Interfaces, Inheritance, and Polymorphism 17

17.1 Interface

17.1.1 The Structure of an Interface

An **interface** is a template for building a Java class. The components of an interface are static methods, constants, and **abstract methods**. An abstract method is a method header followed by a semicolon and is without a body. All abstract methods have public visibility. An abstract method specifies only its syntax, but not its semantics.

A class written based upon an interface must turn each abstract method into one with a body having the same visibility, the same return type, the same name, the same parameter type signature, and the same `throws` declaration (if any) as the abstract one. We call this action **overriding**. A class `C` **implements** an interface `I` if its header formally states so, in the following manner:

```
public class C implements I
```

If the source code of `C` has this declaration, the constants and the methods (both static and abstract) appearing in `I` are automatically included in the source code of `C`, so they can be used without specifying the name of the interface. We call this mechanism **inheritance**. It is not possible to instantiate an object of `I`, but it is possible to declare the data type of an object to be `I`. We call this mechanism **polymorphism**.

An object class may implement multiple interfaces. If a class `C` implements interfaces `I1, ..., Ik`, the declaration uses the keyword `implements` only once and places the interfaces with commas in between in the following manner:

```
public class C implements I1, ..., Ik
```

The names of the interfaces may appear in any order.

17.1.2 A Simple Pizza Application

17.1.2.1 A Simple Pizza Class

Let us learn how to write and use interfaces by working on an application for building a menu at a pizza shop. The application will consist of (a) a class for an individual pizza, (b) a class for a

M. Ogihara, *Fundamentals of Java Programming*,
https://doi.org/10.1007/978-3-319-89491-1_17

menu, and (c) a main application class. The information available for a pizza is its name and price. To simplify the problem, we assume that pizzas can be added to the menu, or removed from the menu, but cannot be edited, meaning that it is not possible to change the names or prices of pizzas. In such a setting, changing the name or price of a pizza on the menu must be accomplished by removing the pizza from the menu, and then adding a revised version to the menu. Since the names or prices of pizza objects cannot be changed, we need only two functions in our pizza objects. The two functions are obtaining their names and obtaining their prices. We encode these requirements in our interface, `PizzaSimpleint`, that is shown next. The required functions, `getName` and `getPrice`, are stated in Lines 3 and 4 as abstract methods.

```
1 public interface PizzaSimpleInt
2 {
3   public String getName();
4   public double getPrice();
5 }
```

Listing 17.1 The code for `PizzaSimpleInt`

We write an implementation, `PizzaSimple`, of the interface. The coding idea for `PizzaSimpleInt` is straightforward. We use a `String` instance variable, `name`, for the name and a `double` instance variable, `price`, for the price. We assume that the value of `price` represents the price in dollars.

The class has a declaration, `implements PizzaSimpleInt` (Line 1). The instance variables are declared in Lines 3 and 4 as private variables. The constructor receives the name and price as parameters, and stores their values in the instance variables. Since the parameters and the instance variables have the same names, the prefix `this.` is attached to the instance variables to distinguish between the two (Line 8 and 9). The getters are overridden in Lines 11–18.

```
1  public class PizzaSimple implements PizzaSimpleInt
2  {
3    private String name;
4    private double price;
5
6    public PizzaSimple( String name, double price )
7    {
8      this.name = name;
9      this.price = price;
10   }
11   public String getName()
12   {
13     return name;
14   }
15   public double getPrice()
16   {
17     return price;
18   }
19 }
```

Listing 17.2 The code for `PizzaSimple`

17.1.2.2 Using an Interface as a Data Type

`PizzaSimpleCollection` is an object class representing a collection of pizzas, and `PizzaSimpleMain` is the main application program. Before halting, `PizzaSimpleMain` records the menu in a text data file. A path to the data file can be specified at the start of the program as `args[ 0 ]`. If the program starts with no such argument, `PizzaSimpleMain` asks for a file path from the user. Before starting interactions for building the menu, `PizzaSimpleMain` checks if the file specified by the file path exists, and if so, reads the data from the file. If the file does not exist, the program starts with no pizzas on the menu, and uses the file only for recording data. Since the file may be used at the start and at the end, we memorize the file path as an instance variable of `PizzaSimpleCollection`.

`PizzaSimpleInt` is an interface, so has no constructor. However, `PizzaSimpleInt` can be used as a data type. For instance, we can declare a `PizzaSimpleInt` variable, and then assign a `PizzaSimple` object to the variable, as shown next:

```
1 PizzaSimpleInt pizzaX;
2 pizzaX = new PizzaSimple( Hawaiian, 15.50 );
```

`PizzaSimpleCollection` uses an array of `PizzaSimpleInt` as the list of pizzas on the menu:

```
  private PizzaSimpleInt[] list;
```

To get started, the constructor instantiates `list` as a 0-element array of `PizzaSimpleInt`:

```
    list = new PizzaSimpleInt[ 0 ];
```

The format of the data files is as follows:

- The first line of the data file is the number of pizzas recorded in the file.
- Following the first line, the information appears for the individual pizzas on the menu. Each pizza is recorded in two lines. The first line for the name and the second line for the price. The number of pizzas recorded appear in this manner must match the number of pizzas stated in the first line.

Here is a sample data file:

```
 1 5
 2 Four Cheese
 3 12.5
 4 Nettuno
 5 11.0
 6 Capriccioso
 7 14.0
 8 Alfredo
 9 14.5
10 Meat Lover's
11 13.5
```

In addition to reading from the data file (at the start of the program) and saving the information to the data file (at the end of the program), `PizzaSimpleCollection` has three functions, adding a pizza, deleting a pizza, and viewing the entire collection.

17.1.2.3 Instance Variables and Constructors

The class `PizzaSimpleCollection` involves file access, so needs imports. They are stated in Lines 1 and 2. In addition to the instance variable `list` of type `PizzaSimpleInt[]`, the class has a `File` instance variable named `theFile` (Lines 5 and 6). The file path to the data file is recorded using this `File` object. During the execution of the program, `list.length` will be equal to the number of pizzas on the menu.

The class `PizzaSimpleCollection` has two constructors. The first constructor (Line 8) receives a `File` data, `f`, as its formal parameter. The constructor stores the value of `f` to the instance variable `theFile` (Line 10), instantiates `list` as a 0-element array of `PizzaSimpleInt`, and then calls the method `read` to add from the data stored in the data file `f`.

The method `read` has a `throws FileNotFoundException` declaration, so the constructor has the same declaration.

```
1  import java.util.*;
2  import java.io.*;
3  public class PizzaSimpleCollection
4  {
5    private PizzaSimpleInt[] list;
6    private File theFile;
7
8    PizzaSimpleCollection( File f ) throws FileNotFoundException
9    {
10     theFile = f;
11     list = new PizzaSimpleInt[ 0 ];
12     read();
13   }
14   PizzaSimpleCollection( String name ) throws FileNotFoundException
15   {
16     this( new File( name ) );
17   }
18
```

Listing 17.3 The code for `PizzaSimpleCollection` (part 1). The instance variables and the constructors

The second constructor (Line 14) receives a `String` data `name` as its formal parameter. The required action of the constructor is to execute the code for the first constructor with `new File( name )` in place of `f`:

```
1      theFile = new File( name );
2      list = new PizzaSimpleInt[ 0 ];
3      read();
```

In writing this code, we can recycle the code for the first constructor by calling it, as shown in Line 16:

```
    this( new File( name ) );
```

If a constructor calls another constructor of the same class, the keyword of `this` is used as the name of the constructor. The call of another constructor from the same class has to appear in the first line of the code. The code like:

```
1     File g = new File( name );
2     this( g );
```

makes sense, but is syntactically incorrect.

