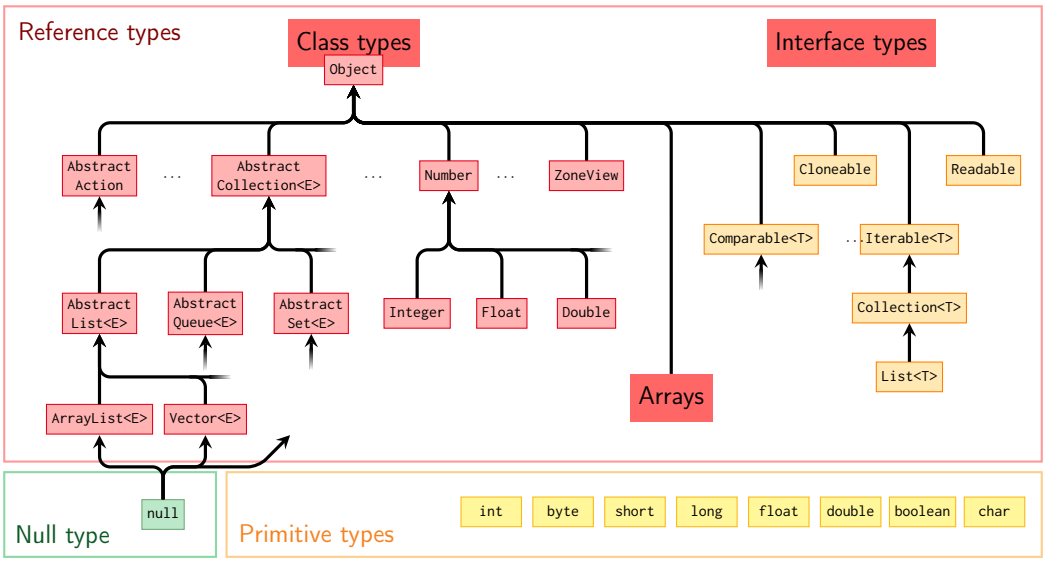
Tutorial #5 - Inclusion polymorphism

Exercice 1: Inclusion polymorphism in Java

The Java type system follows a hierarchy that is described in the language specification (accessible at `http://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html`) and that more or less corresponds to the following diagram :



1. Where are the existential types in this architecture ? The universal types ?
2. What are the subtyping relations between the primitive types ?
   What are the relations between arrays and the other reference types ?
3. Aside from the `Object` type, give two examples of recursive types in the API (i.e types whose definition involves themselves somewhat). The Java classes are listed at `https://docs.oracle.com/javase/8/docs/api/overview-tree.html`.
4. What is the difference, from the point of view of subtyping, between the interface types and class types in Java ?

The Scala programming language is a programming language developed at the EPFL (`http://www.scala-lang.org`), that can be compiled to Java bytecode and run on the Java virtual machine. A key property of the language is the possibility to mix Java and Scala `.class` files together. Its type system is described on this diagram [1].

5. What are the differences between the Scala types and the Java types ?

1. This diagram appears in chap. 11 of *Programming in Scala*, by Odersky et. al.

Exercice 2: Record polymorphism in OCaml

One of the specificities of the OCaml language is its support for objects (the "O" originally standing for "Objective"). The OCaml object layer shares a lot of similarities with the record types studied in the course. For reference, the documentation on objects is reachable at `https://caml.inria.fr/pub/docs/manual-ocaml/objectexamples.html`.

▷ Objects can be defined independently from classes, as *immediate objects* :

```
let o = object (self)
  val mutable x = 0          (* Attribute *)
  method get = x             (* Method *)
  method move d = x ← x + d
end;;
```

The object `o` has the following type :

```
< get : int; move : int → unit >
```

The type of the object is inferred as a *record type*. Notice that the attributes do not appear in the type, and that the methods may have a constant type (such as `get:int`) even though they are not constant. The `(self)` part names the reference to the object inside the code.

