In this chapter, we explore programming language constructs that support data abstraction. Among the new ideas of the last 50 years in programming methodologies and programming language design, data abstraction is one of the most profound.

We begin by discussing the general concept of abstraction in programming and programming languages. Data abstraction is then defined and illustrated with an example. This topic is followed by descriptions of the support for data abstraction in Ada, C++, Objective-C, Java, C#, and Ruby. To illuminate the similarities and differences in the design of the language facilities that support data abstraction, implementations of the same example data abstraction are given in Ada, C++, Objective-C, Java, and Ruby. Next, the capabilities of Ada, C++, Java 5.0, and C# 2005 to build parameterized abstract data types are discussed.

All the languages used in this chapter to illustrate the concepts and constructs of abstract data types support object-oriented programming. The reason is that virtually all contemporary languages support object-oriented programming and nearly all of those that do not, and yet support abstract data types, have faded into obscurity.

Constructs that support abstract data types are encapsulations of the data and operations on objects of the type. Encapsulations that contain multiple types are required for the construction of larger programs. These encapsulations and the associated namespace issues are also discussed in this chapter.

Some programming languages support logical, as opposed to physical, encapsulations, which are actually used to encapsulate names. These are discussed in Section 11.7.

11.1 The Concept of Abstraction

An **abstraction** is a view or representation of an entity that includes only the most significant attributes. In a general sense, abstraction allows one to collect instances of entities into groups in which their common attributes need not be considered. For example, suppose we define birds to be creatures with the following attributes: two wings, two legs, a tail, and feathers. Then, if we say a crow is a bird, a description of a crow need not include those attributes. The same is true for robins, sparrows, and yellow-bellied sapsuckers. These common attributes in the descriptions of specific species of birds can be abstracted away, because all species have them. Within a particular species, only the attributes that distinguish that species need be considered. For example, crows have the attributes of being black, being of a particular size, and being noisy. A description of a crow needs to provide those attributes, but not the others that are common to all birds. This results in significant simplification of the descriptions of members of the species. A less abstract view of a species, that of a bird, may be considered when it is necessary to see a higher level of detail, rather than just the special attributes.

In the world of programming languages, abstraction is a weapon against the complexity of programming; its purpose is to simplify the programming process. It is an effective weapon because it allows programmers to focus on essential attributes, while ignoring subordinate attributes.

The two fundamental kinds of abstraction in contemporary programming languages are process abstraction and data abstraction.

The concept of **process abstraction** is among the oldest in programming language design (Plankalkül supported process abstraction in the 1940s). All subprograms are process abstractions because they provide a way for a program to specify a process, without providing the details of how it performs its task (at least in the calling program). For example, when a program needs to sort an array of numeric data of some type, it usually uses a subprogram for the sorting process. At the point where the sorting process is required, a statement such as

```
sortInt(list, listLen)
```

is placed in the program. This call is an abstraction of the actual sorting process, whose algorithm is not specified. The call is independent of the algorithm implemented in the called subprogram.

In the case of the subprogram `sortInt`, the only essential attributes are the name of the array to be sorted, the type of its elements, the array's length, and the fact that the call to `sortInt` will result in the array being sorted. The particular algorithm that `sortInt` implements is an attribute that is not essential to the user. The user needs to see only the name and protocol of the sorting subprogram to be able to use it.

The widespread use of data abstraction necessarily followed that of process abstraction because an integral and essential part of every data abstraction is its operations, which are defined as process abstractions.

11.2 Introduction to Data Abstraction

The evolution of data abstraction began in 1960 with the first version of COBOL, which included the record data structure.[1] The C-based languages have structs, which are also records. An abstract data type is a data structure, in the form of a record, but which includes subprograms that manipulate its data.

Syntactically, an abstract data type is an enclosure that includes only the data representation of one specific data type and the subprograms that provide the operations for that type. Through access controls, unnecessary details of the type can be hidden from units outside the enclosure that use the type. Program units that use an abstract data type can declare variables of that type, even though the actual representation is hidden from them. An instance of an abstract data type is called an **object**.

One of the motivations for data abstraction is similar to that of process abstraction. It is a weapon against complexity; a means of making large and/or complicated programs more manageable. Other motivations for and advantages of abstract data types are discussed later in this section.

1. Recall from Chapter 2, that a record is a data structure that stores fields, which have names and can be of different types.

program's modularity and it is a clear separation of design and implementation. If both the declarations and the definitions of types and operations are in the same syntactic unit, there must be some means of hiding from client program units the parts of the unit that specify the definitions.

