From ADT to Objects to OOP

- Recall that a type is a set of values and each type is usually associated with some operations.
- New types can be defined in terms of existing ones by *type composition*.
- Some desired data structures may not be defined completely using just type composition.
  e.g. `type Rational = Integer` $\times$ `Integer`
  - The representation type might have values that do **not** correspond to any values of the desired type.
    e.g. $1/0$
  - The representation type might have several values that correspond to the same value of the desired type. *Simple-minded comparisons* would then yield incorrect results.
    e.g. $3/2 \stackrel{\text{vs}}{=} 6/4$ (multiple representation)
  - Values of the desired type can be confused with values of the representation type and other similar types.
    e.g. `type Position = Integer` $\times$ `Integer`
    We may confuse a variable of type `Rational` and another type `Position` in a language that supports *structural equivalence* of types.
- An abstract data type (ADT) is a type defined by a type *representation* and a group of *operations*.

- The set of values of an ADT is defined only *indirectly*; the type consists of all values that can be generated by successive applications of the operations, *starting with some constants*.

A phone directory ADT in **Ada**:

```
package directory_type is
  type Directory is limited private;
  procedure insert(dir: in out Directory;
                   newname: in Name;
                   newnumber: in Number);
  procedure lookup(dir: in Directory;
                   oldname: in Name;
                   oldnumber: out Number;
                   found: out Boolean);

private
  type DirNode;
  type Directory is access DirNode;
  type DirNode is record ... end record;
end directory_type;

package body directory_type is
  procedure insert(dir: in out Directory;
                   newname: in Name;
                   newnumber: in Number) is
  ...;
  procedure lookup(dir: in Directory;
                   oldname: in Name;
                   oldnumber: out Number;
                   found: out Boolean) is
  ...;
  procedure sort(...) is
  ...;
end directory_type;
```

- Typically, the programmer chooses a *representation* for the values of the ADT, and *implements* the operations in terms of this chosen representation.
- The key point is that the representation and the implementation details are *hidden*; the module exports only the abstract type and its operations.
  Note: The word "hidden" does not necessarily imply *invisibility* in the literal sense; *inaccessibility* is good enough.
- The defined "`directory_type`" ADT can be subsequently used to declare new variables. *e.g.*

Using the phone directory ADT in **Ada**:

```
...
use directory_type;
homedir: Directory;
workdir: Directory;
...
insert(workdir,me,6041);
insert(homedir,me,8715);
lookup(workdir,me,mynumber,ok);
...
```

- With an ADT, it does not matter that a given value of the type has several possible representations, because the representations are hidden from the users and the provided operations should *ensure* consistency of values.
- Only desired properties of the values are *observable* using the operations associated with the ADT.

- An ADT's representation and implementation can always be *changed* without forcing any changes outside the module.
- An ADT must generally provide constructor operations for composing values of the type, and destructor operations for decomposing such values when they are no longer needed. That means that our last example is *incomplete*.
- The **C** language does not provide any language constructs for defining ADTs. We can, however, imitate ADT in **C** using header files, the "`#include`" preprocessor directive, "`static`" declarations, separate file compilation, and a peculiarity of "`struct`" and "`typedef`." The phone directory example can be implemented in **C** as follows.

A phone directory ADT in C:

"directory.h"

```
typedef struct _dirNode Dir;
typedef Name *char;
typedef Num int;
typedef Bool int;

extern void insert(Dir *,Name,Num);
extern void lookup(Dir,Name,Num *,Bool *);
```

"directory.c"

```
struct _dirNode {...};
...
void insert(Dir *dir,Name newname,Num newnum) {...}
void lookup(Dir dir,Name oldname,Num *oldnum,Bool *found) {...}
static void sort(...) {...}
...
```

- *Private* procedures/functions are declared as `static` in the implementation (.c) file.
- The users of an ADT are supplied with only the header (.h) file together with the object code of the implementation resulting from separate file compilation.

