onlinecampus.bu.edu

Module 1

140-178 minutes

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

In this module, we first present a brief review of the basics of Java programming language. Then we discuss the basics of object-oriented programming and fundamental data structures. We end this module with basic mathematical facts that are used in this course.

After successfully completing this module, students will be able to:

1. Write a program implementing a class definition.
2. Write a program implementing a class that extends a superclass or implements an interface.
3. Use exception handlings in a program.
4. Implement generic methods.
5. Illustrate how insertion-sort works.
6. Implement list structures, including singly linked lists, circularly linked lists, and doubly linked lists.
7. Implement an *equals* method for a user-defined class, which overrides the *equals* method of the *Object* class.

8. Implement a *clone* method for a user-defined class, which overrides the *clone* method of the *Object* class.

Section 1.1. Java Basics

Overview

This section is prepared for students who are not familiar with the basics of Java programming language. However, we assume that students are familiar with at least one object-oriented programming language.

Since many basic features of Java, such as expressions and control flows, are similar to those of typical high-level programming languages, we discuss only those features that are unique to Java or different from those of other languages in a nontrivial way.

Section 1.1.1. Getting Started

We begin with a simple Java program that prints "Welcome to CS526."

Code Segment 1.1

```
public class HelloCS526 {
    public static void main (String[ ] args) {
        System.out.println ("Welcome to CS526");
    }
}
```

Every Java program must have at least one *class,* and this program has a class named *HelloCS526*. The keyword *public* is an *access* *control modifier* (or a *visibility modifier*) specifying that the class, method, or data field for which it is used, is accessible by any code. So, the class *HelloCS526* or the method *main* can be run by any other code.

A Java program is executed from the *main* method. The main method receives the command-line parameters as an array of strings via *String*[ ] *args* and does not return anything as noted by *void*.

The keyword *static* specifies that the method belongs to the class, not an object.

The main method in this program contains the *System.out.println* statement, which is a predefined method, and it prints the string "*Welcome to CS526*."

Comments can be written in two ways:

- Block comment—This is a traditional comment. All text between /* and */ is ignored. See the following example:

```
    /*
     * This is a block comment
     */
```

- Inline comment—This is also called *an end-of-line* comment. All text from // to the end of the line is ignored. See the following example:

```
    // This is an inline comment.
```

Section 1.1.2. Primitive Types

The Java programming language supports two kinds of data types: *primitive types* and *reference types*.

Primitive types are also called *base types* and include the following types:

- byte—8-bit signed 2’s complement integer; from -128 to 127, inclusive
- short—16-bit signed 2’s complement integer; from -32768 to 32767, inclusive
- int—32-bit signed 2’s complement integer; from -2147483648 to 2147483647, inclusive
- long—64-bit signed 2’s complement integer; from -9223372036854775808 to 9223372036854775807, inclusive
- char—16-bit Unicode character; from "\u0000" to "\uffff" inclusive, that is, from 0 to 65535
- float—single-precision, 32-bit floating-point number (IEEE 754-1985)
- double—double-precision, 64-bit floating-point number (IEEE 754-1985)
- Boolean—*true* or *false*

The following illustrates the declaration and initialization of primitive type variables.

```
    byte a, count = 0;  // only count is
initialized
    short b, c; // declared but not initialized
    int i, j, k = 1; // only k is initialized
    long l = 1000L; // "L" is used to denote the
long type
    float pi = 3.14159F; // "F" is used to denote
the float type
    double x = 1.568, y = 4.243e15; // both
initialized
    boolean done = false;
```

When a primitive-type instance variable is not initialized, Java automatically assigns a default value as the initial value. A numeric variable is initialized to *zero*, a character variable is initialized to *null*, and a Boolean variable is initialized to *false*.

A variable of *reference* type stores the *pointer* to (or the *address* of) an object, not the value of an object. The default value of a reference variable is *null*.

Section 1.1.3. Classes and Objects - Basics

In this section, we briefly review the basic concepts of classes and objects in Java.

Section 1.1.3.1. Basics

In object-oriented programming (OOP), *objects* are used to represent real-world entities. A set of objects with the same properties is defined as a *class*, and every object is an instance of the class. We use *object* and *instance* interchangeably. A class defines the *type* of objects as a *blueprint* or *template*. A class defines the data that the objects of the class store as *instance variables* and the behavior of the objects as *methods*. These are the two most important members of a class.

- *Instance variables*—These are also called *fields* or *data fields*. Instance variables represent the data of an object of the class. The instance variables of a class define the *state* of an object. Each instance variable has a *type*, which is either a primitive type or a reference type.
- *Methods*—Methods of a class define how to *access* and *modify* the data of the objects of the class. *Accessor methods* return the values of data fields and *update methods* change the values of instance variables.

A simple class definition is shown below:

Code Segment 1.2

```
public class Counter {

private int count;                      // a simple integer instance variable
public Counter() { }                    // default constructor (count is 0)
public Counter(int initial) { count = initial; } // an alternate constructor

/* returns the current count */
public int getCount() { return count; } // an accessor method

/* increment the count */
6    public void increment() { count++; }
// an update method

/* increase the count by the specified delta */
7    public void increment(int delta) { count +=
delta; }  // an update method

/* reset the count to 0 */
8    public void reset() { count = 0; }
// an update method
9  }
```

Line 2 declares an instance variable *count*. Lines 3 and 4 declare constructors, line 5 declares an accessor method, and lines 6, 7, and 8 declare update methods.

A constructor is a special method used to create a new object. The name of a constructor is always the same as the name of the class and there can be more than one constructor. Line 3 is a default constructor, which does not have an argument. When invoked, it creates a new *Counter* object with the initial value of *count* = 0. Line 4 has an alternate constructor, which receives as an argument the initial value of *count*.

Note that the above *Counter* class definition does not include a *main* method. So, it cannot be executed as a program. Instead, it is supposed to be used to create (counter) instances in a larger program.

Section 1.1.3. Classes and Objects - Creating and Using Objects

Section 1.1.3.2. Creating and Using Objects

The following code illustrates how objects are created and used:

Code Segment 1.3

```
public class CounterDemo {
  public static void main(String[] args) {
    Counter c;            // declares a
variable; no counter yet constructed
    c = new Counter();    // constructs a
counter; assigns its reference to c
    System.out.println("After c = new
Counter()");
    System.out.println("value of c is " +
c.getCount());
    c.increment();              // increases
its value by one
    System.out.println("After c.increment()");
    System.out.println("value of c is " +
c.getCount());
    c.increment(3);             // increases
its value by three more
    System.out.println("After c.increment(3)");
    System.out.println("value of c is " +
c.getCount());
    int temp = c.getCount();    // will be 4
    c.reset();                  // value
becomes 0
```

```java
    System.out.println("After c.reset()");
    System.out.println("value of c is " + c.getCount());
    Counter d = new Counter(5);// declares and constructs a counter having value 5
    System.out.println("After d = new Counter(5)");
    System.out.println("value of d is " + d.getCount());
    d.increment();              // value becomes 6
    System.out.println("After d.increment()");
    System.out.println("value of d is " + d.getCount());
    Counter e = d;              // assigns e to reference the same object as d
    System.out.println("After e = d");
    System.out.println("value of d is " + d.getCount());
    System.out.println("value of e is " + e.getCount());
    temp = e.getCount();    // will be 6 (as e and d reference the same counter)
    e.increment(2);             // value of e (also known as d) becomes 8
    System.out.println("After e.increment(2)");
    System.out.println("value of d is " + d.getCount());
    System.out.println("value of e is " + e.getCount());
```

```
32   }

33   public static void badReset(Counter c) {
34     c = new Counter();         // reassigns
local name c to a new counter
35   }

36   public static void goodReset(Counter c) {
37     c.reset();                 // resets the
counter sent by the caller
38   }
39 }
```

The above code segment has sufficient inline comments, most of which are self-explanatory. So, instead of discussing each line of the code, we only briefly discuss here what occurs when an object is created.

The *new* operator, along with the call to a constructor, creates a new object (or a new instance of the class). The constructor initializes instance variables.

- A memory is dynamically allocated to the new object and all instance variables are initialized with the default values.
- The constructor is invoked with the parameters, if specified. The constructor may perform additional functions.
- After the execution of the constructor, the *new* operator returns the *reference* to the new object, and the reference is assigned to an instance variable.

In line 4, the constructor without parameters (as defined in the class definition of *Counter* shown in Section 1.3.1) is invoked, a new object is created, and the reference to the new object is assigned to the instance variable *c*. In line 17, the constructor with a parameter (which is the initial value of *count*) is invoked and the reference to the new object is assigned to the instance variable *d*. Note that *c* or *d* is the reference to an object, not an object itself. But, sometimes we informally call them objects.

When we want to access the members of an object, we use the dot (" . ") operator. For example, to access the method *m* of an object *o* (or, more accurately, the reference to an object), we use the notation *o.m*, which is a method invocation. In lines 6 and 7, the method *getCount*( ) and the method *increment*( ), respectively, are invoked by the *Counter* object *c*.

Section 1.1.3. Classes and Objects - Defining a Class

Section 1.1.3.3. Defining a Class

In this section, we discuss class definitions a little bit more.

In a class definition, *modifiers* can be placed in front of a class, instance variable, or method. Modifiers specify additional properties of their subjects.

Access-Control Modifier (or Access-Level Modifiers)

Access modifiers—which include *public*, *protected*, and *private* —specify other classes' level of access (also called *visibility*) to the aspect to which a modifier is applied. Modifiers implement *encapsulation*, which is one of the most important concepts of the object-oriented paradigm.

- ***public*** modifier—All other classes may access the defined aspect.
- When a public modifier is used for a class it specifies that all other classes may create new instances of the class and declare variables and parameters of the class. For example, in Code Segment 1.2, the class Counter is declared as a public class. So, the class CounerDemo, in Code Segment 1.3, can declare a variable of Counter class (in line 3) and create an object of the Counter class (in line 4).
- If a method is declared public, the method can be invoked by any other class. The method increment() is invoked in line 3 of Code Fragment 1.3, for example.
- If an instance variable is declared public, the variable can be accessed in any other class. In Code Fragment 1.2, if the instance variable count were declared public, then any other class would be able to access the variable. For example, in some other class, a Counter object c (which is a reference variable) can be created and it can access the variable count using the syntax c.count.
- ***protected*** modifier—Classes in the same package and subclasses in a different package can access the defined aspect. (A package is a named group of related classes.)
- ***private*** modifier—The defined aspect is accessible only by the same class.

  In Code Fragment 1.2, the instance variable *count* is declared *private*. So, the *CounterDemo* class in Code Fragment 1.3 cannot directly access *count*. Instead, it has to use the *getCount*( ) method to access the current value of *count* and must invoke the *increment*( ) method to increment the value of *count*, for example.
- ***no modifier is specified*** **(this is *default*)**—The defined aspect is

accessible by the same class and the other classes in the same package.

The following table summarizes the access to members granted by each modifier:

Figure 1.1. Access Modifiers

Modifier	Access Level			
	Class	**Package**	**Subclass**	**World**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

The *static* Modifier

The *static* modifier can be specified for variables or methods of a class. Variables and methods that are declared *static* belong to the class not to an instance of the class.

Each object created from a class has its own copies of instance variables. Sometimes you may want to have a variable that is common to all objects of the class. In such a case, you can declare a *static variable* using the *static* modifier.

Consider the following two class definitions:

Code Segment 1.4

```
public class Student {
```

```java
    private int id;
    private int id;
    private String name;
    private String major;
    public static int numStudents = 0;

    public Student () {numStudents++; }
    public Student (int id, String name, String major) {
        this.id = id;
        this.name = name;
        this.major = major;
        numStudents++;
    }

    public int getId() { return id; }
    public String getName() { return name; }
    public String getMajor() { return major; }
    public int getNumStudents() { return numStudents; }

    public void setName(String name) {
        this.name = name ;
    }

    public void setMajor(String major) {
        this.major = major ;
    }

    public void setId(int id) {
        this.id = id ;
    }
  }
```

Code Segment 1.5

```
public class TestStudent {

   public static void main(String[] args) {
      Student s1 = new Student(1, "John", "CS");
      Student s2 = new Student(2, "Susan", "CS");

      System.out.println("Number of students is: " + Student.numStudents);

      System.out.println("Student 1: id = " + s1.getId() +
                                             "; Name = " + s1.getName() +
                                             "; Major = " + s1.getMajor());

      System.out.println("Student 2: id = " + s2.getId() +
                          "; Name = " + s2.getName() +
                          "; Major = " + s2.getMajor());

    }
  }
```

The first class defines the *Student* class, and the second has a main method that demonstrates the *Student* class.

In line 5 of the *Student* class, a static variable *numStudents* is declared with the initial value of 0. The value of *numStudents* is the number of *Student* objects that have been created. The variable *numStudents* is not associated with individual objects. Instead, it is associated with the class and is common to all objects of the class.

When a new *Student* object is created by the constructor in line 6, the number of student objects is incremented in line 10.

When a static variable is accessed, the variable name is qualified with the name of the class. In line 5 of the *TestStudent* class, *Student.numStudents* is used to retrieve the number of *Student* objects that were created so far. In this example code, "2" will be printed as the number of students.

The *abstract* Modifier

If a method is declared without implementation, it is an *abstract* method.

An *abstract* class is a class that is declared *abstract,* and it may or may not have an abstract method. If a class has one or more abstract methods, then it must be declared as *abstract*.

Abstract classes and abstract methods are important features of object-oriented programming and will be further discussed in Section 1.2.

The *final* Modifier

- *final* variable: If a variable is declared *final*, it may be assigned only once. In other words, once a value is assigned to a final variable, the variable is never assigned a new value. If the variable is a base type, it becomes a constant. If the variable is a reference type, it always refers to the same object (although the state of the object may change).
- *final* method: Declaring a method *final* prevents a subclass from overriding it.
- *final* class: If a class is declared *final*, it can never have subclasses.

Declaring Instance Variables

The general syntax of declaring instance variables is as follows:

```
{modifier} type identifier [= initialValue] {,
identifier [= initialValue2]};
```

Here, {*x*} denotes zero or more occurrences of *x* and [*x*] denotes zero or one occurrence of *x*.

A class definition may have any number of instance variables. Some examples are shown below:

```
public int count;
public int i = 0, j = 1, k ;
private static String s, t;
```

Declaring Methods

A method definition has two parts: the *signature* part (which is also called the *method header*) and the *body* part. The signature part includes the modifier(s), return type, method name, and parameters. The body part consists of statements that implement the method.

```
{modifier} returnType methodName (type parameter
{, type parameter }) {
  // method body
}
```

Return Types

A method may return a value. If it does, the data type of the return value must be specified. If a method does not return a value, the keyword *void* is used.

The following two methods are from the *Student* class of Code Segment 1.4.

```
public int getId() { return id; }

public void setMajor(String newMajor) {
  major = newMajor ;
}
```

The *getID* method does not have any parameter and returns *id*, which is of the integer type.

The *setMajor* method receives one parameter, *newMajor*, and it does not return any value.

A Java method can return only a single value.

Parameters

The parameters in the method signature are called *formal parameters*. When the method is invoked (or called), the caller must pass parameters that, are called *actual parameters*.

In Java, all parameters are *passed by value*. This means that the values of actual parameters are copied to corresponding formal parameters. The formal parameters are treated as local variables in the method and their values may be changed. However, the values of actual parameters in the caller are not affected by the change of formal parameters in the method. If an actual parameter is a reference variable—e.g., the reference to an object—the reference is copied to the formal parameter. In this case, both the actual parameter and the formal parameter refer to the same object and if the object state is changed in the called method, the change is also reflected in the caller.

Consider the following code segment:

Code Segment 1.6

```
public class DoubleTheSalary {
  public static void main(String[] args) {
    int salary = 10000;
    System.out.println("Salary before call is: " + salary);
    doubleSalary(salary);
    System.out.println("Salary after call is: " + salary);
  }
8    public static void doubleSalary (int salary)
{
9      salary = salary * 2;
10     System.out.println("Salary in doubleSalary
method is: " + salary);
11   }
12  }
```

In line 5, *salary* (the value of which is 10000) is passed to the *doubleSalary* method. Inside the method, *salary* is doubled. But, in the *main* method, the values of *salary* before and after the method call are identical, as illustrated by the following output:

```
Salary before call is: 10000
Salary in doubleValue method is: 20000
Salary after call is: 10000
```

Now consider the following code segment, where a *Student* object is passed as a parameter:

Code Segment 1.7

```
1  public class ChangeStudentMajor {
2    public static void main(String[] args) {
3        Student s3 = new Student(3, "Molly",
"CS");
4        System.out.println("Major before call is:
" + s3.getMajor());
5        String newMajor = "EE";
```

```
        changeMajor(s3, newMajor);
        System.out.println("Major after call is:
" + s3.getMajor());
    }
    public static void changeMajor (Student s,
String newMajor) {
        s.setMajor(newMajor);;
    }
}
```

The output of this code is as follows:

```
Major before call is: CS
Major after call is: EE
```

As the above output shows, the change made inside the *changeMajor* method is reflected in the *main* method.

Constructors

A constructor is a special method that is used, along with the *new* operator, to create and initialize an object that is an instance of a class. The general syntax for declaring a constructor is as follows:

```
  {modifier} constructorName (type parameter {,
type parameter }) {
  // constructor body
  }
```

Constructor declarations are similar to method declarations. But there are some differences:

- A constructor cannot be *static*, *abstract*, or *final*. So, the only modifiers that are allowed are visibility modifiers—i.e., *public*, *protected*, *private*, or no modifier (default).
- The name of a constructor must be the same as the name of the class.
- Constructors do not return a value so no return type is specified (not even *void*).

  A class may have multiple constructors with the same name but they must have different *signatures*.

  If no constructor is declared in a class definition, a *default constructor* is provided by Java. The default constructor does not have an argument and initializes all instance variables with default values.

  If one or more non-default constructors are declared, a default is not provided by Java. So, a default constructor must be declared in this case. Line 7 in Code Segment 1.4 declares a non-default constructor and line 6 declares a default constructor.

The Keyword *this*

Within the body of an instance method or a constructor, *this* is a reference to the current object for which the method is invoked or the object that is being created. In addition to storing the reference to the current object, *this* is also used for the following two purposes:

- When a parameter of a method or a constructor has the same name as that of an instance variable (this is sometimes referred to as *instance variable is shadowed by a parameter*), *this* can be used to distinguish the latter from the former. In the non-default constructor of the *Student* class in Code Segment 1.4, which is copied below, the parameter names are the same as the instance variable names.

```
7  public Student (int id, String name, String
major) {
8     this.id = id;
9     this.name = name;
10    this.major = major;
11    numStudents++;
12 }
```

In lines 8, 9, and 10, the values of parameters are assigned to instance variables, and this is used to qualify instance variables.

- The *this* keyword can be used to invoke another constructor inside a constructor. For example, in the *Student* class of Code Segment 1.4, we can declare one more constructor which invokes another constructor, as shown below:

```
public Student (int id, String name) {
   this(id, name, null);
}
```

This constructor receives id and name as parameters and invokes the constructor that receives all three parameters using this. Note that null is passed as the formal parameter of major.

The *Main* Method

A *main* method is a special method whereby the execution of a Java program begins.

In Java, some classes have a *main* method and others don't. Those classes without a *main* method cannot be run and are defined to be used by other classes.

A class with a *main* method can be executed. So, a Java program (that will be run) must have a class with a *main* method and there should be only one *main* method in a Java program.

A *main* method is declared in the following way:

```
public static void main (String[ ] args) {
   // main method body
}
```

The *public* modifier specifies that a *main* method can be called by any other class. The *static* modifier specifies that a *main* method is a method of a class, not an instance method. *Void* specifies that a *main* does not return a value.

A *main* method receives arguments as an array of String objects, the name of which is *args*. A Java program can be invoked from the command line as follows:

```
java className arguments
```

Here, *className* is the name of the class that has a *main* method and *arguments* is a list of command-line arguments that are passed to the *main* method.

