Chapter 5 - Data Aggregates

In addition to the fundamental data types introduced in section 4.3, programming languages have facilities for types that are made up of aggregates of other types called data aggregates. This chapter examines these aggregate types, specifically, the aggregates array, string, record, and stream.

Each of these aggregate types will be examined from the following points of view:

1. Declaration and binding: In addition to the data object bindings that we have seen previously, the bindings of interest in this chapter include the binding of the aggregate type to its constituent types.

2. Manipulation: The fundamental operators on aggregate types are comparison and assignment. In addition, aggregate types need operators known as selectors, which convert from aggregate to constituent values, and constructor operators, which convert from constituent to aggregate values.

3. Implementation: The implementation of an aggregate type refers to special considerations given to the representation of aggregate structures in storage. Data compression, data organization, and indirect storage using pointers are important parts of implementation.

Data aggregate types provided by a programming language are distinct from abstract data types constructed by the programmer from the simple and aggregate types of the language. We address language features related to data abstraction in Chapter 8.

5.1 Data aggregate models

First, we introduce three abstract models for representing data aggregation. These models will serve as useful tools in describing specific data aggregates. This classification of models is taken from Hoare (1972).

For the purpose of this section we use the notation that $T_1, T_2, \ldots, T_n$ to represent types, either simple or aggregate. These types are not necessarily distinct, so that $T_1$ and $T_2$ may represent the same type or different types. When an aggregate type $T$ is defined in terms of other types, these other types are called constituent types. The values of an aggregate type are structures built from values of the constituent types.

We are also interested in the cardinality of a type. Simply stated, the cardinality of type $T$ is the number of possible values of that type. Data types may have either a finite cardinality or a denumerably infinite cardinality. The cardinality of an aggregate type is easily computed from the cardinality of its constituent types. We represent the cardinality of a type $T$ by $C(T)$.

5.1.1 Cartesian product

The Cartesian product type is constructed from a finite set of types and is defined as follows:

$$T_1 \times T_2 \times \ldots \times T_n = \{(t_1, t_2, \ldots, t_n) \mid t_1 \in T_1, t_2 \in T_2, \ldots, t_n \in T_n\}$$
In words, the Cartesian product is the set of all possible tuples that can be formed by choosing one element from each of the \( n \) types participating in the product. A tuple is an ordered collection.

Consider the following example with \( n=3 \) and finite types \( T_1, T_2, \) and \( T_3 \).

\[
T_1 = \{1,2,3,4\} \\
T_2 = \{\text{"A"},\text{"B"},\text{"C"}\} \\
T_3 = \{\text{true}, \text{false}\}
\]

By our definition, the possible elements of type \( T_1 \times T_2 \times T_3 \) are

\[
(1,\text{"A"},\text{true}) \quad (2,\text{"A"},\text{true}) \quad (3,\text{"A"},\text{true}) \quad (4,\text{"A"},\text{true}) \\
(1,\text{"A"},\text{false}) \quad (2,\text{"A"},\text{false}) \quad (3,\text{"A"},\text{false}) \quad (4,\text{"A"},\text{false}) \\
(1,\text{"B"},\text{true}) \quad (2,\text{"B"},\text{true}) \quad (3,\text{"B"},\text{true}) \quad (4,\text{"B"},\text{true}) \\
(1,\text{"B"},\text{false}) \quad (2,\text{"B"},\text{false}) \quad (3,\text{"B"},\text{false}) \quad (4,\text{"B"},\text{false}) \\
(1,\text{"C"},\text{true}) \quad (2,\text{"C"},\text{true}) \quad (3,\text{"C"},\text{true}) \quad (4,\text{"C"},\text{true}) \\
(1,\text{"C"},\text{false}) \quad (2,\text{"C"},\text{false}) \quad (3,\text{"C"},\text{false}) \quad (4,\text{"C"},\text{false})
\]

In general, the cardinality of a Cartesian product can be calculated by

\[
C(T_1 \times \ldots \times T_n) = C(T_1) \times \ldots \times C(T_n)
\]

It is not necessary for types participating in a Cartesian product to be either finite or distinct. For example, \( T_1 \) and \( T_2 \) could each be of type integer, in which case \( T_1 \times T_2 \) would be the set of all integer pairs.

The Cartesian product is the model that represents record structures in programming languages.

Reinforce: Suppose type \( T_1 = \{1,5,9\} \) and \( T_2 = \{a,b\} \).

a. What is type \( T_1 \times T_2 \)?

b. What is type \( T_2 \times T_1 \)?

5.1.2 Mapping

A mapping in its simplest form maps one type (the domain type) into another type (the range type). This mapping is defined so that each element of the domain type is mapped into one and only one element of the range type. A given element of the range type may be associated with several domain elements, or it may be associated with none through a mapping. Such a mapping can be represented by

\[
M: T_1 \rightarrow T_2
\]
where $T_1$ is the domain type and $T_2$ is the range type. Figure 5.1 illustrates this form of mapping. The mapping is finite in the sense that the domain type must consist of a finite number of elements.

**Figure 5.1 Mapping from $T_1$ to $T_2$**

A common representation of such a mapping that corresponds to the mathematical object known as function is a set of pairs of the form

$$\{(d_1, r_1), (d_2, r_2), \ldots, (d_n, r_n)\}$$

where $\{d_i, (i=1, \ldots, n)\}$ is the set of all distinct elements of the domain type and $\{r_i, (i=1, \ldots, n)\}$ is a subset of the set of all elements of the range set and where all $r_i$ are not necessarily distinct. The cardinality of a mapping is given by

$$C(M:D \rightarrow R) = C(R)^{C(D)}$$

This relation is the standard arithmetic representation for raising a number to a power. Therefore, if the domain contains 5 elements and the range contains 3 elements, the mapping has cardinality of $3^5 = 243$.

This model is commonly used to represent arrays, and in some languages, it is also used for strings.

One special case of a mapping type occurs when the domain type is itself a Cartesian product type. Since the domain type must always be finite, all of the constituent types making up the domain of the Cartesian product must also be finite. This form of the mapping type represents multi-indexed arrays, and the application of the two preceding cardinality formulas yields

$$C(M:T_1 \times T_2 \times \ldots \times T_n \rightarrow T_{n+1}) = C(T_{n+1})^{C(T_1) \cdot C(T_2) \cdot \ldots \cdot C(T_n)}$$
Reinforce: Suppose type \( T_1 = \{'a', 'test', 'to', 'see', 'if', 'you', 'know', 'how'\} \) and \( T_2 = \{0, 1, 2, 3, 4, 5\} \). Define a mapping from \( T_1 \) to \( T_2 \) which maps a string into the number of characters it contains.

### 5.1.3 Sequence

Unlike Cartesian product and mapping types, both of which are constructed from two or more other types, a **sequence** type is built from a single constituent type. The sequence type \( T^* \) constructed from constituent type \( T \) is defined as

\[
T^* = \{(t_1, t_2, \ldots, t_n) \text{ where } t_1, t_2, \ldots, t_n \in T \text{ and } n \geq 0\}
\]

In words, the space of the sequence type over constituent type \( T \) is the set of all sequences of elements from the space of \( T \). The cardinality of the sequence type will always be infinite, even when the constituent type has finite cardinality, because there is no limit on the sequence length permitted.

