**[Introduction to Data Abstraction](https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-4.html" \l "%_toc_%_sec_2.1)**

IB Computer Science:

Topic 4 Computational thinking, problem solving, and programming

Objectives:

**Thinking Abstractly**

4.1.17 Identify examples of abstraction.

4.1.18 Explain why abstraction is required in the derivation of computational

solutions for a specified situation.

4.1.19 Construct an abstraction from a specified situation.

4.1.20 Distinguish between a real-world entity and its abstraction.

Introduction:

We have seen models that can be described in *layers*, such as the OSI Model. Being able to describe, define, and implement applications in layers helps us to create more modular systems. We can hide the details of one layer from another layer, allowing more freedom for implementing solutions at each layer. Not only can hardware and software systems be described and implemented this way, data can also be defined this way. In this short lesson we'll see how a representation of a rational number can be implemented in layers in Java. This material has been adopted from *Structure and Interpretations of Computer Programs, Chapter 2.*

Data abstraction is a methodology that enables us to isolate how a compound data object is used from the details of how it is constructed from more primitive data objects.

The basic idea of data abstraction is to structure the programs that are to use compound data objects so that they operate on ``abstract data.'' That is, our programs should use data in such a way as to make no assumptions about the data that are not strictly necessary for performing the task at hand. At the same time, a "concrete'' data representation is defined independent of the programs that use the data. The interface between these two parts of our system will be a set of procedures, called *selectors* and *constructors*, that implement the abstract data in terms of the concrete representation. To illustrate this technique, we will consider how to design a set of procedures for manipulating rational numbers.

[**Example: Arithmetic Operations for Rational Numbers**](https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-4.html#%_toc_%_sec_2.1.1)

Suppose we want to do arithmetic with rational numbers. We want to be able to add, subtract, multiply, and divide them and to test whether two rational numbers are equal.

Let us begin by assuming that we already have a way of constructing a rational number from a numerator and a denominator. We also assume that, given a rational number, we have a way of extracting (or selecting) its numerator and its denominator. Let us further assume that the constructor and selectors are available as procedures. Without knowing the details of how the rational number is stored we could use the Ratl class that has the following methods:

Public methods of public class Ratl

Ratl(int numerator, int denominator) Creates a Ratl with numerator/denominator

int get\_n() Returns the get\_n()ation of a Ratl object int get\_d() Returns the denominator of a Ratl object

set\_n(int num) Sets the numerator of a Ratl object to num

set\_d(int den) Sets the denominator of a Ratl object to den

String toString() Returns the string representation of an Ratl

object as "numerator/denominator"

We haven't yet said how a rational number is represented, or how the procedures get\_n(), get\_d(), and Ratl() should be implemented. Even so, if we did have these three procedures, we could then add, subtract, multiply, divide, and test equality by using the following relations:

https://mitpress.mit.edu/sicp/full-text/book/ch2-Z-G-1.gif

https://mitpress.mit.edu/sicp/full-text/book/ch2-Z-G-2.gif

https://mitpress.mit.edu/sicp/full-text/book/ch2-Z-G-3.gif

https://mitpress.mit.edu/sicp/full-text/book/ch2-Z-G-4.gif

https://mitpress.mit.edu/sicp/full-text/book/ch2-Z-G-5.gif

We can express these rules as procedures.

Given an already constructed java Ratl class, in a file name Ratl.class, we can **extend** the Ratl class by defining **public class Rational extends Ratl** {

When we implement methods add(), subtract(), multiply(), divide(), and equals() we could implement two types of methods – one in which an operation or comparison is performed on the referenced object (and possibly modifies the object), and one that performs operations on two Rational objects.

For example: Rational fractionOne = new Ratl(3, 5);

Rational fractionTwo = new Ratl(1,3);

Rational fractionThree = new Ratl(2, 3);

Rational fractionSum;

fractionOne.add(1, 2) could add one half to the Ratl object fractionOne,

fractionOne.add(fractionTwo) could add the value of fractionTwo to fractionOne

fractionThree = fractionOne.add(fractionTwo, fractionThree);

* notice on this call, the add method of fractionOne is used, but it operates on two other Ratl objects, and not on itself. It will return a Ratl object.
* notice also that fractionSum is not initialized prior to calling fractionOne.add().

**[Abstraction Barriers](https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-4.html" \l "%_toc_%_sec_2.1.2)**

Before continuing with more examples of compound data and data abstraction, let us consider some of the issues raised by the rational number example. We defined the rational number operations in terms of a constructor Ratl() and selectors get\_n() and get\_d(). In general, the underlying idea of data abstraction is to identify for each type of data object a basic set of operations in terms of which all manipulations of data objects of that type will be expressed, and then to use only those operations in manipulating the data.

We can envision the structure of the rational-number system as shown in figure [1](https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-14.html#%_fig_2.1). The horizontal lines represent *abstraction barriers* that isolate different ``levels'' of the system. At each level, the barrier separates the programs (above) that use the data abstraction from the programs (below) that implement the data abstraction.