17.1.2.4 Reading Data

The method `read` checks if the file specified by `theFile` exists, and if so, reads the data from the file. Despite this check, there still remains the possibility that the constructor of a `Scanner` throws `FileNotFoundException`. Therefore, the method `read` has the `throws FileNotFoundException` declaration (Line 19). If `theFile` passes this existence check (Line 21), the program instantiates a `Scanner` object to read data from the file (Line 23). The method reads the number of pizzas recorded in the file using `nextLine` and `Integer.parseInt` (Line 24). The method then instantiates an array of `PizzaSimpleInt` data whose length is equal to the number of elements specified in the first line (Line 25).

After the initialization, the method `read` uses a for-loop to read the remainder of the file contents in pairs of lines. The first line of a pair is the name of the pizza (Line 28) and the second line is the price (Line 29). The method uses `Double.parseDouble` to convert the second line to a `double` value, instantiates a `PizzaSimple` object with the name and price, and stores it in its designated position in the array (Line 30). After completing the process of reading information, the method closes the `Scanner` object (Line 32).

```
19    public void read() throws FileNotFoundException
20    {
21      if ( theFile.exists() )
22      {
23        Scanner scanner = new Scanner( theFile );
24        int size = Integer.parseInt( scanner.nextLine() );
25        list = new PizzaSimpleInt[ size ];
26        for ( int index = 0; index < size; index ++ )
27        {
28          String name = scanner.nextLine();
29          double price = Double.parseDouble( scanner.nextLine() );
30          list[ index ] = new PizzaSimple( name, price );
31        }
32        scanner.close();
33      }
34    }
35
```

Fig. 17.4 The code for `PizzaSimpleCollection` (part 2). The method `read`

17.1.2.5 Writing Data

The method `write` records the data in the file specified by `theFile`. The method has a `throws FileNotFoundException` declaration (Line 36). The method instantiates a `PrintStream` object with `theFile` (Line 38). The method writes the length of the array `list` in one line, and then

uses a for-loop to write the information of the individual pizzas in two lines. To accomplish this, the method uses the getters of `PizzaSimpleInt`. After completing the process of writing the data into the data file, the method closes the `PrintStream` object by calling the method `close` (Line 33).

```
36   public void write() throws FileNotFoundException
37   {
38     PrintStream stream = new PrintStream( theFile );
39     stream.println( list.length );
40     for ( int index = 0; index < list.length; index ++ )
41     {
42       stream.println( list[ index ].getName() );
43       stream.println( list[ index ].getPrice() );
44     }
45     stream.close();
46   }
47
```

Listing 17.5 The code for `PizzaSimpleCollection` (part 3). The method `write`

17.1.2.6 Adding a Pizza to the List

The method `add` receives a `String` data, `name`, and a `double` data, `price`, as formal parameters (Line 48). These parameters represent the name and price of a new pizza to be added to the menu. Since `list.length` must be equal to the number of pizzas on the menu, the array must be extended. The method generates a new array of `PizzaSimpleInt` to increase its capacity by one. This is accomplished by calling `Arrays.copyOf`. The copy length is `list.length + 1` (Line 50). The array returned by `copyOf` is stored in `listNew`. The designated position of the new pizza is `list.length`. The method stores a new `PizzaSimple` object instantiated with the two parameters in the designated position (Line 51). After that, the method stores the array in `list` (Line 52).

```
48   public void add( String name, double price )
49   {
50     PizzaSimpleInt[] listNew = Arrays.copyOf( list, list.length + 1 );
51     listNew[ list.length ] = new PizzaSimple( name, price );
52     list = listNew;
53   }
54
```

Listing 17.6 The code for `PizzaSimpleCollection` (part 4). The method `add`

17.1.2.7 Removing a Pizza from the List

The method `delete` receives an `int` data, `index`. This is the index to the element to be removed (Line 55). The removal necessitates resizing of the array. Like `add`, the method uses `Arrays.copyOf`. This time, the copy length is `list.length - 1` (Line 57). The array `copyOf` returns has the elements correctly at their designated positions for all indexes before `index`. Therefore, the elements located after `index` are copied manually (Line 58-61).

```
55    public void delete( int index )
56    {
57      PizzaSimpleInt[] listNew = Arrays.copyOf( list, list.length - 1 );
58      for ( int pos = index + 1; pos < list.length; pos ++ )
59      {
60        listNew[ pos - 1 ] = list[ pos ];
61      }
62      list = listNew;
63    }
64
```

Listing 17.7 The code for `PizzaSimpleCollection` (part 5). The method `delte`

17.1.2.8 Viewing the Elements of the List

The method `view` (Line 65) is for printing the menu. For each pizza, the method prints its position in the collection, its name, and its price using the format `%3d:%s:$%.2f%n` (Lines 69 and 70).

```
65    public void view()
66    {
67      for ( int index = 0; index < list.length; index ++ )
68      {
69        System.out.printf( "%3d:%s:$%.2f%n", index,
70            list[ index ].getName(), list[ index ].getPrice() );
71      }
72    }
73  }
```

Listing 17.8 The code for `PizzaSimpleCollection` (part 6). The method `view`

The source code of `PizzaSimpleCollection`, has only two occurrences of `PizzaSimple` (Lines 30 and 51). If there is an alternate implementation of `PizzaSimpleInt`, using the alternate in place of `PizzaSample` is easy. We only have to rewrite the two places where `PizzaSimple` appears.

17.1.2.9 The Pizza Collection Main Class

`PizzaSimpleMain` is the main class that handles interactions with the user. The method `main` has a `throws FileNotFoundException` declaration, because it uses `PizzaSimpleCollection`. At the start, if the length of `args` is positive (Line 8), the program instantiates `PizzaSimpleCollection` object named `data` with `args[ 0 ]` (Line 10); otherwise, the program asks the user to enter a file path, and then uses the input for instantiation (Lines 14 and 15). In both cases, the constructor of `PizzaSimpleCollection` attempts to read the data from the specified file only if the file exists, so if the user provides an invalid file path, no data will be read at the start. The program uses a `String` variable named `input` to record the input from the user. The initial value of `input` is an empty `String` (Line 17).

