COMP 110

Prasun Dewan[1]

17. Inheritance

We have seen how we can create arbitrary sequences, such as histories, using arrays. In this chapter, we will use arrays to create several new kinds of types including sets and databases. Some of these types can be considered as special cases of other types. We will see how a specialized type can inherit the code from a more general type much as a child inherits genes from a parent.

Database

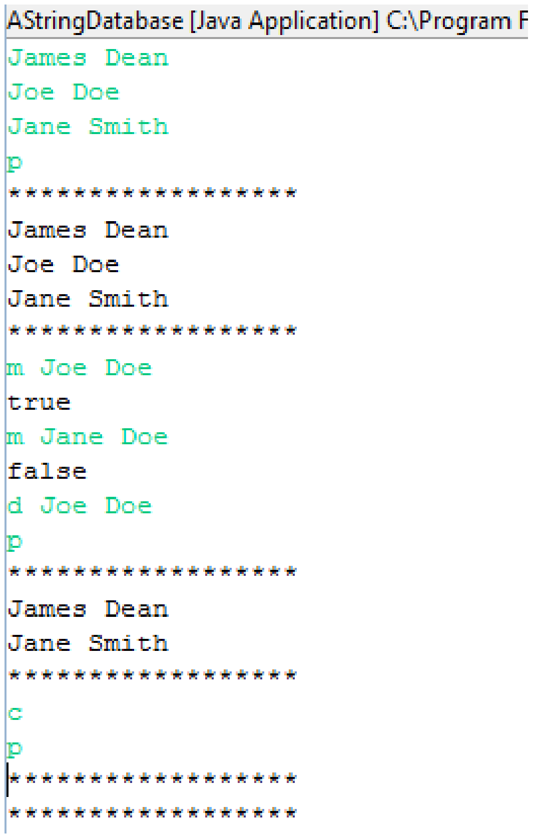
A history, which we saw in the arrays chapter, is perhaps the simplest example of a variable-sized collection. Let us define a more sophisticated collection shown in Figure 1.

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.

[1] © Copyright Prasun Dewan, 2000.



```
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
******************
******************
```

Figure 1. AStringDatabase

Similarly, the implementation of this interface retains the variables and methods of `StringHistory`.

```
public class AStringDatabase implements StringDatabase {
    // code from StringHistory
    …
    // additional code
    …
}
```

containing additional code for the three new operations, which is described below.

Deleting an Entry & Multi-Element List Window

Let us first consider the implementation of the `deleteElement` operation. Deleting an entry from a collection (implemented as an array) is more difficult than adding one. We have assumed that in a collection, the positions of the entries do not matter, or if they do matter, the order in which they are added determines their position (and not, say, the alphabetic order). Therefore, we always assigned the new entry at the first unfilled position of the array, and simply incremented the size field. The delete command allows us to remove an entry from the middle of the array. Our implementation strategy for a

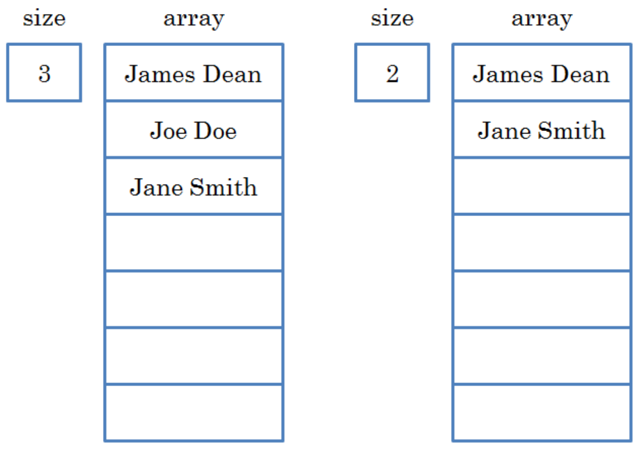


Figure 2. Deleting the second element from a history that contains three elements

variable-size collection assumes that all unfilled entries follow the filled entries. Thus, we cannot leave an unfilled "hole" in the middle of the filled area.

A simple approach would be to take the last element and put it in the slot of the deleted element. However, this does not preserve the order in which the entries were added. Assuming that `print` should list the elements in this order, we must, instead, shift all the elements following the deleted item up one position. Figure 2 shows the contents of the array before and after "Joe Doe" is deleted from the collection.

