chapter 2

Data Design and Implementation

Goals

Measurable goals for this chapter include that you should be able to

- describe the benefits of using an abstract data type (ADT)
- explain the difference between a primitive type and a composite type
- describe an ADT from three perspectives: logical level, application level, and implementation level
- explain how a specification can be used to document the design of an ADT
- describe, at the logical level, the component selector, and describe appropriate applications for the Java built-in types: class and array
- create code examples that demonstrate the ramifications of using references
- describe several hierarchical types, including aggregate objects and multidimensional arrays
- use packages to organize Java compilation units
- use the Java Library classes `String` and `ArrayList`
- identify the scope of a Java variable in a program
- explain the difference between a deep copy and a shallow copy of an object
- identify, define, and use Java exceptions when creating an ADT
- list the steps to follow when creating ADTs with the Java *class* construct

This chapter centers on data and the language structures used to organize data. When problem solving, the way you view the data of your problem domain and how you structure the data that your programs manipulate greatly influence your success. Here you learn how to deal with the complexity of your data using abstraction and how to use the Java language mechanisms that support data abstraction.

In this chapter, we also cover the various data types supported by Java: the primitive types (`int`, `float`, and so on), classes, interfaces, and the array. The Java class mechanism is used to create data types beyond those directly provided by the language. We review some of the class-based types that are provided in the Java Class Library and show you how to create your own class-based types. We use the Java class mechanism to encapsulate the data structures you are studying, as ADTs, throughout the textbook.

2.1 Different Views of Data

Data Types

When we talk about the function of a program, we usually use words like *add*, *read*, *multiply*, *write*, *do*, and so on. The function of a program describes what it does in terms of the verbs in the programming language. The data are the nouns of the programming world: the objects that are manipulated, the information that is processed by a computer program.

Data The representation of information in a manner suitable for communication or analysis by humans or machines

Data type A category of data characterized by the supported elements of the category and the supported operations on those elements

Atomic or primitive type A data type whose elements are single, nondecomposable data items

Humans have evolved many ways of encoding information for analysis and communication, for example letters, words, and numbers. In the context of a programming language, the term *data* refers to the representation of such information, from the problem domain, by the data types available in the language.

A data type can be used to characterize and manipulate a certain variety of data. It is formally defined by describing:

1. the collection of elements that it can represent.
2. the operations that may be performed on those elements.

Most programming languages provide simple data types for representing basic information—types like integers, real numbers, and characters. For example, an integer might represent a person's age; a real number might represent the amount of money in a bank account. An integer data type in a language would be formally defined by listing the range of numbers it can represent and the operations it supports, usually the standard arithmetic operations.

The simple types are also called atomic types or primitive types, because they cannot be broken into parts. Languages usually provide ways for a programmer to combine primitive types into more complex structures, which can capture relationships among the individual data items. For example, a programmer can combine two primitive integer values to represent a point in the x-y plane or create a list of real numbers to represent the scores of a class of students on an assignment. A data type composed of multiple elements is called a composite type.

Composite type A data type whose elements are composed of multiple data items

Data abstraction The separation of a data type's logical properties from its implementation

Just as primitive types are partially defined by describing their domain of values, composite types are partially defined by the relationship among their constituent values.

Composite data types come in two forms: unstructured and structured. An *unstructured* composite type is a collection of components that are not organized with respect to one another. A *structured* composite type is an organized collection of components in which the organization determines the means of accessing individual data components or subsets of the collection. In addition to describing their domain of values, primitive types are defined by describing permitted operations. With composite types, the main operation of interest is accessing the elements that make up the collection.

The mechanisms for building composite types in the Java language are called reference types. (We see why in the next section.) They include arrays and classes, which you are probably familiar with, and interfaces. We review all of these mechanisms in the next section.

In a sense, any data processed by a computer, whether it is primitive or composite, is just a collection of bits that can be turned on or off. The computer itself needs to have data in this form. Human beings, however, tend to think of information in terms of somewhat larger units like numbers and lists, and thus we want at least the human-readable portions of our programs to refer to data in a way that makes sense to us. To separate the computer's view of data from our own, we use data abstraction to create another view.

Data Abstraction

Many people feel more comfortable with things that they perceive as real than with things that they think of as abstract. Thus, *data abstraction* may seem more forbidding than a more concrete entity like *integer*. Let's take a closer look, however, at that very concrete—and very abstract—integer you've been using since you wrote your earliest programs. Just what is an integer? Integers are physically represented in different ways on different computers. In the memory of one machine, an integer may be a binary-coded decimal. In a second machine, it may be a sign-and-magnitude binary. And in a third one, it may be represented in two's-complement binary notation. Although you may not be familiar with these terms, that hasn't stopped you from using integers. (You can learn about these terms in an assembly language or computer organization course, so we do not explain them here.) Figure 2.1 shows some different representations of an integer.

The way that integers are physically represented determines how the computer manipulates them. As a Java programmer, however, you don't usually get involved at this level; you simply use integers. All you need to know is how to declare an `int` type variable and what operations are allowed on integers: assignment, addition, subtraction, multiplication, division, and modulo arithmetic.

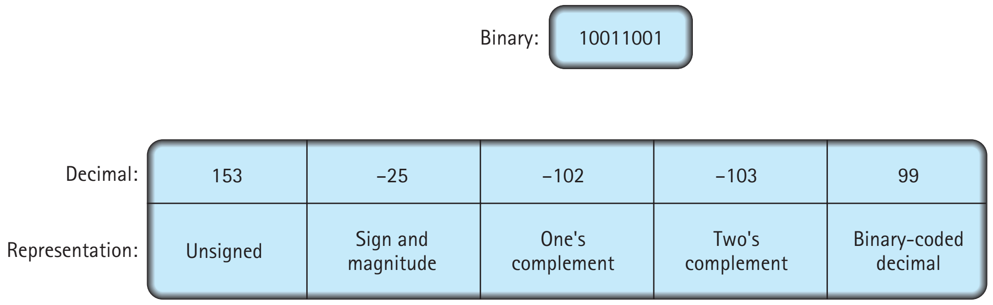


Figure 2.1 *The decimal equivalents of an 8-bit binary number*

Consider the statement

```
distance = rate * time;
```

It's easy to understand the concept behind this statement. The concept of multiplication doesn't depend on whether the operands are, say, integers or real numbers, despite the fact that integer multiplication and floating-point multiplication may be implemented in very different ways on the same computer. Computers would not be very popular if every time we wanted to multiply two numbers we had to get down to the machine-representation level. But we don't have to: Java has provided the `int` data type for us, hiding all the implementation details and giving us just the information we need to create and manipulate data of this type.

We say that Java has encapsulated integers for us. Think of the capsules surrounding the medicine you get from the pharmacist when you're sick. You don't have to know anything about the chemical composition of the medicine inside to recognize the big blue-and-white capsule as your antibiotic or the little yellow capsule as your decongestant. Data encapsulation means that the physical representation of a program's data is hidden by the language. The programmer using the data doesn't see the underlying implementation, but deals with the data only in terms of its logical picture—its abstraction.

Data encapsulation The separation of the representation of data from the applications that use the data at a logical level; a programming language feature that enforces information hiding

But if the data are encapsulated, how can the programmer get to them? Operations must be provided to allow the programmer to create, access, and change the data. Let's look at the operations Java provides for the encapsulated data type `int`. First of all, you can create variables of type `int` using declarations in your program. Then you can assign values to these integer variables by using the assignment operator and perform arithmetic operations on them using `+`, `-`, `*`, `/`, and `%`. Figure 2.2 shows how Java has encapsulated the type `int` in a nice neat black box.

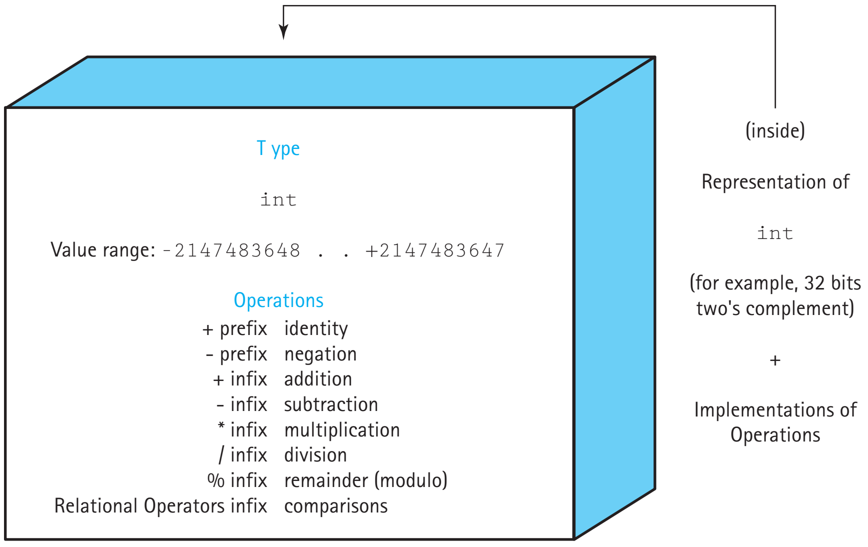

T ype
int
Value range: -2147483648 . . +2147483647
Operations
+ prefix identity
- prefix negation
+ infix addition
- infix subtraction
* infix multiplication
/ infix division
% infix remainder (modulo)
Relational Operators infix comparisons
(inside)
Representation of
int
(for example, 32 bits two's complement)
+
Implementations of Operations

Figure 2.2 *A black box representing an integer*

The point of this discussion is that you have been dealing with a logical data abstraction of integer since the very beginning. The advantages of doing so are clear: you can think of the data and the operations in a logical sense and can consider their use without having to worry about implementation details. The lower levels are still there—they're just hidden from you.

Remember that the goal in design is to reduce complexity through abstraction. We extend this goal with another: to protect our data abstraction through encapsulation. We refer to the set of all possible values (the domain) of an encapsulated data "object," plus the specifications of the operations that are provided to create and manipulate the data, as an abstract data type (ADT for short).

Abstract data type (ADT) A data type whose properties (domain and operations) are specified independently of any particular implementation

In effect, all the Java built-in types are ADTs. A Java programmer can declare variables of those types without understanding the underlying implementation. The programmer can initialize, modify, and access the information held by the variables using the provided operations.

In addition to the built-in ADTs, Java programmers can use the Java *class* mechanism to build their own ADTs. For example, the `Date` class defined in Chapter 1 can be viewed as an ADT. Yes, it is true that the programmers who created it need to know about its underlying implementation; for example, they need to know that a `Date` is composed of three `int` instance variables, and they need to know the names of the instance variables. The application programmers who use the `Date` class, however, do not need this information. They only need to know how to create a `Date` object and how to invoke the exported methods to use the object.

Data Structures

A single integer can be very useful if we need a counter, a sum, or an index in a program. But generally, we must also deal with data that have many parts and complex interrelationships among those parts. We use a language's composite type mechanisms to build structures, called data structures, which mirror those interrelationships. Note that the data elements that make up a data structure can be any combination of primitive types, unstructured composite types, and structured composite types.

Data structure A collection of data elements whose logical organization reflects a relationship among the elements. A data structure is characterized by accessing operations that are used to store and retrieve the individual data elements; the implementation of the composite data members in an ADT

When designing our data structures we must consider how the data is used because our decisions about what structure to impose greatly affect how efficient it is to use the data structure. Computer scientists have developed classic data, such as lists, stacks, queues, trees, and graphs, through the years. They form the major area of focus for this textbook.

In languages like Java, that provide an encapsulation mechanism, it is best to design our data structures as ADTs. We can then hide the detail of how we implement the data structure inside a class that exports methods for using the structure. For example, in Chapter 3 we develop a list data structure as an ADT using the Java *class* and *interface* constructs.

As we saw in Chapter 1, the basic operations that are performed on encapsulated data can be classified into categories. We have already seen three of these: *constructor*, *transformer* and *observer*. As we design operations for data structures, a fourth category becomes important: *iterator*. Let's take a closer look at what each category does.

- A constructor is an operation that creates a new instance (object) of the data type. A constructor that uses the contents of an existing object to create a new object is called a *copy constructor*.
- Transformers (sometimes called *mutators*) are operations that change the state of one or more of the data values, such as inserting an item into an object, deleting an item from an object, or making an object empty.
- An observer is an operation that allows us to observe the state of one or more of the data values without changing them. Observers come in several forms: *predicates* that ask if a certain property is true, *accessor* or *selector* methods that return a value based on the contents of the object, and *summary* methods that return information about the object as a whole. A Boolean method that returns `true` if an object is empty and `false` if it contains any components is an example of a predicate. A method that returns a copy of the last item put into a structure is an example of an accessor method. A method that returns the number of items in a structure is a summary method.
- An iterator is an operation that allows us to process all the components in a data structure sequentially. Operations that return successive list items are iterators.

Data structures have a few features worth noting. First, they can be "decomposed" into their component elements. Second, the organization of the elements is a feature of the structure that affects how each element is accessed. Third, both the arrangement of the elements and the way they are accessed can be encapsulated.

Note that although we design our data structures as ADTs, data structures and ADTs are not equivalent. We could implement a data structure without using any data encapsulation or information hididng whatsoever (but we won't!). Also, the fact that a construct is defined as an ADT does not make it a data structure. For example, the `Date` class defined in Chapter 1 implements a Date ADT, but that is not considered to be a data structure in the classical sense. There is no structural relationship among its components.

Data Levels

An ADT specifies the logical properties of a data type. Its implementation provides a specific representation such as a set of primitive variables, an array, or even another ADT. A third view of a data type is how it is used in a program to solve a particular problem; that is, its application. If we were writing a program to keep track of student grades, we would need a list of students and a way to record the grades for each student. We might take a by-hand grade book and model it in our program. The operations on the grade book might include adding a name, adding a grade, averaging a student's grades, and so forth. Once we have written a specification for our grade-book data type, we must choose an appropriate data structure to use to implement it and design the algorithms to implement the operations on the structure.

In modeling data in a program, we wear many hats. We must determine the abstract properties of the data, choose the representation of the data, and develop the operations that encapsulate this arrangement. During this process, we consider data from three different perspectives, or levels:

1. *Logical (or abstract) level:* An abstract view of the data values (the domain) and the set of operations to manipulate them. At this level, we define the ADT.
2. *Application (or user) level:* A way of modeling real-life data in a specific context; also called the problem domain. Here the application programmer uses the ADT to solve a problem.
3. *Implementation level:* A specific representation of the structure to hold the data items, and the coding of the operations in a programming language. This is how we actually represent and manipulate the data in memory: the underlying structure and the algorithms for the operations that manipulate the items on the structure. For the built-in types, this level is hidden from the programmer.

An Analogy

Let's look at a real-life example: a library. A library can be decomposed into its component elements: books. The collection of individual books can be arranged in a number of ways, as shown in Figure 2.3. Obviously, the way the books are physically arranged on the shelves determines how one would go about looking for a specific volume. The particular library we're concerned with doesn't let its patrons get their own books, however; if you want a book, you must give your request to the librarian, who gets the book for you.

The library "data structure" is composed of elements (books) with a particular interrelationship; for instance, they might be ordered based on the Dewey decimal system.

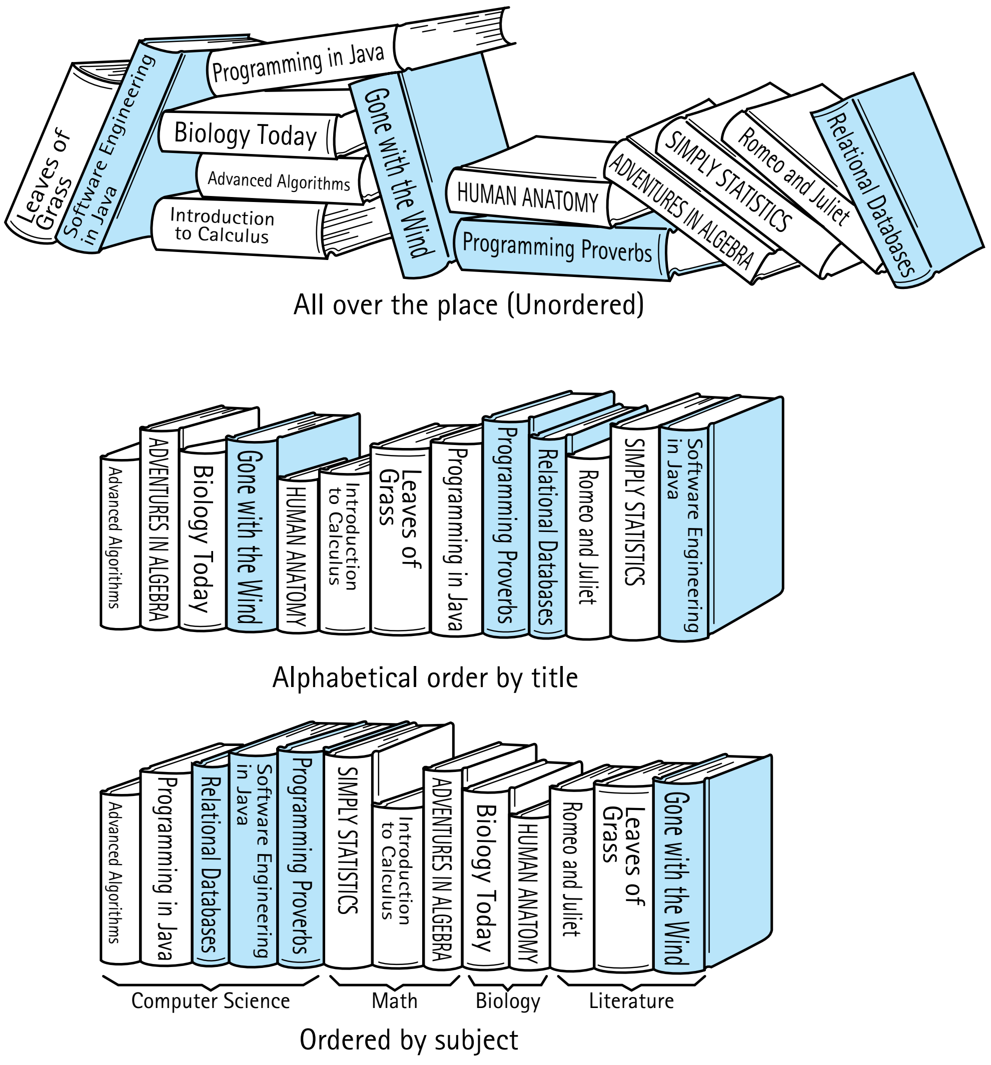


Figure 2.3 *A collection of books ordered in different ways*

Accessing a particular book requires knowledge of the arrangement of the books. The library user doesn't have to know about the structure, though, because it has been encapsulated: Users access books only through the librarian. The physical structure and abstract picture of the books in the library are not the same. The online catalog provides logical views of the library—ordered by subject, author, or title—that are different from its underlying representation.