Programs that use rational numbers manipulate them solely in terms of the procedures supplied "for public use'' by the rational number package: add(), subtract(), multiply(), divide(), and equals(). These, in turn, are implemented solely in terms of the constructor and selectors Ratl(), get\_n(), and get\_d(). The details of how Ratl operates is irrelevant the class Rational that extends it. In effect, procedures at each level are the interfaces that define the abstraction barriers and connect the different levels.

|  |
| --- |
|  |
|  |
| **Figure 1:**  Rational number Data-abstraction barriers |

This simple idea has many advantages. One advantage is that it makes programs much easier to maintain and to modify. Any complex data structure can be represented in a variety of ways with the primitive data structures provided by a programming language. Of course, the choice of representation influences the programs that operate on it; thus, if the representation were to be changed at some later time, all such programs might have to be modified accordingly. This task could be time-consuming and expensive in the case of large programs unless the dependence on the representation were to be confined by design to a very few program modules.

For example, an alternate way to address the problem of reducing rational numbers to lowest terms is to perform the reduction whenever we access the parts of a rational number, rather than when we construct it. This leads to different constructor and selector procedures.

The difference between implementations lies in when to compute the gcd. If in our typical use of rational numbers we access the numerators and denominators of the same rational numbers many times, it would be preferable to compute the gcd when the rational numbers are constructed. If not, we may be better off waiting until access time to compute the gcd. In any case, when we change from one representation to the other, the procedures add(), subtract(), and so on do not have to be modified at all.

Constraining the dependence on the representation to a few interface procedures helps us design programs as well as modify them, because it allows us to maintain the flexibility to consider alternate implementations. To continue with our simple example, suppose we are designing a rational-number package and we can't decide initially whether to perform the gcd at construction time or at selection time. The data-abstraction methodology gives us a way to defer that decision without losing the ability to make progress on the rest of the system.

**[What Is Meant by Data?](https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-4.html" \l "%_toc_%_sec_2.1.3)**

We began the rational-number implementation by implementing the rational-number operations add(), subtract(), and so on in terms of three unspecified procedures: Ratl(), get\_n(), and get\_d(). At that point, we could think of the operations as being defined in terms of data objects -- numerators, denominators, and rational numbers -- whose behavior was specified by the latter three procedures.

But exactly what is meant by *data*? It is not enough to say "whatever is implemented by the given selectors and constructors.'' Clearly, not every arbitrary set of three procedures can serve as an appropriate basis for the rational-number implementation. We need to guarantee that, if we construct a rational number x from a pair of integers n and d, then extracting the get\_n() and the get\_d() of x and dividing them should yield the same result as dividing n by d. In other words, Ratl(), get\_n(), and get\_d() must satisfy the condition that, for any integer n and any non-zero integer d, if x is Ratl(n, d), then

=

In fact, this is the only condition Ratl(), get\_n(), and get\_d() must fulfill in order to form a suitable basis for a rational-number representation. In general, we can think of data as defined by some collection of selectors and constructors, together with specified conditions that these procedures must fulfill in order to be a valid representation.[5](https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-14.html" \l "footnote_Temp_140)

This point of view can serve to define not only ``high-level'' data objects, such as rational numbers, but lower-level objects as well.

Activity One: Review and trace the output of the main program found in module Ratl.class:

public static void main(String[] args) {

Ratl twoThirds = new Ratl(2, 3);

Ratl threeFourths = new Ratl(3, 4);

System.out.printf("two/Thirds: "+ twoThirds + "\n");

System.out.printf("threeFourths " +

threeFourths.toString() + "\n");

} // end main

Activity Two: Preparation for implementing the Rational class

* In your Cloud9 \_java workspace, create a subdirectory named rational
* Use wget to download the Ratl.class file into your rational subdirectory.

wget <http://hwmath.net/IBCS/files/Ratl.class>

* Test your tracing of the main program by running java Ratl
* Run the program a second time, using the command: java Ratl | tee Ratl.out which will display output to the screen and save the output to file Ratl.out

Activity Three: Reading from the AP Computer Science Book [Homework]

* Read the section of Chapter 3 Inheritance and Polymorphism on **Inheritance** pages 131 – 137 in the 7th Edition.

|  |  |
| --- | --- |
| Reference: This work was based on the book Structure and Interpretations of Computer Programs, Chapter 2. The book uses scheme/racket as a programming language and this document and subsequent programming assignments have been translated to use java.  The programs used here to not use "abstract" data classes, which would be appropriate for use in an IB Computer Science course selecting Option D : Object Oriented Programming. |  |

[5](https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-14.html" \l "call_footnote_Temp_140) Surprisingly, this idea is very difficult to formulate rigorously. There are two approaches to giving such a formulation. One, pioneered by C. A. R. Hoare (1972), is known as the method of *abstract models*. It formalizes the ``procedures plus conditions'' specification as outlined in the rational-number example above. Note that the condition on the rational-number representation was stated in terms of facts about integers (equality and division). In general, abstract models define new kinds of data objects in terms of previously defined types of data objects. Assertions about data objects can therefore be checked by reducing them to assertions about previously defined data objects. Another approach, introduced by Zilles at MIT, by Goguen, Thatcher, Wagner, and Wright at IBM (see Thatcher, Wagner, and Wright 1978), and by Guttag at Toronto (see Guttag 1977), is called *algebraic specification*. It regards the ``procedures'' as elements of an abstract algebraic system whose behavior is specified by axioms that correspond to our ``conditions,'' and uses the techniques of abstract algebra to check assertions about data objects. Both methods are surveyed in the paper by Liskov and Zilles (1975).