Exercise 12 Solutions

Further Object Initialisation, and Class Initialisation

1.

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

Note that a field access on a possibly-null receiver is always rejected by the type system. If we take the view that the dataflow analysis which the type-checker performs really changes the types of expressions when it can infer them to be non-null, then this is fine. Note that this also means that nested fields accesses on a raw receiver (e.g., `e.f.f` when `e` is of raw type) are never typeable.

Alternatively, it would be possible to define the cases where `e` has a possibly-null type as exactly the same as the cases when `e` has a non-null type, but still reject the field-lookups in these cases independently of the inferred types. However, there doesn't seem to be a real benefit in defining types for expressions which will always be rejected by the type system.

2.

- For a non-raw reference, we are always allowed to assume that when we access a field, we get back another non-raw reference. In particular, if we look up a non-null field on this new reference, it should be guaranteed to contain a non-null value. This assumption would not be justified if we could store a raw reference in the field of a non-raw reference – fields accessed from the raw reference need not be initialised yet.

  In the example code, we assign the raw reference `this` to the field `f` of an object accessible via the non-raw reference `x`. Now, when we lookup the (supposedly initialised) fields of `x` we actually can access an uninitialized object. In particular, the access `x.f.f` actually evaluates to `null` which is then used to initialise the non-null field `f` of `this`. When the constructor finishes executing, it will have actually failed to properly initialise the `this` object, as its non-null field `f` will still contain `null`.

- Since `x` is a non-raw, non-null reference, the type system states that `x.f` is also a non-raw, non-null reference, as (supposedly) is `x.f.f` as well.

- Behavioural subtyping requires that for every operation which is allowed to be performed on a supertype, the same operation can be performed on a subtype.
- In the first part of the question, we showed that, when e_2 has a raw type, we do not want to permit a field assignment e_1`.f =` e_2 if e_1 has a non-raw type. Since

behavioural subtyping says that whenever we are allowed to perform an operation on a raw type, we must be allowed to perform the operation on the corresponding non-raw type, we should not allow the field assignment when e_1 has a raw type either.

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?
  }
}
```

- When we run the example code, the reference to `x.f.g` will evaluate to null, even though it has a non-null type. This is due to previous line - by allowing the assignment of a raw reference `this` to the field of another raw reference `this.f` we have also made it possible to access an uninitialized object via a non-raw reference `x`. This violates the intentions (and invariants) of the type system.

- We use the type invariants/rules summarised in question 1.
  i. Since `x` is of type `C!` and `C! <: raw C!` we also have that `x` is of type `raw C!`. Then the assignment `this.f = x;` is of the kind we are assuming is allowed, and it also assigns a non-null value to the field, so it type-checks successfully.
  ii. Since `this` is of type `raw C!` and the field `f` is declared of type `C!` then dereferencing `this.f` gives a reference of type `raw C?`. However, since the dataflow analysis can determine that `this.f` was assigned a non-null value on the previous line, we can actually consider `this.f` to have type `raw C!`. Since `this` also has type `raw C!` the second line also type-checks.
  iii. Since `x` is of type `C!` and the field `f` is declared of type `C!` we have that `x.f` is of type `C!`. Since the field `g` is also declared of type `C!` we have that `x.f.g` is of type `C!`. Since `C! <: raw C!` the third field assignment is also allowed.

3. For this question, it is useful to consider two different kinds of objects. Firstly, objects which satisfy the property that *all* current references to them (from anywhere in the program) are raw references. Secondly, those which do not satisfy this property – i.e., at least one non-raw reference exists to the object. For the first kind of object, the type invariants do not imply any requirements about initialisation of the object, while for the second kind it is essential that the object be "fully initialised" – more on this in detail in the next sheet, but essentially the object's fields must be appropriately initialised and so must all other objects reachable from its fields.
   - In the type system presented in the course so far, there are two ways in which a non-raw reference can become available to an object of the kind described. Firstly, if we take a raw reference from which it is possible to reach the object, and assign it to the fields of an object of the second kind above (i.e., an object which can be reached via a non-raw reference), then we create a way of referencing the original

object via a non-raw reference. This is the case which leads to unsoundness, as discussed in the previous question.

Since it is not allowed to down-cast a raw type to a non-raw type (lecture 8.2, slide 26) the only other way we can get hold of new non-raw references to such an object is when constructors finish executing. In particular, if the constructor for the object itself (let's call the object "o") finishes executing, and if the constructor is not declared with any raw parameters, then the reference returned from the constructor has a non-raw type. On the other hand, if o's constructor *does* have raw parameters, the reference returned from it is still of a raw type. In this case, it can be that an enclosing constructor call (which does *not* have raw parameters) may eventually return, creating a new non-raw reference to an object which can reach the object o. In all cases, notice that a non-raw reference to o cannot be created in this way before its constructor has finished executing.

- If it is only possible to refer to an object via raw references, then assigning a raw reference to one of its fields cannot break the type invariants of the system (which only provide strong guarantees about what can be accessed via non-raw references). In other words, assigning to the object's field will only make a difference to what can be reached via raw references, and whether what we assign is initialised or not, nothing can be expected about it via such references.

  In the case of the particular code given, when the constructor of `C` is called from the constructor of `D`, the object referred to by the raw argument is under construction, and no one currently has any expectations about its initialisation (i.e., there are no non-raw references to it). Therefore, whatever is done to its fields in the body of the constructor of `C`, will not create a problem. On the other hand, if the constructor of `C` is called from some other code, passing a fully-initialised object, then the first assignment is questionable – it breaks the type invariant of any existing non-raw references to the argument. In fact, in this particular code, it only breaks such invariants temporarily, since the second assignment will always restore such type invariants. But, if we had a slightly different definition, this could not be guaranteed, e.g.,