In order to do the move, we must examine successive pairs of slots in the array, starting from the deleted position, and store the contents of the slot at the larger index in the slot at the lower index. Thus, for each of these pairs, we must either perform the assignment:

```
contents[index] = contents[index + 1]
```

or

```
contents[index - 1] = contents[index]
```

Let us chose the first assignment since it allows us to start at the index at the position of the deleted item. We are now ready for the code required for deletion:

```
public void deleteElement(String element) {
    shiftUp(indexOf(element));
}

void shiftUp(int startIndex) {
    for (int index = startIndex; index + 1 < size; index++) {
        contents[index] = contents[index + 1];
    }
    size--;
}
```

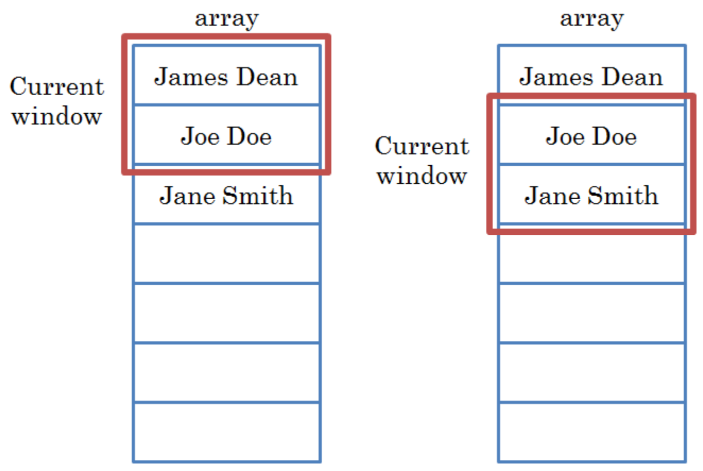


Figure 3. A 2-element window-based list traversal

In our previous list traversals, we examined one entry of the list at a time. In this traversal, we examine two consecutive list elements at a time. Both are examples of *window-based* list traversals, where each list traversal step (loop iteration or recursive step) accesses a fixed size window of neighboring elements in a list, and each subsequent step moves the window up or down by a fixed step as shown in Figure 3.

In a forward (backward) list traversal, the termination condition is the last (first) window element becoming the last (first) list element. This is the reason that our loop for the two-element window is:

```
        index + 1 < size
```

rather than:

```
        index < size
```

which was the condition for the one-element window traversal.

Searching for an Element

Our delete operation assumes we have an operation, `indexOf`, to find the position of a list element. The operation traverses the list using a one-element window, stopping when it finds the element or reaches the end of the list:

```
        int indexOf(String element) {
            int index = 0;
            while ((index < size) && !element.equals(contents[index]))
                index++;
            return index;
        }
```

To decide if the element at the current index is equal to the element being sought, the `indexOf` function uses the predefined `equals` operation in String. The `equals` operation is invoked on a String instance and takes a single formal parameter also of type String. The operation returns `true` if and only

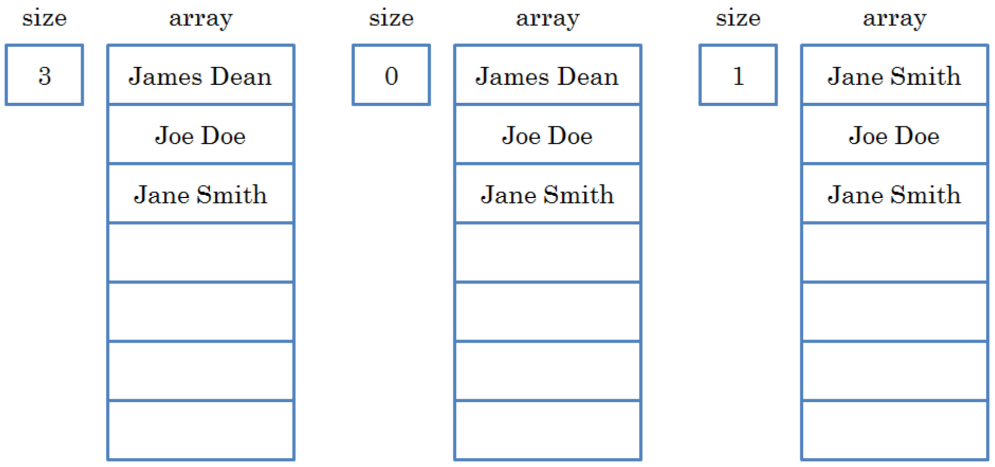


Figure 4. (left) before clear; (middle) after clear; (right) adding a new entry

if the String instance on which it is being invoked is identical (in terms of number of characters and the characters themselves) to the String instance passed as the actual parameter to the formal parameter.

Note that in the case the item exists multiple times in the collection, the `indexOf` function returns the position of the first element that matches the item for which we are searching. If the item is not in the array, it returns the value of the `size` variable. In this case, the loop in `shiftUp` will be skipped and no item will be deleted.

With this method, we can trivially implement the `member` operation:

```
public boolean member(String element) {
    return indexOf(element) < size;
}
```

That leaves us the `clear` method. We could clear the database by deleting each item of the array:

```
public void clear() {
    while (size > 0) {
        deleteElement(size - 1);
    }
}
```

At each iteration of the loop, the element at `size - 1` is the last element in the collection. We can thus clear the database by repeatedly deleting its last item until we have no more items left.

However, it is much simpler and more efficient to make the current size 0:

```
public void clear() {
    size = 0;
}
```

Thus, `clear` (and `deleteElement`) simply adjusts the size field without explicitly removing from the array any element. Figure 4 shows the array contents before and after the clear operation.

