CENG 242 - Chapter 6: Data Abstraction (Encapsulation)

Burak Metehan Tunçel - May 2022

1 Program Units, Packages and Encapsulation

A ***program unit*** is any named part of a program that can be designed and implemented more-or-less independently. A well-designed program unit has a *single purpose*, and has a *simple application program interface*. If well designed, a program unit is likely to be modifiable (capable of being changed without forcing major changes to other program units), and is potentially reusable (capable of being used in many programs).

The ***application program interface*** (or ***API***) of a program unit is the minimum information that application programmers need to know in order to use the program unit successfully.

For instance, a procedure's API consists of its *identifier*, *formal parameters*, and *result type if any*, together with a *specification of its observable behavior*. The procedure's API does *not* include the algorithm used in its definition.

In large-scale program construction, complexity grows, and reusability and abstraction become important. For example,

50 lines	no abstraction is essential, all in main()
500 lines	function/procedure abstraction sufficient
5,000 lines	function groups forming modules, modules are combined to form the application
500,000 lines	heavy abstraction and modularization, all parts designed for reuse (libraries, components etc)

Therefore, a programming language whose only program units are procedures (such as `C`) is suitable only for small-scale program construction. For large-scale program construction, the language should support large-scale program units such as *packages*, *abstract types*, and *classes*.

1.1 Packages

A ***package*** is a group of several components declared for a common purpose. In other words, a group of declarations put into a single body.

These components may be *types*, *constants*, *variables*, *procedures*, or indeed *any entities* that may be declared in the programming language.

`C` has indirect way of packaging per source file. `Python` defines modules per source file. `C++` has namespaces.

```
namespace Trig {
  const double pi=3.14159265358979;
  double sin(double x) { ... }
  double cos(double x) { ... }
  double tan(double x) { ... }
  double atan(double x) { ... }
  ...
};
```

Code 1

These functions can be used as

```
Trig::sin(Trig::pi/2+x) + Trig::cos(x)
```

In `C++`, (::) is *Scope* operator. In this way, *identifier overlap is avoided*. There is no name collisions in `List::insert(...)` and `Tree::insert(...)`.

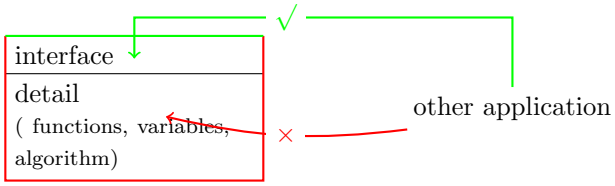
1.2 Encapsulation

In order to keep its API simple, a package typically makes only *some of its components visible* to the application code that uses the package; these components are said to be ***public***. *Other components are visible only inside the package*; these components are said to be ***private***, and serve only to support the implementation of the public components.

A package with private components hides information that is irrelevant to the application code. This technique for making a simple API is called ***encapsulation***.

The ***package specification*** declares only the *public components*, while the ***package body*** declares any *private components*.The package specification gives the package's interface, while the package body provides the implementation details.

Building an independent and self complete set of function and variable declarations is *packaging*. Restricting access to this set only via a set of interface function and variables is *hiding and encapsulation*.



Advantages of Encapsulation

- High volume details reduced to interface definitions (**Ease of development/maintenance**)
- Many different applications use the same module via the same interface (**Code re-usability**)
- Lego like development of code with building blocks (**Ease of development/maintenance**)
- Even details change, applications do not change (as long as interface is kept same) (**Ease of development/maintenance**)
- Module can be used in following projects (**Code re-usability**)

Hiding

A group of functions and variables hidden inside. The others are interface. Abstraction inside of a package:

```
double taylorseries(double);
double sin(double x);
double pi=3.14159265358979;
double randomseed;
double cos(double x);
double errorcorrect(double x);
```

```
{-- only sin, pi and cos are accessible --}
module Trig(sin,pi,cos) where
  taylorseries x = ...
  sin x = ...
  pi=3.14159265358979
  randomseed= ...
  cos x = ...
  errorcorrect x = ...
```

Code 2

2 Abstract Types

Here is a summary of the difficulties that can arise in practice when a package has a public type component:

- Application code can access the public type's representation directly. If we wish to change the representation (e.g., to make it more efficient), we have to modify not only the package but also any application code that accesses the representation directly. (Such code could be anywhere in the application program. Even if there turns out to be no such code, we have to read the whole program to be sure.)
- The public type's representation might have improper values that do not correspond to any values of the desired type. Even if the package's procedures are programmed correctly not to generate improper values, faulty application code might generate improper values.
- The public type's representation might be non-unique. A simple equality test might then yield the wrong result.

