7. Object-Oriented Programming

7.1 Introduction

This chapter describes object-oriented computing and its use in data abstraction, particularly as it is understood in the Java™ language, including various forms of inheritance.

The late 1980's saw a major paradigm shift in the computing industry toward "object-oriented programming". The reason behind the surge in popularity of this paradigm is its relevance to organizing the design and construction of large software systems, its ability to support user-defined data abstractions, and the ability to provide a facility for reuse of program code. These are aside from the fact that many of the entities that have to be dealt with in a computer system are naturally modeled as "objects". Ubiquitous examples of objects are the images and windows that appear on the screen of a typical personal computer or workstation environment. These have the characteristics that they (i) maintain their own state information; (ii) can be created dynamically by the program; (iii) need to interact with other objects as a manner of course.

The paradigm shift is suggested by comparing the two cases in the following figures. In the first figure, we have the conventional view of data processing:

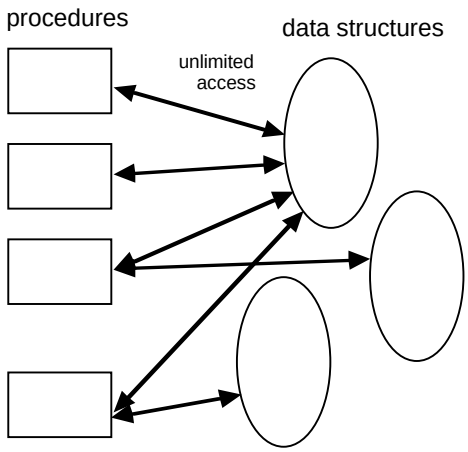

Figure 78: Conventional (pre-object-oriented) paradigm

Data structures are considered to be separate from the procedures. This introduces management problems of how to ensure that the data are operated on only by appropriate procedures, and indeed, problems of how to define what *appropriate* means for particular data. In the second figure, many of the procedures have been folded inside the data, the result being called "objects".

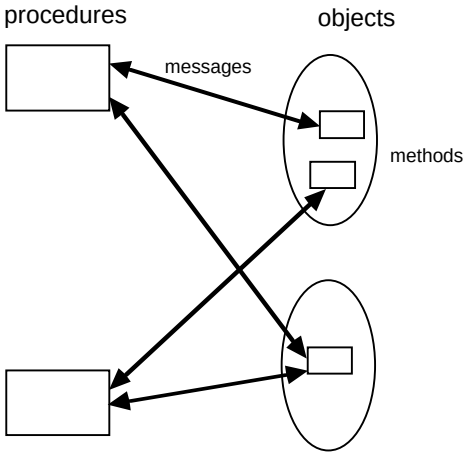

Figure 79: Object-oriented paradigm

Thus each object can be accessed only through its accompanying procedures (called ***methods***). Sometimes the access is referred to as "sending a message" and "receiving a reply" rather than calling a procedure, and sometimes it is implemented quite differently.

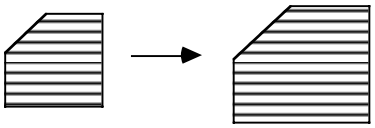

Figure 80: State transition in a graphical object as the result of sending it a resize message

In any case, an enforced connection between procedures and data solves the issue of what procedures are appropriate to what data and the issue of controlling access to the data. Languages differ on how much can be object-oriented vs. conventional: In Smalltalk, "everything is an object", whereas in Java and C++, primitive data types such as integers are not.

7.2 Classes

The concept of **object** relates to both data abstraction and to procedural abstraction. An object is a data abstraction in that it contains data elements that retain their values or state between references to the object. An object is a *procedural* abstraction, in that the principal means of getting information from it, or of changing its state, is through the invocation of procedures. Rather than attempting to access an object through arbitrary procedures, however, the procedures that access the object are associated directly with

the object, or more precisely, with a natural grouping of the objects known as their **class**. In many languages, the syntactic declaration of a class is the central focus of object definition. The class provides the specification of how objects behave and the language permits arbitrarily-many objects to be created from the class mold.

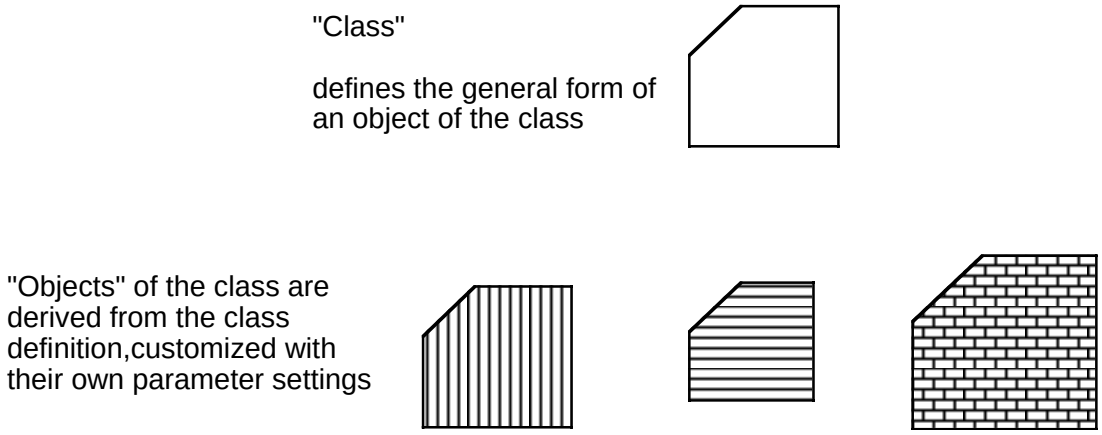

Figure 81: Class vs. Object

7.3 Attributes

Some of the types of information kept in objects may be thought of as *attributes* of the object. Each attribute typically has a value from a set associated with the attribute. Examples of attributes and possible value sets include:

size	{small, medium, large, ...}
shape	{polygonal, elliptical, ...}
color	{red, blue, green, ...}
border	{none, thin, thick, ...}
fill	{vertical, horizontal, diagonal, brick, ...}

Objects are the principal vehicles for providing data abstractions in Java: Each object can contain data values, such as those of attributes, that define its state. An object may also provide access to those values and the provide ability to change them. These things are preferably done by the methods associated with the object, rather than through direct access to the state values themselves, although Java does not prevent the latter type of access. By accessing attribute values only through methods, the representation of the state of the object can be changed while leaving the procedural interface intact. There are numerous benefits of providing a methods-only barrier between the object and its users or **clients**:

- **Principle of Modularity** ("separation of concerns"): This principle asserts that it is easier to develop complex programs if we have techniques for separating the functionality into pieces of more primitive functionality. Objects provide one way of achieving this separation.

- **Evolutionary Development**: The process of developing a program might begin with simple implementations of objects for testing purposes, then evolve to more efficient or robust implementations concurrently with testing and further development.

- **Flexibility in Data Representations**: An object might admit several different data representations, which could change in the interest of efficiency while the program is running. The object notion provides a uniform means of providing access to such changing objects.

If, on the other hand, the client were permitted direct access to the attributes of an object without using methods, the representation could never be changed without invalidating the code that accessed those attributes.

The simplest type of method for setting the value of an attribute is called a *setter*, while the simplest type for getting the value of an attribute is called a *getter*. For example, if we had some kind of shape class, with fill represented by an int, we would expect to see within our class declaration method headers as follows:

```
setFill(int Fill)

int getFill()
```

7.4 Object Allocation

Objects in Java are always dynamically *allocated* (created). It is also possible for the object to reallocate dynamically memory used for its own variables. The origin of the term "class" is to think of a collection of objects with related behaviors as a *class*, a mathematical notion similar to a *set*, of those objects. Rather than defining the behavior for each object individually, a class definition gives the behaviors that will be possessed by *all* objects in the class. The objects themselves are sometimes called *members* of the class, again alluding to the set-like qualities of a class. It is also sometimes said that a particular object is an *instance* of the class.

Using Java as an example language, these are the aspects of objects, as defined by a common class declaration, that will be of interest:

Name Each class has a name, a Java identifier. The name effectively becomes a **type name**, so is usable anywhere a type would be usable.

Constructor The *constructor* identifies the parameters and code for initializing an object. Syntactically it looks like a procedure and uses the name of the class as the constructor name. The constructor is called when an object is created dynamically or automatically. The constructor

does not return a value in its syntactic structure. A constructor is always called by using the Java `new` operator. The result of this operator is a *reference* to the object of the class thus created.

Methods Methods are like procedures that provide the interface between the object and the program using the object. As with ordinary procedures, each method has an explicit return type, or specifies the return type **void**.

Variables Each time an object is created as a "member" of a class, the system allocates a separate set of variables internal to it. These are accessible to each of the methods associated with the object without explicit declaration inside the method. **That is, the variables local to the object appear as if global to the methods, without necessitating re-declaration.**

The reason for the emphasis above is that use of objects can provide a convenience when the number of variables would otherwise become too large to be treated as procedure parameters and use of global variables might be a temptation.

7.5 Static vs. Instance Variables

An exception to having a separate copy of a variable for each object occurs with the concept of **static variable**. When a variable in a class is declared `static`, there is only *one* copy shared by all objects in the class. For example, a static variable could keep a *count* of the number of objects allocated in the class, if that were desired. For that matter, a static variable could maintain a list of references to all objects created within the class.

When it is necessary to distinguish a variable from a static variable, the term **instance variable** is often used, in accordance with the variable being associated with a particular object instance. Sometimes static variables are called **class variables**.

Similarly, a **static method** is one that does not depend on instance variables, and thus not on the state of the object. A static method may depend on static variables, however. Static methods are thus analogous to procedures, or possibly functions, in ordinary languages.

A final note on classes concerns a thread in algorithm development. It has become common to present algorithms using **abstract data types** (**ADT**s), which are viewed as mathematical structures accompanied by procedures and functions for dealing expressly with these structures. For example, a typical ADT might be a *set*, accompanied by functions for constructing sets, testing membership, forming unions, etc. Such structures are characterized by the behavioral interaction of these functions, rather than by the internal representation used to implement the structure.

> Classes seem to be a very appropriate tool for defining ADTs and enforcing the disciplined use of their functions and procedures.

7.6 Example – A Stack Class

We now illustrate these points on a specific ADT or class, a class `Stack`. A stack is simply an ordered sequence of items with a particular discipline for accessing the items:

> The order in which items in a **stack** are removed is the reverse from the order in which they were entered.
>
> This is sometimes called the LIFO (last-in, first-out) property.