The interactions with the user are repeated using a do-while loop until the user enters an input starting with `"Q"` as her choice of action (Lines 18 and 38). In the loop-body, the program prompts the user to enter the choice of action to be performed (Lines 20 and 21), receives an input, and stores it in `input` (Line 22). The execution is then directed using a `switch` statement based upon the first character of the input from the user (Line 24). If the character is `'V'`, the method executes

`data.view()` (Line 26). If the character is `'A'`, the method receives the name (Lines 28 and 29) and the price (Lines 30 and 31) from the user, stores them in variables, `name` and `price`, and calls `data.add( name, price )` (Line 32). If the character is `'D'`, the method receives an input line from the user, converts it to an integer on the fly using `Integer.parseInt`, and calls `data.delete` with this integer as the actual parameter (Lines 35 and 36). If the character is `'Q'`, nothing happens inside the switch statement, and the loop is terminated (Line 38). Before halting the program, the program calls `data.write()` to save the data. `Double.parseInt` and `Integer.parseInt` are used to read the price of the new pizza and the index to the element to be removed.

```
1  import java.util.*;
2  import java.io.*;
3  public class PizzaSimpleMain
4  {
5    public static void main( String[] args ) throws FileNotFoundException {
6      Scanner keyboard = new Scanner( System.in );
7      PizzaSimpleCollection data;
8      if ( args.length > 0 )
9      {
10       data = new PizzaSimpleCollection( args[ 0 ] );
11     }
12     else
13     {
14       System.out.print( "Enter data file name: " );
15       data = new PizzaSimpleCollection( keyboard.nextLine() );
16     }
17     String input = "";
18     do
19     {
20       System.out.println( "Enter your choice by first letter" );
21       System.out.print( "View, Add, Delete, Quit: ");
22       input = keyboard.nextLine();
23
24       switch ( input.charAt( 0 ) )
25       {
26         case 'V': data.view(); break;
27         case 'A':
28           System.out.print( "Enter name: " );
29           String name = keyboard.nextLine();
30           System.out.print( "Enter price: " );
31           double price = Double.parseDouble( keyboard.nextLine() );
32           data.add( name, price );
33           break;
34         case 'D':
35           System.out.printf( "Enter index: " );
36           data.delete( Integer.parseInt( keyboard.nextLine() ) );
37       }
38     } while ( !input.startsWith( "Q" ) );
39     data.write();
40   }
41 }
```

Listing 17.9 The code for `PizzaSimpleMain`

Suppose that the name of the previous sample data file is `pizzaSimpleData.txt`. The program can be initiated with the file path to the file as `args[ 0 ]` as follows:

```
% java PizzaSimpleMain pizzaSimpleData.txt
Enter your choice by first letter
View, Add, Delete, Read, Quit: V
  0:Peperoni:$10.50
  1:Four Cheese:$12.50
  2:Nettuno:$11.00
  3:Meat Lovers:$13.00
Enter your choice by first letter
View, Add, Delete, Quit: A
Enter name: Capricciosa
Enter price: 14
Enter your choice by first letter
View, Add, Delete, Quit: D
Enter index: 0
Enter your choice by first letter
View, Add, Delete, Quit: V
  0:Four Cheese:$12.50
  1:Nettuno:$11.00
  2:Meat Lovers:$13.00
  3:Capricciosa:$14.00
Enter your choice by first letter
View, Add, Delete, Quit: Q
```

The data file has been updated. If we execute the `cat` command on the file, we see that there are now five pizzas.

```
% cat pizzaSimpleData.txt
5
Four Cheese
12.5
Nettuno
11.0
Capriccioso
14.0
Alfredo
14.5
Meat Lover's
13.5
```

17.2 Subclasses and Superclasses

17.2.1 Extending Existing Classes and Interfaces

The pizza menu building program that we have just seen has only the name and the price as information for a pizza. We will extend the program by adding information about the ingredients. We also add a search function that allows the user to look for pizzas having an ingredient. In developing this program, we will recycle `PizzaSimpleInt` and `PizzaSimple` and develop a new interface `PizzaComplexInt` an its implementation `PizzaComplex`. We will then slightly modify `PizzaSimpleCollection` and `PizzaSimpleMain` to obtain `PizzaComplexCollection` and `PizzaComplexMain`. To recycle the source code for `PizzaSimpleInt` and the source code for `PizzaSimple`, we use the keyword of `extends` as follows:

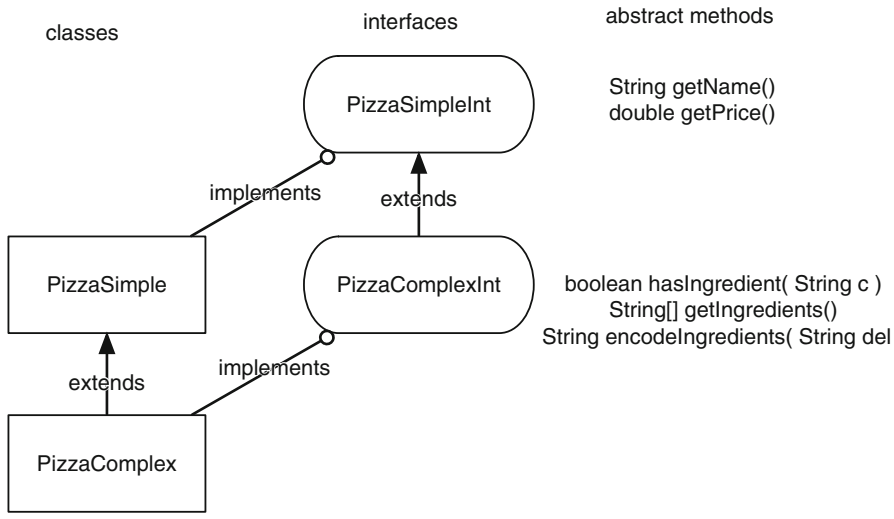

Fig. 17.1 The two interfaces and the two classes. The pointy arrows represent extensions and the circle-head arrows represent implementations

```
public interface PizzaComplexInt extends PizzaSimpleInt
```

and

```
public class PizzaComplex extends PizzaSimple implements PizzaComplexInt
```

When `X` extends `Y`, the unit type must agree between `X` and `Y`, meaning that `X` is a class if and only if `Y` is a class, and that `X` is an interface if and only if `Y` is an interface. If the type of the two units is class, we say that `X` is a **subclass** of `Y` and `Y` is a **superclass** of `X`. If the type is interface, we say that `X` is a **sub-interface** of `Y` and `Y` is a **super-interface** of `X`. As for the source codes we have at hand, `PizzaComplex` is a subclass of `PizzaSimple`, `PizzaSimple` is a superclass of `PizzaComplex`, `PizzaComplexInt` is a sub-interface of `PizzaSimpleInt`, and `PizzaSimpleInt` is a super-interface of `PizzaComplexInt`.

By declaring `X extends Y`, all non-private components that appear in `Y` are automatically included in `X`. This means that the two abstract methods of `PizzaSimpleInt` are already declared (as abstract methods) in `PizzaComplexInt`, and all the public parts of `PizzaSimple` (the constructor and the two getters) are available in `PizzaComplex`. This phenomenon is called **inheritance** too.

If a class `X` extends a class `Y`, the constructor of `Y` can be referenced using the keyword of `super`. The class `X` may override public methods in `Y`. If `X` does that, the pre-override version of the method can be referred to by attaching the prefix of `super.`

Figure 17.1 presents the relations among the two interfaces and the two classes.

Here is the code for `PizzaComplexInt`. Three new abstract methods are introduced.

```
1 public interface PizzaComplexInt extends PizzaSimpleInt
2 {
3   public boolean hasIngredient( String in );
4   public String[] getIngredients();
5   public String encodeIngredients( String del );
6 }
```

Listing 17.10 The code for `PizzaComplexInt`

We expect these abstract methods, when implemented, to operate as follows:

1. `hasIngredient( String c )` returns a `boolean` value representing whether or not the pizza has an ingredient whose name matches the keyword `c`.
2. `getIngredients()` returns the list of ingredients as an array of `String` data.
3. `encodeIngredients( String del )` returns a one-line encoding of the ingredients of the pizza, with the `String` represented by `del` appearing in between.