As we can see, the "deleted" elements still occupy space in the array. However, they will be replaced with any new items we add to the collection – in that sense they do not take space in the array. For instance, if we were to now add the string "Joe Doe" to the collection, it would replace "James Dean" in the first element of the array. Thus, the difference between deleted and undeleted elements is that the slots of the latter are in the "unfilled" space and are used to add new elements.

When the first slot is reassigned the string, "Joe Doe", what happens to the old value of the slot, "James Dean"? Clearly, it is no longer in the array, but is it also no longer in memory? This string is essentially "garbage" in that we are no longer interested in it; thus it should be removed from memory. Java does *garbage collection* to automatically find and de-allocate any such unused space from memory. In languages such as Pascal and C++ that do not do garbage collection, we would have to explicitly dispose of this space. It is very easy to make mistakes by either accidentally de-allocating useful space or not de-allocating useless space. Java's garbage collection, thus, makes our programs easier to write and more reliable. How Java does garbage collection is beyond the scope of this course. What is important is that, like garbage collection in the real world, Java's garbage collection is not done each time some garbage is created, but later, according to criteria determined by the Java garbage collector, often when space is running low. You might see your program slow down when the garbage collector becomes active.

Switch and Ordinal Types

Now that we have seen how the various methods of `AStringDatabase` are implemented, let us complete the implementation of the database problem by giving the main method:

```
public static void main(String args[]) {
      StringDatabase names = new AStringDatabase();
      while (true) {
            String input = Console.readString();
            if (!(input.length() == 0))
                  if (input.charAt(0) == 'q')
                        break;
                  else if (input.charAt(0) == 'p')
                        print(names);
                  else if (input.charAt(0) == 'd')
                        names.deleteElement(input.substring(
                              2, input.length()));
                  else if (input.charAt(0) == 'm')
                        System.out.println(names.member(
                              input.substring(2, input.length())));
                  else if (input.charAt(0) == 'c')
                        names.clear();
                  else
                        names.addElement(input);
      }
}
```

Instead of creating an instance of `AStringHistory`, the method creates an instance of `AStringDatabase`. The loop of this method extends the loop of the main method we created for the history example by processing the additional commands for deleting an entry, clearing the database, and checking for membership. The clear command is processed like the other commands we have seen so far since it is also a one-character command. The other two commands take an operand after the command character. The program extracts this operand using the `substring` operation and passes it as an argument to the method for processing the command.

Notice that we have been careful to do all the branching in the else parts of the if statements that discriminate among the different characters. As it turns, we can further improve the code by using an alternative conditional, called a *switch statement*, illustrated below:

```
public static void main(String args[]) {
      StringDatabase names = new AStringDatabase();
      while (true) {
            String input = Console.readString();
            if (!(input.length() == 0))
                  if (input.charAt(0) == 'q')
                        break;
                  else switch (input.charAt(0)) {
                        case 'p':
                              print(names);
                              break;
                        case 'd':
                              names.deleteElement(
                                    input.substring(2,
                              input.length()));
                              break;
                        case 'm':
                              System.out.println(names.member(
                                    input.substring(2,
                              input.length())));
                              break;
                        case 'c':
                              names.clear();
                              break;
                        default:
                              names.addElement(input);
      }
}
```

The switch statement selects among different values of the expression following the **switch** keyword called the *switch expression*. Each of the listed values is a *case* of the switch expression, and the statement sequence following it is called an *arm* of the switch. If the value of the switch expression is equal to a particular case, then control transfers to the corresponding arm.

As shown above, Java does not require us to list all possible cases for the switch expression. The default clause stands for all unlisted cases.

It is possible to associate multiple cases with the same arm, as shown below:

```
switch (input.charAt(0)) {
      case 'p', 'P':
            print(names);
            break;
      case 'd', 'D':
            names.deleteElement(input.substring(2, input.length()));
            break;
            …
}
```

The switch conditional is shorter, easier to read, and as it turns out, more efficient than an if-else conditional. While an if-else conditional does a two-way branch to the then or the else part, a switch statement does a multi-way branch to its different arms.

A switch is, not, however, suitable for all selection problems, since the type of the switch expression must be an *ordinal type*. An ordinal type is a primitive type with the following properties:

1) The instances/literals of the type are ordered.
2) For each literal, there is a unique successor/predecessor unless it is the last/first literal.

For instance, it can be used to test an `int` but not a `String` or `float` expression. Thus, if-else statements are more general but less high-level conditionals than switch statements.

