

**Hermann Schmitt.**

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmitther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

## **The OO System for Mathematica, Version 3. - Programming Objectoriented in Mathematica -**

### **Introduction.**

This introduction describes the basic concepts of the object-oriented method independent of the special language.

Object-oriented programming (OOP) is a programming language model organized around “objects” rather than “actions” and data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data. The programming challenge was seen as how to write the logic, not how to define the data. Object-oriented programming takes the view that what we really care about are the objects we want to manipulate rather than the logic required to manipulate them. Examples of objects range from human beings (described by name, address, and so forth) to buildings and floors (whose properties can be described and managed) down to the little widgets on your computer desktop (such as buttons and scroll bars).

The first step in OOP is to identify all the objects you want to manipulate and how they relate to each other, an exercise often known as data modeling. Once you have identified an object, you generalize it as a class of objects and define the kind of data it contains and any logic sequences that can manipulate it. Each distinct logic sequence is known as a method. A real instance of a class is called (no surprise here) an “object” or in some environments an “instance of a class”. The object or class instance is what you run in the computer. Its methods provide computer instructions and the class object characteristics provide relevant data. You communicate with objects – and they communicate with each other – with well defined interfaces called messages.

The concepts and rules used in object-oriented programming provide these important benefits:

The concept of a data class makes it possible to define subclasses of data objects that share some or all of the main class characteristics. Called inheritance, this property of OOP forces a more thorough data analysis, reduces development time, and ensures more accurate coding. Since a class defines only the data it needs to be concerned with, when an instance of that class (an object) is run, the code will not be able to accidentally access other program data. This characteristic of data hiding provides greater system security and avoids unintended data corruption.

The definition of a class is reusable not only by the program for which it is initially created but also by other object-oriented programs (and for this reason, can be more easily distributed for use in networks).

### ***Description of the System***

#### **Class Definitions, Objects and Inheritance.**

The central point of an object oriented system is the creation of objects out of classes. Additionally to the original class of the object further classes may be included into the object. This makes it possible to use the features of these classes in several object types. This is called „inheritance“. Additional classes are included either by specifying them directly in the original class as parents or indirectly in parent classes. If in one class there may be specified more than one parent class directly, this is called „multiple inheritance“. In the system to be described multiple inheritance is possible.

Figure 1 shows an example of a class definition

```
{ "Rectangle", "parent[] := (Return[{"Owner"}]);", "
status = { width, height };
", "
RectangleInit[wi_, he_, na_] := Module[{ },
    OwnerInit[na];
    width = wi;
    height = he;
];
getWidth[] := Return[width];
```

## Hermann Schmitt.

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmitther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

```
setWidth[wi_] := width = wi;
getHeight[] := Return[height];
setHeight[hi_] := height = hi;
getSides[] := Return[status];
area[] := Return[width * height];
circumference[] := Return[2 * (width + height)]
"}

```

Figure 1: Example of a Class Definition.

A class is specified as a List in the sense of Mathematica with 4 elements, where every element is a string, which is delimited by quotes. If there are to be used quotes inside an element, they must be specified in the form “\””.

The first element of the class definition specifies the name of the class. This name must be identical with the name of the file, in which the class definition is stored. Additionally the file name must have the extension „.cl”.

The second element of the class definition contains the class methods and data fields, i.e. data common to all objects of the class, and methods, which access these common data. One class method, which must always be present, is parent[.]. This method returns all of the direct parent classes in the form of a List. The List may be empty.

In the third element the data fields are defined. The names of the data fields have to be specified as elements of a List. The name of the List is always „status”. The List should show the logical structure of the data fields as good as possible.

All objects of a class have the same data fields, but the contents of the data fields are normally different.

In the fourth element the methods of the class have to be defined. All objects of a class have the same methods, but they use different data.

In OO it is a common practice, to define the methods get<Name> and set<Name> for each data field of the class. There can also be defined initialization methods, which are automatically called, when an object is created. These methods have the name <classname>Init. In these methods further initialization methods may be called for parent classes. More than one initialization method may be defined for a class, because e.g. function calls with the same function name but different number of parameters lead to different functions. Initialization methods behave in the same way as other methods. They can also be used for other purposes than the initialization of status variables.

Methods and variables are created at the place, where they are defined with the following exception:

**Attention:** If a method is defined without a “” in front of it and there is a method with the same name in a parent class, no method is created, but the method in the nearest subsequent class is redefined.

### Remark:

When a class is loaded by the function loadClass[cl\_] or in connection with the creation of an object, it is checked, if it is formally correct.

With the function

analyseClass[cl\_]

the same checks and additional checks, which are more time consuming, are performed.

