COMP 401

Prasun Dewan[1]

Collections and Inheritance

In the previous chapter, we defined logical structures with fixed number of nodes. In this chapter, we will see how we can use arrays to define types with a dynamic number of components. The collectiom examples will motivate the main topic of this chapter – inheritance. Inheritance in an object-oriented language is not much different than inheritance in the real world. Just it is possible to automatically inherit money from a benefactor or genes from ancestors, it is possible to inherit code from an object type – specifically the methods and variables defined by that type. The idea of inheritance is related to the IS-A relation. Again, this relation is used in the non-computer world. A human is a mammal; a salmon is a fish; and each of you is a student. Similarly, we say that ACartesianPointy is a Point because it implements Point. We will see that inheritance, like interface implementation, is also an is-a relation. We will show how type checking of object variables is based on this relation.

Example Problem

To motivate collections, let us consider the problem of implementing the console-based application shown in Figure ???. (replace the figures with actual text for mobile reading)

```
<terminated> AHistoryDriver
James Dean
p
******************
James Dean
******************
Joe Doe
Jane Smith
p
******************
James Dean
Joe Doe
Jane Smith
******************
q
```

Users of this application can enter names, print the set of entered names, and quit the application. A line beginning with q or P is considered a command to quit the application or print the names, respectively. Any other line is considered the entry of a new name.

Linked Lists vs. Arrays

The application above must collect the elements in memory in order to later print them. There is no limit on the number of elements a user can enter. Thus, we need to create a data-structure in memory that can grow dynamically. Later, we will see predefined Java collections such as Vector, ArrayList and LinkedList that provide dynamic growth. Let us try to see if we can implement such collections using the tools about which we already know.

There are two main ways in in which we can implement such collections. One is to create a "linear" logical structure in which each node refers to the next element in that collection. The structure is linear because a parent has exactly one child – there is no branching. Such a structure links all of the collection nodes, and thus is called a linked-list. In your data structure course, you will explore this technique in great depth; and we will not further discuss it.

The other technique is to use arrays, which is the one on which we will focus here. Arrays, by themselves, are not sufficient as they hold a fixed number of elements. However, it is possible to use a fixed-size array to store a variable-size collection, much as we write variable-sized lists in fixed-size pages. Such a collection can be simulated using an array and two `int` variables. The size of the array is the maximum size of the collection, given by one of the `int` variables. The filled part consists of elements of the variable-sized collection and the unfilled part consists of the unused elements of the array. The size of the filled part, thus, ranges from 0 to the array length. We store this value in the second `int` variable, and update it whenever we add or delete a collection element. Thus, changing the size of the collection involves changing the boundary between the filled and unfilled parts of the array. Since the length of an array cannot change once it has been instantiated, the "variable" storing the value of the maximum size of the collection is usually declared as `final` to make it readonly.

Encapsulating Related Variables

It is considered good programming practice to relate the names of the three variables defining the variable-sized collection. A popular approach is to declare these variables in the class that needs the collection, and name the current and maximum size variables `aSize`, and `A_MAX_SIZE`, respectively, if the array is named `a`.

```
final int A_MAX_SIZE = 50;
String[] a = new String[A_MAX_SIZE];
int aSize = 0;
//process collection a
…
```

If the class needs another collection of this kind, it similarly creates another three variables:

```
final int B_MAX_SIZE = 50;
String[] b = new String[B_MAX_SIZE];
int bSize = 0;
//process collection b
…
```

We can, in fact, do better than this, since each time we need a group of related variables, we should declare them as instance variables of a new class. Here is one way of doing so:

```
public class AVariableSizedStringCollection {
      public final int MAX_SIZE = 50;
      public String[] contents = new String [MAX_SIZE];
      public int size = 0;
}
```

In this declaration, the three variables are declared as public instance variables of the class. Each time we need a new dynamic collection of strings, we can create a new instance of this class and store it in some variable:

```
AVariableSizedStringCollection a = new AVariableSizedStringCollection();
AVariableSizedStringCollection b = new AVariableSizedStringCollection();
```

We can then refer to the array and size components of it, as shown below:

```
a.MAX_SIZE
a.contents
a.size
b.MAX_SIZE
 ….
```

This approach is superior to creating separate variables, such as `aSize` and `a`, because it creates a new type that better matches the problem than the separate array and `int` types do individually. As a result, it gives us the advantages of defining high-level types such as reuse of the declarations defining the variable size collection and allowing a dynamic collection to be returned by a function. However, it does not meet the principle of encapsulation or least privelege. Because both the non-final instance variables have been declared public, users of this class can manipulate them in arbitrary ways. For instance, they can add a new element to the array without incrementing the size field; or make the size field become larger than MAX_SIZE.