   The reason that the method `encodeIngredients` receives a delimiter as a parameter is that one-line encodings of ingredients are generated in two different situations. One is when presenting the information of a pizza on the screen. The other is when recording the information of a pizza in a data file. For the former, we may want to use a very short delimiter so that there will be wraparounds. For the latter, we may want to use a delimiter (such as the tab-stop) that does not appear in the names of pizzas. By parameterizing the delimiter, it is possible to handle the two different situations using just one method.

17.2.2 Writing Extensions

17.2.2.1 Writing a Subclass

Including the two abstract methods declared in `PizzaSimpleInt`, `PizzaComplexInt` has a total of five abstract methods. Since `PizzaComplex` has the `extends PizzaSimple` declaration, `PizzaComplex` only has to implement the three abstract methods that are introduced in `PizzaComplexInt`. We will use a private `String[]` instance variable named `ingredients` to represent the ingredients, where each element of the array is the name of one ingredient (Line 3). The constructor of `PizzaSimple` requires two parameters, `name` and `price`. The constructor of `PizzaComplex` has one more parameter. This parameter is an array of `String` data, `ingredients` (Line 5). Because `PizzaComplex` is a subclass of `PizzaSimple`, a `PizzaSimple` object is automatically included in a `PizzaComplex` object (as shown in Fig. 17.2). If the constructor of `PizzaComplex` does not call the `PizzaSimple`, the instance variables declared in `PizzaSimple` will have their default values (that is, `null` for `name` and 0 for `price`). Since `PizzaSimple` does not have setters for `name` or `price`, it is thus necessary to call the constructor of `PizzaSimple` to assign values to `name` and `price`. The way to call for the `PizzaSimple` constructor is:

```
    super( name, price )
```

This is reminiscent of `this(...)` that is used to refer to another constructor of the same object class. The remaining action of the constructor `PizzaComplex` is to copy the value of the parameter

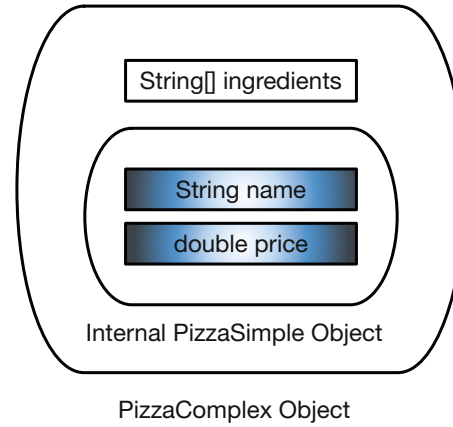

Fig. 17.2 A `PizzaComplex` object

`ingredients` to the instance variable `ingredients`. Because the parameter and the instance variable have the same names, the code attaches the prefix `this.` to the instance variable to distinguish it from the parameter. As in the case of the constructor call `this(...)`, the constructor of the superclass `super(...)` must be called at the start of the constructor.

```
1  public class PizzaComplex extends PizzaSimple implements PizzaComplexInt
2  {
3    private String[] ingredients;
4
5    public PizzaComplex( String name, double price, String[] ingredients )
6    {
7      super( name, price );
8      this.ingredients = ingredients;
9    }
10
```

Listing 17.11 The code for `PizzaComplex` (part 1). The header, the instance variable, and the constructor

The three abstract methods appearing in `PizzaComplexInt` are overridden in `PizzaComplex` as follows: The method `getIngredients` returns the array `ingredients` (Lines 11–14). A program that receives an ingredient list through the method `getIngredients` from a `PizzaComplex` object cannot shrink or extend the array, but can change the individual ingredients of the array. This is because the method provides a reference to the instance variable `ingredients`. To ensure that such modifications will never occur, the method can return a copy of the array instead of the array itself. Such a copy can be generated by `Arrays.copyOf( ingredients, ingredients.length )`.

```
11   public String[] getIngredients()
12   {
13     return ingredients;
14   }
15
```

Listing 17.12 The code for `PizzaComplex` (part 2). The implementation of the method `getIngredients`

To implement `hasIngredient`, the program uses a for-loop called a **"for-each" loop**. Given an array `a` whose elements are of type `T`,

```
for ( T x : a ) { ... }
```

means to iterate over the elements of `a` with the iteration variable `x`. A for-each loop is available for all arrays and all classes that implement an interface `Iterable`. We will discuss the interface `Iterable` in the next chapter. In the case of arrays, the elements appear in the increasing order of their indexes.

In the source code of `PizzaComplex`, we use a for-each loop with the following expression (Line 18): `for ( String in : ingredients )` This loop retrieves the elements of the array `ingredients` one after another and stores them in the variable `in`. The method checks whether or not the keyword `c` appears somewhere in the name `in`. However, since the user may switch between the lowercase and the uppercase on some letter when typing the search key, the method uses the uppercase version of `c` and the uppercase version of `in` instead (Line 20). This means that the search is not case-sensitive. If there is a match, the method returns `true` immediately (Line 22). Completing the loop without returning means that there was no match. For this reason, the method returns `false` (Line 24).

```
16   public boolean hasIngredient( String c )
17   {
18     for ( String in : ingredients )
19     {
20       if ( in.toUpperCase().indexOf( c.toUpperCase() ) >= 0 )
21       {
22         return true;
23       }
24     }
25     return false;
26   }
27
```

Listing 17.13 The code for `PizzaComplex` (part 3). The implementation of the method `hasIngredientsPizzaComplexInt`

The implementation of `encodeIngredients` (Line 28) uses a `StringBuilder` (Line 30). The method uses a for-loop to iterate over the sequence `0, ..., ingredients.length - 1` with a variable named `j` (Line 31). The method appends the `String ingredients[ j ]` (Line 33), and then if `j` is not the last one (Line 34), appends the delimiter `del` (Line 36). This has the effect of concatenating all the ingredients with `del` in between. After completing the loop, the method returns the `String` represented by the `StringBuilder` object (Line 39).

```
28    public String encodeIngredients( String del )
29    {
30      StringBuilder builder = new StringBuilder();
31      for ( int j = 0; j <= ingredients.length - 1; j ++ )
32      {
33        builder.append( ingredients[ j ] );
34        if ( j < ingredients.length - 1 )
35        {
36          builder.append( del );
37        }
38      }
39      return builder.toString();
40    }
41  }
```

Listing 17.14 The code for `PizzaComplex` (part 4). The implementation of the method `encodeIngredients`