Regardless of how the stack is implemented, into the stack is not part of the discipline. The `Stack` class will be specified in Java:

```
class Stack
{

// .... all variables, constructors, and methods
//      used for a Stack are declared here ....

}
```

We postulate methods `push`, `pop`, `and empty` for putting items into the stack, removing them, and for testing whether the stack is empty. Let's suppose that the items are integers, for concreteness.

```
class Stack
{
void push(int x)
  {
  // .... defines how to push x ....
  }

int pop()
  {
  // .... defines how to return top of the stack ....
  return .... value returned ....;
  }

boolean isEmpty()
  {
  // .... defines how to determine emptiness ....
  return ....;
  }
}
```

Method `push` does not return any value, hence the `void`. Method `pop` does not take any argument. Method `empty` returns a boolean (`true` or `false`) value indicating emptiness:

Java allows inclusion of a static method called main with each class. For one class, this will serve as the main program that is called at the outset of execution. For any class, the main method may be used as a test program, which is what we will do here.

```
class Stack
{
// .... other methods defined here

public static void main(String arg[])
  {
  int limit = new Integer(arg[0]).intValue();
  Stack s = new Stack(limit);
  for( int i = 0; i < limit; i++ )
    {
    s.push(i);
    }
  while( !s.isEmpty() )
    {
    System.out.println(s.pop());
    }
  }
}
```

The keyword `static` defines main as a static method. The keyword `public` means that it can be called externally. The argument type `String .... []` designates an array of strings indicating what is typed on the command line; each string separated by space will be a separate element of the array. The line

```
int limit = new Integer(arg[0]).intValue();
```

converts the first command-line argument to an `int` and assigns it to the variable `limit`. The line

```
Stack s = new Stack(limit);
```

uses the Stack constructor, which we have not yet defined, to create the stack. Our first approach will be to use an array for the stack, and the argument to the constructor will say how large to make that array, at least initially.

A second use of limit in the test program is to provide a number of times that information is pushed onto the stack. We see this number being used to control the first `for` loop. The actual pushing is done with

```
s.push(i);
```

Here `s` is the focal stack object, `push` is the method being called, and `i` is the actual argument.

The while loop is used to pop the elements from the stack as long as the stack is not empty. Since by definition of the stack discipline items are popped in reverse order from the order in which they are pushed, the output of this program, if called with a command-line argument of 5, should be

```
4
3
2
1
0
```

indicating the five argument values 0 through 4 that are pushed in the `for` loop.

7.7 Stack Implementation

Now we devote our attention to *implementing* the stack. Having decided to use an array to hold the integers, we will need some way of indicating how many integers are on the stack at a given time. This will be done with a variable `number`. The value of `number-1` then gives the index within the array of the last item pushed on the stack. Likewise, the value of `number` indicates the first available position onto which the next integer will be pushed, if one is indeed pushed before the next pop. The figure below shows the general idea.

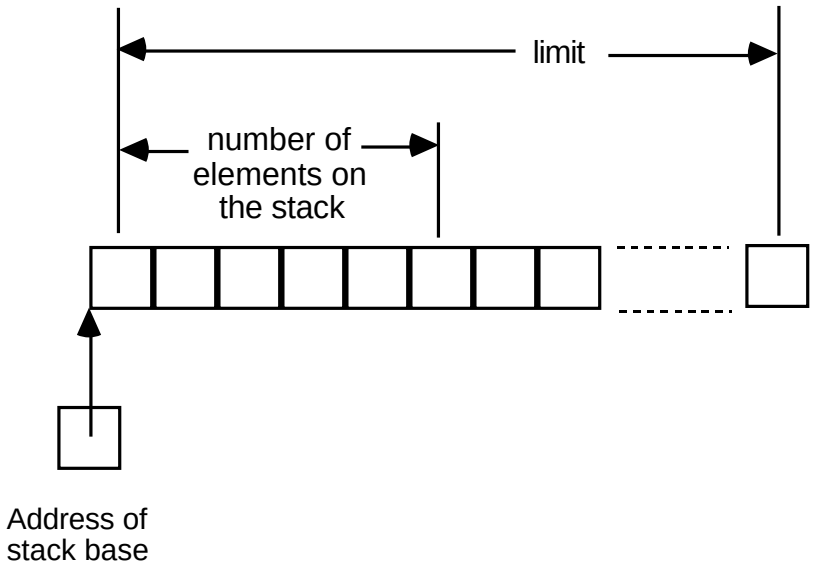

Figure 82: Array implementation of a stack

Here is how our `Stack` class definition now looks, after defining the variables `array` and `number` and adding the constructor `Stack`:

```
class Stack
{
int number;               // number of items in stack
int array[ ];             // stack contents

Stack(int limit)
  {
  array = new int[limit]; // create array
  number = 0;             // stack contains no items yet
  }
....
}
```

Note that the `new` operator is used to create an array inside the constructor. The array as declared is only a type declaration; there is no actual space allocated to it until the `new` operator is used. Although an array is technically not an object, arrays have a behavior that is very much object-like. If the `new` were not used, then any attempt to use the array would result in a terminal execution error.

Now we can fill in the methods `push`, `pop`, and `empty`:

```
void push(int x)
  {
  array[number++] = x;     // put element at position number
  }                        // and increment

int pop()
  {
  return array[--number];  // decrement number and take element
  }

boolean isEmpty()
  {
  return number == 0;      // see if number is 0
  }
```

Note that `number++` means that we increment number *after* using the value, and `--number` means that we decrement number *before* using the value. These are the correct actions, in view of our explanation above.

We can now test the complete program by

```
java -cs Stack 5
```

The argument `-cs` means to compile the source first. A command line is treated as a sequence of strings. The string `"5"`, the first after the name of class whose main is to be invoked, is the first command-line argument, which becomes `arg[0]` of the program.

7.8 Improved Stack Implementation

This first sketch of a Stack class leaves something to be desired. For one thing, if we try to push more items onto the stack than limit specified, there will be a run-time error, or *stack overflow*. It is an annoyance for a program to have to be aware of such a limit. So our first enhancement might be to allow the stack array to grow if more space is needed.

> A program should not abort an application due to having allocated a fixed array size. The program should make provisions for extending arrays and other structures as needed, unless absolutely no memory is left.

In other words, programs that preset the size of internal arrays arbitrarily are less robust than ones that are able to expand those arrays. In terms of object-oriented programming, each object should manage its own storage requirements to preclude premature failure due to lack of space.

To design our class with a growing array, we will add a new method `ensure` that ensures there is enough space before insertion into the array is attempted. If there is not enough space, then `ensure` will replace the array with a larger one.

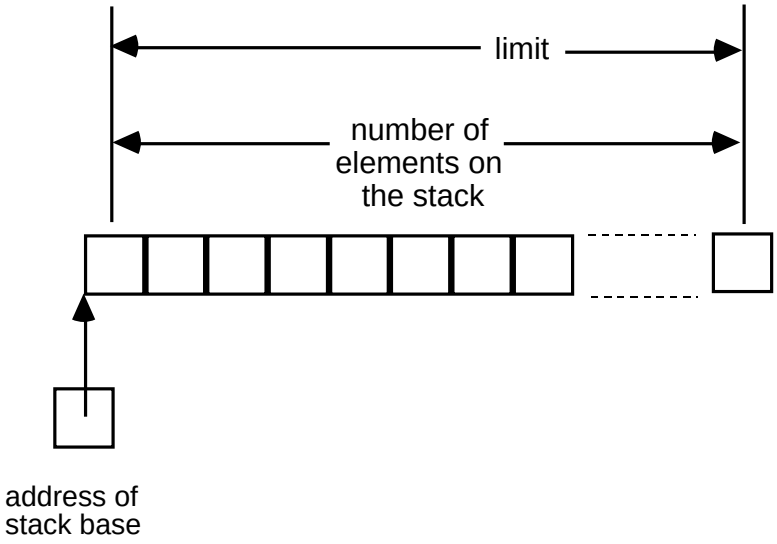

Figure 83: Full stack before extension

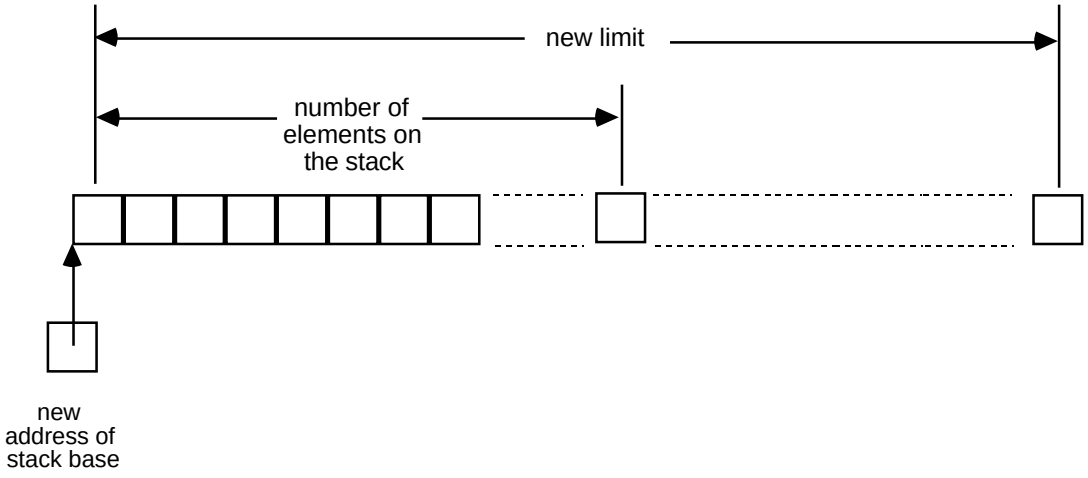

Figure 84: Stack after extension

The incremental size will be controlled by a variable `increment` in the object. For convenience, we will initialize the increment to be the same as the initial limit. Note that we also must keep track of what the current limit is, which adds another variable. We will call this variable `limit`. The value of the variable `limit` in each object will be distinguished from the constructor argument of the same name by qualifying it with `this`, which is a Java keyword meaning *the current object.*

```
class Stack
{
int number;                // number of items in the stack
int limit;                 // limit on number of items in the stack
int increment;             // incremental number to be added
int array[ ];              // stack contents

Stack(int limit)
  {
  this.limit = limit;      // set instance variable to argument value
  increment = limit;       // use increment for limit
  array = new int[limit];  // create array
  number = 0;              // stack contains no items initially
  }

void ensure()              // make sure push is possible
  {
  if( number >= limit )
    {
    int newArray[] = new int[limit+increment];  // create new array
    for( int i = 0; i < limit; i++ )
      {
      newArray[i] = array[i]; // copy elements in stack
      }
    array = newArray;         // replace array with new one
    limit += increment;       // augment the limit
    }
  }
```

```
void push(int x)
  {
  ensure();
  array[number++] = x; // put element at position number, increment
  }
....                   // other methods remain unchanged
}
```

Exercises

1 • Add to the code a method that returns the number of items currently on the stack.