## Creation of Objects and Invocation of Methods of an Object.

By the following function an object is created from one or more classes:

**obj = MathNew[cl, par]**

Here „cl” is the classname as a string and „par” represents optional parameters, with which the object may be initialized. If parameters are specified, a corresponding initialization function with the name <cl>Init must be defined in the class.

A value with the type (Head) „MathObject” is assigned to the variable obj. This value is a reference to the object.

A variable “this” is defined, which contains the reference to the object, too. This variable can already be used in <class>Init for storing a reference to the object somewhere, although the object itself may not yet be initialized.

## Hermann Schmitt.

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmitther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

A new object may also be created by “cloning” an existing object.

**obj = clone[obj]**

The new object has the same values as the original object.

After the object is created methods of the object may be invoked in the following way:

**obj @ amethod[..]**

The system implements unrestricted polymorphism, i.e. methods with the same name may be invoked for objects of different classes. At the invocation it need not be known, which the class of the object is. Unrestricted polymorphism is not possible in languages, where each data field is of a special type.

For the access to the methods and variables of an object the classes contained in the object (the root object and its parents) have to be conceived as sequentially ordered according to the following principles:

A class precedes the parent class(es) defined in it.

If a class has more than one parent classes, a parent class B, which is located behind a parent class A, is ordered behind this parent class A, and the parent classes of class A.

If a method/variable is accessed from outside the object (e.g. by obj @ method[]), the object is searched from the beginning for the method/variable. The method/variable found first is selected.

If a method/variable is accessed in a method A of the object, the method/variable is searched for in the class, where method A is located, and in the subsequent classes. The method/variable found first is selected.

There is a difference in the treatment of methods and status variables:

### Methods:

Methods with the same name are admitted more than once in an object. In this oo system this is achieved by putting a “~” before the method name.(see above). “super” (see below) assists the use of this constellation: It ignores a method in the same class and executes the method with same name next in the parent class. This constellation is often used, when a method is to be modified in an child class.

### Status Variables:

Names of status variables normally exist only once in an object. Methods in child classes can access the variable.

In the oo system described here the variable “status” is an exception of this rule:

A variable with the name status exists in every class. The variable “status” can only be accessed by methods of this class directly. Methods in child classes can only access the variable indirectly by methods of the class, where the variable status is located.

The variable “status” is introduced into the oo system in order that aggregations of variables can be accessed together. It may be useful, to also access parts of the List “status”:

The system offers maximal possibilities for the access to the methods and variables of an object. But it is not always optimal to use all possibilities:

It could make sense – depending on the circumstances -, to adhere to the following restrictions:

From outside the object, only methods are to be accessed not variables. If values of variables have to be used, they are accessed indirectly by so called get- and set- methods (see Fig. 1).

As further restriction methods are to access only variables of the same class, variables of subsequent classes have to be accessed indirectly by methods of that class.

The following two method invocations are only to be used inside object methods.

**self @ amethod[...]**

In this way a method is invoked in the object, in which the expression is.

**super @ amethod[...]**

This expression is only allowed in classes, in which „amethod“ is defined. It has the effect, that not this method in the class is invoked but a method with this name is searched for in direct and indirect parent classes and the method such found is invoked.

With the function

## Hermann Schmitt.

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmitther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

### releaseObject[obj]

all symbols of the object specified in the parameter are removed. The symbol obj remains existent.

## Example 1

### The Classes.

```
{ "Rectangle", "parent[]:= (Return[{"Owner"}]);  
", "  
status = { width, height};  
", "  
RectangleInit[wi_, he_, na_] := Module[{  
    OwnerInit[na];  
    width = wi;  
    height = he;  
};  
getWidth[] := Return[width];  
setWidth[wi_] := width = wi;  
getHeight[] := Return[height];  
setHeight[hi_] := height = hi;  
getSides[] := Return[status];  
area[] := Return[width * height];  
circumference[] := Return[2 * (width + height);  
"}]
```

```
{ "Circle", "parent[]:= (Return[{"Owner"}]);", "  
status = {radius};  
", "  
CircleInit[ra_, na_] := Module[{  
    OwnerInit[na];  
    radius = ra;  
};  
getRadius[] := Return[radius];  
setRadius[ra_] := radius = ra;  
getSides[] := Return[status];  
area[] := Return[Pi*radius^2];  
circumference[] := Return[2*Pi*radius]  
"}]
```

```
{ "Triangle", "parent[]:= (Return[{"Owner"}]);", "  
status = {side1, side2, side3};  
", "  
TriangleInit[s1_, s2_, s3_, na_] := Module[{  
    OwnerInit[na];  
    side1 = s1;  
    side2 = s2;  
    side3 = s3;  
};  
getSide1[] := Return[side1];  
setSide1[s1_] := side1 = s1;  
getSide2[] := Return[side2];  
setSide2[s2_] := side2 = s2;  
getSide3[] := Return[side3];
```