Notice that we put a break statement at the end of each arm of the switch. In general, once it finishes executing an arm, a switch statement executes *all* of the subsequent arms until it finds a break**.** Consider what happens when we forget to put break statements

```
switch (input.charAt(0)) {
      case 'p':
            print(names);
      case 'd':
            names.deleteElement(input.substring(2, input.length()));
      case 'm':
            System.out.println(names.member(
                  input.substring(2, input.length())));
      case 'c':
            names.clear();
      default:
            names.addElement(input);
```

and the input character is 'm'. The statement will executes the 'm' arm and all of the arms of the cases below it, thus executing the statements:

```
names.deleteElement(input.substring(2, input.length()));
System.out.println(names.member(input.substring(2, input.length())));
names.clear();
names.addElement(input);
```

The break statement makes the program jump out of the immediately enclosing statement block, which may be a loop or a switch. It is not possible to use it to break out a statement block that is not immediately enclosing it. Consider the following alternative main method:

```
public static void main(String args[]) {
      StringDatabase names = new AStringDatabase();
      while (true) {
            String input = Console.readString();
            if (!(input.length() == 0))
                  switch (input.charAt(0)) {
                        case 'q':
                              break;
                        case 'p':
                              print(names);
                              break;
                        case 'd':
                              names.deleteElement(
                                    input.substring(2,
                              input.length()));
                              break;
                        case 'm':
                              System.out.println(names.member(
                                    input.substring(2,
                              input.length())));
                              break;
                        case 'c':
                              names.clear();
                              break;
                        default:
                              names.addElement(input);
      }
}
```

This solution is more elegant in that it avoids the if conditional that tests if the input character is 'q', replacing it with an extra arm in the switch. Unfortunately, it does not work, because the break in the new arm terminates the enclosing switch, not the loop. Since in this example, the main method does not execute any statement after the loop, we can execute return instead of break in the arm corresponding to 'q':

```
switch (input.charAt(0)) {
      case 'q':
            return;
      …
```

This solution will work since the main method will return (to the interpreter) when this arm is executed, terminating the program.

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`. Figure 5 shows the members (methods and instance variables) declared in the interfaces and classes.

