Chapter 6

Even More Object-Oriented Concepts

Peter Müller
Globewide Network Academy (GNA)
pmueller@uu-gna.mit.edu

We continue with our tour through the world of object-oriented concepts by presenting a short introduction to static versus dynamic binding. With this, we can introduce polymorphism as a mechanism which let objects figure out what to do at runtime. But first, here is a brief overview about generic types.

6.1 Generic Types

We already know generic types from chapter 3 when we have talked about generic abstract data types. When defining a class, we actually define a *user defined type*. Some of these types can operate on other types. For example, there could be lists of apples, list of cars, lists of complex numbers of even lists of lists.

At the time, when we write down a class definition, we must be able to say that this class should define a generic type. However, we don't know with which types the class will be used. Consequently, we must be able to define the class with help of a "placeholder" to which we refer as if it is the type on which the class operates. Thus, the class definition provides us with a *template* of an actual class. The actual class definition is created once we declare a particular object. Let's illustrate this with the following example. Suppose, you want to define a list class which should be a generic type. Thus, it should be possible to declare list objects for apples, cars or any other type.

```
template class List for T {
  attributes:
```

```
    ...       /* Data structure needed to implement */
              /* the list */

  methods:
    append(T element)
    T getFirst()
    T getNext()
    bool more()
}
```

The above template class *List* looks like any other class definition. However, the first line declares *List* to be a template for various types. The identifier T is used as a placeholder for an actual type. For example, *append()* takes one element as an argument. The type of this element will be the data type with which an actual list object is created. For example, we can declare a list object for apples if a definition fot the type Apple exists:

```
List for Apple appleList
Apple anApple,
      anotherApple
appleList.append(anotherApple)
appleList.append(anApple)
```

The first line declares *appleList* to be a list for apples. At this time, the compiler uses the template definition, substitutes every occurrence of T with *Apple* and creates an actual class definition for it. This leads to a class definition similar to the one that follows:

```
class List {
  attributes:
    ...       /* Data structure needed to implement */
              /* the list */

  methods:
    append(Apple element)
    Apple getFirst()
    Apple getNext()
    bool more()
}
```

This is not exactly, what the compiler generates. The compiler must ensure that we can create multiple lists for different types at any time. For example, if we need another list for, say pears, we can write:

```
List for Apple appleList
List for Pear pearList
...
```

In both cases the compiler generates an actual class definition. The reason why both do not conflict by their name is that the compiler generates unique names. However, since this is not viewable to us, we don't go in more detail here. In any case, if you declare just another list of apples, the compiler can figure out if there already is an actual class definition and use it or if it has to be created. Thus,

```
List for Apple aList
List for Apple anotherList
```

will create the actual class definition for *aList* and will reuse it for *anotherList*. Consequently, both are of the same type. We summarize this in the following definition:

Definition 6.1.1 (Template Class) *If a class A is parameterized with a data type B, A is called* **template class**. *Once an object of A is created, B is replaced by an* **actual data type**. *This allows the definition of an* **actual class** *based on the template specified for A and the actual data type.*

We are able to define template classes with more than one parameter. For example, *directories* are collections of objects where each object can be referenced by a *key*. Of course, a directory should be able to store any type of object. But there are also various possibilities for keys. For instance, they might be strings or numbers. Consequently, we would define a template class *Directory* which is based on two type parameters, one for the key and one for the stored objects.

6.2 Static and Dynamic Binding

In strongly typed programming languages you typically have to *declare* variables prior to their use. This also implies the variable's *definition* where the compiler reserves space for the variable. For example, in Pascal an expression like

```
var i : integer;
```

declares variable i to be of type *integer*. Additionally, it defines enough memory space to hold an integer value.

With the declaration we *bind* the name i to the type *integer*. This binding is true within the scope in which i is declared. This enables the compiler to check at compilation time for type consistency. For example, the following assignment will result in a type mismatch error when you try to compile it:

```
var i : integer;
...
i := 'string';
```

We call this particular type of binding "static" because it is fixed at compile time.

Definition 6.2.1 (Static Binding) *If the type T of a variable is explicitly associated with its name N by declaration, we say, that N is* **statically bound** *to T. The association process is called* **static binding**.

There exist programming languages which are not using explicitly typed variables. For example, some languages allow to introduce variables once they are needed:

```
...       /* No appearance of i */
i := 123  /* Creation of i as an integer */
```

The type of i is known as soon as its value is set. In this case, i is of type integer since we have assigned a whole number to it. Thus, because the *content* of i is a whole number, the *type* of i is integer.

Definition 6.2.2 (Dynamic Binding) *If the type T of a variable with name N is implicitly associated by its content, we say, that N is* **dynamically bound** *to T. The association process is called* **dynamic binding**.

Both bindings differ in the time when the type is bound to the variable. Consider the following example which is only possible with dynamic binding:

```
if somecondition() == TRUE then
  n := 123
else
  n := 'abc'
endif
```

The type of n after the `if` statement depends on the evaluation of *somecondition()*. If it is TRUE, n is of type integer whereas in the other case it is of type string.