## Hermann Schmitt.

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmittther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

```
setSide3[s3_] := side3 = s3;
getSides[] := Return[status];
area[] := Module[{s, ar},
  s = (side1 + side2 + side3) / 2; ar = Sqrt[s * (s - side1)* (s - side2) * (s - side3)]; Return[ar];
];
circumference[] := Return[side1 + side2 + side3]
"} }
```

```
{"Owner", "parent[]:= (Return[{}]);", "
status = {name1};
", "
getName[] := Module[{},Return[name1]];
setName[na_] := Module[{},name1 = na;];
OwnerInit[na_] := Module[{},
  name1 = na;];
"}.
```

## The Program

### Description of the Program.

Classes are defined for the geometrical objects rectangle, circle and triangle, additionally a class Owner, which is parent class for the classes of the geometrical objects.

The program creates one object for each of the three classes of geometrical objects, initializes them, and stores them in a List for future processing.

Then - in a loop - for each object the class name is printed, and four methods are invoked.

The program shows, that the system implements polymorphism. There are no special data fields for the objects of different classes and the methods are invoked in the same way for the objects of the three classes.

### The Program.

```
$classPath = "C:\\Math_OO\\classes7";
objLst = {};
obj = MathNew["Rectangle", 3, 2, "Schmitt"];
objLst = Append[objLst, obj];
obj = MathNew["Circle", 4., "Meyer"];
objLst = Append[objLst, obj];
obj = MathNew["Triangle", 2, 3, 4., "Schulze"];
objLst = Append[objLst, obj];
Print["\nAnalysis of Geometrical Objects\n"];
For[iter = 1, iter <= Length[objLst], iter++,
  obj = Extract[objLst, {iter}];
  Print[" "];
  Print["Geometrical Object: ", iter];
  Print["ClassName of Object: ", className[obj]];
  Print["Sides: ", obj @ getSides[]];
  Print["Area: ", obj @ area[]];
  Print["Circumference: ", obj @ circumference[]];
  Print["Owner: ", obj @ getName[]];
];
Print["\nEnd of Program"];
```

## The Output.

Analysis of Geometrical Objects

```
Geometrical Object: 1
ClassName of Object: Rectangle
Sides: {3, 2}
```

## Hermann Schmitt.

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmittther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

Area: 6

Circumference: 10

Owner: Schmitt

Geometrical Object: 2

ClassName of Object: Circle

Sides: {4.}

Area: 50.2655

Circumference: 25.1327

Owner: Meyer

Geometrical Object: 3

ClassName of Object: Triangle

Sides: {2, 3, 4.}

Area: 2.90474

Circumference: 9.

Owner: Schulze

End of Program

## The Work with Classes and Class Methods.

If an object is created, all the classes needed are loaded, in so far they are not yet loaded.

With the function

**cls1 = loadClass[cl]**

A class is loaded, if it is not yet loaded.

Classes loaded remain available.

The class variables and class methods have the context: "classname".

loadClass always returns a value of type (Head) „Class“. With this value class methods may be invoked in the same way as instance methods are invoked with values of type (head) „MathObject“:

**cls1 @ amethod[.]**

To access class variables and class methods the context "classname" can be used in any case.

The variable with the head "Class" can be used in objects. It is preferable to use it in objects, because it is independent of the classname and is therefore reusable for other classes.

The directories, in which classes, that are needed, are stored, must be specified in the variable \$classPath. For \$classPath the same rules are valid as for \$Path. It is recommended to join the directories to \$classPath rather than to assign them.

With the function

**cls2 = createClass[clcont]**

an executable class may be created in main storage, with clcont = {S1, S2, S3, S4}, where S1, S2, S3, S4 are the 4 strings of a class definition (see Fig. 1). This form may also be obtained by reading a class definition with Get.

If a class with the same name is already in main storage, it is replaced. The function returns a data element of type (Head) Class as the return value of loadClass.

## An Example for the Work with Class Variables and Class Methods.

The example counts the objects created in the program and prints the number at the end of the program.

