still further, and be able to clone people. A cloned `Person` should also have its dog (if any) cloned along with it: we add the following extra constructor and method to class `Person`:

```
Person(Person toClone) {
    Dog? d = toClone.dog;
    if(d!=null) {
        this.dog = new Dog(d, this);
    }
}

public Person clone() {
    return new Person(this);
}
```

- Annotate this extra code with appropriate non-null and Construction Types annotations. You should guarantee that each of the `clone` methods (which belong to the public interface) return a committed reference. You should ensure that your answers guarantee that all of the code type-checks - explain your choices.

  Hint: think carefully about how constructor calls are typed, and what happens if the constructors are called in more than one situation.

**Solution**

Here are the annotations for the first version of the code:

```
public class Person {
    Dog? dog;    // people might have a dog

    public Person() { }
}
```

```
public class Dog {
   Person! owner; // Dogs must have an owner
   Bone! bone;       // Dogs must have a bone
   String! breed; // Dogs must have a breed

   public Dog(unc Person ! owner, unc String ! breed) {
      this.owner = owner;
      this.bone = new Bone(this);
      this.breed = breed;
   }
}

public class Bone {
   Dog! dog;         // Bones must belong to a dog..

   public Bone(unc Dog ! toOwn) {
      this.dog = toOwn;
   }
}
```

Note that we choose the parameter to the construction of `Bone` to be unclassified - since it is public then it probably should be callable with a committed parameter from client code, but it is also called inside the body of the constructor of `Dog`, with a free parameter. Note that the returned reference from these two kinds of call will be different - committed in the former case, and free in the latter. For the `Dog` constructor, we can also choose to make the parameters unclassified. Although in this case we do not directly need to permit "free" arguments being passed to the constructor, we may as well be as permissive as possible. In general, if it is possible to type a constructor body using "unclassified" argument types then this should be the preferred choice of signature as it is the most permissive. Note that the same does not apply for method signatures, since any overriding method definitions are then also be forced to cope with unclassified arguments, which may be much less convenient than using committed ones.

It isn't reasonable to have constructors for `Dog` and `Bone` without parameters, since we need some way of initialising their non-null fields. Although it would be possible to do this by calling e.g., the `Person` constructor from the `Dog` constructor, this doesn't seem very intuitive (nor would it be easy to establish the intuitive invariants of the code - that a `Dog`'s owner refers back to the same `Dog`, etc.). In particular, if all of the constructors need to take no parameters, they would need to call each other infinitely. This is because, we can't set up a cyclic object structure without some kind of mutual initialization (in this case we can only build an infinite object structure to satisfy the non-null requirements of all the objects).

Here is the fully annotated code for the cloning case:

```
public class Person {
   Dog? dog;   // people might have a dog

   public Person() { }

   Person(Person! toClone) {
      Dog d? = toClone.dog;
      if(d != null) {
         this.dog = new Dog(d, this);
      }
   }

   public Person! clone() {
      return new Person(this);
   }
}
public class Dog {
```

```
    Person! owner; // Dogs must have an owner
    Bone! bone;       // Dogs must have a bone
    String! breed; // Dogs must have a breed

    public Dog(unc Person ! owner, unc String ! breed) {
        this.owner = owner;
        this.bone = new Bone(this);
        this.breed = breed;
    }

    Dog(Dog! toClone, unc Person! newOwner) {
        this.owner = newOwner;
        this.breed = toClone.breed;
        this.bone = new Bone(this);
    }

    public Dog! clone(Person! toOwn) {
        return new Dog(this, toOwn);
    }

}
public class Bone {
    Dog! dog;         // Bones must belong to a dog..

    public Bone(unc Dog ! toOwn) {
        this.dog = toOwn;
    }

    public Bone! clone(Dog! toOwn) {
        return new Bone(toOwn);
    }
}
```

Note that all parameters to the new constructors and methods need to have non-null type annotations, since they are each either dereferenced, used to initialize non-null-declared fields or passed on as further parameters to calls that require non-null parameters.