EXERCISE: What can go wrong with the **C** approach?

- Object is another special and important kind of module that consists of a *hidden variable* together with a group of *exported operations* on that variable.
- The variable is typically a data structure such as a table or database. Being hidden, the variable can be accessed *only* through the exported operations.

A phone directory object in **Ada**:

```
package directory_object is
  procedure insert(newname: in Name;
                   newnumber: in Number);
  procedure lookup(oldname: in Name;
                   oldnumber: out Number;
                   found: out Boolean);
end directory_object;

package body directory_object is
  type DirNode;
  type DirPtr is access DirNode;
  type DirNode is record ... end record;
  root: DirPtr; /* root is hidden */

  procedure insert(newname: in Name;
                   newnumber: in Number) is
  ...;
  procedure lookup(oldname: in Name;
                   oldnumber: out Number;
                   found: out Boolean) is
  ...;
  procedure sort(...) is
  ...;

begin
  ...; /* Code for initializing the directory */
end directory_object;

/* Main program */ ...
directory_object.insert(me,6041);
...
directory_object.lookup(me,mynumber,ok);
...
```

- We can also imitate objects in C!

A phone directory object in C:
"directory.h"

```
typedef Name *char;
typedef Num int;
typedef Bool int;

extern void insert(Name,Num);
extern void lookup(Name,Num *,Bool *);
```

"directory.c"

```
struct _dirNode {...};
typedef struct _dirNode Dir;
static Dir root;
...
void insert(Name newname,Num newnum) {...}
void lookup(Name oldname,Num *oldnum,Bool *found) {...}
static void sort(...) {...}
...
```

- Again, the main program using the object has to *include* the header file and *link* with the object file of the implementation.
- The last example can only create a *single* phone directory object. We would like to be able to create a whole *class* of similar objects.

- In **Ada**, all we have to do is to make the package `generic`.

A phone directory object in **Ada**:

```
generic package directory_class is
  ...
end directory_class;

package body directory_class is
  ...
end directory_class;
```

- To create individual objects, we must *instantiate* the generic package.

A phone directory object in **Ada**:

```
package homedir is new directory_class;
package workdir is new directory_class;
```

- The declarations create two *similar* but *distinct* objects, denoted by "`homedir`" and "`workdir`" respectively, which are instances of the object class "`directory_class`."
- Unfortunately, there is **no** mechanism in **C** to mimic object classes.
- One thing that we learn here is that a language can support the concepts of objects *without* necessarily supporting the concept of object classes.

- The concepts of ADT and object classes have much in *common*: each allows the creation of several variables of a type whose representation is hidden, and to access the variables only by operations provided for the purpose.
- They are also *subtly* different. Let us use the phone directory example to illustrate.
  - With ADT, only *one* procedure "`insert`" is defined, regardless of how many variables of type `Directory` are created.
  - With object classes, several instantiations of the same class define *several distinct* procedures (*e.g.* "`homedir.insert`" and "`workdir.insert`"), each of which accesses a different object, namely "`homedir`" and "`workdir`" respectively.
  - With ADT, we write "`insert(workdir,me,6041)`." Here obviously, the particular directory to be accessed is an *argument* to the procedure.
  - We would write "`workdir.insert(me,6041)`" with object classes. Here the particular phone directory to be accessed is a kind of *implicit* argument, but it is an argument *fixed* when the object is *created*, not when the procedure is called.

- The ADT approach has certain advantages over the object class approach.
  - ADTs are similar to *built-in types*, and defining a new ADT smoothly extends the variety of types available to the programmers.
  - In "most" languages (such as **Ada** and **ML**), values of ADTs are *first-class* values but objects are not.
  - The notation for calling the operations of an ADT is more *natural*.
  - ADTs are useful in *all* programming paradigms supporting the notion of types. Objects, being updatable variables, fits only into an imperative style of programming.
- Nevertheless, objects and object classes are the basis of the important paradigm of *object-oriented programming*, which will be our next topic of discussion.