**The Class.**

```
{"Rectangle", "parent[] := (Return[{"Owner"}]);
```

```
setNull[] := AnzObj = 0;
```

```
addAnzObj[] := AnzObj = AnzObj + 1;
```

```
PAnzObj[] := Print[AnzObj];
```

## Hermann Schmitt.

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmittther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

```
","
status = { width, height };
","
RectangleInit[wi_, he_, na_] := Module[{ },
    OwnerInit[na];
    Rectangle`addAnzObj[];
    width = wi;
    height = he;
];
getWidth[] := Return[width];
setWidth[wi_] := width = wi;
getHeight[] := Return[height];
setHeight[hi_] := height = hi;
getSides[] := Return[status];
area[] := Return[width * height];
circumference[] := Return[2 * (width + height)];
"}

```

### The Program.

```
<<D:\\tst_dav\\oosys.m
$classPath = "D:\\tst_dav\\classes";
cls1 = loadClass["Rectangle"];
cls1 @ setNull[];
objLst = { };
For [iter=1, iter<=10,iter++,
    obj = MathNew["Rectangle", 3, 2, "Schmitt"];
    objLst = Append[objLst, obj];
];
Print["Number of created Objects: ", cls1 @ AnzObj];
Print["\nEnd of Program"];

```

### The Output.

Number of created Objects: 10

End of Program

## Delegation.

A different case is the so called „delegation“.

In the case of delegation, an object A contains other objects as data. It may execute methods of these objects. It can be said, that the object A delegates tasks to these object.

The problems solved by inheritance and delegation differ partly.

An object may also contain a reference to a class (see “the Work with Classes and Class Methods” for the description of the reference). In this way the class methods of a class may be executed. There may be classes, which have no instance methods.

Example 3 uses delegation to objects and classes.

## Storing of Objects on External Media.

The storage of objects on external media is called “Serialization” in OO. Speaking more precisely – not the object is stored, but only the “status” of the object. Reloading(deserialization) means more precisely, that a new object is created and initialized with the status of the old object.

In the system to be described, an object is serialized by the following statement:

**srobj = serialize[object, directory, filename]**

## Hermann Schmitt.

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmittther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

The first parameter is a data element of type (Head) "MathObject", the second parameter specifies the directory, into which the object is to be stored, and the third parameter the filename of the file, into which the object is to be stored. As "directory" the full directory path must be specified. The statement returns a data element of type (Head) "SerObject", which represents the serialized object.

The serialized Object is reloaded (deserialized) by the following statement:

**obj = deserialize[srobj]**

The parameter is the data element of type "SerObject", which represents the serialized object. A data element of type (Head) "MathObject" is returned.

In the function "serialize", additional optional parameters may be added to the three required parameters. These parameters are transferred unchanged to the data element of type "SerObject", which is returned by this function. This parameters may be used as help, to search for the object to be reloaded. In example 2 the owner is such an additional optional parameter.

Objects, which are part of the status of the object, are not serialized. If those objects are present, they have to be serialized before the serialization of the object. If an object is reloaded (deserialized), normally serialized objects, which are part of the status, have to be deserialized subsequently. But it is also feasible, that those objects are left serialized, if they are not needed, or that they are deserialized, when required.

## Example 2

Example 2 uses the same classes as example 1.

**Part 1:** The program creates some geometrical objects and writes them to disk (serialized). The Owner is added as additional parameter in the serialize function. The SerObjects are stored in a list as index. The index is stored on disk at the end of part 1.

**Part 2:** The index is read in and the SerObjects with "Schmitt" are selected. The corresponding serialized objects are deserialized and information about them is printed.

### The Program.

```
<<D:\\tst_dav\\oosys.m
$classPath = "D:\\tst_dav\\classes";
objLst = {};
obj1 = MathNew["Rectangle", 2, 3, "Schmitt"];
srobj = serialize[obj1, "D:\\tst_dav\\objects", objId[obj1], obj1 @
getName[]];
objLst = Append[objLst, Apply[List, srobj]];

obj1 = MathNew["Circle", 4., "Schmitt"];
srobj = serialize[obj1, "D:\\tst_dav\\objects", objId[obj1], obj1 @
getName[]];
objLst = Append[objLst, Apply[List, srobj]];

obj1 = MathNew["Triangle", 2, 3, 4., "Schulze"];
srobj = serialize[obj1, "D:\\tst_dav\\objects", objId[obj1], obj1 @
getName[]];

objLst = Append[objLst, Apply[List, srobj]];
Print["objLst: ", objLst];
Put[objLst, "C:\\index.m"];

$classPath = "D:\\tst_dav\\classes";
objLst1 = Get["C:\\index.m"];
prtLst = Map[Take[#, {3}]&, objLst1];
sel = Position[prtLst, "Schmitt"];
sel = Map[Take[#, 1]&, sel];
```