2 • Add to the stack class a second constructor of two arguments, one giving the initial stack size and one giving the increment.

3 •• Suppose we wished for the stack to reduce the size of the allocated array whenever number is less than limit - increment. Modify the code accordingly.

4 ••• Change the policy for incrementing stack size to one that increases the size by a factor, such as 1.5, rather than simply adding an increment. Do you see any advantages or disadvantages of this method vs. the fixed increment method?

5 ••• For the methods presented here, there is no requirement that the items in a stack be in a contiguous array. Instead a linked list could be used. Although a linked list will require extra space for pointers, the amount of space allocated is exactly proportional to the number of items on the stack. Implement a stack based on linked lists.

6 •••• Along the lines of the preceding linked list idea, but rather than linking individual items, link together *chunks* of items. Each chunk is an array. Thus the overhead for the links themselves can be made as small as we wish by making the chunks large enough. This stack gives the incremental allocation of our example, but does not require copying the array on each extension. As such, it is superior, although its implementation is more complicated. Implement such a stack.

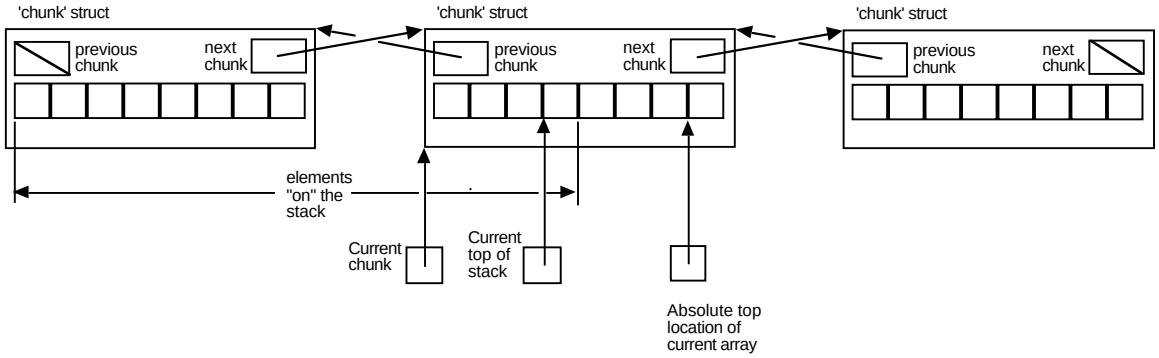

Figure 85: A stack using chunked array allocation

7.9 Classes of Data Containers

A stack is just one form of **data container**, which in turn is just one form of ADT. Different types of container can be created depending on the discipline of access we wish to have. The stack, for example, exhibits a last-in, first-out (LIFO) discipline. Items are extracted in the reverse order of their entry. In other words, extraction from a stack is "youngest out first". A different discipline with a different set of uses is a **queue**. Extraction from a queue is "oldest out first", or first-in, first-out (FIFO). A queue's operations are often called **enqueue** (for inserting) and **dequeue** for extraction. Yet another discipline uses an ordering relation among the data values themselves: "minimum out first". Such a discipline is called a **priority queue**. The figure below illustrates some of these disciplines. A discipline, not shown, which combines both and stack and queue capabilities is a **deque**, for "double-ended queue". The operations might be called **enqueue_top**, **enqueue_bottom**, **dequeue_top**, and **dequeue_bottom**.

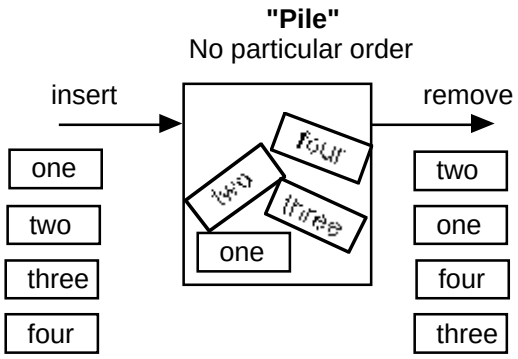

Figure 86: A container with no particular discipline

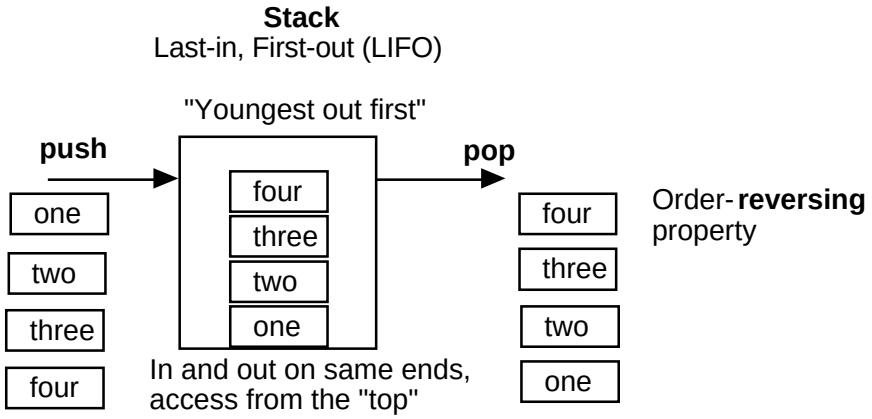

Figure 87: A container with a stack discipline

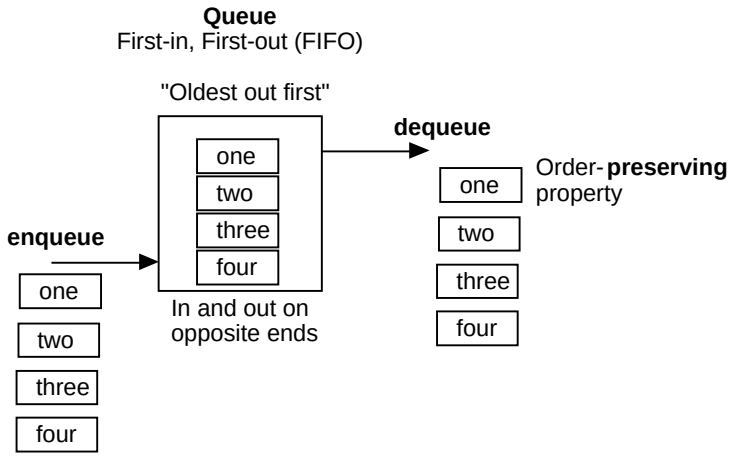

Figure 88: A container with a queue discipline

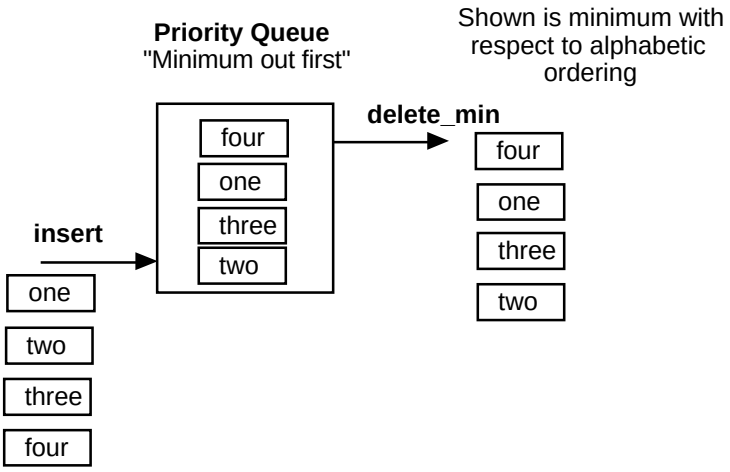

Figure 89: A container with a priority queue discipline

7.10 Implementing a Queue as a Circular Array

Consider implementing a queue data abstraction using an array. The straightforward means of doing this is to use two indices, one indicating the oldest item in the array, the other indicating the youngest. Enqueues are made near the youngest, while dequeues are done near the oldest. The following diagram shows a typical queue state:

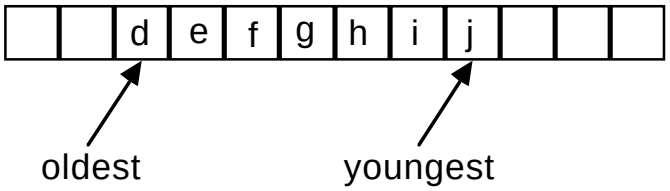

Figure 90: Queue implemented with an array, before enqueuing k

If the next operation is enqueue(k), then the resulting queue will be:

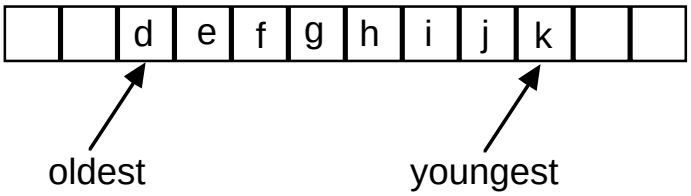

Figure 91: Queue implemented with an array, after enqueuing k

From this state, if the next operation is dequeue(), then d would be dequeued and the resulting state would be:

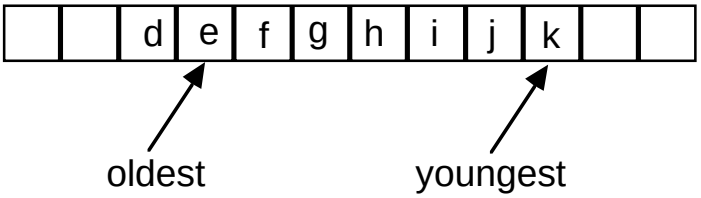

Figure 92: Queue implemented with an array, after dequeuing d

Of course, the value being dequeued need not actually be obliterated. It is the pair of indices that tell us which values in the queue are valid, not the values in the cells themselves. Things get more interesting as additional items are enqueued, until youngest points to the top end of the array. Note that there still may be available space in the queue, namely that below oldest. It is desirable to use that space, by "wrapping around" the pointer to the other end of the queue. Therefore, after having enqueued l, m, n, o, and p, the queue would appear as:

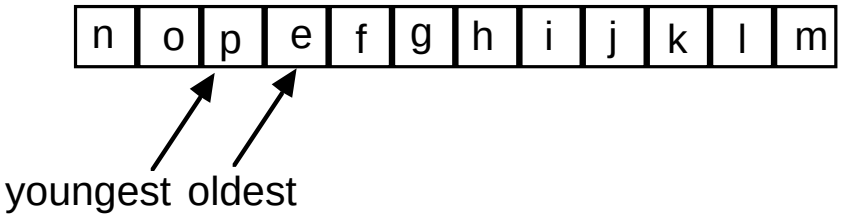

Figure 93: Queue implemented with an array, after wrap-around

When the index values meet, as they have here, we need to allocate more space. The simplest way to do this is to allocate a new array and copy the current valid values into it. From the previous figure, attempting to enqueue q now would cause an overflow condition to result. Assuming we can double the space allocated were the same as that in the original queue, we would then have the following, or one of its many equivalents:

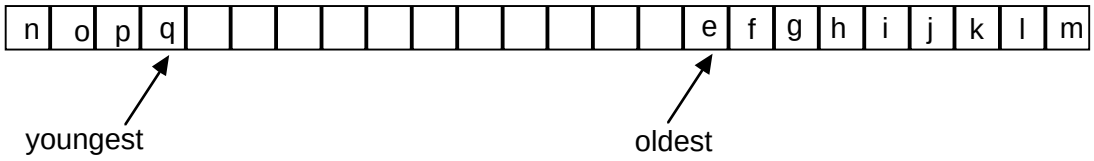

Figure 94: Queue implemented with an array, after space extension

How do we detect when additional allocation is necessary? It is tempting to try to use the relationship between the values of the two indices to do this. However, this may be clumsy to manage (why?). A simpler technique is to just maintain a count of the number in the queue at any given time and compare it with the total space available. Maintaining the number might have other uses as well, such as providing that information to the clients of the queue through an appropriate call.