We use this same approach to data structures in our programs. A data structure is defined by (1) the logical arrangement of data elements, combined with (2) the set of operations we need to access the elements. Let's see what our different viewpoints mean in terms of our library analogy. At the application level, there are entities like the Library of Congress, the Dimsdale Collection of Rare Books, the Austin City Library, and the North Amherst branch library.

At the logical level, we deal with the "what" questions. What is a library? What services (operations) can a library perform? The library may be seen abstractly as "a collection of books" for which the following operations are specified:

- Check out a book.
- Check in a book.
- Reserve a book that is currently checked out.
- Pay a fine for an overdue book.
- Pay for a lost book.

How the books are organized on the shelves is not important at the logical level, because the patrons don't actually have direct access to the books. The abstract viewer of library services is not concerned with how the librarian actually organizes the books in the library. The library user only needs to know the correct way to invoke the desired operation. For instance, here is the user's view of the operation to check in a book: Present the book at the check-in window of the library from which the book was checked out, and receive a fine slip if the book is overdue.

At the implementation level, we deal with the answers to the "how" questions. How are the books cataloged? How are they organized on the shelf? How does the librarian process a book when it is checked in? For instance, the implementation information includes the fact that the books are cataloged according to the Dewey decimal system and arranged in four levels of stacks, with 14 rows of shelves on each level. The librarian needs such knowledge to be able to locate a book. This information also includes the details of what happens when each of the operations takes place. For example, when a book is checked back in, the librarian may use the following algorithm to implement the check-in operation:

CheckInBook

```
Examine due date to see whether the book is late.
if book is late
    Calculate fine.
    Issue fine slip.
Update library records to show that the book has been returned.
Check reserve list to see if someone is waiting for the book.
if book is on reserve list
    Put the book on the reserve shelf.
else
    Replace the book on the proper shelf, according to the library's shelf arrangement scheme.
```

All this, of course, is invisible to the library user. The goal of our design approach is to hide the implementation level from the user.

Picture a wall separating the application level from the implementation level, as shown in Figure 2.4. Imagine yourself on one side and another programmer on the other side. How do the two of you, with your separate views of the data, communicate across this wall? Similarly, how do the library user's view and the librarian's view of the library come together? The library user and the librarian communicate through the data abstraction. The abstract view provides the specification of the accessing operations without telling how the operations work. It tells *what* but not *how*. For instance, the abstract view of checking in a book can be summarized in the following specification:

float CheckIn (book)

Effect:	Accesses book and checks it into this library. Returns a fine amount (0 if there is no fine).
Preconditions:	Book was checked out of this library; book is presented at the check-in desk.
Postconditions:	return value = (amount of fine due); contents of this library is the original contents + book
Exception:	This library is not open

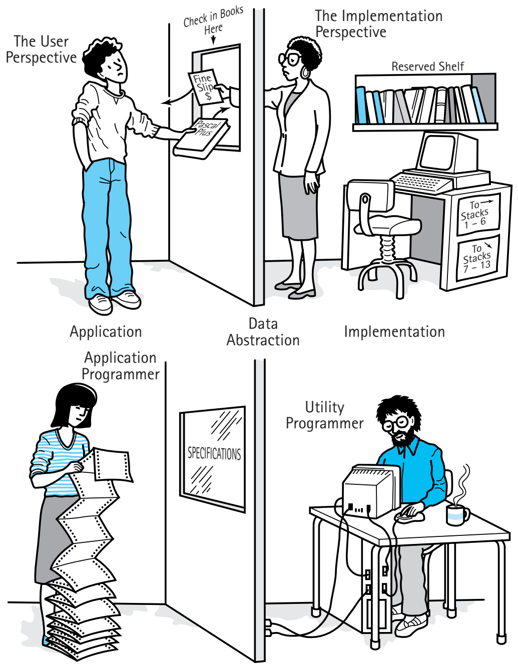


Figure 2.4 *Communication between the application level and implementation level*

The only communication from the user into the implementation level is in terms of input specifications and allowable assumptions—the preconditions of the accessing routines. The only output from the implementation level back to the user is the transformed data structure described by the output specifications, or postconditions, of the routines, or the possibility of an exception being raised. Remember that exceptions are extraordinary situations that disrupt the normal processing of the operation. The abstract view hides the underlying structure but provides functionality through the specified accessing operations.

Although in our example there is a clean separation, provided by the library wall, between the use of the library and the inside organization of the library, there is one way that the organization can affect the users—efficiency. For example, how long does a user have to wait to check out a book? If the library shelves are kept in an organized fashion, as described above, then it should be relatively easy for a librarian to retrieve a book for a customer and the waiting time should be reasonable. On the other hand, if the books are just kept in unordered piles, scattered around the building, shoved into corners and piled on staircases, the wait time for checking out a book could be very long. But in such a library it sure would be easy for the librarian to handle checking in a book—just throw it on the closest pile!

The decisions we make about the way data are structured affect how efficiently we can implement the various operations on that data. One structure leads to efficient implementation of some operations, while another structure leads to efficient implementation of other operations. Efficiency of operations can be important to the users of the data. As we look at data structures throughout this textbook we discuss the benefits and drawbacks of various design structure decisions. We often study alternative organizations, with differing efficiency ramifications.

When you write a program as a class assignment, you often deal with data at each of our three levels. In a job situation, however, you may not. Sometimes you may program an application that uses a data type that has been implemented by another programmer. Other times you may develop "utilities" that are called by other programs. In this book we ask you to move back and forth between these levels.

2.2 Java's Built-In Types

Java's classification of built-in data types is shown in Figure 2.5. As you can see, there are eight primitive types and three composite types; of the composite types, two are unstructured and one is structured. You are probably somewhat familiar with several of the primitive types and the composite types `class` and `array`.

In this section, we review all of the built-in types. We discuss them from the point of view of two of the levels defined in the previous section: the logical (or abstract) level and the application level. We do not look at the implementation level for the built-in types, since the Java environment hides it and we, as programmers, do not need to understand this level in order to use the built-in types. (Note, however, that when we begin to build our own types and structures, the implementation view becomes one of our major concerns.) For the built-in types we can interpret the remaining two levels as follows:

- The logical or abstract level involves understanding the domain of the data type and the operations that can be performed on data of that type. For the composite

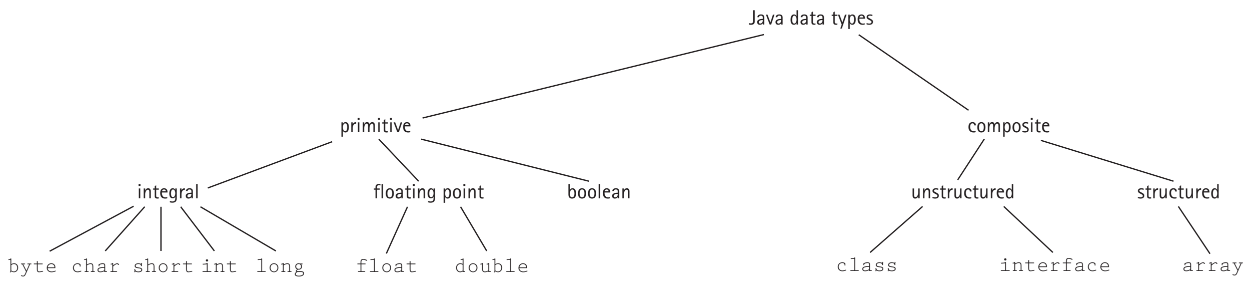


Figure 2.5 *Java data types*

types, the main operation of concern is how to access the various components of the type.

- The application level—in other words, the view of how we use the data types—includes the rules for declaring and using variables of the type, in addition to considerations of what the type can be used to model.

Primitive Data Types

Java's primitive types are `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`. These primitive types share similar properties. We first look closely at the `int` type from our two points of view, and then we give a summary review of all the others. We understand that you are already familiar with the `int` type; we are using this opportunity to show you how we apply our two levels to the built-in types.

Logical Level

In Java, variables of type `int` can hold an integer value between −2147483648 and 2147483647. Java provides the standard prefix operations of unary plus (+) and unary minus (-). Also, of course, the infix operations of addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). We are sure you are familiar with all of these operations; remember that integer division results in an integer, with no fractional part.

Application Level

We declare variables of type `int` by using the keyword `int`, followed by the name of the variable, followed by a semicolon. For example

```
int numStudents;
```

You can declare more than one variable of type `int`, by separating the variable names with commas, but we prefer one variable per declaration statement. You can also provide an initial value for an `int` variable by following the name of the variable with an "= value" expression. For example

```
int numStudents = 50;
```

If you do not initialize an `int` variable, the system initializes it to the value 0. However, many compilers refuse to generate Java byte code if they determine that you could be using an uninitialized variable, so it is always a good idea to ensure that your variables are assigned values before they are used in your programs.

Variables of type `int` are handled within a program "by value." This means the variable name represents the location in memory of the value of the variable. This information may seem to belong in a subsection on implementation. However, it does directly affect how we use the variables in our programs, which is the concern of the application level. We treat this topic more completely when we reach Java's composite types, which are not handled by value.

For completeness sake, we should mention what an `int` variable can be used to model: Essentially anything that can be characterized by an integer value in the range stated above. Programs that can be modeled with an integer between negative two billion and positive two billion include the number of students in a class, test grades, city populations, and so forth.

We could repeat the analysis we made above of the `int` type for each of the primitive data types, but the discussion would quickly become redundant. Note that `byte`, `short`, and `long` types are also used to hold integer values, `char` is used to store Unicode characters, `float` and `double` are used to store "real" numbers, and the `boolean` type represents either `true` or `false`. Appendix C contains a table showing, for each primitive type, the kind of value stored by the type, the default value, the number of bits used to implement the type, and the possible range of values.

Let's move on to the composite types.

The Class Type

Primitive data types are the building blocks for composite types. A composite type gathers together a set of component values, sometimes imposing a specific arrangement on them (see Figure 2.6). If the composite type is a built-in type such as an array, the accessing mechanism is provided in the syntax of the language. If the composite type is

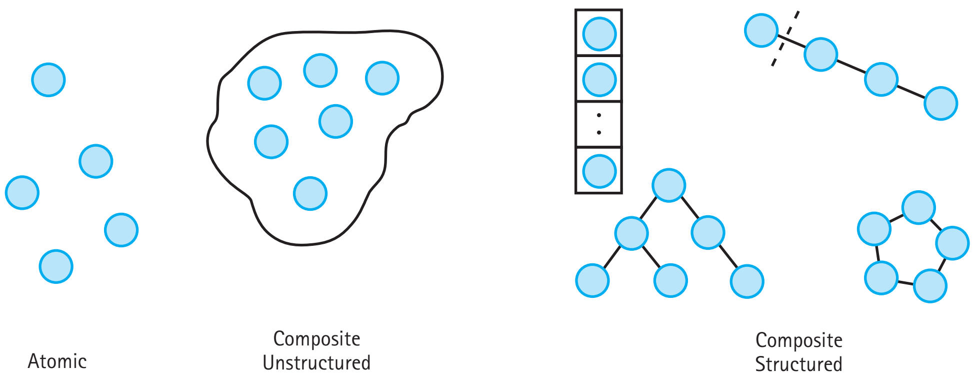


Figure 2.6 *Atomic (simple) and composite data types*

a user-defined type, such as the `Date` class defined in Chapter 1, the accessing mechanism is built into the methods provided with the class.

You are already familiar with the Java *class* construct from your previous courses and from the review in Chapter 1. The class can be a mechanism for creating composite data types. A specific class has a name and is composed of named data fields (class and instance variables—sometimes called *attributes*) and methods. The data elements and methods are also known as members of the class. The members of a class and methods can be accessed individually by name. A class is unstructured because the meaning is not dependent on the ordering of the members within the source code. That is, the order in which the members of the class are listed can be changed without changing the function of the class.

In object-oriented programming, classes are usually defined to hold and hide data and to provide operations on that data. In that case, we say that the programmer has used the *class* construct to build his or her own ADT—and that is the focus of this textbook. However, in this section on built-in types, we use the class strictly to hold data. We do not hide the data and we do not define any methods for our classes. The class variables are public, not private. We use a class strictly to provide unstructured composite data collections. This type of construct has classically been called a *record*. The record is not available in all programming languages. FORTRAN, for instance, historically has not supported records; newer versions may. However, COBOL, a business-oriented language, uses records extensively. C and C++ programmers are able to implement records. Java classes provide the Java programmer with a record mechanism.

Many textbooks that use Java do not present this use of the Java class mechanism, since it is not considered a pure object-oriented construct. We agree that when practicing object-oriented design you should not use classes in the manner presented in this section. However, we present the approach for several reasons:

1. Other languages support the record mechanism, and you may find yourself working with those languages at some time.
2. Using this approach allows us to address the declaration, creation, and use of objects without the added complexity of dealing with class methods.
3. Later, when we discuss using classes to hide data, we can compare the information-hiding approach to the approach described here. The benefits of information hiding might not be as obvious if you hadn't seen any other approach.

In the following discussion, to differentiate the simple use of the *class* construct used here, from its later use to create ADTs, we use the generic term *record* in place of *class*.

Logical Level

A record is a composite data type made up of a finite collection of not necessarily homogeneous elements called fields. Accessing is done directly through a set of named field selectors.

We illustrate the syntax and semantics of the component selector within the context of the following program:

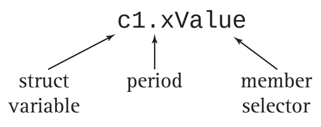
```
public class TestCircle
{
  static class Circle
  {
    int xValue;       // Horizontal position of center
    int yValue;       // Vertical position of center
    float radius;
    boolean solid;    // True means circle filled
  }

  public static void main(String[] args)

  {
    Circle c1 = new Circle();
    c1.xValue = 5;
    c1.yValue = 3;
    c1.radius = 3.5f;
    c1.solid = true;

    System.out.println("c1:   " + c1);
    System.out.println("c1 x: " + c1.xValue);
  }
}
```

The above program declares a record structure called `Circle`. The main method instantiates and initializes the fields of the `Circle` record `c1`, and then prints the record and the `xValue` field of the record to the output. The output looks like this:

```
c1:   TestCircle$Circle[at]111f71
c1 x: 5
```

The `Circle` record variable (the circle object) `c1` is made up of four components (or fields, or instance variables). The first two, `xValue` and `yValue`, are of type `int`. The third, `radius`, is a `float` number. The fourth, `solid`, is a `boolean`. The names of the components make up the set of member selectors.

The syntax of the component selector is the record variable name, followed by a period, followed by the member selector for the component you are interested in:



If this expression is on the left-hand side of an assignment statement, a value is being stored in that member of the record; for example:

```
c1.xValue = 5;
```

If it is used somewhere else, a value is being extracted from that place; for example:

```
output.println("c1 x: " + c1.xValue);
```

Application Level

Records are useful for modeling objects that have a number of characteristics. Records allow us to associate various types of data with each other in the form of a single item. We can refer to the composite item by a single name. We also can refer to the different members of the item by name. You probably have seen many examples of records used in this way to represent items.

We declare and instantiate a record the same way we declare and instantiate any Java object; we use the *new* command:

```
Circle c1 = new Circle();
```

Notice that we did not supply a constructor method in our definition of the `Circle` class in the above program. When using the class as a record mechanism it is not necessary to provide a constructor, since the record components are not hidden and can be initialized directly from the application. Of course, you can provide your own constructor if you like, and that may simplify the use of the record. If no constructor is defined, Java provides a default constructor that initializes the constituent parts of the record to their default values.