17.2.2.2 A Class for a Collection of `PizzaComplex` Data

In `PizzaComplexCollection`, the type of the pizza array is `PizzaComplexIt[]` (Line 5). The class defines two delimiters to be used as a parameter for the method `encodeIngredients` of `PizzaComplexInt`. The delimiters are called `SEPARATOR` and `COMMA`. `SEPARATOR` is used when recording the data in a file and its value is `"\t"` (Line 8). `COMMA` is used when printing the ingredients on the screen, and its value is `", "` (Line 9). As before, there are two constructors, one that receives a file path (Line 11) and the other that takes a `File` object (Line 15). The order between the two is deliberately switched in this version, to emphasize that the order of appearance of the two constructors can be arbitrary.

```
1   import java.util.*;
2   import java.io.*;
3   public class PizzaComplexCollection
4   {
5     private PizzaComplexInt[] list;
6     private File theFile;
7
8     private static final String SEPARATOR = "\t";
9     private static final String COMMA = ", ";
10
11    PizzaComplexCollection( String name ) throws FileNotFoundException
12    {
13      this( new File( name ) );
14    }
15    PizzaComplexCollection( File f ) throws FileNotFoundException
16    {
17      theFile = f;
18      list = new PizzaComplexInt[ 0 ];
19      read();
20    }
21
```

Listing 17.15 The code for `PizzaComplexCollection` (part 1). The header, the constants, the instance variables, and the constructor

The data file now has three lines for each pizza. The third line is an encoding of the ingredients generated by calling `encodeIngredients( SEPARATOR )`. The data file looks like this one:

```
6
Four Cheese
14.0
Mozzarella      Parmesan        Ricotta Gorgonzola
Pepperoni
10.0
Pepperoni
Nettuno
14.0
Tuna    Pasta Sauce     Onion   Green Pepper
Capricciosa
14.5
Olive   Ham     Mozzarella      Artichoke
Meat Lovers
15.0
Pepperoni       Ham     Sausage Bacon   Mozzarella
Veggie
12.5
Onion   Green Pepper    Tomato  Mozzarella      Olive
```

The method `read` is similar to the previous one. To split the ingredient line to an array, `read` calls the method `split` for `String` with `SEPARATOR` as the delimiter (Line 33). The name, price, and ingredients that are read from the file are then given to the constructor of `PizzaComplex`, and the `PizzaComplex` object generated by the constructor is stored in the list (Line 34).

```
22   public void read() throws FileNotFoundException
23   {
24     if ( theFile.exists() )
25     {
26       Scanner scanner = new Scanner( theFile );
27       int size = Integer.parseInt( scanner.nextLine() );
28       list = new PizzaComplexInt[ size ];
29       for ( int index = 0; index < size; index ++ )
30       {
31         String name = scanner.nextLine();
32         double price = Double.parseDouble( scanner.nextLine() );
33         String[] ingredients = scanner.nextLine().split( SEPARATOR );
34         list[ index ] = new PizzaComplex( name, price, ingredients );
35       }
36       scanner.close();
37     }
38   }
39
```

Listing 17.16 The code for `PizzaComplexCollection` (part 2). Reading from file

The method `write` works similarly to the method `write` of `PizzaSimple`. It records the data in the file specified by `theFile` (Line 40). The difference from the previous version is that there is the third line for each a pizza. The method uses the method call `encodeIngredients( SEPARATOR )` to generate the encoding for the third line (Line 48).

```
40   public void write() throws IOException
41   {
42     PrintStream stream = new PrintStream( theFile );
43     stream.println( list.length );
44     for ( int index = 0; index < list.length; index ++ )
45     {
46       stream.println( list[ index ].getName() );
47       stream.println( list[ index ].getPrice() );
48       stream.println( list[ index ].encodeIngredients( SEPARATOR ) );
49     }
50     stream.close();
51   }
52
```

Listing 17.17 The code for `PizzaComplexCollection` (part 3). The method `write`

The method `add` (Line 53) works similarly to the method `add` that is defined in `PizzaSimple`. It receives the name, price, and ingredients for a new pizza and adds the `PizzaComplex` object instantiated with the three values at the end of the list (Lines 56–57). It uses `Arrays.copyOf` to generate a copy of the present list with one additional slot (Line 55). The method `delete` (Line 61) works in the same manner as before.

```
53   public void add( String name, double price, String[] ingredients )
54   {
55     PizzaComplexInt[] listNew = Arrays.copyOf( list, list.length + 1 );
56     listNew[ list.length ]
57         = new PizzaComplex( name, price, ingredients );
58     list = listNew;
59   }
60
61   public void delete( int index )
62   {
63     PizzaComplexInt[] listNew = Arrays.copyOf( list, list.length - 1 );
64     for ( int pos = index + 1; pos < list.length; pos ++ )
65     {
66       listNew[ pos - 1 ] = list[ pos ];
67     }
68     list = listNew;
69   }
70
```

Listing 17.18 The code for `PizzaComplexCollection` (part 4). The methods `add` and `delete`

The method `view` works in a very similar manner to the method `view` in `PizzaSimple` (Line 71). This time, there is an ingredient list to be attached after the price. The list is generated using `encodeIngredients` with `COMMA` as the delimiter (Line 77). The method `search` takes `String` data `c` as the search key (Line 81). For each element in the list, `search` calls the method `hasIngredient` with `c` as the search key (Line 85), and if `hasIngredient` returns `true`, prints the pizza contents using the same format as the format `view` uses (Lines 87–89).

```
71    public void view()
72    {
73      for ( int index = 0; index < list.length; index ++ )
74      {
75        System.out.printf( "%3d:%s:$%.2f:%s\n", index,
76            list[ index ].getName(), list[ index ].getPrice(),
77            list[ index ].encodeIngredients( COMMA ) );
78      }
79    }
80
81    public void search( String c )
82    {
83      for ( int index = 0; index < list.length; index ++ )
84      {
85        if ( list[ index ].hasIngredient( c ) )
86        {
87          System.out.printf( "%3d:%s:$%.2f:%s\n", index,
88              list[ index ].getName(), list[ index ].getPrice(),
89              list[ index ].encodeIngredients( COMMA ) );
90        }
91      }
92    }
93  }
```

Listing 17.19 The code for `PizzaComplexCollection` (part 5). The methods `view` and `search`

The main class `PizzaComplexMain` works in a very similar manner to `PizzaSimpleMain`. The major differences are as follows:

- The pizza menu is now represented by a `PizzaComplexCollection` object (Line 6).
- The menu for actions that is presented to the user now includes an option for searching (Line 21).
- To add a new pizza, the user is asked to enter the name, the price, and then the ingredients with commas in between (Line 32). The input is then split using the comma as the delimiter (Line 33). The array becomes the third parameter of the method call to `add`.
- The switch statement has one additional anchor with the literal `'S'` (Line 40). This anchor is for directing the action to search. The program receives a search key from the user (Line 41), and calls the method `search` (Line 42).

```
import java.util.*;
import java.io.*;
public class PizzaComplexMain
{
  public static void main( String[] args ) throws IOException {
    PizzaComplexCollection data;
    Scanner keyboard = new Scanner( System.in );
    String input = "";
    if ( args.length > 0 )
    {
      data = new PizzaComplexCollection( args[ 0 ] );
    }
    else
    {
      System.out.print( "Enter data file name: " );
      data = new PizzaComplexCollection( keyboard.nextLine() );
    }

    do {
      System.out.println( "Enter your choice by first letter" );
      System.out.print( "View, Add, Delete, Search, Quit: ");
      input = keyboard.nextLine();

      switch ( input.charAt( 0 ) )
      {
        case 'V': data.view(); break;
        case 'A':
          System.out.print( "Enter name: " );
          String name = keyboard.nextLine();
          System.out.print( "Enter price: " );
          double price = Double.parseDouble( keyboard.nextLine() );
          System.out.print( "Enter ingredients separated by comma: " );
          String[] ing = keyboard.nextLine().split( "," );
          data.add( name, price, ing );
          break;
        case 'D':
          System.out.printf( "Enter index: " );
          data.delete( Integer.parseInt( keyboard.nextLine() ) );
          break;
        case 'S':
          System.out.printf( "Enter key: " );
          data.search( keyboard.nextLine() );
      }
    } while ( !input.startsWith( "Q" ) );
    data.write();
  }
}
```

