13. Classes

The outline of this part:

- Classes and objects
- Reference and access control
- Inheritance
- Polymorphism
- Generic classes
- Example of class hierarchy
- Example of using objects in modules
13.1 Classes and objects

Definition 13.1 A class is a user-defined type, which defines a collection of objects with the same features. The features of objects include attributes describing their data resources and operations offering the means for manipulating their data resources and providing functional services for other objects.

Definition 13.2 An object is an instance of a class or a value of a class (remember a class is a type) with an unique identity.
Example:

**Student** is a class that contains a set of specific Students. For example, Mike, Jean, and John are three students of the class; each of them has attributes: *id* (identification number) and *dept* (department) he or she belongs to, and can perform the operations: *study* and *take_exam*. 
Student

id
department
study
take_exam

Class

Mike
id = 001
department = CS
study
take_exam

Jean
id = 005
department = CE
study
take_exam

John
id = 008
department = EE
study
take_exam

Object 1  Object 2  Object 3
13.1.1 Class definition

The structure of a class definition:

```
class ClassName / SuperClassName;
const ConstantDeclaration;
type TypeDeclaration;
var VariableDeclaration;
inv TypeandStateInvariants;
behav CDFD_no;  /* optional for a class */
method Init;    /* method Init is the constructor of the class */
method_1;   /* methods cannot be decomposed into CDFDs */
method_2;
...
method_n;
function_1;
function_2;
...
function_m;
end_class;
```
Example:

class Student;

type
    Record = string * real; /* A record of the type Record is a
        pair: (course title, score). */

var
    id: nat0;
    dept: Dept;
    study_records: set of Record;
    /* All the records are different due to their different id */

inv
    id <= 9999;

method Init ()
post id = 0 and dept = <CS> and study_records = { }
end_method;
method study(course: string)
  ext wr study_records;
  post study_records = union(~study_record,
    {mk_Record(course, 0)})
end_method;

method take_exam(course: string) score: real
  ext wr study_records;
  post (exists[x: nat0] | 0 <= x <= 100 and
    score = x) and
    study_records =
    union(diff(~study_records, {(course, 0)}),
      {(course, score)})
end_method;
end_class;
13.1.2 Objects

An object of a class is instantiated from a class through the operator new.

To hold an object, we need to have a variable of the class declared, like

```plaintext
obj: @C;
```

Where C is a class, presumably defined somewhere else in the specification, and obj is a variable of class C. The "at" mark @ before C is used to indicate that C is a class, not a built-in type.

An object to be held by the variable obj is then created in either the form:

```plaintext
obj := new C;
```

or

```plaintext
obj = new C
```

The first form is used in an explicit specification of a method, while the second one is usually used in a pre or postcondition of an implicit specification for a method. In fact, an object is created by executing the constructor Init of its class.
Example: let us first declare a variable s:

```plaintext
s: @Student;
```

Then we derive an object by applying the new operator to the class Student:

```plaintext
s := new Student;
```

According to the specification of constructor “Init” of the class Student, object “s” satisfies the following properties:

```plaintext
s.id = 0 and s.dept = <CS> and s.study_records = { }
```

Where “s.id” (s.dept, s.study_records) is a reference of the attribute “id” (dept, study_records) of “s”.
13.1.3 Identity of objects

After an object is created, it is assigned to a unique identity distinguishing from other objects, and this identity is sustained throughout the execution of the entire specification.

This is similar to the situation that a student is given a unique identification number after he or she enters a university and the number is kept in use until he or she graduates from the university.

It is sufficient for us just to know that every object is different and kept alive until the termination of the execution of the entire system.
13.2 Reference and access control

Let

\[ \text{obj: @}C; \]

where class C is defined as:

\begin{verbatim}
class C;
  var
    a_1: T1;    a_2: T2;    ...    a_n: Tn;    m_1(...);
    m_2(...);
    ...
    m_q(...);
end_class;
\end{verbatim}
Then the attributes and methods of object “obj” can be referenced in the forms:

\[ \text{obj.a}_i \]

and

\[ \text{obj.m}_j(...) \]

Where \( i = 1..n \) and \( j = 1..q \).
Example:

class A;
var
    s: @Student;
method Init()
ext wr s
explicit
    s := new Student;
end_method;
method Check_Score(course: string)
    exam_score: nat0
    exam_score := s.take_exam(course);
end_method;
end_class;
13.3 Inheritance

13.3.1 What is inheritance?

Inheritance is a mechanism for building new classes based on existing classes, and therefore allow the reuse of attributes and behaviors (methods) of some classes by some other classes.
Example: class "Student_with_Scholarship" inherits from class "Student":

class Student_with_Scholarship / Student;
    var
        scholarship: int; /* amount of the money provided by the scholarship */
    method Init()
        post scholarship = 0
    end_method;
end_class;
13.3.2 Superclasses and subclasses
In general, if class B inherits from class A, we define B in the form:

\[
\text{class B / A;}
\]

\[
\text{...}
\]

\[
\text{end\_class;}
\]