In the previous section we discussed how primitive types such as `int`s are handled "by value." This is in contrast to how all nonprimitive types, including records or any objects, are handled. The variable of a primitive type holds the value of the variable, whereas a variable of a nonprimitive type holds a *reference* to the value of the variable. That is, the variable holds the address where the system can find the value of the variable. We say that the nonprimitive types are handled "by reference." This is why, in Java, composite types are known officially as reference types. *Understanding the ramifications of handling variables by reference is very important, whether we are dealing with records, other objects, or arrays.*

The differences between the ways "by value" and "by reference" variables are handled is seen most dramatically in the result of a simple assignment statement. Figure 2.7 shows the result of the assignment of one `int` variable to another `int` variable, and the result of the assignment of one `Circle` object to another `Circle` object. Actual circles represent the `Circle` objects in the figure.

When we assign a variable of a primitive type to another variable of the same type, the latter becomes a copy of the former. But, as you can see from the figure, this is not the case with reference types. When we assign object `c2` to object `c1`, `c1` does *not* become a copy of `c2`. Instead, the reference associated with `c1` becomes a copy of the reference associated with `c2`. This means that both `c1` and `c2` now reference the same object. The feature section below looks at the ramifications of using references from four perspectives: aliases, garbage, comparison, and use as parameters.

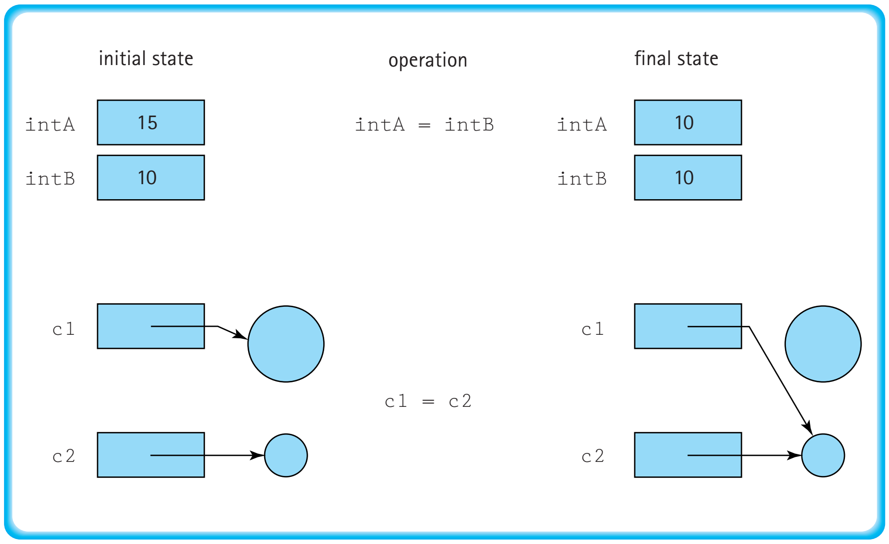


Figure 2.7 *Results of assignment statements*

Java includes a reserved word `null` that indicates an absence of reference. If a reference variable is declared without being assigned an instantiated object, it is automatically initialized to the value `null`. You can also assign `null` to a variable, for example:

```
c1 = null;
```

And you can use `null` in a comparison:

```
if (c1 == null)
  output.println("The Circle is not instantiated");
```

Ramifications of Using References

Aliases

The assignment of one object to another object, as shown in Figure 2.7, results in both object variables referring to the same object. Thus, we have two names for the same object. In this case we say that we have an "alias" of the object. Good programmers avoid aliases because they make programs hard to understand. An object's state can change, even though it appears that the program did not access the object, when the object is accessed through the alias. For example, consider the `IncDate` class that was defined in Chapter 1. If `date1` and `date2` are aliases for the same `IncDate` object, then the code

```
output.println(date1);
date2.increment();
output.println(date1);
```

would print out two different dates, even though at first glance it would appear that it should print out the same date twice. This type of behavior can be very confusing for a maintenance programmer and lead to hours of frustrating testing and debugging.

Garbage

It would be fair to ask in the situation depicted in the lower half of Figure 2.7, what happens to the space being used by the larger circle? After the assignment statement, the program has lost its reference to the large circle, and so it can no longer be accessed. Memory space like this, that has been allocated to a program but that can no longer be accessed by a program, is called garbage. There are other ways that garbage can be created in a Java program. For example, the following code would create 100 objects of class `Circle`; but only one of them can be accessed through `c1` after the loop is finished executing:

```
Circle c1;
for (n = 1; n <= 100; n++)
{
  Circle c1 = new Circle();
  // code to initialize and use c1 goes here
}
```

The other 99 objects cannot be reached by the program. They are garbage.

When an object is unreachable, the Java run time system marks it as garbage., The system regularly performs an operation known as garbage collection, in which it finds unreachable objects and deallocates their storage space, making it once again available in the free pool for the creation of new objects.

This approach, of creating and destroying objects at different points in the application by allocating and deallocating space in the free pool is called dynamic memory management. Without it, the computer would be much more likely to run out of storage space for data.

Garbage The set of currently unreachable objects

Garbage collection The process of finding all unreachable objects and deallocating their storage space

Deallocate To return the storage space for an object to the pool of free memory so that it can be reallocated to new objects

Dynamic memory management The allocation and deallocation of storage space as needed while an application is executing

Comparing Objects

The fact that nonprimitive types are handled by reference impacts the results returned by the == comparison operator. Two variables of a nonprimitive type are considered identical, in terms of

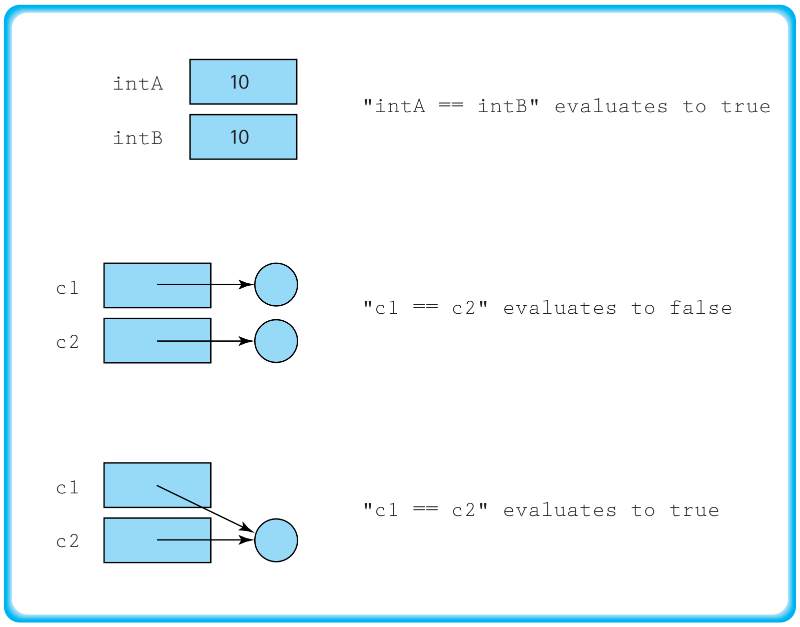

intA
10
intB
10
"intA == intB" evaluates to true
c1
c2
"c1 == c2" evaluates to false
c1
c2
"c1 == c2" evaluates to true

Figure 2.8 *Comparing primitive and nonprimitive variables*

the == operator, only if they are aliases for one another. This makes sense when you consider that the system compares the contents of the two variables. That is, it compares the two references that those variables contain. So even if two variables of type `Circle` have the same `xValue` values, the same `yValue` values, the same `radius` values, and the same `solid` values, they are not considered equal. Figure 2.8 shows the results of using the comparison operator in various situations.

Parameter Passing

When methods are invoked, they are often passed information (arguments) through their parameters. Some programming languages allow the programmer to control whether arguments are passed by value (a copy of the argument's value is used) or by reference (a copy of the argument's memory location is used). Java does not allow such control. Whenever a variable is passed as an argument, the value stored in that variable is copied into the method's corresponding parameter. All Java arguments are passed by value. Therefore, if the variable is of a primitive type, the actual value (`int`, `double`, and so on) is passed to the method; and if it is a reference type, then the reference that it contains is passed to the method.

Notice that passing a reference variable as an argument causes the receiving method to receive an alias of the object that is referenced by the variable. If it uses the alias to make changes to the object, then when the method returns, an access via the variable finds the object in its modified state.

We return many times to these subtle, but important, considerations.

Interfaces

The word *interface* means a common boundary shared by two interacting systems. We use the term in many ways in computer science. For example, the user interface of a program is the part of the program that interacts with the user, and the interface of an object's method is its set of parameters and the return value it provides.

In Java, the word interface has a very specific meaning. In fact, `interface` is a Java keyword. We look briefly at interfaces in this subsection. Throughout the textbook we find places to use the Java *interface* mechanism, at which times we expand our coverage of the topic.

Logical Level

A Java looks very similar to a Java interface. It can include data, that is, variable declarations, and methods. However, all variables declared in an interface must be `final`, `static` variables; in other words, they must be constants. And only the interface descriptions of methods are included; no method bodies or implementations are allowed. Perhaps this is why the language designers decided to call this construct an interface. Methods that are declared without bodies are called abstract methods.

Abstract method A method declared in a class or an interface without a method body

Here is an example of an interface, with one constant, `Pi`, and three abstract methods, `perimeter`, `area`, and `weight`:

```
public interface FigureGeometry
{
  public static final float Pi = 3.14;

  public abstract float perimeter();
  // Returns perimeter of current object

  public abstract float area();
  // Returns area of current object

  public abstract int weight(int scale);
  // Returns weight of current object
}
```

Java provides the keyword `abstract` that we must use when declaring an abstract method in a class. But we do not need to use it when defining the methods in an interface. Its use is redundant, since all methods of an interface must be abstract. We could have omitted it from the above code segment, but chose to show how it may optionally be used, as added documentation, to remind us that the methods are abstract.

At the logical level we look at the domain of values of a data type and the available operations to manipulate them. The domain of values for an interface is made up of classes! Interfaces are used by being "implemented" by classes. For example, a programmer has a `Circle` class implement the `FigureGeometry` interface by using the following line to begin the `Circle` class:

```
public class Circle implements FigureGeometry
```

When a class implements an interface, it receives access to all of the constants defined in the interface. It must provide an implementation, that is, a body, for all the abstract methods declared in the interface. So, the `Circle` class and any other class that implements the `FigureGeometry` interface, would be required to repeat the declarations of the three methods and also provide code for their bodies. Classes that implement an interface are not constrained to only implementing the abstract methods; they can also add data fields and methods of their own.

There are some other issues with interfaces (relationship to abstract classes, use of subinterfaces) that we address, when needed, later in the text.

Application Level

Interfaces are a versatile and powerful programming construct. They can be used in the following ways.

As a contract If we have an abstract view of a class that can have several different implementations, we can capture our abstract view in an interface. Then we can have separate classes implement the interface, with each class providing one of the alternate implementations. This way we are sure that all of the classes provide the same abstraction; we should be able to use them interchangeably in our application programs.

To share constants If there is a set of constant values that we want to use in several different classes, we can define the constants in an interface and have each of the classes implement the interface. Implementing the interface provides access rights to the constants.

To replace multiple inheritance Some languages allow classes to inherit from more than one superclass. This is called *multiple inheritance*. Java does not support multiple inheritance because it can lead to obtuse programs and would greatly complicate the underlying Java environment. However, there are many situations for which we would like to relate the definition of a new class to more than one previously defined class. In these cases, in Java, we use interfaces. A class can extend one superclass, but it can implement many interfaces. So for example, we might see a declaration such as:

```
public class Circle extends Figure implements FigureGeometry, Comparable
```

`Circle` inherits methods and data from the `Figure` class, and must implement any abstract classes defined in the `FigureGeometry` and `Comparable` interfaces. The prime benefit of this is that objects of type `Circle` can be used as if they were objects of type `Figure`, `FigureGeometry`, or `Comparable`.

To provide a generic type mechanism We can design and build ADTs to help us organize data of a specific type. For example, in Chapter 3 we implement an ADT that provides a list of strings. This ADT, and any ADT, would be more reusable if we did not limit it to a specific contained type, in this case, strings. It would be better to have an ADT that lets us manipulate lists of anything. Then, at our discretion, we could use it for lists of letters or lists of integers or whatever. We call such ADTs *generic structures*. In the latter part of Chapter 3 you learn how to use the Java interface construct to provide generic structures.

Arrays

Classes provide programmers a way to collect into one construct several different attributes of an entity and refer to those attributes by name. Many problems, however, have so many components that it is difficult to process them if each one must have a unique name. An array—the last of Java's built-in types—is the data type that allows us to solve problems of this kind. We are sure that you have studied and used arrays in your previous work. Here we revisit arrays, using the terminology and views established in this chapter.

In general terminology, an array differs from a class in three fundamental ways:

1. An array is a homogeneous structure (all components in the structure are of the same data type), whereas classes are heterogeneous structures (their components may be of different types).
2. A component of an array is accessed by its position in the structure, whereas a component of a class is accessed by an identifier (the name).
3. Because array components are accessed by position, an array is a structured composite type.

Logical Level

A one-dimensional array is a structured composite data type made up of a finite, fixed-size collection of ordered homogeneous elements to which there is direct access. *Finite* indicates that there is a last element. *Fixed size* means that the size of the array must be known at compile time, but it doesn't mean that all of the slots in the array must contain meaningful values. *Ordered* means that there is a first element, a second element, and so on. (It is the relative position of the elements that is ordered, not necessarily the values stored there.) Because the elements in an array must all be of the same type, they are physically *homogeneous*; that is, they are all of the same data type. In general, it is desirable for the array elements to be logically homogeneous as well—that is, for all of the elements to have the same purpose. (If we kept a list of numbers in an array of integers, with the length of the list—an integer—kept in the first array slot, the array elements would be physically, but not logically, homogeneous.)

The component selection mechanism of an array is *direct access*, which means we can access any element directly, without first accessing the preceding elements. The desired element is specified using an index, which gives its relative position in the collection.

The semantics (meaning) of the component selector is "Locate the element associated with the index expression in the collection of elements identified by the array name." Suppose, for example, we are using an array of integers, called `numbers`, with 10 elements. The component selector can be used in two ways:

1. It can be used to specify a place into which a value is to be copied, such as

   ```
   numbers[2] = 5;
   ```

2. It can be used to specify a place from which a value is to be retrieved, such as

   ```
   value = numbers[4];
   ```

If the component selector is used on the left-hand side of the assignment statement, it is being used as a transformer: the storage structure is changing. If the component selector is used on the right-hand side of the assignment statement, it is being used as an observer: It returns the value stored in a place in the array without changing it. Declaring an array and accessing individual array elements are operations predefined in nearly all high-level programming languages.

In addition to component selection, there is one other "operation" available for our arrays. In Java, each array that is instantiated has a public instance variable, called `length`, associated with it that contains the number of components in the array. You access the variable using the same syntax you use to invoked object methods: You use the name of the object followed by a period, followed by the name of the instance variable. For the `numbers` example, the expression:

```
numbers.length
```

would have the value 10.

Application Level

A one-dimensional array is the natural structure for the storage of lists of like data elements. Some examples are grocery lists, price lists, lists of phone numbers, and lists of student records. You have probably used one-dimensional arrays in similar ways in some of your programs.

The declaration of a one-dimensional array is similar to the declaration of a simple variable (a variable of a primitive data type), with one exception. You must indicate that it is an array by putting square brackets next to the type:

```
int[] numbers;
```

Alternately, the brackets can go next to the name of the array:

```
int numbers[];
```

We prefer the former approach to declaring arrays, since it is more consistent with the way we declare other variables in Java.

Arrays are handled by reference, just like classes. This means they need to be treated carefully, just like classes, in terms of aliases, comparison, and their use as parameters. And like classes, in addition to being declared, an array must be instantiated. At instantiation you specify how large the array is to be:

```
numbers = new int[10];
```

As with objects, you can both declare and instantiate arrays with a single command:

```
int[] numbers = new int[10];
```

A few more questions you may have about arrays:

- What are the initial values in an array instantiated by using `new`? If the array components are primitive types, they are set to their default value. If the array components are reference types, the components are set to `null`.
- Can you provide initial values for an array? An alternate way to create an array is with an initializer list. For example, the following line of code declares, instantiates, and initializes the array `numbers`:

  ```
  int numbers[] = {5, 32, -23, 57, 1, 0, 27, 13, 32, 32};
  ```

  What happens if we try to execute the statement

  ```
  numbers[n] = value;
  ```

  when `n` is less than 0 or when `n` is greater than 9? The result is that a memory location outside the array would be accessed, which causes an error. This error is called an *out-of-bounds error*. Some languages, C++ for instance, do not check for this error, but Java does. If your program attempts to use an index that is not within the bounds of the array, an `ArrayIndexOutOfBoundsException` is thrown. Rather than trying to catch this error, you should write your code to prevent it. Exceptions are covered in more detail later in this chapter.

Type Hierarchies

In all of our examples of composite types, notably with records and arrays, we have used composite types whose components have been primitive types. We looked at a record, `Circle`, that had four primitive type fields, and an array, `numbers`, of the primitive `int` type. We used this approach to simplify the discussion; it allowed us to concentrate on the structuring mechanism without introducing unnecessary complications. In practice, however, the components of these types can be any Java type or class: built-in primitive types like we have used so far, built-in nonprimitive types, or even user-defined types.

In this subsection we introduce several ways of combining our built-in types and classes into versatile hierarchies.

Aggregate Objects

The instance variables of our objects can themselves be references to objects. In fact, this is a very common approach to the organization of objects in our world. For example, a page object might be part of a book object that is part of a shelf that is part of a library, and so on.