Consider the following example code:

```
public class ArgTest {
        public static void main(String[] args) {
                String s1 = args[0];
                int a = Integer.parseInt(args[1]);
                double x =
Double.parseDouble(args[2]);

                System.out.println("s1 = " + s1 +
"\na = " + a + "\nx = " + x);
        }
}
```

Here, all arguments are passed as strings and the *parseInt* method of the Integer class and the *parseDouble* method of the Double class are used to convert a string to an integer and a double, respectively. We will discuss the Integer class and the Double class shortly.

Suppose the *ArgTest* program is invoked from the command line as follows:

```
java ArgTest "a string" 10 3.14
```

Then, "a string" is passed to args[0], 10 is passed to args[1], 3.14 is passed to args[3], and the expected output is as follows:

```
s1 = a string
a = 10
x = 3.14
```

Section 1.1.4. Strings

The primitive data type *char* stores a single character. Java provides a *String* class for manipulation of character strings. All character strings, including the *null* string, are instances of the *String* class.

The *String* class includes a large number of methods to facilitate manipulation of strings. The following example code illustrates the *charAt* method, the *indexOf* method, and a concatenation operator +.

```
public class stringTest {
  public static void main(String[] args) {
    String s1 = "Data Structures and Algorithms";
    char c = s1.charAt(5);
    int i = s1.indexOf('t');
    String s2 = s1 + " Course";

    System.out.println("c = " + c + "\ni = " + i +
"\ns2 = " + s2);
  }
}
```

The *charAt*(i) method returns the character of which the index is *i*. Note that the index of a character in a string is the position of the character in the string and it begins with 0. So, for example, *charAt*(5) returns the sixth character.

The *indexOf*(‘c’) method returns the index of the first occurrence of the character *c* in the string.

To concatenate two strings, the concatenation operator + is used.

The above code, when executed, will issue the following output:

```
c = S
i = 2
s2 = Data Structures and Algorithms Course
```

In Java, String objects are *immutable* so the values of String objects cannot be changed once created. The following statement creates a String object "Hello", and a variable *greeting*, which references the String object, is created.

```
String greeting = "Hello";
```

Suppose that we assign a different string *literal* to *greeting* with the following assignment:

```
greeting = "Ciao";
```

The variable *greeting* is now referencing a new string "Ciao", but the string object "Hello" still exists (even though it is not accessible any more).

Java provides a *StringBuilder* class to support *mutable* strings. The *StringBuilder* class is more flexible then the *String* class and supports more efficient manipulation of strings. A string is implemented as a sequence of characters and its content and length can be changed using StringBuilder's methods. Some of the methods are shown below:

```
void setLength(int newLength)—Set the length of
```

```
the sequence.
StringBuilder append(String s)—Append string s to
the end of the sequence.
StringBuilder delete(int start, int end)—Delete
subsequence from start to end-1
StringBuilder insert(int offset, String s)—Insert
string s before offset.
void setCharAt(int index, char c)—Replace the
character at index with c.
StringBuilder reverse()—Reverse the sequence of
characters.
String toString()—Return a String object that
contains the character string.
```

Section 1.1.5. Wrappers

When we manipulate numbers in a program, we can use number primitive data types, which are not objects. However, sometimes number objects are required instead of numbers of primitive types. One of the reasons is that a method may expect an object as an argument. Java provides a *wrapper* class for each primitive data type. These classes *wrap* the primitive type in an object. The following table shows primitive types and their corresponding wrapper classes. The table also shows examples of how to create and access objects.

Figure 1.2. Wrapper classes for primitive types

Primitive Type	**Wrapper Class**	**Creating Object**	**Accessing Object**
boolean	Boolean	obj = new	obj.booleanValue()

		Boolean(true)	
char	Character	obj = new Character('A')	obj.charValue()
byte	Byte	obj = new Byte((byte) 16)	obj.byteValue()
short	Short	obj = new Short((short) 128)	obj.shortValue()
int	Integer	obj = new Integer(1024)	obj.intValue()
long	Long	obj = new Long(4096L)	obj.longValue()
float	Float	obj = new Float(3.14F)	obj.floatValue()
double	Double	obj = new Double(3.14)	obj.doubleValue()

Suppose the following code is executed:

```
public class WrapperTest {
  public static void main(String[] args) {
    Character c = new Character('A');
    Integer a = new Integer(1024);
    Double x = new Double(3.14);

    System.out.println("c is " + c.charValue());
    System.out.println("a is " + a.intValue());
    System.out.println("x is " + x.doubleValue());
  }
}
```

They we will get the following output:

```
c is A
a is 1024
x is 3.14
```

The *wrapping* can be done by the compiler, which is called *autoboxing*. When you use a primitive type for which an object is needed, the compiler automatically *wraps* the primitive type in an object. The compiler also can do the opposite. If you use an object for which a primitive type is required, the compiler automatically converts the object to the corresponding primitive type. This is called *autounboxing*. The following example code illustrates autoboxing and autounboxing:

```
public class BoxingTest {
          public static void main(String[] args) {
    Integer a = 1024; // primitive value 1024 is
boxed into an object
    System.out.println("a is " + a.intValue());
    int b = a + 10; // object a is unboxed to
primitive type
    System.out.println("b is " + b);
          }
}
```

The output of the example code is as follows:

```
a is 1024
b is 1034
```

Section 1.1.6. Arrays

In Java, an array is a container object, but not an instance of any particular class. A variable of an array type is a reference to the array object. The length of an array is fixed when the array is created. Elements of an array are accessed by indexes. The index of the first element is 0. So, the index of the fifth element is 4. The following example shows how to declare and create arrays:

```
// declaring arrays
int[ ] age;
double[ ] salary;
String[ ] name;

// creating arrays
age = new int[4];
salary = new double[12];
name = new String[10];

// declaring and creating arrays at the same time
int[ ] age = new int[4];
double[ ] salary = new double[12];
String[ ] name = new String[10];
```

The following code segment illustrate how to create and access array elements:

Code Segment 1.8

```java
public class CreateArray {
  public static void main(String[] args) {
    int[] oddArray = {1,3,5};
    int[] evenArray = new int[3];
    evenArray[0] = 2;
    evenArray[1] = 4;
    evenArray[2] = 6;

    System.out.println("Element of oddArray at index 0: " + oddArray[0]);
    System.out.println("Element of oddArray at index 1: " + oddArray[1]);
    System.out.println("Element of oddArray at index 2: " + oddArray[2]);
    System.out.println("Element of evenArray at index 0: " + evenArray[0]);
    System.out.println("Element of evenArray at index 1: " + evenArray[1]);
    System.out.println("Element of evenArray at index 2: " + evenArray[2]);
  }
}
```

The expected output is as follows:

```
Element of oddArray at index 0: 1
Element of oddArray at index 1: 3
Element of oddArray at index 2: 5
Element of evenArray at index 0: 2
Element of evenArray at index 1: 4
Element of evenArray at index 2: 6
```

We need to be careful that we do not try to access an array with an index outside the valid range of the array’s indexes. Suppose that the size of an array is *n*, so the range of indexes is 0 to *n* – 1. If we try to access the array with an index outside this range, then *ArrayIndexOutOfBoundsException* occurs. We will discuss *exceptions* in a later section.

Section 1.1.7. *Enum* Types

Enum is a special data type with which we can predefine a finite set of constants for a variable. For example, we can predefine a data type *Direction,* which has four constants: EAST, WEST, NORTH, and SOUTH. The following code illustrates how to define a variable of *enum* data type and how it can be used in a program.

Code Segment 1.9

```
public class EnumTest {
  public enum Direction {EAST, WEST, NORTH, SOUTH};

  Direction dir;
  double speed;

  public EnumTest(Direction dir, double speed){
```

```
6      this.dir = dir;
7      this.speed = speed;
8    }

9    public void windForecast(String day){
10     System.out.println(day + "'s wind forecast
is " + dir + " " +
11                                   speed + "
mph");
12   }

13   public static void main(String[] args) {
14     EnumTest todayForecast = new
EnumTest(Direction.NORTH, 15);
15     EnumTest tomorrowForecast = new
EnumTest(Direction.EAST, 20);
16     todayForecast.windForecast("Today");
17     tomorrowForecast.windForecast("Tomorrow");
18   }
19 }
```

The *enum* type *Direction* is declared in line 2, and the variable *dir* of *Direction* type is declared in line 3. In lines 14 and 15, the constants *NORTH* and *EAST* are used, which are qualified by the *enum* type name.

The expected out of the code is as follows:

```
Today's wind forecast is NORTH 15.0 mph
Tomorrow's wind forecast is EAST 20.0 mph
```

Section 1.1.8. Simple I/O

In this section, we discuss simple input from and output to the Java console window.

Java provides the *System.out* object to write to the standard output device, which usually is a display screen and the *System.in* object to read from the standard input, which is a keyboard.

The *System.out* object is an instance of the *java.io.PrintStream* class. Some methods defined in *java.io.Stream* are shown below:

```
void print(String s): Print string s
void print(Object o): Print object o
void print(baseType b): Print baseType value b
void println(String s): Print string s, followed
by newline
void println(Object o): Print object o, followed
by newline
void println(baseType b): Print baseType value b,
followed by newline
```

Here, *baseType* represents any of the primitive data types.

One simple way of reading from the standard input is to create the *Scanner* class. The *Scanner* breaks down the input into tokens separated by delimiters. The default delimiter is *whitespace*.
Once a *Scanner* object is created with the *System.in* object, you can read the next token using an appropriate *next* method.

The following code illustrates reading from the standard input and writing to the standard output:

Code Segment 1.10

```
import java.util.Scanner;

public class SimpleIOTest1 {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    System.out.print("Enter a string: ");
    String s = in.next(); // convert to string
    System.out.print("Enter an integer: ");
    int a = in.nextInt(); // convert to integer
    System.out.print("Enter a doubler: ");
    double x = in.nextDouble(); // convert to double

    System.out.println("s = " + s);
    System.out.println("a = " + a);
    System.out.println("x = " + x);
  }
}
```

An example output of the code is as follows:

```
Enter a string: Hello
Enter an integer: 10
Enter a doubler: 3.14
s = Hello
a = 10
x = 3.14
```

Note that the *next*( ) method is used to read a string, the *nextInt*( ) method is used to read an integer, and the *nextDouble*( ) method is used to read a double.

Here is another example:

Code Segment 1.11

```
import java.util.Scanner;

public class SimpleIOTest2 {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);

    System.out.print("Enter a string, an integer and a double: ");
    String s = in.next();
    int a = in.nextInt();
    double x = in.nextDouble();

    System.out.println("s = " + s);
    System.out.println("a = " + a);
    System.out.println("x = " + x);
  }
}
```

The following is an example output:

```
Enter a string, an integer and a double: Java 20
3.14159
```

```
s = Java
a = 20
x = 3.14159
```

In the above example, the string "Java 20 3.14159" is split into three tokens—*Java*, *20*, and *3.14159*—and they are converted to String, Integer, and Double by the *next*( ), *nextInt*( ), and *nextDouble*( ) methods, respectively.

Section 1.1.9. An Example Program

In this section, we discuss an example Java class which supports simplified credit card applications. Instead of discussing the complete code, which can be found here (CreditCard.java), we only briefly describe instance variables and high-level behavior.

This class has five instance variables, which are shown below along with brief description:

```
  private String customer;      // name of the
customer (e.g., "John Bowman")
  private String bank;          // name of the
bank (e.g., "California Savings")
  private String account;       // account
identifier (e.g., "5391 0375 9387 5309")
  private int limit;            // credit limit
(measured in dollars)
  protected double balance;     // current balance
(measured in dollars)
```

It has two constructors:

```
  public CreditCard(String cust, String bk, String
acnt, int lim, double initialBal)
  {
    // implementation omitted
  }

  public CreditCard(String cust, String bk, String
acnt, int lim) {
    // implementation omitted
  }
```

The first constructor receives as its argument all five variables. The second constructor receives all but *initalBal*, which will be initialized with 0.

There are five access methods—one for each instance variable, shown below:

```
  public String getCustomer() { return customer; }
  public String getBank() { return bank; }
  public String getAccount() { return account; }
  public int getLimit() { return limit; }
  public double getBalance() { return balance; }
```

Two updates are defined—one to charge a credit card and the other to process a payment:

```
  public boolean charge(double price) { // make a
charge
  // implementation omitted
  }
  public void makePayment(double amount) { // make
a payment
    // implementation omitted
  }
```

There is one utility method that displays a credit card's information:

```
  public static void printSummary(CreditCard card)
{
    // implementation omitted
  }
```

There is also a *main* method which simulates simple credit card transactions and can be found in the complete code of *CreditCard.java*.

Section 1.1.10. Packages and Imports

A stand-alone public class must be in a separate file, and the file name must be the same as the class name plus the *java* extension. So, the name of the file that has the *public class StudentTest* definition is *StudentTest.java*. A file with a public class may have other stand-alone classes, but these other classes cannot have *public* visibility.

Packages are an effective way of organizing and managing related types. Packages allow the grouping of related types providing access protection and name-space management. Note that *types* refer to classes, interfaces, enumerations, and annotation types.

A package can be created by including a *package statement* at the top of every source file that contains the types that will be included in the package.

A package statement consists of the keyword *package* and the name of the package. An example package statement follows:

```
package cs526;
```

Some advantages of packages are as follow:

- You and other programmers can easily determine that these types are related.
- The names of your types won't conflict with the type names in other packages because the package creates a new namespace.
- You can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

In a file system, types belonging to the same packages are placed in the same directory, and the package name and directory name must be identical. When referencing a type in a package, you must qualify the name of the type with the package name. For example, the *Scanner* class is defined in the *java.util* package so we use *java.util.Scanner* to refer to the *Scanner* class.

So, whenever we want to use *Scanner* in a program, we need to write *java.util.Scanner*. For example, to create a new *Scanner* object, we should write the following:

```
java.utl.Scanner input = new
java.util.Scanner(System.in);
```

But it is cumbersome to write the whole package hierarchy whenever we need *Scanner*. This can be avoided using an *import* statement. First, we import the *java.util.Scanner* package with the following import statement (this was done in the example programs *SimpleIOTest*1 and *SimpleIOTest*2 in Section 1.1.8):

```
import java.util.Scanner;
```

Then we do not have to write the whole package hierarchy any more. Instead the *input* object can be created using only *Scanner,* as shown below:

```
Scanner input = new Scanner(System.in);
```

The import statement shown above imports the *Scanner* class of the *java.util* package. If you use many types in the *java.util* package (in addition to the *Scanner* class), you can import the whole java.util package with the following import statement:

```
import java.util.*;
```

Test Yourself 1.1

Write a Java program that consists of three classes A, B, and C, such that B extends A and C extends B. Each class should have an instance variable named "x" (that is, each has its own variable named x). Describe a way for a method in C to access and set A’s version of x to a given value, without changing B or C’s version.

Please think carefully, practice and write your own program, and then click "Show Answer" to compare yours to the suggested possible solution.

Section 1.2. Object-Oriented Design

It is assumed that students who are taking this course are already familiar with the basic concepts of object-oriented design. We briefly discuss object-oriented features in Java.

Section 1.2.1. Inheritance

One of the important features of object-oriented programming is hierarchy of classes and inheritance of certain properties of a class by its subclasses. We briefly review basics of inheritance in this section.

Section 1.2.1.1. Extending the CreditCard Class

We use the credit card example program (of Section 1.1.9) to illustrate inheritance in Java. We define a subclass of the *CreditCard* class named *PredatoryCreditCard*. The UML diagram of this class hierarchy is shown below:

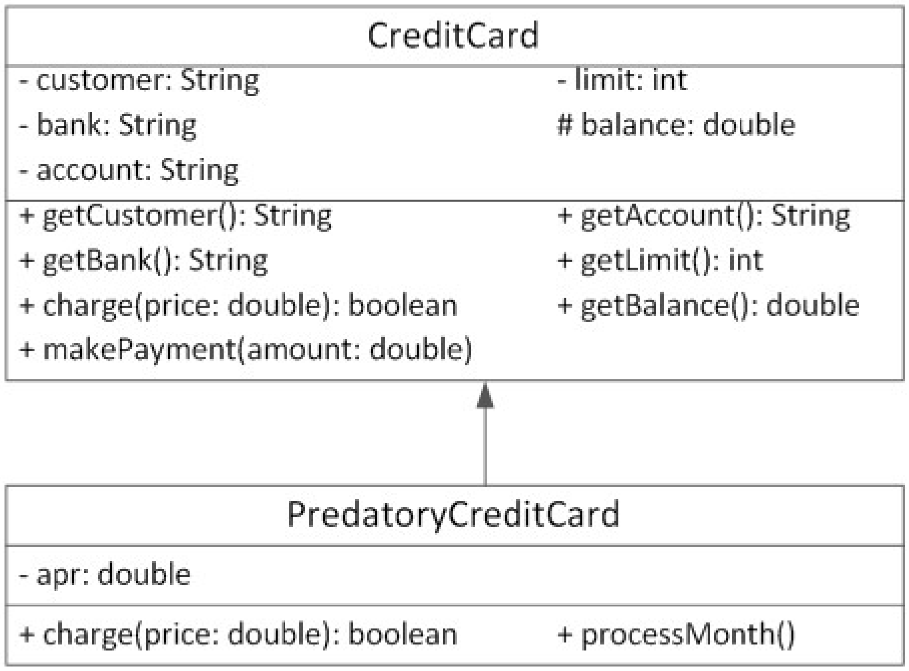


Figure 1.3. UML diagram for *PredatoryCreditCard* class as a subclass of *CreditCard* class.

Here, the — sign denotes a *private* modifier, the + sign denotes a *public* modifier, and the # sign denotes a *protected* modifier.

The *PredatoryCreditCard* inherits all variables and methods of the *CreditCardit* class and defines an additional variable *apr* and an additional method *processMonth*. It also redefines the *charge* method by overriding the same method in the superclass.

Code Segment 1.12

```
public class PredatoryCreditCard extends CreditCard {
  // Additional instance variable
  private double apr; // annual percentage rate
  public PredatoryCreditCard(String cust, String bk, String acnt, int lim,
                             double initialBal, double rate) {
    super(cust, bk, acnt, lim, initialBal); // initialize superclass attributes
    apr = rate;
  }
  /** Assess monthly interest on any outstanding balance. */
  public void processMonth() {
    if (balance > 0) {  // only charge interest on a positive balance
      double monthlyFactor = Math.pow(1 + apr, 1.0/12); // compute monthly rate
      balance *= monthlyFactor;
// assess interest
14     }
15   }
16   public boolean charge(double price) {
17     boolean isSuccess = super.charge(price);
// call inherited method
18     if (!isSuccess)
19       balance += 5;
// assess a \$5 penalty
20     return isSuccess;
21   }
22 }
```

The keyword *extends* in line 1 specifies that *PredatoryCreditCard* is a subclass of *the CreditCard* class. So, *PredatoryCreditCard* inherits all accessible variables and methods of *CreditCard*. Lines 4 through 8 define the constructor (note that constructors are not inherited in Java). Line 6 invokes the constructor of the superclass to initialize the instance variables of the superclass. If the constructor in a subclass does not call the superclass's constructor or another constructor with *this*, then a default constructor of its superclass, *super*( ), is implicitly invoked. Line 7 initializes the specialized attribute *apr* in the subclass. The *processMonth* method is a specialized method in the subclass. It calculates and assesses the monthly interest charge. This method also updates *balance*. It is possible for this subclass to access *balance*, which is an instance variable in the superclass, because *balance* is declared as *protected* in the superclass. The *charge* method overrides the one in the superclass. It first calls the *charge* method in the superclass (line 17). If the superclass's *charge* method returns *false* because the total balance exceeds the credit limit (line 18), then it assesses a $5 penalty (line 19).

Section 1.2.1.2. Polymorphism and Dynamic Dispatch

Suppose that we declare a reference variable of *CreditCard* type as follows:

CreditCard card;

Then we can create an instance of the *CreditCard* class and assign it to the *card* reference variable. It is also possible to create an instance of the *PredatoryCreditCard* class and assign it to the *card* reference variable, as shown below:

CreditCard card = new *PredatoryCreditCard*(. . .);

This is allowed because the *PredatoryCreditCard* class is a subclass of the *CreditCard* class. So, the reference variable *card* can refer to an instance of *the CreditCard* class or to an instance of the *PredatoryCreditCard* class. We say that the reference variable *card* is *polymorphic*. In general, *polymorphic* means that a reference of a class can be assigned an instance of any direct or indirect subclass of that class.