Sequences are most often used as a model for strings in programming languages, where the constituent type \( T \) is a character type.

**Reinforce: Suppose** \( T_1 = \{1, 2\} \) and \( T_2 = \{\text{true, false}\} \).

a. Define \( T_1^* \times T_2^* \)

b. Define \( (T_1 \times T_2)^* \)

### 5.2 Arrays

Arrays are found in almost every programming language and represent the oldest aggregate data structure. The **array** is based on the mapping model where the domain type is the set of indices and the range is what is commonly called the type of the array. We will use the terminology **domain type** and **range type** in our discussion.

#### 5.2.1 Declaration and binding

There are two types that must be bound to the array type, the domain and range. Consider, for example, the following Pascal array type declaration.

```pascal
type A = array [1..10] of integer;
```

This general form permits the binding of both the domain (in this case, \( 1 \ldots 10 \)) and the range (in this case \( \text{integer} \)) to the array type \( A \).

For implementation purposes, the domain type must be of finite cardinality. Some languages are quite restrictive in the permissible types for the domain, frequently limiting them to
finite subranges of the integers starting at a given value such as zero or one. Other languages are much more flexible, permitting any finite range of an integer, character, or enumerated type.

Similarly, some languages restrict the range type to be a simple or scalar type, but other languages permit more generality in the range type, so that any type, simple or aggregate, may be used. In object-oriented languages, classes are also suitable for range types. Since arrays themselves are thus included as possible range types, this provides one alternative for representing arrays with multiple indices. For example, Pascal permits the following declarations

```pascal
type ROW = array [1..10] of integer;
TWOD = array [1..5] of ROW;
```

The aggregate type TWOD consists of 5 elements of type ROW, each of which contains 10 integers. In this case, the domain type of TWOD is the integer subrange 1..5, and the range type is the array type ROW. An alternative model for multi-indexed arrays is considered later.

Pascal and many other imperative languages bind the domain type to the array aggregate at compile time. This restriction prevents the programmer from defining general procedures having array parameters of general domain since the array type is bound to domain type at compile time. For example, consider the following Pascal declarations:

```pascal
type ATYPE1 = array [1..10] of integer;
ATYPE2 = array [11..20] of integer;
var A1:ATYPE1;
A2:ATYPE2;
```

If one wishes to find the mean of both A1 and A2, separate procedures are needed, because the formal parameters are bound to a type at compile time, and that type must be fixed as either ATYPE1 or ATYPE2.

Ada addresses this situation by requiring the binding of the domain to only the base type at compile time while permitting the binding to a subrange of that base type at run time. This is called an unconstrained array definition.

To illustrate, consider the following Ada type declaration:

```ada
type VECTOR is array (INTEGER range <>) of FLOAT;
```

This declares VECTOR to be of type array with an unspecified integer subrange for its domain and having range type FLOAT. In effect, this makes the subrange of the domain a parameter of type VECTOR rather than a part of its definition. The domain can then be bound at run time by the variable declaration

```ada
V : VECTOR (LO..HI);
```

where LO and HI are bound at run time to their values in the containing block. A further use of the unconstrained array is in the passing of parameters. The unconstrained type is all that needs to
be specified for the formal parameter as it takes on the domain subrange of the actual parameter when the procedure is called at run time. This will be discussed further in Chapter 7.

Java also permits run-time binding of an array to its domain type, although it is more restrictive than Ada in what it allows. In Java, the domain type is always a subset of consecutive non-negative integers beginning at zero. The format for declaring such a type is

\[ \text{int[]} \ a; \]

This declares that \( a \) is a variable whose type is a reference to an array of \text{ints}, but does not define the domain. The domain is defined by specifying the size of the array when binding the name \( a \) to an actual array in storage.

\[ a = \text{new int[20]}; \]

Here the domain of \( a \) is bound to \( 0..19 \) as \( a \) is bound to the actual location of these 20 \text{ints}. This binding occurs at run time so statements like

\[ a = \text{new int[n]}; \]

are permissible. We examine how this works in the array implementation section (5.2.3).

"Discuss: Discuss how the various forms of binding for array domain types might be used. How might run-time binding be more effectively used than compile-time binding?"

"Discuss: Why do you think the designers of C++ and Java forced all array domain types to be integers that begin at zero?"

5.2.2 Manipulation

Several classes of operations on data aggregates are frequently provided by programming languages. We examine some of the more common ones here as they apply to arrays.

1. Selection - Indexing is the ubiquitous selection operation for arrays. It generates a range reference from the array identifier and a domain value. Two common delimiters used for expressing selection are parentheses and brackets. For example, the general form of selection in Java is

\[ <\text{array-identifier}> \ [ <\text{domain-expression}> ] \]

whereas Ada uses the alternative

\[ <\text{array-identifier}> \ ( <\text{domain-expression}> ) \]

Another form of selection operator, known as the slice, selects a subarray of its array operand. It accepts as operands the array and a subrange of the domain space of the array and returns
the corresponding subset of the range. Ada contains this operation and a slice is represented as an indexed reference with the index expression replaced by a subrange specification. For example, a slice in Ada could be represented by

\[ V(3..N) \]

The result of a slice operation is itself an array that can be used anywhere an array is appropriate, including array assignment as in

\[ W(1..N-2) := V(3..N); \]

2. Construction - A construction operator in its simplest form accepts values from the range space as operands and constructs an array. A general form of such construction in Java is

\{<range-expression> {, <range-expression> } \}

In this case, the first expression is assigned to the array element corresponding to the first domain value (zero), the second to the second, and so on. For example, a boolean array of five elements could be represented by the array literal

\{true, false, true, true, false\}

An extension to this positional association is to specify both the domain and the range values when building an array. The specification of subranges and alternatives are possible in Ada. In addition, a default value can be assigned to all array elements otherwise undefined by the use of the keyword `other`. The following example illustrates all of these possibilities.

\( (1=>7.0, 5..12=>1.0, 3|15..17=>2.0, others=>0.0) \)

The preceding example also illustrates a further distinction between Java and Ada. Java constructs the entire array with its construction, because there is no possibility of omitting elements. Ada permits a partial construction with the remainder of the range taking on default values.

3. Assignment - When an assignment operator is defined for arrays, it is usually defined as an extension of the assignment operator for simple types. In general, this is of the form

\[ <array-identifier> = <array-expression>; \]

where some languages use `:=` instead of `=` as the assignment operator.

The permissible expressions for the right side of the assignment depend on the allowable array operations within the language. At minimum, a compatible array variable or a construction may be used.

The meaning of array assignment differs greatly in different languages. We will use Ada
and Java as representatives of this difference. Consider the following code in Ada:

```ada
  type ATYPE is array[0..3] of integer;
  A : ATYPE = (0,1,2,3);
  B : ATYPE;
  B := A;
```

Here A and B are two separate arrays, each with its own storage. The assignment statement directs the four values of A to be copied into the four locations of B.