Consider the example from the section entitled The Class Type, of a class modeling a circle that includes variables for horizontal and vertical positions. Instead of these two instance variables, we could have defined a `Point` class to model a point in two-dimensional space, as follows:

```
public class Point
{
  public int xValue;
  public int yValue;
}
```

Then, we could define a new circle class as:

```
public class NewCircle
{
  public Point location;
  public float radius;
  public boolean solid;
}
```

An object of class `NewCircle` has three instance variables, one of which is an object of class `Point`, which in turn has two instance variables. An object, like `NewCircle`, made up of other objects is called an aggregate object. We call the relationship between the classes `NewCircle` and `Point` a "has a" relationship, as in "a `NewCircle` object *has a* `Point` object" as an instance variable. The *has a* relationship is depicted in UML with a diamond on the composite end of a link between the two classes, as shown in Figure 2.9.

Aggregate object An object whose class definition includes variables that are themselves references to classes.

When we instantiate and initialize an object of type `NewCircle`, we must remember to also instantiate the composite `Point` object. For example, to create a solid circle at position <5, 7> with a radius of 2.5, we would have to code:

```
NewCircle myNewCircle = new NewCircle();
myNewCircle.location = new Point();
myNewCircle.location.xValue = 5;
myNewCircle.location.yValue = 3;
myNewCircle.radius = 2.5f;
myNewCircle.solid = true;
```

Although this is a syntactically correct approach to structuring data, the use of composite objects in this fashion quickly becomes tedious for the application programmer. It is

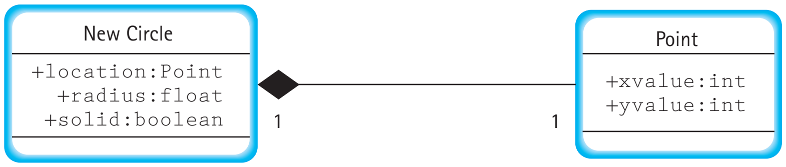


Figure 2.9 *UML diagram showing has a relationship*

much easier if we define methods, such as a constructor method, to access and manipulate our objects. That is the approach we take below in the section on user-defined types, when we move from using classes as records to using classes to create true ADTs.

Arrays of Objects

Although arrays with atomic components are very common, many applications require a collection of composite objects. For example, a business may need a list of parts records or a teacher may need a list of students in a class. Arrays are ideal for these applications. We simply define an array whose components are objects.

Let's define an array of `NewCircle` objects. Declaring and creating the array of objects is exactly like declaring and creating an array in which the components are atomic types:

```
NewCircle[] allCircles = new NewCircle[10];
```

`allCircles` is an array that can hold ten references to `NewCircle` objects. What are the locations and radii of the circles? We don't know yet. The array of circles has been instantiated, but the `NewCircle` objects themselves have not. Another way of saying this is that `allCircles` is an array of references to `NewCircle` objects, which are set to `null` when the array is instantiated. The objects must be instantiated separately. The following code segment initializes the first and second circles. It assumes that a `NewCircle` object `myNewCircle` has been instantiated and initialized as described in the preceding section, Aggregate Objects.

```
NewCircle[] allCircles = new NewCircle[10];
allCircles[0] = new NewCircle();
allCircles[0] = myNewCircle;
allCircles[1] = new NewCircle();
allCircles[1].location = new Point();
allCircles[1].location.xValue = 6;
allCircles[1].location.yValue = 6;
allCircles[1].radius = 1.3f;
allCircles[1].solid = false;
```

Normally an array like this would be initialized using a *for* loop and a constructor method, but we used the above approach so that we could demonstrate several of the subtleties of the construct. Figure 2.10 shows what the array looks like with values in it.

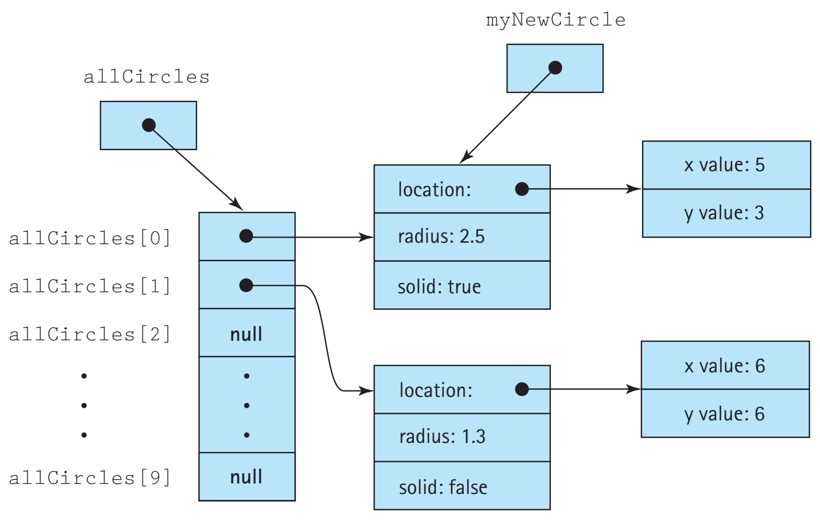


Figure 2.10 *The `allCircles` array*

Study the code above and Figure 2.10. In particular, notice how we must instantiate each element in the array with the new command. Also, notice that `myNewCircle` and `allCircles[0]` are aliases.

Keep in mind that an array name with no brackets is the array object. An array name with brackets is a component. The component can be manipulated just like any other variable of that type. The following table demonstrates these relationships:

Expression	Class/Type
`allCircles`	An array
`allCircles[1]`	A `NewCircle`
`allCircles[1].location`	A `Point`
`allCircles[1].location.xValue`	An `int`

Two-Dimensional Arrays

A one-dimensional array is used to represent items in a list or a sequence of values. A two-dimensional array is used to represent items in a table with rows and columns, provided each item in the table is of the same type or class. A component in a two-dimensional array is accessed by specifying the row and column indexes of the item in the array. This is a familiar task. For example, if you want to find a street on a map, you look up the street name on the back of the map to find the coordinates of the street, usually a number and a letter. The number specifies a row, and the letter specifies a column. You find the street where the row and column meet.

An inline figure of a map demonstrating the previous to come

Figure 2.11 shows a two-dimensional array with 100 rows and 9 columns. The rows are accessed by an integer ranging from 0 through 99; the columns are accessed by an integer ranging from 0 through 8. Each component is accessed by a row–column pair—for example, `[0][5]`.

A two-dimensional array variable is declared in exactly the same way as a one-dimensional array variable, except that there are two pairs of brackets. A two-dimensional array object is instantiated in exactly the same way, except that sizes must be specified for two dimensions.

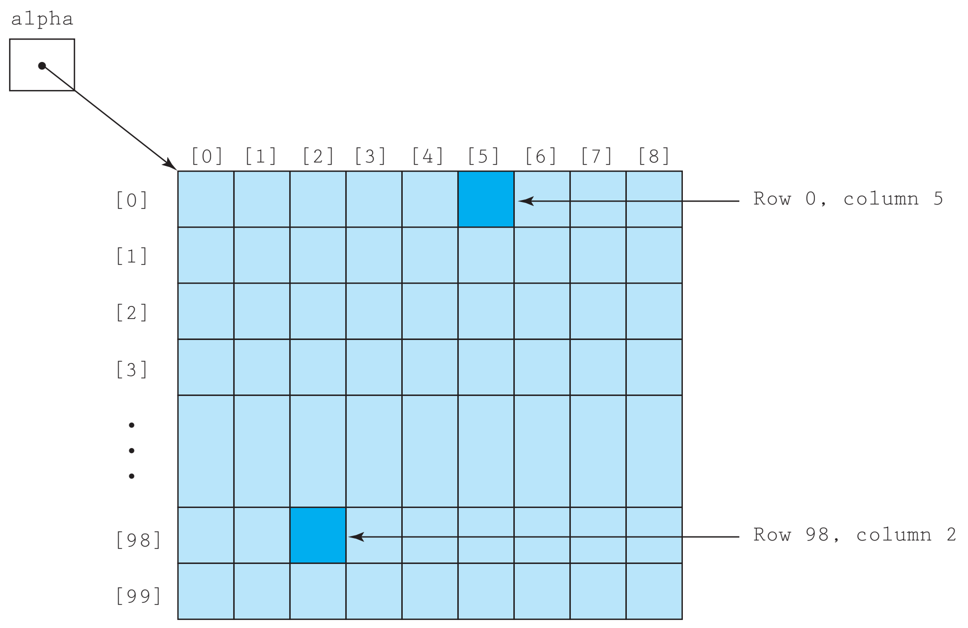


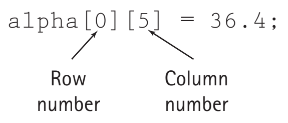
Figure 2.11 `alpha` *array*

The following code fragment would create the array shown in Figure 2.11, where the data in the table are floating-point numbers.

```
double[][] alpha;
alpha = new double[100][9];
```

The first dimension specifies the number of rows, and the second dimension specifies the number of columns.

To access an individual component of the `alpha` array, two expressions (one for each dimension) are used to specify its position. We place each expression in its own pair of brackets next to the name of the array:



```
alpha[0][5] = 36.4;
```

Note that `alpha.length` would give the number of rows in the array. To obtain the number of columns in a row of an array, we access the `length` attribute for the specific row. For example, the statement

```
rowLength = alpha[30].length;
```

stores the length of row 30 of the array `alpha`, which is 9, into the `int` variable `rowLength`.

The moral here is that in Java each row of a two-dimensional array is itself a one-dimensional array. Many programming languages directly support two-dimensional arrays; Java doesn't. In Java, a two-dimensional array is an array of references to array objects. Because of the way that Java handles two-dimensional arrays, the drawing in Figure 2.11 is not quite accurate. Figure 2.12 shows how Java actually implements the array `alpha`. From the Java programmer's perspective, however, the two views are synonymous in the majority of applications.

Multilevel Hierarchies

We have just looked at various ways of combining Java's built-in type mechanisms to create composite objects, arrays of objects, and two-dimensional arrays. We do not have to stop there. We can continue along these lines to create whatever sort of structure best matches our data. Classes can have arrays as variables, aggregate objects can be made from other aggregate objects, and we can create arrays of three, four, or more dimensions.

Consider, for example, how a programmer might structure data that represents students for a professor's grading program. This professor grades each test with both a numerical grade and a letter grade. Therefore, the programmer decides to represent a test as a record, called `test`, with two fields: `score` of type `int` and `grade` of type `char`. Each student takes a sequence of tests—these are represented by an array of `test` called `marks`. A student also has a name and an attendance record. So a student

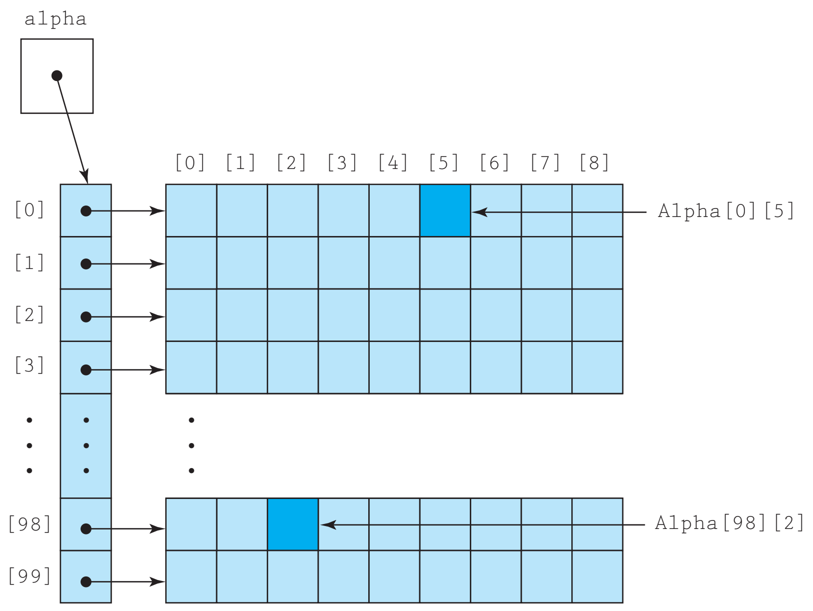


Figure 2.12 *Java implementation of the* `alpha` *array*

could be represented by a record with three fields: `name` of type `String`, `marks` of type array of `test`, and `attendance` of type `int`. Since the professor has many students in a course, the programmer creates another array, called `course`, that is an array of `student`. Wow! We have an array of records of three fields, one of which is itself an array of records of two fields. See Figure 2.13 for a logical view of this multilevel structure.

The idea is to use the built-in typing mechanisms to model the real world structure of the data. This makes it easier for us to organize our processing of the data.

In the next section we look at how we can extend Java's built-in types by encapsulating composite types with programmer-defined methods, to simplify their access and manipulation. When we do this we are creating our own ADTs.

2.3 Class-Based Types

The class construct sits at the center of the Java programming world. In the previous section, you learned how the Java class could be used to structure data into records. As we stated then, that is not a proper use of the class construct when practicing object-oriented design. In this section, you learn how to use classes to implement ADTs. This is the correct way to use the class construct.

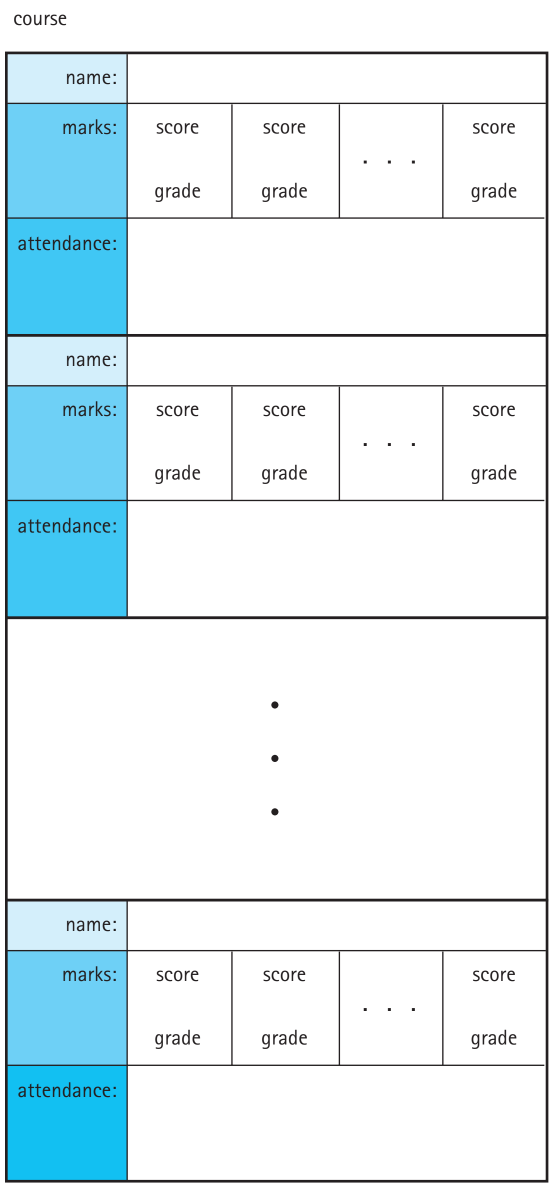


Figure 2.13 *Logical view of array of student records*

Meaning of "Type"

The Java language specification reserves the word *type* to mean those abstract data types (ADTs) that are built into the language, such as `int`, `double`, `char`, `array`, and `class`. Every ADT that we design and implement as a class in Java is considered by the language to be a member of the same type, which is the Java `class` type.

More generally, the word *type* is often used to refer to an ADT and its implementation in whatever programming language is being used. Thus, there is a potential for some minor confusion with respect to Java's use of the term and the use of the term in general. Wherever we use the word *type*, and the context of the usage does not clarify the meaning, we modify the term to provide clarification. Thus, we may use "Java type," "built-in type," or "primitive type" to indicate that we are using the term in the strict Java sense. Elsewhere, we use the term in its more general sense, for example, to refer to the implementation of a programmer defined ADT. Thus, we may refer to the `Date` type or the `Circle` type.

Using Classes in Our Programs

Once a programmer has defined a class, objects of the class type can be declared, instantiated, and used in many other classes. For purposes of this discussion, we call the class being used the *tool class*, and the class using it the *client class*. The client class could be an *application*, that is, a class with a `main` method that would be executed when we invoke the Java interpreter. For the client class to use the tool class, the definition of the tool class must be visible to the Java compiler/interpreter, when the client class is compiled or interpreted. There are several ways you can ensure this:

Inner class   A class defined as a member of another class

Package   A set of related classes, grouped together to provide efficient access and use

1. Insert the tool class code directly into the client class file. In this case, we call the tool class an inner class. There are some situations, especially with respect to dynamic event handling, where inner classes provide an elegant solution to difficult problems. Usually, however, their use is too restrictive.
2. Computer systems that support Java have a well-defined set of subdirectories to search when a Java class is needed. Usually an environment variable called `ClassPath` defines this set of subdirectories. Place the tool class file in one of these subdirectories.
3. The Java package construct is used by programmers to collect into a single unit a group of related classes. Put the tool class in a package and import the package into the client that uses it. Note that the compiler/interpreter must be able to find the package, so it must be located in an appropriate subdirectory on the `ClassPath`. The feature section below describes the details of using Java packages.