To avoid these problems we need an alternative way to define a new type. An ***abstract type*** is a type whose identifier is public but whose representation is private. An abstract type must be equipped with a group of ***operations*** to access its representation. The operations are typically procedures and constants. The values of the abstract type are defined to be just those values that can be generated by repeated use of the operations.

In other words, *internals of the datatype is hidden and only interface functions provide the access.*

Example

rational numbers: 3/4 , 2/5, 19/3

```
data Rational = Rat (Integer, Integer)
x = Rat (3,4)
add (Rat(a,b)) (Rat(c,d)) = Rat (a*d+b*c,b*d)
```

What about

Invalid value? `Rat (3,0)`

Multiple representations of the same value? `Rat (2,4) = Rat (1,2) = Rat(3,6)`

Solution: avoid arbitrary values by the user.

It does not matter whether an abstract type's representation is non-unique, because the representation is private. The key point is that only desired properties of the abstract type can actually be observed by application code using the operations with which the abstract type is equipped.

We can easily change an abstract type's representation: we need change only the package body and the private part of the package specification. We can be sure that we need not change the application code, since the latter cannot access the representation directly.

We can classify operations on an abstract type T as follows.

- A *constructor* is an operation that computes a new value of type T, possibly using values of other types but not using an existing value of type T.
- A *transformer* is an operation that computes a new value of type T, using an existing value of type T.
- An *accessor* is an operation that computes a value of some other type, using an existing value of type T.

In general, we must equip each abstract type with *at least one constructor*, *at least one transformer*, and *at least one accessor*. Between them, the constructors and transformers must be capable of generating all desired values of the abstract type.

Main purpose of abstract data types is to use them transparently (as if they were built-in) without loosing data integrity.

```
module Rational(Rational,rat,add,subtract,multiply,divide) where
  data Rational = Rat (Integer,Integer)
  rat (x,y) = simplify (Rat(x,y))
  add (Rat(a,b)) (Rat(c,d)) = rat (a*d+b*c,b*d)
  subtract(Rat(a,b)) (Rat(c,d)) = rat (a*d-b*c,b*d)
  multiply(Rat(a,b)) (Rat(c,d)) = rat (a*c,b*d)
  divide (Rat(a,b)) (Rat(c,d)) = rat (a*d,b*c)
  gcd x y = if (x==0) then y
                      else if (y==0) then x
                      else if (x<y) then gcd x (y-x)
                      else gcd y (x-y)
  simplify (Rat(x,y)) = if y==0 then error "invalid value"
                                else let a=gcd x y
                                in Rat(div x a, div y a)
```

3 Objects and Classes

An ***object*** is a group of variable components, equipped with a group of operations that access these variables.

Objects are/In objects;

- Packages containing hidden variables and access is restricted to interface functions.
- Variables with state.
- Data integrity and abstraction provided by the interface functions.
- Entities in software can be modelled in terms of functions (server, customer record, document content, etc). Object oriented design.

3.1 Classes

A ***class*** is a set of similar objects. All the objects of a given class have the same variable components, and are equipped with the same operations.

Classes are supported by all the object-oriented languages including `C++` and `JAVA`. In object-oriented terminology, an object's variable components are variously called *instance variables* or *member variables*, and its operations are usually called *constructors* and *methods*.

A ***constructor*** is an operation that creates (and typically initializes) a new object of the class. In both `C++` and `JAVA`, a constructor is always named after the class to which it belongs.

A ***method*** is an operation that inspects and/or updates an existing object of the class.