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() { … }
}
```

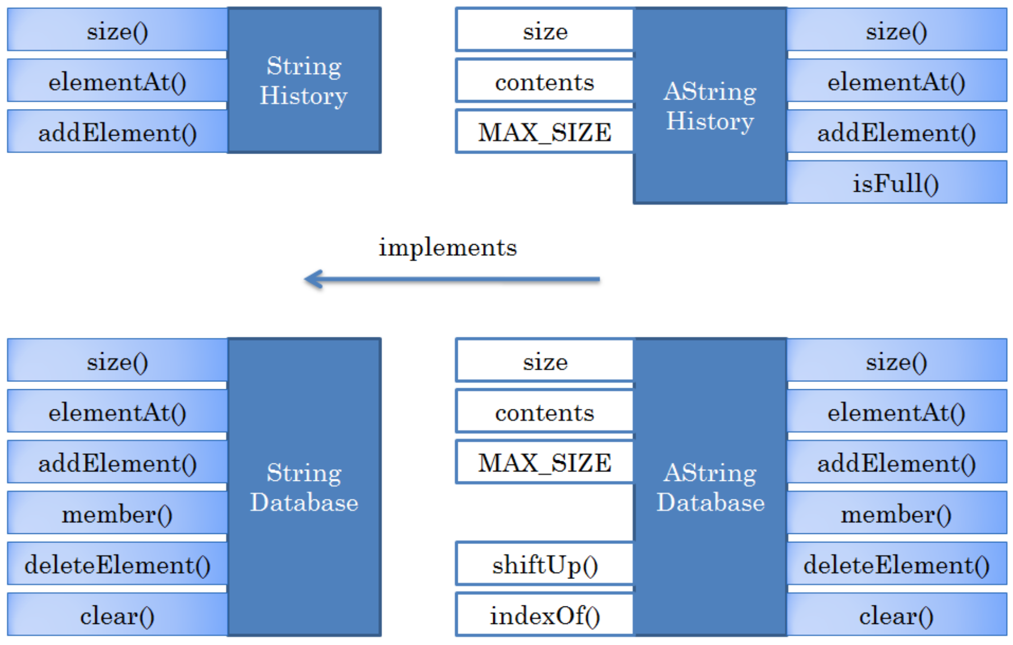


Figure 5. Logical but not physical extensions

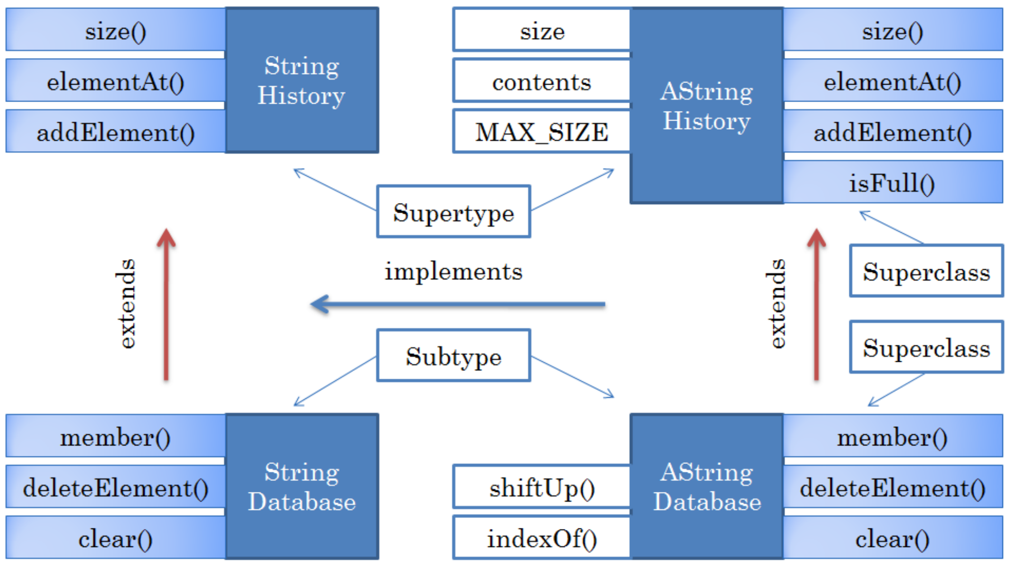


Figure 6. Physical and logical extensions

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 (Figure 6).

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.

Why Inheritance

There are several reasons for extending interfaces and classes, as we have done above, rather than creating new ones from scratch, as we did before:

- *Reduced Programming/Storage Costs*: The most obvious reason is that we do not have to write and store on the computer a copy of the code in the base type, thereby reducing programming and storage costs. The programming cost, of course, is minimal if we had a convenient facility to cut and paste. However, the source code of the base type may not always be available, which does not prevent it from being sub typed.
- *Easier Evolution*: Code tends to change. We may decide to change the `MAX_SIZE` constant of `AStringHistory`, in which case, would have to find and change all other classes that are logical but not physical extensions.
- *Polymorphism*: Inheritance allows us to support new kinds of polymorphism, as explained below.
- *Modularity* & *Reusability*: We assume above that the base class and interface already existed when we created subclasses of them. For instance, we assumed first that we needed string histories and created appropriate interfaces and classes to support them. Later, when we found the need for string databases, we simply extended existing software.

What if we have not had the need for string histories, and were told to create string histories from scratch? Even in this case, we may want to first create string histories and then extend them rather than create unextended string databases, because the extension approach increases modularity, thereby giving the accompanying advantages. In this example, it makes us understand, code, and prove correct the two interfaces and classes separately. Moreover, if we later end up needing string histories, we have the interface and class for instantiating them. The approach requires us to *design for reuse,* something that is very difficult to do in practice.

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

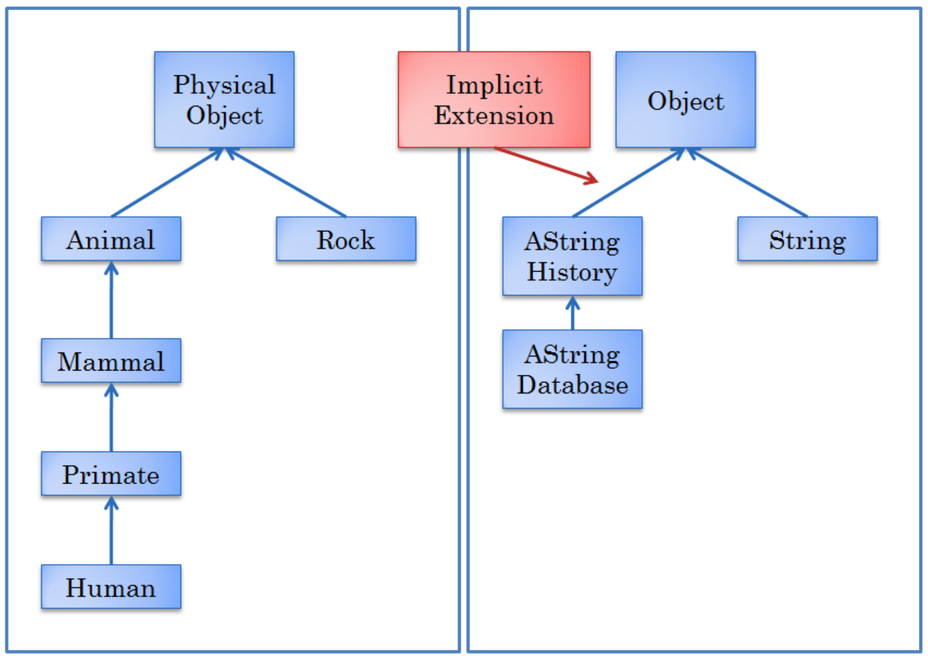


Figure 7. Logical but not physical extensions

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, as shown in Figure 7, a human is 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 (Figure 7). 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, which we will not study here, 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.

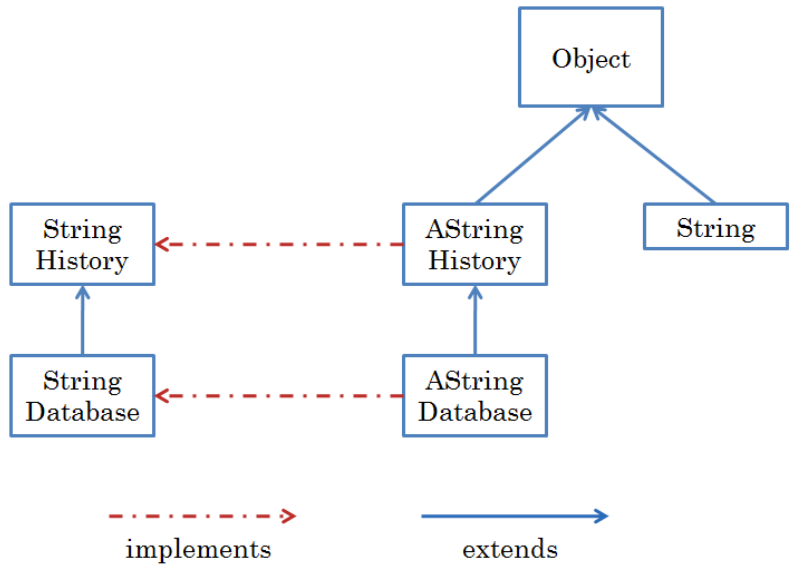


Figure 8. IS-A Relationship

IS-A Relationships

An inheritance relationship between a subtype and a supertype is a special case of the more general IS-A relationship among entities. Intuitively, we might say:

AStringDatabase IS-A AStringHistory

This assertion seems right since a string database is also a string history. In ordinary language, we say some entity e1 IS-A e2 if e1 has all the properties of e2. Thus, a primate is a mammal since it has all the properties of a mammal. In the context of an object-oriented programming language, the entities are object types (classes and interfaces) and their instances (objects), and the properties we use to determine IS-A relationships among them are their public members (methods and variables).

We can now formally define the IS-A relationship among Java object types and their instances. Given two object types, T1 and T2, and arbitrary instances t1 and t2 of these types, respectively:

T1 IS-A T2

is true if

t1 IS-A t2

is true, which in turn, is true, if all public members of t2 are also public members of t1.

From this definition, we can derive, that:

T2 extends T1 => T2 IS-A T1

because every instance of T2 has not only the members declared in its type T2, but also all members declared in the super type of T2, T1. The reverse is not true:

T2 extends T1 => T1 IS-A T2

since T2 can define additional public variables and methods that instances of T1 do not have.

Inheritance is only one example of an IS-A relationship. The implements relationship between a class and an interface is another example:

T2 implements T1 => T2 IS-A T1

because every instance of T2 has all the members defined in T1 (plus, optionally, some more since a class is free to define public members not declared in its interface).

Thus, some IS-A relationships defined by Figure 8 are:

StringDatabase IS-A StringHistory
AStringDatabase IS-A AStringHistory
AStringHistory IS-A StringHistory
AStringDatabase IS-A StringDatabse

In other words, each of the arrows in the figure denotes an IS-A relationship. Figure 8 shows both inheritance and implements forms of IS-A relationships among some of the classes and interfaces we have seen.

The IS-A rule is transitive:

T3 IS-A T2 IS-A T1 => T3 IS-A T1

This follows from transitivity of the inheritance relationship. Thus:

AStringDatabase IS-A StringHistory

By our definition, it is also reflexive:

T1 IS-A T1

Thus:

AStringHistory IS-A AStringHistory

which should not be surprising!

Type Rules

The IS-A relationship gives the basis for type-checking rules in Java. Consider the following declarations:

