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.

In this chapter, we will mainly use collections to illustrate and motivate inheritance and the is-a relation. We have seen how we can create certain kinds of sequences, such as histories and databases, using arrays. 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.

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:

```java
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:

```java
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 (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 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 \texttt{Human} Java type, which would be a subtype of \texttt{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:

```java
public class AStringHistory implements StringHistory
```

and

```java
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:

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

might print:

```
AStringHistory@1eed58
```

while an execution of:

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

might print:

```
AStringDatabase@1eed58
```

where “1eed58" 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:

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

is, in fact, equivalent to:

```java
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.

It defines a vari
It defines a variety of methods, three of which, shown in the figure below, we will look in some depth later.

**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:

\[
\text{AStringDatabase IS-A AStringHistory}
\]

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:

\[
T1 \text{ IS-A } T2
\]
is true if

\[ t1 \text{ IS-A } t2 \]

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:

\[ T2 \text{ extends } T1 \implies T2 \text{ IS-A } T1 \]

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:

\[ T2 \text{ extends } T1 \implies T1 \text{ IS-A } T2 \]

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:

\[ T2 \text{ implements } T1 \implies T2 \text{ IS-A } T1 \]

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:

- \( \text{StringDatabase IS-A StringHistory} \)
- \( \text{AStringDatabase IS-A AStringHistory} \)
- \( \text{AStringHistory IS-A StringHistory} \)
- \( \text{AStringDatabase IS-A StringDatabase} \)

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:

\[ T3 \text{ IS-A } T2 \text{ IS-A } T1 \implies T3 \text{ IS-A } T1 \]

This follows from transitivity of the inheritance relationship. Thus:

\[ \text{AStringDatabase IS-A StringHistory} \]

By our definition, it is also reflexive:

\[ T1 \text{ IS-A } T1 \]

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:

```java
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:

```java
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`:

```java
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:

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

should the following be legal?

```java
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:

```java
((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:

```java
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:

```java
(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:

```java
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:

```java
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:

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

```java
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:

```java
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 and super Calls**

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 (ordered) 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:

```java
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 for adding an element to the collection. The
following implementation fixes this problem:

```java
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. 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, C2, .. Object, where Cn is the super class of Ci and C1 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.

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. The notion of overriding is shown pictorially in the following figure.

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:

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

with:

```java
StringDatabase names = new AStringSet();
```
When the `addElement` method is invoked on `name`:

```java
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.

**Inheritance among Geometric Classes**

We have used inheritance above to reuse code in related collections. In fact, it can be used in almost any application. In particular, it is useful to reuse code among different geometric classes, as is demonstrated in depth in the exercises. Here is a simple example to make this point:

```java
public class ABoundedPoint extends ACartesianPoint implements BoundedPoint {
    Point upperLeftCorner, lowerRightCorner;
    public ABoundedPoint (int initX, int initY, Point initUpperLeftCorner, Point initLowerRightCorner) {
        super(initX, initY);
        upperLeftCorner = initUpperLeftCorner;
        lowerRightCorner = initLowerRightCorner;
    }
    public Point getUpperLeftCorner() {...}
    public void setUpperLeftCorner(Point newVal) {....}
    ...
}
```

This class creates an extension of `ACartesianPoint` that confines the point to a rectangular region whose upper and lower right corners are defined by two properties. The exact semantics of the class do not matter – this is the reason why the implementation of the methods of this class is not given. What is important is that this class inherits the instance variables of the super class and can override methods to ensure that the point cannot be moved outside the associated rectangle.

We will use this class, together with `AStringDatabase` and `AStringSet`, to illustrate below some subtle issues with inheritance and the methods of class `Object`.

**Inheritance and Memory Representation**

One of these issues is the memory representation of instances of subclasses. Specifically, are the instance variables defined by the subclass and super classes stored in the same or different memory
blocks? The following figure provides the answer. It shows the memory representation of the following instance of ABoundedPoint():

```java
new ABoundedPoint(75, 75, new ACartesianPoint(50,50), new ACartesianPoint(100,100) )
```

As it shows, when an instance of a class is created, Java allocates a single memory block for the instance variables defined by the class and all classes in its superclass chain. In this example, the block consists of the two primitive variables, x and y, defined by the super class, and the two new Object variables, upperLeftCorner and lowerRightCorner, defined by the class. Recall that each instance variable takes exactly one word and that, as the figure shows, a primitive variable directly stores a value while an object variable stores a memory address or pointer. In this figure, pointers are underlined. As it shows, the upperLeftCorner contains the address 8, which points to an instance of ACartesianPoint, which in turn, is a memory block containing two int variables defining the x and y coordinates.

By storing all variables of an object, Java makes it simpler to allocate, copy, access and free up the memory associated with these variables.
**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 the three methods of it identified above: toString(), equals(), and clone.

