

Inheritance of Behaviour



In this chapter you will learn the concept of **inheritance of behaviour**. A special form of **inheritance** will be presented – **polymorphism** – which comes in two shapes: **overloading** and **overriding**.

Inheritance of Operations

The inheritance of structure or attributes has proven to be a very useful concept. The resulting class hierarchies not only reduce the number of classes that have to be defined it also simplifies the maintenance of the model that has been built. It is not only structure that can be inherited, behaviour also can be inherited. The **inheritance of operations** works in exactly the same way as the inheritance of properties.

Let's recap how inheritance works: you begin with an existing class and use it as a starting point for creating another class. When you inherit from a class, the new class (the child class) maintains a connection with the original class (the parent class). This connection assures that any change in the 'parent' is automatically also made in all its 'children' as well.

Let's use our Sloopy family to demonstrate the workings of this concept. Every Sloopy can speak, so we model this behaviour at the most generic (top) level, which is our little friend Kojak.



Figure 1: Speaking Sloopy Kojak

If we consider Sloopy Kojak as the 'parent' class then and 'children' classes will inherit this behaviour and be able to speak also. Now, given that every Sloopy will be able to speak, we can represent this using the already learned INHERITS keyword in pseudo code:

```

CLASS Sloopy
ATTRIBUTES:
METHODS:
    speak(word) RETURN none
    <implementation>

CLASS SloopyBoy INHERITS Sloopy
ATTRIBUTES:
METHODS:
    fight(sloopy) RETURN none

CLASS SloopyGirl INHERITS Sloopy
ATTRIBUTES:
METHODS:
    play(sloopy) RETURN none

```

Each Sloopy created in the *SloopyBoy* class has two methods: *speak*, which has been inherited from *Sloopy* and *fight*, which is solely available in the *SloopyBoy* class. Similarly, each Sloopy created in the *SloopyGirl* class has two methods: *speak*, which has been inherited from *Sloopy* and *play*, which is only available in the *SloopyGirls* class.

As before, behaviour is represented in UML in the bottom box. The notation for inheritance remains the same as before: it connects the parent class with a child class through a link called **generalisation**, indicated by a solid arrow with an unfilled triangular head. The above piece of pseudo code – with some added properties – can be represented in UML as follows:

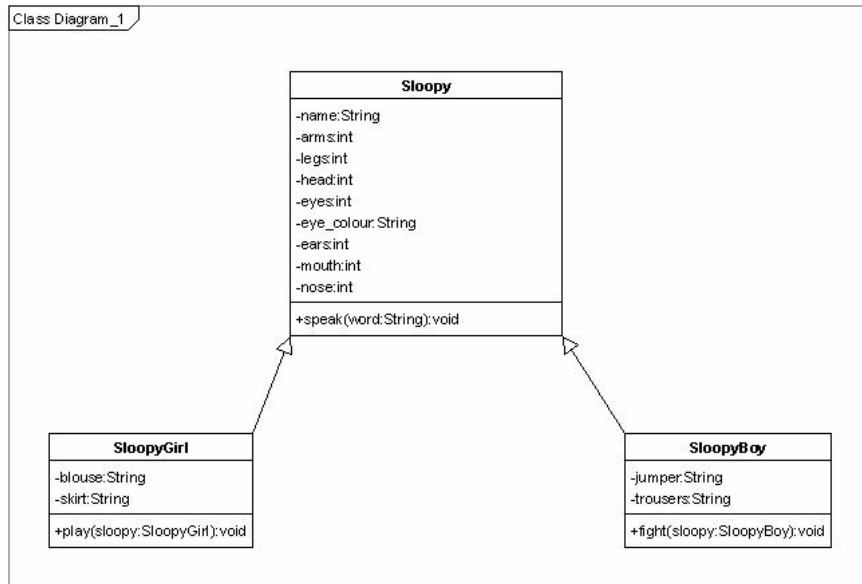


Figure 2: Inheritance of Behaviour in UML

All instances of *SloopyBoy* and *SloopyGirl* are able to speak, while girls can also play with other girls and boys will be fighting other boys.