## Hermann Schmitt.

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmittther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

```
Print["The Geometrical Objects with the Owner Schmitt"];
For[iter = 1, iter <= Length[sel], iter++,
  srobj = Extract[objLst1, sel[[iter]]];
  obj2 = deserialize[Apply[SerObject, srobj]];
  Print[" "];
  Print["Geometrical Object: ", iter];

  Print["ClassName of Object: ", className[obj2]];
  Print["Sides: ", obj2 @ getSides[]];
  Print["Area: ", obj2 @ area[]];
  Print["Circumference: ", obj2 @ circumference[]];
];
Print["\nEnd of Program"];
```

### Output

```
objLst: {{D:\tst_dav\objects, OI$7.ser, Schmitt}, {D:\tst_dav\objects, OI$60.ser, Schmitt},
```

```
{D:\tst_dav\objects, OI$110.ser, Schulze}}
```

```
The Geometrical Objects with the Owner Schmitt
```

```
Geometrical Object: 1
```

```
ClassName of Object: Rectangle
```

```
Sides: {2., 3.}
```

```
Area: 6.
```

```
Circumference: 10.
```

```
Geometrical Object: 2
```

```
ClassName of Object: Circle
```

```
Sides: {4.}
```

```
Area: 50.2655
```

```
Circumference: 25.1327
```

```
End of Program
```

## Class Arrays in Objects

In this case a parent of a class consists of an array of class elements. In this way e.g. the geometrical objects dealt with earlier can be specified by points: rectangles and triangles by 3 points, circles by 2 points.

The parent specification for the array of class elements consists of a list of 2 elements, the first element contains the name of the class, from which the class elements are built, the second element specifies the number of class elements to be constructed.

The definition of the class, from which the class elements are constructed, is a normal class definition, but the system adds an index to all methods of the class. This index has to be used, when calling the methods.

It may be advantageous to use the class methods of a "support" class to handle the point elements in all classes, containing point arrays.

There seems to be no other object oriented language, in which class arrays are implemented.

Example 3 is an example for this feature.

## Additional Features

### The Invocation of Functions in Packages from Objects.

The same kind of objects may be created in different programs. Therefore the objects must be as independent as possible of the programs. Therefore functions from packages used in objects are loaded within the objects. To achieve this, a new way for invoking functions in packages is implemented:

## Hermann Schmitt.

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmittther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

The oo system inspects, if the package is already loaded. If this is not the case it loads the package and invokes the function.

This is achieved by the following statement:

```
pck["context"] @ f[...]
```

where "context" is the context of the packages.

The method has the advantage, that functions with the same name in different packages may be used at the same time. No shadowing problems occur. The contexts of the packages are not inserted into the context path.

In this way the invocation of methods in classes and functions in packages is similar. The statement may also be used outside objects, if the oo software is loaded.

The new statement also leads to a close connection between objects and packages from another point of view: Instead of programming routines in methods, it may be possible to call functions in an appropriate package, and seen from the other side: packages may be upgraded to objects, by calling the functions of the package in an object.

The method has the advantage, that functions with the same name in different packages may be used at the same time. No shadowing problems occur. The contexts of the packages are not inserted into the context path.

### An Example.

```
Print["\nInvocation of Functions in Packages"];
Print["Vor Aufruf EasterSunday"];
Print["$Context: ", $Context];
Print["$ContextPath: ", $ContextPath];
Print["Packages: ", $Packages];
Print["Date of Easter Sunday 2004: ", pck["Miscellaneous`Calendar`"] @
EasterSunday[2004]];
Print["Nach Aufruf EasterSunday"];
Print["$Context: ", $Context];
Print["$ContextPath: ", $ContextPath];
Print["Packages: ", $Packages];
```

### The Output of the Example.

```
Invocation of Functions in Packages
Vor Aufruf EasterSunday
$Context: Global`
$ContextPath: {oosys`, Global`, System`}
Packages: {oosys`, Global`, System`}
Date of Easter Sunday 2004: {2004, 4, 11}
Nach Aufruf EasterSunday
$Context: Global`
$ContextPath: {oosys`, Global`, System`}
Packages: {Miscellaneous`Calendar`, oosys`, Global`, System`}
```

## Blocks.

An additional feature of oo programming are blocks. Blocks are implemented in Mathematica as "Pure Functions". This feature is an alternative way to access methods of an object.

The "Pure Function" in this case contains variables and/or methods of an object.

In the Example below the radius of the circle is set by a block.

The Class:

```
{"Circle", "parent[]:= (Return[{"Owner"}]);", "
```

## Hermann Schmitt.

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmittther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

```
status = {radius, prf1};
",
CircleInit[ra_, na_] := Module[{} ,
    OwnerInit[na];
    radius = ra;
    prf1 = (setRadius[#1]&);
];
getRadius[] := Return[radius];
setRadius[ra_] := Module[{} ,
    radius = ra;
];
getPrf1[] := Module[{} ,
    Return[prf1];
];
setRadius2[ra_] := Module[{} ,
    Print["Anfang setRadius2"];
    self @ setRadius[ra];
];
getsR[] := Module[{} ,
    prf1 = (Function[x, setRadius[x]]);
    Return[prf1];
];
getSides[] := Return[status];
arc[] := Return[Pi*radius^2];
circumference[] := Module[{} ,
    Return[2*Pi*radius];
];
"}

```

The pure function is stored in the variable prf1, and prf1 is stored in the List “status”.