Exercises

1 ••• Construct a class definition and implementation for a queue of items. Use the stack class example developed here as a model. Use circular arrays as the implementation, so that it is not necessary to extend the storage in the stack unless all space is used up. Be very careful about copying the queue contents when the queue is extended.

2 ••• Construct a class definition and implementation for a *deque* (*double-ended queue*, in which enqueuing and dequeuing can take place at either end). This should observe the same storage economies defined for the queue in the previous exercise.

7.11 Code Normalization and Class Hierarchies

"Normalization" is the term (borrowed from database theory) used to describe an on-going effort, during code development, of concentrating the specification of functionality in as few places as possible,. It has also come to be called “factoring”.

> **Reasons to "normalize" code:**
>
> **Intellectual economy**: We would prefer to gain an understanding of as much functionality as possible through as little code reading as possible.
>
> **Maintainability/evolvability**: Most programs are not written then left alone. Programs that get used tend to evolve as their users desire new features. This is often done by building on existing code. The fewer places within the code one has to visit to make a given conceptual change, the better.

An example of normalization we all hopefully use is through the *procedure* concept. Rather than supporting several segments of code performing similar tasks, we try to generalize the task, package it as a procedure, and replace the would-be separate code segments by procedure calls with appropriate parameters.

Other forms of normalization are:

> Using identifiers to represent key constants.
>
> The *class* concept, as used in object-oriented programming, which encourages procedural normalization by encapsulating procedures for specific abstract data types along with the specifications of those data types.

As we have seen, classes can be built up out of the raw materials of a programming language. However, an important leveraging technique is to build classes out of other classes as well. In other words, an object X can employ other objects Y, Z, ... to achieve X's functionality. The programmer or class designer has a number of means for doing this:

- Variables in a class definition can be objects of other classes. We say that the outer object is **composed** of, or **aggregated** from, the inner objects.

- A class definition can be directly based on the definition of another class (which could be based on yet another class, etc.). This is known as ***inheritance***, since the functionality of one class can be used by the other without a redefinition of this functionality.

As an example of composing an object of one class from an object of another, recall the stack example. The stack was built using an array. One of the functionalities provided for the array was extendability, the ability to make the array larger when more space is needed. But this kind of capability might be needed for many different types of container built from arrays. Furthermore, since the array extension aspect is the trickiest part of the code, it would be helpful to isolate it into functionality associated with the array, and not have to deal with it in classes built *using* the array. Thus we might construct a class `Array` that gives us the array access capability with extension and use this class in building other classes such as stacks, queues, etc. so that we don't have to reimplement this functionality in each class.

```
class Array
{
int increment;                // incremental number to be added
int array[ ];                 // actual array contents

Array(int limit)              // constructor
  {
  increment = limit;          // use increment for limit
  array = new int[limit];     // create actual array
  }

void ensure(int desiredSize)  // make sure size is at least desiredSize
  {
  if( desiredSize > array.length )
    {
    int newArray[] = new int[desiredSize]; // create new array
    for( int i = 0; i < array.length; i++ )
      {
      newArray[i] = array[i]; // copy elements
      }
    array = newArray;         // replace array with new one
    }
  }
}

class Stack             // Stack built using class Array
{
int number;                 // number of items in the stack
int increment;              // incremental number to be added
Array a;                    // stack contents

Stack(int limit)
  {
  a = new Array(limit);     // create array for stack
  increment = limit;        // use increment for limit
  number = 0;               // stack contains no items initially
  }
```

```
void ensure()               // make sure push is possible
  {
  if( number >= a.array.length )
    {
    a.ensure(a.array.length + increment);
    }
  }

void push(int x)
  {
  ensure();
  a.array[number++] = x; // put element at position number, increment
  }

int pop()
  {
  return a.array[--number]; // decrement number and take element
  }
....                         // other methods remain unchanged
}
```

Within class `Stack`, a reference to an object of class `Array` is allocated, here identified by `a`. Notice how the use of the `Array` class to implement the `Stack` class results in a net simplification of the latter. By moving the array extension code to the underlying `Array` class, there is less confusion in the `Stack` class itself. Thus using two classes rather than one results in a separation of concerns, which may make debugging easier.

7.12 Inheritance

A concept linked to that of class, and sometimes thought to be required in object-oriented programming, is that of *inheritance*. This can be viewed as a form of normalization. The motivation for inheritance is that different classes of data abstractions can have functions that are both similar among classes and ones that are different among classes. Inheritance attempts to normalize class definitions by providing a way to merge similar functions across classes.

> Inheritance entails defining a "parent class" that contains common methods and one or more child classes, each potentially with their separate functions, but which also *inherit* functions from the parent class.

With the inheritance mechanism, we do not have to recode common functions in each child class; instead we put them in the parent class.

The following diagram suggests inheritance among classes. At the programmer's option, the sets of methods associated with the child classes either augment or over-ride the methods in the parent class. This diagram is expressed in a standard known as UML (Unified Modeling Language).

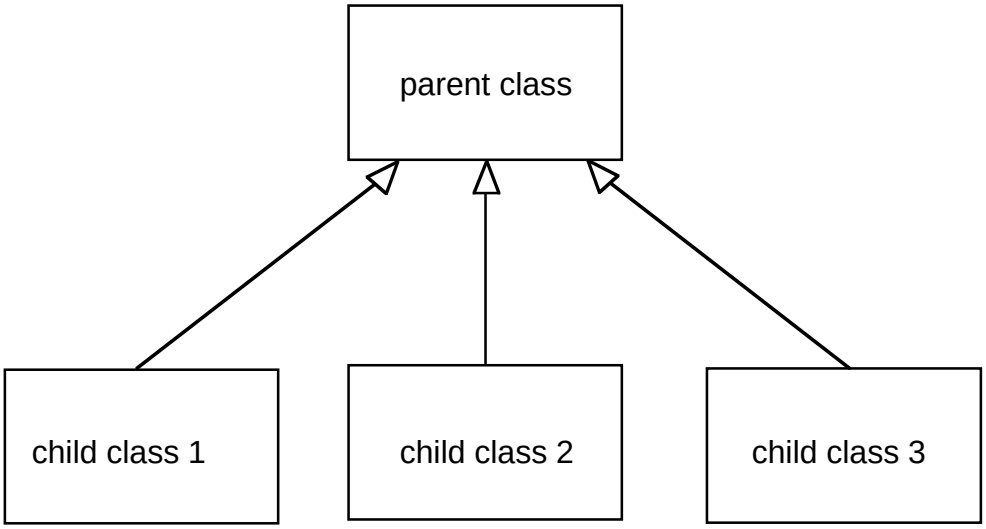

Figure 95: Inheritance among classes

This concept can be extended to any number of children of the same parent. Moreover, it can be extended to a class hierarchy, in which the children have children, etc. The terminology ***base class*** is also used for parent class and ***derived class*** for the children classes. It is also said that the derived class ***extends*** the base class (note that this is a different idea from extending an array).

Possible Advantages of Inheritance:

- Code for a given method in a base class is implemented once, but the method may be used in all derived classes.
- A method declared in a base class can be "customized" for a derived class by *over-riding* it with a different method.
- A method that accepts an object of the base class as an argument will also accept objects of the derived classes.

As an example, let's derive a new class `IndexableStack` from class `Stack`. The idea of the new class is that we can index the elements on the stack. However, indexing takes place from the top element downward, so it is not the same as ordinary array index. In other words, if `s` is an `IndexableStack`, `s.fromTop(0)` represents the top element, `s.fromTop(1)` represents the element next to the top, and so on.

The following Java code demonstrates this derived class. Note that the keyword *extends* indicates that this class is being derived from another.

```
class IndexableStack extends Stack
```

```
{
IndexableStack(int limit)
  {
  super(limit);
  }

int fromTop(int index)
  {
  return a.array[number-1-index];
  }

int size()
  {
  return number;
  }
}
```

Note the use of the keyword `super` in the constructor. This keyword refers to the object in the base class `Stack` that underlies the `IndexableStack`. The use of the keyword with an argument means that the constructor of the base class is called with this argument. In other words, whenever we create an `IndexableStack`, we are creating a `Stack` with the same value of argument `limit`.

Note the use of the identifiers `a` and `number` in the method `fromTop`. These identifiers are not declared in `IndexableStack`. Instead, they represent the identifiers of the same name in the underlying class `Stack`. Every variable in the underlying base class can be used in a similar fashion in the derived class.

Next consider the idea of *over-riding*. A very useful example of over-riding occurs in the class `Applet`, from which user applets are derived. An *applet* was intended to be an application program callable within special contexts, such as web pages. However, applets can also be run in a free-standing fashion. A major advantage of the class `Applet` is that there are pre-implemented methods for handling mouse events (down, up, and drag) and keyboard events. By creating a class derived from class `Applet`, the programmer can over-ride the event-handling methods to be ones of her own choosing, without getting involved in the low-level details of event handling. This makes the creation of interactive applets relatively simple.

The following example illustrates handling of mouse events by over-riding methods `mouseDown`, `mouseDrag`, etc. which are defined in class `Applet`. The reader will note the absence of a main program control thread. Instead actions in this program are driven by mouse events. Each time an event occurs, one of the methods is called and some commands are executed.