11.2.3 An Example

A stack is a widely applicable data structure that stores some number of data elements and only allows access to the data element at one of its ends, the top. Suppose an abstract data type is to be constructed for a stack that has the following abstract operations:

create(stack)	Creates and possibly initializes a stack object
destroy(stack)	Deallocates the storage for the stack
empty(stack)	A predicate (or Boolean) function that returns true if the specified stack is empty and false otherwise
push(stack, element)	Pushes the specified element on the specified stack
pop(stack)	Removes the top element from the specified stack
top(stack)	Returns a copy of the top element from the specified stack

Note that some implementations of abstract data types do not require the create and destroy operations. For example, simply defining a variable to be of an abstract data type may implicitly create the underlying data structure and initialize it. The storage for such a variable may be implicitly deallocated at the end of the variable's scope.

A client of the stack type could have a code sequence such as the following:

```
...
create(stk1);
push(stk1, color1);
push(stk1, color2);
temp = top(stk1);
...
```

11.3 Design Issues for Abstract Data Types

A facility for defining abstract data types in a language must provide a syntactic unit that encloses the declaration of the type and the prototypes of the subprograms that implement the operations on objects of the type. It must be possible to make these visible to clients of the abstraction. This allows clients to declare variables of the abstract type and manipulate their values. Although the type

11.4.2.3 Constructors and Destructors

C++ allows the user to include **constructor** functions in class definitions, which are used to initialize the data members of newly created objects. A constructor may also allocate the heap-dynamic data that are referenced by the pointer members of the new object. Constructors are implicitly called when an object of the class type is created. A constructor has the same name as the class whose objects it initializes. Constructors can be overloaded, but of course each constructor of a class must have a unique parameter profile.

A C++ class can also include a function called a **destructor**, which is implicitly called when the lifetime of an instance of the class ends. As stated earlier, stack-dynamic class instances can contain pointer members that reference heap-dynamic data. The destructor function for such an instance can include a **`delete`** operator on the pointer members to deallocate the heap space they reference. Destructors are often used as a debugging aid, in which case they simply display or print the values of some or all of the object's data members before those members are deallocated. The name of a destructor is the class's name, preceded by a tilde (~).

Neither constructors nor destructors have return types, and neither use **`return`** statements. Both constructors and destructors can be explicitly called.

11.4.2.4 An Example

Our examle of a C++ abstract data type is, once again, a stack:

```
#include <iostream.h>
class Stack {
  private:  //** These members are visible only to other
            //** members and friends (see Section 11.6.4)
    int *stackPtr;
    int maxLen;
    int topSub;
  public:   //** These members are visible to clients
    Stack() {   //** A constructor
      stackPtr = new int [100];
      maxLen = 99;
      topSub = -1;
    }
    ~Stack() {delete [] stackPtr;};  //** A destructor
    void push(int number) {
      if (topSub == maxLen)
        cerr << "Error in push--stack is full\n";
      else stackPtr[++topSub] = number;
    }
    void pop() {
      if (empty())
```

```
        cerr << "Error in pop--stack is empty\n";
      else topSub--;
    }
    int top() {
      if (empty())
        cerr << "Error in top--stack is empty\n";
      else
        return (stackPtr[topSub]);
    }
    int empty() {return (topSub == -1);}
}
```

We discuss only a few aspects of this class definition, because it is not necessary to understand all of the details of the code. Objects of the `Stack` class are stack dynamic but include a pointer that references heap-dynamic data. The `Stack` class has three data members—`stackPtr`, `maxLen`, and `topSub`—all of which are private. `stackPtr` is used to reference the heap-dynamic data, which is the array that implements the stack. The class also has four public member functions—`push`, `pop`, `top`, and `empty`—as well as a constructor and a destructor. All of the member function definitions are included in this class, although they could have been externally defined. Because the bodies of the member functions are included, they are all implicitly inlined. The constructor uses the **`new`** operator to allocate an array of 100 **`int`** elements from the heap. It also initializes `maxLen` and `topSub`.