If we declare a reference variable of the *PredatoryCreditCard* class, however, it cannot be assigned an instance of the *CreditCard* class.

In the *CreditCard* class, there is a method called *charge*. In the *PredatoryCreditCard* class, the *charge* method is redefined to override the one in the superclass. So, there is a *charge* method in both classes (with the same name), and their implementations are different. One interesting question is, if the *charge* method is invoked by *card*, which implementation is executed?

The *declared type* of the instance *card* is *CreditCard* and the *actual type* of the instance *card* is *PredatoryCreditCard*. Java then invokes the implementation of the *actual type* during the runtime. So, in this case, the implementation of the *PredatoryCreditCard* class is invoked. This is called *dynamic dispatch* or *dynamic binding*.

Section 1.2.1.3. Inheritance Hierarchies

In this section, we describe another example that illustrates inheritance. The example has one superclass, *Progression*, and three subclasses: *ArithmeticProgression*, *GeometricProgression*, and *FibonacciProgression*.

A *numeric progression* is a sequence of numbers, each of which depends on one or more of the previous numbers. Each number in a progression is called a *term*.

In an arithmetic progression, the differences between consecutive terms are the same. So, a term can be calculated by adding a fixed constant to the previous term. In a geometric progression, the ratios of consecutive terms are the same. In this case, a term can be obtained by multiplying the previous term by a fixed constant. In the Fibonacci progression, a term is obtained by adding two previous terms.

The *Progression* class has one instance variable and three methods, as well as two constructors. Its implementation is shown below:

Code Segment 1.13

```
public class Progression {
  // instance variable
  protected long current;
  /** Constructs a progression starting at zero. */
  public Progression() { this(0); }
  /** Constructs a progression with given start value. */
  public Progression(long start) { current = start; }
  /** Returns the next value of the progression. */
  public long nextValue() {
    long answer = current;
    advance();    // this protected call advances the current value
    return answer;
  }
  /** Advances the current value to the next value of the progression. */
  protected void advance() {
    current++;
  }
  /** Prints the next n values of the progression, separated by spaces. */
  public void printProgression(int n) {
    System.out.print(nextValue());   // print first value without leading space
    for (int j=1; j < n; j++)
      System.out.print(" " + nextValue());  // print leading space before others
```

```
23     System.out.println();                    // end the line
24   }
25 }
```

The instance variable *current* (line 3) keeps the value of the current term. The constructor in line 7 creates a new *Progression* instance and initializes the start value with *start*, which is passed as an argument. The constructor in line 5 creates an instance with the initial value of 0 by calling the constructor of line 7. The *nextValue* method in line 9 returns the value of the next term, as determined by the *advance* method in line 15. The *advance* method simply increments the current value (this is overridden in subclasses). The *printProgression* method in line 19 prints the sequence of numbers of the progression.

A code of *ArithmeticProgression* is shown below:

Code Segment 1.14

```
1  public class ArithmeticProgression extends Progression {
2    protected long increment;
3    /** Constructs progression 0, 1, 2, ... */
4    public ArithmeticProgression() { this(1, 0); } // start at 0 with increment of 1
5    /** Constructs progression 0, stepsize, 2*stepsize, ... */
6    public ArithmeticProgression(long stepsize) { this(stepsize, 0); } // start at 0
7    /** Constructs arithmetic progression with
arbitrary start and increment. */
8    public ArithmeticProgression(long stepsize,
long start) {
9      super(start);
10     increment = stepsize;
11   }
12   /** Adds the arithmetic increment to the
current value. */
13   protected void advance() {
14     current += increment;
15   }
16 }
```

The line 2 declares a specialized variable *increment*. There are three constructors. The constructor in line 4 creates an instance that starts at 0 with the increment of 1. The constructor in line 6 creates an instance that starts at 0 with the increment *stepsize* which is passed as an argument. The one in line 8 creates an instance that starts at *start* with the increment *stepsize*, both of which are passed as arguments. Note that the first two constructors call the third constructor with the *this* keyword. The *advance* method in line 13 overrides the one in the superclass and calculates the next term by adding *increment* to the current term.

The *GeometricProgression* class and the *Fibonacci* class are defined in a similar way. A code of the *GeometricProgression* class is shown below:

Code Segment 1.15

```
1  public class GeometricProgression extends Progression {
2    protected long base;
3    /** Constructs progression 1, 2, 4, 8, 16, ... */
4    public GeometricProgression() { this(2, 1); } // start at 1 with base of 2
5    /** Constructs progression 1, b, b^2, b^3, b^4, ... for base b. */
6    public GeometricProgression(long b) { this(b, 1); }  // start at 1
7    /** Constructs geometric progression with arbitrary base and start. */
8    public GeometricProgression(long b, long start) {
9      super(start);
10     base = b;
11   }
12   /** Multiplies the current value by the geometric base. */
13   protected void advance() {
14     current *= base;                // multiply current by the geometric base
15   }
16 }
```

The *base* in line 2 declares the *base* or *common ratio* of a *GeometricProgression* instance. There are three constructors. The first constructor creates a progression that starts at 1 with the common ratio of 2; the second constructor creates a progression starting at 1 with the common ratio of *b,* which is passed as an argument; and the third constructor creates a progression starting at *start* with the common ratio *b,* both of which are passed as arguments.

A code of the *FibonacciProgression* is shown below:

Code Segment 1.16

```
public class FibonacciProgression extends Progression {
  protected long prev;
  /** Creates traditional Fibonacci, starting 0, 1, 1, 2, 3, ... */
  public FibonacciProgression() { this(0, 1); }
  /** Creates generalized Fibonacci, with give first and second values. */
  public FibonacciProgression(long first, long second) {
    super(first);
    prev = second - first;     // fictitious value preceding the first
  }
  /** Replaces (prev,current) with (current, current+prev). */
  protected void advance() {
    long temp = prev;
    prev = current;
    current += temp;
15    }
16 }
```

In the *Fibonacci* progression, the value of a term is obtained by adding previous two terms. So, it has two instance variables: *current*, which is inherited from the superclass, and *prev* which is a specialized variable. The first constructor method in line 4 creates an instance with 0 and 1 as the first two terms. The second constructor in line 6 receives *first* and *second* as arguments and uses these two as the first two terms of a new instance. The *advance* method in line 11 calculates the next term from *current* and *prev*. It also updates the instance variables *current* and *prev*. In line 13, the old *current* becomes the new *prev*. In line 14, the next term is calculated and becomes the new *current*.

The class-hierarchy diagram is shown below:

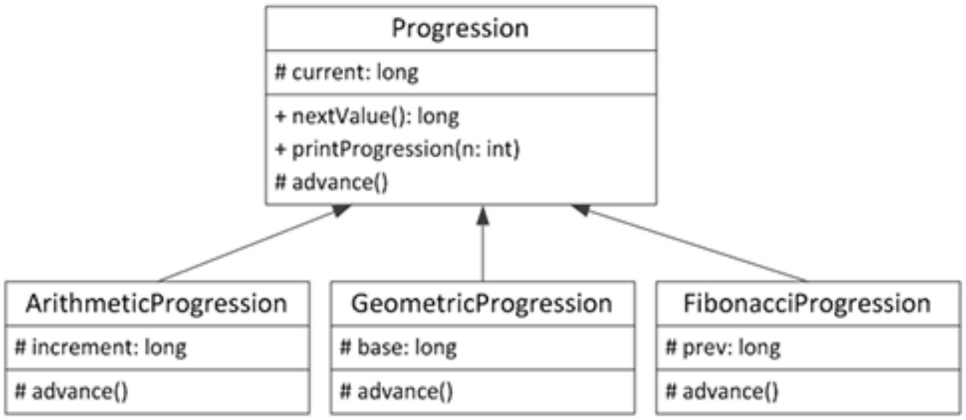


Figure 1.4. The *Progression* class and its subclasses.

Section 1.2.2. Interfaces and Abstract Classes

An interface is a *contract* between different groups of programmers which specifies how their programs interact. The following example from Oracle Documentation describes the general concept of an interface:

> Automobile manufacturers write software that operates the automobile—stop, start, accelerate, turn left, and so forth. Another industrial group, electronic guidance manufacturers, make computer systems that receive GPS (Global Positioning System) position data and wireless transmission of traffic conditions and use that information to drive the car.
>
> The auto manufacturers must publish an *industry-standard interface* that spells out in detail what methods can be invoked to make the car move (any car, from any manufacturer). The guidance manufacturers can then write software that invokes the methods described in the *interface* to command the car. Neither industrial group needs to know *how* the other group's software is implemented. In fact, each group considers its software highly proprietary and reserves the right to modify it at any time, as long as it continues to adhere to the published interface.

This example also illustrates that an interface serves as an *application programming interface* (API).

Section 1.2.2.1. Interfaces in Java

In Java, an interface is similar to a class but it can contain *only* constants, method signatures, default methods, static methods, and nested classes. All methods are empty except default methods and static methods, for which method bodies exist. Interfaces do not have constructors and cannot be directly instantiated. Interfaces can only be implemented by classes or extended by other interfaces.

When a class implements an interface, it must implement all methods, the signatures of which are specified in the interface. In this way, the users of the interface adhere to the *contract* as spelled out by the signatures of the methods.

Suppose, for example, that we have a collection of antique objects. Assume that we want to identify sellable objects and enforce a uniform set of behaviors on these objects. Then, we can define an interface that contains a set of method signatures. When sellable objects are defined, they will all implement the interface and the methods in the interface. So, all sellable objects will have the same behavior, as specified by the method signatures.

A code of the interface is given below:

Code Segment 1.17

```
/** Interface for objects that can be sold. */
public interface Sellable {
  /** Returns a description of the object. */
  public String description();
  /** Returns the list price in cents. */
  public int listPrice();
  /** Returns the lowest price in cents we will
accept. */
  public int lowestPrice();
}
```

If we identify photographs as sellable objects, we can define a *Photograph* class which implements the *Sellable* interface, as shown below:

Code Segment 1.18

```
/** Class for photographs that can be sold. */
public class Photograph implements Sellable {
  private String descript;
// description of this photo
  private int price;
// the price we are setting
  private boolean color;
// true if photo is in color
  public Photograph(String desc, int p, boolean
c) {  // constructor
    descript = desc;
    price = p;
    color = c;
  }
  public String description() { return
descript; }
  public int listPrice() { return price; }
  public int lowestPrice() { return price/2; }
  public boolean isColor() { return color; }
}
```

The *Photograph* class implements the methods *description*, *listPrice*, and *lowestPrice* which are defined in the *Sellable* interface. Additionally, it defines the *isColor* method.

For the objects that can be transported, we can define another interface: *Transportable*. A code of the interface is given below:

Code Segment 1.19

```
/** Interface for objects that can be transported. */
public interface Transportable {
  /** Returns the weight in grams. */
  public int weight();
  /** Returns whether the object is hazardous. */
  public boolean isHazardous();
}
```

Then we can define a class for objects that are sellable and transportable by implementing these two interfaces. The following code defines such a class called *BoxedItem*:

Code Segment 1.20

```
/** Class for objects that can be sold, packed, and shipped. */
public class BoxedItem implements Sellable, Transportable {
  private String descript;      // description of this item
  private int price;            // list price in cents
  private int weight;           // weight in grams
  private boolean haz;          // true if object is hazardous
  private int height=0;         // box height
in centimeters
  private int width=0;          // box width in centimeters
  private int depth=0;          // box depth in centimeters
  /** Constructor */
  public BoxedItem(String desc, int p, int w, boolean h) {
    descript = desc;
    price = p;
    weight = w;
    haz = h;
  }
  public String description() { return descript; }
  public int listPrice() { return price; }
  public int lowestPrice() { return price/2;  }
  public int weight() { return weight; }
  public boolean isHazardous() { return haz; }
  public int insuredValue() { return price*2; }
  public void setBox(int h, int w, int d) {
    height = h;
    width = w;
    depth = d;
  }
}
```

In this example, the *BoxedItem* class implements both the *Sellable* interface and the *Transportable* interface. It implements all methods from the two interfaces and defines two additional methods *insuredValue* and *setBox*.

One of the features of the object-oriented paradigm is *multiple inheritance*, whereby a subclass inherits from two or more different superclasses (or a subclass has two or more superclasses). Java does not allow multiple inheritance through regular class hierarchy. In other words, a class cannot extend two or more classes. However, if multiple inheritance is necessary, it can be achieved using interfaces, as demonstrated by the above example of *BoxedItem*.

Section 1.2.2.2. Abstract Classes

An *abstract method* is a method that does not have an implementation. It has only a declaration. An abstract method is declared with a keyword *abstract,* as follows:

```
protected abstract void advance();
```

An abstract class is a class that is declared with *abstract*. It may or may not have abstract methods. If a class includes an abstract method, then it must be an abstract class. An abstract class can have variables and it can have *concrete methods*, which have implementations. An abstract class cannot be instantiated but it can be extended by subclasses. When a class extends an abstract class, usually it implements all abstract methods. If a subclass does not implement all abstract methods, then the subclass itself becomes an abstract class.

Note that methods in interfaces do not have implementation. So, they are implicitly abstract methods and are declared without the keyword *abstract*.

Abstract classes are similar to interfaces, but they are not the same. Usually we use abstract classes when we want to share code among related classes, or if the classes that extend a class have many common variables and methods. And we usually use interfaces when unrelated classes are expected to implement an interface, or if we want to have multiple inheritance.

We show an example of an abstract class by slightly modifying the *Progression* class. This abstract class is named *AbstractProgression*, and its code is shown below:

Code Segment 1.21

```
public abstract class AbstractProgression {
  protected long current;
  public AbstractProgression() { this(0); }
  public AbstractProgression(long start) { current = start; }
  public long nextValue() { // this is a concrete method
    long answer = current;
    advance();  // this protected call advances the current value
    return answer;
  }
  public void printProgression(int n) {    // this is a concrete method
    System.out.print(nextValue());     // print first value without leading space
    for (int j=1; j < n; j++)
13        System.out.print(" " + nextValue());  //
print leading space before others
14      System.out.println();                   //
end the line
15    }
16    protected abstract void advance();        //
notice the lack of a method body
17 }
```

The code is essentially the same as the code of the *Progression* class, except for two differences. First, the class is declared with the *abstract* modifier. Second, the *advance* method is declared as an abstract method without an implementation. A concrete implementation is provided in each subclass that extends the *AbstractProgression* class. It is implemented in the *ArithmeticProgression*, *GeometricProgression*, and *FibonacciProgression* subclasses, as shown in Section 1.2.1.3.

Note that, in line 7, the *advance* method is invoked even though there is no implementation in this class. This is allowed because once it is implemented in a subclass it will be well defined. This is an object-oriented design pattern called the *template method pattern*, whereby a concrete method in an abstract class invokes an abstract method.

Section 1.2.3. Exceptions

Oracle's Java documentation defines an exception as follows:

> An *exception*, which is shorthand for the phrase *exceptional event*, is an event that occurs during the execution of a program, which disrupts the normal flow of the program's instructions.

In this section, we discuss how to handle exceptional events in a Java program.

Section 1.2.3.1. Basic Concepts

When an exceptional event occurs in a method, an *exception object* is created and handed over to the runtime system, which is called *throwing an exception*.

An *exception* may refer to an *exception object* or *an exceptional event,* depending on the context in which it is used. Often we use these three terms interchangeably.

Once an exception is thrown, the runtime system tries to find an *exception handler*, which is a code segment that handles the exception, by searching the *call stack*. A call stack represents a sequence of method calls. For example, if *main* calls *method*1, *method*1 calls *method*2, and *method*2 calls *method*3, then the following is what the corresponding call stack looks like:

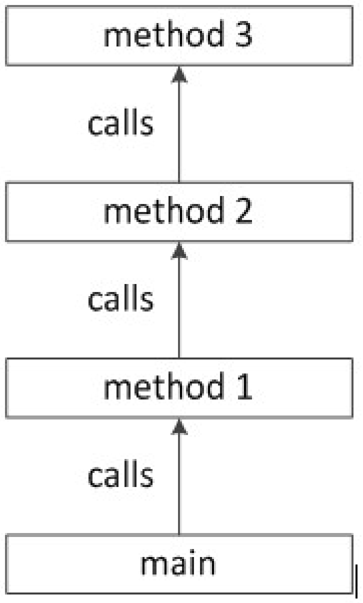


Figure 1.5. Call stack.

Suppose that an exception occurs in *method3*. The runtime system searches the call stack in the reverse order of method calls. It first searches *method3* for an exception handler. If it is found, then the handler will be executed. If not, it searches method2, and so on. The search path is shown in the following figure:

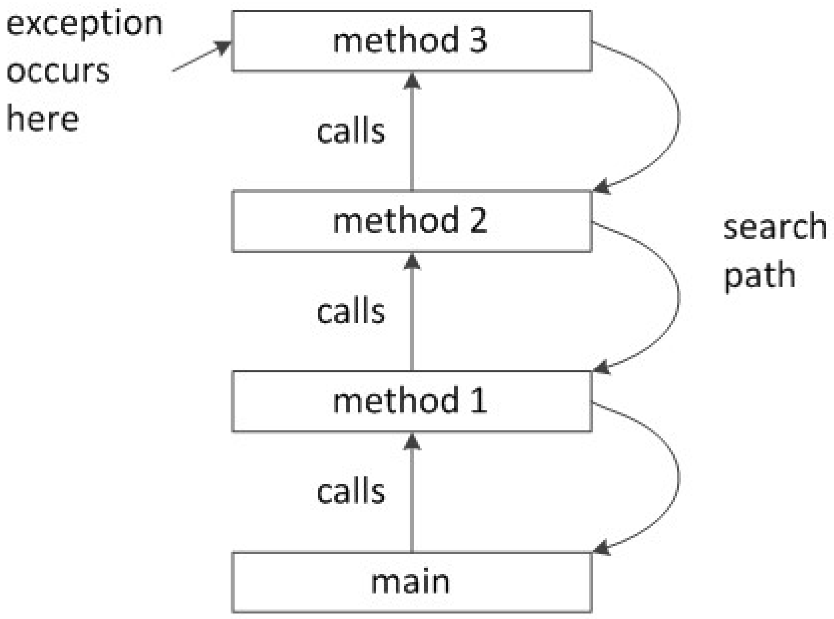


Figure 1.6. Exception-handler search path.

When an appropriate handler is found, we say the handler *caught* the exception. If the runtime system fails to find an appropriate exception handler, then the program terminates.

Section 1.2.3.2. Catching Exceptions and Try-Catch Statement

The exception-handling mechanism in Java consists of three blocks: *try*, *catch*, and *finally*. We will briefly describe the *finally* block at the end of this section; for now, we discuss only the *try* and *catch* blocks. A code that may throw an exception must be enclosed within a *try* block, and a code that handles (or catches) exceptions must be included in *catch* blocks. The general syntax of a try-catch statement is shown below:

```
try {
    guardedBody
} catch (exceptionType1 variable1) {
    remedyBody1
} catch (exceptionType1 variable2) {
    remedyBody2
} . . .
. . .
```

The runtime system first executes the code in the *try* block. If no exception occurs in the *try* block, all *try* blocks are skipped and the execution continues with the statement that immediately follows the entire *try-catch* statement. If an exception occurs, it is caught by the *catch* block with the exception type that matches the type of exception object thrown from the *catch* block. Then the control flow jumps to the *catch* block—i.e., the execution of the code in the *try* block is interrupted, and the code in the *catch* block is executed. After the execution of the *catch* block, the statement that immediately follows the *try-catch* statement is executed.

If more than one *catch* block matches the thrown object, the first of these is executed.

If an exception occurs and no matching *catch* block is found, the exception is *rethrown* to its surrounding context. Suppose that *method*1 invokes *method*2, and a *try-catch* statement is in *method*2. If an exception occurs in the *try* block and there is no matching *catch* block in *method*2, then the exception is passed to its caller *method*1. This is called the *rethrowing* of an exception.