6.3 Polymorphism

Polymorphism allows an entity (for example, variable, function or object) to take a variety of representations. Therefore we have to distinguish different types of polymorphism which will be outlined here.

The first type is similar to the concept of dynamic binding. Here, the type of a variable depends on its content. Thus, its type depends on the content at a specific time:

```
v := 123        /* v is integer */
...             /* use v as integer */
v := 'abc'      /* v "switches" to string */
...             /* use v as string */
```

Definition 6.3.1 (Polymorphism (1)) *The concept of dynamic binding allows a variable to take different types dependent on the content at a particular time. This ability of a variable is called* **polymorphism***.*

Another type of polymorphism can be defined for functions. For example, suppose you want to define a function *isNull()* which returns TRUE if its argument is 0 (zero) and FALSE otherwise. For integer numbers this is easy:

```
boolean isNull(int i) {
  if (i == 0) then
    return TRUE
  else
    return FALSE
  endif
}
```

However, if we want to check this for real numbers, we should use another comparison due to the precision problem:

```
boolean isNull(real r) {
  if (r < 0.01 and r > -0.99) then
    return TRUE
  else
    return FALSE
  endif
}
```

In both cases we want the function to have the name *isNull*. In programming languages without polymorphism for functions we cannot declare these two functions because the name *isNull* would be doubly defined. Without polymorphism for functions, doubly defined names would be ambiguous. However, if the language would take the *parameters* of the function into account it would work. Thus, functions (or methods) are uniquely identified by:

- the *name* of the function (or method) and
- the *types* of its *parameter list*.

Since the parameter list of both *isNull* functions differ, the compiler is able to figure out the correct function call by using the actual types of the arguments:

```
var i : integer
var r : real

i = 0
r = 0.0

...
```

```
if (isNull(i)) then ...   /* Use isNull(int) */
...
if (isNull(r)) then ...   /* Use isNull(real) */
```

Definition 6.3.2 (Polymorphism (2)) *If a function (or method) is defined by the combination of*

- *its name and*
- *the list of types of its parameters*

we speak of **polymorphism**.

This type of polymorphism allows us to reuse the same name for functions (or methods) as long as the parameter list differs. Sometimes this type of polymorphism is called *overloading*.

The last type of polymorphism allows an object to choose correct methods. Consider the function *move()* again, which takes an object of class *Point* as its argument. We have used this function with any object of derived classes, because the is-a relation holds.

Now consider a function *display()* which should be used to display drawable objects. The declaration of this function might look like this:

```
display(DrawableObject o) {
  ...
  o.print()
  ...
}
```

We would like to use this function with objects of classes derived from *DrawableObject*:

```
Circle acircle
Point apoint
Rectangle arectangle

display(apoint)      /* Should invoke apoint.print() */
display(acircle)     /* Should invoke acircle.print() */
display(arectangle)  /* Should invoke arectangle.print() */
```

The actual method should be defined by the *content* of the object *o* of function *display()*. Since this is somewhat complicated, here is a more abstract example:

```
class Base {
attributes:

methods:
```

```
  virtual foo()
  bar()
}

class Derived inherits from Base {
attributes:

methods:
  virtual foo()
  bar()
}

demo(Base o) {
  o.foo()
  o.bar()
}

Base abase
Derived aderived

demo(abase)
demo(aderived)
```

In this example we define two classes *Base* and *Derived*. Each class defines two methods *foo()* and *bar()*. The first method is defined as `virtual`. This means that if this method is invoked its definition should be evaluated by the content of the object.

We then define a function *demo()* which takes a *Base* object as its argument. Consequently, we can use this function with objects of class *Derived* as the is-a relation holds. We call this function with a *Base* object and a *Derived* object, respectively.

Suppose, that *foo()* and *bar()* are defined to just print out their name and the class in which they are defined. Then the output is as follows:

```
foo() of Base called.
bar() of Base called.
foo() of Derived called.
bar() of Base called.
```

Why is this so? Let's see what happens. The first call to *demo()* uses a *Base* object. Thus, the function's argument is "filled" with an object of class *Base*. When it is time to invoke method *foo()* it's actual functionality is chosen based on the current content of the corresponding object *o*. This time, it is a *Base* object. Consequently, *foo()* as defined in class *Base* is called.

The call to *bar()* is *not* subject to this content resolution. It is not marked as `virtual`. Consequently, *bar()* is called in the scope of class *Base*.

The second call to *demo()* takes a *Derived* object as its argument. Thus, the argument *o* is filled with a *Derived* object. However, *o* itself just represents the *Base* part of the provided object *aderived*.

Now, the call to *foo()* is evaluated by examining the content of *o*, hence, it is called within the scope of *Derived*. On the other hand, *bar()* is still evaluated within the scope of *Base*.

Definition 6.3.3 (Polymorphism (3)) *Objects of superclasses can be filled with objects of their subclasses. Operators and methods of subclasses can be defined to be evaluated in two contexts:*

1. *Based on object type, leading to an evaluation within the scope of the superclass.*
2. *Based on object content, leading to an evaluation within the scope of the contained subclass.*

The second type is called **polymorphism**.