Thus, class B inherits all the attributes and methods defined in class A.
A class inheritance hierarchy: notice we do not allow multiple inheritance in SOFL for two reasons: one is to avoid possible name conflicts, and another is to facilitate verification of SOFL specifications.
13.3.3 Method overloading

Method overloading is a way to define different methods with the same name. SOFL allows several methods of the same name to be defined as long as these methods have different sets of parameters (based on the number of parameters, the types of the parameters, and the order of the parameters).
class Exam;

... 

method square(x: int) res: int
post res = x ** 2
end_method;

method sequare(x: real) res: real
post res = x ** 2
end_method;
end_class;
13.3.4 Method overriding
Method overriding is a mechanism that allows a subclass to redefine methods of its superclass while sustaining their interfaces.
Example:
class A;

... 
method m1(x: int) y: int
post y ** 2 = x
end_method;

method m2()
...
end_method;
end_class;
class B / A;
...
method m1(x: int) y: int
post y ** 2 = x and y > 0
end_method;
end_class;

If
    obj: @B;
    obj := new B;
then,
    obj.m1(5)
will refer to the method “m1” defined in class B. However, if “m1” is not redefined in class B, “obj.m1(5)” will refer to the method m1 defined in class A.
13.4 Polymorphism

Polymorphism is a mechanism by which a single method or attribute variable may be defined upon more than one class and may take on different implementations in each of those classes. It is usually implemented dynamically on the basis of inheritance of classes.
Example: let us take the classes A and B declared previously as an example, where B is a subclass of A. Suppose method “d” is defined in class D, as

class D;
  ...
  method d(x: @A)
     explicit
     x.m1(5); /* the m1 of B and A may be functionally different. */
  ...
  end_method;
  ...
end_class;

then when method d is invoked, its input variable x can be bound to objects of either class A or class B. This is because objects of B are objects of A due to the inheritance relationship between B and A.
However, if the signature of method d is changed to:

```ruby
method d(x: @B)
explicit
  x.m1(5);
...
end_method;
```

since objects of class A are not regarded as objects of class B, invoking method d with an object of class A will be disallowed.
An operator known as downcast allows to convert an object of a class to that of its subclass. The downcast operator is given in the form:

\( (\text{className}) \)

Where “className” is the name of the class to be converted to.

When applying this operator to an object, say \( x \) given in the previous example, of class \( B \), the object \( x \) is converted into an object of class \( B \). For example,

\( (B) \ x; \)

Thus, the invocation of methods defined in class \( B \) can be carried out by means of object \( x \).
To avoid any potential confusion in interpreting expressions involving the downcast operator, we specify that the downcast operator has the highest priority of application. For example, the method invocation:

\[(B) \ x.\text{m1}(5);\]

means that the downcast operator \( (B) \) is first applied to object \( x \), and then the method \( \text{m1} \) of the resulting object is invoked. A clearer expression may be obtained by using parentheses, like

\[((B) \ x).\text{m1}(5);\]
A class is a generic class if it allows parameters that will be bound to types (or type identifiers). The parameters are used as types to declare variables within the class definition, and must be bound to specific types when variables are declared using the class.

Let A be a generic class with type parameter T. Then A is declared in the form:

```plaintext
class A[T];
...
end_class;
```
For example, the type parameter T is used to declare a state variable s1 and the input parameter of method m:

```java
class A[T];
var
 s1: seq of T;
method m(s: set of T)
  ...
end_method;
...
end_class;
```

Semantically, a generic class represents a mapping from a set of types to a set of classes. Consider class A[T] as an example, it is defined as:

\[ A[\_]: \text{Types} \rightarrow \text{Classes} \]

Where Types is a set of types, and Classes is a set of classes.
When a variable “x” is declared with class A, it is necessary to specify a concrete type, say real, for T. Thus a declaration of “x” can be:

\[
x: \texttt{@A[real]};
\]

The state variable “s1” in class A then becomes a variable of type seq of real, and the parameter “s” will be bound to values of type set of real.
The principle of generic class described above can be extended to multiple type parameters of classes. That is, we allow a class, say B, to be declared in the form:

```plaintext
class B[T_1, T_2, ..., T_n];
...
end_class
```

Thus, a variable “y” of class B may be declared as follows:

```plaintext
y: @B[int, real, set of nat0]
```
We give an example of building classes **Point**, **Circle**, and **Cylinder** to show how to build an inheritance hierarchy. Since a point consists of coordinates x and y, they should be declared as the attributes of class **Point**.
class Point;
  var
    x, y: int;
  method Init()
    ext wr x, y
    post x = 0 and y = 0
  end_method;

  method Set_Point(a: int, b: int)
    ext wr x, y
    post x = a and y = a
  end_method;
end_class;
Next we define Circle as a subclass of Point:

class Circle extends Point;
const
  PI = 3.14;
var
  radius: real; /* radius of Circle */
method Init()
  wrradius posradius = 0.0
end_method;