Listing 17.20 The code for `PizzaComplexMain`

Suppose the name of the `PizzaComplex` menu data file is `pizzaComplexData.txt`. Here is an execution example that starts with this file as the data file:

```
% java PizzaComplexMain
Enter data file name: pizzaComplexData.txt
Enter your choice by first letter
View, Add, Delete, Search, Quit: V
  0:Four Cheese:$14.00:Mozzarella, Parmesan, Ricotta, Gorgonzola
  1:Pepperoni:$10.00:Pepperoni
  2:Nettuno:$14.00:Tuna, Pasta Sauce, Onion, Green Pepper
  3:Capricciosa:$14.50:Olive, Ham, Mozzarella, Artichoke
  4:Meat Lovers:$15.00:Pepperoni, Ham, Sausage, Bacon, Mozzarella
  5:Veggie:$12.50:Onion, Green Pepper, Tomato, Mozzarella, Olive
Enter your choice by first letter
View, Add, Delete, Search, Quit: A
Enter name: Hawaiian
Enter price: 13.00
Enter ingredients separated by comma: Ham,Pineapple
Enter your choice by first letter
View, Add, Delete, Search, Quit: V
  0:Four Cheese:$14.00:Mozzarella, Parmesan, Ricotta, Gorgonzola
  1:Pepperoni:$10.00:Pepperoni
  2:Nettuno:$14.00:Tuna, Pasta Sauce, Onion, Green Pepper
  3:Capricciosa:$14.50:Olive, Ham, Mozzarella, Artichoke
  4:Meat Lovers:$15.00:Pepperoni, Ham, Sausage, Bacon, Mozzarella
  5:Veggie:$12.50:Onion, Green Pepper, Tomato, Mozzarella, Olive
  6:Hawaiian:$13.00:Ham, Pineapple
Enter your choice by first letter
View, Add, Delete, Search, Quit: S
Enter key: ham
  3:Capricciosa:$14.50:Olive, Ham, Mozzarella, Artichoke
  4:Meat Lovers:$15.00:Pepperoni, Ham, Sausage, Bacon, Mozzarella
  6:Hawaiian:$13.00:Ham, Pineapple
Enter your choice by first letter
View, Add, Delete, Search, Quit: Q
```

17.3 Polymorphism

An object of class `PizzaComplex` can be treated as a data of type `PizzaComplex`, type `PizzaComplexInt`, type `PizzaSimple`, and of `PizzaSimpleInt`. To treat it as a different (but legitimate) type, we use casting. Casting is accomplished by placing, in front of the data, the alternate data type in parentheses. For example, to treat a `PizzaComplex` object `p` as a `PizzaSimple` object, we write: `(PizzaSimple)p` When a data is treated as a super-type, we call it **up-casting**. When a data is treated as a subtype, we call it **down-casting**. Up-casting is always possible, but down-casting may not be possible. To treat a `PizzaSimple` object as a `PizzaComplex` object, we need to ensure that the object is indeed a `PizzaComplex` object. Suppose we have an array of `PizzaSimpleInt` type, some of whose elements may be `PizzaComplex` objects. To see if an element can be treated as a `PizzaComplex` object, we can use a special operation `instanceof`:

```
( x instanceof T )
```

The operation examines the type of `x`, and returns `true` if the actual type of `x` can be treated as type `T`.

We can write, for instance, the following code to utilize the casting and `instanceof`:

```
1  public void pizzaTest( PizzaSimpleInt p )
2  {
3    System.out.println( p.getName() );
4    System.out.println( p.getPrice() );
5    if ( p instanceof PizzaComplex )
6    {
7      String[] in = ( (PizzaComplex)p ).getIngredients();
8    }
9  }
```

We call the idea that an object can be treated as more than one type **polymorphism**. In Java, every object is a subclass of class `Object`. This organizational structure comes in handy when one wants to create an array of objects with mixed types.

As mentioned earlier, the run-time errors are in a tree. For instance, `FileNotFoundException` is a subclass of `IOException`, `IOException` is a subclass of `Exception`, `Exception` is a subclass of `Throwable`, and `Throwable` is a subclass of `Object`.

17.4 Boxed Data Types

The primitive data types are not object types, so they are not subclasses of `Object`. The Java Development Kit contains many useful classes and interfaces that take data types parameters (see the next chapter). Those parameters cannot be primitive data types. To make such classes and interfaces available for primitive data types, Java offers eight object data types that correspond to the eight primitive data types. We call these data types the **boxed data types**. The eight boxed data types are: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, and `Short`. They correspond to: `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`. Java provides automatic conversions from each primitive data type to its boxed type, and vice versa. For example, if a data of type `int` is supplied where a data of type `Integer` is demanded, Java automatically converts the `int` data to an equivalent `Integer` data, and if a data of type `Integers` is supplied where a data of type `int` is demanded, Java automatically converts the `Integer` data to an equivalent `int` data. We call the automatic conversion from a primitive data type to its boxed type **auto-boxing**, and the automatic conversion from a boxed data type to its un-boxed data type **auto-unboxing**. The automatic conversion fails to work in the case where the value of `null` needs to be unboxed. This results in `NullPointerException`.

We can write programs using the boxed type of integers with `null` being interpreted as "undefined". For example, consider computing, with a `Scanner` object named `sc`, the maximum of the numbers appearing in an indefinitely long sequence of `int` values. Suppose we use a variable named `max` to record the maximum of the numbers we have received so far. For each number received, the program will compare it with `max` to update `max`. When the first number arrives, there is no previous number. Therefore, the first number must be used as the initial value of `max`. By using a `boolean` variable named `noNumbersYet` that records whether or not at least one number has been received, we can distinguish the case between the first number and the rest as follows:

```
boolean noNumbersYet = true;
int max;
while ( sc.hasNext() )
{
  int nextNumber = sc.nextInt();
  if ( noNumbersYet || max < nextNumber )
  {
    max = nextNumber;
  }
  noNumbersYet = false;
}
if ( noNumbersYet )
{
  System.out.println( "max is undefined" );
}
else
{
  System.out.println( "max is " + max );
}
```

We use the disjunction || in the first condition inside the while-loop to truncate the conditional evaluation. At the end of the loop-body, we set the value of `noNumbersYet` to `false`, so the second time around, the second condition, `max < number`, is tested.