With the method “getPrf1[] the pure function can be transferred into the user program.

```
b11 = obj1 @ getPrf1[]
```

With the statement

```
b11[9];
```

the radius is set to 9.

If a method is called several times, a block is faster and consumes less resources.

## Getting Information about Classes and Objects.

With the function

**className[cls]**

the classname of the class, which is represented by cls is returned.

With the function

**className[obj]**

the classname of the class from which the object is created is returned.

The function

**classNames[obj]**

prints the names of all classes the object obj1 consists of.

With the function

**getClass[obj]**

the class, from which obj1 is created, is returned.

The function

**instanceOf[obj, cls]**

## Hermann Schmitt.

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmittther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

returns True, if obj is created from cls, False otherwise.

Every object has a unique object identification.

With the function

### **objId[obj]**

this object identification is returned.

Attention: This object identification changes when serializing/deserializing the object. It is only unique within a single program.

With the function

### **parentClass[obj]**

the direct parent classes of the object are returned as List.

### **allSymb[obj]**

returns all symbols of the object, subdivided into the classes the object consists of.

## Debugging.

### The Classes.

The directories specified in the variable \$classPath are searched for the classes needed in the program.

A class is a List in the sense of Mathematica, consisting of 4 elements, which all have to be texts.

This is checked, if a class is loaded.

If quotes are to be coded in one of the four elements, e.g. when printing texts, the quotes have to be coded in the form. \"

Additionally the following attributes are tested, when the class is read in:

- Element 1 has to contain the name of the class. This name must be identical with the name of the file, in which the class is stored. (Additionally the file name has the extension "cl").
- Element 2 must contain a parent statement (which may be empty).

No other checks are done for the contents of the elements in the reading phase. This examination is for some kinds of errors done, when the objects are created, other errors are not found until the program is executed.

This is dealt with in the following.

### Errors in the process of building objects

With the function call MathNew[.....] objects are created out of one or more classes. The classes involved are worked on in a loop and inserted into the object by the statement ToExpression[.....]. The elements of the class are worked on separately.

The error messages of the Mathematica system cannot help to find the cause of the errors, because the errors are caused by the code in the classes.

In this case the oo system creates error messages of the following kind:

```
Error in Class: Circle2 Part: Instance Methods
```

```
{ToExpression::sntx }
```

```
Exit
```

The error message shows the class and the part of the class, which caused the error.

Then the program stops.

## Conclusion.

The system is a rather complete implementation of the principles of an oo system.

Inheritance is implemented well. The system offers maximal capabilities for the access to the methods and variables of an object. In order to make applications more reliable, it may be advantageous to restrict the access in some cases.

The system implements multiple inheritance. Multiple inheritance makes it easier to create applications in some cases.

C++ implements multiple inheritance. The interfaces of Java cannot replace multiple inheritance fully.

## **Hermann Schmitt.**

Dipl. Volksw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmittther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

Polymorphism is only complete, if the applications need not know the classes of the objects they access. This is only possible, if all objects can be stored in the same variable. This is not possible, when the variables are typed, as it is the case e.g. in Java and C++. Example 1 below shows the use of polymorphism

It is very useful, if objects can be stored on external media (serialization). In this way they get independent of the program, in which they were created. In the system this can be done in a very easy way. The system also offers a feature, which makes it possible to search for an object with specified attributes. This may be seen as a primitive database feature. The storage of objects on external media is supported by the described polymorphism. The application may not know the class of an object on external media.

Example 2 is an example of serialization, where objects with a special value of an attribute are searched for.

Serialization is not possible in C++, because classes only exist in the source, not in the executable module. Java implements serialization very well.

The system supports class arrays as parents of classes. This feature is not supported in any other oo language. The feature is very useful as example 3 shows. Here the geometrical objects, which are used as example objects throughout the documentation, are represented by points.

The problems of debugging are often not paid attention. But for the practical use of the system these problems may be very important. In the system good solutions are offered for debugging.

If a class is loaded, it is checked, if the class has formally the correct structure.

When executing a program, in the the default case Mathematica collects the error messages and prints them at the end. This is not suitable, because sometimes it is very difficult – if not impossible – to find the input, which causes the error message. This system chooses another way, which makes it easy, to find the input, which causes the error message.

