Exercise 12

Further Object Initialisation, and Class Initialisation

In this course, we simplified the treatment of initialisation in the non-null type system (the "raw" types presented in the course are a simpler approach than that which is taken in the literature and in the implementation). Unfortunately, it turns out that we over-simplified, and we need to add some extra restrictions to make the type system sound. In particular, we need to be more restrictive about which field updates are permitted by the type system. Question 2 of this sheet shows where the problem is, while question 3 shows how soundness can be restored.

The phrase, "a raw type" means a type of the form `raw C!` or `raw C?`.
Correspondingly, "a non-raw type" means a type of the form `C!` or `C?`.
Similarly, "a non-null type" means a type of the form `C!` or `raw C!`.

1. In slide 30 of lecture 8, the rules for calculating the appropriate non-null and raw type for **reading** a field expression `e.f` is defined: assuming `f` to be a field of reference type (not a primitive type, for which non-null and raw types do not apply), the rules are:
   - `e.f` is of a raw type ⇔ `e` is of a raw type.
   - `e.f` is of a non-null type ⇔ `e` is of a non-raw type and `f` is declared with a non-null type.

   Using these rules, fill in the following table, detailing the inferred type of `e.f` in each case (supposing that the class type of `e` is `C` and the class type declared for `f` is `D`).

Type of `e`	Declared type of `f`	
	`D!`	`D?`
`C!`		
`C?`		
`raw C!`		
`raw C?`		

2. In various discussions and examples so far, we have allowed the assignment of expressions of raw types to the fields of other expressions of raw type. That is, we have been assuming that it is ok for a field assignment $e_1.f = e_2$ to be permitted when e_1 and e_2 both have raw types. This certainly adds flexibility to the system, when initialising many objects at once. On the other hand, if e_2 has a raw type and e_1 has a non-raw type, we do not want to allow an assignment $e_1.f = e_2$
   - Explain carefully why not – you may wish to refer to the following example code:

```
public class C {
  C! f, g;

  public C(C! x) {
    x.f = this;
    this.f = x.f.f; // what happens here?
  }
}
```

   - In the example code above, what guarantees is the type system supposed to make about the access to `x.f.f`?

Having established that we should not permit `e1.f = e2` when `e2` has a raw type and `e1` has a non-raw type, let us consider the possibility of allowing such an assignment when *both* `e1` and `e2` have a raw type. Recall (lecture 8.2, slide 26) the subtyping relationships which exist between raw and non-raw types.

- In general, what are the requirements for a subtype relationship to satisfy behavioural subtyping (lecture 2.3, exercises 2 and 3)?
- Why do these requirements suggest that we should not permit the assignment `e1.f = e2` when both `e1` and `e2` have a raw type?

Consider the following variant of the example above:

```
public class C {
  C! f, g;
  public C(C! x) {
    this.f = x;
    this.f.f = this; // should this be allowed?
    this.g = x.f.g;  // what happens here?
  }
}
```

- What will happen if we run this example?
- Show that the example can be type-checked if we allow assignments `e1.f = e2` when both `e1` and `e2` have a raw type.

3. (NOTE: this question is hard, and we will spend a fair amount of time on it in the session. Try to do it, or at least think about the issues, but don't feel bad about skipping to the rest of the sheet if you need to. Either way, you should read the statement in italics at the end of this question, which is a new restriction for the type system.)

The previous question demonstrates that it is (in general) unsound to assign a raw reference to the fields of an object referred to by another raw reference. In particular, even if the reference to the object is raw, at the same time there might exist other non-raw references to the object, which may assume that the object is fully initialised.

- Suppose that we have a reference to an object (not null) with a raw type. Suppose that we somehow also know that **all** current references to this object have raw types. As the program continues to execute, at what point might it become possible for a non-raw reference to the object to exist?
- If we could somehow know that **all** current references to an object have raw types, would it be safe to permit the assignment of a raw reference to the object's fields? In particular, in the following code, is the execution of class `D`'s constructor always safe? What about `C`'s constructor?

```
public class C {
  D! f;
  public C(raw D! x) {
    x.f = this; // safe?
    this.f = x; // safe?
  }
}
```

```
public class D {
  C! f;
  public D() {
    C! z = new C(this);
    this.f = z;
  }
}
```

From now on, let's assume that we have restricted the type system *somehow* to guarantee that a field assignment `e1.f = e2` will never be permitted when `e2` has a raw type and a non-raw reference *might exist* which refers to the same object `e1` does

(a way to guarantee this property in practice is shown at the end of the question).

- Suppose also that the constructor for a particular object has not yet finished executing (i.e., we are currently executing either the body of the constructor, or the body of some other method/constructor which was called from it). Is it safe to assume that all current references to the object are references with raw types?

In general, we cannot check modularly whether the constructor for an object has finished executing or not. In particular, when we have a raw reference to an object, we usually have no way of knowing whether that object's constructor is currently executing or not. The only time we can know for sure that an object's constructor has not yet been completely run is inside the body of the constructor itself.

- Using your previous answers, explain why the following rule for restricting the permitted assignments of raw references to fields, fixes the unsoundness shown in Question 2:

  "If e_2 *is of a raw type, then* e_1`.f = `e_2 *is only allowed when the field assignment is in the body of a constructor and* e_1 *is the expression* `this`*"*

  From now on, this restriction is adopted for the raw/non-null type system presented in the lectures.

4. Consider the class `MarriedPerson` from exercise 4 of the previous exercise sheet.
   - Even with appropriate "raw" annotations, this class will not type-check according to the newly-restricted rule for field assignment introduced in the previous question (as quoted in italics). Explain why not.
   - Write a new version of the class `MarriedPerson` which is acceptable according to the new version of the type system, by removing the method `setSpouse` and replacing the constructor with two separate constructors, one of which takes no parameters, and one of which takes a single non-null parameter. Use raw and non-null annotations where necessary.

5. The Java approach to static (class) initialisation is to permit `static` blocks, defining code to be executed when the class is initialised. A class begins its initialisation immediately after it is loaded, which can be triggered by various criteria (see slide 53 of lecture 8.3). The C# approach is similar. Because the `static` block can contain unrestricted Java code, it is possible that executing a `static` block triggers the loading of other classes. In this case, execution of the current `static` block will be postponed, and the `static` block for the new class executed first. The exception is this rule is that if initialisation for the new class has already been started, the "trigger" is ignored (to avoid cycles), and the previous class continues with its initialisation.

   Bearing in mind this semantics, consider the following questions:
   - One criterion to trigger the loading of a class is an access to a static field or method of the class. Given this criterion, is it safe for the code in the body of a static method to assume that that code of the class' `static` block has already been executed (i.e., class initialisation has already taken place)?
   - Another criterion to trigger class initialisation is an attempt to create a new instance of a class. Given this criterion, is it safe for the code in the body of a constructor

to assume that the class initialisation has already taken place? What about code in the body of instance methods of the class?

- A further triggering criterion is that initialisation of a superclass will be triggered by an attempt to initialise a subclass. Given this criterion, is it safe for code in the `static` block of the subclass to assume that the superclass initialisation has already taken place? What about code in the bodies of instance methods in the subclass?

6. 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?