The following is an example program that uses the `Stack` abstract data type:

```
void main() {
  int topOne;
  Stack stk;  //** Create an instance of the Stack class
  stk.push(42);
  stk.push(17);
  topOne = stk.top();
  stk.pop();
  ...
}
```

Following is a definition of the `Stack` class with only prototypes of the member functions. This code is stored in a header file with the `.h` file name extension. The definitions of the member functions follow the class definition. These use the scope resolution operator, `::`, to indicate the class to which they belong. These definitions are stored in a code file with the file name extension `.cpp`.

```
// Stack.h - the header file for the Stack class
#include <iostream.h>
```

```
class Stack {
  private:  //** These members are visible only to other
            //** members and friends (see Section 11.6.4)
    int *stackPtr;
    int maxLen;
    int topSub;
  public:   //** These members are visible to clients
    Stack();  //** A constructor
    ~Stack();  //** A destructor
    void push(int);
    void pop();
    int top();
    int empty();
}

// Stack.cpp - the implementation file for the Stack class
#include <iostream.h>
#include "Stack.h"
using std::cout;
Stack::Stack() {  //** A constructor
  stackPtr = new int [100];
  maxLen = 99;
  topSub = -1;
}

Stack::~Stack() {delete [] stackPtr;};  //** A destructor

void Stack::push(int number) {
  if (topSub == maxLen)
    cerr << "Error in push--stack is full\n";
  else stackPtr[++topSub] = number;
}
void Stack::pop() {
  if (topSub == -1)
    cerr << "Error in pop--stack is empty\n";
  else topSub--;
}
int top() {
      if (topSub == -1)
        cerr << "Error in top--stack is empty\n";
      else
        return (stackPtr[topSub]);
    }
int Stack::empty() {return (topSub == -1);}
```

11.4.2.5 Evaluation

C++ support for abstract data types, through its class construct, is similar in expressive power to that of Ada, through its packages. Both provide effective mechanisms for encapsulation and information hiding of abstract data types. The primary difference is that classes are types, whereas Ada packages are more general encapsulations. Furthermore, the package construct of Ada was designed for more than data abstraction, as discussed in Chapter 12.

11.4.3 Abstract Data Types in Objective-C

As has been previously stated, Objective-C is similar to C++ in that its initial design was the C language with extensions to support object-oriented programming. One of the fundamental differences between the two is that Objective-C uses the syntax of Smalltalk for its method calls.

11.4.3.1 Encapsulation

The interface part of an Objective-C class is defined in a container called an **interface** with the following general syntax:

```
@interface class-name : parent-class {
 instance variable declarations
}
 method prototypes
@end
```

The first and last lines, which begin with at signs (@), are directives.

The implementation of a class is packaged in a container naturally named *implementation*, which has the following syntax:

```
@implementation class-name
 method definitions
@end
```

As in C++, in Objective-C classes are types.

Method prototypes have the following syntax:

```
(+ | -)(return-type) method-name [: (formal-parameters)];
```

When present, the plus sign indicates that the method is a class method; a minus sign indicates an instance method. The brackets around the formal parameters indicate that they are optional. Neither the parentheses nor the colon are present when there are no parameters. As in most other languages that support object-oriented programming, all instances of an Objective-C class share a single copy of its instance methods, but each instance has its own copy of the instance data.

directives in place of language constructs to indicate class interfaces and implementation sections also differs from most other programming languages. One minor deficiency is the lack of a way to restrict access to methods. So, even methods meant only to be used inside a class are accessible to clients. Another minor deficiency is that constructors must be explicitly called, thereby requiring programmers to remember to call them, and also leading to further clutter in the client program.

11.4.4 Abstract Data Types in Java

Java support for abstract data types is similar to that of C++. There are, however, a few important differences. All objects are allocated from the heap and accessed through reference variables. Methods in Java must be defined completely in a class. A method body must appear with its corresponding method header.[4] Therefore, a Java abstract data type is both declared and defined in a single syntactic unit. A Java compiler can inline any method that is not overridden. Definitions are hidden from clients by declaring them to be private.

Rather than having private and public clauses in its class definitions, in Java access modifiers can be attached to method and variable definitions. If an instance variable or method does not have an access modifier, it has package access, which is discussed in Section 11.7.2.

11.4.4.1 An Example

The following is a Java class definition for our stack example:

```
class StackClass {
  private int [] stackRef;
  private int maxLen,
              topIndex;
  public StackClass() {  // A constructor
    stackRef = new int [100];
    maxLen = 99;
    topIndex = -1;
  }
  public void push(int number) {
    if (topIndex == maxLen)
      System.out.println("Error in push—stack is full");
    else stackRef[++topIndex] = number;
  }
  public void pop() {
    if (empty())
      System.out.println("Error in pop—stack is empty");
```

4. Java interfaces are an exception to this—an interface has method headers but cannot include their bodies.

```
    else --topIndex;
  }
  public int top() {
    if (empty()) {
      System.out.println("Error in top—stack is empty");
      return 9999;
    }
    else
      return (stackRef[topIndex]);
  }
  public boolean empty() {return (topIndex == -1);}
}
```