On the other hand, similar code in Java would be:

```java
  int[] A = {0,1,2,3};
  int[] B = new int[4];
  B = A;
```

In this case, the Java assignment statement assigns the actual array referenced by A to B. This is a result of the fact that the implementation of the array in Java is a reference as we will see shortly. In the case of Java, after the preceding statements, the statement

```
  B[0] = 10;
```

results in A[0] becoming 10 as well, because A and B share the same storage. In the Ada example, A and B have no such connection.

One issue here is the requirement for assignment compatibility. Although the requirement for identical ranges is obviously necessary, more flexibility can be provided for the domain in a language like Ada. A language might require identical domains or, more flexibly, domains with the same cardinality for both operands of the assignment. This would determine, for example, whether A:=B is legal when the domain of A is 1..10 and the domain of B is 11..20.

4. Composite Operators - **Composite operators** are the extensions of operators defined on the range space to operators on arrays. The operands must all be of the same cardinality for composite operators to be valid. The resulting array consists of components obtained from a component-by-component application of the operator on the range space.

For example, if

```
  A=(1,2,3,4) and B=(5,9,7,1)
```

then

```
  A + B = (6,11,10,5)
```

When they are present in a language, the composite operators applicable to an array depend on the type of the range. Numeric operators may apply if the range is numeric, logical
operators if the range is boolean, and so on. Most languages have no operators of this type, although the language APL has an extensive set of such operators in addition to many composite operators.

5. Aggregate Operators - Aggregate operators operate on elements of the array type as a whole. They are more complex than the element-by-element application of a composite operator. An example of such an aggregate operator is the unary plus, defined as summing the components of an array. For example,

\[+(1,4,12) = 17\]

As with composite operators, aggregate operators are rarely found in modern programming languages, though APL features them prominently.

6. Attributes - Attributes of an array are values that are derived from the array's structure. Four attributes provided by Ada are \texttt{FIRST}, \texttt{LAST}, \texttt{RANGE} and \texttt{LENGTH}, which return the first element of the domain space, the last element of the domain space, the range of values in the domain space, and the cardinality of the domain space. Ada specifies the attribute of an element by

\texttt{<identifier> ' <attribute-name>}

For example, if array \texttt{A} is declared by

\begin{verbatim}
type ATYPE is array (integer range <> ) of character; 
A : ATYPE (4..12);
\end{verbatim}

then \texttt{A'FIRST} is 4, \texttt{A'LAST} is 12, \texttt{A'RANGE} is 4..12, and \texttt{A'LENGTH} is 9.

In Java, an array's length is available as an attribute by referencing its instance variable length. This means that for any Java array \texttt{a}, its length is referenced by \texttt{a.length}.

7. Comparison - The comparison operators for equality and inequality can be defined in a natural way on two operand arrays with domains of identical cardinality. Ordered comparisons of arrays are also provided by some languages if the arrays have compatible range types. In the case of Ada, ordering comparisons represent comparison in lexicographic order.

Lexicographic order is an extension of the standard alphabetical order for character arrays. We define "less than" in lexicographic order as follows:

\[A \prec^*_B \text{ means}
  \begin{align*}
  &\text{if } B \text{ is null then false} \\
  &\text{else if } A \text{ is null then true} \\
  &\text{else if } A(A'FIRST)=B(B'FIRST) \text{ then} \\
  &\quad A(A'FIRST+1..A'LAST) \prec^*_B (B'BFIRST+1..B'LAST) \\
  &\text{else } A(A'FIRST) < B'(B'FIRST)
  \end{align*}\]
In the above recursive definition, the attributes A'FIRST and A'LAST refer to the first and last elements in the domain space of A. The symbol \( <_A \) is used to distinguish the less than operator for arrays from the less than operator defined on the range space which is denoted by \(<\).

**Discuss:** Why might the use of parentheses as index delimiters in a language be confusing?

**Discuss:** How might you define \( A < B \), where \( A \) and \( B \) are arrays, as a composite operator?

**Laboratory:** With a language assigned to you by your instructor, answer the following questions by writing sample programs and observing the results:

a. When is the index of an array checked against its domain type?
b. When is the domain of a array bound to the array?
c. What types are permitted for the domain of an array?
d. What types are permitted for the range of an array?
e. Is array assignment permitted? How are arrays with different domains and the same cardinality handled?
f. Is ordering defined on arrays? Is so, how?

### 5.2.3 Multi-indexed arrays

As mentioned earlier, it is possible to define a **multi-indexed array** as an array whose range type is itself an array. The notation for expressing such arrays can become burdensome, however. Hence, many programming languages provide a separate facility for defining this construct, a facility based on a different model.

We will introduce this new model through the use of an example, which we can compare to the earlier model. Recall the example given by

```pascal
type ROW = array [1..10] of integer;
TWOD = array [1..5] of ROW;
```

This case results in the following two array structures:

<table>
<thead>
<tr>
<th>type</th>
<th>domain</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROW</td>
<td>1..10</td>
<td>integer</td>
</tr>
<tr>
<td>TWOD</td>
<td>1..5</td>
<td>ROW</td>
</tr>
</tbody>
</table>

Selection reference to an element \( A \) of type \( \text{TWOD} \) would be of the form

\[
A[\text{ROWNUM}][\text{COLNUM}]
\]

where \( \text{ROWNUM} \) is in \( 1..5 \) and \( \text{COLNUM} \) is in \( 1..10 \).

The other view of this structure, using Pascal as our example language, is by type declara-
type TWOD2 = array [1..5,1..10] of integer;

In this case, a single array is defined with the following structure:

\[
\begin{array}{ll}
\text{type} & \text{domain} \\
\text{TWOD2} & (1..5) \times (1..10) \\
\end{array}
\]

Here, the domain of the array is the Cartesian product of the two subranges 1..5 and 1..10 and selection reference to A of type TWOD2 would be of the form

\[ A[\text{ROWNUM, COLNUM}] \]

This latter view is preferred because of its more convenient selection notation. The extension of this representation to \( n > 2 \) indices is a straightforward extension to a Cartesian product of \( n \) finite subrange spaces to form the array domain.

### 5.2.4 Implementation

We represent the implementation of aggregate data structures in two parts, a descriptor and the data itself. The descriptor contains all the necessary information about the structure and a reference to the data. We describe each type's implementation by giving the fields of the descriptor, the organization of the data storage, and algorithms for common operations such as selection.

The descriptor may be needed only at compile time if all components of the descriptor remain unchanged throughout the execution of the program. Frequently, a run-time descriptor is needed to permit more dynamic structures.