When an object is built from classes, the name of the class and the part of the class are specified, if it contains an error.

## Hermann Schmitt.

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmittther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

### Example 3

#### Description of the Example.

This example is similar to Example 1. But here the geometrcal objects are represented by points.

#### The classes.

```
"PtSupport",  
parent[]:= (Return[{}]);  
sides[lobj_, anz_] := Module[{} ,  
    pts = Table[(lobj @ getPt[i][]), {i, 1, anz}];  
    pts2 = RotateLeft[pts,1];  
    str = pts - pts2;  
    sdlst = N[Map[Function[e, Sqrt[Dot[e, e]]], str]];  
    Return[sdlst];  
];  
", "  
status = {pt};  
", "  
"}  
  
{"Rectangle2", "parent[]:= (Return[{"Owner", {"Point", 3}]);  
", "  
status = {suppKlass, lobj};  
", "  
", "  
Rectangle2Init[ pn_, p11_, p12_, p21_, p22_, p31_, p32_] := Module[{} ,  
    OwnerInit[pn];  
    PointInit[1][p11, p12];  
    PointInit[2][p21, p22];  
    PointInit[3][p31, p32];  
  
];  
getlobj[] := Return[lobj];  
setlobj[hlobj_] := lobj = hlobj;  
setsKl[sKl_] := suppKlass = sKl;  
getsKl[] := Return[suppKlass];  
getPts[] := Table[getPt[i][] , {i, 1, 3}];  
  
area[] := Module[ {sdLst, ber} ,  
    sdLst = suppKlass @ sides[lobj, 3];  
    sdLst = Drop[sdLst, -1];  
    ar =Apply[Times, sdLst];  
    Return[ar];  
];  
circumference[] := Module[ {sdLst, circ} ,  
    sdLst = suppKlass @ sides[lobj, 3];  
    sdLst = Drop[sdLst, -1];  
    circ = 2 * Apply[Plus, sdLst];  
    Return[circ];  
];  
draw[]:= Module[{} ,  
    plst={getPt[1][],getPt[2][],getPt[3][],getPt[3][]+(getPt[1][]-getPt[2][]),getPt[1][]};  
    gobjline = Line[plst];  
    gobjgraph =Show[Graphics[gobjline], Axes -> True, AspectRatio ->1];
```

## Hermann Schmitt.

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmittther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

```
];  
"}
```

```
{"Circle2", "parent[]:= (Return[{"Owner", {"Point", 2}}]);  
", "  
status = {suppKlass, lobj};  
", "
```

```
Circle2Init[pn_, p11_, p12_, p21_, p22_] := Module[{},  
  OwnerInit[pn];  
  PointInit[1][p11, p12];  
  PointInit[2][p21, p22];  
  ];  
setsKl[sKl_] := suppKlass = sKl;  
getsKl[] := Return[suppKlass];  
getlobj[] := Return[lobj];  
setlobj[hlobj_] := lobj = hlobj;  
getPts[] := Table[getPt[i][], {i, 1, 2}];  
area[] := Module[{sdLst, ar},  
  sdLst = suppKlass @ sides[lobj, 2];  
  sdLst = Drop[sdLst, -1];  
  ar = Pi * First[sdLst]^2;  
  Return[ar];  
];  
circumference[] := Module[{sdLst, circ},  
  sdLst = suppKlass @ sides[lobj, 2];  
  sdLst = Drop[sdLst, -1];  
  circ = 2 * Pi * First[sdLst];  
  Return[circ];  
];  
draw[] := Module[{},  
sdLst = suppKlass @ sides[lobj, 2];  
  
sawcircle = Circle[ getPt[1][], First[sdLst]];  
sawgraph = Show[Graphics[sawcircle], Axes -> True, AspectRatio -> 1];  
];  
"}
```

```
{"Triangle2", "parent[]:= (Return[{"Owner", {"Point", 3}}]);  
", "  
status = {suppKlass, lobj};  
", "  
Triangle2Init[ pn_, p11_, p12_, p21_, p22_, p31_, p32_] := Module[{},  
  OwnerInit[pn];  
  PointInit[1][p11, p12];  
  PointInit[2][p21, p22];  
  PointInit[3][p31, p32];  
  ];  
getlobj[] := Return[lobj];  
setlobj[hlobj_] := lobj = hlobj;  
setsKl[sKl_] := suppKlass = sKl;  
getsKl[] := Return[suppKlass];  
getPts[] := Table[getPt[i][], {i, 1, 3}];  
area[] := Module[{s, s1, ar},
```