An example class that uses `StackClass` follows:

```
public class TstStack {
  public static void main(String[] args) {
    StackClass myStack = new StackClass();
    myStack.push(42);
    myStack.push(29);
    System.out.println("29 is: " + myStack.top());
    myStack.pop();
    System.out.println("42 is: " + myStack.top());
    myStack.pop();
    myStack.pop();  // Produces an error message
  }
}
```

One obvious difference is the lack of a destructor in the Java version, obviated by Java's implicit garbage collection.[5]

11.4.4.2 Evaluation

Although different in some primarily cosmetic ways, Java's support for abstract data types is similar to that of C++. Java clearly provides for what is necessary to design abstract data types.

11.4.5 Abstract Data Types in C#

Recall that C# is based on both C++ and Java and that it also includes some new constructs. Like Java, all C# class instances are heap dynamic. Default constructors, which provide initial values for instance data, are predefined for all classes. These constructors provide typical initial values, such as `0` for **`int`** types and **`false`** for **`boolean`** types. A user can furnish one or more constructors for any

5. In Java, the **`finalize`** method serves as a kind of destructor.

11.5.2 C++

C++ also supports parameterized abstract data types. To make the example C++ stack class of Section 11.4.2.4 generic in the stack size, only the constructor function needs to be changed, as in the following:

```
Stack(int size) {
  stackPtr = new int [size];
  maxLen = size - 1;
  topSub = -1;
}
```

The declaration for a stack object now may appear as follows:

```
Stack stk(150);
```

The class definition for `Stack` can include both constructors, so users can use the default-size stack or specify some other size.

The element type of the stack can be made generic by making the class a templated class. Then, the element type can be a template parameter. The definition of the templated class for a stack type is as follows:

```
#include <iostream.h>
template <typename Type>  // Type is the template parameter
class Stack {
  private:
    Type *stackPtr;
    int maxLen;
    int topSub;
  public:
// A constructor for 100 element stacks
    Stack() {
      stackPtr = new Type [100];
      maxLen = 99;
      topSub = -1;
    }
// A constructor for a given number of elements
    Stack(int size) {
      stackPtr = new Type [size];
      maxLen = size - 1;
      topSub = -1;
    }
    ~Stack() {delete stackPtr;};  // A destructor
    void push(Type number) {
      if (topSub == maxLen)
        cout << "Error in push—stack is full\n";
      else stackPtr[++ topSub] = number;
```

```
    }
    void pop() {
      if (empty())
        cout << "Error in pop—stack is empty\n";
      else topSub --;
    }
    Type top() {
      if (empty())
        cerr << "Error in top--stack is empty\n";
      else
        return (stackPtr[topSub]);
    }
    int empty() {return (topSub == -1);}
}
```

As in Ada, C++ templated classes are instantiated to become typed classes at compile time. For example, an instance of the templated `Stack` class, as well as an instance of the typed class, can be created with the following declaration:

```
Stack<int> myIntStack;
```

However, if an instance of the templated `Stack` class has already been created for the **`int`** type, the typed class need not be created.

11.5.3 Java 5.0

Java 5.0 supports a form of parameterized abstract data types in which the generic parameters must be classes. Recall that these were briefly discussed in Chapter 9.

The most common generic types are collection types, such as `LinkedList` and `ArrayList`, which were in the Java class library before support for generics was added. The original collection types stored `Object` class instances, so they could store any objects (but not primitive types). Therefore, the collection types have always been able to store multiple types (as long as they are classes). There were three issues with this: First, every time an object was removed from the collection, it had to be cast to the appropriate type. Second, there was no error checking when elements were added to the collection. This meant that once the collection was created, objects of any class could be added to the collection, even if the collection was meant to store only `Integer` objects. Third, the collection types could not store primitive types. So, to store **`int`** values in an `ArrayList`, the value first had to be put in an `Integer` class instance. For example, consider the following code:

```
//* Create an ArrayList object
ArrayList myArray = new ArrayList();
//* Create an element
myArray.add(0, new Integer(47));
```

```
//* Get first object
Integer myInt = (Integer)myArray.get(0);
```

In Java 5.0, the collection classes, the most commonly used of which is `ArrayList`, became a generic class. Such classes are instantiated by calling **new** on the class constructor and passing it the generic parameter in pointed brackets. For example, the `ArrayList` class can be instantiated to store `Integer` objects with the following statement:

```
ArrayList <Integer> myArray = new ArrayList <Integer>();
```

This new class overcomes two of the problems with pre-Java 5.0 collections. Only `Integer` objects can be put into the `myArray` collection. Furthermore, there is no need to cast an object being removed from the collection.