By using `Integer` instead, we can make `max` assume the role of the `boolean` variable.

```
Integer max = null;
while ( sc.hasNext() )
{
  int nextNumber = sc.nextInt();
  if ( max == null || max < nextNumber )
  {
    max = nextNumber;
  }
}
if ( max == null )
{
  System.out.println( "max is undefined" );
}
else
{
  System.out.println( "max is " + max );
}
```

17.5 Interface `Comparable`

`Comparable` is a frequently used interface. Objects of a class implementing this interface can be compared with each other using the method `compareTo`. If a class `T` implements this interface, an array of elements of `T` can be sorted using `Arrays.sort`. Since the `compareTo` method takes as its parameter another object to compare to, the specification of `Comparable` in the declaration of a class is a bit complicated. Instead of simply stating:

```
public class T implements Comparable
```

the declaration must state:

```
public class T implements Comparable <T>
```

The `<T>` part is called the **generic type parameter**. We study generic parameters in Chap. 18. We extend `PizzaComplex` so that its objects are comparable with each other by adding an implementation of `Comparable`. The new class is called `PizzaUltra`.

Since `PizzaUltra` does not have an additional instance variable, the constructor of `PizzaUltra` calls the constructor of the superclass `PizzaComplex` (Line 5) with `super`. All the instance methods in `PizzaComplex` are available in `PizzaUltra`.

The header of the new class is:

```
1 public class PizzaUltra extends PizzaComplex implements
2     Comparable<PizzaUltra>
3 {
4   public PizzaUltra( String name, double price, String[] ingredients )
5   {
6     super( name, price, ingredients );
7   }
8 }
```

The code does not compile in the above form, because the class has not yet overridden the method `compareTo`. The error message shown next is generated when an attempt is made to compile the code.

```
1 PizzaUltra.java:1: error: PizzaUltra is not abstract and does not override
    abstract method compareTo(PizzaUltra) in Comparable
2 public class PizzaUltra extends PizzaComplex implements
    Comparable<PizzaUltra>
3       ^
```

To resolve the problem, we must write a public `int` method `compareTo` that takes another `PizzaUltra` object as a parameter. The header of the method must look like this one:

```
public int compareTo( PizzaUltra o )
```

There are three instance variables in `PizzaComplex`. They are a `String` variable, `name`, a `double` variable, `price`, and a `String[]` variable, `ingredients`. Assuming that no two pizzas on the menu have the same names, we naturally choose to compare pizzas by their names. Because the instance variables are declared to be private variables, `PizzaUltra` does not have direct access to the names. For this reason, the following code for `compareTo` fails:

```
1 ...
2 public int compareTo( PizzaUltra o )
3 {
4   return this.name.compareTo( o.name );
5 }
6 ...
```

with the following error message:

```
PizzaUltra.java:11: error: name has private access in PizzaSimple
    return this.name.compareTo( o.name );
                     ^
PizzaUltra.java:11: error: name has private access in PizzaSimple
    return this.name.compareTo( o.name );
               ^
2 errors
```

To resolve the problem, we use the getter, `getName`, as follows:

```
public class PizzaUltra extends PizzaComplex implements
    Comparable<PizzaUltra>
{
  public PizzaUltra( String name, double price, String[] ingredients )
  {
    super( name, price, ingredients );
  }

  public int compareTo( PizzaUltra o )
  {
    return getName().compareTo( o.getName() );
  }
}
```

Listing 17.21 The code for `PizzaUltra`

Now that we have implemented `Comparable`, we can sort any pizza list. We specifically use the capability at two occasions. One is after reading the data from the data file. The other is after adding a new pizza. The code for the collection class is essentially the same as before, with the use of `PizzaUltra`. Since the elements of the array are now `PizzaUltra`, at the end of the methods `add` and `read`, the array is sorted using `Arrays.sort`. The source code for `PizzaUltraCollection` is presented next with the differences from `PizzaComplexCollection` highlighted:

```
import java.util.*;
import java.io.*;
public class PizzaUltraCollection
{
  private PizzaUltra[] list;
  private File theFile;

  private static final String SEPARATOR = "\t";
  private static final String COMMA = ", ";

  PizzaUltraCollection( String name ) throws IOException
  {
    this( new File( name ) );
  }
```

Listing 17.22 The code for `PizzaUltraCollection` (part 1)

```
15    PizzaUltraCollection( File f ) throws IOException
16    {
17      theFile = f;
18      list = new PizzaUltra[ 0 ];
19      read();
20    }
21
22    public void read() throws FileNotFoundException
23    {
24      Scanner scanner = new Scanner( theFile );
25      int size = Integer.parseInt( scanner.nextLine() );
26      list = new PizzaUltra[ size ];
27      for ( int index = 0; index < size; index ++ )
28      {
29        String name = scanner.nextLine();
30        double price = Double.parseDouble( scanner.nextLine() );
31        String[] ingredients = scanner.nextLine().split( SEPARATOR );
32        list[ index ] = new PizzaUltra( name, price, ingredients );
33      }
34      Arrays.sort( list );
35      scanner.close();
36    }
37
38    public void write() throws IOException
39    {
40      PrintStream stream = new PrintStream( theFile );
41      stream.println( list.length );
42      for ( int index = 0; index < list.length; index ++ )
43      {
44        stream.println( list[ index ].getName() );
45        stream.println( list[ index ].getPrice() );
46        stream.println( list[ index ].encodeIngredients( SEPARATOR ) );
47      }
48      stream.close();
49    }
50
51    public void add( String name, double price, String[] ingredients )
52    {
53      PizzaUltra addition = new PizzaUltra( name, price, ingredients );
54      PizzaUltra[] listNew = Arrays.copyOf( list, list.length + 1 );
55      listNew[ list.length ] = addition;
56      Arrays.sort( listNew );
57      list = listNew;
58    }
59
60    public void delete( int pos )
61    {
62      PizzaUltra[] listNew = new PizzaUltra[ list.length - 1 ];
63      for ( int i = 0; i < pos; i ++ )
64      {
65        listNew[ i ] = list[ i ];
66      }
```

Listing 17.23 The code for `PizzaUltraCollection` (part 2)

```
67      for ( int i = pos + 1; i < list.length; i ++ )
68      {
69        listNew[ i - 1 ] = list[ i ];
70      }
71      list = listNew;
72    }
73
74    public void view()
75    {
76      for ( int index = 0; index < list.length; index ++ )
77      {
78        System.out.printf( "%3d:%s:$%.2f:%s%n", index,
79            list[ index ].getName(), list[ index ].getPrice(),
80            list[ index ].encodeIngredients( COMMA ) );
81      }
82    }
83
84    public void search( String c )
85    {
86      for ( int index = 0; index < list.length; index ++ )
87      {
88        if ( list[ index ].hasIngredient( c ) )
89        {
90          System.out.printf( "%3d:%s:$%.2f:%s%n", index,
91              list[ index ].getName(), list[ index ].getPrice(),
92              list[ index ].encodeIngredients( COMMA ) );
93        }
94      }
95    }
96  }
```

Listing 17.24 The code for `PizzaUltraCollection` (part 3)

The main class, `PizzaUltraMain`, has the same code as `PizzaComplexMain`, except that it now uses `PizzaUltraCollection` instead of `PizzaComplexCollection`.

```
1   import java.util.*;
2   import java.io.*;
3   public class PizzaUltraMain
4   {
5     public static void main( String[] args ) throws IOException {
6       PizzaUltraCollection data;
7       Scanner keyboard = new Scanner( System.in );
8       String input = "";
9       if ( args.length > 0 )
10      {
11        data = new PizzaUltraCollection( args[ 0 ] );
12      }
13      else
14      {
15        System.out.println( "Enter data file name: " );
16        data = new PizzaUltraCollection( keyboard.nextLine() );
17      }
18
```