## Hermann Schmitt.

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmittther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

```
sdLst = suppKlass @ sides[lobj, 3];
(** area = Sqrt[s(s-a)(s-b)(s-c)] where s=1/2(a+b+c) **)
s = Apply[Plus, sdLst] / 2;
s1 = s - sdLst;
ar = Sqrt[s * Apply[Times, s1]];
Return[ar];
];
circumference[] := Module[{sdLst, circ},
sdLst = suppKlass @ sides[lobj, 3];
circ = Apply[Plus, sdLst];
Return[circ];
];
draw[]:= Module[{ },
plst = {getPt[1][],getPt[2][], getPt[3][],getPt[1][]};
gobjline = Line[plst];
gobjgraph =Show[Graphics[gobjline], Axes -> True, AspectRatio ->1];
];
" }
```

```
{ "Owner", "parent[]:= (Return[{}]);", "
status = {name1};
", "
getName[] := Module[{ },Return[name1]];
setName[na_] := Module[{ },name1 = na;];
OwnerInit[na_] := Module[{ },name1 = na;];
" }
```

```
{ "Point", "parent[]:= (Return[{}]);", "
status = {pt};
", "
PointInit[p1_, p2_] := pt = {p1, p2};
getPt[] := Return[pt];
setPt[p_] := pt = p;
" }
```

## The Program

```
<< "D:\ootst01\system\oosys.m"
$classPath = "D:\ootst01\classes_b";
$MessagePrePrint = (Print["Error -> Quit"]; Quit[]) &;
MFDebug = False;
Print["Anfang Programm"];
suppKlass = loadClass["PtSupport"];
objLst = { };
obj1 = MathNew["Rectangle2", "Schmitt", 1, 3, 1, 1, 5, 1];
obj1 @ setsKl[suppKlass];
obj1 @ setobj[obj1];
objLst = Append[objLst, obj1];
obj2 = MathNew["Circle2", "Meyer", 0, 0, 0, 3];
obj2 @ setsKl[suppKlass];
obj2 @ setobj[obj2];
objLst = Append[objLst, obj2];
obj3 = MathNew["Triangle2", "Mueller", 1, 3, 5, 5, 4, 2];
obj3 @ setsKl[suppKlass];
obj3 @ setobj[obj3];
```

## Hermann Schmitt.

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmittther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

```
objLst = Append[objLst, obj3];
Print["\nAnalysis of Geometrical Objects"];
For[iter = 1, iter <= Length[objLst], iter++,
  obj = Extract[objLst, {iter}];
  Print[" "];
  Print["nClassName of Object: ", className[obj]];
  Print["Points: ", obj @ getPts[]];
  Print["Area: ", obj @ area[]];
  Print["Circumference: ", obj @ circumference[]];
  Print["Owner: ", obj @ getName[]];
];
```

## The Output of the program

Begin of Program

Analysis of Geometrical Objects

nClassName of Object: Rectangle2

Points: {{1., 3.}, {1., 1.}, {5., 1.}}

Area: 8.

Circumference: 12.

Owner: Schmitt

nClassName of Object: Circle2

Points: {{0., 0.}, {0., 3.}}

Area: 28.2743

Circumference: 18.8496

Owner: Meyer

nClassName of Object: Triangle2

Points: {{1., 3.}, {5., 5.}, {4., 2.}}

Area: 5.

Circumference: 10.7967

Owner: Mueller

**Hermann Schmitt.**

Dipl. Volkw., Dipl. Math.

Tel.: 07251 2720

E-Mail: hermanns@schmittther.de

Heidelbergerstr. 39

76646 Bruchsal

Jan . 3, 2007

Germany

## ***Changes in Version 1.1 Compared with the Original System***

Version 1.1 is upward compatible with the original system.

The following features have been added:

An easy way to serialize/deserialize objects

The ability to create classes in the program

several functions, which make available information about classes and objects.

## ***Changes in Version 1.2 compared with Version 1.1.***

The following features were added:

- Classes may be checked for formal correctness.

Version 1.2. is upward compatible with version 1.1, with the exception, that objects serialized with version 1.1 cannot be deserialized with version 1.2.. The kind of invocation of the functions serialize and deserialize remain the same.

## ***Changes in Version 1.3 compared with Version 1.2.***

The following features were added:

- an easy way for invoking functions in packages from objects.
- A function clone[obj], which creates a new object from an existing object. The new object has the same values as the original object.
- When creating an new object, a symbol "this" is defined, which contains the object, which is created.

## ***Changes in Version 2 compared with Version 1.3.***

- The functions @, self, super were modified.

- Class Arrays in Objects were implemented.

- The use of blocks in the oo system was described.

## ***Changes in Version 3 compared with Version 2.***

The Debugging features are decisively enhanced