Another example of over-riding that exists in this program is in the update and paint methods. The standard applet protocol is that the program does not update the screen directly; instead, the program calls `repaint()`, which will call `update(g)`, where `g` is the applet's graphics. An examination of the code reveals that `update` is never called explicitly. Instead this is done in the underlying Applet code. The reason that `update` calls `paint` rather than doing the painting directly is that the applet also makes implicit

calls to `paint` in the case the screen needs repainting due to being covered then re-exposed, such as due to user actions involving moving the window on the screen.

```
// file:    miniMouse.java
// As mouse events occur, the event and its coordinates
// appear on the screen.

// This applet also prescribes a model for the use of double-buffering
// to avoid flicker: drawing occurs in an image buffer, which is then
// painted onto the screen as needed. This also simplifies drawing,
// since each event creates a blank slate and then draws onto it.

import java.applet.*;           // applet class
import java.awt.*;              // Abstract Window Toolkit


public class miniMouse extends Applet
  {
  Image image;          // Image to be drawn on screen by paint method
  Graphics graphics;    // Graphics part of image, acts as buffer


  // Initialize the applet.

  public void init()
    {
    makeImageBuffer();
    }


   // mouseDown is called when the mouse button is depressed.

  public boolean mouseDown(Event e, int x, int y)
    {
    return show("mouseDown", x, y);
    }


   // mouseDrag is called when the mouse is dragged.

  public boolean mouseDrag(Event e, int x, int y)
    {
    return show("mouseDrag", x, y);
    }


   // mouseUp is called when the mouse button is released.

  public boolean mouseUp(Event v, int x, int y)
    {
    return show("mouseUp", x, y);
    }
```

```
 // mouseMove is called when the mouse moves without being dragged

public boolean mouseMove(Event v, int x, int y)
  {
  return show("mouseMove", x, y);
  }

// show paints the mouse coordinates into the graphics buffer

boolean show(String message, int x, int y)
  {
  clear();
  graphics.setColor(Color.black);
  graphics.drawString(message +
                           " at (" + x + ", " + y + ")", 50, 100);
  repaint();
  return true;
  }


// update is implicitly called when repaint() is called
// g will be bound to the Graphics object in the Applet,
// not the one in the image. paint will draw the image into g.

public void update(Graphics g)
  {
  paint(g);
  }


// paint(Graphics) is called by update(g) and whenever
// the screen needs painting (such as when it is newly exposed)

public void paint(Graphics g)
  {
  g.drawImage(image, 0, 0, null);
  }


// clear clears the image

void clear()
  {
  graphics.clearRect(0, 0, size().width, size().height);
  }

// Make image buffer based on size of the applet.

void makeImageBuffer()
  {
  image = createImage(size().width, size().height);
  graphics = image.getGraphics();
  }
}
```

Drawbacks of Inheritance

While inheritance can be a wonderful time-saving tool, we offer these cautions:

Possible drawbacks of inheritance:

- Once an inheritance hierarchy is built, functionality in base classes cannot be changed unless the impact of this change on derived classes is clearly understood and managed.
- The user of a derived class may have to refer to the base class (and its base class, etc., if any) to understand the full functionality of the class.

There is a fine art in developing the inheritance hierarchy for a large library; each level in the hierarchy should represent carefully-chosen abstractions.

7.12 Inheritance vs. Composition

Inheritance is just one of two major ways to build hierarchies of classes. The second way, which is called composition, is for the new class to make use of one or more objects of other classes. Although these two ways appear similar, they are actually distinct. For example, whereas inheritance adds a new layer of functionality to that of an existing class, composition *uses* functionality of embedded objects but does not necessarily provide similar functionality to the outside world. The following diagram is meant to suggest this distinction.

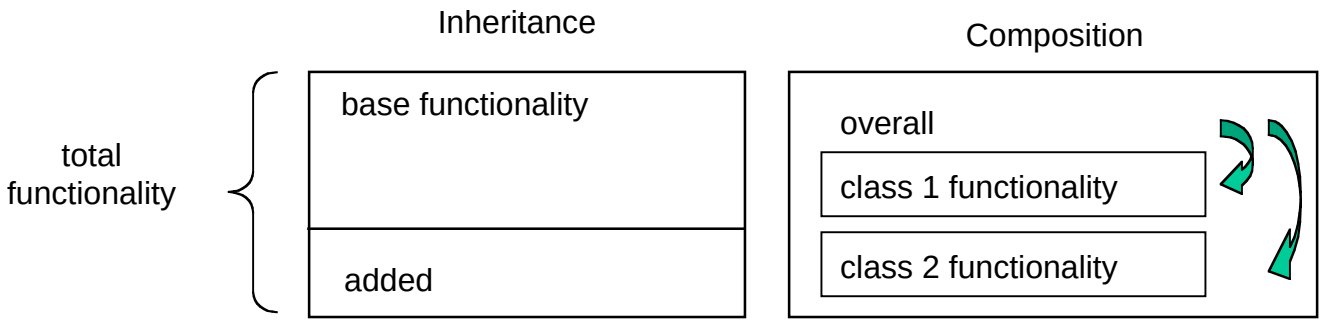

Figure 96: Inheritance vs. Composition

With composition, if the outer classs wishes to provide functionality of inner clases to its clients, it must explicitly provide methods for that purpose. For example, an alternate way to have built the `Stack` class above would be to have `Stack` inherit from `Array`, rather than be composed of an `Array`. In this case, methods such as extend that are available in `Array` would automatically be available in `Stack` as well. Whether or not this is desirable

would depend on the client expectations for `Stack` and other considerations. An advantage is that there is less code for the extension; a disadvantage is that it exposes array-like functionality in the `Stack` definition, upon which the client may come to rely.

Composition should also not be confused with function composition, despite there being a similarity. Yet another construction similar to composition is called aggregation. The technical distinction is that with composition the component objects are not free-standing but are instead a part of the composing object, whereas with aggregation, the components exist independently from the aggregating object. This means that a single object may be aggregated in more than one object, much like structure sharing discussed in Chapter 2.

Exercises

1 ••• Using inheritance from class `Array`, construct a class `BiasedArray` that behaves like an `Array` except that the lower limit is an arbitrary integer (rather than just 0) called the *bias*. In this class, an indexing method, say `elementAt`, must be used in lieu of the usual [...] notation so that the bias is taken into account on access.

2 ••• Code a class `Queue` using class `Array` in two different ways, one using composition and the other using inheritance. Use the *circular array* technique described earlier.

3 ••• Code a class `Deque` using class `Array`.

4 ••• Using aggregation, construct a class `Varray` for virtually concatenating arrays, using the *Principle of Virtual Contiguity* described in *Implementing Information Structures*. One of the constructors for this class should take two arguments of class `Array` that we have already presented and provide a method `elementAt` for indexing. This indexing should translate into indexing for an appropriate one of the arrays being concatenated. Also provide constructors that take one array and one virtual array and a constructor that takes two virtual arrays. Provide as much of the functionality of the class array as seems sensible. The following diagram suggests how varrays work:

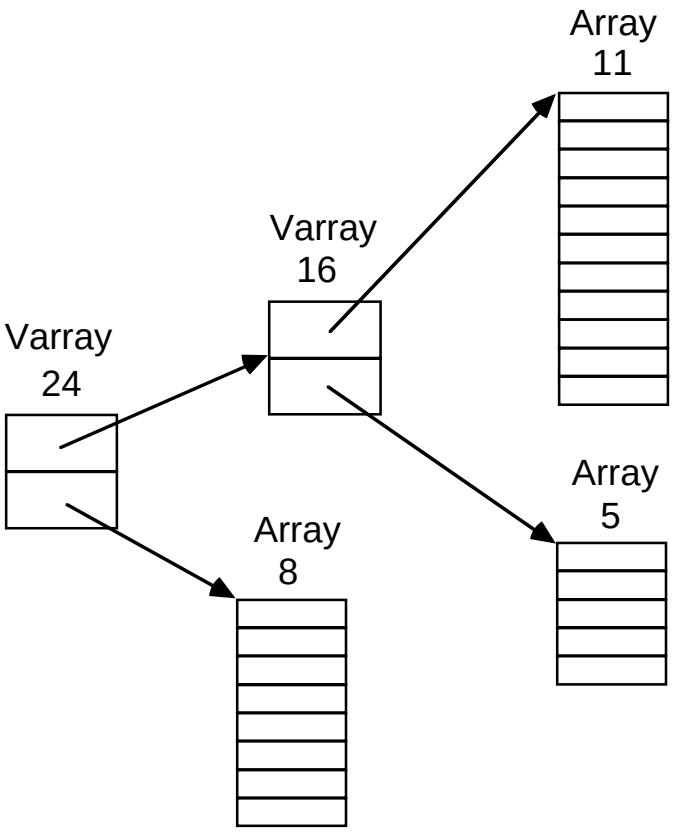

Figure 97: Aggregated objects in a class of virtual arrays

5 •••• Arrays can be constructed of any dimension. Create a class definition that takes the number of dimensions as an *argument* to a constructor. Use a single-dimension array of indices to access arrays so constructed.

7.13 The *is-a* Concept and Sub-Classing

When a class is constructed as a derived class using inheritance, the derived class inherits, by default, the characteristics of the underlying base class. Unless the essential characteristics are redefined, in a sense every derived class object ***is a*** base class object, since it has the capabilities of the base class object but possibly more. For example, an `IndexableStack` ***is a*** `Stack`, according to our definition. This is in the same sense that additional capabilities are possessed by people and things. For example, if the base class is *person* and the derived class is *student*, then a student has the characteristics of a person and possibly more. Another way of saying this is that class *student* is a **sub-class** of class *person*. Every student is a person but not necessarily conversely.

It is common to find class hierarchies in which branching according to characteristics occurs. The programmer should design such hierarchies to best reflect the enterprise underlying the application. For example, if we are developing a computer window system, then there might be a base class window with sub-classes such as:

text_window

graphics_window

window_with_vertical_scrollbar

window_with_horizontal_and_vertical_scrollbars

text_window_with_vertical_scrollbar

graphics_window_with_horizontal_and_vertical_scrollbars

and so on. It is the job of the designer to organize these into a meaningful hierarchy for use by the client and also to do it in such a way that as much code functionality as possible is shared through inheritance.

7.14 Using Inheritance for Interface Abstraction

The type of inheritance discussed could be called **implementation inheritance**, since the objects of the base class are being used as a means of implementing objects of the derived class. Another type of inheritance is called **interface inheritance**. In this form, the *specification* of the interface methods is what is being inherited. As before, each object in the derived class still *is an* object in the base class, so that a method parameter could specify an object of base type and any of the derived types could be passed as a special case.

In Java, there is a special class-like construct used to achieve interface inheritance: The base class is called an `interface` rather than a class. As an example, consider the two container classes `Stack` vs. `Queue`. Both of these have certain characteristics in common: They both have methods for putting data in, removing data, and checking for emptiness. In certain domains, such as search algorithms, a stack or a queue could be used, with attendant effects on the resulting search order.

