11.1 Object-oriented programming

Concepts

Object-oriented programming (OOP)
Abstract data types (ADTs): objects and operations
Encapsulation (information hiding)
ADTs and higher mathematics
Logical ADT operations
  Construction and destruction
  Selection and output
  Equality and assignment
Nonlogical operations
Overloading
^:= definitions in Mathematica

Mathematica’s graphics features and the add-on package twoDG described in this chapter employ some ideas from object-oriented programming (OOP). Both feature the use of abstract data types (ADTs). The twoDG user interface emphasizes operations on graphics objects, rather than implementation details of the objects themselves. To designate new operations it overloads conventional algebraic symbols instead of inventing new ones—this makes the code easier to read and use. Its packaging as a Mathematica add-on is an example of encapsulation, or information hiding.

Techniques such as these are routine for many expert programmers. They originated long ago, but were explicitly implemented in programming languages, such as C++, only in recent decades. They now form part of the discipline of object-oriented analysis, on which is based the design of many large contemporary information systems, perhaps even your computer’s operating system and the underlying Mathematica code. This section provides a glimpse of the field, but not an overview, because Mathematica doesn’t encourage users to practice OOP. The text describes only those OOP ideas involved in Mathematica graphics and the twoDG package.¹ For a somewhat broader

¹ For example, the OOP inheritance concept, which is analogous to an important feature of higher mathematics, isn’t employed here at all. It seems particularly unwieldy to implement inheritance in Mathematica programming.
view, featuring many of the same mathematical applications as this book but using the C++ language, consult the author's earlier work (Smith 1999).

**Abstract data types**

The ADT (abstract data type) concept evolved as a way to modularize software design and limit the effect of programming errors. To that end, it hides details of data organization from all but the most fundamental operations on the data. To an ADT user, these details appear only in the background, if at all. The emphasis is on the operations. That feature of ADTs is the most characteristic of their use in *Mathematica* programming.

The ADT concept is so general that it's hard to describe. To avoid writing something obviously false, you tend to write as little as possible. An ADT consists of a set $F$ of related operations, and a data structure definition $D$. The latter defines the set of all data, called *objects*, that are handled interchangeably by the operations in $F$. An ADT user should have no access to the objects except by invoking operations in $F$. The only aspects of the ADT that a user needs to know are a general description of $D$ and the effects of the operations in $F$—what values they yield when applied to arguments in $D$. Details of data storage and algorithms for the operations may be hidden. This OOP information hiding strategy is called *encapsulation*. Its justification is that ADT users who have no access to those details can never misuse them.

**Parallel with higher mathematics**

For example, consider a vector algebra ADT. $F$ should contain operations for vector addition, scalar multiplication, etc., and $D$ should define objects that represent vectors. ADT users know—through the description of the addition operation—that the sum of two vectors is a vector. But they needn’t know how vectors are stored nor how addition is performed. Conventional vector algebra software might store vectors as lists of their scalar components, but software for handling very long sparse vectors (with mostly zero components) might store them as much shorter lists consisting just of the pairs of nonzero components and their indices. The corresponding addition algorithms would be quite different.
This example shows how closely the ADT notion parallels modern higher mathematics. You study vector algebra by considering vector spaces. A vector space consists of a set \( D = \{ \alpha, \beta, \ldots \} \) of vectors and a set \( F \) containing the addition and scalar multiplication operators \( + \) and \( \cdot \). A short list of axioms such as \( t \cdot (\alpha + \beta) = t \cdot \alpha + t \cdot \beta \) tells what rules the vectors and operators obey. Vector space theory derives theorems from these axioms. Some are very deep and involved. Many different vector spaces arise in practical mathematics. Numerical analysis considers spaces whose vectors are \( n \)-tuples of real numbers, \( n \)-tuples of complex numbers, polynomials, integrable functions, power series, matrices, and so on. A vector algebra ADT as described in the previous paragraph represents a vector space if its addition and scalar multiplication operations satisfy the axioms. By presenting vector algebra in the abstract context of vector space theory, we modularize the mathematics. We ensure that the theorems you learn about vector spaces apply equally well to all those concrete examples. Virtually all modern higher mathematics uses this modularization technique.