Figure 5.2 shows the descriptor for an array. With this descriptor, the selection of a location that contains the range value for a given domain value \( \text{DV} \) is calculated by

\[ \text{Range\_Location} + (\text{DV} - \text{Domain\_First}) \times \text{Range\_Size} \]

\( \text{Range\_Location} \) is the location of the beginning of the array storage, \( \text{Domain\_First} \) is the first value in the domain space, and \( \text{Range\_Size} \) is the number of addressable locations occupied by each element of the array. This calculation assumes that either the domain is a subrange of the integers or \((\text{DV} - \text{Domain\_First})\) is defined as the position number of \( \text{DV} \) in the Domain space minus the position of the element \( \text{Domain\_First} \). It also assumes that the array elements are stored in contiguous addressable locations, a commonly valid assumption for arrays.

**Figure 5.2 Array implementation**
<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Array Name</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Domain Type</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Domain First</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Domain Last</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Range Type</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Range Size</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Range Location</strong></td>
<td>•</td>
</tr>
</tbody>
</table>
This idea can be extended to doubly indexed arrays, as shown in Figure 5.3. The standard formula for calculating the location of the array element with indices $DV_1$ and $DV_2$ is

$$Range\_Location + (DV_1 - Domain_1\_First) * \frac{(Domain_2\_Last - Domain_2\_First + 1)*Range\_Size}{(Domain_2\_Last - Domain_2\_First + 1)} + (DV_2 - Domain_2\_First) * Range\_Size$$

**Figure 5.3 Multi-indexed array implementation**

Here the first term added to $Range\_Location$ represents the number of locations skipped to pass over the first $DV_1 - Domain_1\_First$ rows while the second term is the number of locations to skip over the first $DV_2 - Domain_2\_First$ columns in row $DV_1$. Here we assume that the array is stored by row, meaning that the first row is stored first, then the second, then the third, and so on. In this case the second index will vary the most rapidly as we progress through the memory locations of the data. This form of array storage is called **row major order**. In the case of $A[1..5, 4..7]$, the order of storage would be
The alternative order of storage is **column major order**, where the first index varies most rapidly. Although row major order is the most common approach, FORTRAN is one well-known language that uses column major order. Note that column major order requires an adjustment in the formula for calculating an array element location (see Exercise 5).

Another factor in the implementation of arrays is packing. The **packing** of array data refers to compressing each data element into storage space that is smaller than the computer's addressable unit. Representing Boolean data with 1 bit or character data with 1 byte are examples of packing. If a computer has a 32-bit word, a Boolean array might fit into 1/32 of the storage space that would be required if it were stored in an unpacked form.

The decision of whether or not to pack an array is based on a trade-off between saving storage space and saving time, because accessing packed data requires more complicated algorithms. This decision is usually made by the compiler, although some languages, such as Pascal, permit the programmer to specify when arrays are to be packed.

**Discuss:** What are the relative advantages and disadvantages in expressing a multi-indexed array as an array whose range is another array?

**Discuss:** Some languages permit non-homogeneous arrays, that is, arrays whose components are not necessarily of the same type. What properties should a language have for such a structure to be compatible with the language, and how might such a structure be implemented?

**Discuss:** Row major ordering and column major ordering of array storage really has no effect on the amount of storage used, but affects array access. Discuss if a compiler writer might favor one over the other.

**Reinforce:** Extend the formula for computing the location of an element of a doubly indexed array to an array with N indices.

**Reinforce:** How would the formula for computing the location of an element of a doubly indexed array be changed if the array were stored in column major order?

**Reinforce:** Assume doubly indexed array A has domain (3..12) x (5..9) and each range element occupies two addressable storage cells. What is the address of each of the following if the array is stored beginning at location 1000 in row major order?

- A[3,5]
- A[12,9]
- A[7,5]
- A[8,7]
5.2.5 Arrays in Java

Java restricts the domain space to be int with lowest index value zero. The declaration of an array variable takes the form

```
<array_variable_declaration>::=  
  <component_type>[<identifier> [= <array_definition>]];
```

This creates a variable that will represent an unspecified array of type `<component_type>`. An array variable in Java is a reference variable, where the element it references is an array. If no `array_definition` is given, the array variable is initialized as the null reference.

Java provides two forms for `<array_definition>`. The first lists the component values in the array. Its form is

```
<array_definition> ::= {<expression>}{,<expression>}
```

Here the `<expression>` must be assignment compatible with the array’s component type. As an example, the last line of

```
short s = 5;
int i = 10;
long l = 20;
int[] ia = {s,i,(int)l}
```

is a legal array definition, because a short and a long cast to an int are both assignment compatible with int, the component type of the array. If the cast were absent, the array definition would be in error since a long cannot be assigned to an int. The number of components in the array created is equal to the number of expressions within the curly braces.

The second form of an array definition is the array instance creation expression. This takes the form

```
<array_definition> ::= new <component_type> [<int_expression>]
```

Here the int_expression must be assignment compatible with int and specify the number of components in the array.

To illustrate the preceding definitions, consider the following array definitions in Java:

```
double[] a1 = {1.5, 2.6, 7};
char[] a2 = new char[10];
int[] a3;
```

After these array definitions, `a1` will be a reference to a size 3 array of doubles, whose components are 1.5, 2.6, and 7.0. Note that 7 will be converted to double because it is assign-
ment compatible with double. The variable a2 will be a reference to a char array of size 10, where the values of the ten components will be the default char value (\u0000). The variable a3 will be a reference to an int array whose present value is null. The allowable index values for a1 are \{0,1,2\}, for a2 \{0,1,2,3,4,5,6,7,8,9\}, and for a3 there are no allowable indices.

Arrays in Java are accessed by

```
<array_reference> [<int_expression>]
```

The int_expression can be any expression that is assignment compatible with int and evaluates to an int between zero and the size of the array minus one. The legality of the index expression is checked at run time. If the expression is outside the permissible bounds, an ArrayIndexOutOfBoundsException exception is thrown.

The only member data element of an array is length, which is the number of components in the array. Therefore, filling an int array ia with zeros can be accomplished by

```
for (int i=0; i<ia.length; i++)
    ia[i] = 0;
```

Use of arrays in assignment statements is legal in Java, but must be done with care as the result is the assignment of the reference and not the creation of a copy of the assigned array. Consider the example below:

```
int[] a1 = {1,2,3}
int[] a2;
a2 = a1;
a1[0] = 0;
```

After this sequence of statements is executed, a2[0] will be 0 since a1 and a2 reference the same array, an array whose first element was set to zero by the final assignment statement.

To make a copy of an array in Java, the clone method must be used. Rewriting the preceding code sequence to enable this, we have

```
int[] a1 = {1,2,3}
int[] a2;
a2 = (int[])a1.clone();
a1[0] = 0;
```

We see that at the end of execution a2[0] will be 1 as a1 and a2 reference two different arrays.

Multi-dimensional arrays are represented by arrays of arrays in Java. Therefore, a two-dimensional array can be declared by

```
int[][] a;
```
meaning that a is a reference to an int[]. Array definitions of multi-dimensional arrays are also
the natural extension of the single-dimensional form. For example,

    int[][] a1 = {{1,2},{3,4},{5,6}}

creates a reference to an array of 3 int components, each component of size 2. Accessing the
array is also a natural extension.

    a1[3][1]

accesses the position that was initialized as 5.