We could consider both `Stack` and `Queue` to be instances of a common interface, say `Pile`. We'd have to use the same names for addition and removal of data in both classes. So rather than use `push` and `enqueue`, we might simply use `add`, and rather than use `pop` and `dequeue`, we might use `remove`. Our interface declaration might then be:

```
interface Pile
{
void add(int x);

int remove();

boolean isEmpty();
}
```

Note that the interface declaration only declares the types of the methods. The definition of the methods themselves are in the classes that implement the interface. Each such class must define all of the methods in the interface. However, a class may define other methods as well. Each class will define its own constructor, since when we actually create a `Pile`, we must be specific about how it is implemented.

The following shows how class `Stack` might be declared to implement interface `Pile`:

```
class Stack implements Pile  // Stack built using class Array
{
int number;                     // number of items in the stack
int increment;                  // incremental number to be added
Array a;                        // stack contents

Stack(int limit)
  {
  a = new Array(limit);         // create array for stack
  increment = limit;            // use increment for limit
  number = 0;                   // stack contains no items initially
  }

void ensure()               // make sure add is possible
  {
  if( number >= a.array.length )
    {
    a.ensure(a.array.length + increment);
    }
  }

public void add(int x)
  {
  ensure();
  a.array[number++] = x; // put element at position number and increment
  }

public int remove()
  {
  return a.array[--number]; // decrement number and take element
  }

public boolean isEmpty()
  {
  return number == 0;       // see if number is 0
  }
}
```

Note the `public` modifiers before the methods that are declared in the interface. Since those methods are by default public, these modifiers are required in the implementing class. Similarly we might have an implementation of class `Queue`:

```
class Queue implements Pile     // Queue built using class Array
{
int number;                 // number of items in the Queue
int increment;              // incremental number to be added
int oldest;                 // index of first element to be removed
int newest;                 // index of last element added
Array a;                    // Queue contents

Queue(int limit)
  {
  a = new Array(limit);     // create array for Queue
  increment = limit;        // use increment for limit
  number = 0;               // Queue contains no items initially
  oldest = 0;
  newest = -1;
  }
... definition of methods add, remove, empty ...
}
```

Now let's give a sample method that uses a `Pile` as a parameter. We do this in the context of a test program for this class. We are going to test both `Stack` and `Queue`:

```
class TestPile
{
public static void main(String arg[])
  {
  int limit = new Integer(arg[0]).intValue();
  int cycles = new Integer(arg[1]).intValue();

  testPile(new Stack(limit), cycles);
  testPile(new Queue(limit), cycles);
  }

static void testPile(Pile p, int cycles)
  {
  for( int i = 0; i < cycles; i++ )
    {
    p.add(i);
    }
  while( !p.isEmpty() )
    {
    System.out.println(p.remove());
    }
  }
}
```

The important thing to note here is the type of the first parameter `Pile` to `testPile`. Since both `Stack` and `Queue` are special cases of pile, we can use either type as a parameter to `testPile`, as shown in main above.

7.15 Abstract Classes

An idea similar to implementation of an interface is that of an *abstract base class*. In Java terminology, a base class is *abstract* if it is intended to tie together similar derived classes, but there is to be no direct creation of objects of the base class itself. Unlike an interface, objects can actually exist in the abstract class. There might be methods and constructors defined in the abstract class as well. However, similar to an interface, those objects are never created by calling their constructors directly. Instead, their constructors are called in the constructors of classes derived from the abstract class.

Abstract classes can also contain `abstract` method declarations. These methods are similar to the declarations in an interface; they do not specify an implementation; instead this is done in the derived classes.

An interesting example of abstract classes is in a shape-drawing program. There are typically several different types of shapes that can be drawn with the mouse, for example:

Box
Oval
Line

Each of these is an object of a different class. Each has a different way of drawing itself. At the same time, there are certain things we wish to do with shapes that do not need to differentiate between these individual classes. For example, each shape has some reference position that defines a relative offset from a corner of the screen. We would expect to find a `move` method that changes this reference position, and that method will be the same for all shapes.

Our inheritance diagram would appear as in Figure 98, with `Shape` being the abstract class. `Shape` would have an abstract method `draw`, which would be defined specially by each shape class, and a concrete method `move` that changes its reference coordinates.

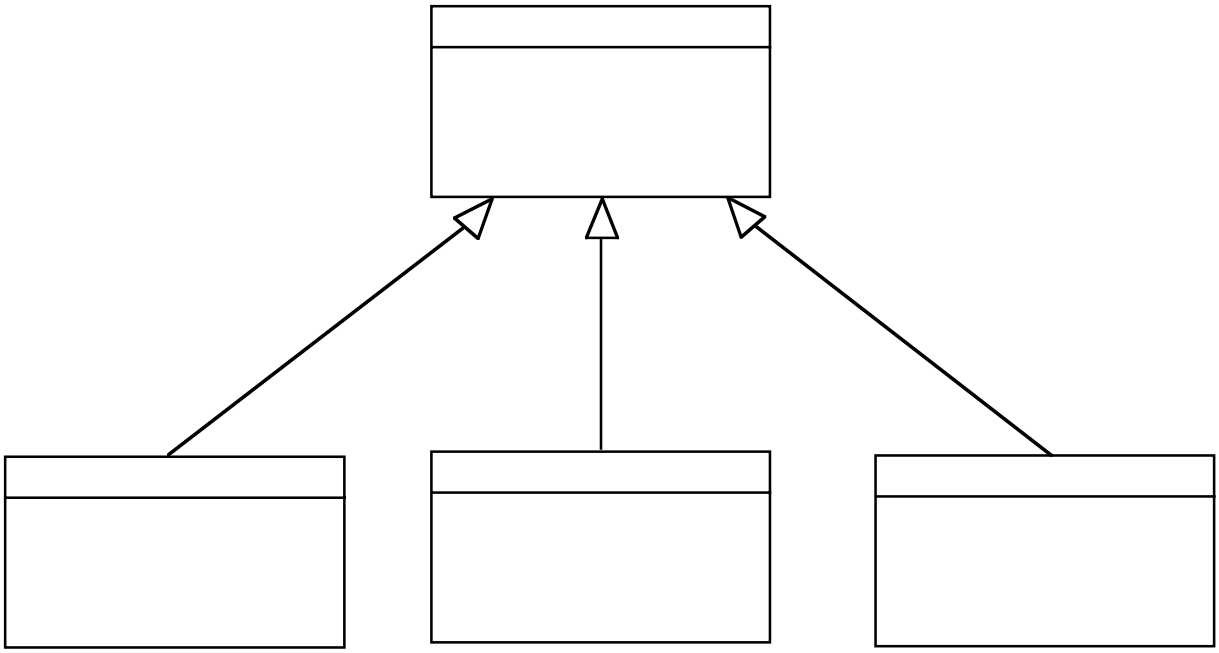

Figure 98: Implementing shapes with inheritance

```
abstract class Shape
  {
  int x, y;                          // coordinates

  Shape(int x, int y)                // constructor
    {
    this.x = x;
    this.y = y;
    }

  void move(int x, int y)            // concrete method
    {
    this.x = x;
    this.y = y;
    }

  abstract void draw(int x, int y);  // defined in derived classes
  }

class Box extends Shape
  {
  Box(int x, int y, ....)            // Box constructor
    {
    super(x, y);                     // call base constructor
    }

  void draw(int x, int y)            // draw method for Box
    {
      ....
    }
  }
```

```
class Oval extends Shape
  {
  Oval(int x, int y, ....)          // Oval constructor
    {
    super(x, y);                    // call base constructor
    }

  void draw(int x, int y)           // draw method for Oval
    {
      ....
    }
  }
....
```

As in the case of interfaces, where both `Stack` and `Queue` could be used for a `Pile`, here both `Box` and `Oval` can be used for a `Shape`. If we have a variable of type `Shape`, we can call its draw method without knowing what kind of shape it is:

```
Box box = new Box(....);
Oval oval = new Oval(....);

Shape shape;

shape = box;
shape.draw(....);

shape = oval;
shape.draw(....);
```

Exactly the same statement may be used to draw either kind of object.

An interesting further step would be to add a class `Group` as a sub-class of `Shape`, with the idea being that a `Group` could hold a list of shapes that can be moved as a unit.

Another example of a class hierarchy with some abstract classes occurs in the Java Abstract Window Toolkit (awt). We show only a portion of this hierarchy, to give the reader a feel for why it is structured as it is. The classes in this hierarchy that we'll mention are:

Component: An abstract class that contains screen graphics and methods to paint the graphics as well as to handle mouse events that occur within.

Specific sub-classes of Component that do not contain other components include:

TextComponent, which has sub-classes
TextField
TextArea.

Label
Scrollbar
Button

List (a type of menu)

Container: An abstract sub-class of Component containing zero or more components. Two specific sub-classes of Container are:

Window, a bordered object that has sub-classes
Frame, which is a Window with certain added features
Dialog

Panel, which has a sub-class **Applet**. A panel has methods for catching mouse events that occur within it.

7.16 The *Object* Class

In Java, there is a single master class from which all classes inherit. This class is called `Object`. If we view the inheritance hierarchy as a tree, then `Object` is the root of the tree.

One approach to creating container classes for different classes of objects is to make the contained type be of class `Object`. Since each class is derived from class `Object`, each object of any class *is* a member of class `Object`. There are two problems with this approach:

1. Not everything to be contained is an object. For example, if we wanted to make a stack of `int`, this type is not an object.

2. Different types of objects can be stored in the same container. This might lead to confusion or errors. The code for accessing objects in the container may get more complicated by the fact that the class of the object will need to be checked dynamically.

Problem 1 can be addressed by *wrapper classes*, to be discussed subsequently. In order to implement checking called for in problem 2, we can make use of the built-in Java operator `instanceof`. An expression involving the latter has the following form:

Object-Reference `instanceof` *Class-Name*

As an example, we could have constructed our class `Stack` using `Object` rather than `int` as the contained type. Then the value returned by the `pop` method would be `Object`. In order to test whether the object popped is of a class `C`, we would have code such as:

```
Stack s = new Stack();
      ....
Object ob = s.pop();

if( ob instanceOf C )
  { .... }
```

7.17 Wrapper Classes

In Java, primitive data items such as `int` and `float` are not objects. However, it is frequently desired to treat them as such. For example, as discussed above, rather than create a different stack class for each different type of datum that we may wish to put in stacks, we could create a single class of type `Object`. The Java language libraries provide classes that serve the purposes of making objects out of primitive objects. But if they didn't, we could still define them ourselves. Frequently used wrapper classes, and the type of data each contains, are:

Wrapper class	Wrapped Type
Integer	int
Long	long
Float	float
Double	double
Char	char
Boolean	boolean

Each wrapper object contains a single object of the wrapped type. Also, these objects are *immutable*, meaning that their values cannot be changed once created.