The `toClone` parameter of the new constructor of `Person` needs to be a committed parameter, otherwise when we dereference `toClone.dog` we will obtain a an unclassified value, which will not be suitable to use as a parameter for the new `Dog` constructor.

The `toClone` parameter of the new constructor of `Dog` needs to be a committed parameter, since when a field is read from it, we need to obtain a result with a non-null type. However, the `newOwner` parameter of the new constructor of `Dog` needs to be an unclassified parameter. This is because this parameter is sometimes supplied from a free reference (in the new constructor of `Person`), and sometimes from a committed reference (in the clone method of `Dog`).

For similar reasons, the `toOwn` parameter of the constructor of `Bone` needs to be an unclassified parameter (as was suggested for the previous part of the question). This is because this parameter is sometimes supplied from a free reference (in the new constructor of `Dog`), and sometimes from a committed reference (in the clone method of `Bone`).

This is an important usage of the unclassified types in the Construction Types system - they are useful for constructors which get called sometimes with free and sometimes with committed parameters. Recall that the type of a new expression is determined from the static types of the actual parameters at a particular call, and not from the formal parameters in the constructor signature. For example, in the `clone` method of the `Bone` class, the new expression `new Bone(toOwn)` is given a committed type because the actual parameter `toOwn` has a static type which is committed, despite the fact that the constructor argument type is declared as unclassified in its signature. This means that the same constructor can produce committed/free results

depending on the particular arguments provided in each call (`new` expression). In particular, the return type of the `clone` method can be a committed reference, as required in the question (the same applies to all of the clone methods in the code, since they each call constructors with only committed arguments).

## Task 4

In the Construction Types system, when we read from the field of an expression of committed type, we obtain a reference of committed type, i.e., if $e_1$ has a committed type then $e_1$`.f` has a committed type. Similarly, if $e_1$ has an unclassified type then $e_1$`.f` has an unclassified type. However, if $e_1$ has a free type then $e_1$`.f` does not have a free type, but instead has an unclassified type. Explain why the alternative choice would be unsound (given the existing rules of the system), giving an example of what would go wrong.

**Solution**

Because anything (in terms of Construction Type annotation) can be stored in the fields of a free reference, when we read something back out of such a field we cannot make any guarantees about what is stored there. In particular, it is possible to store a committed reference in the field of a free reference, and if we could then read it back as free, this would be unsound. For example, the following code would type-check:

```
public class C {
   C! f, g;
   public C(C! x) {    // x is committed, this is free
      this.f = x;      // assigning free to committed - ok
      this.f.f = this; // this.f free(?), so this would be ok
      this.g = x.f.g;  // what happens here?
   }
}
```

## Task 5

With non-null types, any class type `T` can be annotated to explicitly declare non-nullity (`T!`) and possible-nullity (`T?`). In the Construction Types system, further variants of these types are introduced, for "free", "committed" (the default), and "unclassified" (`unc`) types. These types are all treated differently by the type system taught in the lectures.

- Explain at least one difference between the treatments of a reference of type `T!` and a reference of type `T?` , giving an illustrative example.
- Explain at least one difference between the treatments of a reference of type `free T!` and a reference of type `unc T!` , giving an illustrative example.
- Explain at least two differences between the treatments of a reference of type `T!` (a committed reference) and a reference of type `unc T!` , giving illustrative examples.
- Explain at least three differences between the treatments of a reference of type `T!` and a reference of type `free T!`, giving illustrative examples.

**Solution**

For all examples below, let us suppose that class `T` has the following field declarations:

```
T! f;
T? g;
```

- If `x` is a reference of type `T!` then `x.f` is a permitted field read (without any if-checks /dataflow analysis), but if `x` is a reference of type `T?` then it is not.

  Also, `x` can only be assigned to the `f` field of an object in the former case and not the latter (`T!` is a subtype of `T?` but not vice versa).

- Suppose `y` is a reference of type `free T!`. If `x` is also a reference of type `free T!` then `x.f = y;` is a permitted field update, but if `x` is a reference of type `unc T!` then it is not.

  Also, `free T!` is a subtype of `unc T!` but not vice versa.