Similarly, the array instance creation is extended to multiple dimensions as illustrated by
the following example:

    int[][] a2 = int[3][2];
    int cnt = 1;
    for (int i1=0; i1<3; i1++)
        for (int i2=0; i2<2; i2++)
            a2[i1][i2] = cnt++;

This code constructs an array that is identical to the array a1 created in the preceding array cre-
ation example.

It should also be noted that with this definition of multi-dimensional arrays, such array
components do not need to be the same length. As an example, consider the following code that
generates the Pascal’s triangle.

    int pas[][] = new int[size][];
    pas[0] = new int[1];
    pas[0][0] = 1;
    for (int i=1; i<pas.length; i++)
    {
        pas[i] = new int[i+1];
        pas[i][0] = 1;
        pas[i][i] = 1;
        for (int j=1; j<i; j++)
            pas[i][j] = pas[i-1][j-1] + pas[i-1][j];
    }

The component at index i of the pas array is an array with i+1 components. This duplicates the
structure of Pascal’s triangle.

Research: Research the language APL. Examine the set of array operators built into
that language and classify each as composite or aggregate.

5.3 Strings

There are two different approaches taken by languages in the modeling of character strings. The first approach considers the strings as a special case of an array where the range type is type character. This approach, found in Ada and C++, uses the mapping model and provides the same operations for strings as for arrays.

There are a number of difficulties associated with the string-as-mapping model. First, the string, by its very nature, changes its size dynamically at run time. Array implementations often fix the size of an array at compile time. Even those languages that permit run-time binding to size usually prevent that size from being dynamically changed once it is set. A second difficulty is that the operations that are commonly needed for strings are not available for arrays in general. Finally, the notation for expressing string constants is different than the notation used for expressing array constants.

For these reasons, it is advantageous to consider the alternative model of sequence for representing character strings. The primary advantages of a sequence are that it is dynamic and has unlimited length. BASIC and SNOBOL are two languages which use this approach. The string as sequence model is often implemented as a class in languages that support the object-oriented model. Java has a built-in class for strings that we examine in detail in Section 5.3.5.

5.3.1 Declaration and binding

Because the string is dynamic by nature, the declaration does not include a size restriction except perhaps in specifying a maximum size. When such a maximum is required by the language, the compiler reserves storage space for the maximum number of characters even though the actual size of the string can vary during run time up to that maximum. If no maximum is enforced, the binding of the string object to its location must be postponed until run time to permit flexibility in storage allocation.

5.3.2 Manipulation

We use the same general classes of operations here that we used for arrays, with attention to how they pertain to strings.

1. Selection - It is common for the selection of characters from a string to involve the selection of several consecutive characters, known as a substring. Therefore, the most common selection operation is substring selection that requires two parameters, either starting location and length or starting location and ending location. The latter form is equivalent in usage to an array slice. Note that selection of a substring of length 1 is equivalent to single component selection, except that its type will be string rather than the constituent type, character.

2. Construction - Placing individual characters into a string is accomplished by placing the
appropriate character sequence between delimiters. The use of single or double quotes (' or ") is by far the most common choice for a delimiter, and which is used depends on the syntax of the language.

3. **Assignment** - String assignment is usually provided as a direct extension of scalar assignment, using the same operator as well. Assignment to a string will dynamically change its size to the size of the assigned string. When strings are stored as a reference, the assignment will not result in a copy being made, and a separate facility must be supplied to permit copy assignment.

4. **Composite operators** - These are not applicable to strings.

5. **Aggregate operators** - The most common aggregate operator is concatenation. This creates a new string consisting of the left operand string followed by the right operand string. The length function is another common aggregate operator. It has one operand--the string itself--and returns the number of characters in the string.

   Another frequently provided aggregate operator is the substring search function. The operands are the target string and a pattern string. The result is an integer specifying the position of the first occurrence of the pattern within the target.

6. **Attributes** - Although an ideal implementation for a string would not limit the length, it is often necessary to do so. When this is the case, the maximum length is an attribute.

7. **Comparison** - Comparison of strings is done according to lexicographic ordering.

5.3.3 Implementation

The implementation of strings is illustrated by Figure 5.4. The descriptor includes the length of the string plus a pointer to the first character. The string itself is frequently stored as a linked list to permit the length to grow without limit. The characters are typically blocked together to form each list element. In languages enforcing a maximum length, the implementation utilizes contiguous storage similar to that used in the implementation of arrays.
One additional consideration is that the dynamic nature of strings makes it necessary to reclaim string storage no longer in use through a technique known as garbage collection. We outline this process here:

1. All available, unused blocks for string storage are linked together into a list known as the free list.
2. As string storage is required, blocks are moved from the free list to the list for that string.
3. When the free list is nearly exhausted, the garbage collection process is activated. This process traces through the list of all current strings, marking each block encountered as "in use". It then recreates the free list by linking together all blocks not marked "in use".

### 5.3.4 Strings in C++: A Modified Mapping Model

The programming language C++ inherits its string handling from its predecessor language C. It considers strings to be an array of characters with a number of enhancements.

The declaration of a string is identical to the declaration of an array of characters. For example,

```c
char s[10];
```

creates an array of 10 chars. This array can be used to store a string of up to 9 characters. The rea-
son it can store only 9 rather than 10 characters is that C++ uses what is called null termination for strings. This means a string is terminated by the first null character (‘\0’) that is encountered in the array. An empty string, for example, has the null character in the first array position, \texttt{s[0]}. For this reason, a string declared by

\begin{verbatim}
  char a_string[n];
\end{verbatim}

can be of any length from 0 to \(n-1\). Assignment to a string component is done just like any other array component assignment. The insertion and deletion of characters cannot be done directly, however, but rather require additional manipulation of the character array.

Several other amenities are given to C++ strings beyond what is given to other arrays. A special format for string literals is provided, namely, enclosing the characters of a string in double quotes ("'). For example,

\begin{verbatim}
  char a_string[10] = "good luck";
\end{verbatim}

stores the nine-character string on the right-hand side plus the null termination character in the 10 positions of \texttt{a_string}.

Since strings, like all C++ arrays, are stored by reference, assignment and comparison for equality, though legal, can be problematic. For this reason, C++ provides a library of functions, \texttt{string.h}, that provides string copying, concatenation, and comparison in addition to a function that will return the length of a string as determined by the position of the null character.

### 5.3.5 Strings in Java

Java implements strings using the sequence model through classes. We defer a detailed discussion of classes until Chapter 8, but we discuss two Java classes, \texttt{String} and \texttt{StringBuffer}, as implementations of the sequence model. Many of the features described in this section are later identified as class attributes, but we limit our perspective to their use with the two implementations of strings described below.