```
StringHistory stringHistory = new AStringDatabase();
StringDatabase stringDatabase = new AStringHistory();
```

They assign to variable of type T1 an object of another type T2. Should these be allowed?

In the first case, we are trying to assign an instance of `AStringDatabase` to a variable expecting `StringHistory`. Since:

```
AStringDatabase IS-A StringDatabase IS-A StringHistory
```

the assignment is legal. On the other hand, in the second case we are trying to assign an instance of `AStringHistory` to a variable expecting `StringDatabase`. Since `AStringHistory` is not, directly or indirectly, `StringDatabase`, the second assignment is illegal.

To understand what may go wrong in the second assignment, consider the following operation invocation:

```
stringDatabase.clear();
```

Because the type of `stringDatabase` is `StringDatabase`, this invocation will be considered legal at compile time. However, if `stringDatabase` is actually assigned an instance of `AStringHistory`, the instance will not have the `clear` member, and we will get a runtime error.

To understand why the first assignment is safe, consider an operation invocation on `stringHistory`:

```
stringHistory.size();
```

If `stringHistory` has been actually assigned an instance of `AStringDatabase`, the instance is guaranteed to have all the publically accessible members of an instance of `StringHistory`, since indirectly `AStringDatabase` IS-A `StringHistory`.

Given the first assignment:

```
StringHistory stringHistory = new AStringDatabase();
```

should the following be legal?