When an exception is caught, it can be handled in several ways. An error message may be issued and the program terminated. An error message may be issued and the user prompted to decide what needs to be done next. A new exception may also be created with additional information, and then thrown.

An example of exception handling is shown below:

Code Segment 1.22

```
import java.util.Scanner;

public class ExceptionDemo {
  static final int DEFAULT = 100;
  public static void main(String[] args) {
    int n = DEFAULT;
    try {
      n = Integer.parseInt(args[0]);
      if (n <= 0) {
        System.out.println("n must be positive. Using default.");
        n = DEFAULT;
      }
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("No argument specified for n. Using default.");
    } catch (NumberFormatException e) {
      System.out.println("Invalid integer argument. Using default.");
    }
    System.out.println("n has value: " + n);
18   }
19 }
```

Line 7 inside the *try* block reads a command-line argument from the *args*[ ] array and it is converted to an integer by the *Integer.parsInt* method. This statement may throw one of two exceptions. If the user does not provide a command-line argument, the *args*[ ] array will be empty, and trying to access the first element with *args*[0] will throw the *ArrayIndexOutOfBoundsException* exception. If the argument is not an integer—for example 3.14—then the *NumberFormatException* exception will be thrown. These two exceptions are caught in the *catch* blocks of lines 12 and 14, respectively, and appropriate error messages are displayed. After one of the two *try* blocks is executed, line 17 prints *n* with the default value of 100.

If no exception is thrown from line 7, then the program checks whether the argument is a positive number in line 8. If it is, *try* blocks are skipped, and line 17 prints the value of *n,* which is entered by the user. If the argument is not a positive number, then an error message is displayed (line 9), *catch* blocks are skipped, and line 17 prints *n* with the default value of 100.

It is possible to catch and handle more than one type of exception in a single *catch* block. This can be done if the actions taken by multiple *catch* blocks are the same. This can reduce the duplication of code. To do this, we can specify multiple exception types, separated by vertical bars. An example is shown below:

```
} catch(ArrayIndexOutOfBoundsException |
NumberFormatException e) {
```

```
//catch block body
}
```

An optional *finally* block can be placed after the last *catch* block. The statements within the *finally* block are always executed regardless of what happens in the *try* block.

If no exception occurs, then after all statements in the *try* block are executed, the statements in the *finally* block are executed, followed by the first statement after the entire *try-catch-finally* statement.

If an exception occurs and it is caught by a *catch* block, the remaining statements in the *try* block are skipped, the statements within the *catch* block are executed and the statements in the *finally* block are executed. After that, the first statement after the entire *try-catch-finally* statement is executed.

If an exception occurs and it is not caught by any *catch* block, the remaining statements in the *try* block are skipped and the statements in the *finally* block are executed. After that, the exception is rethrown to the caller of the method.

Section 1.2.3.3. Throwing Exceptions

As discussed earlier, an exception must be thrown before it is handled. An exception can be thrown from any code. The syntax of throwing an exception is as follow:

```
throw new exceptionType(parameters);
```

Here, *exceptionType* is the type of the exception and *parameters* are passed to the type's constructor. An example of throwing an exception is shown below:

```
public void ensurePositive(int n) {
  if (n < 0)
    throw new IllegalArgumentException("That's not
positive!");
  // ...
}
```

This code checks whether the integer parameter *n* is positive or not. If it is not positive, it creates a new instance of the *IllegalArgumentException* object and throws it. Note that *IlligalArgumentException* is a subclass of *java.lang.Throwable* (which will be discussed in the next section). You can throw only an exception object that is an instantiation of an exception class that is a subclass of *Throwable*, unless you create your own exception classes.

If a method may throw an exception, you can include the potential exception(s) in the method's signature using the keyword *throws* to warn the users of the method about the possibility of the exception. The exception may be thrown by the *throw* statement in the method or it may be thrown indirectly from a method that is invoked by the method.

For example, the *parseInt* method in the *Integer* class may throw the *NumberFormatException*, and its signature includes the exception as follows:

```
public static int parseInt(String s) throws
NumberFormatException
```

If a method may throw one of many exceptions, you can include all these exceptions in the signature separated by commas. If, for example, a method *method*1 may throw *the IOException* and *IllegalAgumentException*, you can declare them as follows:

```
public static method1( ) throws IOException,
IllegalArgumentException
```

Section 1.2.3.4. Java's Exception Hierarchy

In Java, all exceptions classes are subclasses of the *java.lang.Throwable* class. The *Throwable* class has two subclasses: *Error* and *Exception*. A part of the Java exception hierarchy is shown below:

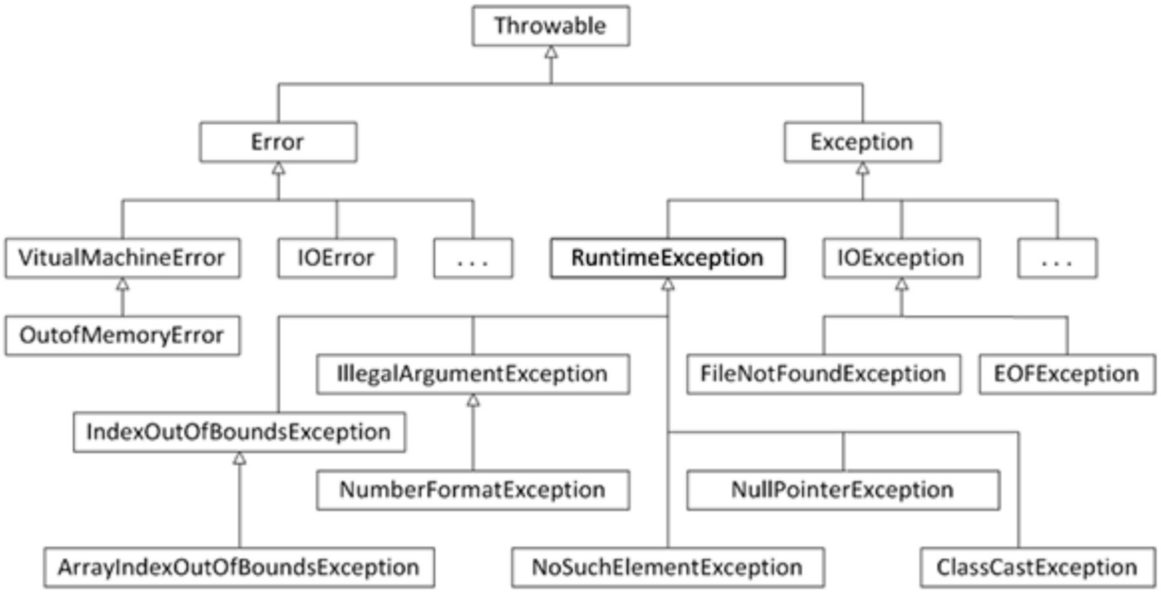


Figure 1.7. A part of Java exception hierarchy.

In general, there are three kinds of exceptions in Java—*errors*, *runtime exceptions*, and *checked exceptions*.

Errors are exception objects of the *Error* class and all of its subclasses. They are exceptional conditions that are external to the application, and they are thrown by JVM. When an *Error* exception is thrown, the application cannot be expected to recover from it; usually there is nothing an application can do except notify the user. Suppose that an application opens a file successfully and tries to read it, but due to some hardware failure, it cannot read the file. In this case, the *IOError* exception is thrown, which is a subclass of *Error*.

Runtime exceptions are exception objects of the *RuntimeException* class and all of its subclasses. They are exceptional events internal to the application, and the application usually cannot anticipate or recover from them. These usually indicate programming errors. The *IllegalArgumentException* is an example (and a subclass) of *RuntimeException*.

The *Error* and *RuntimeException* exceptions are called *unchecked* exceptions.

All other exceptions are *checked* exceptions. These are the exceptions that applications must anticipate and recover from. If a code may throw a *checked* exception, then it must be in a *try-catch* statement or it must be in a method that is declared with a *throws* clause.

Suppose that *method*1 calls *method*2, and *method*2 is declared with a checked exception (with *throws*). This means the call to *method*2 must be in a *try-catch* statement, or *method*1 must be declared with a *throws* clause.

Section 1.2.4. Casting

Casting refers to converting a variable (or a value) from one type to another. If a type is converted to a *wider* type, it is called *widening conversion*. If a type is converted to a *narrower* type, it is referred to as *narrowing conversion*. The widening conversion is performed automatically by Java, but the narrowing conversion must be done explicitly.

Casting of a type is done by specifying the target type in parentheses in front of the source variable (or value). The following example illustrates type casting of primitive types:

```
double x = 3.14
int a = (int)x; // narrowing conversion from
double to int
double y = a;   // widening conversion from int to
double
```

After executing this code segment, the value of *a* = 3, and the value of *y* = 3.0. Note that *x* is not changed—i.e., the value of *x* is still 3.14.

Type casting of objects is performed in the same way. Usually, a widening conversion occurs from type *T* to type *U* under the following circumstances:

- *U* is a superclass of *T*.
- *U* is a superinterface of *T*.
- *T* is a class that implements interface *U*.

Suppose we have a class named *Book,* and *Fiction* is a subclass of the *Book* class. Then we can create an object of the *Fiction* type and assign it to a reference variable of the *Book* type as follows:

```
Book b = new Fiction(parameters); // widening
conversion
```

A narrowing conversion from type *T* to type *S* occurs under the following circumstances:

- *S* is a subclass of *T*.
- *S* is a subinterface of *T*.
- *S* is a class that implements interface *T*.

The following example illustrates a narrowing conversion:

```
Book b = new Fiction(parameters); // widening
conversion
Fiction f = (Fiction) b; // narrowing conversion
```

As mentioned above, a narrowing conversion must be done by explicitly specifying the target (narrower) type.

When performing a narrowing conversion of an object *o* from type *T* to type *S*, if the actual type of the object *o* is not *S*, an exception occurs. Suppose that *Nonfiction* is also a subclass of the *Book* class, and we have an object *b* that is of *Book* type. The object *b* may be a fiction or a nonfiction. If *b* is of *Fiction* type, then we cannot convert *b* to an object of *Nonfiction* type. This is illustrated below:

```
Book b = new Fiction(parameters); // b is Fiction
type and Book type
Fiction f = (Fiction) b;       // legal narrowing
conversion
Nonfiction n = (Nonfiction) b; // illegal
narrowing conversion
```

To avoid making an illegal narrowing conversion, we can use the *instanceof* operator, which tests whether an object pointed to by a reference variable is of particular type. The above code segment can be rewritten using the *instanceof* operator as follows:

```
Book b = new Fiction(parameters); // b is Fiction
type and Book type
if (b instanceof Fiction)
Fiction f = (Fiction) b;       // legal narrowing
conversion
if (b instanceof Nonfiction)
  Nonfiction n = (Nonfiction) b; // legal but this
conversion will not occur
```

A similar situation may occur with interfaces. Suppose we have the *Person* interface, defined as follows:

Code Segment 1.23

```
public interface Person {
  public boolean equals(Person other);   // is
this the same person?
  public String getName();               // get
this person's name
  public int getAge();                   // get
this person's age
}
```

Consider the following *Student* class, which implements the *Person* class:

Code Segment 1.24

```
public class Student implements Person {
  String id;
  String name;
  int age;
  public Student(String i, String n, int a) {
// simple constructor
    id = i;
    name = n;
    age = a;
  }
  protected int studyHours() { return age/2;}
// just a guess
  public String getID() { return id;}
// ID of the student
  public String getName() { return name; }
// from Person interface
  public int getAge() { return age; }
// from Person interface
  public boolean equals(Person other) {
// from Person interface
    if (!(other instanceof Student)) return
false;  // cannot possibly be equal
    Student s = (Student) other;
// explicit cast now safe
    return id.equals(s.id);
// compare IDs
  }
19   public String toString() {
// for printing
20     return "Student(ID:" + id + ", Name:" +
name + ", Age:" + age + ")";
21   }
22 }
```

The *equals* method of lines 14 through 18 receives as its argument a *Person* object *other*, casts the type of *other* to *Student* type (line 16), and compares the *ids* of two students, returning true if two students have the same *id* in (line 17). If, however, the *other* object is not of the *Student* type, then the comparison cannot be done. So, line 15 tests whether the type of *other* is the *Student* type using *instanceof*. If it is true, it proceeds. Otherwise, it returns *false* (line 15).

Section 1.2.5. Generics

Generics allows you to use *types* (classes and interfaces) as parameters. When we use parameters in a method, we pass *values*. With generics, we pass *types* as parameters. This can be done not only with methods; we can also use *types* as parameters in classes and interfaces. In this section, we briefly review the generics in Java.

Section 1.2.5.1. Basic Concept

Consider the following small code example:

```
1  List list = new ArrayList();
2  list.add("hello");
```

```
3  String s = (String) list.get(0);
```

In this example, the *ArrayList* class stores a list of objects and implements the interface *List*.

This example does not use generics. Line 1 creates an object *list* of *List* type, and line 2 adds a string "hello" to *list*. Line 3 retrieves the first element of *list* using the *get* method, and the object is assigned to a String variable *s*. Since the type of the object returned by the *get* method is *Object*, an explicit type casting is needed. This type casting makes the code not only cluttered, but error prone, because a programmer may forget to perform the type casting.

The same code can be rewritten using generics.

```
1  List<String>list = new ArrayList<String>();
2  list.add("hello");
3  String s = list.get(0);   // no cast
```

In this example, the type of object stored in *list* is specified as *String* (in a pair of angle brackets). Then it is not necessary to cast the type in line 3.

One important advantage of generics is *strong type checking*. Since a type is specified for the objects in *list*, a compiler can check the correctness of the type during compile time. So, strong type checking is provided.

Suppose that we want to define a class that stores a pair of objects. Further, we want to define a *generic* class that can store a pair of objects of arbitrary types. For example, we want to be able to implement a class storing a pair of a *String* object and a *Double* object, or a class storing a pair of a *Student* object and an *Integer* object, and so on. This can be done without using generics. We can use the *Object* class to define such a generic class. The following code shows the *ObjectPair* class definition.

Code Segment 1.25

```
1  public class ObjectPair {
2    Object first;
3    Object second;
4    public ObjectPair(Object a, Object b) {
// constructor
5      first = a;
6      second = b;
7    }
8    public Object getFirst() { return first; }
9    public Object getSecond() { return second;}
10   public static void main(String[] args) {
11     ObjectPair bid = new ObjectPair("ORCL",
32.07);
12     /*
13     String stock = bid.getFirst();           //
illegal; compile error
14     */
15     String stock = (String) bid.getFirst();  //
narrowing cast: Object to String
16   }
17 }
```

The constructor in line 4 receives two arguments *a* and *b*. The types of both arguments are *Object*, and two *get* methods in lines 8 and 9 return objects of *Object* type. In line 11, an object *bid* of *the ObjectPair* class is created by invoking the constructor with two actual parameters, "ORCL" and 32.07. The first parameter is a String and the second one is Double, each of which is automatically converted to the formal parameter types of *Object*. So, there is no problem here. However, if we try to retrieve the first element of the object *bid* with the *getFirst* method in line 13, we will get a compiler error. This is because the *getFirst* method returns an object of the *Object* type, and the type of the variable *stock* on the left-hand side is *String*, which requires a *narrowing type conversion*. As we discussed earlier, a narrowing conversion requires an explicit casting, which is not done in line 13. In line 15, an explicit type casting is applied and, in this case, there is no error.

The following code defines the same class, but using generics:

Code Segment 1.26

```
1  public class Pair <A,B>{
2    A first;
3    B second;
4    public Pair(A a, B b) {                  //
constructor
5      first = a;
6      second = b;
7    }
8    public A getFirst() { return first; }
9    public B getSecond() { return second;}
10   }
11   public static void main(String[] args) {
```

```
12     Pair<String,Double> bid;
13     bid = new Pair<>("ORCL", 32.07);
// rely on type inference
14     String stock = bid.getFirst();
15     double price = bid.getSecond();
16     System.out.println("bid = " + bid);
17   }
18 }
```

In line 1, the *Pair* class is declared with two formal type parameters, *A* and *B,* in angle brackets. By convention, we use a single uppercase letter for a formal type parameter. The constructor in line 4 uses these formal type parameters and they are also used as return types of two *get* methods in lines 8 and 9.

When we declare a variable of the *Pair* class, we need to specify the actual type parameters, as shown in line 12. In this example, a variable *bid* is declared; and the type of the first parameter is *String*, and that of the second parameter is *Double*. Note that the types of actual parameters for generics must be object types, not primitive types. So, *Double* is used instead of *double*. Then, an object of the *Pair* class can be instantiated as shown in line 13. Notice that, after the name of the class *Pair*, there is an empty pair of angle brackets (called *diamond*) before actual parameters in parentheses. This feature is called *type inference* because the system *infers* the types of actual parameters based on the declaration of the variable (which was done in line 12). The *type inference* was introduced in Java SE 7.

Before Java SE 7, the actual parameter types had to be explicitly specified, and we can still use this this style, as shown below:

```
bid = new Pair<String,Double>("ORCL", 32.07);
```

Here, the types *String* and *Double* are explicitly specified.

It is also possible to use the classic style without using angle brackets, as follows:

```
bid = new Pair("MS", 32.07);
```

In this case, the types of two parameters are automatically converted to *Object*, and a warning message will be issued by a compiler. So, we should avoid using this classic style.

Lines 14 and 15 retrieve two elements of the pair using *get* methods and, as discussed earlier, explicit type casting is not used here.

Section 1.2.5.2. Generics and Arrays

One of the restrictions on Java generics is that Java does not allow the instantiation of an array with parameterized types. In the following two situations, we can get around this restriction:

- We have a generic class with parameterized types. We want to declare, outside of the generic class, an array storing objects of the generic class with actual type parameters.
- We have a generic class with parameterized types. We want to declare, as an instance variable of the class, an array storing objects of one of the formal parameter types.

The following example illustrates the first situation:

```
1  Pair<String,Double>[] holdings; // declaring
```

```
with actual type
                                // type
parameters are allowed
2  holdings = new Pair<String,Double>[25]; //
illegal
3  holdings = new Pair[25];          // this is
allowed
4  holdings[0] = new Pair<>("ORCL", 3.14); // this
assignment is legal
```

Line 1 declares an array *holdings*, the elements of which are of *Pair*<*String*, *Double*> type. Recall that *Pair* is a generic class with two formal type parameters *A* and *B*, which was used as an example to illustrate the concept of Java generics. Here, two actual type parameters, *String* and *Double*, are specified when *holdings* is declared. This is allowed. However, we cannot instantiate the array *holdings* with parameterized types. So, line 2 is illegal. We have to instantiate with an unparameterized type, as shown in line 3. Once an array is instantiated, we can create a *Pair* object with empty pair of brackets and actual parameter values, and assign it to an array element.

For the case of the second situation, let’s consider the following code example:

Code Segment 1.27

```
1  public class Portfolio<T> {
2    T[] data;
3    public Portfolio(int capacity) {
```

```
4      /*
5      data = new T[capacity];              // illegal; compiler error
6      */
7      data = (T[]) new Object[capacity];   // legal, but compiler warning
8    }
9    public T get(int index) { return data[index]; }
10   public void set(int index, T element) { data[index] = element; }
11 }
```

The *Portfolio* class is a generic class with a formal type parameter *T*. An instance variable *data* in this class is an array of type *T* (line 2). When the array *data* is declared, we cannot directly instantiate the array. Instead, we create an array of *Object* type and type cast to *T*[ ], as shown in line 7.

Section 1.2.5.3. Generic Method

The concept of generics can also be applied to methods, in which the formal type parameters can be used as arguments.