### Logical operations

Some operations, such as vector addition, are specific to certain ADTs. Others, however, are found in nearly every properly defined ADT. There’s no standard term for them; this book calls them logical operations. In informal discussion—even of ADTs developed to implement higher mathematics—they’re often ignored, because our common sense performs the same services tacitly. But common sense must be designed into computer programs explicitly, so these operations are discussed here in more detail than you might expect.

The first logical operation is construction. This usually involves reserving a portion of computer memory, storing data there to represent an object, and perhaps setting some system variables that indicate its relationship to its environment. You might specify the data explicitly, as for the vector \(<1, 2, 3>\) or implicitly, as for the vector with 100 entries, each equal to \(2.3\). In each case the data are specified by some parameters. Antithetical to construction is destruction: returning that memory to general use when the object is no longer needed, and resetting those system variables.

It’s good programming practice to provide a way to recover parameter values and closely related information from the objects constructed. Otherwise, an ADT user might need to retain separate copies of those values. Such redundancy is an invitation to consistency errors—one copy may be changed but not the other. Therefore, most
ADTs include logical operators called selectors, that return simple information about the objects.

Details of output operations clearly vary with the ADT. For example, we use different output styles for vectors, polynomials, etc. Output operations are often the most difficult to program, because output should be easy to read. But before you start any serious ADT programming you must have some rudimentary output just for troubleshooting. It needn’t be easy or attractive to read. This is a logical output operation: nearly every ADT must have one.

In mathematical discussions, we often rely on common sense to determine whether two objects under consideration are the same or not. For instance, we don’t usually feel a need for explicit agreement about deciding whether two integers written on scratch paper are the same, even if one is written with Roman numerals. In some areas of elementary mathematics, we do make such a convention explicit. For example, some books emphasize that functions are equal just when they have the same domain, and the same values for each argument in the domain; how they’re described by formulas doesn’t matter. (Less meticulous books expect students to use that convention but don’t bother to state it.) In higher mathematics, the need for agreement on equality is even more apparent. For example, some geometry texts may regard two solids as the same if one can be moved to coincide with the other by a direct motion (rotation followed by translation), while others may also allow indirect motions such as mirror reflections. Right-handed and left-handed screws are always different under the first convention, but may be regarded as the same under the second. When we use ADTs in software design, we can’t relegate the logical notion of equality to common sense. We must define it explicitly. Given two objects as arguments, this operation returns a Boolean value: true if they’re equal, false if not. Making equality explicit is one of the most fruitful OOP techniques. Many errors in mathematics and software design result from confusion about equality.

Manipulating data usually requires the ability to make one object \( x \) equal to another object \( y \). In mathematical discussion this is usually phrased, let \( x \) equal \( y \). In programming we say, assign to the variable \( x \) the value of the variable \( y \). We distinguish the variable, which is either the symbol or the portion of memory that represents the object, from its value—the object itself. The assignment operation is somewhat more complex than you might expect, because we don’t normally require that \( x \) and \( y \) be different symbols or memory areas. If they’re the same, no action is required. If they’re different, we must destroy the current object \( x \), then make \( x \) a copy of \( y \).
Implementing ADTs in Mathematica

The remaining part of this section introduces techniques for implementing ADTs in Mathematica. Later sections use them to build the twoDG package.