```
stringHistory.clear();
```

Since `stringHistory` has been assigned an instance of `AStringDatabase`, the instance will have the member, `clear`. However, the compiler will complain. This is because, at compile time, we do not know the exact value of a variable, and have to take the conservative approach of assuming it has only those members that are indicated by its type (and no other member). However, if, at runtime, we are sure about the actual type of the object, we can use a cast, as shown below:

```
((StringDatabase) stringHistory).clear()
```

The cast assures the compiler that the type of the object stored in `stringHistory` is actually `StringDatabase`. Unlike other languages such as C that allow casting, Java keeps the type of a variable at runtime, and will throw an exception if the actual type, T2, does not match the type, T1, given in the cast, that is T2 is not a T1. Thus, if we executed:

```
stringHistory = new AStringHistory();
((StringDatabase) stringHistory).clear()
```

we would get a `ClassCastException`.

We can now precisely state the complete type rules used by the compiler. Assume we assign to some variable v of type T1 an expression e of type T2. T1 and T2 may be interfaces, classes, or primitive types. The assignment is legal if either of the two conditions holds true:

- T2 IS-A T1
- T2 IS-NARROWER-THAN T1

where the narrow relationship was defined in the chapter on types.

Typing an expression is ambiguous if a cast is used:

```
(StringDatabase) stringHistory
```

A cast creates two types for the expression being cast: a static type and a dynamic type. The *static type* is the type used for cast. Thus in the above example, `StringDatabase` is the static type of the expression. The *dynamic type* is the actual type of the expression being cast, which is determined at runtime. Thus, in the above example, it is determined by the object that has been assigned to `stringHistory`. The type checking rules above use the static type at compile time. A separate type-checking phase occurs at runtime, which uses the actual type, T2 of the cast expression to ensure it is compatible with static type, T1, used for casting, that is, T2 IS-A T1, as discussed above.

To understand these type rules intuitively, let us consider again the real world. The following is legal:

```
ARegularModel myCar = new ADeluxeModel();
myCar.accelerate();
```

If our rental or buying plan assumes a regular model, and we are upgraded to deluxe model, that is safe, because all operations on a regular model such as accelerate are also applicable on a deluxe model. However, the following is not legal:

```
ADeluxeModel myCar = new ARegularModel();
myCar.setCruiseControl();
```

If our plan assumes a deluxe model, and we are downgraded to a regular model, we will be unhappy, and potentially unsafe, because some operations such as `setCruiseControl` are applicable only to deluxe models. However, the following is safe:

```
ARegularModel myCar = new ADeluxeModel();
```

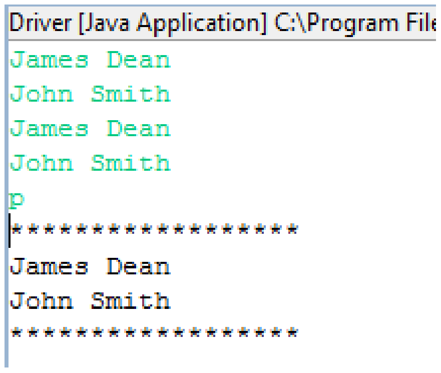

```
Driver [Java Application] C:\Program File
James Dean
John Smith
James Dean
John Smith
p
******************
James Dean
John Smith
******************
```

Figure 9. AStringSet

```
	ADeluxeModel hisCar = (ADeluxeModel) myCar;
```

In other words, we should be able to perform operations of the upgrade that could not be performed on the car we reserved, as long it can be assured that we did indeed get an upgrade, that is, the cast is successful.

Thus, compile-time type checking is equivalent to checking that our plan for driving the car is consistent with the car we have reserved, while runtime checking is equivalent to checking that it is consistent with the actual car we obtained. It is important to note that both checks occur before we use the car in an inappropriate way, for instance, before we actually try to set the cruise control.

Inheritance and Polymorphism

A consequence of our type rules is that the method we defined for printing `StringHistory`:

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

will also work for printing instances of `StringDatabase`. That is, we can safely invoke:

```
	print(stringDatabase);
```

This is because the following assignment is made during parameter passing:

```
	StringHistory strings = stringDatabase;
```

which is allowed by the assignment rules. The only member of the argument accessed by `print` is `elementAt`, which is also a member of `stringDatabase`.

Recall that a method such as `print` that takes arguments of multiple types is called polymorphic. Recall also that creating IS-A relationships via the implements relationships allowed us to write such methods. Here we see that creating IS-A relationships through inheritance also supports such methods. The type checking rules described above have been designed to support polymorphism.

Overriding Inherited Methods

Returning to the database application, suppose we did not want to print or store duplicates in the database, and instead wanted the output shown in Figure 9. To support this application, we need the collection to behave like a mathematical set, which allows no duplicates. To define such a collection, we do not need to add to the operations we defined for a database. Instead we can simply re-implement the `addElement` operation inherited from `AStringHistory` so that does not add duplicates to the collection:

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

What we have done here is to replace or *override* an inherited method implementation. When you study `super`, we will see a more efficient way of overriding methods. We have not implemented a new interface, only provided a new implementation of an existing interface, `StringDatabase`.

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`.

The above collection does not completely model a Mathematical set in that it does not define several useful 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.

To gain more practice with overriding methods, let us override in `AStringSet` the `toString` method inherited from class Object:

```
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.