- If `x` is a reference of type `T!` then `x.f.f` is a permitted field read, since `x.f` also has the type `T!`. But if `x` is a reference of type `unc T!` then it is not permitted, since `x.f` has the type `unc T?`.

  If `y` is a further reference of type `unc T!` then `y.f = x` is allowed when `x` has the type `T!` but not when `x` has the type `unc T!`.

  Also, `T!` is a subtype of `unc T!` but not vice versa.

  Furthermore, a constructor call `new C(x)` will be given a committed type if `x` is committed, but instead a `free` type if `x` is unclassified.

- If `x` is a reference of type `T!` then `x.f.f` is a permitted field read, since `x.f` also has the type `T!`. But if `x` is a reference of type `free T!` then it is not permitted, since `x.f` has the type `unc T?`.

  If `y` is a further reference of type `unc T!` then `y.f = x` is allowed when `x` has the type `T!` but not when `x` has the type `free T!`.

  Similarly, `x.f = y` is allowed when `x` has the type `free T!` but not when `x` has the type `T!`.

  Furthermore, a constructor call `new C(x)` will be given a committed type if `x` is committed, but instead a `free` type if `x` is `free`.

## Task 6

*(From a previous exam)*

Consider the following code in a Java-like language enriched with the non-null types system of the course:

```
class Node
{
   int depth;
   public Node! parent;
   public Node! left;
   public Node! right;

  Node(int d)
  { ... }

  ...
}
```

The constructor shown above, when invoked with a positive integer, as in

```
new Node(d)
```

must create a complete binary tree (type `Node!`) of depth `d` containing exactly $2^{d+1} - 1$ nodes. The root node has depth 0. The `depth` field of every node in the constructed tree must be initialized to the depth of that node in the tree. The `parent` field of the root node should point to the root node itself. Similarly the `left` and `right` fields of leaf nodes should point to the leaf nodes themselves.

(a) Write the body of the constructor. You may introduce other constructors and methods. Make sure that you adhere to the rules of the non-null types system including construction types.

(b) Consider the following method:

```
void foo(unc Node! o)
{
   unc Node! x = new Node(2);
   free Node! y = new Node(2);
   Node! z = new Node(2);
   o.right = new Node(2);
}
```

There are four assignments in the body of this method. Which of them type-check? Which of them do not? Why?

**Solution**

(a) Here is a possible implementation

```
Node(int d)
{
   depth = 0;
   parent = this;
   if(d == 0) {
      left=this;
      right=this;
   } else {
      left = new Node(d, 1, this);
      right = new Node(d, 1, this);
   }
}

Node(int goal, int d, free Node! p)    // can be unc Node!
{
   depth = d;
   parent = p;
   if(d == goal) {
      left=this;
      right=this;
   } else {
      left = new Node(goal, d+1, this);
      right = new Node(goal, d+1, this);
   }
}
```

(b) The type of `new Node(2)` is committed. This can be shown trivially, because no references are passed to the constructor.

The fourth assignment is allowed. By the rules for assignments to fields, we know that a committed reference can be assigned to non-null fields of unclassified, free and committed objects.

From the assignments to local variables, the second one is not allowed because it violates the subtyping rules. The other two are allowed.

## Task 7

Consider the following Java classes:

```
public class A {
   public static final int value = B.value + 1;
}
```

```
public class B {
   public static final int value = C.value + 1;
}

public class C {
   public static final int value = A.value + 1;
}
```

Will these classes compile? If not, how could we modify them so that they do?

What would the output of running the following program be?

```
public class Program {
   public static void main(String[] args) {
      System.out.println(A.value);
      System.out.println(B.value);
      System.out.println(C.value);
   }
}
```

In what ways can you change the output of the program by reordering the statements?

**Solution**

The classes will compile. When the program is run, the output will be:

```
3
2
1
```

This is because, starting to initialize `A` causes `B` to start being initialized which causes `C` to start being initialized (at which point Java realizes `A` has already started initialization and just carries on initializing `C`). When `C.value` gets assigned, `A.value` still contains the default value 0.

The class we first mention will always get loaded first, and so complete initialization last. By changing the order of the second two classes, we can vary the output between the one above, and:

```
3
1
2
```