First, we analyze the difference between the models of these two classes. A \texttt{String} in Java, once created, never changes, a property known as immutability. Here it is important to distinguish between the string itself and the variable that references it. A variable can reference many different strings at different points of program execution, but each of those strings is immutable. As an illustration, consider the following:

\begin{verbatim}
  String s = new String("lover");
  s = new String("lovely");
\end{verbatim}

Here we see that Java provides \texttt{String} literals (enclosed in double quotes) and that it creates a \texttt{String} with the keyword \texttt{new}. There are a number of other ways \texttt{Strings} can be created, but we do not discuss them here. We also note that the variable \texttt{s} in the first statement references a five-character \texttt{String} and then, after the second statement, references a six-character \texttt{String}. Neither of
these strings is changeable, but either can be assigned to the same variable at different times.

In contrast, while the same strategy is possible with the StringBuffer class, namely

```java
StringBuffer s = new StringBuffer("lover");
s = new StringBuffer("lovely");
```

again creating two StringBuffers and making s reference them in sequence, StringBuffer presents another option. A StringBuffer object can be modified to represent a different string of characters. Consider the following:

```java
StringBuffer s = new StringBuffer("lover");
(s.append('y')).setCharAt(4,'l');
```

In this case only one StringBuffer is created. It begins as "lover", is modified to become "lovery" by the append, and finally becomes "lovely" after the setCharAt. This was not possible with String due to that class' immutability.

The preceding example shows the two ways that a StringBuffer can be modified: by changing the char at a specified position and by appending a char (or a String or objects of a number of other types).

We complete this section by examining how the classes of aggregate operations are provided by Java String and StringBuffer classes.

1. Selection - Both String and StringBuffer provide a charAt method that returns the char at a specified position in the string. For example,

```java
String s = new String("abcde");
char c = s.charAt(2);
```

assigns ‘c’ to c. Just as with arrays, Java begins at zero when numbering string positions. StringBuffer handles character selection in an identical way.

The selection of a substring is handled differently by the two classes. String class has a method substring that returns a new String that duplicates a range of indices from the operand String.

```java
String s = new String("disgusted");
String t = s.substring(3,7);
```

Here t is assigned the String “gust” as the parameters specify the substring starting at position 3 and ending at position 6 (the position immediately before position 7). The substring method is not present in class StringBuffer, though both StringBuffer and String have a method getChars that puts a substring into an array of chars.

2. Construction - Both String and StringBuffer provide for the creation of an empty string
and the creation of a new string that is a copy of another String. These constructions are illustrated by the following:

```java
String s1 = new String();
String s2 = new String("hello");
StringBuffer sb1 = new StringBuffer("goodbye");
StringBuffer sb2 = new StringBuffer(s2);
StringBuffer sb3 = new StringBuffer();
String s3 = new String(sb1);
```

In this example we see that in addition to creating empty strings and strings from Strings, it is also possible to create a String from a StringBuffer as illustrated by the final statement. A StringBuffer may not be created directly from another StringBuffer, however.

3. Assignment - Since all variables for classes in Java are references, the assignment operator applies to both String and StringBuffer but does not result in a copy being assigned. As an example, consider

```java
String s = new String("first");
String t;
    t = s;
```

Here t and s both reference the same String. Notice this is not a potential problem, however, since Strings are immutable, so there is no danger of changing the String references by one variable and have the other variable’s reference changed as a side-effect. Therefore, the assignment operator is “safe” with class String.

All is not so safe with class StringBuffer, however. We can see the danger in the following code:

```java
StringBuffer sb1 = new String("first");
StringBuffer sb2 = sb1;
    sb1.append("time");
```

At the conclusion of execution of these statements, sb2 will be “firsttime” as it references the same StringBuffer as sb1. In fact, copying a StringBuffer requires passing it through the String class as in the following statement:

```java
sb2 = new StringBuffer(sb1.toString());
```

4. Composite Operators - None are supported.

5. Aggregate Operators - Two important aggregate operators in String are concatenation and the location of a pattern string. Concatenation is accomplished by
s1.concat(s2)

which results in a new String that is the concatenation of Strings s1 and s2.

The following presents a pattern searching example:

s1.indexOf(s2)

This call returns an int that is the position of the first occurrence of String s2 within s1. If there is no such occurrence, the int returned is -1. There are also several variations of pattern searching including searching only after a specified position and searching from the right-hand end of the String.

Concatenation in StringBuffer can occur by use of the append method. Therefore, two StringBuffers are concatenated by

sb1.append(sb2);

Here sb1 has been modified from its original value by concatenating sb2 to it. There is no corresponding pattern search method for StringBuffer, but if required, pattern searching can be done by converting the StringBuffer to String. Therefore, a search for the first occurrence of sb2 in sb1 is written as

(new String(sb1)).indexOf(new String(sb2))

or

(sb1.toString()).indexOf(sb2.toString());

6. Attributes - The length attribute is available in identical form for both String and StringBuffer. The form is

s.length()

7. Comparison - Comparison is provided for class String by the compareTo method. Its form is

s1.compareTo(s2)

and it returns an int that is positive if s1 is greater than s2, zero if s1 and s2 are identical strings (though not necessarily the same string), and negative if s1 is less than s2. Here “less than” and “greater than” refer to lexicographic order.

StringBuffer provides no comparison and again requires a conversion to String.

Laboratory: For the language assigned to you, Is there a maximum length enforced on strings? Is it user definable?
5.4 Records

The implementation of the Cartesian product model in a programming language is called a record or structure. A record type is the Cartesian product of two or more other types, each of which may be simple or aggregate types themselves. The record type was first introduced in COBOL and is now present in some form in all recent programming languages.

5.4.1 Declaration and binding

Record types are declared by specifying the name of the record type and, for each component of the record, the name and type of the component. For example, in Ada the simple form of a record declaration is

record
  <identifier-list> : <component-type> [:= <expression>];
{<identifier-list> : <component-type> [:= <expression>]};
end record;

The bindings that occur are of the components to the record type and the component types and component names to each component. All of these bindings occur at compile time.

Similarly, in C and C++, the record type is specified by a struct, whose general form is

struct <name-of-structure> {  
  <component-type> <identifier-list>;
  {<component-type> <identifier-list>;}
};

The <expression> being present in the record declaration causes run-time initialization of any data element of this <component-type> or types to the value of the <expression> at the time the record element is created. For example, if we have the declaration

    type EXREC is record
      A : integer := 0;
      B : character := '*';
    end record;

then when a block is entered with the variable declaration

    C : EXREC;

the components of C are initialized to 0 and '*'.
5.4.2 Manipulation

1. Selection - The selection of a component of a record is done by specifying the name of the record element and the component name. The form is commonly

   `<record-identifier> . <component-name>`

although alternative notations can be used, such as in ALGOL 68 which selects by

   `<component-name> of <record-identifier>`

Some languages, such as Pascal, permit a default specification of the record identifier within a block by prefacing the block with a clause that identifies the default record identifier.

2. Construction - Record literals can be constructed in a manner similar to array literal construction, with a list of expressions either in the order of the components or paired with the component name.

3. Assignment - Record assignment is of the usual form

   `<record-identifier> := <record-expression>`

where both sides are of the same record type. As usual, careful consideration must be given to the implementation to determine whether this is a reference or a copy assignment.