The objects of a Mathematica ADT are explicitly constructed by formulas. There's no other formalism, and no notation for the ADT itself. In a sense no information is hidden. But you see little of the apparatus that Mathematica uses to manipulate the parameters of the construction formulas and store their values. For example, formulas

\[
\text{Line[{{$0,1}$},{$1,0$}}]} \quad \text{lineq[1,1,-1]}
\]

are constructors for the built-in Mathematica Line ADT and the twoDG lineq ADT. They look like function calls, but aren't, because no functions with these names are defined. In such cases, Mathematica regards such expressions simply as strings for processing. They contain all the information required to represent the objects they're intended to represent, so in effect, they are the objects. These examples represent the same line, which passes through points \(<0,1>\) and \(<1,0>\) and has equation \(1x + 1y + (-1) = 0\) relative to a Cartesian \(x,y\) coordinate system. The memory allocated to the objects stores information that depends on the specified parameters. Actually Mathematica just stores the full forms of the expressions. The symbols Line and lineq identify them as objects of these types. The fact that you rarely see full forms and don’t really know exactly how they’re stored is evidence of information hiding. These two ADTs have different uses. Objects of type Line can represent polygons as well—that’s why the Line parameter is a list of pairs of coordinates—and you can incorporate them in figures for display by the Show function. Later in this chapter, lineq objects are used to manipulate lines according to their equations. Drawing segments of them requires generation of appropriate Line objects.

*Mathematica* provides simple logical output operations for these types. Just evaluate the formulas, and they’re repeated in the output cell. Or use FullForm to see more detail:

\[
in: \quad \text{Line[{$0,1$},{$1,0$}]}; \\
\text{FullForm[\%]} \\
\text{out: Line[List[1,0,1],List[1,0]]}
\]

---

2 Interpreters of most other programming languages return an error message when they encounter an expression whose meaning is undefined. Mathematica is different.
Mathematica implements equality operations for ADTs only partially; you have to provide the rest. Here’s what Mathematica does automatically. First, the \texttt{===} operator returns \texttt{True} when applied to objects whose stored representations are identical. For example, consider

\begin{verbatim}
in: Line[{{0,1},{1,0}}] === Line[{{0,1},{1,0}}]
Line[{{0,1},{1,0}}] === Line[{{1,0},{0,1}}]
\end{verbatim}

\begin{verbatim}
out: True False
\end{verbatim}

That output isn’t very useful. The \texttt{=} operator is intended for mathematical work, but it’s only partially implemented:

\begin{verbatim}
in: Line[{{0,1},{1,0}}] \texttt{=} Line[{{0,1},{1,0}}]
Line[{{0,1},{1,0}}] \texttt{=} Line[{{1,0},{0,1}}]
\end{verbatim}

\begin{verbatim}
out: True Line[{{0,1},{1,0}}] \texttt{=} Line[{{1,0},{0,1}}]
\end{verbatim}

Mathematica left the second \texttt{=} expression unevaluated because it only defines \texttt{=} \textit{a priori} for cases where \texttt{===} returns \texttt{True}. You must implement the remaining cases yourself, if you want to use the \texttt{=} operator with \texttt{Line} objects. You probably remember that it’s difficult to test equality. The line through distinct points \(P, Q\) is the same as that through distinct points \(R, S\) just when triples \(P, Q, R\) and \(P, Q, S\) are both collinear, and that happens just when certain determinants are zero. Mathematica leaves that programming to you, should you need it.\footnote{You may need to face these questions: (1) to what extent can Mathematica decide reliably whether an algebraic formula for a determinant can be simplified to zero, and (2) how small must a floating-point approximation be to justify regarding a determinant as zero?} Equality for the \texttt{twoDG \_lineq} ADT is defined in the next section.

Defining a new meaning for a Mathematica operator such as \texttt{=} that’s already partially defined is called \textit{overloading}. This common technique, prevalent in the \texttt{twoDG} package, helps make Mathematica code more readable by curtailing symbol proliferation.