Method call,

```
O.M(E_1, ..., E_n)
```

where

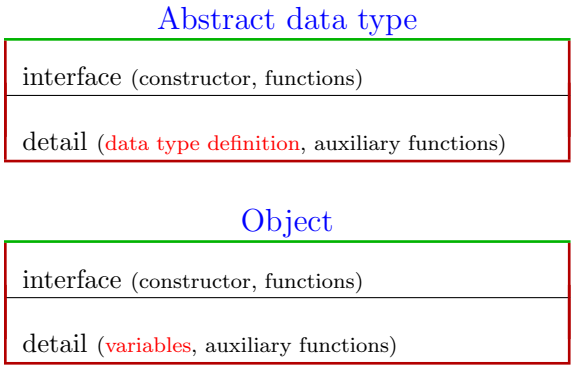
- O identifies the target object.
- M is the name of method. If there is not method named M, it causes an error.
- E_i are evaluated to yield the arguments.

Inside the method's body, `this` denotes the target object.

An *object* is an ***instance*** of the class that it belongs to (a counter type instead of a single counter). Classes have similar purposes to abstract data types.

```
class Counter {
private:   int counter;
public:    Counter() { counter=0; }
           int get() { return counter;}
           void increment() { counter++; }
} men,vehicles;
men.increment(); vehicles.increment();
men.get(); vehicles.get();
```

Code 3: C++ class declaration



Purpose

- preserving data integrity,
- abstraction,
- re-usable codes.

3.2 Subclasses

Let C denotes a class. A ***subclass*** of C is a set of objects that are similar to one another but richer than the objects of class C. An object of the subclass has all the variable components of an object of class C, but may have extra variable components. Likewise, an object of the subclass is equipped with all the methods of class C, but may be equipped with extra methods.

If S is a subclass of C, we also say that C is a ***superclass*** of S.

A subclass is said to ***inherit*** its superclass's variable components and methods. In certain circumstances, however, a subclass may ***override*** some of its superclass's methods, by providing more specialized versions of these methods.

A *private* component is visible only in its own class, while a *public* component is visible everywhere. A ***protected*** component is *visible not only in its own class but also in any subclasses*. Protected status is therefore intermediate between private status and public status.

4 Closure

Closure is an abstraction method using the saved environment state in a scope. When a function returns a *local object* or *function* as its result and language keeps the environment state along with the returned value, it becomes a *closure*.

```
def newid():
  c = 0  # this is the hidden variable
         # in the environment
  def incget():
    nonlocal c      #python 3, binds c above
    c += 1
    return c
  return incget

>>> a = newid()
>>> b = newid()
>>> a()
1
>>> b()
1
>>> b()
2
```

Code 4

Local variables of closures stay alive after call, as long as returned value is alive. **Closures** can be used for generating new functions as in higher order functions:

```
def mult(a):
  def nested(b):
    return a*b
  return nested # a different behaviour,
                # for each a value

twice = mult(2)
tentimes = mult(10)
a=twice(4)+tentimes(50)
```

Code 5

Also can be used for prototyping objects. `Javascript` example:

```
function counter() {
  var c = 0 // this is jailed in
            //local environment, hidden
  var newobj = {} // create a new empty object
  newobj.incr = function () { c++; }
  newobj.get = function () { return c;}
  return newobj
}
a = counter()
b = counter()
a.incr()
a.get()
b.get()
```

Code 6

C++ 2011 implements closures in lambda expressions by adding a set of captured variables within [ ]. This copy or get reference of auto variables in the environment in an object.

However C++ closures do not extend lifetime of captured variables. After exit, the behaviour is undetermined.

`[a, &b] (int x) { return a+x+b;}` captures a and b from current environment, a is by copy, b by reference.

```
std::function<int(int)>  multiply(int a) {
  // capture by value
  return [&] (int b) { return a*b;};
};
std::function<int()> cid() {
  int c = 0;
  // capture by copy
  return [=] () mutable { return ++c; };
};
int main() {
  std::function<int(int)> twice = multiply(2);
  std::function<int(int)> three = multiply(3);

  cout << twice(12) << ' ' << three(34) << endl;

  auto c1 = cid();
  auto c2 = cid();

  cout << c1() << ' ' << c2() << endl;
  c1(); c1(); c1();
  cout << c1() << ' ' << c2() << endl;
  return 0;
}
```

Code 7