Java Packages

Java lets us group related classes together into a unit called a package. Packages provide several advantages:

- They let us organize our files.
- They can be compiled separately and imported into our programs.
- They make it easier for programs to use common class files.
- They help us avoid naming conflicts (two classes can have the same name if they are in different packages).

Package Syntax

The syntax for a package is extremely simple. All we have to do is to specify the package name at the start of the file containing the class. The first noncomment nonblank line of the file must contain the keyword `package` followed by an identifier and a semicolon. By convention, Java programmers start a package identifier with a lowercase letter to distinguish package names from class names:

```
package someName;
```

After this we can write import declarations, to make the contents of other packages available to the classes inside the package we are defining, and then one or more declarations of classes. Java calls this file a *compilation unit*. The classes defined in the file are members of the package. Note that the imported classes are not members of the package.

Although we can declare multiple classes in a compilation unit, only one of them can be declared public. The others are hidden from the world outside the package. (We investigate visibility topics later in this section.) If a compilation unit can hold at most one public class, how do we create packages with multiple public classes? We have to use multiple compilation units, as we describe next.

Packages with Multiple Compilation Units

Each Java compilation unit is stored in its own file. The Java system identifies the file using a combination of the package name and the name of the public class in the compilation unit. Java restricts us to having a single public class in a file so that it can use file names to locate all public classes. Thus, a package with multiple public classes must be implemented with multiple compilation units, each in a separate file.

Using multiple compilation units has the further advantage that it provides us with more flexibility in developing the classes of a package. Team programming projects would be very cumbersome if Java made multiple programmers share a single package file.

We split a package among multiple files simply by placing its members into separate compilation units with the same package name. For example, we can create one file containing the following code (the ... between the braces represents the code for each class):

```
package someName;
public class One{ ...  }
class Two{ ...  }
```

and a second file containing:

```
package someName;
class Three{ ...  }
public class Four{ ...  }
```

with the result that the package `someName` contains four classes. Two of the classes, `One` and `Four` are public, and so are available to be imported by application code. The two file names must match the two public class names; thus the files must be named `One.java` and `Four.java`.

Many programmers simply place every class in its own compilation unit. Others gather the nonpublic classes into one unit, separate from the public classes. How you organize your packages is up to you, but you should be consistent to make it easy to find a specific member of a package among all of its files.

How does the Java compiler manage to find these pieces and put them together? The answer is that it requires that all compilation unit files for a package be kept in a single directory or folder that matches the name of the package. For our preceding example, a Java system would store the source code in files called `One.java` and `Four.java`, both in a directory called `someName`.

The import Statement

In order to access the contents of a package from within a program, you must import it into your program. You can use either of the following forms of import statements:

```
import packagename.*;
import packagename.Classname;
```

An import declaration begins with the keyword `import`, the name of a package and a dot (period). Following the period you can either write the name of a class in the package, or an asterisk (`*`). The declaration ends with a semicolon. If you know that you want to use exactly one class in a particular package, then you can simply give its name in the import declaration. More often, however, you want to use more than one of the classes in a package, and the asterisk is a shorthand notation to the compiler that says, "Import whatever classes from this package that this program uses."

Packages and Subdirectories

Many computer platforms use a hierarchical file system. The Java package rules are defined to work seamlessly with such systems. Java package names may also be hierarchical; they may contain periods separating different parts of the name, for example, `ch03.stringLists`. In such a case, the package files must be placed underneath a set of subdirectories that match the separate parts of the package name. Following the same example, the package files should be placed in a directory named `stringLists` that is a subdirectory of a directory named `ch03`. You can import the entire package into your program with the following statement:

```
import ch03.stringLists.*;
```

As long as the directory that contains the `ch03` directory is on the `ClassPath` of your system, the compiler will be able to find the package you requested. The compiler automatically looks in all the directories listed in the `ClassPath`. In this case it will actually look in the `ClassPath` directories for a subdirectory named `ch03` that contains a subdirectory named `stringLists`, and upon finding such a subdirectory, it will import all of the members of the `ch03.stringLists` package that it finds there.

Many of the files created to support this textbook are organized into packages. They are organized exactly as described above and can be found on our web site. All the files are found in a directory named `bookFiles`. It contains a separate subdirectory for each chapter of the book: `ch01`, `ch02`, ..., `ch10`. Where packages are used, you will find the corresponding subdirectories underneath the chapter subdirectories. For example, the `ch03` subdirectory does indeed contain a subdirectory named `stringLists` that contains four files that define Java classes related to a string list ADT. Each of the class files begins with the statement

```
package ch03.stringLists;
```

Thus, they are all in the `ch03.stringLists` package. If you write a program that needs to use these files you simply need to import the package into your program and make sure the parent directory of the `ch03` directory, i.e., the `bookFiles` directory, is included in your computer's `ClassPath`.

We suggest that you copy the entire `bookFiles` directory to your computer's hard drive, ensuring easy access to all the book's files and maintaining the crucial subdirectory structure required by the packages. Also, make sure you extend your computer's `ClassPath` to include your new `bookFiles` dirctory. See the Preface for more information.

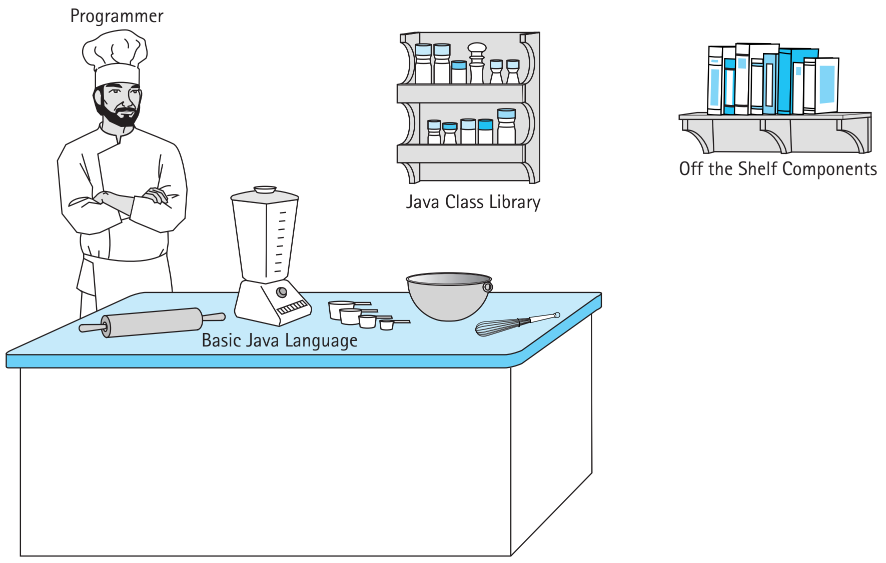
Sources for Classes

Java programs are built using a combination of the basic language and pre-existing classes. In effect, the pre-existing classes act as extensions to the basic language; this extended Java language is large, complex, robust, powerful and ever changing. Java programmers should never stop learning about the nuances of the extended language—an exciting prospect for those who like an intellectual challenge.

When designing a Java-based system to solve a problem, we first determine what classes are needed. Next we determine if any of these classes already exist; and if not, we try to discover classes that do exist that can be used to build the needed classes. Additionally, we often create our own classes, "helper" classes that are used to build the needed classes.

Where do the classes come from? There are three sources:

1. The Java Class Library—The Java language is bundled with a class library that includes hundreds of useful classes. We look at the library in a subsection below.
2. Build your own—Suppose you determine that a certain class would be useful to aid in solving your programming problem, but the class does not exist. Therefore, you create the needed class, possibly using pre-existing classes in the process. The new class becomes part of the extended language, and can be used on future projects. We look at how to build our own classes in a later section, and throughout the rest of the textbook.
3. Off the shelf—Software components, such as classes or packages of classes, which are obtained from third party sources, are called off-the-shelf components. When they are bought, we call them "commercial off-the-shelf" components, or COTS. Java components can be bought from software shops, or even found free on the web. When you obtain software, or anything else, from the web for your own use, you should make sure you are not violating a copyright. You also need to use care in determining that free components work correctly and do not contain viruses or other code that could cause problems.



As our study of data structures, abstract data types, and Java continues, we sometimes investigate how to build a class that mirrors the functionality of a pre-existing class, for example a class in the Java Class Library. There are two reasons we may do this: convenience and computer science content. It may be that the class provides a good, convenient example of a language construct or programming approach—for example, our use of the `Date` example throughout the first two chapters—even though the library provides ways of creating and using Date objects. Alternately, it may be that the study of the class is crucial to the content of this textbook—classic data structures. For example, in Chapter 4 we study how to implement a Stack ADT, even though a Stack ADT is provided in the library. Understanding the possible implementations of stacks, and the ramifications of implementation choices, is considered crucial for serious students of computing.

There are other reasons that a programmer might want to create his or her own class that mimic the functionality of a library class: simplicity and control. The Java developers designed library classes to provide robust functionality. Robustness is an important quality for library classes. Sometimes, however, the robustness of a class equates to complexity or inefficiency. Additionally, you must remember that the Java Class Library is not a static construct. The changes to the library are usually in the form of enhancements, but there have also been cases where features of the library have been deprecated. A deprecated feature is one that may not be supported in future versions of the library. Deprecation acts as a warning to programmers—use this construct at your own risk; it works now, but might not work later!

Deprecated A Java construct is deprecated when the Java developers have decided that the construct might not be supported in future releases of the language; use of deprecated features is discouraged.

Consider the history of dates in Java. In the original public release of Java, JDK 1.0 in 1995, the library included a `Date` class that allowed a programmer to represent dates and times. This class could be used to specify and manipulate a date/time in two forms: the number of milliseconds between the date/time and January 1, 1970, midnight, or by using discrete attributes of the date/time, such as month, day, year, hour. As you can imagine, for most purposes the latter form was easier to use. Nevertheless, the latter form of use was deprecated with the release of JDK 1.1 in 1996 because it did not support Java's goal of internationalization. Although many countries use the Gregorian calendar that the `Date` class is based on, there are other calendars in use around the world, for example the Chinese calendar.

The `Calendar` class was introduced in Java 1.1 to support all kinds of calendars. It provided features to replace the deprecated functionality of `Date`. The `Calendar` class is well designed and very useful; but it is not trivial to use. The `Calendar` class cannot be directly instantiated; programmers must use its `getInstance` method to obtain a local instance of a calendar, and instantiate this local instance as a subclass of `Calendar`. To use the "standard" solar calendar, with years numbered from the birth of Christ, a programmer would use the `GregorianCalendar` subclass of `Calendar`. The `GregorianCalendar` class exports 28 methods and defines 42 constants for use with the methods of the class.

Considering all of this, it is no wonder that programmers who need a simple date class—perhaps one that allows a month, day, and year to be passed to the constructor, provides three simple observer methods, and provides methods to increment a date and compare two dates—might decide to implement their own class.

The Java Class Library

Programming with an object-oriented language depends heavily on the use of classes from the language's standard library. The Java standard class library includes over 70 packages and subpackages, with hundreds of classes and interfaces, and thousands of exported methods and constants. It is not our goal in this textbook to teach the standard library. However, we do encourage the reader to begin to learn about the library, and to continue studying the library.

Sun Microsystems, Inc., the developers of Java, maintains a public web site[1] where they have provided extensive documentation about the class libraries. The list below briefly describes some of the prominent packages and subpackages found on the Sun site. Visit their site for more information. In this subsection, we review some of the most important classes, especially with respect to the goals of this textbook. Throughout the text, as we reach places where we need to use library constructs in a new way, we expand on this coverage.

Some Important Library Packages

`java.awt`	(Abstract Windowing Toolkit) Contains tools for creating user interfaces, graphics, and images.
`java.awt.event`	Provides interfaces and classes for handling the different types of events created by AWT components.
`java.io`	System input and output through data streams, serialization, and the file system.
`java.lang`	Provides basic classes for use in creating Java programs.
`java.math`	Provides classes for performing mathematical operations.
`java.text`	Provides classes and interfaces for handling text, dates, numbers, and messages.
`java.util`	Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).
`java.util.jar`	Provides classes for reading and writing the JAR (Java ARchive) file format, which is based on the standard ZIP file format.

Some Useful General Library Classes

As we are studying data structures with the Java language, we use various utility classes that are available in the Java library. In this subsection, we introduce some of these classes.

[1] http://java.sun.com/j2se/1.3/docs/api/index.html

The System class The System class is part of the java.lang package. All of the System class's methods and variables are class methods and variables. They are defined to be static—they are unique to the class, rather than to objects of the class. We simply use the System class methods and variables directly in our programs; we access them through the class name rather than through the name of an instantiated object. For example, in the TestCircle program listed above in Section 2.2, we used the System variable out as a destination for our output:

```
System.out.println("c1:   " + c1);
```

We can also use the System class to obtain current system properties, such as the amount of available memory.

The Random class The Random class is part of the util package. Programmers use it to generate a list of random numbers. Random numbers are useful when creating simulations, or models of real-world situations, with our programs. We use the Random class in Chapter 10 to generate lists of random numbers for sorting.

The DecimalFormat class The DecimalFormat class is part of the java.text package. To use it a programmer calls one of its constructors to define a format pattern. Then this instance can be used to format numbers for output. In Chapter 4 we use the DecimalFormat class to format numbers so that output columns line up nicely.

The Throwable and Exception Classes We introduced the concept of exceptions in Chapter 1. Recall that an exception is associated with an unusual, often unpredictable event, detectable by software or hardware, that requires special processing. One system unit raises or throws an exception, and another unit catches the exception and processes it. Processing an exception is also called handling the exception.

Throw an exception Interrupt the normal processing of a program to raise an exception that must be handled or rethrown by the surrounding block of code

Catch an exception Code that is executed to handle a thrown exception is said to catch the exception

When a part of a Java system determines that an exception has occurred, it "announces" the exception using the Java throw statement. This can occur within the Java interpreter, within a library method, or within our own code. (We discuss how to define, and how to determine when to throw our own exceptions, in the section below about building our own ADTs. For now, we look at predefined exceptions.) When an exception is thrown, it must either be caught and handled by the surrounding block of code, or thrown again to the next outer block of code. If an exception is thrown all the way out of a method, it propagates to the calling method. An exception that is continually thrown until it makes it all the way up the chain of calling methods and is thrown by the main method to the Java interpreter, is handled by the interpreter: An error message is printed along with some system information (a system stack trace) and the program exits.

All exceptions in Java are subclasses of the `java.lang.Throwable` class. Only objects or instances of this class (or subclasses of this class) are thrown within a Java system. The `Throwable` class provides several methods related to exceptions, notably the `getMessage` method that returns the error message associated with the `Throwable` object, and the `printStackTrace` method that prints a trace of the sequence of system calls that led to the `throw` statement.

The `Throwable` class has two standard subclasses: `java.lang.Error` and `java.lang.Exception`. The former is used for defining catastrophic exceptional situations that are best handled by simple program termination. We are concerned with the latter subclass, the `Exception` class, which is used for defining exceptional situations from which we may be able to recover. The `Exception` class extends the `Throwable` class with two methods, both constructors:

Method Name	Parameter Type	Returns	Operation Performed
`Exception`	(none)	`Exception`	Constructs an exception with no specified message.
`Exception`	`String`	`Exception`	Constructs an exception with the specified message.

Exceptions are defined by extending the `Exception` class. If you look at the Java library information you see dozens of predefined subclasses of the `Exception` class, each of which might also have many subclasses. The result is that there are hundreds of exceptions defined in the Java library.

Let's look at a few examples of throwing and handling predefined exceptions.

Review the `IncDate` test driver program, `IDTestDriver`, developed at the end of Chapter 1. Notice the heading of the program's `main` method:

```
public static void main(String[] args) throws IOException
```

As you can see, in the declaration of the `main` method we have told the system that this method can throw the predefined `IOException` exception. Where would `IOException` be raised in the program? This program uses the `readLine` method defined in the `BufferedReader` class. A quick look at the documentation of the `readLine` method shows that it throws an `IOException` "if an I/O error occurs." Since it is possible for that exception to be thrown by the `readLine` method, the surrounding code (the `main` method), must either catch and handle the exception, or throw the exception. In this case, we have decided to just throw the exception out to the interpreter, which would terminate the program. Note that this is a perfectly valid option; in fact, if there is not enough information to properly handle an exception at one level of a program, the best approach is to throw the exception out to the next level, where it may be handled more properly.

If we decided to handle the exception within the test driver program itself we would surround the section of the program where the exception can be raised with a *try-catch* statement. For example:

```
try
{
  month = Integer.parseInt(dataFile.readLine());
  day = Integer.parseInt(dataFile.readLine());
  year = Integer.parseInt(dataFile.readLine());
}
catch (IOException readExcp)
{
  outFile.println("There was trouble reading in month, day, year.");
  outFile.println("Exception: " + readExcp.getMessage());
  System.exit();
}
```

Now, if the `IOException` exception is raised by any of the `readLine` methods within the *try* block, it is handled by the code in the *catch* block. Notice the rather unusual syntax of the *catch* statement:

```
catch (ExceptionClassName varName)
```

If the exception class referenced in the *catch* statement is thrown by any of the statements in the *try* block, the *catch* statement defines a new object of that exception class, and that object becomes equated with the thrown exception. So in this example, the variable `readExcp` represents the exception that is caught. Because `readExp` is an instantiation of a subclass of `Throwable`, it has a `getMessage` method. In the *catch* block we can use `readExcp.getMessage()` to print the message associated with the exception.