Polymorphism

Polymorphism is a big word for a simple concept. "Polymorphic" is Greek and stands for "many forms". Effectively, this means that you can ask many different objects to perform the same action.

An abstract example is that of the '+' operator in mathematics. It can be used with a range of different variable types. For instance,

- $3 + 4 = 7$
- $3.14 + 7.22 = 10.36$
- "Gentleware" + " " + "Poseidon" = "Gentleware Poseidon"

The '+' method can be used with integer numeric objects, with floating point numbers and with strings / text. While the usage is identical from a modeller's perspective, the programmer will have to implement three different methods internally.

This concept of polymorphism is very powerful when it comes to modelling in an object-oriented fashion. When dealing with inherited properties it is possible to change their values on a lower level in the class hierarchy. Like properties, methods can also be inherited. But in order to change their behaviour an alternative mechanism is required. Two mechanisms that are based on polymorphism, called **overloading** and **overriding**, are used to achieve the same for inherited behaviour.

Overloading

We will go straight back to our Sloopy family. As shown above, all Sloopies are able to speak (Sloopy language), so it makes sense to have this basic behaviour modelled at the top level of the Sloopy hierarchy. As expected, the speaking behaviour is now inherited from the root (top) level down to all children. Now let us say we have a very linguistic Sloopy who can speak more than one language.

One way to model this ability is to add a method for this Sloopy dedicated to the new language (in addition to the generic speak method). A far more elegant method is to use **overloading**, which is demonstrated in the pseudo code below:

```

CLASS Sloopy
  PROPERTIES:
  METHODS:
    speak (word) RETURN sound
CLASS Language_Sloopy INHERITS Sloopy
  METHODS:
    speak (word, [language]) RETURN sound

```

The code creates a class called *Sloopy* with a method called *speak* that takes a parameter *word* and returns a sound. Every instance of this class can only speak Sloopy language, a very special language which uses only punctuation marks and no letters or numbers! The implementation part is ignored, but it is assumed that all words are in Sloopy language, for example @#\$%!. It creates a second class called *Language_Sloopy*, which inherits the *speak* method from the *Sloopy* class. It **overloads** the speak method, which now takes a *word* parameter and an optional *language* parameter. As before it returns a *sound* that represents the vocals to be made.

In UML, overloaded methods are represented in the same way as any other method. The bottom part of each class shows its interface. Polymorphism is assumed when the same method is declared in a sub-class, as it has been done in the *Language_Sloopy* class below:

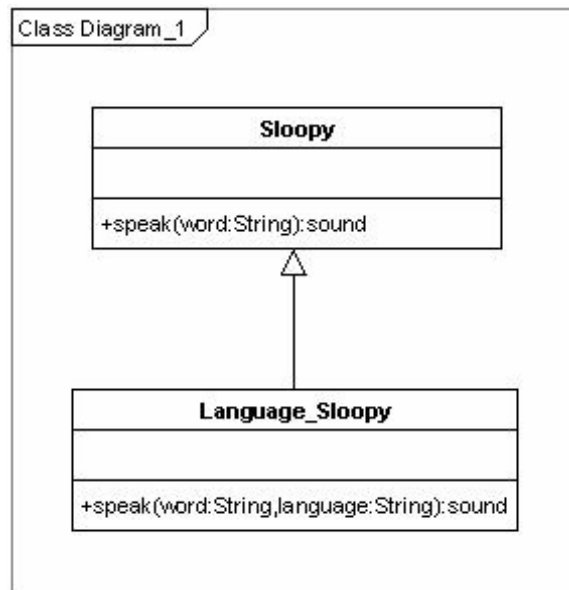


Figure 3: Overloading in UML

Using these definitions / classes, we can now create 2 Sloopies:

```

CREATE Sloopy normal_sloopy
normal_sloopy. speak ("@#$£%")

CREATE Language_Sloopy lingo_sloopy
lingo_sloopy. speak ("@#$£%")
lingo_sloopy. speak ("Games", English)
lingo_sloopy. speak ("Spiele", German)
lingo_sloopy. speak ("Jeux", French)
  
```

This slightly more complex snippet of pseudo code does the following:

A 'normal' Sloopy instance – called *normal_sloopy* is created. It can say the word "@#\$£%" in the default Sloopy language. Then a 'linguistic' Sloopy instance (called *lingo_sloopy*) is created. It can speak the word "@#\$£%", which is an inherited method from the Sloopy class.