Consider the following example:

Code Segment 1.28

```
public class GenericDemo {
  public static <T> void reverse(T[] data) {
    int low = 0, high = data.length - 1;
4      while (low < high) {                    //
swap data[low] and data[high]
5        T temp = data[low];
6        data[low++] = data[high];             //
post-increment of low
7        data[high--] = temp;                  //
post-decrement of high
8      }
9    }
10 }
```

The *reverse* method receives an array of elements of an arbitrary type, and reverses the order of elements in the array. Note that the declaration of the method includes <*T*> after the *static* keyword. The formal type parameter *T* is used as an argument (line 2) and within the method body when declaring a local variable (line 5). When another method invokes the *reverse* method, an actual parameter type will be passed and will replace *T* in the method. This method can be invoked using *GenericDemo.reverse*, as shown in the following example:

```
String[]  names = new String[]{"john", "susan",
"molly"};
System.out.print("names:  ");
for  (int i=0; i<names.length; i++){
System.out.print(names[i] + "  ");
}

GenericDemo.reverse(names);
System.out.print("\nnames  reversed: ");
```

```
for  (int i=0; i<names.length; i++){
System.out.print(names[i] + "  ");
}
```

The expected output is as follows:

```
names: john susan molly
names reversed: molly susan john
```

Section 1.2.5.4. Bounded Generic Types

When a formal type parameter *T* is used in a generic class or method, any object type can be used as an actual type parameter. If we want, we can restrict the type of actual parameter to subclasses of a certain type. To do this, we use the keyword *extends*, followed by a class or interface.

Suppose that we want to design a *ShoppingCart* class that can only be instantiated with a type that implements the *Sellable* interface (in Section 1.2.2.1). Then we can define a generic class as follows:

```
public class ShoppingCart<T extends Sellable>{ . .
. }
```

Once an object of type *T* is instantiated, any method defined in *Sellable* (which is implemented in *T*) can be invoked on the object.

Section 1.2.6. Nested Classes

In Java, you can define a class within another class, as illustrated below:

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

We use nested classes for the following reasons:

1. The *NestedClass* is used only for the *OuterClass*.
2. We want to declare members of the *OuterClass* as private but, at the same time, we want the *NestedClass* to be able to access members of the *OuterClass*.
3. The code becomes more readable and is easy to maintain.

Nested classes are also used when we want to implement a data structure that has another data structure as its member. Nested classes also help reduce name conflict.

Suppose that we have a *CreditCard* class (shown in Section 1.1.9) and we want to keep the record of all transactions by this credit card. Then, we can create a *Transaction* class as a nested class, as shown below:

```
public class CreditCard {
    private static class Transaction { . . . }

    // instance variable to keep the record of
transactions
    Transaction[ ] history;
}
```

A nested transaction can be referenced with its name qualified by the name of the Outer class. For example, we can use *CreditCard.Transaction* to reference the above nested class.

A nested class can be *static* or *nonstatic*. Nonstatic nested classes are called *inner classes*.

A static nested class is associated with the outer *class* and behaves like a traditional, top-level class. It cannot directly access instance variables and methods in the outer class. It can access those variables only through object references (like any other top-level classes do).

An inner class (nonstatic nested class) is associated with an *instance* of the outer class. So, it can directly access the variables and methods of the outer class. An instance of an inner class can be created only after you first create an object of the outer class. Then, you can create an inner object from the outer object within a nonstatic method of the outer class.

Section 1.3. Fundamental Data Structures
In this section, we discuss two fundamental data structures: arrays and linked lists. More complex data structures will be discussed later.

Section 1.3.1. Arrays

Arrays were briefly mentioned in Section 1.1.6. In this section, we illustrate how to create and manipulate an array that stores objects. Each object represents the name and the score of a video-game player, and each element is a reference to such an object. The array satisfies the following property:

- Elements are sorted in decreasing order of scores.

- Elements are stored in consecutive array slots.
- Suppose that the array is full and a new element (player) is being inserted. Then the new element is kept in the array only if the score of the new element is higher than the lowest score in the current array.

First, we define a class that represents players.

Code Segment 1.29

```
public class GameEntry {
  private String name;        // name of the person earning this score
  private int score;          // the score value
  /** Constructs a game entry with given parameters.. */
  public GameEntry(String n, int s) {
    name = n;
    score = s;
  }
  /** Returns the name field. */
  public String getName() { return name; }
  /** Returns the score field. */
  public int getScore() { return score; }
  /** Returns a string representation of this entry. */
  public String toString() {
    return "(" + name + ", " + score + ")";
  }
}
```

In addition to two instance variables, *name* and *score*, and a constructor, the *GameEntry* class also has the *getName*( ) method, *getScore*( ) method, and *toString*( ) method.

The *toString*( ) method returns the string representation of the object. It is defined in the *Object* class and overridden in the *GameEntry* class to return a string including the name and the score of a player.

Next, we define the *Scoreboard* class, which represents the array of *GameEntry* objects.

Code Segment 1.30

```
/** Class for storing high scores in an array in nondecreasing order. */
public class Scoreboard {
  private int numEntries = 0;    // number of actual entries
  private GameEntry[] board;     // array of game entries (names & scores)
  /** Constructs an empty scoreboard with the given capacity */
  public Scoreboard(int capacity) {
    board = new GameEntry[capacity];
  }
   . . .     // more methods, shown separately
}
```

The *Scoreboard* class has two instance variables. The variable *board* is an array of *GameEntry* objects. The variable *numEntries* stores the number of entries that are currently stored in the array. The array has a fixed size. So, *numEntry* is always smaller than or equal to the array size.

The constructor in lines 6 and 7 creates an empty array with the given *capacity* (input argument). Initially, all elements are *null*.

In this implementation there are three more methods that are not shown in the above code. They are the *add* method, *remove* method, and *main* method.

The *add* method inserts a new element into the array. If the array is not full, then the new element will be inserted into the appropriate position in the array (remember that array elements are sorted). If the array is full, there are two possible cases: If the score of the new element is lower than or equal to the lowest score in the array, the new element is not inserted. If the score of the new element is higher than the lowest score in the array, the element with the lowest score is removed from the array, and the new element is inserted in the appropriate position. The code of the *add* method is shown below:

Code Segment 1.31

```
public void add(GameEntry e) {
  int newScore = e.getScore();
  // is the new entry e really a high score?
  if (numEntries < board.length ||
          newScore > board[numEntries-
1].getScore()) {
6      if (numEntries < board.length)        // no
score drops from the board
7        numEntries++;                        // so
overall number increases
8      // shift any lower scores rightward to make
room for the new entry
9      int j = numEntries - 1;
10     while (j > 0 && board[j-1].getScore() <
newScore) {
11       board[j] = board[j-1];       // shift
entry from j-1 to j
12       j--;                         // and
decrement j
13     }
14     board[j] = e;                          //
when done, add new entry
15   }
16 }
```

Lines 4 and 5 check whether a new element needs to be inserted. A new element will be inserted if the array is not full (line 4) or the score of the new element is higher than the lowest score in the array. If this is the case, then the number of entries, *numEntries*, is incremented (line 7) and then, in lines 9 through 14, the new element is inserted in the right position.

The *remove* method receives, as an argument, the index of an element. The then removes the element pointed to by the index and shifts all the elements with lower scores upward. The code also checks that the index is not outside the bounds. The code is given below:

Code Segment 1.32

```
1  public GameEntry remove(int i) throws IndexOutOfBoundsException {
2    if (i < 0 || i >= numEntries)
3      throw new IndexOutOfBoundsException("Invalid index: " + i);
4    GameEntry temp = board[i];         // save the object to be removed
5    for (int j = i; j < numEntries - 1; j++)   // count up from i (not down)
6      board[j] = board[j+1];                   // move one cell to the left
7    board[numEntries -1 ] = null;              // null out the old last score
8    numEntries--;
9    return temp;                               // return the removed object
10 }
```

If the index *i* is smaller than 0 or larger than or equal to the number of entries—i.e. it is not pointing to an element that is currently in the array—an exception is thrown (lines 2 and 3). Otherwise, the element is removed (and saved to be returned) in line 4, elements with lower scores are moved upward in lines 5 and 6, and the removed element is returned in line 9.

The following *main* method illustrates the insertion and removal of elements:

Code Segment 1.33

```
public static void main(String[] args) {
     // The main method
     Scoreboard highscores = new Scoreboard(4);
     String[] names = {"Rob", "Mike", "Rose", "Jill", "Jack", "Anna", "Paul"};
     int[] scores = {750, 1105, 590, 740, 510, 860, 740};
     for (int i=0; i < names.length; i++) {
       GameEntry gE = new GameEntry(names[i], scores[i]);
      System.out.println("Adding " + gE);
       highscores.add(gE);
      System.out.println(" Scoreboard: " + highscores);
     }
     System.out.println("Removing score at index " + 2);
     highscores.remove(2);
     System.out.println(highscores);
     System.out.println("Removing score at index " + 0);
     highscores.remove(0);
     System.out.println(highscores);
     System.out.println("Removing score at index " + 1);
```

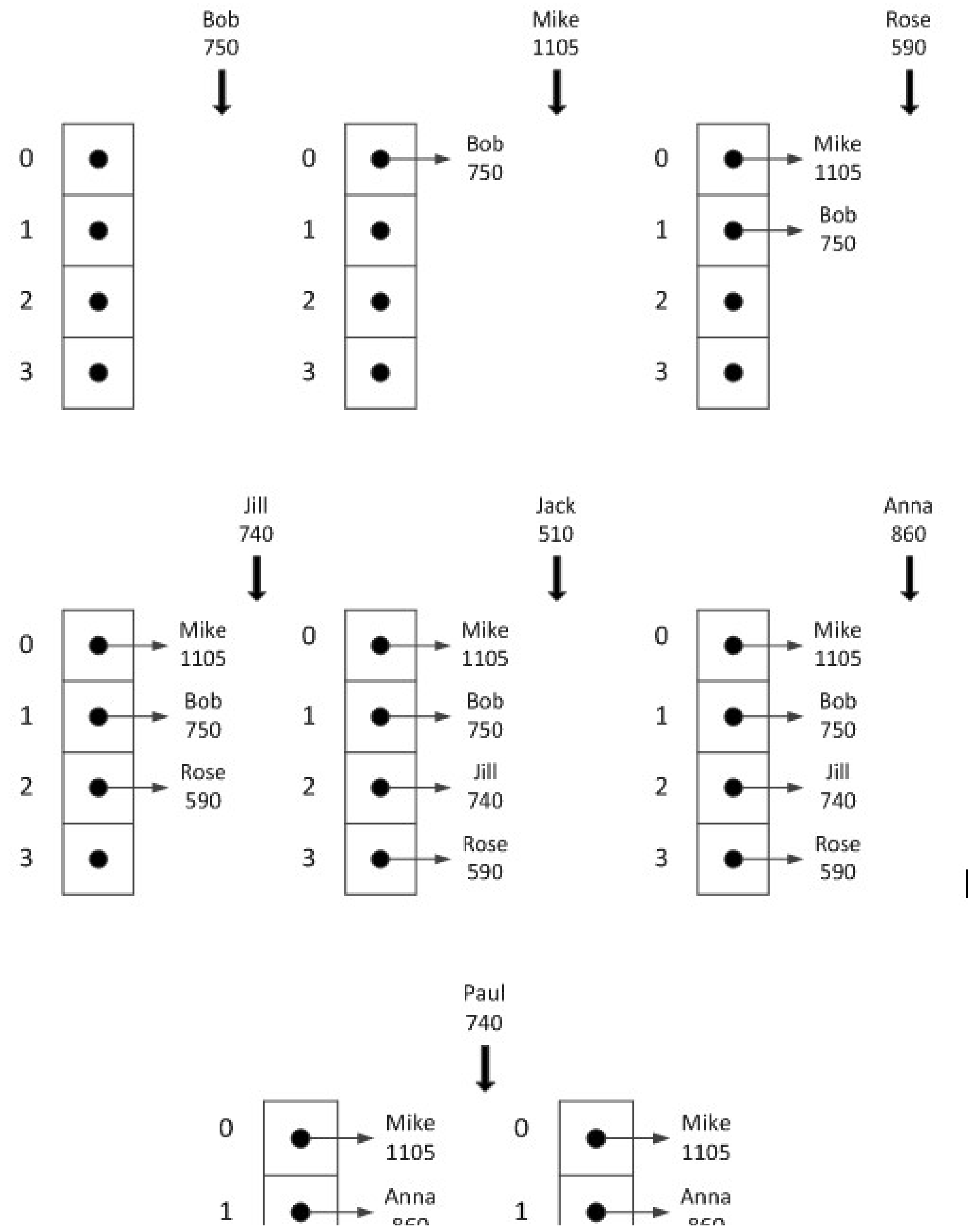
```
        highscores.remove(1);
        System.out.println(highscores);
        System.out.println("Removing score at index " + 0);
        highscores.remove(0);
        System.out.println(highscores);
    }
```

The expected output is:

```
Adding (Rob, 750)
 Scoreboard: [(Rob, 750)]
Adding (Mike, 1105)
 Scoreboard: [(Mike, 1105), (Rob, 750)]
Adding (Rose, 590)
 Scoreboard: [(Mike, 1105), (Rob, 750), (Rose, 590)]
Adding (Jill, 740)
 Scoreboard: [(Mike, 1105), (Rob, 750), (Jill, 740), (Rose, 590)]
Adding (Jack, 510)
 Scoreboard: [(Mike, 1105), (Rob, 750), (Jill, 740), (Rose, 590)]
Adding (Anna, 860)
 Scoreboard: [(Mike, 1105), (Anna, 860), (Rob, 750), (Jill, 740)]
Adding (Paul, 740)
 Scoreboard: [(Mike, 1105), (Anna, 860), (Rob, 750), (Jill, 740)]
Removing score at index 2
```

```
[(Mike, 1105), (Anna, 860), (Jill, 740)]
Removing score at index 0
[(Anna, 860), (Jill, 740)]
Removing score at index 1
[(Anna, 860)]
Removing score at index 0
[]
```

The change of the content of *the highscores* array is illustrated below:



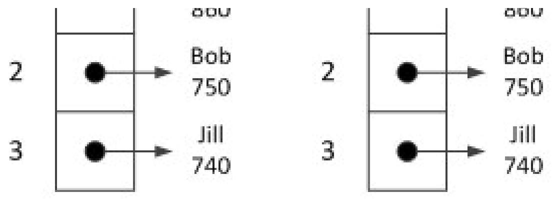


Figure 1.8. Adding *GameEntry* objects to the *highscores* array.

Section 1.3.2. Sorting an Array

Sorting is one of the most fundamental algorithms in computer programming. Sorting rearranges a given sequence of elements in ascending or descending order. There are different sorting algorithms. In this section, we discuss one simple sorting algorithm called *insertion-sort*. We will discuss other sorting algorithms later.

A pseudocode of the insertion-sort algorithm is shown below:

```
Algorithm InsertionSort(A)
    Input: Array A of n comparable elements
    Output: Array A with elements rearranged in
dondecreasing order
    for k from 1 to n - 1 do
        Insert A[k] at its proper location within
A[0 … k]
```

Here, *A*[0 … *k*] is a subarray with elements *A*[0], *A*[1], … *A*[*k*].

An implementation of the pseudocode is shown below. The *main* method is for testing.

Code Segment 1.34

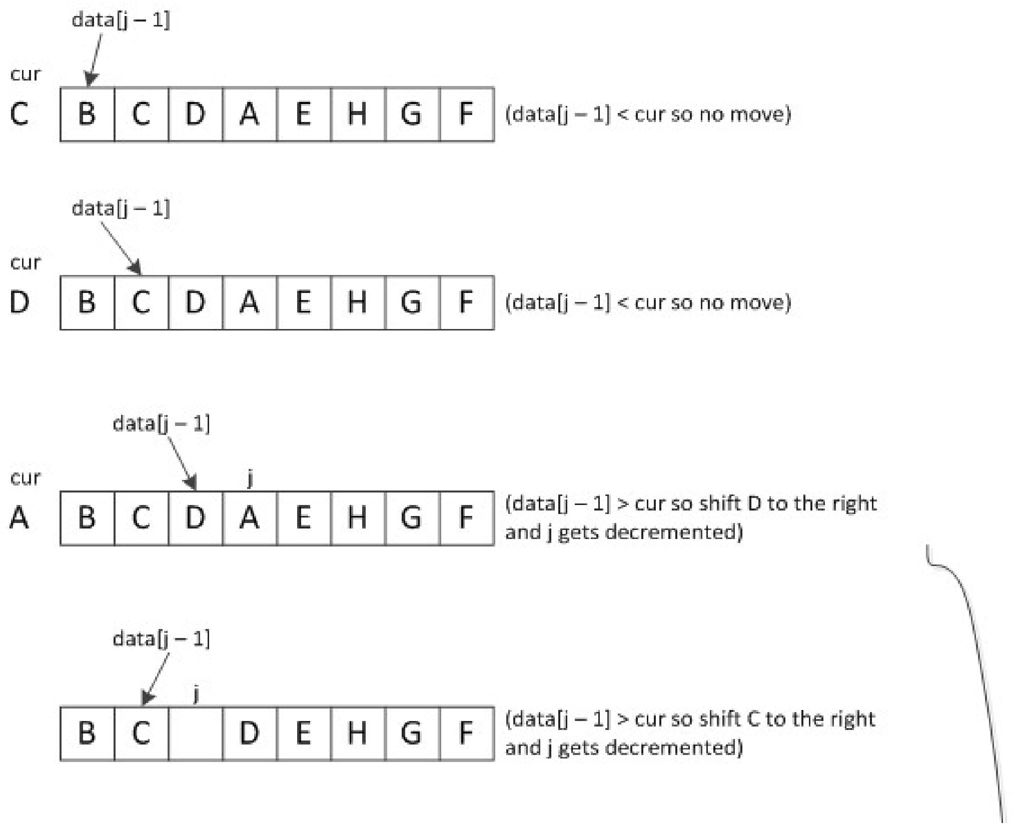
```
public class InsertionSort {
  /** Insertion-sort of an array of characters into nondecreasing order */
  public static void insertionSort(char[] data) {
    int n = data.length;
    for (int k = 1; k < n; k++) {             // begin with second character
      char cur = data[k];                     // time to insert cur=data[k]
      int j = k;                              // find correct index j for cur
      while (j > 0 && data[j-1] > cur) {      // thus, data[j-1] must go after cur
        data[j] = data[j-1];                  // slide data[j-1] rightward
        j--;                                  // and consider previous j for cur
      }
      data[j] = cur;                          // this is the proper place for cur
    }
  }

  public static void main(String[] args) {
    char[] a = {'B', 'C', 'D', 'A', 'E', 'H', 'G', 'F'};
    System.out.println(java.util.Arrays.toString(a));
    insertionSort(a);
```

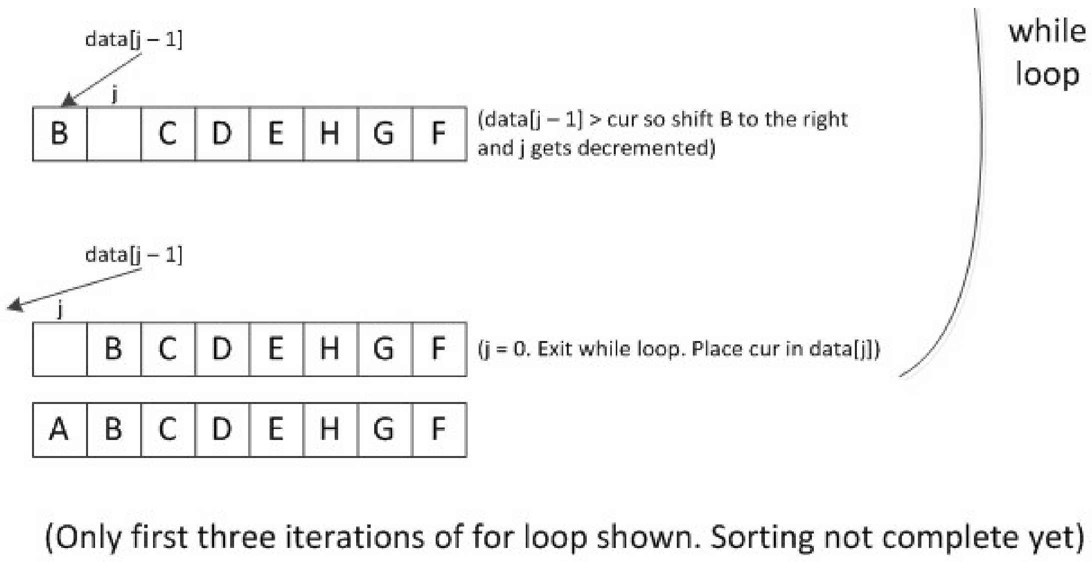
```
19
System.out.println(java.util.Arrays.toString(a));
20  }
21 }
```

The *for* loop in line 5 scans elements to the right, beginning with the second element (*k* = 1). In each iteration, *data*[*k*] is moved to the right position in the *while* loop of line 8.