In this example, we are handling the exception by printing our own brief error message, then printing the error message associated with the exception, and then terminating the program. Realistically, there is not much more we can do in this situation. Since this is essentially the same action the interpreter does for us anyway, it is probably better to just throw the exception. Besides, as we explained when we developed the test driver program, it is not important that the test driver be robust, since we are only using it to test another class; the test driver is not delivered to a customer.

There are some other nuances involved with handling predefined exceptions—for example, the use of Java's `finally` clause, and the option of handling and still rethrowing the exception. It could quickly become confusing if we tried to cover all of these topics at once, so we put off a discussion of other options until we reach an example that requires their use.

One last note about predefined exceptions. The `java.lang.RunTimeException` class is treated uniquely by the Java environment. Exceptions of this class are thrown during the normal operation of the Java Virtual Machine when a standard run-time program error occurs. Examples of run-time errors are division by zero and array-index-out-of-bounds situations. Since run-time exceptions can happen in virtually any method or segment of code, we are not required to explicitly handle these exceptions. If it were required, our programs would become unreadable because of all the necessary *try*, *catch*, and *throw* statements. These exceptions are classified as unchecked exceptions.

Wrappers There are situations where a Java programmer wants to use a variable of class `Object` to reference many different kinds of objects. This is possible, since `Object` is a superclass of all other classes. This feature provides a powerful tool; however, it does suffer from one limitation—the variable of class `Object` cannot reference primitive type values, since the primitive types are not objects. To resolve this deficiency, the Java Class Library includes a wrapper class for each of the primitive types. To store a primitive value in the `Object` variable, the programmer first "wraps" it in the appropriate wrapper class. These classes are known as wrapper classes since they literally wrap a primitive valued variable in an object's structure, as shown in Figure 2.14. The following table lists the primitive types and the built-in class to which each corresponds.

Unchecked exception An exception of the `RunTimeException` class, it does not have to be explicitly handled by the method within which it might be raised.

Wrapper class A Java class that wraps a primitive type, letting it be manipulated as an object, and providing some useful utility methods related to the type.

Primitive Type	Object Type
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`double`	`Double`
`float`	`Float`
`int`	`Integer`
`long`	`Long`
`short`	`Short`

As you can see, the general rule is that the class name is the same as the name of the built-in type, except that its first letter is capitalized. The two cases that differ are that the class corresponding to `int` is called `Integer` and the class corresponding to `char` is `Character`.

The wrapper classes are a part of the `java.lang` package.

In addition to allowing us to treat a primitive type as an object, the wrapper classes provide many useful conversion and utility methods related to their associated primitive type. For example, we used the `Integer` wrapper class method `parseInt` in the `IDTestDriver` program in Chapter 1:

```
month = Integer.parseInt(dataFile.readLine());
day = Integer.parseInt(dataFile.readLine());
year = Integer.parseInt(dataFile.readLine());
```

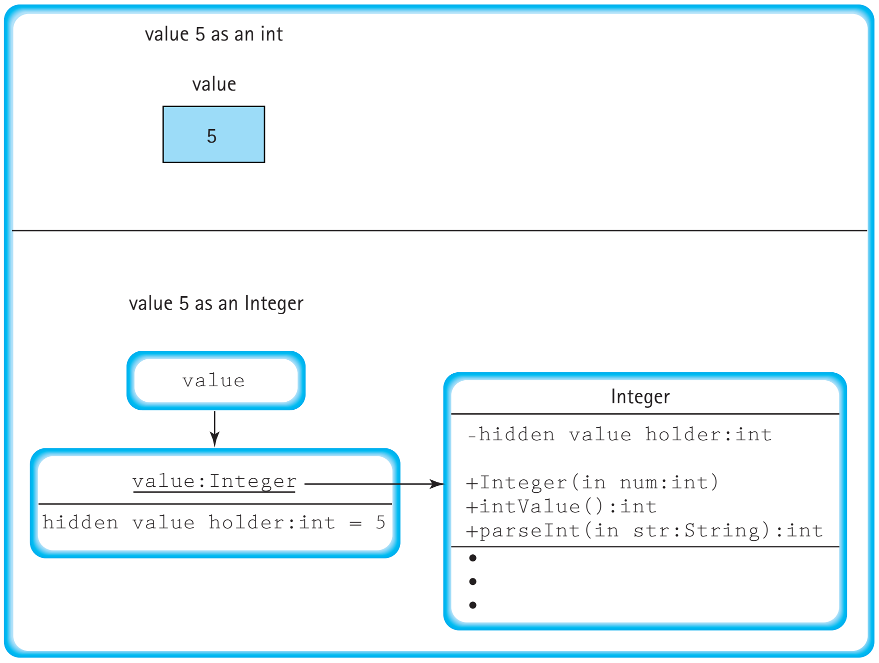


Figure 2.14 *The integer value 5 as an* `int` *variable* `value`*, and an* `Integer` *object* `value`

The `parseInt` method accepts a string as a parameter and transforms it into the corresponding integer. For example if it is passed the string "`27`" it returns the `int` value 27. Since the `BufferedReader datafile` we defined in the `IDTestDriver` program returns all input in the form of strings, the `parseInt` method allows us to transform the input into a more useful form.

Some Class Library ADTs

In addition to the utility classes just described, the Java Class Libraries include some ADTs that are pertinent to your study. A class provides an ADT if its basic purpose is to allow the programmer to store data in an abstract structure, hiding the implementation of the structure from the programmer but allowing the programmer to access the data through various exported methods.

In some sense the wrapper classes described at the end of the previous section provide ADTs—but the main way we use those classes is to access their general utility class methods, such as the `parseInt` method of `Integer`. Such methods are not really acting on an object; `parseInt` accepts a string parameter and returns a primitive `int`. It is invoked through the `Integer` class and not through a specific object of the class.

In this section we look at the Java Class Library ADTs `String` and `ArrayList` from the logical-level and application-level viewpoints. For array lists we also take a peek at the implementation level, since it is instructive to do so.

Strings The Java `String` class is part of the `java.lang` package. Remember that this package provides classes that are fundamental to the design of the Java programming language. In fact, this package is automatically imported into every Java program.

Strings are a fundamental building block for many programs. We have already been using them extensively in this textbook, for input and output to our programs and to indicate file names within our test drivers. We assume you have some experience using strings in your previous programming. Nevertheless, we provide a brief review of the Java `String` class here.

Logical Level The first thing we want to remind you about strings is that they are immutable. If an object doesn't have any methods that can change its state, it is immutable. A string is an immutable object; we can only retrieve its contents. There is no way to change a string object. We can only assign a new reference to a `String` variable. In other words, a `String` variable references a `String` object. Once created, that object cannot be changed; however, we can change the `String` variable so that it references a different `String` object.

Immutable object An object whose state cannot be changed once it is created

The Java `String` class provides operations for joining strings, copying portions of strings, changing the case of letters in strings, converting numbers to strings, and converting strings to numbers. Their use is straightforward and we leave it to you to review them. Notice that, due to the immutability of strings, any operation that appears to change a string, for example the `toUpperCase` method, actually returns a new `String` object rather than changing the current string. For example, if the string `objectnameB` has an associated value "Adam", the statement

```
nameA = nameB.toUpperCase();
```

creates a new `String` object with value "ADAM", assigns its reference to the `nameA` string variable, but leaves the `nameB` string variable and object unchanged.

You cannot compare strings using the relational operators. Syntactically, Java lets you write the comparisons for equality (==) and inequality (!=) between values of class `String`, but the comparison that this represents is not what you typically want. Since `String` is a reference type, when you compare two strings this way, Java checks to see that they have the same address. It does not check to see whether they contain the same sequence of characters.

Rather than using the relational operators, we compare strings with a set of value-returning instance methods that Java supplies as part of the `String` class. Because they are instance methods, the method name is written following a `String` object, separated by a dot. The string that the method name is appended to is one of the strings in the comparison, and the string in the parameter list is the other. The two most useful comparison methods are summarized in the following table.

Method Name	Parameter Type	Returns	Operation Performed
`equals`	`String`	`boolean`	Tests for equality of string contents.
`compareTo`	`String`	`int`	Returns 0 if equal, a positive integer if the string in the parameter comes before the string associated with the method, and a negative integer if the parameter comes after it.

For example, if `lastName` is a `String` variable, you can use

```
lastName.equals("Olson")
```

to test whether `lastName` equals "Olson."

Application Level Since the use of strings in programs is so prevalent, the Java language provides a few shortcuts for using the `String` class that differentiate it from all the other classes in the Java library. We saw one of these in Chapter 1 when we noted how a `toString` method is automatically applied to an object that is being used as a string. Let's look at two more special conventions for strings, string literals, and the concatenation operator.

String Literals Just as Java provides literals for all of its primitive types (for example `-154` is a literal of type `int` and `true` is a literal of type `boolean`), it provides a literal string mechanism. To indicate a literal string, you simply enclose the sequence of characters between double quotation marks. For example:

```
"this is a literal string"
```

A literal string actually represents an object of class `String`. Enclosing a sequence of characters in the double quotation marks is equivalent to declaring and instantiating a new `String` object. Therefore, the following two code sequences are equivalent:

```
String myString;
myString = new String ("The Cat in the Hat");
---------------------------------------------
String myString = "The Cat in the Hat";
```

The String Concatenation Operator The `String` class exports a method `concat` that allows a programmer to concatenate two strings together to form a third string. However, this operation is so prevalent in Java programs that the language designers provide us with a shortcut, the + string operation. The result of concatenating two strings is a new string containing the characters from both strings. For example, given the statements

```
String first = "The Cat in the Hat";
String second = "Comes Back";
String third = first + second;
output.println(third);
```

the string "The Cat in the Hat Comes Back" appears in the `output` stream. Notice that the system does not automatically insert blanks between two concatenated strings.

Concatenation works only with values of type `String`. However, if we try to concatenate a value of one of Java's built-in types to a string, Java automatically converts the value into an equivalent string and performs the concatenation. In fact, we can concatenate an object of any class to a string; the system looks for the object's `toString` operation to transform the object into a string before the concatenation.

Array Lists The `ArrayList` class is part of the `java.util` package. The functionality of the `ArrayList` class is similar to that of the array. In fact, the array is the underlying implementation structure used in the class. In contrast to an array, however, an array list can grow and shrink; its size is not fixed for its lifetime.

The `ArrayList` class was added to the library with the release of Java 1.2. It provides essentially the same functionality as the original library's `Vector` class, with which you may be familiar from a previous course. However, the `Vector` class supports concurrent programming; that is, it supports programs that have more than one active thread. A *thread* is a flow of control in a program. Advanced Java programs can have multiple control flows that execute simultaneously and interact with each other. The support that is necessary to enable concurrent programming requires extra processing whenever a `Vector` method is invoked, even when we aren't using multiple threads. The extra processing makes the `Vector` class a poor choice for use with single-threaded programs, such as the programs of this textbook. For single-threaded programs, you should use the `ArrayList` class instead of the `Vector` class.

Logical Level We approach the logical view of array lists by comparing and contrasting them with arrays. Like an array, an array list is a structured composite data type, made up of a collection of ordered elements. As with an array, we can access an element of an array list directly by specifying an index. However, arrays and array lists differ in many ways:

1. Arrays can be declared to hold data of a specific type; array lists hold variables of type `Object`. Therefore, *every* array list can hold virtually any type of data, even a primitive type if it is contained within a wrapper object.
2. An array remains at a fixed capacity throughout its lifetime; the capacities of array lists grow and shrink, depending on need.
3. An array has a length; an array list has a size, indicating how many objects it is currently holding, and a capacity, indicating how many elements the underlying implementation could hold without having to be increased.

The following table describes some of the interesting `ArrayList` operations.

Method Name	Parameter Type	Returns	Operation Performed
`ArrayList`	(none)		Constructs an empty array list of capacity 10.
`ArrayList`	`int`		Constructs an empty array list of the capacity indicated by the parameter.
`add`	`int`, `Object`	`void`	Inserts the specified `Object` at the specified position; shifts all subsequent elements to the right one place.
`add`	`Object`	`void`	Inserts the specified `Object` at the end.
`ensureCapacity`	`int`	`void`	Increases the capacity of the array list to at least the specified capacity, if it is currently less than the specified capacity.
`get`	`int`	`Object`	Returns the element at the specified position.
`isEmpty`	(none)	`boolean`	Returns `true` if the array list is empty, `false` otherwise.
`remove`	`int`	`Object`	Removes the element at the specified position, shifts all subsequent elements to the left one place, and returns the removed element.
`size`	(none)	`int`	Returns current size.
`trimToSize`	(none)	`void`	Trims the capacity of the array list to its size.

Implementation Level It is not necessary to peek at the underlying implementation of array lists in order to use them in our programs. Nevertheless, it is an instructive exercise, and helps us understand when to choose an array list structure over an array and vice versa.

We can imagine a Java array list consisting of an array and two integer variables that hold the capacity (length) and size (number of current elements) of the array. The underlying array is always "left-justified;" in other words, any empty slots are at higher indices then the slots being used.

It is easy to see how observer methods, like `get`, `isEmpty`, and `size` are implemented; the appropriate information is simply calculated and returned. But what about operations that change the contents of the array list; for example, the `add` operation? These are more interesting.

First, we consider a "standard" `add` operation, one that does not require a change in the size of an array list. Suppose we have an array list `letters` that we are using to hold characters. Suppose its capacity is 8 and it has a current size of 6. (See the "Before" section of Figure 2.15, which represents this situation. In the figure we follow several simplifying conventions: we show characters inside the array locations rather than show each of them as separate objects; we label the underlying array with the name of the array list.)

Now suppose we perform the operation

```
letters.add(2, 'X');
```

To make room for the addition of the character `'X'` at index 2, the underlying implementation would first copy the elements at positions 5 down to 2 into locations 6 down to 3. That frees location 2 so that the `'X'` can be copied into it. Additionally, the `size` variable would need to be updated. (See the "Processing" section of Figure 2.15, which represents the activity taking place during the add operation, and the "After" section, which represents the state of the array list after the operation is completed.)

Notice that inserting one element into the array list requires many steps; depending on where the element is inserted, it could require shifting the entire contents of the underlying array (if inserted at index 0), or it could require no shifting whatsoever (if inserted at location size).

The processing becomes even more complicated if we try to add an element to an array list that is already at its capacity. In this case, the underlying implementation creates a new array to hold the array list information—an array that is larger than the current array list. It then copies the contents of the old array into the new array, leaving an empty slot for the additional element. Finally, it copies the new element into the appropriate location of the new array and updates the `capacity` and `size` variables. The new array is now the array list; the old array is garbage and is eventually reclaimed by the run-time garbage-collection process.

There is actually more that goes on behind the scenes than we have described here; however, we think we have covered enough for you to get the point. With an array, memory is reserved ahead of time to hold the array elements; with an array list, memory can be allocated "on the fly," as needed. Array lists can be a useful construct for saving space, but the space savings might be at the expense of extra processing time. Time/space tradeoffs are common in computer programming.

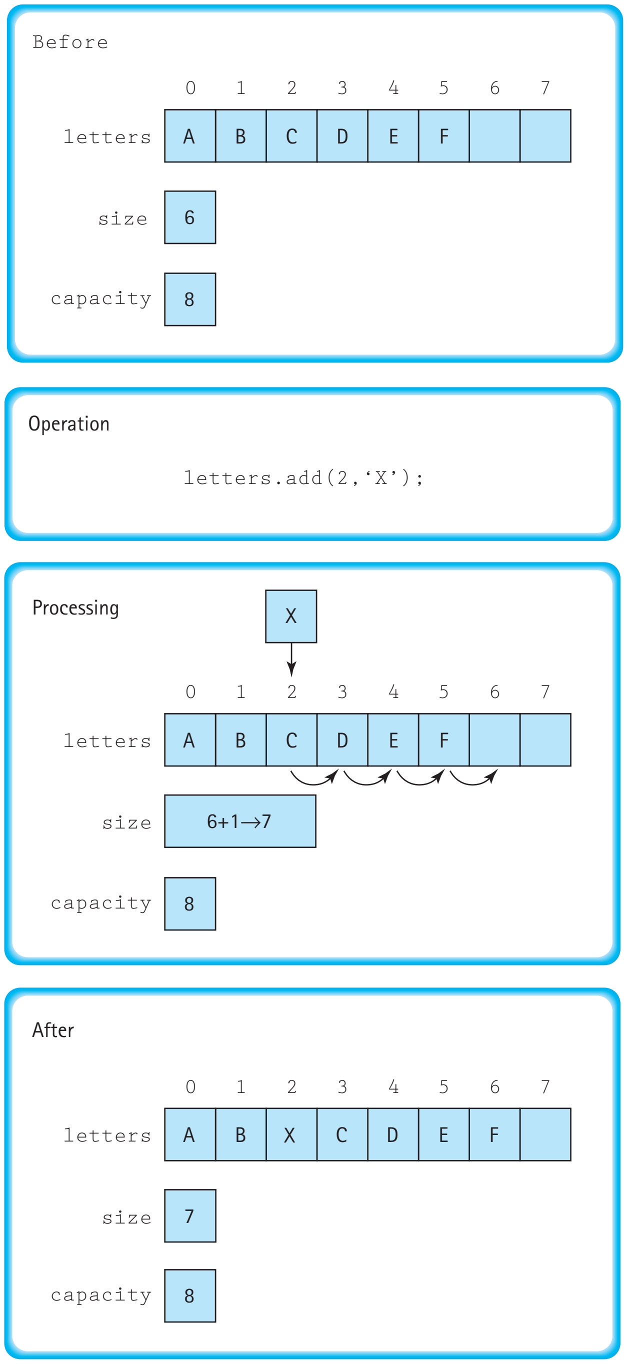


Figure 2.15 *Array list implementation*

Application Level Due to the similarities between arrays and array lists, we can use an array list in place of an array in virtually any application. However, the differences between arrays and array lists often mean that for a specific application, one or the other of these structures is the appropriate choice. It is impossible to list definite rules on when to choose one approach over the other, since there can be multiple factors to consider, and since each application has its own requirements. Nevertheless, we offer the following short set of guidelines:

Use an array when

1. space is not an issue.
2. execution time is an issue.
3. the amount of space required does not change much from one execution of the program to the next.
4. the actively used size of the array does not change much during execution.
5. the position of an element in the array has relevance to the application. (For example, the value in location *n* represents the profits for day *n* of a business period.)