History

Therefore, we should *encapsulate* the variables, that is, make them non-public and provide access to them through public methods. These methods depend on the nature of the collection we want. In the problem of Figure ???, we need operations to:

- add an element to the collection,

- examine each element of the collection so that we can print it.

We do not need operations to modify or delete elements. Thus, the collection we need is really a *history* of strings – as in the case of a history of past events, its filled portion cannot be overwritten.

The following interface defines these operations:

```
public interface StringHistory {
      public void addElement(String element);
      public String elementAt (int index);
      public int size();
}
```

Like `String`, the interface provides a method to access an element at a particular index. It is, of course, more convenient to index a collection like an array using the square brackets, `[` and `]`. Unfortunately, in Java, such indexing cannot be used for programmer-defined types, requiring special support from the programming language.

Like `String`, this interface also provides a method to find the size of the collection. Finally, unlike the immutable `String`, it provides a method to add an element to the collection.

The three variables creating a variable-size storage for the history elements are defined, not in the interface, but in its implementation, `AStringHistory`:

```
public class AStringHistory implements StringHistory {
      public final int MAX_SIZE = 50;
      String[] contents = new String[MAX_SIZE];
      int size = 0;
      public int size() {
            return size;
      }
      public String elementAt (int index) {
            return contents[index];
      }
      boolean isFull() {
            return size == MAX_SIZE;
      }
      public void addElement(String element) {
            if (isFull())
                  System.out.println("Adding item to a full history");
            else {
                  contents[size] = element;
                  size++;
            }
      }
}
```

Unlike `AVariableStringCollection`, `AStringHistory` does not make these instance variables public. As a result, it supports encapsulation – all access to these variables is through the public methods

of the class. There is no danger these variables will be manipulated in an inconsistent manner from outside the class. Moreover, if we were to change the variables by, for instance, renaming them, increasing the maximum size, or replacing an array with another data structure (such as Vector, discussed later), the users of the class would not know the difference and thus could be reused with the new implementation.

Most of the methods of this class are trivial. The method `size` returns the current size, `elementAt` indexes the array to return its result, and `isFull` returns true if the current size has reached its maximum value.

The method `addElement` requires some thought. If the collection is not full, the value of `size` is the index of the first element of the unfilled part of the array. In this case, it simply assigns to this element the new value, and increments `size`. The tricky issue here is what should happen when the collection is full? If we have guessed the maximum size right, this condition should not arise. Nonetheless, we must decide how to handle this situation. Some choices are:

- *Subscript Exception*: We could ignore this situation, and let Java give an `ArrayIndexOutOfBoundsException` error when the method accesses the array beyond the last element. However, this may not be very illuminating to the user, who may not know that the implementation uses an array (rather than, say, a vector, discussed later).
- *Specialized Exception:* We could define a special "exception" for this situation, `CollectionIsFullException`, and "throw" it. We will see later how this is done.
- *Print Message*: We would print a message for the user. Unfortunately, the caller of `addElement` gets no feedback with this message.
- *Increase Size*: If it is not an error to add more elements than the original array can accommodate, we could increase the size of the collection by creating a larger array, add all the elements of the previous array to it, and assign the new array to the contents field.

Our program assumes it is an error to add to a full collection and simply prints a message. As the discussion above points out, had we known how to define one at this point, a specialized exception would have been a better alternative under this assumption.

The following main method illustrates how `StringHistory` and its implementation `AStringHistory`, are used to solve our problem:

```
public static void main(String[] args) {

      StringHistory names = new AStringHistory(); // create an empty history
      while (true) {
             String input = Console.readString();
             if (input.length > 0)
                   if (input.charAt(0) == 'q')
                         // exit loop if user enters 'q'
                         break;
                  else if (input.charAt(0) == 'p')
```

```
                        // print database if user enters 'p'
                        print(names);
                else // add input to history
                        names.addElement(input);
                }
        }
```

It gathers history entries incrementally in the loop. If a non-null input line does not begin with any of the recognized commands, the loop assumes it is a new entry, and calls the `addElement` method to add it to the `StringHistory` instance. The `print` method calls the size() and elementAt() methods of the history to access all elements of the history.

```
static void print(StringHistory strings) {
        System.out.println("******************");
        for ( int elementNum = 0; elementNum < strings.size(); elementNum++)
                System.out.println(strings.elementAt(elementNum));
}
```