We then ask it to say the same word in English ("Games"), German ("Spiele") and French ("jeux"), for which the overloaded *speak* method is used.

Overloading occurs when several methods have the same names with different method signatures. In the example above, the same method *speak* can have 2 different forms or method signatures. The first one only includes a single parameter *word*, the second one includes 2 parameters: *word* and *language*.

Overriding

Now let us assume that there is another Sloopy who can only sing, but cannot speak! There are now **two** ways of modelling this. The inherited *speak* method could be deleted and replaced with a *sing* method. A more elegant way would be to **override**

(ie replace) the inherited method. It would still be called *speak* and would still take a single *word* parameter; however, its internal implementation part would have to be changed, which is not the concern of the modeller but of the programmer!

In pseudo code, the existing method is overridden when a method with the same signature is defined in a sub-class:

```

CLASS Sloopy
  PROPERTIES:
  METHODS:
    speak (word) RETURN sound
      <implementation>
CLASS Singing_Sloopy INHERITS Sloopy
  METHODS:
    speak (word) RETURN sound
      <implementation>

```

The two methods look identical, but implementations, which are not shown here, are different.

In UML, overridden methods are represented in the same way as overloaded methods. Again, polymorphism is assumed when the same method is declared in a sub-class, as it has been done in the *Singing_Sloopy* class below:

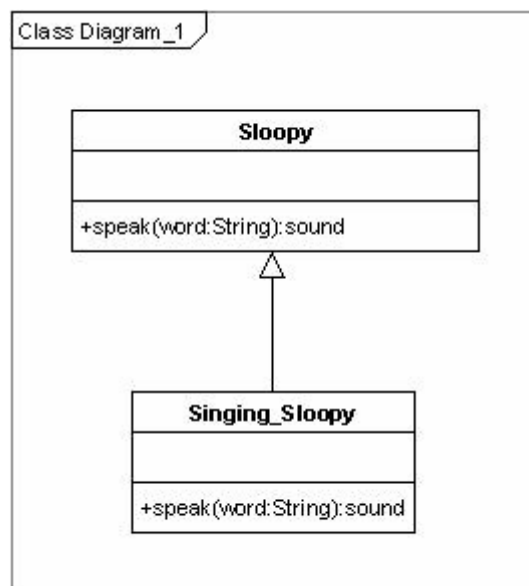


Figure 4: Overriding in UML

Overriding occurs when a method has the same name and signature (the combination of its return type and parameters) as a method in its parent class.

In conclusion

Polymorphism gives us the ultimate flexibility in extensibility. Polymorphism is a term that describes a situation where one name may refer to different methods. There are two type of polymorphism: overloading and overriding.

Overriding occurs when a method has the same name and signature as a method in its parent class, while overloading occurs when several methods have the same names with different method signatures.

Here are the key **characteristics of overloaded methods**:

- They supplement each other
- They can exist in any number within same class
- They must have different arguments
- Their return type may be freely chosen
- They are determined at compile time

Here are they key **characteristics of overridden methods**:

- They (largely) replace the method it overrides
- Each method in a parent class can be overridden only once in any one class
- They must have an argument list of identical type and order
- Their return type must be identical
- They are determined at runtime.



Inheritance of behaviour and **polymorphism** give us the ultimate flexibility in extensibility. Polymorphism is a term that describes a situation where one name may refer to different methods. There are two type of polymorphism: **overloading** and **overriding**.