Use an array list when

1. space is an issue.
2. execution time is not an issue.
3. the amount of space required changes drastically from one execution of the program to the next.
4. the actively used size of the array list changes a great deal during execution.
5. the position of an element in the array list has no relevance to the application.
6. most of the insertions and deletions to the array list take place at the size index. (Therefore, no extra overhead is incurred by these operations.)

We use an array list in Chapter 4 to implement a Stack ADT.

Building Our Own ADTs

In Chapter 1 we emphasized that the central task in the object-oriented design of software is the identification of classes. Once we identify the logical properties of the classes that we use to solve our problem, we must either find pre-existing versions of the classes or build them ourselves. Designing and building classes as ADTs allows us to take advantage of the benefits of abstraction and information hiding.

Remember that ADTs can be considered at three levels: The logical level specifies the interface and functionality, the implementation level is where the coding details take place, and the application level is where the ADT is used. Sometimes one programmer is involved in all three levels of an ADT—the same individual describes it, builds it, and uses it. At other times the design might come from one programmer, the implementation from another, and a third might be the one to use it. In the course of our discussions, we typically assume that the designer and implementer are the same person; we call this person the programmer. We also assume that the same person, or perhaps other people, are the ones to use the ADT at the application level; in that role we call them the application programmers. Finally, there are the people who use the application programs; we call them the users.

To help us understand what makes a class an ADT, we return to two previous examples, `Circle` and `Date`. Figure 2.16 lists a version of each of these, side by side, so that you can easily compare their implementations. `Circle` is an example of a record structure. It is not an ADT since its instance variables are not hidden. `Date` is an ADT. Its instance variables are hidden and cannot be directly accessed from outside the class.

```
the Circle record
public class Circle
{
  public int xValue;
  public int yValue;
  public float radius;
  public boolean solid;
}
```

```
the Date ADT
public class Date
{
  protected int year;
  protected int month;
  protected int day;
  protected static final int MINYEAR = 1583;

  public Date(int newMonth, int newDay, int newYear)
  {
    month = newMonth;
    day = newDay;
    year = newYear;
  }

  public int yearIs()
  {
    return year;
  }

  public int monthIs()
  {
    return month;
  }

  public int dayIs()
  {
    return day;
  }
}
```

Figure 2.16 *`Circle` and `Date` implementations*

Access Modifiers

The difference in visibility of the `Circle` data and the `Date` data is due to the access modifiers used in the declaration of the data. Java allows a wide spectrum of access control, as summarized in the following table:

Modifier	Visibility
`public`	Within the class, subclasses in the same package, subclasses in other packages, everywhere
`protected`	Within the class, subclasses in the same package, subclasses in other packages
`package`	Within the class, subclasses in the same package
`private`	Within the class

The `public` access modifier used in `Circle` makes its data "publicly" available; any code that can "see" an object of the class can access and change its data. Additionally, any class derived from the `Circle` class inherits its public parts.

Public access sits at one end of the access spectrum, allowing open access to the data. At the other end of the spectrum is private access. When a programmer declares a class's variables and methods as `private`, they can be used only inside the class itself and they are not inherited by subclasses. We often use private access within our ADTs to hide their data. However, if we intend to extend our ADTs with subclasses, we may want to use the protected or package access instead.

The `protected` access modifier used in `Date` is similar to private access, only slightly less rigid. It "protects" its data from outside access, but allows it to be accessed from within its own class or from any class derived from its class. You may recall that in Chapter 1 we created a subclass of `Date` called `IncDate` that included a transformer method `increment`. The `increment` method required access to the instance variables of `Date`, since it would update the represented date to the next day. Therefore, the `Date` instance variables were assigned protected access. (An even better approach might have been to include `Date` and `IncDate` in the same package, perhaps a `Calendar` package, and use package access as described in the next paragraph.)

The remaining type of access is called *package access*. A variable or method of a class defaults to package access if none of the other three modifiers are used. Package access means that the variable or method is accessible to any other class in the same package; also the variable or method is inherited by any of its subclasses that are in the same package.

Note that the same rules for visibility and inheritance described above for instance variables apply equally well to the methods, constants, and inner classes of a class.

Exported Methods

If we hide the data of our ADTs, then how can other classes use the data? The answer is through publicly available methods of the class. By restricting access of the data of a class to the methods of the class, we reap the benefits of abstraction and information hiding that were described in Chapter 1.

Consider once again the implementation of the Date ADT in Figure 2.16. The `year`, `month`, and `day` variables are all protected from outside access. This particular ADT provides one constructor method, `Date`, which accepts three integer parameters and initializes the variables of the `Date` object accordingly. The Date ADT also provides three observer methods: `yearIs`, `monthIs`, and `dayIs`. Using the constructor and observer methods, another class can create `Date` objects and "observe" the constituent data.

It is not hard to imagine creating some more interesting methods for the `Date` class. For example, as was suggested before, we could include a transformer method called `increment` that would change the value of the `Date` to the next day. We could also create a method that operates on more than one `Date` object—for example, a `difference` method that returns the number of days between two dates. The method could accept one date as a parameter and use the `Date` instance through which it is invoked as the other date. Its declaration might look something like this:

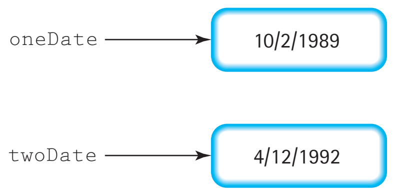
```
public int difference(Date inDate);
```

In that case, the following program segment would assign the value 5 to the variable `daysLeft`.

```
Date holiday = new Date (12, 25, 2002);
Date today = new Date (12, 20, 2002);
int daysLeft;

daysLeft = holiday.difference(today);
```

Copying Objects

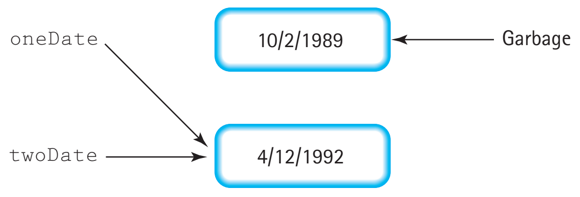
In the course of using an ADT, an application programmer might need to make a copy of the ADT object. Since ADTs are implemented as classes, they are handled by reference; if you simply use Java's assignment operator (=) to perform the copy, you end up with an alias of the copied object. For example, suppose `oneDate` and `twoDate` are both `Date` objects, representing the dates 10/2/1989 and 4/12/1992, respectively:



Then the statement

```
oneDate = twoDate;
```

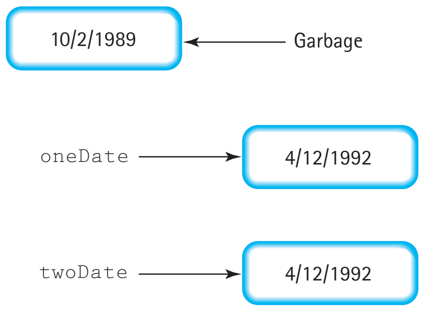
would create aliases, and garbage, as follows:



To create a true copy, and not just an alias, a programmer could use the `Date` constructor and observer methods as follows:

```
oneDate = new Date (twoDate.monthIs(), twoDate.dayIs(), twoDate.yearIs());
```

This approach would create a new `Date` object with the same variable values as the `twoDate` object. The result of the operation would look like this:



This approach eliminates the creation of an alias. Now, `oneDate` and `twoDate` are separate objects, and changes to one do not affect the other.

Since creating a copy of an ADT is a common operation, it is appropriate to include a special constructor for an ADT, called a *copy constructor*, which encapsulates the above operation. We pass the copy constructor an instance of the ADT and it creates a new instance of the ADT that is a copy of the argument. For the `Date` class the copy constructor would be:

```
public Date (Date inDate)
{
  year = inDate.year;
  month = inDate.month;
  day = inDate.day;
}
```

Notice that within the copy constructor the system has direct access to the instance variables of the `Date` parameter `inDate`, even though those variables were declared as protected. This works because this code resides inside the `Date` class and therefore has access to the private and protected members. Using the copy constructor, we can now create a true copy as follows:

```
oneDate = new Date (twoDate);
```

Creating the copy constructor for the `Date` class was fairly straightforward. We simply had to copy the variables of the `Date` parameter to the fields of the new `Date` object. This approach works fine for a simple ADT like `Date`. However, we must be more careful when working with composite ADTs.

Previously in this chapter, in the section about Aggregate Objects, we listed the following definitions of the `Point` and `NewCircle` classes:

```
public class Point          public class NewCircle
{                           {
  public int xValue;          public Point location;
  public int yValue;          public float radius;
}                             public boolean solid;
                            }
```

As you can see, an object of the class `NewCircle` is a composite object, since one of its instance variables is an object of the class `Point`. Consider the following code that implements a copy constructor for `NewCircle` in the same straightforward manner that was used for the `Date` class above:

```
public NewCircle (NewCircle inNewCircle)
// This code is incorrect
{
  location = inNewCircle.location;
  radius = inNewCircle.radius;
  solid = inNewCircle.solid;
}
```

At first glance this seems as if it would provide a reasonable copy of a `Circle` object. However, upon closer scrutiny, we see that there is a hidden alias that has been created. The line in the constructor that copies the location variable

```
location = inNewCircle.location;
```

is using the standard assignment statement on an object. Since all objects are handled by reference, what is actually copied is the reference to that object, rather than the contents of the object. We end up with two separate `Circle` objects that are both referencing the same `Point` object. The `NewCircle` copy constructor above is an example of a shallow copy. Shallow copying is rarely useful.

Shallow copy An operation that copies a source class instance to a destination class instance, simply copying all references so that the destination instance contains duplicate references to values that are also referred to by the source.

To rectify the problems created with a shallow copy, we need to create new instances of any nonprimitive variables of the object that we are copying. This approach results in a deep copy. The correct code for the copy constructor for `NewCircle` is:

```
public NewCircle (NewCircle inNewCircle)
{
  location = new Point;
  location.xValue = inNewCircle.location.xValue;
  location.yValue = inNewCircle.location.yValue;
  radius = inNewCircle.radius;
  solid = inNewCircle.solid;
}
```

Deep copy An operation that copies one class instance to another, using observer methods as necessary to eliminate nested references and copy only the primitive types that they refer to. The result is that the two instances do not contain any duplicate references.

The key statement in the code above is the first statement where we use the new command to create a new instance of a `Point` object.

Notice that in this example, since the classes we are using have public instance variables, we were able to just directly access the `x` and `y` values of the `location` variables of the `inNewCircle` parameter. If we were dealing with ADTs we would have to use the appropriate observer methods. Alternately, if the `Point` class included its own copy constructor, we could use it to create the new `Point` object:

```
location = new Point(inNewCircle.location);
```

Figure 2.17 summarizes our discussion of copying objects. It shows the results of all three approaches to copying a `Circle` object: using a simple assignment statement, using a shallow copy, and using a deep copy. In the figure, both `oneCircle` and `twoCircle` are objects of type `NewCircle`.

Exceptions

When creating our own ADTs it is possible to identify exceptional situations that require special processing. If it is the case that the special processing cannot be determined ahead of time. It is application dependent; we should use the Java *exception* mechanism to throw the problem out of the ADT and force application programmers to handle the exceptional situation on their own. On the other hand, if handling the exceptional situation can be hidden within the ADT, then there is no need to burden the application programmers with the task of handling exceptions.

For an example of an exception created to support a programmer-defined ADT, let's return to our `Date` class. As currently defined, a `Date` constructor could be used to create dates with nonexistent months—for example, 15/15/2000 or even −5/15/2000. We could avoid the creation of such dates by checking the legality of the month argument

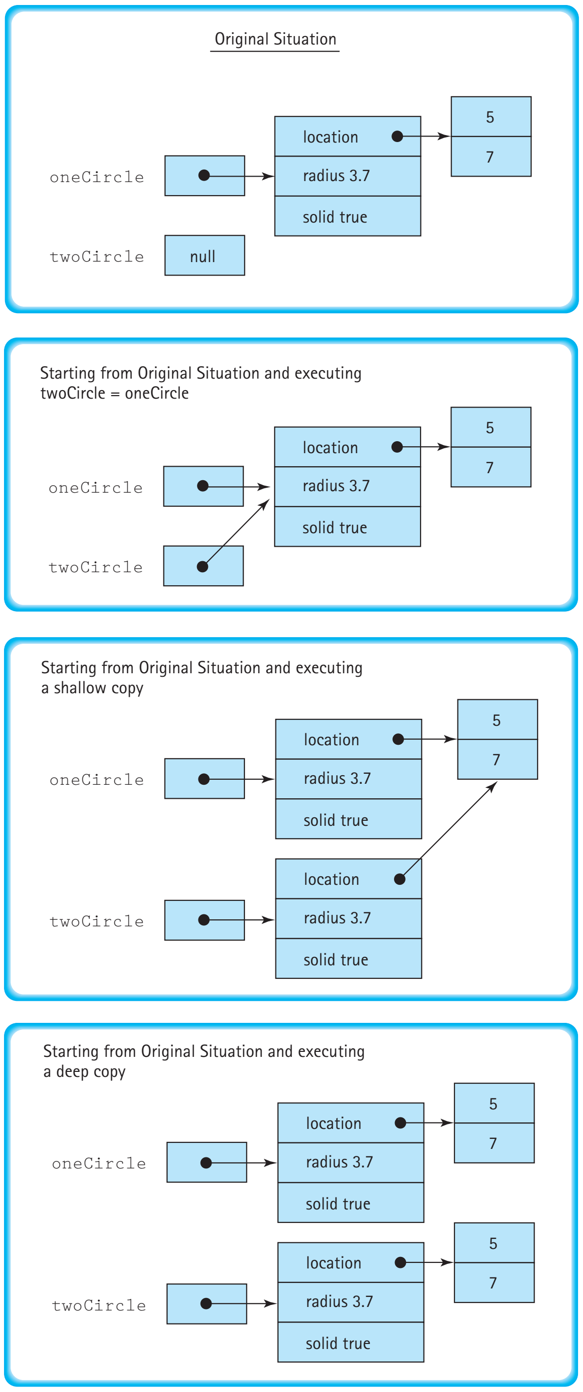


Figure 2.17 *Copying objects*

passed to the constructor. But what should our constructor do if it discovers an illegal argument? Some options:

- Write a warning message to the output stream. That's not a very good option because within the Date ADT we don't really know what output stream is being used by the application.
- Instantiate the new `Date` object to some default date, perhaps 0/0/0. The problem with this approach is that the application program may just continue processing as if nothing is wrong, and produce erroneous results. In general it is better for a program to "bomb" then to produce erroneous results that may be used to make bad decisions.
- Throw an exception. This way, normal processing is interrupted and the constructor does not have to return a new object; instead, the application program is forced to acknowledge the problem and either handle it or throw it out to the next level.

Once we have decided to handle the situation with an exception, we must decide whether to use one of the library's predefined exceptions, or to create one of our own. A study of the library in this case reveals a candidate exception called `DataFormatException`, to be used to signal data format errors. We could use that exception but we decide it doesn't really fit, since its not the format of the data that is the problem in this case, it is the values of the data.

So, we decided to create our own exception, `DateOutOfBounds`. We could call it "MonthoutofBounds" but we decide that we want to use the exception to indicate other potential problems with dates, and not just problems with the month value. For example, in the `Date` class we defined a class variable `MINYEAR` (set to 1583), representing the first complete year in which the Gregorian calendar was in use. Application programmers should not use our `Date` class to represent dates earlier than that year. The idea is that date calculations get very complicated if you allow dates before 1583. For one thing, leap year rules were different; for another, there were 10 days that were skipped in the middle of 1582. We are imagining that we have added methods to the class that would be affected by such things, for example a method that returns the number of days between two dates. Therefore, we wish to disallow such dates.

We create our `DateOutOfBounds` exception by extending the library `Exception` class. It is customary when creating your own exceptions to define two constructors, mirroring the two constructors of the `Exception` class. In fact, the easiest thing to do is define the constructors so that they just call the corresponding constructors of the superclass:

```
public class DateOutOfBoundsException extends Exception
{
  public DateOutOfBoundsException()
  {
    super();
  }
  public DateOutOfBoundsException(String message)
  {
    super(message);
  }
}
```

The first constructor is used to create an exception without an associated message; the second constructor creates an exception with a message equal to the string argument passed to the constructor.