/* Since class Circle inherits from class Point, there is no need to carry out explicitly an initialization of the inherited attributes x and y in the specification of the constructor Init of class Circle, because it is assumed to be implicitly done through the constructor Init of the super class Point by SOFL semantics. */
method Set_Radius(r: real)
  ext wr radius: real
  pre r >= 0.0
  post radius = r
end_method;

method Create_Circle(x1, y1: int, r: real)
  ext wr x, y: int
    wr radius: real
  pre r >= 0.0
  explicit
    Set_Point(x1, y1);
    radius = r;
  comment
    This method is defined with a mixed specification of implicit and explicit style. The precondition requires that parameter r be greater than 0.0, while the explicit specification defines how to create a circle.
end_method;
method Compute_Area() area: real
ext rd radius
post area = PI * radius ** 2
end_method;
end_class;

As a subclass of Circle, the class Cylinder is defined as follows:
class Cylinder / Circle;

var
  height: real; /* height of Cylinder */
method Init()
  ext wr height
  post height = 0
end_method;

method Set_Height(h: real)
  ext wr height
  pre h >= 0.0
  post height = h
end_method;
method Create_Cylinder(x1, y1: int, r, h: real)
  ext wr x, y: int
    wr radius: real
    wr height
  pre r >= 0.0 and h >= 0.0
  explicit
  begin
    Set_Point(x1, y1);
    Set_Radius(r);
    Set_Height(h)
  end
end_method;
method Compute_Area() area: real
   /*surface area of Cylinder */
ext rd radius: real
post area = Circle.PI * radius ** 2
   /*the constant PI defined in the class Circle is used. */
end_method;
method Compute_Volume() vol: real
   /*surface area of Cylinder */
ext rd height: real
explicit
   vol := Computer_Area() * height;
   /* a method of the superclass is invoked. */
end_method;
end_class;
13.7 Example of using objects in modules

The example presented in this part aims to explain how classes, objects, and their methods are used in building modules, including CDFDs and process specifications.

We want to build a module whose CDFD describes the behavior of creating an object of Cylinder and carrying out some interesting evaluations.
The CDFD of the module:

- Create_Point
- Cylinder
- Display_Error_Message
- Evaluate_Area_Volume
- Draw_Cylinder
- x
- y

Goto 1
Goto 2
module Cylinder_Test;
var
cylinder: @Cylinder;
/*class Cylinder is assumed to have been defined somewhere else.*/

process Init()
ext wr cylinder
explicit
cylinder := new Cylinder
comment
This process initializes the only store variable cylinder.
end_process;
process Create_Point(x, y: int) point: @Point explicit begin
    point := new Point;
    point := point.Set_Point(x, y)
end

comment
A point is created by means of the invocation of method Set_Point of object point.
end_process;
process Create_Cylinder(point: @Point, r, h: real)
    err: string | goto: sign

ext wr cylinder
explicit
    if r >= 0.0 and h >= 0.0
    then
        begin
            cylinder := cylinder.Set_Cylinder(point.x, point.y, r, h);
            goto := !; /* making goto available */
        end
    else
        err := "The radius or height of the cylinder is negative"
comment This process creates an object cylinder of class Cylinder by invoking method Set_Cylinder of cylinder based on the inputs: point, r, and h.
end_process;
process Display_Error_Message(err: string)
post Display(err)
comment
   The error message err is displayed on an output device by means of the function application Display(err). The Display function is defined later in this module.
end_process;
process Evaluate_Area_Volume(goto1: sign)
    area: real, vol: real
ext rd cylinder
explicit
begin
    area := cylinder.Compute_Area();
    vol := cylinder.Compute_Volume()
end
comment
    The area and vol are generated by assigning the results of invoking the methods Compute_Area and Compute_Volume of the object cylinder, respectively.
end_process;
process Draw_Display(goto2: sign)
ext rd cylinder
post Draw(cylinder)
comment
   The drawing of cylinder is done by the function application Draw(cylinder). The function Draw is defined below.
end_process;
/* Display is intended to display the given error message on an output device */
function Display(meg: string): bool == undefined
end_function;

/* Draw is intended to draw the given cylinder on an output device */
function Draw(cylinder: @Cylinder): bool == undefined
end_function
end_module.
Exercise 13

1. Answer the questions:
   a. what is a class?
   b. what is an object?
   c. what is inheritance?
   d. what is superclass and subclass?
   e. what is polymorphism?
   f. what is a generic class?

2. Define the class Polygon as the superclass of the classes Triangle, Rectangle, and Diamond. Define an attribute variable area and a method Get_Area in each of the classes, but with different specifications, depending on the specific shapes.

3. Specify a module whose CDFD creates a required shape that can be one of the objects of the classes Triangle, Rectangle, and Diamond, and compute its area.