In the while loop, the index *j* is used as a pointer that scans the elements to the left of *data*[*k*], or *cur*. Initially, *j* is set to *k*. In each iteration, *cur* is compared with *data*[*j* – 1], and if *data*[*j* – 1] is larger than *cur*, it is shifted to the right (line 9). Then *j* gets decremented (line 10), and the same is repeated until *data*[*j* – 1] is smaller than or equal to *cur,* or *j* becomes 0. Once this occurs, *data*[*j*] is the correct position of *cur* (line 12).

The Figure 1.9 illustrates the first three iterations of the *for* loop, the third includes three iterations of the *while* loop:


data[j – 1]
cur
C
B C D A E H G F
(data[j – 1] < cur so no move)
data[j – 1]
cur
D
B C D A E H G F
(data[j – 1] < cur so no move)
data[j – 1]
cur
j
A
B C D A E H G F
(data[j – 1] > cur so shift D to the right and j gets decremented)
data[j – 1]
j
B C D E H G F
(data[j – 1] > cur so shift C to the right and j gets decremented)



data[j – 1]
j
B | | C | D | E | H | G | F
(data[j – 1] > cur so shift B to the right and j gets decremented)
data[j – 1]
j
| B | C | D | E | H | G | F
(j = 0. Exit while loop. Place cur in data[j])
A | B | C | D | E | H | G | F
while loop
(Only first three iterations of for loop shown. Sorting not complete yet)

Figure 1.9. Illustration of insertion sort.

Section 1.3.3. Arrays Class and Methods

Java provides the *Arrays* class, which is defined in the *java.util* package, to support efficient manipulation of arrays. Some methods defined in the *Arrays* class and a code that illustrates those methods are shown below:

```
equals(A, B)—Returns true if A and B are
identical, and false otherwise.
fill (A, x)—Fills (all elements of) A with x.
copyOf(A, n)—Copies A, truncating or padding with
zeros (if necessary) so that the
              copy has n elements.
copyOfRange(A, s, t)—Copies the specified range of
A, A[s] to A[t-1], into a
              new array.
toString(A)—Returns a string representation of A.
sort(A)—Sorts A into ascending order.
binarySearch(A, x)—Searches A for x using the
```

```
binary search algorithm and
              returns the index where it is found.
If x is not in A, returns
              the position where it could be
inserted.
```

Code Segment 1.35

```
import java.util.Arrays;

public class ArraysTest {
  public static void main(String[] args) {
    int[] A = {1, 2, 3, 4, 5};

    int[] copyOfA = Arrays.copyOf(A,
A.length);

    System.out.println("Array A: " +
Arrays.toString(A));
    System.out.println("Array copyOfA: " +
Arrays.toString(copyOfA));

    System.out.println("Array A equals Array
copyof A is " +
    Arrays.equals(A, copyOfA));
    Arrays.fill(copyOfA, 10);
    System.out.println("Array copyOfA after
filling with 10: " +
Arrays.toString(copyOfA));
      System.out.println("Array A equals Array
copyof A is " +
      Arrays.equals(A, copyOfA));

      int[] C = Arrays.copyOfRange(A,  1, 4);
      System.out.println("Array C: " +
Arrays.toString(C));

      int position = Arrays.binarySearch(A, 3);
      System.out.println("Index of 3 in Array A
is: " + position);

      Arrays.sort(A);
      System.out.println("Array A: " +
Arrays.toString(A));
   }
}
```

The following is the output:

```
Array A: [1, 2, 3, 4, 5]
Array copyOfA: [1, 2, 3, 4, 5]
Array A equals Array copyof A is true
Array copyOfA after filling with 10: [10, 10, 10,
10, 10]
Array A equals Array copyof A is false
Array C: [2, 3, 4]
Index of the element 3 in Array A is: 2
Array A: [1, 2, 3, 4, 5]
```

Section 1.3.4. PseudoRandom Numbers

In some applications, such as simulation and game programs, it is often necessary to generate random numbers. Most high-level programming languages have built-in pseudorandom number generators. These random numbers are called *pseudo* because they are not truly random in the sense that the next number is predictable from the current number. For example, the next pseudorandom number can be calculated from the current pseudorandom number using the following formula:

```
next = (a * current + b) % n
```

Here, *a*, *b*, and *n* are integer constants, and % is the modulus operator. Initially, a *seed* number is used to generate the first pseudorandom number using the above formula, the second pseudorandom number is generated from the first pseudorandom number, and so on. If the same seed is used, then the sequence of numbers generated from the seed is always the same. So, if needed, you must change the seed each time you generate pseudorandom numbers. One way to do this is to use the current time in milliseconds (which can be obtained using *Syste.currentTimeMillis* in Java) to initialize a seed, which is different each time pseudorandom numbers are generated.

In Java, the *Random* class is defined in the *java.util* package. An instance of the *Random* class is used to generate pseudorandom numbers.

Some of the methods defined in the *Random* class are shown below:

`nextBoolean()`—Returns the next pseudorandom ***boolean value.***

`nextDouble()`—Returns the next pseudorandom ***double value between 0.0 and 1.0***.

`nextInt()`—Returns the next pseudorandom ***int value.***

`nextInt(n)`—Returns the next pseudorandom ***int value from 0 to n, not including***.

`setSeed(s)`—Sets the seed of this pseudorandom number generator to *long s*.

The following example illustrates the generation of pseudorandom numbers:

Code Segment 1.36

```
import java.util.Arrays;
import java.util.Random;

public class ArrayTest {
  public static void main(String[] args) {
    int data[] = new int[10];
    Random rand = new Random();
    rand.setSeed(System.currentTimeMillis());
    for (int i = 0; i < data.length; i++)
      data[i] = rand.nextInt(100);
    int[] orig = Arrays.copyOf(data,
data.length);
    System.out.println("arrays equal before
sort: " +
12
Arrays.equals(data, orig));
13     Arrays.sort(data);
14     System.out.println("arrays equal after
sort: " +
15                            Arrays.equals(data,
orig));
16     System.out.println("orig = " +
Arrays.toString(orig));
17     System.out.println("data = " +
Arrays.toString(data));
18   }
19  }
```

A pseudorandom-number generator *rand* is created in line 6 and the seed set in line 7. In the *for* loop of line 8, 10 pseudorandom numbers are generated in placed in the array *data*. A new array *orig* is created and is a copy of the array *data* in line 10. The two are compared in line 11 and then again after *data* is sorted.

A sample output follows:

```
arrays equal before sort: true
arrays equal after sort: false
orig = [49, 30, 70, 53, 38, 14, 85, 57, 53, 74]
data = [14, 30, 38, 49, 53, 53, 57, 70, 74, 85]
```

Another run of the program will show different pseudo random numbers:

```
arrays equal before sort: true
arrays equal after sort: false
orig = [1, 58, 57, 77, 96, 38, 38, 29, 74, 29]
data = [1, 29, 29, 38, 38, 57, 58, 74, 77, 96]
```

Section 1.3.5. Simple Cryptography

Cryptography is a science of converting a message in such a way that it may be stored or transmitted securely. It usually involves converting a message to a secret message and then converting the secret message back to the original.

An encryption process involves converting an original message, called *plaintext*, to a scrambled message called *ciphertext*. Decryption is the reverse process of converting a ciphertext back to the original plaintext.

One of the simplest and earliest cryptosystems (or *cipher*s) is the *Caesar cipher* (or *Caesar's cipher*), which Caesar used around 100 BC to send secret messages to his generals. In the Caesar cipher, each letter in a message is encrypted to the letter that appears a certain number of letters after it in the alphabet. Using English as an example, if the distance between the original letter and the encrypted letter is 3, *A* is replaced with *D*, *B* is replaced with *E*, and so on. Continuing this, *W* is replaced with *Z*. After that, encryption is done in a *wrap around* way. So, *X* is replaced with *A*, *Y* with *B*, and *Z* with *C*. We can consider replacing a character with the character that is *r* characters away from the original character as *shifting* the original character by *r*. In what follows, this *r* will be referred to as the *rotational shift*. (The reason the term *rotational* is used is, as we will see shortly, characters are rotated around an array of characters.)

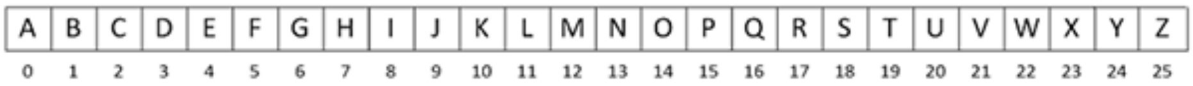
It is relatively simple to implement the Caesar cipher in Java. It is assumed that both the plaintext and the ciphertext are represented as strings. We can convert the plaintext string to an array of characters, create an encrypted array of characters by shifting each character by a certain number of positions in the alphabet, and convert it back to a string, which is the ciphertext string.

We can use the *toCharArray*( ) method, which is defined in the *String* class, to convert a string to an array of characters. For example, a string *s* = "cipher" will be converted to an array *A* = {'c', 'y', 'p', 'h', 'e', 'r'} by *A* = *s.toCharString*( ).

A character array can be converted to a string using one of constructors of the *String* class. This constructor receives as an argument a character array and creates a new string. So, if *A* is a character array {'c', 'y', 'p', 'h', 'e', 'r'} and we execute the statement *Sting s* = *new String(A)*, then a new string *s* = "cipher" will be created.

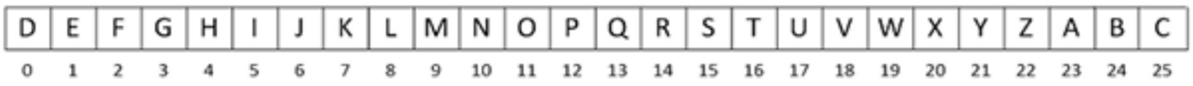
Shifting a character to the encrypted character is also relatively easy because characters are represented as integer *code points*. Because of this, we can perform a subtraction operation on two characters, and the difference is an integer that represents the number of characters by which two characters are separated in the alphabet. For example, A – A gives us 0 and D – A gives us 3. We can also add an integer to a character to obtain another character. For example, we can add 3 to A to obtain D.

To make the discussion simple, we assume that all characters in the text are English uppercase letters. One way of implementing the Caesar cipher is to use a precomputed *encoder array*. First, we assume an array of 26 upper-case letters, as shown below, which we will call the *initial array*:



A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Then, if the rotational shiftis is 3, we construct another array—called *encoder array* —for this shift, by shifting all characters to the left by three positions. The characters that are pushed out from the left (A, B, and C) go the right of the array in a *wrap around* way, as shown below:



D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

The encoder array can be constructed in the following way. Suppose that *r* is the rotation shift and *k* is the index in the initial array of the letter to be encrypted. In the slot of index *k* in the encoder array, we can put the letter with index (*k* + *r*) % 26 in the initial array. For example, to determine the letter in the slot of index *k* = 10 in the encoder array, we first calculate (10 + 3) % 26 = 13. Then we use the index 13 in the initial array to find the letter *N*. Suppose we want to determine the letter in the slot of index *k* = 24: (24 + 3) % 26 = 1. So, the letter B will go into that slot in the encoder array.

Implementation is done slightly differently. Let's call (*k* + *r*) % 26 *offset*. Instead of using the initial array, which was used above to illustrate how an encoder array can be constructed, we determine the letter that will be placed in the encoder array by adding the offset to the character A. For example, if we add 3 to the character *A* we will get the character *D* (which was shown as an example earlier).

The decoder array can be constructed in a similar way. We can use (*k* – *r* + 26) % 26 as an index to find the letter in the initial array, and place it in the slot with index *k* in the decoder array. Here, it is necessary to add 26 to *k* – *r* because the result of the subtraction can be a negative number. Again, when implementing, the offset is added to the character *A* to determine the letter that will go into the decoder array.

To encrypt a letter in the plaintext, we find the index of the letter in the initial array. This is done by subtracting the letter *A* from that letter. Then, we use the result of the subtraction as the index of the encrypted letter in the encoder array. For example, to encrypt the letter *D*, we first subtract *A* from *D*, which gives us 3. Then we find the letter in the encoder array with index 3, which is *G*. So, *D* in the plaintext is encrypted to *G* in the ciphertext.

An example code that illustrates the construction of an encoder array, a decoder array, and encryption and decryption using the Caesar cipher can be found at the file *CeasarCipher.java*.

Line 7 is a constructor that creates a *CaesarCipher* object. Note that the rotational shift *rotation* is passed as an argument. So, this *CaesarCipher* class can be used for an arbitrary rotational shift. Lines 9 and 10 build an encoder array and a decoder array, respectively. Line 25 calculates the index of a letter of the message in the encoder array (or the decoder array). Line 26 finds the encrypted letter (or decrypted letter) from the encoder array (or the decoder array) using the index, and that letter replaces the original letter in the message. Note that the message here can be either a plaintext or a ciphertext, depending on whether the *transforms* method (line 22) is invoked by the *encrypt* method (line 14) or the *decrypt* method (line 18).

Section 1.3.6. Two-Dimensional Arrays and

Positional Games

In Java, a two-dimensional array is implemented as an array of arrays, in which each row is itself an array. An integer two-dimensional array with eight rows and ten columns can be declared as follows:

```
int[][] data = new int[8][10];
```

The number of rows can be retrieved with *data.length* and the number of columns can be retrieved with *data*[0].*length* (here, any row index can be used).

A simple tic-tac-toe program is used to illustrate the use of two-dimensional arrays.

The game board consists of three-by-three cells. Two players, X and O, play the game by placing their respective marks on a cell in an alternating manner. In the example code shown below, the board configuration is stored in the three-by-three two-dimensional array *board*. This program does not implement an interactive game. The sequence of nine moves made by the players is hardcoded in the *main* method. The complete code can be found at the file *TicTacToe.java*.

The game board *board* is declared as a three-by-three two-dimensional array in line 5. Each cell assumes one of three values: 1 for player X's move, —1 for the player O's move, and 0 for an empty cell. Three constants are declared in lines 3 and 4 to represent these three integer values: X for 1, O for —1, and EMPTY for 0. The constructor in line 8 invokes the *clearBoard* method, which initializes all cells of the array with 0 and set the X as the first player.

The *isWin* method (line 26) checks whether a player won by checking each row, each column, and each of two diagonals. The *winner* (line 37) method returns 1 if the player X won, —1 if the player O won, and 0 otherwise.

The *toString* method (line 46) returns a string that represents a board configuration. This method is automatically invoked when *System.out.println* method is called with the *TicTacToe* object as an argument (line 70).

Section 1.3.7. Singly Linked Lists

The arrays have some drawbacks. First, the size of an array is determined when it is created and cannot be changed. Another drawback is that a large number of elements may have to be shifted when a new element is inserted or an element is deleted, which may be expensive.

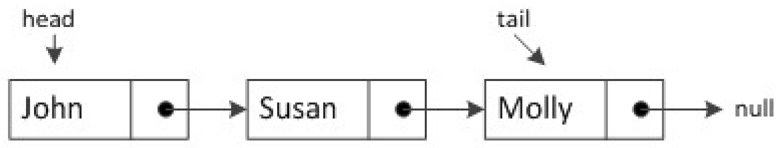
An alternative data structure, which tries to mitigate theses drawbacks, is *linked lists*. A linked list is a collection of *nodes, which* nodes are connected by *links* in a linear fashion. There are different types of linked lists. In this section, we discuss *singly linked lists*.

A node consists of an *element* and the *next* reference, which refers to the node in the list. An element may be a variable of a primitive data type or a reference to an object. In the example used in this section, it is assumed that an element is a reference to an object.

A *reference* is sometimes referred to as a *pointer*, and sometimes we say *point to* instead of *refer to*. So, *the next reference of a node A refers to node B* can be also stated as *the next pointer of node A* *points to node B*. We use these terms interchangeably.

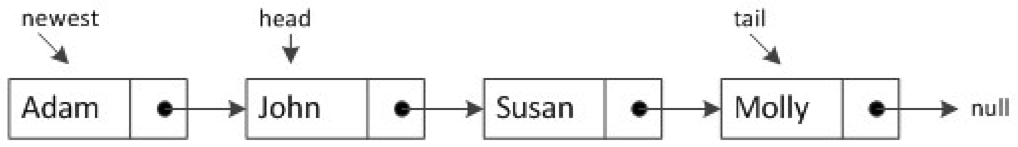
We usually keep two references, *head* and *tail*, for a singly linked list. The *head* refers to the first node in the list, and the *tail* to the last. A list must have the *head,* but the *tail* is not required. The *tail* reference can be found by traversing the list beginning at the *head*. But it is more efficient to keep an explicit *tail* reference.

An example of a singly linked list is shown below. The elements of the list are strings representing the names of students in a class.

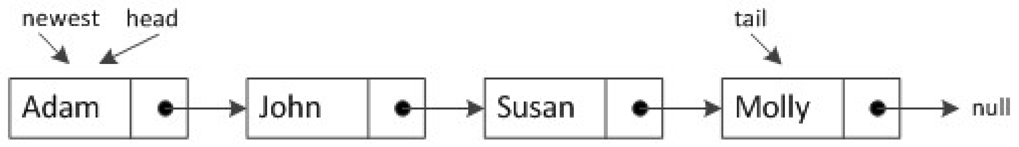


When a student registers for the class, the student can be inserted at the head of the list. The following illustrates the steps involved in inserting new student Adam to the head of the list:

A new node with the element *Adam* is created, and its *next* reference refers to the node that is pointed to by *head*. Note that the *newest* reference points to the new node.



The *head* reference is moved to point to the new node.



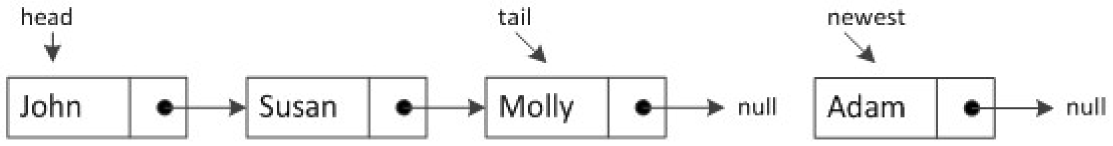
The pseudocode of the algorithm *addFirst*, which performs the operation described above, is shown below:

```
**Algorithm** addFirst(*e*)
```

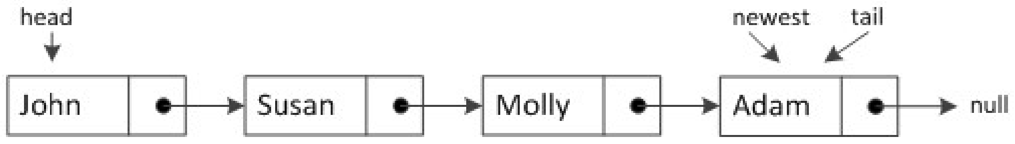
```
    newest = Node(e) // new node with the
reference to element e
    newest.next = head // new node’s next is set
to refer to old head node
    head = newest      // head refers to new node
    size = size + 1    // list size (node count)
is incremented
```

A new node can also be added to the tail of the list. The illustration of this operation, along with the pseudocode of the algorithm is shown below. In this illustration, new student Adam is inserted to the tail of the list.

A new node with the element Adam is created, and its *next* reference refers to *null*. Note that the *newest* reference points to the new node.