Next we need to consider when, within our Date ADT, we throw the exception. All places within our ADT where a date value is created or changed should be examined to see if the resultant value could be an illegal date. If so, we should create an object of our exception class with an appropriate message, and throw the exception. Here is how we might write a `Date` constructor to check for legal months and years. (Checking for legal days is much more complicated and we leave that as an exercise.)

```
public Date(int newMonth, int newDay, int newYear) throws DateOutOfBound-
sException
{
  if ((newMonth <= 0) || (newMonth > 12))
    throw new DateOutOfBoundsException("month must be in range 1 to 12");
  else
    month = newMonth;

  day = newDay;

  if (newYear < MINYEAR)
    throw new DateOutOfBoundsException("year " + newYear +
                                       " is too early");
  else
    year = newYear;
 }
```

Notice that the message defined for each *throws* clause pertains to the problem discovered at that point in the code. This should help the application program that is handling the exception, or at least provide pertinent information to the user of the program if the exception is propagated all the way out to the user level.

Finally, let's see how an application program might now use the `Date` class. Consider a program called `UseDates` that prompts the user for a month, day, and year, and create a `Date` object based on the user's responses. In the following code we ignore the details of how the prompt and response are handled, to concentrate on the topics of our current discussion:

```
public class UseDates
{
  public static void main(String[] args) throws DateOutOfBoundsException
  {
    Date theDate;
    // Program prompts user for a date
    // M is set equal to user's month
    // D is set equal to user's day
    // Y is set equal to user's year
    theDate = new Date(M, D, Y);

    // Program continues
  }
}
```

When this program runs, and the user responds with a legal month, day, and year, there is no problem. However, if the user responds with an illegal value—for example, a year value of 1051—the `DateOutOfBoundsException` is thrown by the `Date` constructor; since it is not caught within the program, it is thrown out to the interpreter. The interpreter stops execution of the program after displaying a message like this:

```
Exception in thread "main" DateOutOfBoundsException: year 1051 is too early
          at Date.<init>(Date.java:18)
          at UseDates.main(UseDates.java:57)
```

The interpreter's message includes the name and message string of the exception, and a trace of what calls were made leading up to the exception being thrown.

Alternately, the `UseDates` class could be defined to catch and handle the exception itself, rather than throwing it to the interpreter. The application programmer could reprompt for the date in the case of the exception being raised. Then `UseDates` might be written as follows (again we ignore the user interface details):

```
public class UseDates
{
  public static void main(String[] args)
  {
    Date theDate;
    boolean DateOK = false;

    while (!DateOK)
    {
      // Program prompts user for a date
      // M is set equal to user's month
      // D is set equal to user's day
      // Y is set equal to user's year
      try
      {
        theDate = new Date(M, D, Y);
        DateOK = true;
      }
      catch(DateOutOfBoundsException OB)
      {
        output.println(OB.getMessage());
      }
    }

    // Program continues
  }
}
```

If the *new* statement executes without any trouble, meaning the `Date` constructor did not throw an exception, then the `DateOK` variable is set to `true` and the *while* loop terminates. On the other hand, if the `DateOutOfBounds` exception is thrown by the `Date` constructor, it is caught by the *catch* statement. This in turn prints out the message associated with the exception and the *while* loop is re-executed, again prompting the user for a date. The program repeatedly prompts for date information until it is given a legal date.

Notice that the `main` method no longer throws `DateOutOfBoundsException`, since it handles the exception itself.

There are several factors to consider when determining how to use exceptions when creating our own ADTs. First of all, we should decide what types of events can trigger exceptions. Remember that exceptions can be used to signal any out-of-the-ordinary event that requires special processing—there is no language-based rule that says the event must be error related. For example, it would be possible to break out of an input loop in reaction to an exception you raise when you try to read past the end of a file. Reading the end-of-file marker is not really an error; it is something we expect to happen eventually when we read files. It is, in a sense, an exceptional condition, and we can use Java's exception mechanisms to help us handle its occurrence.

To simplify our ADT definitions, and to support a common approach to the way we define our ADTs, we throw programmer-defined exceptions from our ADTs only in situations involving errors. For example, unexpected date values being passed to a method or illegal sequencing of methods calls are errors. However, this does not mean we always use exceptions in these cases.

When dealing with error situations within our ADT methods, we have several options:

1. We can detect and handle the error within the method itself. This is the best approach if the error can be handled internally and if it does not greatly complicate design.
2. We can throw an exception related to the error and force the calling method to either handle the exception or to rethrow it. If it is not clear how to handle a particular error situation, the best approach might be to throw it out to a level where it can be handled.
3. We can ignore the error situation. Recall the "programming by contract" discussion, related to preconditions, in the Designing for Correctness section of Chapter 1. If the preconditions of a method are not met, the method is not responsible for the consequences. This approach is best if we are confident that the contract is usually met by the application classes.

Therefore, when we define our ADTs, we partition potential error situations into three sets: those to be handled internally to the ADT, those to be thrown as an exception back to the calling process, and those that are assumed not to occur. We document this third approach in the preconditions of the appropriate methods. We attempt to strike a balance between the complexity required to handle all possible error situations internally, and the lack of safety involved with handling everything by contract.

As a general rule, an exceptional situation should be handled at the lowest level that "knows" how to handle it. If the information needed to handle the exception is not available at a level, then the exception should be thrown. As we create ADTs to be used in applications we see that quite often it is the application level that can best handle the exceptions raised within the ADTs. We see examples of this as we proceed through the text.

The feature section below suggests a sequence of steps to follow when designing and creating ADTs. The steps include many of the techniques introduced in this subsection.

Designing ADTs

When you design and create your own ADTs you can follow these steps:

1. Determine the general purpose of the ADT; determine how the application programmers use the ADT to help solve their problems in a general sense.
2. List the specific types of operations the application program performs with the ADT. If possible, note how often the different operations are used, that is, the expected relative frequency of operation calls.
3. Identify a set of public methods to be provided by the ADT class that allow the application program to perform the desired operations. Note that there might not be a one-to-one correspondence between the desired operations and the exported methods. It may be that a single operation requires several method invocations. For example, in Chapter 3 we define a list ADT with methods `lengthIs`, `reset`, and `getNextItem`. An application program

must use all three of these methods to implement a "Print List" operation. In addition, a specific method might be needed for more than one operation. For example, the `lengthIs` list method might be used by a "Print List" operation and by a "Report List Size" operation.

4. Identify other public methods, based on experience and general guidelines, which help make the ADT generally usable. For example, the copy constructor described in the earlier section titled Copying Objects is usually a good method to include. You might organize all your identified methods into constructors, observers, transformers, and iterators.
5. Identify potential error situations and classify into
   a. Those that are handled by throwing an exception
   b. Those that are handled by contract
   c. Those that are ignored
6. Define the needed exception classes.
7. Decide how to structure the data to best support the needed operations and identified methods. Remember that alternate organizations may support some operations better than others. This is where the frequency of operation information may be useful.
8. Decide on a protection level for the identified data. Hide the data as much as possible.
9. Identify private structures and methods that support the required public methods. Functional decomposition of the required actions of the public methods may help identify common requirements that can be supported by shared private methods.
10. Implement the ADT, possibly collecting all related files into a single package.
11. Create a test driver like the one at the end of Chapter 1 and test your ADT with a wide variety of operations.

Note that the classic data structures, modeled as ADTs created in the remainder of this text have evolved over the last 50 years. Therefore, we can draw from a great deal of previous research and experience when designing these structures, instead of analyzing specific problem situations as suggested above.

Summary

We have discussed how data can be viewed from multiple perspectives, and we have seen how Java encapsulates the implementations of its predefined types and allows us to encapsulate our own class implementations.

As we create data structures, using built-in data types such as arrays and classes to implement them, we see that there are actually many levels of data abstraction. The abstract view of an array might be seen as the implementation level of the programmer-defined data structure `List`, which uses an array to hold its elements. At the logical level, we do not access the elements of `List` through their array indexes but through a set of accessing operations defined especially for objects of `List` type. A data type that is designed to hold other objects is called a *container* or collection type. Moving up a level, we might see the abstract view of `List` as the implementation level of another programmer-defined data type, `ProductInventory`, and so on.

Perspectives on Data

Application or user view	Logical or abstract view	Implementation view
Product Inventory	List	Array
List	Array	Row major access function
Array	Row major access function	32-bit words

What do we gain by separating the views of the data? First, we reduce complexity at the higher levels of the design, making the program easier to understand. Second, we make the program more easily modifiable: The implementation can be completely changed without affecting the program that uses the data structure. We use this advantage in this text, developing various implementations of the same objects in different chapters. Third, we develop software that is *reusable*: The structure and its accessing operations can be used by other programs, for completely different applications, as long as the correct interfaces are maintained. You saw in the first chapter of this book that the design, implementation, and verification of high-quality computer software is a very laborious process. Being able to reuse pieces that are already designed, coded, and tested cuts down on the amount of work we have to do.

In the chapters that follow we extend these ideas to build other container classes: lists, stacks, queues, priority queues, trees, and graphs. While the Java Class Library provides many of these data structures (along with generic algorithms and iterator structures), the techniques for building these structures is so important in computer science that we believe you should learn them now.

We consider these data structures from the logical view. What is our abstract picture of the data, and what accessing operations can we use to create, assign to, and manipulate the data elements? We express our logical view as an abstract data type (ADT) and record its description in a data specification.

Next, we take the application view of the data, using an instance of the ADT in a short example.

Finally, we change hats and turn to the implementation view of the ADT. We consider the Java type declarations that represent the data structure, as well as the design of the methods that implement the specifications of the abstract view. Data structures can be implemented in more than one way, so we often look at alternative representations and methods for comparing them. In some of the chapters, we include a longer Case Study in which instances of the ADT are used to solve a problem.

Summary of Classes and Support Files

Here are the classes defined in Chapter 2. The classes are listed in the order in which they appear in the text. The summary includes the name of the class file, the page on which the file is first referenced, and a few notes. The notes explain how the class was used in the text, followed by additional notes if appropriate. Note that we do not include classes defined within other classes (inner classes), such as the `Circle` class that was defined within the `TestCircle` class, in the table. The class files are available on our web site in the `ch02` subdirectory.

Classes Defined in Chapter 2

File	First Ref.	Notes
`TestCircle.java`	page 83	Illustrates records and record component selection.
`FigureGeometry.java`	page 88	An example of an interface.
`Point.java`	page 93	Very small class; it is used to build an example of an aggregate object.
`NewCircle.java`	page 93	Example of a class that defines aggregate objects. `NewCircle` includes an instance variable of the class `Point`.

Other than the `Exception` class, which was discussed in Section 2.3, no Java Library Classes were used in any examples for the first time in this text within this chapter. Of course, many library classes were discussed; but they were not used in programs.

Exercises

2.1 Different Views of Data

1. Why are primitive types sometimes called atomic types?
2. Explain what we mean by data abstraction.
3. What is data encapsulation? Explain the programming goal “to protect our data abstraction through encapsulation.”
4. Describe the four categories of operations that can be performed on encapsulated data. Give an example of each operation using a Library analogy.
5. Name three different perspectives from which we can view data. Using the logical data structure “a list of student academic records,” give examples of what each perspective might tell us about the data.
6. Consider the abstract data type `GroceryStore`.
   a. At the application level, describe `GroceryStore`.
   b. At the logical level, what grocery store operations might be defined for the customer?

   c. Specify (at the logical level) the operation `CheckOut`.
   d. Write an algorithm (at the implementation level) for the operation `CheckOut`.
   e. Explain how parts (c) and (d) represent information hiding.

2.2 Java's Built-in Types

7. What primitive types are predefined in the Java language?
8. What composite types are predefined in the Java language?
9. Describe the component selector for classes, when they are used as records.
10. Define a `toString` method for the circle class listed on the following pages:
    a. page 83
    b. page 93
11. What is an alias? Show an example of how it is created by a Java program. Explain the dangers of aliases.
12. Assume that `date1` and `date2` are objects of type `IncDate` as defined in Chapter 1. What would be the output of the following code?

```
date1 = new IncDate(5, 5, 2000);
date2 = date1;
System.out.println(date1);
System.out.println(date2);
date1.increment();
System.out.println(date1);
System.out.println(date2);
```

13. Assume that `date1` and `date2` are objects of type `IncDate` as defined in Chapter 1. What would be the output of the following code?

```
date1 = new IncDate(5, 5, 2000);
date2 = new IncDate(5, 5, 2000);
if (date1 == date2)
  System.out.println("equal");
else
  System.out.println("not equal");
date1 = date2;
if (date1 == date2)
  System.out.println("equal");
else
  System.out.println("not equal");
date1.increment();
if (date1 == date2)
  System.out.println("equal");
else
  System.out.println("not equal");
```

14. What is garbage? Show an example of how it is created by a Java program.
15. What is an abstract method?
16. What sorts of constructs can be declared in a Java interface?
17. Briefly describe four uses for Java interfaces.
18. What are the fundamental differences between classes and arrays?
19. Describe the component selectors for one-dimensional arrays.
20. Write a program that declares a ten-element array of `int`, uses a *for* loop to initialize each element to the value of its index squared, and then uses another *for* loop to print the contents of the array, one integer per line.
21. Define a three-dimensional array at the logical level.
22. Suggest some applications for three-dimensional arrays.
23. Indicate which Java types would most appropriately model each of the following (more than one may be appropriate for each):
    a. A chessboard
    b. Information about a single product in an inventory-control program
    c. A list of famous quotations
    d. The casualty figures (number of deaths per year) for highway accidents in Texas from 1954 to 1974
    e. The casualty figures for highway accidents in each of the states from 1954 to 1974
    f. The casualty figures for highway accidents in each of the states from 1954 to 1974, subdivided by month
    g. An electronic address book (name, address, and phone information for all your friends)
    h. A collection of hourly temperatures for a 24-hour period

2.3 Class-Based Types

24. What Java construct is used to represent abstract data types?
25. Explain the difference between using a Java class to create a record and to create an ADT
26. Explain how packages are used to organize Java files.
27. List and briefly describe the contents of five Java library packages.
28. List the eight Java Library “wrapper” classes that support the objectification of Java’s primitive types.
29. List and describe five Java Library classes that are not described in this chapter.
30. Research the Java Library `Random` class. Use it in a program to do the following.
    a. Generate a sequence of 10,000 random integers between 1 and 100 and output the average value generated

b. Play the high/low guessing game with the user; the program generates a random integer between 1 and 100,000. The user must repeatedly guess the number until it is correct. After each guess, the program informs the user if the secret number is higher or lower than the guess.

Be sure to carefully test your program(s).

31. Write a program that declares a ten-element array of `Integer`, uses a *for* loop to initialize each element to the value of its index squared, and then uses another *for* loop to print the contents of the array, one integer per line.

32. Describe the output of the following code that uses `String` variables `S1`, `S2`, and `S3`.

```
S1 = "Alex";
S2 = "Bob";
S3 = S1 + S2;
System.out.println(S3);
S2 = S1.toUpperCase();
System.out.println(S2);
S3 = "Chris".
if (S1.compareTo(S3) < 0)
  System.out.println("less than zero");
else
  System.out.println("not less than zero");
```

33. Explain the differences between arrays and array lists.

34. For each of the following situations, state whether it is best to use an array list or an array.
    a. To hold student test grades, where the size of the class of students is always between 15 and 20
    b. To hold student test grades, where the size of classes varies widely
    c. To hold the number of miles traveled each day of a month
    d. To hold a list of items, where you need to repeatedly insert elements into random locations in the list
    e. To hold a list of items, where you insert and remove items only from the far end of the list.

33. Describe each of the four levels of visibility provided by Java's visibility modifiers.

34. Illustrate with a figure the difference between a shallow copy and a deep copy of an aggregate object.

35. Consider an ADT `SquareMatrix`. (A square matrix can be represented by a two-dimensional array with *n* rows and *n* columns.)

a. Write the specification for the ADT, assuming a maximum size of 50 rows and columns. Include the following operations:

   `MakeEmpty(n)`, which sets the first `n` rows and columns to zero

   `StoreValue(i, j, value)`, which stores value into the position at row `i`, column `j`

   `Add`, which adds two matrices together

   `Subtract`, which subtracts one matrix from another

   `Copy`, which copies one matrix into another

b. Convert your specification to a Java class declaration.

c. Implement the member methods.

d. Write a test plan for your class.

36. Expand your solution to Exercise 34 of Chapter 1, where you implemented the `Date` and `IncDate` classes, to include the appropriate throwing of the `DateOutOfBoundsException`, as described in this chapter.

37. Write a class `Array` that encapsulates an array and provides bounds checked access. The private instance variables should be `int index`, and `int array[10]`. The public members should be a default constructor and methods (signatures shown below) to provide read and write access to the array:

```
void insert(int location, int value);

int retrieve(int location);
```

If the `location` is within the correct range for the array, the `insert` method should set that location of the array to the value. Likewise, if the `location` is within the correct range for the array, the `retrieve` method should return the value of that location of the array. In either case, if the `location` is not within the correct range, the method should throw an exception of type `ArrayoutofBoundsException`. Write a driver to check the array accesses. Your driver should assign values to the array by using the `insert` method, using the `retrieve` method to read these values back from the array. It should also try calling both methods with illegal location values. Catch any exceptions thrown by placing the calls in a *try* block with an appropriate *catch* block following.

38. Describe the steps to follow when designing your own ADTs and implementing them with the Java class mechanism.