Database

A history is perhaps the simplest example of a variable-sized collection. Let us motivate the need for a more sophisticated collection using an extension of our example application in Figure ???:

```
AStringDatabase [Java Application] C:\Program F
James Dean
Joe Doe
Jane Smith
p
******************
James Dean
Joe Doe
Jane Smith
******************
m Joe Doe
true
m Jane Doe
false
d Joe Doe
p
******************
James Dean
Jane Smith
******************
c
p
******************
******************
```

In addition to the commands to add to and print the collection, this application provides commands to delete an entry (d), check if an entry is a member of the collection (m), and clear the whole collection (c). Thus, the collection forms a simple string *database,* providing commands for searching, adding, and deleting entries. The following interface describes the new type:

```
public interface StringDatabase {
    // methods of StringHistory
    public String elementAt(int index);
    public void addElement(String element);
    public int size();
    // additional methods
    public void deleteElement(String element);
    public void clear();
    public boolean member(String element);
}
```

It retains all the methods of the `StringHistory`, including additional methods to delete an element, clear the collection, and check if an item is present in the collection.

We will not cover the details of implementing the new methods as these are more relevant for a CS-1 course. The complete code of AStringHistory is given in the companion source code.

Inheritance

We have seen two kinds of collections created using arrays, a history and a database. A history is defined by the interface, `StringHistory` and implemented by the class, `AStringHistory`; while a database is defined by the interface, `StringDatabase`, and implemented by the class `AStringDatabase`.

Let us compare the database class and interfaces with the history class and interface. The database interface defines all the methods of the history interface, plus some more. In other words, *logically*, it is an "extension'' of the history interface, containing copies of the methods `elementAt`, `addElement`, and `size`, and adding the methods, `deleteElement`, and `clear`. Similarly, logically, the database class is an extension of the history class, in that it contains a copy of all of the members defined in the latter – the `size` and `contents` instance variables, and the `addElement`, `size`, and `elementAt` instance methods. As a result, `AStringDatabase` implements a superset of the functionality of `AStringHistory`. However, *physically*, the database interface and class are not extensions of the history interface and class, because they duplicate code in the latter – each member of the history interface/class is re-declared in the database interface/class.

In fact, Java allows us to create logical interface and class extensions as also physical extensions. The database interface can share the declarations of the history interface as shown here:

```
public interface StringDatabase extends StringHistory {
      public void deleteElement(String element);
      public void clear();
      public boolean member(String element);
}
```

The keyword **extends** tells Java that the interface `StringDatabase` is a physical extension of `StringHistory`, which implies that it implicitly includes or *inherits* the constants and methods declared in the latter. As a result, only the additional declarations must be explicitly included in the definition of this interface.

Below, we see how the database class can share the code of the history class:

```
public class AStringDatabase
                  extends AStringHistory implements StringDatabase {
      public void deleteElement(String element) { … }
      int indexOf(String element) { … }
      void shiftUp(int startIndex) { … }
      public boolean member(String element) { … }
      public void clear() { … }
}
```

The method bodies are not given here since they are the same as we saw earlier. The important thing to note here is that the class does not contain copies of the methods and instance variables declared in `AStringHistory` because it now (physically) extends it.

Given an interface or class, A, and an extension, B, of it, we will refer to A as a *base* or *supertype* of B; and to B as a *derivation, subtype,* or simply *extension* of A

Any class that implements an extension of an interface must implement all the methods declared in both the extended interface and the extension. Thus, in our example, `AStringDatabase` must implement not only the methods declared in `StringDatabase`, the interface it implements, but also the ones declared in the interface, `StringHistory`, the interface extended by `StringDatabase`. Thus, the new definition of `StringDatabase` and `AStringDatabase` are equivalent to the ones given before.

Here we were good about following the rule of putting all public methods of the extending and extended class in interfaces. To better understand the Java semantics, let us consider a version of AStringDatabse in which we remove the implements clause without making any other change:

```
public class AStringDatabase
                    extends AStringHistory {
      …
}
```

Even though this class has no explicit implements clause, as it is an extension of AStringHistory, it implicitly implements StringHistory. In general, a class implicitly implements the interfaces of its (direct and indirect) superclasses.

Real-World Inheritance

Programming using objects, classes, and inheritance is called an *object-oriented programming*. In contrast, a programming using only objects and classes is called *object-based programming*. Thus, with the use of inheritance, we are making a transition from object-based programming to object-oriented programming.