The *next* of the Molly node is set to point to the new node and *tail* is moved to refer to the new node.



```
Algorithm addLast(e)
    newest = Node(e)   // new node with the
reference to element e
    newest.next = null // new node’s next is set
to null
    tail.next = newest // old tail node’s next
```

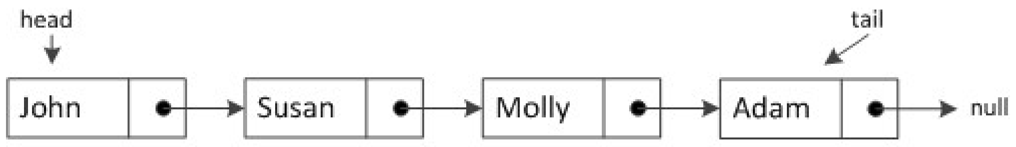
```
points to new node
    tail = newest       // tail points to new node
    size = size + 1     // list size (node count)
is incremented
```

Given an arbitrary location in the singly linked list, inserting a new node at that location is also possible, but inefficient. If such operations are required, it is better to implement a doubly-linked list, which will be discussed later.

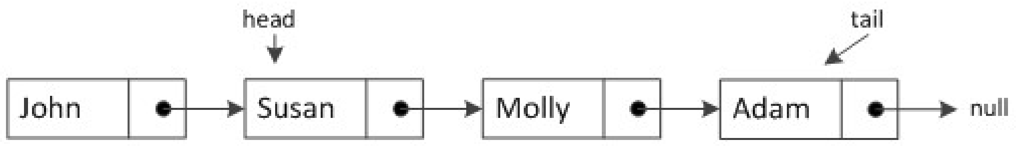
Removing an arbitrary node from a singly linked list is neither trivial nor efficient. If such operations are needed, again, a doubly linked list is a better option.

Here, we discuss how to remove a node from the head of a singly linked list. This operation is illustrated below.

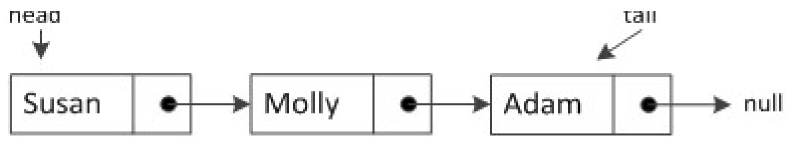
This is the list before removal:



The *head* is moved to point to the node that is pointed to by the *next* reference of the current *head* node.



After this step, it is not necessary to sever the link between the *John* node and the *Susan* node because, as far as this list is concerned, the list begins at the *head* reference. The memory allocated for the *John* node will eventually be reclaimed by the Java's garbage-collection mechanism. The following is the final configuration of the list ignoring the John node:



```
Algorithm removeFirst( )
    if head == null
        the list is empty
    head = head.next   // new head points to next
node
    size = size - 1    // list size (node count)
is decremented
```

An implementation of a singly linked list class can be found at the file *SinglyLinkedList.java*. To make the singly linked list independent of the type of the elements in the list, this implementation uses *generics framework*. The class definition includes the following methods:

```
size()—Returns the number of elements in the list.
isEmpty()—Returns true if the list is empty, false
otherwise.
first()—Returns (but does not remove)the first
element in the list.
last()—Returns (but does not remove) the last
element in the list.
addFirst(e)—Inserts a new element at the head.
addLast(e)—Inserts a new element to the end of the
list.
removeFirst()—Removes and returns the first
element in the list.
```

The singly linked list class also has a *nested class* that defines a node of the list. The definition of the *Node* class is shown below:

Code Segment 1.37

```
1  private static class Node<E> {
2    private E element;            // reference to
the element stored at this node
3    private Node<E> next;         // reference to
the subsequent node in the list
     // constructor
4    public Node(E e, Node<E> n) {
5      element = e;
6      next = n;
7    }
    // Returns the element stored at the node.
8   public E getElement() { return element; }
    // Returns the node that follows this one (or
null if no such node).
9   public Node<E> getNext() { return next; }

     // Sets the node's next reference to point to
Node n.
10   public void setNext(Node<E> n) { next = n; }
11 } //----------- end of nested Node class
-----------
```

In line 1, *E* is a formal type parameter. A *Node* object has two instance variables, *element* and *next* (in lines 2 and 3). Line 4 defines a constructor. Additionally, there are three more methods:

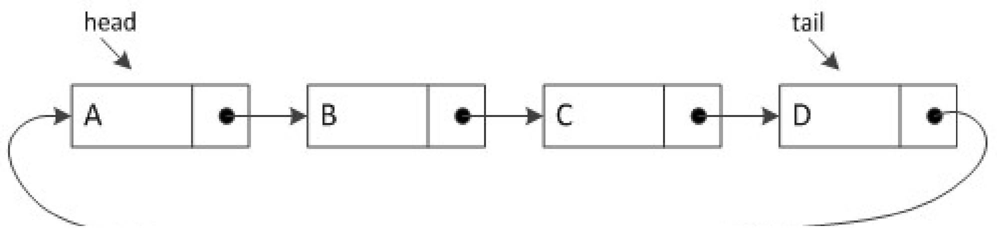
the *getElement*( ) method returns the element of the node, the *getNext*( ) method returns the next node, and the *setNext*(*Node<E> e*) method is used to set the *next* reference of the node to point to the node *n*.

Section 1.3.8. Circularly Linked Lists

A *circularly linked list* is a singly linked list in which the last element is connect to the first element, forming a circle. A circularly linked list is used in application in which objects are manipulated in a round-robin manner.

One such application is *process scheduling* of an operating system. Suppose there is one CPU, and multiple processes—say 26 processes, A to Z, for example—that need to be run on it. A *round-robin* scheduling algorithm works in the following way. All processes are kept in a *pool*. First, the scheduler removes process *A* from the pool, and the CPU runs it for a certain amount of time, called a *time slice*. After the *time slice* elapsed, process *A* is returned to the pool. Then, process *B* is removed from the pool and executed on the CPU for the duration of a *time slice*, after which it is returned to the pool. This is repeated for all processes in the alphabetical order of process names, and eventually *A* will be executed again on the CPU, followed by process *B*, and so on. In this application, we can use a circularly linked list to implement the *pool* of processes.

An example of a circularly linked list with four processes is shown below:



Since the *head* reference is easily obtained by the *tail.getNext*( ) method, we don't need to keep the *head* reference. By not keeping the *head* reference, less memory is used and the implementation becomes more efficient and simpler.

Then, given a circularly linked list of processes *C*, the round-robin scheduling can be implemented as follows:

```
Give a time slice to C.first()
C.rotate()
```

Here, the *rotate*( ) method moves the first element to the end of the list.

The following figure illustrates the *rotate* method:

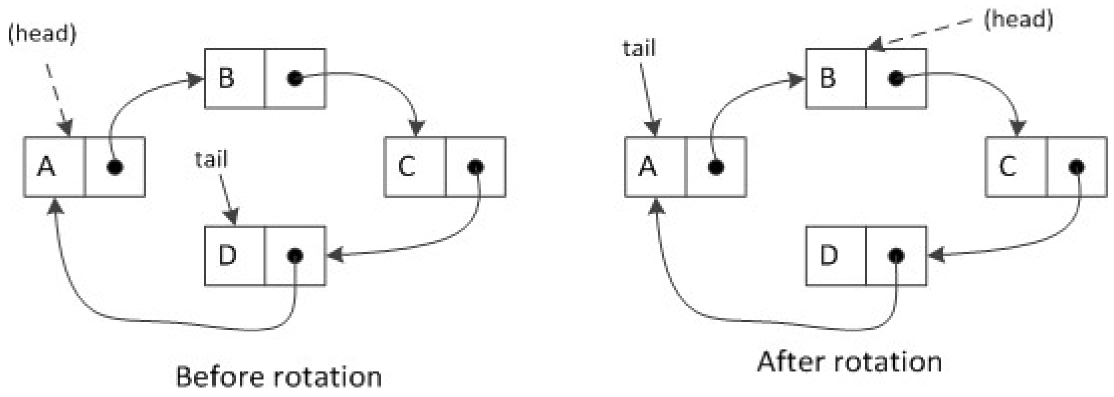


Figure 1.10. Rotation operation.

In Figure 1.10, the parentheses around *head* and the dotted arrow represent that the *head* reference is implicit. A code implementing the method is shown below:

```
public void rotate() {         // rotate the first
element to the back of the list
```

```
  if (tail != null)            // if empty, do
nothing
    tail = tail.getNext();     // the old head
becomes the new tail
}
```

We can implement a circularly linked list by slightly modifying the code of singly linked list implementation. This implementation will support all public behavior of the singly linked list plus the *rotate* method.

Existing methods need to be modified slightly (because there is no *head* reference any more). For example, the *addFirst* method is modified in the following way. We first create a new node to be added and insert it between the *tail* node and the implicit *head* node. For example, we can insert a new process *X* at the head of the list, as follows:

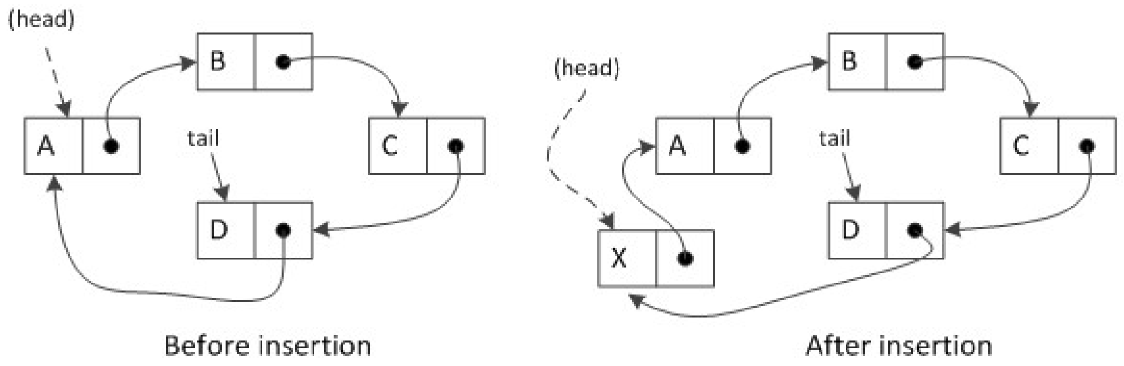


Figure 1.11. Insertion operation.

Note that there is no change in the *tail* node, but the new node *X* now becomes the implicit head node.

An implementation of the new *addFirst* method is shown below:

Code Segment 1.38

```
public void addFirst(E e) {        // adds element e to the front of the list
  if (size == 0) {
    tail = new Node<>(e, null);
    tail.setNext(tail);            // link to itself circularly
  } else {
    Node<E> newest = new Node<>(e, tail.getNext());
    tail.setNext(newest);
  }
  size++;
}
```

The complete implementation of a circularly linked list can be found at the file *CircularlyLinkedList.java*.

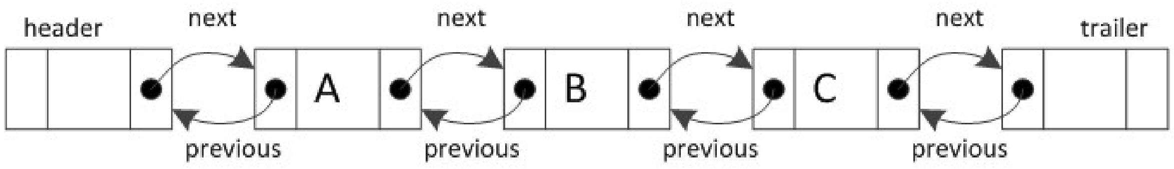
Section 1.3.9. Doubly Linked Lists

As discussed earlier, singly linked lists have some drawbacks. In general, it is not easy to insert a node at an arbitrary position in a singly linked list. It is also nontrivial to delete a node from an arbitrary position in a singly linked list. These operations, however, can be performed relatively efficiently with doubly linked lists.

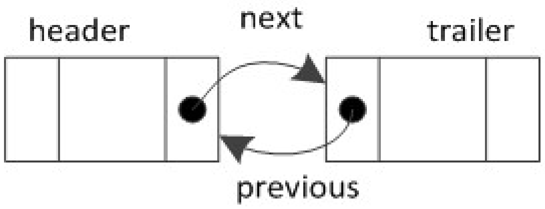
In a doubly linked list, a node has two references. In addition to the *next* reference (like in a singly linked list), each node also keeps a node that refers to the previous node.

To make manipulation of doubly linked list easier, we usually also keep two dummy nodes: a *head* node and a *trailer* (or *tail*) node.

These two dummy nodes are sometimes referred to as *sentinels*. An example of a doubly linked list with sentinels is shown below:



An empty list has only sentinels:



The sentinels enable efficient manipulation of doubly linked lists. In general, insertion and deletion operations become simpler. If there are no sentinels, inserting a node at the head, for example, should be different from inserting a node into some other position in the list. This means that an insertion code needs to test—with some conditional statement(s)—whether a new node is inserted at the head or not and perform different actions based on the test result. If there are sentinels, such a test is not necessary because a node is always inserted between two nodes, and writing a code becomes simpler.

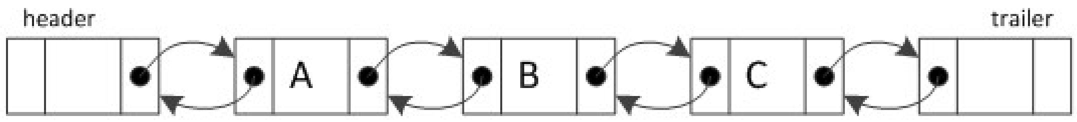
This is true for deletion, too. If there are sentinels, a node to be deleted always has two neighbors: the next node and the previous node. So, writing a code for deletion becomes simpler.

All insertions and deletions can be treated in the same manner, respectively, regardless of their locations.

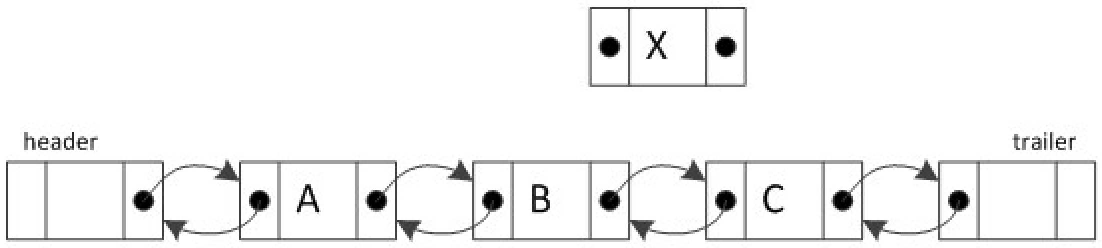
Another advantage is that the *head* node and the *trailer* node never have to be changed. If there are no sentinels, the *head* and *trailer* references may have to be changed (or updated).

Insertion of a node into a doubly linked list with sentinels is illustrated below.

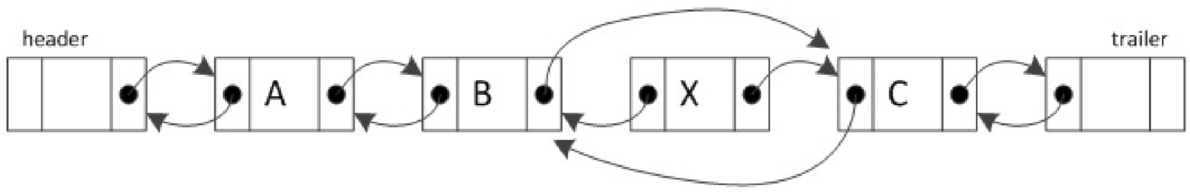
A list before insertion:



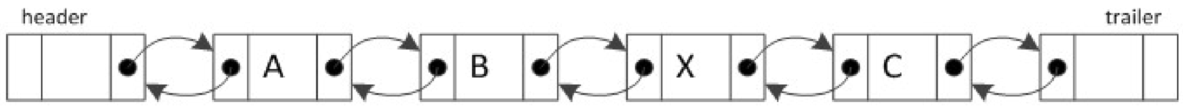
A new node with element *X* is inserted between *B* and *C*. First, a new node is created:



The *previous* reference and the *next* reference of *X* are set to point to *B* and *C*, respectively:



The *next* reference of *B* and the *previous* reference of *C* are updated to point to *X*:



An implementation is shown below:

Code Segment 1.39

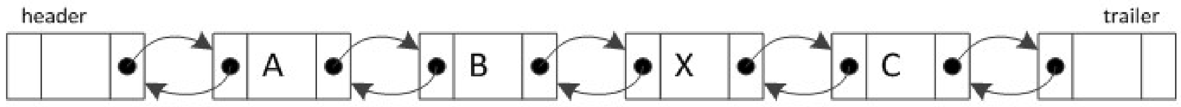
```
private void addBetween(E e, Node<E>
predecessor, Node<E> successor) {
    // create and link a new node
2    Node<E> newest = new Node<>(e, predecessor,
successor);
3    predecessor.setNext(newest);
4    successor.setPrev(newest);
5    size++;
6  }
```

This insertion method receives three arguments: the reference to the node to be inserted, and references to the two nodes between which the first node is to be inserted. In line 2, a new node is created and, at the same time, its *previous* reference and *next* reference are set to point to its predecessor and successor, respectively. Line 3 updates the *next* reference of the predecessor node and line 4 updates the *previous* reference of the successor node.

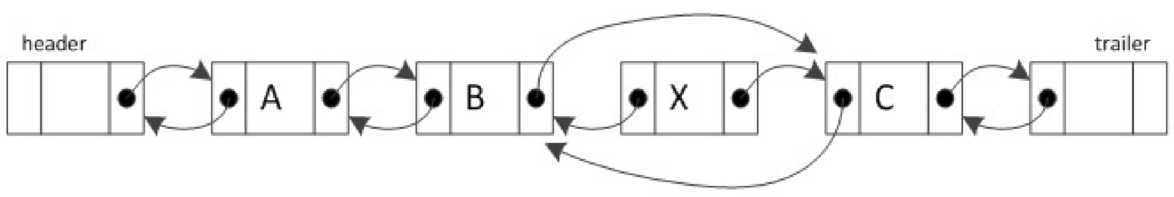
If a new node is inserted at the head of the list, it will be inserted between the *header* node and its next node. This is also true when a new node is added to the end of the list. The new node will be inserted between the *trailer* node and its previous node. So, the same code can be used for all insertions.

When deleting a node, we *splice out* the node to be deleted by directly connecting its previous and next nodes. The process is illustrated below.

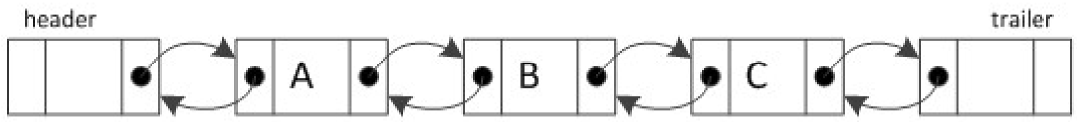
The node *X* will be deleted from the following list:



Set the next reference of *B* to point to *C* and set the previous reference of *C* to point to *B*:



The *X* node is not a part of the list any more. An updated list after deletion follows:



An implementation of deletion is shown below:

Code Segment 1.40

```
1  private E remove(Node<E> node) {
2    Node<E> predecessor = node.getPrev();
3    Node<E> successor = node.getNext();
4    predecessor.setNext(successor);
5    successor.setPrev(predecessor);
6    size--;
7    return node.getElement();
8  }
```

The *remove* method receives as its argument the reference to the node to be deleted. Suppose the node to be deleted is *X*, its previous node is *B*, and its next node is *C*. Line 2 obtains the reference to *B*, and line 3 gets the reference to *C*. Line 4 updates the next reference of *B* to point to *C*, and line 5 updates the previous reference of *C* to point to *B*.