Mathematica symbols \texttt{=}\texttt{!=} and \texttt{\#} designate the negatives of \texttt{===} and \texttt{=} — they are \textit{inequality} operators. When you overload an equality operator, you must overload its negative explicitly, too.
The assignment operator \( \texttt{=} \) works with ADT objects in Mathematica as you’d expect:

\[
\begin{align*}
\text{in: } & \quad A = \text{Line}\[\{0,1\},\{1,0\}\]; \\
& \quad A === \text{Line}\[\{0,1\},\{1,0\}\] \\
& \quad A === \text{Line}\[\{1,0\},\{0,1\}\]
\end{align*}
\]

\[
\begin{align*}
\text{out: } & \quad \text{True} \\
& \quad \text{False}
\end{align*}
\]

You should test self assignment \( A = A \) and the interplay of the assignment and \( \texttt{=} \) operators.

You’ve seen examples of ADT operations provided automatically by Mathematica, and the need for an equality operation that is not. To develop your own ADTs you must be able to define such operations. The equality operator described earlier is somewhat too complex for a first example. It’s easier to discuss the selector \( c_1 \) for the first coefficient of a twoDG lineq object. If this object represents a line with equation \( ax + by + c = 0 \), then \( c_1 \) returns \( a \). You can define

\[
c_1[\text{lineq}[a_,b_,c_]] ^:= a;
\]

The underscores, as usual, indicate that the definition will be applied when values are substituted for symbols \( a, b, c \). The carat \(^\text{}\) indicates that you’re defining a property of the symbol \( \text{lineq} \), not of \( c_1 \). (This technicality stems from the way that pattern matching techniques underlie all Mathematica processing.)

There are many more examples of such definitions later in this chapter. You’ll see throughout that there’s little attention to what the objects are, but everything depends on the operations defined on them. And in section 11.11, you’ll see that the Mathematica add-on packaging conventions hide their implementations from users. These two features are the essence of OOP.
### Special Mathematica features covered in this section

<table>
<thead>
<tr>
<th>Feature</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>5</td>
</tr>
<tr>
<td>!==</td>
<td>5</td>
</tr>
<tr>
<td>≠</td>
<td>6</td>
</tr>
<tr>
<td>^=</td>
<td>6</td>
</tr>
<tr>
<td>=</td>
<td>6</td>
</tr>
</tbody>
</table>

### Index

- **twoDG package**: 1
- **Abstract data type**: 1, 2
- **ADT**: 1, 2
- **implementing**: 5
- **assignment**: 1
- **of objects**: 4, 6
- **C++**: 1
- **construction**: 1, 3, 5
- **copying**: 4
- **equality**: 6
- **floating-point**: 6
- **of algebraic formulas**: 6
- **of functions**: 4
- **of lines**: 6
- **of objects**: 1, 4, 5
- **of solids**: 4
- **error**: 3
- **consistency**: 3
- **full form**: 5
- **information**: 1, 2, 5, 7
- **Line**: 1, 2, 5, 7
- **coefficient**: 5
- **equation**: 5
- **segment**: 5
- **lineq**: 5
- **modularizing mathematics**: 3
- **software design**: 2
- **motion**: 1-3, 7
- **Object-oriented programming**: 1
- **OOP**: 1, 7
- **operating system**: 1
- **operation**: 1, 2, 7
- **logical, on ADT**: 1, 3
- **nonlogical**: 1
- **output**: 1, 2, 7
- **overloading**: 1, 6
- **package**: 7
- **parameter**: 3, 5
- **data**: 5
- **polygon**: 4
- **reflection**: 4
- **rotation**: 4
- **screw**: 4
- **selection**: 1, 4
- **sense**: 3, 4
- **Smith, James T.**: 2
- **translation**: 4
- **value**: 4
- **of variable**: 4
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>4</td>
</tr>
<tr>
<td>vector</td>
<td>2, 3</td>
</tr>
<tr>
<td>algebra</td>
<td>2, 3</td>
</tr>
<tr>
<td>space</td>
<td>3</td>
</tr>
<tr>
<td>sparse</td>
<td>2</td>
</tr>
</tbody>
</table>