Calling a method on an object is done the following way :

```
o#get;;     (* → 0 *)
o#move 2;;  (* → unit *)
o#get;;     (* → 2 *)
```

▷ A *class* is simply a constructor for an object :

```
class point = fun init → object (self)
  val mutable x = init
  method get    = x
  method move d = x ← x + d
end;;
```

```
class point : int → object
  val mutable x : int
  method get_x : int
  method move : int → unit
end
```

Class definitions possess a nominal type (in this case point), whereas immediate objects possess a structural type, but in OCaml they are interchangeable. The creation of an object from a class is done the following way :

```
let o = new point 3;;
```

It is possible to *coerce* a value `v` into a supertype `t` with the expression `v :> t`.

Let us approximate the Object type to the following :

$$\text{Object} ::= \{\texttt{clone} : \text{Unit} \to \text{Object}, \texttt{equals} : \text{Object} \to \text{Bool}, \}$$

1. Create the record type for Object in OCaml, and propose an implementation of the class. *Remark :* both constructions must be recursive.

Let us now consider a more utilitarian class :

2. Create a (recursive) record type olist to represent lists constructed with a `head` and a `tail`. The type should contain 3 methods : `hd`, `tl`, and `is_empty`. *Remark :* for the sake of this exercise, it is not necessary to create a polymorphic list.
3. Write an immediate object that represents the empty list. Its non-trivial functions may return an error using `failwith`.

   Write a class that represents a non-empty list, with the usual head and tail elements passed to the constructor.
4. What would be an example of a subtype of olist ?

Consider now writing an external function dealing with objects of type olist :

5. Write the `length` function for such lists, and analyze its type. Write this type, once with a universal variable and once with an existential variable.
6. Write the type for a dictionary containing pairs (key,value), that is compatible with the `length` function.

Exercice 3: Java selection method

Consider the selection algorithm for the Java 1.1 language that was described in the *Java Language Reference* by Mark Grand (1997) [2]. Given an expression in Java containing a method call, this algorithm selects the appropriate method to apply when multiple choices are possible. It is described at `http://web.deu.edu.tr/doc/oreily/java/langref/ch04_js.htm`, corresponding in the book to the chapter 4, subsection « Method call expression ». The examples from this chapter are given in the sources of this tutorial.

2. The older version of the Java language has a significatively simpler algorithm for method selection than the subsequent versions.

```java
// First class hierarchy A ← B ← C ← D
class A {}
class B extends A {}
class C extends B {}
class D extends C {}

// Second class hierarchy W ← X ← Y ← Z
class W {
    void foo(D d) {System.out.println("W.D");}
}
class X extends W {
    void foo(A a) {System.out.println("X.A");}
    void foo(B b) {System.out.println("X.B");}
}
class Y extends X {
    void foo(B b) {System.out.println("Y.B");}
}
class Z extends Y {
    void foo(C c) {System.out.println("Z.C");}
}

// Main program
public class CallSelection {
    public static void main(String [] argv) {
        Z z = new Z();
        ((X) z).foo(new C());
    }
}
```

```java
// Small class hierarchy A ← B
class A {}
class B extends A {}

// Examples of ambiguous calls
class AmbiguousCall {
    void foo(B b, double x){}
    void foo(A a, int i){}
    void doit() {
        foo(new A(), 8);    // Matches foo(A, int)
        foo(new A(), 8.0); // Error: doesn't match anything
        foo(new B(), 8);    // Error: ambiguous, matches both
        foo(new B(), 8.0); // Matches foo(B, double)
    }
}
```

1. Read the article and describe the three (3) steps of the selection algorithm.
2. Determine in the article the two (2) potential sources of errors that can occur during the selection algorithm.
3. Determine which part of this algorithm deals with overloading (adhoc polymorphism) and which part deals with overriding (inclusion polymorphism).
4. Determine which part of this algorithm is done statically and which part is done dynamically (*Remark* : the answer is different from the previous question)