  ```
  public class C {
    D! F;
    public C(raw D! x, raw D! y) {
      x.f = y;    // dangerous
      this.f = x; // safe
    }
  }
  ```

  In the case where `x` refers to an object which is meant to be initialised (and non-raw references refer to it) while `y` refers to an uninitialised object, this will break the type invariants of any non-raw references to `x`.

- While an object's constructor is executing, only raw references to the object can exist. When the object's constructor starts, no existing objects can have a reference to the new object. During execution of the constructor, the object cannot be assigned to the fields of other objects which can be reached by non-raw references (this is the assumption we have made about how the system will be restricted), and so new non-raw references cannot be created that way. Any other

objects which are allocated during the execution of the object's constructor can only get references to the object if the object is passed as an argument to another constructor call. But in this case, the reference returned from that constructor call will also be a raw reference – this doesn't create a non-raw reference to the original object either.

- Inside the body of a constructor, the `this` expression is guaranteed to refer to an object which cannot currently be reached by any non-raw reference (by the previous part of the question). Therefore (by the part before that), it is safe to assign raw references to the fields of `this` in this case.

4.

- The body of the `setSpouse` method cannot be type-checked according to the new restriction. In general, it is never allowed for a raw reference to be assigned to the fields of another object during a method (rather than constructor) call. This means there is no hope of writing a different version of the same method which would type-check.
-
```
public class MarriedPerson {
    private MarriedPerson! spouse;
    // invariant: this.spouse.spouse == this;

    public MarriedPerson() {
      this.spouse = new MarriedPerson(this); //invent one!
    }

    public MarriedPerson(raw MarriedPerson! spouse) {
      this.spouse = spouse;
    }
 }
```

5.

- No – here is an example (consider calling `B.bar()` when `A` hasn't been loaded):

```
public class A {
  public static B b;
  public static int x;

  static {
    b = new B();
    x = 1;
  }

  public static void foo() {
    assert x > 0; // safe?
  }
}
```

```
public class B {
  public B() {
    A.foo();
  }

  public static int bar() {
    return A.x;
  }
}
```

- No – here is an example (consider calling `B.bar()` when `A` hasn't been loaded):

```
public class A {
  public static B b;
  public static int x;

  static {
    b = new B();
```

```
public class B {
  public B() {
    A temp = new A();
  }

  public static int bar() {
```

```
    x = 1;
  }

  public A() {
    assert x > 0; // safe?
  }
}
```

```
    return A.x;
  }
}
```

• No – here is an example (consider calling `A.foo()` when neither class is loaded):

```
public class A {
  public static B b;
  public static int x;

  static {
    b = new B();
    b.bar();
    x = 1;
  }

  public static void foo(){}
}
```

```
public class B extends A {
  static {
    assert A.x > 0; //safe?
  }

  public void bar() {
    assert A.x > 0; //safe?
  }
}
```

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

```
3
2
1
```

This is because, starting to initialise `A` causes `B` to start being initialised which causes `C` to start being initialised (at which point Java realises `A` has already started initialisation and just carries on initialising `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 initialisation last. By changing the order of the second two classes, we can vary the output between the one above, and:

```
3
1
2
```