The complete code of an implementation can be found at the file *DoublyLinkedList.java*. The implementation supports the following public methods:

```
size()—Returns the number of elements in the list.
isEmpty()—Returns true if the list is empty, false
otherwise.
first()—Returns (but does not remove)the first
element in the list.
last()—Returns (but does not remove) the last
element in the list.
addFirst(e)—Inserts a new element at the head.
addLast(e)—Inserts a new element to the end of the
list.
removeFirst()—Removes and returns the first
element in the list.
removeLast()—Removes and returns the last element
in the list.
```

Test Yourself 1.2

Consider a doubly linked list that has *header* and *trailer* sentinels. Write a Java method that finds and returns the middle node of the list without knowing the explicit knowledge of the size of the list. If there are even number of nodes, return the node that is left of center. Assume that each node is of *Node* type (that is, each node is an instance of *Node* class). Also assume that this method is invoked by a doubly linked list object, that is, this method is a member method of a class that defines the doubly linked list.

Please think carefully, practice and write your own program, and then click "Show Answer" to compare yours to the suggested possible solution.

Section 1.3.10. Equivalent Testing

In this section, we discuss how to compare two instance variables. More specifically, we discuss how to compare two objects.

Section 1.3.10.1. Two Notions of Equivalence

When comparing two scalar variables, we say they are identical (or one is equal to the other) if their values are the same. However, the notion of *equality* or *equivalence* is different when we compare two reference variables. Consider the following String variables:

```
String s1 = new String("data structure");
String s2 = new String("data structure");
```

The variable s1 is a reference to a String object that stores a string "data structure". The variable s2 is a reference to a String object that stores the same string "data structure". So, when we ask "is s1 equal to s2?" it can be interpreted as one of the following two questions: (1) Is the value of s1 the same as the value of s2? (In other words, do s1 and s2 refer to the same object?) (2) Are the strings (or two String objects) referenced by s1 and s2 are the same string? If we use the first interpretation, then the answer is no (because they are referencing two different objects). If we use the second interpretation, then the answer is yes (because both objects store the same string).

So, there are two different notions of equivalence when we compare two reference variables. To determine whether two references are equivalent, we can use the comparison operator "= =", or we can use the *equals* method, which is defined in the *Object* class and inherited by all objects.

If we want to determine whether the two objects referenced by the reference variables are equivalent, then we have to implement an *equals* method, which is appropriate for those objects, by overriding the *equals* method of the *Object* class.

The String class implemented its own *equals* method, overriding that of the *Object* class. This method compares two String objects character by character (instead of comparing the references to the objects).

The following code segment illustrates different notions of equivalence:

Code Segment 1.41

```
public class StringTest {
     public static void main(String[] args) {
       String s1 = new String("data structure");
       String s2 = s1;
       String s3 = new String("data structure");
      System.out.println("string s1 equals string s2: " + s1.equals(s2));
       System.out.println("string s1 equals string s3: " + s1.equals(s3));
8         System.out.println("reference s1 equals
reference s2: " + (s1 == s2));
9         System.out.println("reference s1 equals
reference s3: " + (s1 == s3));
10      }
11 }
```

The expected output is as follows:

```
string s1 equals string s2: true
string s1 equals string s3: true
reference s1 equals reference s2: true
reference s1 equals reference s3: false
```

Line 3 creates a new String object, line 4 sets the string reference variable s2 to s1 (so s1 and s2 point to the same String object), and line 5 creates a new String object with the same character string.

Lines 6 and 7 compare strings using the *equals* method. Since the strings store the same sequence of characters, the results are all true. Lines 8 and 9 compare the strings using the "= =" operator, which compares th ereferences, not the objects. In this case, (s1 = = s2) returns *true*, but (s1 = = s3) returns *false* because s1 and s3 are referencing two different objects (though they store the same character strings).

When you define a class and you want to be able to compare objects based on what they actually are, not on references, you need to implement your own *equals* method for this class. In this case, you need to make sure that your implementation satisfies the following properties:

```
Reflexivity—For any nonnull reference variable x,
x.equals(x) returns true.
Symmetry—For any nonnull reference variables x and
y, x.equals(y) = y.equals(x).
Transitivity—For any nonnull reference  variables
x, y, and z, if x.equals(y)  is true and
y.equals(z) is true, then x.equals(z) is true.
null property—For any nonnull reference variable
x, x.equals(null) returns false.
```

In mathematics, a relation that satisfies the first three properties is called an *equivalence relation*. In Java, one more property is needed to account for *null*.

1.3.10.2. Equivalence Testing with Arrays

Arrays are references. Like strings, arrays can be compared in two different ways: one can compare references (to see whether two references refer to the same array) or compare the content of arrays. Java's *java.util.Arrays* class provides the following:

```
a == b: Tests if a and b refer to the same array
instance.
a.equals(b): This is identical to a == b.
Arrays.equals(a, b): Returns true if the arrays
have the same number of
elements and all pairs of corresponding elements
are equal to each
other. If elements are primitives, == operator is
used. If elements are
reference types, then a[k].equals(b[k]) is used.
```

Comparing multi-dimensional arrays is slightly complicated. Consider the following two two-by-three, two-dimensional arrays:

```
int[][] a = {{1,3,5}, {2,4,6}};
int[][] b = {{1,3,5}, {2,4,6}};
```

If we compare these two arrays with *Arrays.equals*(*a,b*), the result will be *false*, even though the two arrays have the same content. The reason is as follows. As mentioned earlier, the *Arrays.equals* method compares corresponding elements of two arrays using the *a*[*k*]*.equals*(*b*[*k*]) method. For example, it compares the pair of the first elements of the two arrays: *a*[0]= {1, 3, 5} and *b*[0] = {1, 3, 5}. Note that both *a*[0] and *b*[0] are references to array instances. *a*[0]*.equals*(*b*[0]) tests whether two references refer to the same array instance and, in this case, it is obviously not the case (they are stored at different memory locations). So, it returns *false*.

Java provides the *Arrays.deepEqual*(*a,b*) method, which we can use to compare the contents of two arrays. This method is different from the *Arrays.equals*(*a,b*) method. When comparing a pair of elements from two arrays, it recursively calls *Arrays.deepEquals*(*a*[*k*], *b*[*k*]) instead of *a*[*k*]*.equals*(*b*[*k*]). So, the method returns *true* only if the contents of the two arrays are identical.

1.3.10.3. Equivalence Testing with Linked Lists

In this section, we discuss how to implement an *equals* method that compares two singly linked lists. The method scans two lists in parallel and compares a pair of elements from them—say, *x* and *y*, using *x.equals*(*y*). An implementation, in the context of *SinglyLinkedList*, is shown below:

Code Segment 1.40

```
public boolean equals(Object o) {
  if (o == null) return false;
  if (getClass() != o.getClass()) return false;
  SinglyLinkedList other = (SinglyLinkedList) o;   // use nonparameterized type
  if (size != other.size) return false;
  Node walkA = head; // traverse the primary list
  Node walkB = other.head; // traverse the secondary list
  while (walkA != null) {
    if (!walkA.getElement().equals(walkB.getElement())) return false; //mismatch
    walkA = walkA.getNext();
    walkB = walkB.getNext();
  }
  return true;   // if we reach this, everything matched successfully
}
```

This method compares the *this* object, which invokes the method, with another object *o*, which is passed as an argument. It receives as an argument an object of arbitrary type. So, *Object* is used as the type of the formal parameter *o*.

Line 2 checks whether *o* is *null*, and line 3 tests whether the type of *this* object is identical to the type of *o*. In line 4, the type of *o* is cast to *SinglyLinkedList*. Line 5 checks whether the sizes of the two lists are identical. If all these conditions are satisfied, then the two lists are traversed in parallel, and a pairwise comparison is made (line 9).

Section 1.3.11. Cloning Data Structures

In this section, we discuss how to create a separate copy of a data structure.

Section 1.3.11.1. Shallow Copy and Deep Copy

When we make a copy of an object (or a data structure that is usually implemented as an object) and some instance variables are references to other objects, there are two ways to make a copy of the object: *shallow copy* and *deep copy*.

When creating a shallow copy of an object, we copy all instance variables, including reference variables, using a standard assignment statement. In this case, reference variables in the copy will refer to the same objects referenced by those in the original objects. An example of a shallow copy is shown below.

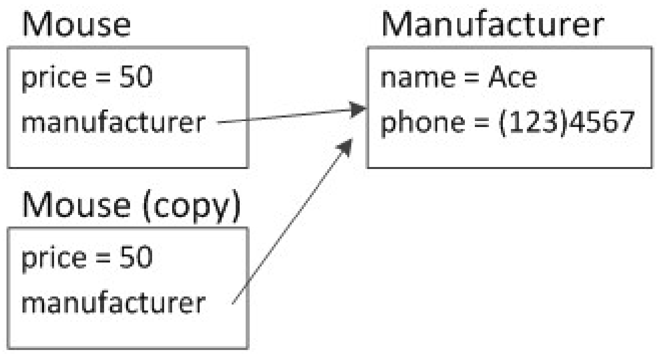


Figure 1.12. Shallow copy.

In this example, a computer mouse object has two instance variables. *Price* is a primitive type variable, and *manufacturer* is a reference variable that refers to a *Manufacturer* object. Both instance variables have the same values in the copy. So, the reference variable *manufacturer* in the copy points to the same *Manufacturer* object pointed to by the original object.

When a *deep copy* is created, primitive type instance variables are copied by a standard assignment, but reference type instance variables are not copied in this way. Instead, a new object is created, and the reference variable in the copy points to this new object, as illustrated below:

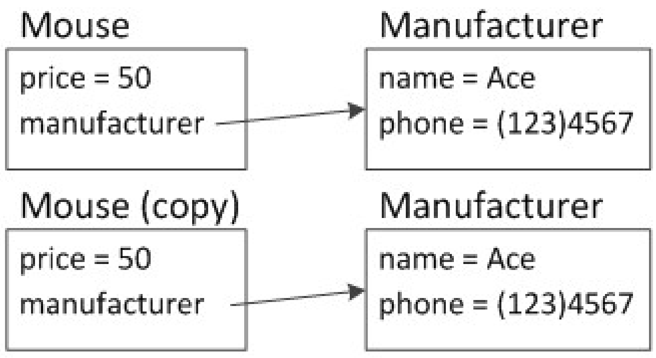


Figure 1.13. Deep copy.

The primitive instance variable *price* is copied as before. But the reference variable *manufacturer* will point to a newly created *Manufacturer* object with the same content.

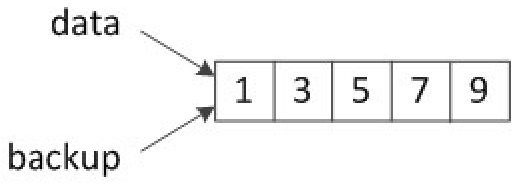
Java has the *clone* method defined in the *Object* class. This *clone* method performs a shallow copy and the method is declared as *protected*. If you want to use the *clone* method in your class definition, you first need to specify that the class implements the *Cloneable* interface and then declare a public version of the *clone*( ) method.

If you want to allow the creation of deep copies of the objects, then it is your responsibility to implement a cloning method that is appropriate for the class.

Section 1.3.11.2. Cloning Arrays

Sometimes it is necessary to make a copy of an array. There are different ways of doing this. First, we can declare an array of the same type and assign the original array variable to this new array variable. In this case, since an array variable is a reference variable, the new array variable will simply refer to the original array, as shown below:

```
int[] data = {1,3,5,7,9};
int[] backup;
backup = data;
```



We can make a copy using the *clone* method. If the type of element of the array is a primitive type, such as an integer, then a separate copy of the array will be created:

```
backup = data.clone();
```

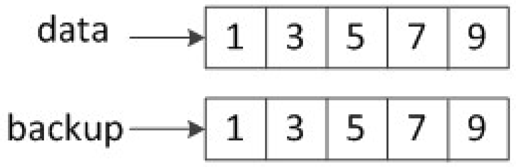


Figure 1.14. Cloning an array with elements of primitive type.

If the elements of the array are reference variables, then the *clone* method does not create an independent array. Instead, elements (which are reference variables) in the new array will simply refer to the objects referenced by the elements in the original array. Suppose that we have an array named *contacts*, which stores a list of persons. In the array, each element is a reference variable that refers to a *Person* object. If we create a copy of the array, named *guests*, using the *clone* method, the result will look like the following:

```
guests = contacts.clone( );
```

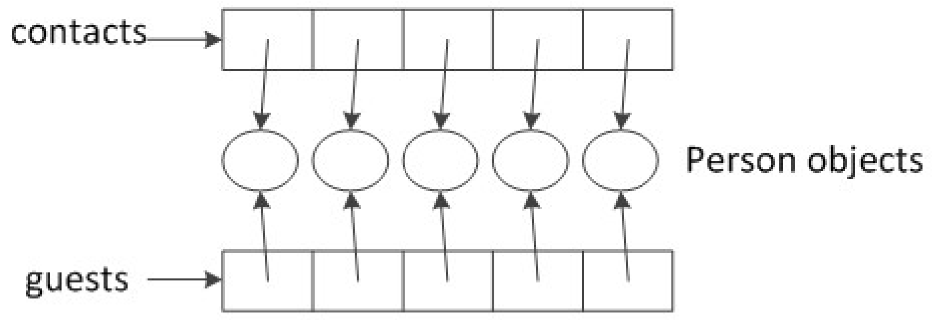


Figure 1.15. Cloning an array with elements of reference type.

If we want a *deep copy* of the *guests* array, we need to implement our own method that creates independent copies of the *Person* objects in the new array. To be able to do this, we must declare the *Person* class as *Cloneable*. Then, we can write a code as follows:

```
Person[] guests = new Person[contacts.lenght];
for (int k=0; k<contacts.length; k++)
    guests[k] = (Person)contacts[k].clone();
```

As discussed earlier, a two-dimensional array is a one-dimensional array of one-dimensional arrays. So, the same issue of *shallow copy* and *deep copy* exists. Java does not provide a cloning method that creates a *deep copy*. We can implement a *deep cloning* method by cloning individual rows of an array. One implementation is shown below:

Code Segment 1.43

```
1  public static[][] deepClone(int[][] original) {
2    int[][] backup = new int[original.length][];
3    for (int k=0; k<original.length; k++)
4      backup[k] = original[k].clone();
5    return backup;
6  }
```

In the *for* loop of line 3, each row of the top-level array, which is a one-dimensional array, is copied. So, a *deep copy* of the original array is created.

Section 1.3.11.3. Cloning Linked Lists

In this section, we discuss how to make a copy of a singly linked list. We define a public *clone* method in the *SinglyLinkedList* class. To do this, it is necessary to declare that the *SinglyLinkedList* class implements the *Cloneable* interface, as follows:

```
public  class SinglyLinkedList<E> implements
Cloneable { . . . }
```

An implementation is shown below:

Code Segment 1.44

```
1  public SingslyLinkedList<E> clone() throws CloneNotSupportedException {
```

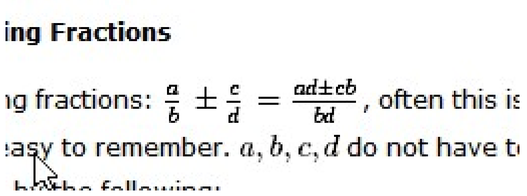
```
1  public SinglyLinkedList<E> clone() throws CloneNotSupportedException {
2    // always use inherited Object.clone() to create the initial copy
3    SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone(); // safe cast
4    if (size > 0) {                    // we need independent chain of nodes
5      other.head = new Node<>(head.getElement(), null);
6      Node<E> walk = head.getNext();      // walk through remainder of original list
7      Node<E> otherTail = other.head;     // remember most recently created node
8      while (walk != null) {              // make a new node storing same element
9        Node<E> newest = new Node<>(walk.getElement(), null);
10        otherTail.setNext(newest);     // link previous node to this one
11        otherTail = newest;
12        walk = walk.getNext();
13      }
14    }
15    return other;
16  }
```

In line 3, a new *SinglyLinkedList* instance *other* is created. It is done by calling the *clone* method of the *Object* class, *super.clone*( ), and casting the type to *SinglyLinkedList*. The singly linked list *other* is a shallow copy of the original list (recall that the *clone* method in the *Object* class creates a shallow copy). This means that the *head* reference is shared by the original list and the *other* list. So, to create an independent list with its own *head* reference, in line 5, the *head* reference of the *other* list is created. Then, the *while* loop of lines 8 through 13 constructs the *other* list. In line 9, a new node with the reference to an element in the original list is created and, in line 10, it is added to the tail of the *other* list. This is repeated until all nodes in the original list are traversed. Note that this *clone* method creates a new list but the elements of the new list still refer to the objects referred to by the elements of original list.

Section 1.4. Basic Math Facts
Appendix A, *Useful Mathematical Facts*, which is available at www.wiley.com/college/goodrich,has some mathematical facts that are useful for this course. This section shows some of these; for more, please refer to *Appendix A*.

MathJax

Variables, formulae, and equations in this course are rendered using MathJax.



ing Fractions

ıg fractions: $\frac{a}{b} \pm \frac{c}{d} = \frac{ad \pm cb}{bd}$, often this is

ːasy to remember. a, b, c, d do not have t

To enable its features in your browser, right-click (or ctrl-click on a single-mouse-button Mac) on a variable or equation to see your MathJax settings.

MathJax can be used with the MathPlayer plugin for Internet Explorer, which converts math to speech and highlights the math as it is spoken.

Identities for logarithms and exponents:

1. logbac=logba+logbc$\log_b ac = \log_b a + \log_b c$
2. logbac=logba–logbc$\log_b \frac{a}{c} = \log_b a - \log_b c$
3. logbac=clogba$\log_b a^c = c log_b a$
4. logba=(logca)logcb$\log_b a = \frac{(\log_c a)}{\log_c b}$
5. blogca=alogcb$b^{\log_c a} = a^{\log_c b}$
6. (ba)c=bac$(b^a)^c = b^{ac}$
7. babc=ba+c$b^a b^c = b^{a+c}$
8. babc=ba−c$\frac{b^a}{b^c} = b^{a-c}$

Integer functions:

- Floor function: ⌊x⌋$\lfloor x \rfloor$ = the largest integer less than or equal to xx
- Ceiling function: ⌈x⌉$\lceil x \rceil$ = the smallest integer greater than or equal to xx
- Modulo function: for a>0$a > 0$ and b>0$b > 0$

  aa mod bb = a−⌊ab⌋$a - \left\lfloor \frac{a}{b} \right\rfloor$
- Factorial function: *n*! = 1 x 2 x 3 x . . . (*n*-1) x *n*

Summations:

1. ∑ni=1i=n(n+1)2$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
2. ∑ni=1i2=n(n+1)(2n+1)6$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

3. ∑ni=0ai=an+1−1a−1$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$
4. ∑∞i=0ai=11−a$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}$, where 0<a<1$0 < a < 1$

Module 1 Practice Questions

The following are some review questions for you to practice. Please read each question, think carefully, figure out your own answer or write your own program first, and then click "Show Answer" to compare yours to the suggested answer or the possible solution.

Test Yourself 1.11

Write a Java method that counts the number of vowels in a given character string.

Test Yourself 1.12

Write a Java method that takes an array of integers and determines if all integers are different from each other.

Test Yourself 1.16

Consider the Progression class in Code Segment 1.13 (in the textbook, it is defined in Code Fragment 2.2). Rewrite the Progression class that produces a sequence of values of generic type T and has a single constructor that accepts an initial value.

Test Yourself 1.17

Consider the following code segment:

```
public static void insertionSort(char[] data) {
  int n = data.length;
  for (int k = 1; k < n; k++) {
    char cur = data[k];
    int j = k;
    while (j > 0 && data[j-1] > cur) {
      data[j] = data[j-1];
      j--;
    }
    data[j] = cur;
  }
}
```

This method sorts characters in non-decreasing order using the insertion-sort algorithm. Modify the code so that characters are sorted in non-increasing order.

Test Yourself 1.20

Consider the *CircularlyLinkedList* class. You can find the link to the *CircularlyLinkedList.java* file at the end of Section 1.3.8. The code can also be found in Code Fragment 3.16 in the textbook. Write the *size*( ) method for the *CircularlyLinkedList* class, assuming that we do not have an instance variable that keeps the size of a list.

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Java* (6th ed.) Wiley.
- Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2015). *The Java® language specification, Java SE 8 edition.* Oracle America Inc.

- Oracle. *The Java Tutorials*. Retrived from https://docs.oracle.com /javase/tutorial/index.html.