Methods of the wrapper classes provide ways to extract the wrapped data, and also to construct objects from other types, such as from `String`s. Consult the reference manual for details. The first four of the wrappers mentioned above extend an abstract class called `Number`. By using `Number` as a type, one can extract information using methods such as `floatValue`, without requiring knowledge of whether the actual number is an `Integer`, `Long`, `Float`, or `Double`.

```
Object ob = s.pop();

if( ob instanceOf Number )
  {
  float v = ((Number)ob).floatValue();
  ....
  }
```

7.18 Copying Objects

What does it *mean* to copy an object? Suppose, for example, an object is a list of lists. Does copying this object mean copying the entire list but allowing the elements to be shared among both copies, or does it mean that the elements are copied too?

By **shallow copying**, we mean copying only the references to the elements of the list. A consequence of shallow copying is that the lists cells themselves are shared between the original and the copy. This might lead to unintended side-effects, since a change made in the copy can now change the original. By **deep copying**, we mean copying all of the elements in the list and, if those elements are lists, copying them, and so on. If the list elements are each deep copied recursively, then there is no connection between the copy and the original, other than they have the same shape and atomic values. Obviously we can have types of copying between totally shallow and totally deep copying. For example, we could copy the list elements, but shallow copy them. If those elements are only atoms there would be no sharing. If they are pointers to objects, there still would be some sharing.

Below we illustrate shallow vs. deep copying of an object that references an array. For example, this could be the implementation of a stack as discussed earlier.

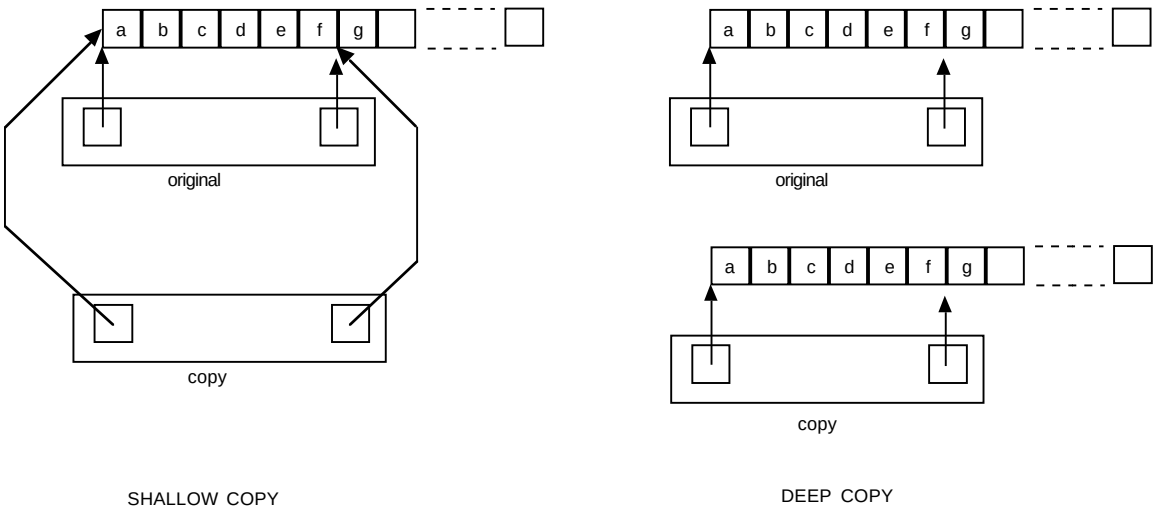

Figure 99: ***Shallow vs. deep copying***

7.19 Equality for Objects

Similar to the copying issue, there is the issue of how objects are compared for equality. We could just compare references to the objects, which would mean that two objects are equal only when they are in exactly the same storage location. This is not a very robust

form of comparison, and is generally meaningful only if an object with a given structure is stored in one unique place, or if we are trying to determine literal identity of objects rather than structural equality. Alternatively, we could compare them more deeply, component-by-component. In this case, there is the issue of how those components are compared, e.g. by reference or more deeply, and so on. It is important to be aware of what equality methods are really doing. The same is true for inequality methods.

7.20 Principle of Interning

For certain cases of read-only objects, such as a set of strings, it is sometimes useful to guarantee that there is *at most one copy* of any object value. Not only does this save space, it allows objects to be compared for equality just by comparing references to them and not delving into their internal structure. This generally improves speed if there are many comparisons to be done. The **principle of interning**, then, is: prior to creating a new (read-only) object, check to see if there is already an object with the same value. If there is, return a reference to the pre-existing object. If not, then create a new object and return a reference to it. The principle of interning is built into languages like Lisp and Prolog: every time a string is read, it is "interned", i.e. the procedure mentioned above is done.

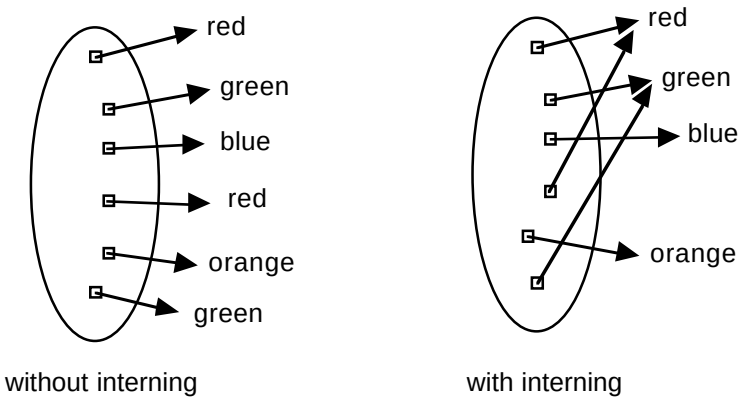

Figure 100: Illustrating use of interning for pointers to read-only strings

A special case of interning can be useful in Java: If we store references to the objects in an array, and refer to the objects by the array index, generally a relatively small index, we can use the **switch** statement to dispatch on the value of an object. Let us call this special case **small-integer interning**.

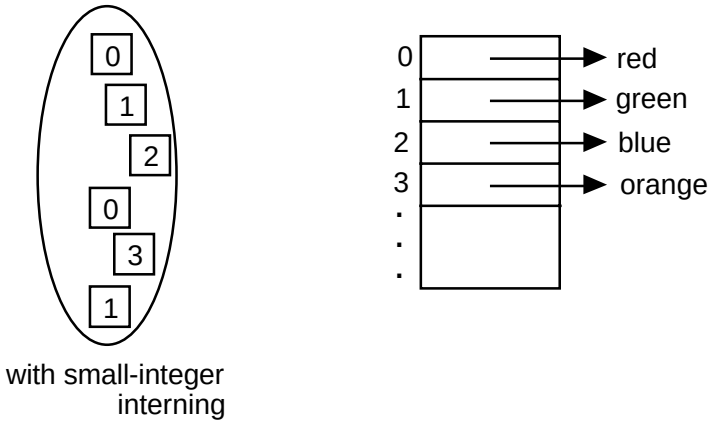

Figure 101: Illustration of small-integer interning

With either ordinary or small-integer interning, a table or list of some kind must be maintained to keep track of the objects that have been allocated. This allows us to search through the set of previously-interned items when a request is made to intern a new item. The use of an array rather than a list for the table, in the case of small-integer interning, allows us to quickly get to the contents of the actual object when necessary.

Exercise

•• 1 Determine whether any standard Java class provides interning. If so, explain how this feature could be used.

7.21 Linked-Lists in Object-Oriented Form

To close this section, we revisit the linked-list implementation discussed in chapter 5. In that implementation, all list manipulation was done with static methods. In our current implementation we will replace many of these with regular methods. For example, if `L` is a list, we will use

```
L.first()
```

to get its first element rather than

```
first(L)
```

One reason this is attractive in Java programming is that to use the static method outside of the list class itself, we would have to qualify the method name with the class name, as in:

```
List.first(L)
```

whereas with the form `L.first()` we would not, since the class is implied from the type of `L` itself.

In presenting our list class, we will use `Object` as the type of a member of list. This is similar in philosophy to some standard Java classes, such as `Vector`. The implication of this choice is that lists may be highly heterogeneous due to the *polymorphic* nature of the Object class. For example, some of the elements of a list may be lists themselves, and some of those lists can have lists as elements, and so on. This gives an easy way to achieve the list functionality of a language such as rex, which was heavily exercised in the early chapters.

Due to the attendant polymorphism of this approach, we call our list class `Polylist`. Another purpose of doing so is that the class `List` is commonly imported into applications employing the Java awt (abstract window toolkit) and we wish to avoid conflict with that name.

We can also introduce input and output methods that are capable of casting Polylists to Strings in the form of S expressions. This is very convenient for building software prototypes where we wish to concentrate on the inner structure rather than the format of data. We can change the input and output syntax to a different form if desired, without disturbing the essence of the application program.

```
public class Polylist
  {
  // nil is the empty-list constant

  public static final Polylist nil = new Polylist();

  private polycell ptr;

  // The constructors are not intended for general use;
  // cons is preferred instead.

  // construct empty Polylist

  Polylist()
    {
    ptr = null;
    }

   // construct non-empty Polylist

  Polylist(Object First, Polylist Rest)
    {
    ptr = new polycell(First, Rest);
    }

   // isEmpty() tells whether the Polylist is empty.

  public boolean isEmpty()
    {
    return ptr == null;
    }
```

```
  // nonEmpty() tells whether the Polylist is non-empty.

  public boolean nonEmpty()
    {
    return ptr != null;
    }

  // first() returns the first element of a non-empty list.

  public Object first()
    {
    return ptr.first();
    }

  // rest() returns the rest of a non-empty Polylist.

  public Polylist rest()
    {
    return ptr.rest();
    }

   // cons returns a new Polylist given a First, with this as a Rest

  public Polylist cons(Object First)
    {
    return new Polylist(First, this);
    }


   // static cons returns a new Polylist given a First and a Rest.

  public static Polylist cons(Object First, Polylist Rest)
    {
    return Rest.cons(First);
    }
  }


public class polycell
  {
  Object First;
  Polylist Rest;

  // first() returns the first element of a NonEmptyList.

  public Object first()
    {
    return First;
    }

  // rest() returns the rest of a NonEmptyList.

  public Polylist rest()
    {
    return Rest;
    }
```

```
// polycell is the constructor for the cell of a Polylist,
// given a First and a Rest.

public polycell(Object First, Polylist Rest)
  {
  this.First = First;
  this.Rest = Rest;
  }
}
```

One possible reason for preferring the static 2-argument form of `cons` is as follows: suppose we construct a list using the 1-argument `cons` method:

```
nil.cons(a).cons(b).cons(c)
```

This doesn't look bad, except that the list constructed has the elements in the reverse order from how they are listed. That is, the first element of this list will be `c`, not `a`.