Listing 17.25 The code for `PizzaUltraMain` (part 1)

```
19      do {
20        System.out.println( "Enter your choice by first letter" );
21        System.out.print( "View, Add, Delete, Search, Quit: ");
22        input = keyboard.nextLine();
23
24        switch ( input.charAt( 0 ) )
25        {
26          case 'V': data.view(); break;
27          case 'A':
28            System.out.print( "Enter name: " );
29            String name = keyboard.nextLine();
30            System.out.print( "Enter price: " );
31            double price = Double.parseDouble( keyboard.nextLine() );
32            System.out.print( "Enter ingredients separated by comma: " );
33            String[] ing = keyboard.nextLine().split( "," );
34            data.add( name, price, ing );
35            break;
36          case 'D':
37            System.out.printf( "Enter index: " );
38            data.delete( Integer.parseInt( keyboard.nextLine() ) );
39            break;
40          case 'S':
41            System.out.printf( "Enter key: " );
42            data.search( keyboard.nextLine() );
43        }
44      } while ( !input.startsWith( "Q" ) );
45      data.write();
46    }
47  }
```

Listing 17.26 The code for `PizzaUltraMain` (part 2)

Summary

- An interface is a template for a class.
- An interface may declare static methods and constants.
- Each instance method declared in an interface is abstract.
- Each component appearing an interface must have the public visibility.
- An interface cannot declare instance variables.
- An interface cannot be instantiated, but can be used as a data type.
- To declare formally that a class is built upon an interface, the keyword `implements` must be used.
- The keyword `implements` appears after the class name in the class declaration.
- A class that implements an interface must override all the abstract methods appearing in the interface.
- All the components of an interface are available to each class that implements it.
- An interface can be extended to another interface.
- A class can be extended to another class.
- The keyword to use in declaring an extension is `extends`.
- The prefixes of "super-" and "sub-" are used to refer to the original unit that is extended and the unit that extends the original respectively.

- The public components of a superclass are available to its subclasses. They can be accessed as if they are part of the subclasses.
- The private components of a superclass are not available to its subclasses.
- Inheritance refers to a concept that an implementation has all the public components of an interface it implements and that a subclass can use all the public components of a subclass.
- When a class has multiple constructors and one constructor uses another, the other constructor is referenced to by `this(...)`.
- For a class to initialize the instance variables defined in its superclass, `super(...)` is used to call the constructor of the superclass.
- `this(...)` and `super(...)` must appear at the start of the constructor.
- An instance method of a superclass can be called with the prefix of `super`.
- It one class/interface `X` extends another, `Y`, a data of type `X` can be treated as type of `Y`. We call this phenomenon polymorphism.
- To check whether or not an object `x` can be treated as a data of type `T`, the operator `instanceof` can be used as: `x instanceof T`.
- A boxed data type is an object-class version of a primitive data type. There are eight boxed data types corresponding to the eight primitive data types.
- Java automatic converts between a primitive data type and its boxed type. The automatic conversion from a primitive data type to its boxed type is called auto-boxing and the conversion in the reverse direction is called auto-unboxing.
- `Comparable` is an interface that defines an abstract method `compareTo`. If a class implements this interface, the objects from the class can be compared.
- Declaring an implementation of `Comparable` requires a type parameter.

Exercises

1. **Implementing an Interface `MyCarInt`** Let `MyCarInt` be an interface defined as follows:

```
1 public interface MyCarInt
2 {
3   public int getYear();
4   public String getMake();
5   public String getModel();
6 }
```

   Write a class named `MyCar` that implements this interface.

2. **Implementing an Interface `DogInt`** Let `DogInt` be an interface defined as follows:

```
1 public interface DogInt
2 {
3   public int getAge();
4   public String[] getBreed();
5   public boolean hasBlood( String key );
6 }
```

We assume that the age of a dog is represented in months. The first method returns the age of the dog as an `int` representing the number of months. The method returns an array of `String` data representing the names of the breeds in the mix, for instance, { `"Yorkshire Terrier"`,} {`"Dachshund", "Maltese"` }. The method `hasBlood` checks if a breed name given as the parameter matches one of the elements in the breed list. Write a class named `Dog` that implements this interface.

3. **Writing an interface for `BankAccount`** We previously studied a class `BankAccount` for recording the name and the balance of a bank account. Suppose that the class is one that implements an interface `BankAccountInt` in which all the instance methods appear as abstract methods. Write this interface.
4. **Writing an interface `DateInt`** Consider the following interface, which is for recording the year, the month, and the day value of a date on or after January 1, 1900.

```
1  public interface DateInt
2  {
3    public int getYear();
4    public int getMonth();
5    public int getDay();
6    public static boolean isLeapYear( int year )
7    {
8      ...
9    }
10 }
```

The static method `isLeapYear` returns a `boolean` representing whether or not the year that the formal parameter specifies is a leap year. After the introduction of the Gregorian calendar in the year 1582, the determination of a leap year is made using the following rule: a year Y is a leap year if and only if Y is either a multiple of 400 or a multiple of 4 and not a multiple of 100. For example, the year 2000 is a leap year but 2100 is not one. Assuming the year to be tested for a leap year is greater than 1582, write the static method `isLeapYear`.

5. **Implementing an interface `DateInt`** Write a class named `DataZero` that implements the interface `DateInt` from the previous question. The class has one constructor. The constructor receives three `int` values as parameters. The three values represent the year, the month, and the day. Write the constructor so that if the combination of year, month, and day is invalid, it throws an `IllegalArgumentException`.
6. **Implementing an interface `DateInt`** Continuing on the previous question, write a new class `DateNew` that extends `Date` and implements `Comparable<DateNew>`.

Programming Projects

7. **Comparable class `StudentBasicInt`** Write an interface named `StudentBasicInt` for recording information of a single student. The interface has three abstract methods that are expected to be implemented as getters, `String getFamilyName()`, `String getOtherNames()`, and `int getRank()`. The expected actions of these methods are to return the family name, the other names, and an integer representing the academic rank (1 for freshman, 2 for sophomore, 3 for junior, and 4 for senior). The interface should define `int` constants, `FRESHMAN`, `SOPHOMORE`, `JUNIOR`, and `SENIOR` representing these four ranks. The interface must define setters

`void setFamilyName( String o )`, `void setOtherNames( String o )`, and `void setRank( int o )`, which store the names and the rank.

8. **Implementing the interface `StudentBaseInt`** Write a class named `StudentBase` that implements the interface `StudentBaseInt` from the previous question. Furthermore, write a class `StudentBaseMaster` with just one static method `public static StudentBaseInt create( String a, String b, int c )` that returns a `StudentBaseInt` object instantiated with the three parameters `a`, `b`, and `c`, that represent the family name, the other names, and the rank.
9. **Writing an application for reading and presenting `StudentBaseInt` data** Write an application class named `StudentBaseApp` that reads `StudentBaseInt` data from a file, and stores it in the same file after modifications. Possible modifications are adding a new student and changing information of a student. The file path must be received from the user. The data file must contain the number of students in the first line. After stating the number of students in the data file, the information for each student appears in three lines. The three lines representing the family name, the other names, and the rank. The program should store the student record in an array of `StudentBaseInt` and use `StudentBaseMaster` to instantiate each element of the array.