Let us go back to the real-world analogy to better understand inheritance and why it is important. Often physical products are extensions of other physical products. For instance, a deluxe model of an Accord has all the features of a regular model, and several more such as cruise control. When specifying the deluxe model, it is better to create an addendum to the existing specification of a regular model, rather than create a fresh specification. As a result, if we later decide to update the specification of a regular model, we do not have to go back and update the specification of the deluxe model, which is always constrained to be an extension of a regular model.

Similarly, we may want to implement an extended interface by extending a factory, rather than creating a new factory. For instance, when we need to create a deluxe model, we may first send it to a factory that creates a regular model, and then add new features to this model.

We do not have to look at the man-made world for examples of these relationships. For instance, a human is a (inherits traits of a) primate, which is a mammal, which is an animal, which is a living organism, which is a physical object; and, a rock can be directly classified as a physical object. Thus, both a human and a rock inherit properties of physical objects – for instance, we can see, touch, and feel them.

Thus, like objects and classes, inheritance feels "natural" and allows us to directly model the inheritance relationships among physical objects simulated by the computer. For example, we could model a human being as an instance of a `Human` Java type, which would be a subtype of `Primate`, and so on. Without inheritance, we would have to manually create these relationships.

The Class Object

Just as, in the real world, we defined the group, `PhysicalObject`, to group all physical objects in the universe and define their common properties, Java provides a class, *Object,* to group all Java objects and define their common methods. It is the top-level class in the inheritance hierarchy Java creates for classes (**Error! Reference source not found.**). If we do not explicitly list the superclass of a new class, Java automatically makes `Object` its superclass. Thus, the following declarations are equivalent:

```
public class AStringHistory implements StringHistory
```

and

```
public class AStringHistory extends Object implements StringHistory
```

The methods defined by `Object`, thus, are inherited by all classes in the system. An example of such a method is `toString`, which returns a string representation of the object on which it is invoked. The implementation of this method in class `Object` simply returns the name of its class followed by an internal address (in hexadecimal form) of the object. Thus, an execution of:

```
System.out.println((new AStringHistory()).toString())
```

might print:

```
AStringHistory@1eed58
```

while an execution of:

```
System.out.println((new AStringDatabase()).toString())
```

might print:

```
AStringDatabase@1eed58
```

where "leed58" is assumed to be the internal address of the object in both cases. In fact, we did not need to explicitly call `toString` in the examples above. `println` automatically calls it when deciding how to display an object. Thus:

```
System.out.println(new AStringDatabase))
```

is, in fact, equivalent to:

```
System.out.println((new AStringDatabase()).toString())
```

`Object` defines other methods, we will study later, which can also be invoked on all Java objects.

Despite its name, `Object` is a class, and not an instance. It defines the behavior of a generic Java object, hence the name.

It defines a variety of methods, three of which, shown in the figure below, we will look in some depth later. The class Object does not implement any interface. Not only is this bad design, according to our rule that every class must implement one (or more) interfaces, but it raises some conceptual problems in variable assignment, as we will see later.

Overriding Inherited Methods and super Calls

Returning to our example application, suppose we did not want to print or store duplicates in the database, and instead wanted the interaction shown in Figure ???.

```
Driver [Java Application] C:\Program File
James Dean
John Smith
James Dean
John Smith
p
******************
James Dean
John Smith
******************
```

To support this application, we need the collection to behave like a mathematical (ordered) set, which allows no duplicates. We do not need to completely model a Mathematical set, which would require set operations such as union, intersection, and difference, which we did not need in this problem. Question 6 motivates the use of a more complete implementation of a set. In fact, we do not need to add any operations we defined for a database. Instead we can simply re-implement the `addElement` operation inherited from `AStringHistory` so that it does not add duplicates to the collection:

```
public class AStringSet extends AStringDatabase implements
StringDatabase {
      public void addElement(String element) {
            if (member(element)) // check for duplicates
                  return;
            // code from AStringHistory
            if (isFull())
                  System.out.println("Adding item to a full history");
            else {
                  contents[size] = element;
                  size++;
            }
      }
}
```

What we have done here is to replace or *override* an inherited method implementation.

In the above implementation, we duplicated the code in AStringHistory for adding an element to the collection. We would like to somehow refer in the addElement() of a AStringSet to the addElement() of AStringHistory. If we refer to it addElement():

```
public void addElement(String element) {
        if (member(element)) return;
        addElement(element);
 }
```

we would make a non-terminating recursive call, as the addElement() call refers to addElement() of AStringSet rather AStringHistory.