To give an example of how coding might look using the object-oriented style, we present the familiar append method. Here append produces a new list by following elements of the current list with the elements in the argument list. In effect, the current list is copied, while the argument list is shared.

```
// append(M) returns a Polylist consisting of the elements of this
// followed by those of M.

public Polylist append(Polylist M)
  {
  if( isEmpty() )
    return M;
  else
    return cons(first(), rest().append(M));
  }
```

Exercises

•• 1 Implement a method that creates a range of Integers given the endpoints of the range.

•• 2 Implement a method that returns a list of the elements in an array of objects.

••• 3 Implement a `Stack` class using composition of a `Polylist`.

••• 4 Implement a `Queue` class using linked lists. You probably won't want to use the `Polylist` class directly, since the natural way to implement a queue requires a closed list rather than an open one.

7.22 Enumeration Interfaces

A common technique for iterating over things such as lists is to use the interface `Enumeration` defined in java.util. To qualify as an implementation of an `Enumeration`, a class must provide two methods:

```
public Object nextElement()
public boolean hasMoreElements()
```

The idea is that an `Enumeration` is created from a sequence, such as a list, to contain the elements of the sequence. This is typically done by a method of the underlying sequence class of type

```
public Enumeration elements()
```

that returns the `Enumeration`. The two methods are then used to get one element of a time from the sequence. Note that `nextElement()` returns the next element and has the side-effect of advancing on to the next element after that. If there are no more elements, an exception will be thrown.

Using `Enumeration` interfaces takes some getting used to, but once the idea is understood, they can be quite handy. A typical use of `Enumeration` to sequence through a Polylist would look like:

```
for( Enumeration e = L.elements(); e.hasMoreElements(); )
  {
  Object ob = e.nextElement();
  .... use ob ....
  }
```

This can be contrasted with simply using a Polylist variable, say `T`, to do the sequencing:

```
for( Polylist T = L; T.nonEmpty(); T = T.rest() )
  {
  Object ob = T.first();
  .... use ob ....
  }
```

Note that the `for` statement in the `Enumeration` case has an empty updating step; this is because updating is done as a side effect of the `nextElement` method. A specific example is the following iterative implementation of the `reverse` method, which constructs the reverse of a list. Here `elements()` refers to the elements of *this* list.

```
// reverse(L) returns the reverse of this

public Polylist reverse()
  {
  Polylist rev = nil;
  for( Enumeration e = elements(); e.hasMoreElements(); )
    {
    rev = rev.cons(e.nextElement());
    }
  return rev;
  }
```

Another example is the method `member` that tests whether a list contains the argument:

```
 // member(A) tells whether A is a member of this list

public boolean member(Object A)
  {
  for( Enumeration e = elements(); e.hasMoreElements(); )
    if( A.equals(e.nextElement() )
      return true;
  return false;
  }
```

This form of iteration will not be used for every occasion; for example, recursion is still more natural for methods such as append, which build the result list from the outside-in.

One possible reason to prefer an `Enumeration` is that it is a type, just as a class is a type. Thus a method can be constructed to use an `Enumeration` argument without regard to whether the thing being enumerated is a list, an array, or something else. Thus an `Enumeration` is just an abstraction for a set of items that can be enumerated.

Now we have a look at how the Polylist enumeration is implemented. As with most enumerations, we try not to actually build a new structure to hold the elements, but rather use the elements in place. This entails implementing some kind of *cursor* mechanism to sequence through the list. In the present case, the Polylist class itself serves as the cursor, just by replacing the list with its rest upon advancing the cursor.

The class that implements the enumeration is called PolylistEnum. The method `elements` of class Polylist returns an object of this type, as shown:

```
//  elements() returns a PolylistEnum object, which implements the
//  interface Enumeration.

public PolylistEnum elements()
  {
  return new PolylistEnum(this);
  }
```

The implementation class, `PolylistEnum`, then appears as:

```
public class PolylistEnum implements Enumeration
  {
```

```
Polylist L;               // current list ("cursor")

// construct a PolylistEnum from a Polylist.

public PolylistEnum(Polylist L)
  {
  this.L = L;
  }


// hasMoreElements() indicates whether there are more elements left
// in the enumeration.

public boolean hasMoreElements()
  {
  return L.nonEmpty();
  }


// nextElement returns the next element in the enumeration.

public Object nextElement()
  {
  if( L.isEmpty() )
    throw new NoSuchElementException("No next in Polylist");

  Object result = L.first();
  L = L.rest();
  return result;
  }
}
```

Let's recap how this particular enumeration works:

1. The programmer wants to enumerate the elements of a Polylist for some purpose. She calls the method `elements()` on the list, which returns an `Enumeration` (actually a `PolylistEnum`, but this never needs to be shown in the calling code, since `PolylistEnum` merely implements `Enumeration`.)

2. Method `elements()` calls the constructor of `PolylistEnum`, which initializes `L` of the latter to be the original list.

3. With each call of `nextElement()`, the first of the current list `L` is reserved, then `L` is replaced with its rest. The reserved first element is returned.

4. `nextElement()` can be called repeatedly until `hasMoreElements()`, which tests whether `L` is non-empty, returns false.

Exercises

•• 1 Implement the method `nth` that returns the n*th* element of a list by using an enumeration.

•• 2 Implement a method that returns an array of objects given a list.

•• 3 Locate an implementation of `Enumeration` for the Java class Vector and achieve an understanding of how it works.

••• 4 Implement an `Enumeration` class for an array of `Objects`.

••• 5 Implement an `Enumeration` class that enumerates an array of Objects in reverse order.

7.23 Higher-Order Functions as Objects

The preceding material has shown how we can implement nested lists as in rex. We earlier promised that all of the functional programming techniques that we illustrated could be implemented using Java. The one item unfulfilled in this promise is higher-order functions"higher-order functions" : functions that can take functions as arguments and ones that can return functions as results. We now address this issue.

Java definitely does not allow methods to be passed as arguments. In order to implement the equivalent of higher-order functions, we shall have to use objects as functions, since these *can* be passed as arguments. Here's the trick: the objects we pass or create as functions will have a pre-convened method, say `apply`, that takes an `Object` as an argument and returns an `Object` as a result. We define this class of objects by an interface definition, called `Function1` (for 1-argument function):

```
public interface Function1
  {
  Object apply(Object x);
  }
```

To be used as a function, a class must implement this interface. An example of such a class is one that concatenates the string representation of an object with the String "xxx":

```
Object concatXXX implements Function1
  {
  Object apply(Object arg)
    {
    return "xxx" + arg.toString();
    }

  concatXXX()     // constructor
    {}
  }
```

Note that this particular implementation has a static character, but this will not always be the case, as will be seen shortly. An application of a `concatXXX` method could be shown as:

```
(new concatXXX()) . apply("yyy");    // note: Strings are Objects
```

which would return a String `"xxxyyy"`.

Here's the way `map`, a method which applies a `Function1` to each element of a `Polylist`, would be coded:

```
// map maps an object of class Function1 over a Polylist returning a
// Polylist

Polylist map(Function1 F)
  {
  if( isEmpty() )
    return nil;
  else
    return cons(F.apply(first()), rest().map(F));
  }
```

For example, if the list `L` contained `["foo", "bar", "baz"]` then

```
L.map(new concatXXX)
```

would produce a list containing `["xxxfoo", "xxxbar", "xxxbaz"]`.

More interesting is the case where the object being applied gets some data from "the outside", i.e. through its constructor. Suppose we want a function that returns a function that concatenates a specified prefix, not just "xxx" invariably. Here's how we can modify the class definition `concatXXX` to one, call it `concat`, that does this:

```
Object concat implements Function1
  {
  String prefix;              // prefix to be concatenated

  Object apply(Object arg)
    {
    return prefix + arg.toString();
    }

  concat(String prefix) // constructor
    {
    this.prefix = prefix;
    }
  }
```

Now we can use `map` to create a method that concatenates an argument string to each element of a list:

```
static Polylist concatToAll(String prefix, Polylist L)
  {
  return L.map(new concat(prefix));
  }
```

In rex, the same idea could be shown as:

```
concatToAll(prefix, L) = map((X) => prefix + x, L);
```

We are now just one step away from functions that return functions as results. All such a function needs to do is to call the constructor of a `Function1` object to return a new object that can be applied. For example, the rex definition:

```
f(X) = (Y) => X + Y;
```

represents a function that takes an argument (X) and returns a function. In Java this would be the method:

```
static Object f(Object X)
  {
  return new concat(X.toString());
  }
```

The object-oriented version of higher-order functions might be a little harder to understand than the rex version, which is one reason we wanted to present the rex version first. Underneath the syntax, the implementation using objects is very similar to standard implementations using functions, where the function objects are called *closures* (meaning that they are a closed environment in which otherwise free variables are bound).

Exercises

••• 1 The higher-order function reduce takes a function argument that has two arguments. Define an interface definition `Function2` analogous to `Function1` above, then give the implementation of reduce so as to take a `Function2` as an argument.

•••• 2 Define a method `compose` that takes two `Function1` arguments and returns their composition as a `Function1` argument.

••• 3 Develop a class `FunctionFromArray` that implements a `Function` given an array. The function applied to the integer `i` is to give the ith element of the array.

7.24 Conclusion

This chapter has presented a number of ideas concerning object-oriented programming. Java is an object-oriented language accompanied by a rich library of classes, including an abstract window toolkit for doing graphics, menus, etc. As this book does not attempt to be a tutorial on Java, we recommend other books that can be studied for the finer details. Attempting to program applications in Java is the best way to understand the concepts described here, including inheritance and interfaces.

7.25 Chapter Review

Define the following terms:

abstract base class
abstract data type (ADT)
aggregation
applet
attribute
circular array
class
client
composition
constructor
container class
deep copying
deque
derived class
enumeration (Java-style)
equality
extend (a class)
getter
inheritance
implement (an interface)
interface
interning
is-a
message
method
modularity
normalization (of code)
object
Object class
over-ride
priority-queue
queue
setter
shallow copying
stack
static method
static variable
sub-class
wrapper

7.26 Further Reading

G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug, K. Nygaard, *Simula Begin*, Auerbach, Philadelphia, 1973. [Describes the *Simula* language, generally regarded as the original object-oriented programming language. Moderate.]

Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988. [Moderate.]

David Flanagan, *Java in a Nutshell*, Second Edition, O'Reilly, 1997. [A succinct guide to Java for moderately-experienced programmers. Moderate.]

James Gosling, Frank Yellin, the Java Team, *The Java™ application programming interface*, Volumes 1 and 2, Addison-Wesley, 1996. [A thorough guide to the essential Java classes. Moderate.]

James Rumbaugh, Ivar Jacobson, Grady Booch, The unified modeling language reference manual, Reading, Mass : Addison-Wesley, 1998. [One of many references on UML.]