

Operations



So far we have covered the structure of objects and classes. In order to 'do' something with these objects we will add **behaviour**, which is represented by **operations**. Operations, also known as **methods**, are introduced in this Chapter.

Methods

So far we have described objects using properties/attributes. This has proven a very powerful concept, especially when organised in classes. However, none of the objects described are actually able to do anything. This is about to change in this chapter with the introduction of **object behaviour**. Object behaviour describes **actions** that objects are able or allowed to carry out.

Objects are described using attributes which have certain characteristics such as a type (for example color) and an optional default value (for example green). The behaviour of objects is described using so-called **methods**.

Behaviour is represented by **operations** which are also called **methods**.

A method refers to a piece of code (written in an object-oriented programming language such as C++ or Java) that is exclusively associated either with an object or class. This code is called an **instance method** when associated with an object or a **class method** or **static method** when associated with a class. This is identical to the concept of object properties and class properties.

A method consists of a sequence of statements to perform an action. Each method has a set of input parameters to define and contain those actions and possibly an output value (called **return value**) of some kind. The purpose of methods is to provide a mechanism for accessing the private data stored in an object or a class.

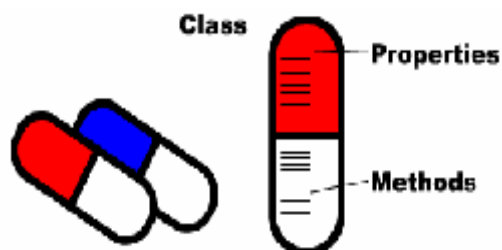


Figure 1: Properties and methods

Let us demonstrate this with an example of Nina, the dancing Sloopy. Have a look at the two (object) instances of Nina below. The first one is a static version and we have been able to describe her with properties, a concept that has already been learned.

The second one however is able to sing and dance! So far we cannot describe such behaviour. In addition to the properties which have been dealt with previously, Nina also has a range of abilities which have to be modelled, namely singing and dancing.



Figure 2: Static Nina and Dancing Nina

UML fully supports the **representation of methods** as seen in Figure 3. The bottom part of the class has been reserved to represent these methods.

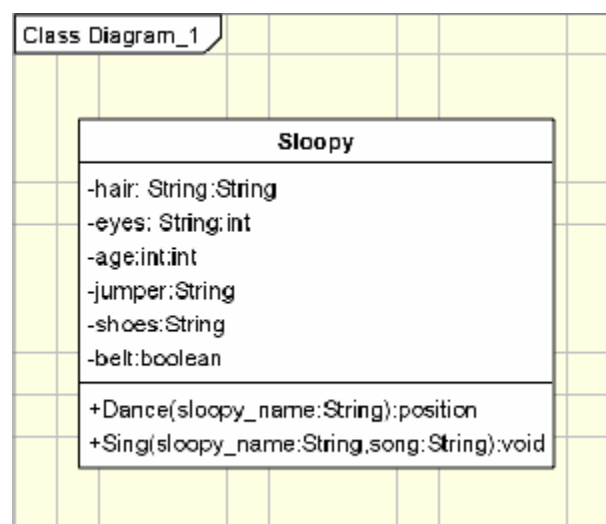


Figure 3: UML representation of the *Sloopy* class with 2 methods (*Dance* and *Sing*)

Look at the fragment piece of pseudo code shown below. Two methods called *Dance* and *Sing* have been introduced that can only be used with the *Sloopy* class.

```

CLASS Sloopy
ATTRIBUTES:
    .
    .
    .
METHODS:
    Dance(sloopy_name) RETURN position
        <implementation>

    Sing(sloopy_name, song)
        <implementation>

```

The pseudo code can be explained as: The input parameter *sloopy_name* tells the method *Dance* which Sloopy has to be used for the dancing operation. The return value is called *position*, which tells where the Sloopy *sloopy_name* will be when the dancing method is finished. The *Sing* method accepts 2 parameters. The first one is the again the name of the Sloopy, whereas the second one tells the Sloopy which *song* to sing. The method has no return values. The *<implementation>* blocks are the placeholders for the actual code that has to be programmed using an object-oriented programming language.

The lines starting with the keyword *Dance* and *Sing* are called the **headers** of the method or **interface**. It is the only part of the method the outside world can see and is the only part that is relevant from a modelling perspective. The *<implementation>* part of the method (the **body**) is treated as a black box – it is the programmer's job to make this work, not the modeller's!

Encapsulation

The idea behind **encapsulation** (or **information hiding**) is to hide design decisions that are most likely to change in a computer program, thus protecting other parts of the program from change if the design decision is changed. **Design decisions** specify exactly how an operation is implemented by the programmer, which is irrelevant to the outside world and therefore treated as a black box. Encapsulation protects design decisions by providing a stable interface which shields the remainder of the system from the implementation (the details that are most likely to change).

Coming back to our *Dance* and *Sing* examples above, it is not relevant from a design/modelling perspective how the operations are performed, only that they are performed.

Encapsulation reduces software development risk by shifting the code's dependency on an uncertain implementation (design decision) onto a well-defined

interface. Programs using the interface perform operations purely through the interface so, if the implementation changes, the programs do not have to change. In addition to reducing maintenance of systems, this approach has an additional advantage that is highly relevant in a commercial setting. It is possible to pass on classes to third parties without giving access to the inner workings, which is intellectual property of the developing company.



Methods have been introduced to model behaviour of objects. They include a **header** (or **interface**) and a **body** (or **implementation**), which guarantees the correct working of multiple objects. This guarantee will be dealt with in more detail in the next section.