Java 5.0 also includes interfaces for collections for lists, queues, and sets.

Users also can define generic classes in Java 5.0. For example, we could have the following:

```
public class MyClass<T> {
  ...
}
```

This class could be instantiated with the following:

```
MyClass<String> myString;
```

There are some drawbacks to these user-defined generic classes. For one thing, they cannot store primitives. Second, the elements cannot be indexed. Elements must be added to user-defined generic collections with the `add` method. Next, we implement the generic stack example using an `ArrayList`. Note that the last element of an `ArrayList` is found using the `size` method, which returns the number of elements in the structure. Elements are deleted from the structure with the `remove` method. Following is the generic class:

```
import java.util.*;
public class Stack2<T> {

  private ArrayList<T> stackRef;
  private int maxLen;
  public Stack2() {  // A constructor
    stackRef = new ArrayList<T> ();
    maxLen = 99;
  }
  public void push(T newValue) {
    if (stackRef.size() == maxLen)
```

```
      System.out.println("Error in push—stack is full");
    else
      stackRef.add(newValue);
  }
  public void pop() {
    if (empty())
      System.out.println("Error in pop—stack is empty");
    else
      stackRef.remove(stackRef.size() - 1);
  }
  public T top() {
    if (empty()) {
      System.out.println("Error in top—stack is empty");
      return null;
    }
    else
      return (stackRef.get(stackRef.size() - 1));
  }
  public boolean empty() {return (stackRef.isEmpty());}
```

This class could be instantiated for the `String` type with the following:

```
Stack2<String> myStack = new Stack2<String>();
```

Recall from Chapter 9, that Java 5.0 supports wildcard classes. For example, `Collection<?>` is a wildcard class for all collection classes. This allows a method to be written that can accept any collection type as a parameter. Because a collection can itself be generic, the `Collection<?>` class is in a sense a generic of a generic class.

Some care must be taken with objects of the wildcard type. For example, because the components of a particular object of this type have a type, other type objects cannot be added to the collection. For example, consider

```
Collection<?> c = new ArrayList<String>();
```

It would be illegal to use the `add` method to put something into this collection unless its type were `String`.

A generic class can easily be defined in Java 5.0 that will work only for a restricted set of types. For example, a class can declare a variable of the generic type and call a method such as `compareTo` through that variable. If the class is instantiated for a type that does not include a `compareTo` method, the class cannot be used. To prevent a generic class from being instantiated for a type that does not support `compareTo`, it could be defined with the following generic parameter:

```
<T extends Comparable>
```

`Comparable` is the interface in which `compareTo` is declared. If this generic type is used on a class definition, the class cannot be instantiated for any type that does not implement `Comparable`. The choice of the reserved word **`extends`** seems odd here, but its use is related to the concept of a subtype. Apparently, the designers of Java did not want to add another more connotative reserved word to the language.

11.5.4 C# 2005

As was the case with Java, the first version of C# defined collection classes that stored objects of any class. These were `ArrayList`, `Stack`, and `Queue`. These classes had the same problems as the collection classes of pre-Java 5.0.

Generic classes were added to C# in its 2005 version. The five predefined generic collections are `Array`, `List`, `Stack`, `Queue`, and `Dictionary` (the `Dictionary` class implements hashes). Exactly as in Java 5.0, these classes eliminate the problems of allowing mixed types in collections and requiring casts when objects are removed from the collections.

As with Java 5.0, users can define generic classes in C# 2005. One capability of the user-defined C# generic collections is that any of them can be defined to allow its elements to be indexed (accessed through subscripting). Although the indexes are usually integers, an alternative is to use strings as indexes.

One capability that Java 5.0 provides that C# 2005 does not is wildcard classes.

11.6 Encapsulation Constructs

The first five sections of this chapter discuss abstract data types, which are minimal encapsulations.[6] This section describes the multiple-type encapsulations that are needed for larger programs.

11.6.1 Introduction

When the size of a program reaches beyond a few thousand lines, two practical problems become evident. From the programmer's point of view, having such a program appear as a single collection of subprograms or abstract data type definitions does not impose an adequate level of organization on the program to keep it intellectually manageable. The second practical problem for larger programs is recompilation. For relatively small programs, recompiling the whole program after each modification is not costly. But for large programs, the cost of recompilation is significant. So, there is an obvious need to find ways to avoid recompilation of the parts of a program that are not affected by

6. In the case of Ada, the package encapsulation can be used for single types and also for multiple types.