Summary

- An array is an indexable fixed-size collection of elements of the same type.
- An array type defines a variable structure, that is, different instances of it can have different sizes. The size of an array instance is specified at runtime when it is created.
- An array variable may be initialized or uninitialized, and an array assigned to an initialized array variable may have elements that are themselves uninitialized.
- A dynamic collection can be simulated by a named constant specifying the maximum size of the collection, a variable specifying the current size of the collection, an array for storing the elements of the collection.
- These three components of the collection should be encapsulated in a class and protected from direct external access by public methods.
- Deleting an element of an ordered collection involves moving a two-element window along the array, assigning the second element of the window to the first one.
- A deleted element does not have to be explicitly removed from memory since Java automatically garbage collects unused memory space.
- 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.
- Inheritance and implementation are examples of IS-A relationships.
- If T2 IS-A T1, then a value of type T2 can be assigned to a variable of type T1.

Exercises

1) What is encapsulation and why is it important?

2) What is inheritance and why is it useful?

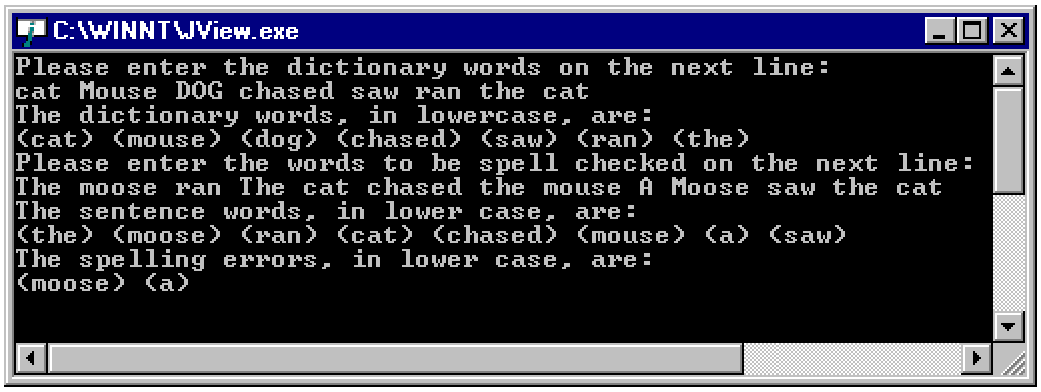
3) When should you use an if-conditional instead of a switch statement?

4) Rewrite the `addElement` method of `AStringHistory` to double the maximum size of the collection when it is invoked on a full collection, and then add the element to this new collection.

5) Create an extension of `AStringSet`, called `ASortedStringSet`, that keeps its elements sorted in ascending order.

6) Use the upper case enumeration of the previous chapter to print the upper case letters in an input string in reverse order. Thus, if the user inputs the string: John F. Kennedy, the program should input the letters: KFJ. You can assume that a string will not have more than 50 uppercase letters. Create a special type defining a character history collection to do this problem.

7) Extend your solution to problem 6 of the previous chapter by creating a spelling checker, shown below. The users of the program will enter, on the first line, a sequence of words they want to put in the dictionary of the program, and then, on the second line, a sequence of words they want spell checked. Let us call the second word sequence the "paragraph." After the paragraph has been entered, your program should output all words in the paragraph that are not in the dictionary:



```
Please enter the dictionary words on the next line:
cat Mouse DOG chased saw ran the cat
The dictionary words, in lowercase, are:
(cat) (mouse) (dog) (chased) (saw) (ran) (the)
Please enter the words to be spell checked on the next line:
The moose ran The cat chased the mouse A Moose saw the cat
The sentence words, in lower case, are:
(the) (moose) (ran) (cat) (chased) (mouse) (a) (saw)
The spelling errors, in lower case, are:
(moose) (a)
```

As before, you can assume that users will enter only letters and spaces in a line and that they will always enter a single space after each word (including the last word). Thus each paragraph/dictionary word will be a contiguous sequence of letters ending with a single blank. You can also assume that a user will not enter more than 50 words in the dictionary or make more than 50 spelling mistakes in the paragraph. Finally, you can assume no syntax errors will be made - so you do not have to do any syntax-error checking. The dictionary and paragraph words may be entered in uppercase or lowercase letters - however case does not influence the spelling check. Thus the words `dog' and `Dog' are to be considered the same. You should not print a dictionary, paragraph, or a misspelled word twice, as shown above. For instance, in the interaction above, both the words 'A' and 'Moose' are misspelled twice - but the program gives only one error for each word.

To avoid code duplication, think of using a set for storing the various collections of words you need to process in this program. The type will extend the set type given in this chapter with other set operations you will need such as intersect.