One solution to this problem is to define another method, addElementHistory(), that is not overridden by AStringSet() and have both the overridden and overriding addElement() in the two classes call this method. Thus, the addElement( ) in AStringHistory is now replaced by the following two methods:

```
public void addElementHistory(String element) {
     if (isFull())
        System.out.println("Adding item to a full history");
     else {
        contents[size] = element;
        size++;
}
 }
public void addElement (String element) {
   addElementHistory(element);}
 }
```

The overridding addElement() can now refer to addElementHistory:

```
public void addElement(String element) {
        if (member(element)) return;
        addElementHistory(element:;
 }
```

There are times we have to resort to such a solution when we override methods. However, for this case, Java provides a more direct solution, shown below:

```
public class AStringSet extends AStringDatabase implements
StringDatabase {
      public void addElement(String element) {
            if (member(element)) // check for duplicates
                  return;
            // code from AStringHistory
            super.addElement(element);
      }
}
```

Here, we directly invoke the addElement() method in the superclass AStringHistory rather than execute a copy of its code or refer to an auxiliary method that is not overdden. In general, when we use the keyword **super** in a class C as a prefix to a method call, we ask Java to follow the super class chain of C

to find the first class with a definition that matches the method call, and invoke that method. The super class chain of C_1, C_2, .. Object, where C_{i+1} is the super class of C_i and C_1 is the super class of C. Thus, the super class chain of AStringSet is AStringDatabase, AStringHistory, Object. When **super**.addElement(element) method invoked by AStringSet, Java first checks if a matching definition is provided by AStringDatabase. As it is not, it then goes to the next super class in the chain, AStringHistory, and performs the same check. As it does find a definition this time, it uses it and does not look further up the chain. The matching method definition can be found at compile time – there is no need to wait until execution time to resolve the call. We could not have used this solution if AStringDatabase had also overridden addElement() and in AStringSet, we wanted to call the addElelement() in AStringHistory rather than AStringDatabase. We would have add to use the solution of creating an auxiliary method in AStringHistory that is not overridden.

In the implementation of AStringSet, we have not implemented a new interface, only provided a new implementation of an existing interface, `StringDatabase` by overriding one of the inherited methods.

To implement the above application, the main method remains the same as the one we used for the database application, except that we replace the line:

```
StringDatabase names = new AStringDatabase();
```

with:

```
StringDatabase names = new AStringSet();
```

When the `addElement` method is invoked on `name`:

```
names.addElement(input);
```

the implementation defined by the class of the assigned value (`AStringSet`) is used since it overrides the inherited implementation of the operation from `AStringDatabase`.

Overridding toString()

To better understand inheritance and Object, the last class in the superclass chain of all classes, let us try and understand in more depth and override its toString() method in `AStringSet`:

```
public String toString() {
      String retVal = "";
      for (int i = 0; i < size; i++)
            retVal += ":" + contents[i];
      return retVal;
}
```

The method returns a ":" separated list of the elements of the collection:

```
stringSet.toString() → "James Dean:John Smith"
```

Recall that the implementation inherited from Object gives us the class name followed by the memory address:

```
stringSet.toString() → "AStringSet@1eed58"
```

Many classes override the `toString` method, since the default implementation of it inherited from `Object` is not very informative, returning, as we saw before, the class name followed by the object address. Recall also that `println` calls this method on an object when displaying it. The reason why `println` tends to display a reasonable string for most of the existing Java classes is that these classes have overridden the default implementation inherited from Object.

Protected, Private and Default Visibility of Variables and Methods

Our final implementation of AStringSet consists of a single overridden method, which does access any non-public variables or methods of its superclasses. This leads to the question: should non-public members - variables and methods - declared in a class even be visible in suberclasses? The answer, it turns out, is depends. While AStringSet does not need to access its superclass members, AStringDatabase does. For example, to delete an element or clear the collection, the class must change the size variable.

To support examples such as AStringDatabase, Java could allow a subclass to access all variables and methods of a super class, but that breaks the principle of least privilege, which says that a class should be given access to only those variables and methods to which it needs access. Therefore, Java allows programmers to choose which non-public members are visible in subclasses. For similar reasons, it allows them to choose which non-public members are visible in other classes in the same package.