4. Composite Operators - No composite operators are defined for records.

5. Aggregate Operators - No aggregate operators are defined for records.

6. Attributes - The size of a record, either in number of components or bytes of storage, can be an attribute.

7. Comparison - Records of identical composition may be compared for equality and inequality.

   Research: Invent an algorithm for the assignment of records.

5.4.3 Implementation

A record descriptor can be easily constructed at run time, because all components are of fixed size. The descriptor, as shown in Figure 5.5, contains the name and a reference to the location of each component. Components are usually stored contiguously as illustrated here, so the pointer to a component can be given as the offset from a base address.
The nesting of data aggregates can occur at any level. When an aggregate is a component of another aggregate, the data reference points to the descriptor of the aggregate rather than the data itself. An example of the implementation of a complex nested data aggregate is given in Figure 5.6. Note that all components of a record are stored contiguously with the descriptor occupying that space if the component is an aggregate itself.
Figure 5.6 Nested data aggregates

X : record;
   A : String;
   B : array (1..3) of character;
   C : integer;
end record;

Reinforce: The storage of records can be complicated. Design a formula that computes the storage needs of a record.

5.4.4 Records in Java

Java does not provide any direct form of records but rather provides them as a specialized restricted class. Classes must be declared outside of a program unit. We look in more detail at the scope and access control for classes in Chapter 8. Here we look at a simple example of a class as a record through the following definition:

class Student {
   String id;
   in age;
   String name;
   float gpa;
   int[] grades;
}
In any program unit within the scope of this class, the components can be referenced as illustrated by the following:

```java
Student s1 = new Student();
s1.age = 21;
s1.name = "Sally";
s1.grades = new int[20];
s1.grades[0] = 95;
...
```

### 5.5 Streams

A pure implementation of the sequence model is the stream, an aggregate commonly used for input and output. We will examine how streams are declared and manipulated in programming languages.

#### 5.5.1 Declaration and Binding

Streams have three bindings in addition to the standard bindings present for any data type. They are

1. Mode binding - input or output
2. Source or destination binding - if input, the stream must be bound to a data source and if output to a data destination
3. Component type - the type of all components in the stream.

Streams are classified as input or output depending upon their role. An input stream is a sequence that already exists and whose values are retrieved by the program. An output stream is a sequence that is constructed by the program as data is added to the stream. In its declaration, a stream is bound to its role, either input or output.

A stream must also be bound to its source (if input) or destination (if output). Source and destination are usually files, although they can be structures internal to the program such as arrays, strings, or other streams.

The third binding required is from a stream to its component type. Frequently, this type is the generic byte, but some languages support streams of other legal types including composite types. The most frequently supported composite type for streams is the record.

#### 5.5.2 Manipulation

Because a stream is a pure sequence, there is only one basic operation for each mode. For an output stream, that operation is adding an object onto the end of the stream, while for an input stream, it is taking an object from the front.

Other operations are sometimes provided for streams. These commonly include the concatenation of several streams to form a new stream, the ability to skip over one or more objects in
an input stream, and the ability to move back to the beginning of an input stream.

5.5.3 Streams in Java

Like strings, streams are also implemented via classes in Java. Rather than binding streams to their mode and their source-destination type, Java provides separate classes for each mode/source-destination type combination.

For example, the class FileInputStream represents input streams with a file source. These streams are bound to their source file in the constructor via a string that specifies the complete operating system path of the file. The type of each element of the file is limited to byte. Three forms of the read operation are available:

```java
int read()
void read(byte[] b)
void read(byte[] b, int off, int len)
```

The first form reads a single byte from the front of the stream and returns it as an int with value 0 to 255. The second form reads the first b.length bytes into the byte array b. The third form reads the first len bytes into b[off]...b[off+len-1].

Skipping the next n bytes in an input stream is specified by

```java
long skip(long n)
```

where the value returned is the actual number of bytes skipped. The number of bytes remaining in an input stream can be obtained in Java by the method

```java
int available()
```

Sources other than files are supported for Java streams, with the same set of operations available for each. These sources are byte array (ByteArrayInputStream), string (StringBufferInputStream), and output stream (PipedInputStream). A facility to concatenate a number of input streams into a single stream is also included.

Output streams are handled similarly by Java, with three destination types: file (FileOutputStream), byte array (ByteArrayOutputStream), and input stream (PipedOutputStream). The only appropriate operation for Java output streams is write where, as with read, a single byte or an array of bytes may be written to the stream with a single call.

5.6 Aggregates as Collection Classes

The object-oriented view of aggregates has included a hierarchy of classes used to store multiple objects. These are called collection classes. This was first included in Smalltalk, then later as a part of the C++ Standard Template Library (STL). In this section, we examine the collection classes as instituted in Java 1.2 and connect them to our aggregate models.
5.6.1 Java Collection Classes

We have already seen two aggregates that are implemented by Java classes: string and stream. We now look at three other forms of aggregates that fall within the class framework of Java. Again, we postpone a careful discussion of the object-oriented concepts involved here and concentrate on the ways aggregates are constructed and used. We highlight the Java distinction between an interface and an implementation in our discussion, however.

A Java interface is a collection of constants and method call protocols that can be implemented by a number of different implementation classes. The implementation classes must provide implementations of all the methods in the interface plus optionally other class components.

There is an inheritance hierarchy to both the collection interfaces and their implementation classes, but that will be ignored in the present discussion. We focus our attention on the three collection interfaces from Java: Set, List, and Map. This is a simplified discussion of the collection interface structure, and more details will be given later. Here we examine how these three interfaces relate to our aggregate models.

5.6.2 Java Collections

The Collection interface in Java contains methods that are common to the Set and the List interfaces. We use a different classification scheme for operations on Java collections than we have used in the earlier sections of this chapter for aggregate operators. We classify operations in Java collections into five categories.

1. Construction - Java collections are constructed with two different formats. A parameterless constructor creates an initially empty collection, while a constructor with a single parameter, which references another collection, constructs a new collection whose structure and interface are possibly different from the parameter collection, but whose contained elements are the same.

2. Addition - Java collections support two methods that add elements to the collection.

   ```java
   boolean add(item)
   boolean addAll(collection)
   ```

   The `add` method has a single parameter and adds that parameter to the collection. The `addAll` method also has a single parameter, but its parameter must be another collection, all of whose elements are added to the target collection.

3. Removal - There are four generic collection removal methods. They are

   ```java
   void clear()
   boolean remove(item)
   boolean removeAll(collection)
   boolean retainAll(collection)
   ```
The method **clear** removes all elements, **remove** removes a single element, **removeAll** removes all elements in the collection parameter, and **retainAll** removes all elements not in the collection parameter.

4. **Query** - The query operations request information about the collection. The four query operations are

```java
boolean contains(element)
boolean containsAll(collection)
boolean isEmpty()
int size()
```

The roles of these operations should be clear from their names.