In other words, when it comes to sharing, some classes are more equal than others. The subclasses of a class and classes in the same package may be tied to its implementation, and thus, deserve more access than those that simply use its interface. To make a real world analogy, consider the implementation of a car. Given a regular engine, we may want to extend it to deluxe higher-rev engine by allowing the latter to access the implementation details of the former that are not visible, for instance to other components of the car such as the tires. This is analogous to giving subclasses special access. Similarly, we may want the distributor and other parts that interact with the engine to make it work to have special access to the engine that other parts such as tires do not have. This is like giving co-packaged classes special access, as the classes in the same package, together, implement some logical unit of the entire program.

Java defines a hierarchical scheme of access which associates each member with one four access levels:

- public: It is accessible in all classes.
- protected: It is visible in all subclasses of its class and all classes in its package.
- default: It is accessible in all classes in its package.
- private: It is accessible only in its class.

We saw earlier that to make a variable or method public, we simply use the keyword **public.** Similarly, we can use the keywords **protected** and **private** to give these two reduced levels of access. We give default access by using none of these three keywords – hence the name, default. Thus, the opposite of public access is not private access as Java defined four access levels. So far, we have given only public and default access. We now know how to give two additional levels of access.

Let us look at some examples to illustrate these access levels. Suppose we put AStringHistory and AStringDatabase in the same package:

```
package lectures.collections;
 public class AStringHistory implements StringHistory {
  int size = 0;
        ...
package lectures.collections;
 public class AStringDatabase extends AStringHistory
                implements StringDatabase {
  public void clear() {
   size = 0;
  }
        ...
```

Does AStringDatabase have access to the size variable declared in AStringHistory? As size variable has been given default access, and both classes are in the same package, the answer is yes.

If we were to move AStringDatabase to another package:

```
package lectures.inheritance;
 public class AStringDatabase extends AStringHistory
                implements StringDatabase {
  public void clear() {
   size = 0;
  }
```

It would not longer have access to the variable, and the code above would not compile. If we do want the two classes to be in different packages, then to make this code compile, we must change the access of the variable to protected:

```
package lectures.collections;
 public class AStringHistory implements StringHistory {
  protected int size = 0;
  …
```

While variables and implementation-specific methods should not be made public, there is no general rule for deciding which of the three access levels should be given to such members. As mentioned

above, the exact level depends on the specific class being implemented and how we think it may be extended.

What should we do when we cannot anticipate how a class may be extended? To maximize flexibility and reusability, we may make all variables and implementation-specific methods protected so that subclasses in other packages can also use these members without duplicating their function. Java's designers apparently believed that by default, only classes within the same package should access them. When we do not have enough information, we could simply use this principle, as it saves typing an extra keyword. Many software engineers take a far more conservative stand and require all variables to be private and access be given to other classes using getters and setters with the appropriate access level. This approach minimizes the chances that other classes will make the values of these variables inconsistent. While this approach is wonderful in theory, my own experience, based on examining popular open-source software systems, is that programmers are quick to make variables private but lazy about providing non-public getters and setters to access them, thereby limiting flexibility and reusability. Thus, choosing the appropriate access level involves making several tradeoffs involving programming effort, consistency, extensibility, and reusability. You are free to choose how you want to make these tradeoffs. The recommendation is to make all variables private as long as you are not lazy about providing the getter and setters. This recommendation is violated in all code presented in this course, mainly because it bloats the size of the examples presented.

Java Collection Inheritance

We have defined here three different collections types, with the following inheritance relationships among the involved interfaces and classes:

Interfaces: StringSet → StringDatabase → StringHistory
Classes: AStringSet → AStringDatabase → AStringHistory

Collections are so important that, as some of you already know, far more sophisticated versions of them are provided by Java. Here are some example names and inheritance relatonships:

Interfaces: Set, List → Collection
Classes: Vector, ArrayList → AbstractList → AbstractCollection

It defines two interfaces, Set and List, which are both subtypes of Collection. This is analogous to StringDatabase being subtypes of StringDatabase and StringDatabase being a subtype of StringHistory . As we saw before, Vector and ArrayList are two implementations of List. They extend Abstract List, which in turn, extends AbstractCollection. This is analogous to AStringSet extending AStringDatabase , and AStringDatabase extending AStringHistory.

By outlining the design and implementation of our example classes, we are able to gain some understanding of the insides of the predefined collection classes, and more important, from the point of

view of this chapter the inheritance relationships among them. We will study the API provided by them later.

More on Inheritance

There is far more to inheritance than what we have seen here. We will unravel additional aspects of it in later chapters.

Summary

- Java allows classes and interfaces to inherit declarations in existing classes and methods, adding only the definitions needed to extend the latter.
- An inherited method can be overridden by a new method.

Exercises