5. **Retrieval** - Retrieval from a Java collection is done through an iterator. An iterator is an object of class **Iterator** that can be created from a collection \( c \) by the statement

```java
Iterator it = c.iterator();
```

where \( c \) references a collection object. The iterator has two methods, **next** and **hasNext**, and with these, the iterator can successively retrieve all elements in collection \( c \) by

```java
for (Iterator it = c.iterator(); it.hasNext(); )
    process(it.next());
```

The iterator will retrieve all elements in its collection \( c \), but the order of retrieval depends on the implementation of the collection.

5.6.3 **The Set Interface**

The **Set** interface is a collection that contains no duplicate items. It includes all of the **Collection** operations from Section 5.6.2 with only the proviso that the **add** operation fails to insert any element that is already in the collection and, in that case, returns **false**. The **addAll** method will also refuse to add a duplicate element but will return **false** only if no new elements are added to the set.

The Java Set interface supports all the standard mathematical set operations through the Collection class methods. The mathematical operators and their corresponding method calls are as follows:

```java
a = a ∪ b  a.addAll(b)
(a = a ∩ b  a.retainAll(b)
a = a - b  a.removeAll(b)
```
b ⊆ a \quad a\text{.containsAll}(b)

There are two implementations of the Set interface in Java: HashSet and TreeSet. They differ in the structure used to store the elements in the set. The HashSet uses a hash table and therefore has very efficient insertion, removal, and query operations. Retrieval order is non-deterministic for HashSet. Two additional parameters can be optionally provided in the constructor of a HashSet: the initial capacity of the hash table and the load factor of the hash table. These factors can be used to tune the storage efficiency of a HashSet.

The TreeSet stores the collection in a balanced tree structure. It requires that an ordering be defined on the elements in the structure and an iterator returns the elements from first to last in that ordering. Additional operations are applicable to a TreeSet, namely

Object first()
Object last()
TreeSet subSet(startElement, endElement)

Details concerning these methods and how ordering is specified are not given here, but their roles are obvious.

5.6.4 The List Interface

The List interface is the Java implementation of the sequence aggregate model. The List differs from the Set in that duplicates are allowed and in the List, elements are accessible by their position in the List. The position of an element is referred to by its numerical index, where the first element has index 0, the second has index 1, etc.

The List interface includes all the operations of the Collection interface, but some have added meaning for List. We summarize these by category in the following.

1. Construction - These constructors are of identical form to those of the Collection interface.

2. Addition - The add and addAll methods from the Collection interface append parameter elements to the end of a List. Two positional addition methods are also available for List. They are

    void add(index, element)
    boolean addAll(index, collection)

These insert their parameter elements starting at position index rather than at the end of the List.

3. Removal - Since duplicates are permitted in a List, it is necessary to specify that remove removes the first occurrence of the specified elements. Positional removal is also possible by

    Object remove(index)
which removes the element at position \texttt{index} and returns that element as the result of the method. There is also an operation that changes the value at a specified position. This is identical in function to a \texttt{remove} immediately followed by an \texttt{add} to the same position. This method is

\begin{verbatim}
Object set(index,element)
\end{verbatim}

The return value is the element that has been removed.

4. \textbf{Query} - In addition to \texttt{contains} and \texttt{containsAll} from the Collection interface, elements can be searched for to obtain their index. This is done by

\begin{verbatim}
int indexOf(element)
int lastIndexOf(element)
\end{verbatim}

These return the position of the first and last occurrences of element in the List, respectively.

5. \textbf{Retrieval} - The iterator for a List has added functionality beyond that of a Collection iterator. The elements are retrieved in positional sequence, but the List may be traversed in either direction. This is provided by the methods \texttt{previous} and \texttt{hasPrevious}, which are similar in function to \texttt{next} and \texttt{hasNext}. The List interface also provides individual and range retrievals by position. The methods for this are

\begin{verbatim}
Object get(index)
List subList(from, to)
\end{verbatim}

where the parameters are all \texttt{int} specifying positional index values. It is necessary to point out that \texttt{from} is the index of the first element retrieved and \texttt{to} is the index of the first element not retrieved. In other words, as with substrings earlier, elements with index \texttt{from..to-1} are retrieved by \texttt{subList}.

There are two implementations of the List interface in Java: \texttt{ArrayList} and \texttt{LinkedList}. The \texttt{ArrayList} implementation stores the List in an array, which supports efficient retrieval, both sequential and positional, and is efficient in adding elements to the rear of the List. The \texttt{LinkedList} implementation retains efficiency in sequential retrieval but also provides efficient internal additions and removals. Its disadvantage is its much slower positional retrieval.

5.6.5 \textbf{The Map Interface}

The Java Map interface is a direct implementation of the mapping aggregate model from this chapter. It consists of a collection of key/value pairs, where both keys and values can be a reference to any class of object. The keys of a Map are unique, but the values are not necessarily unique.

The operations for the Map interface are as follows:
1. **Construction** - The constructions are the same as those for the Collection interface.

2. **Addition**- The two forms of addition are

   ```
   Object put(key, value)
   void putAll(anotherMap)
   ```

   The method `put` adds the specified key/value pair while `putAll` adds all of the key/value pairs in `anotherMap`. In either case, if the key of an element in the parameter is already present in the Map, the key/value pair in the Map will be replaced by the pair with the same key from the parameter. The previous value associated with the key is returned for the method `put` in this situation.

3. **Removal** - Either one or all pairs can be removed by

   ```
   Object remove(key)
   void clear()
   ```

   The returned Object for `remove` is the value part of the key/value pair that is actually removed.

4. **Query** - Seven query methods are provided with the Map interface:

   ```
   boolean containsKey(key)
   boolean containsValue(value)
   int size()
   boolean isEmpty()
   Set keySet()
   Collection values()
   Set entrySet()
   ```

   Only the final three of these require further explanation. The method `keySet` returns the set of all keys in the Map. This is a set because no duplicate keys are allowed. Method `values` returns the collection of all values in the Map—a collection because duplicates may occur. The final method, `entrySet`, returns the set of all key/value pairs—a set because all pairs are unique. The class of the elements in this set is a special class called `Entry` that is provided along with interface Map. An `Entry` consists of a key/value pair.

5. **Retrieval** - No iterator is defined for maps, but an iterator can be defined on the key set. Therefore, if `m` is a Map, all values can be retrieved in key sequence by

   ```
   HashSet k = m.keySet();
   for (Iterator it = k.iterator(); it.hasNext(); )
     process(m.get(it.next()));
   ```
There are two implementations of the Map interface: `HashMap` and `TreeMap`, which are a hashtable and a balanced tree structure to store the values. The implementation chosen will again depend upon the operational requirements of the collection.

**Terms - Chapter 5**
- data aggregate
- constituent type
- cardinality
- Cartesian product
- mapping
- domain
- range
- discriminated union
- sequence
- powerset
- array
- unconstrained array
- composite operators
- aggregate operators
- attribute
- lexicographic order
- multi-indexed array
- row major order
- column major order
- packing
- garbage collection
- record
- variant record
- discriminant
- file