Let us begin by overriding in AStringSet the toString() method inherited from class Object:

```java
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.

**equals()**

Java already provides an operator to check for equality, ==, so why do we need a method that, based on its name, seems to do the same thing? To illustrate the difference between == and equals(), consider the following statements:

```java
Point p1 = new ACartesianPoint(200, 200);
System.out.println(p1 == p2);
p1 = new ACartesianPoint (200, 200);
System.out.println(p1 == p2);
```

The == operator dereferences the two pointers, and compares the resulting objects. When the first statement is executed, both p1 and p2 refer to the same object. Therefore, we can expect the first print statement to print “true”. But what about the second print statement? Both variables refer to the same logical point in the coordinate space, the point with the coordinates (200,200). However, they refer to different physical objects, as shown below.
The `==` operator, in fact, simply checks if its left and right hand side are the same physical object. If not, it returns the false value. It does not understand the concept of two physical objects being the same logical entity. It is the responsibility of each object to define a method that checks if two objects represent the same logical entity. The convention is to call this method, `equals()`. Several predefined classes such as `String` provide such a method.

In `String`, this method does a character-by-character comparison of the strings that are compared, and returns true if the two strings have the same sequence of characters. The following interaction shows the difference between `==` and `equals()` for strings:

```java
String s1 = "hello world";
String s2 = "hello world";
System.out.println(s1==s2);
System.out.println(s1.equals(s2));
s1 = s2;
System.out.println(s1==s2);
System.out.println(s1.equals(s2));
```

All print statements except the first one will print “true”. The reason why the first one returns false is that the two strings are stored in separate memory blocks, as shown in the figure below.
Consider now the following code:

```java
Point p1 = new ACartesianPoint(200, 200);
p1 = new ACartesianPoint(200, 200);
System.out.println(p1.equals(p2));

StringHistory stringHistory1 = new AStringHistory();
StringHistory stringHistory2 = new AStringHistory();
stringHistory1.equals(stringHistory2);
```

Both println() calls print `false`. In other words, the equals() method, in these two cases, has the same behavior as `==`. The reason is that we have not redefined equals() for instances of StringHistory and Point and the default behavior of equals() is the same as `==`:

```java
//implementation in Object
public boolean equals(Object otherObject) {
    return this == otherObject;
}
```

Each class that declares instance variables whose values influence equality must redefine equals(). To illustrate, suppose users of AStringSet are not happy with the default behavior of equals(). We can add to the class the following to override the default implementation:

```java
public boolean equals(Object otherObject) {
    if (!((otherObject instanceof AStringHistory))
        return false;
    AStringHistory otherStringHistory = (AStringHistory) otherObject;
    if (size != otherStringHistory.size)
        return false;
    for (int index = 0; index < size; index++)
        if (!contents[index].equals(otherStringHistory.contents[index]))
```
The class does an element-by-element comparison of the two sets to determine if they are equal. To do so, it accesses the contents and size variables of the two instances that are compared. These variables are actually defined in the superclass AStringHistory. In fact, AStringDatabase and AStringSet do not define any instance variables. Thus, it is better to move the method to AStringHistory, thereby allowing all instances of AStringHistory and its subtypes to use it.

### Accessing arbitrary variables/methods of remote instances

The implementation above illustrates a feature of Java we have not seen before. When a method is called on an instance I of class C, it is possible not only to access arbitrary variables and methods of I but also all other instances of class C. As we saw earlier, to access instances of I, we either omit the target instance:

```java
contents[index];
```
or use the keyword `this` to identify it:

```java
this. contents[index];
```

To access the variables of some other instance of the class, which we will refer to as a remote instance, we replace this with some variable holding a pointer to the instance. In the example above, `otherStringHistory` holds a pointer to the remote instance. Hence we use this variable to indicate the target object:

```java
otherStringHistory.contents[index];
```

We can use the same syntax to refer to arbitrary methods of the remote instance. In general, in a class definition, all methods and variables of all instances of the class are visible.

### Accessing public vs. arbitrary members of remote instances

In the example above, we broke an important rule given before by using a class to type a variable:

```java
AStringHistory otherStringHistory = (AStringHistory) otherObject;
```

Had we used the interface of the class to type it

```java
StringHistory otherStringHistory = (StringHistory) otherObject;
```
we would not have been able to access non public members (methods/variables) of the remote instance as these are not defined by interface. Sometimes in a class, it is necessary to access internal members of remote instances. In that case, we are forced to use it for typing. However, in this example, the interfaces exposes the required information, as illustrated by the following rewrite of equals:

```java
public boolean equals(Object otherObject) {
```
if (!(otherObject instanceof StringHistory))
    return false;
StringHistory otherStringHistory = (StringHistory) otherObject;
if (size != otherStringHistory.size())
    return false;
for (int index = 0; index < size; index++)
    if (!contents[index].equals(otherStringHistory.elementAt(index)))
        return false;
return true;
}

This implementation is more polymorphic in that it allows us to compare arbitrary implementations of StringHistory and not only those that are instances of AStringHistory. It is slightly less efficient because the variables of the remote instance are access indirectly, through public methods, rather than directly. In most applications, this inefficiency is not a problem.

**Overriding vs. Overloading**

An even more elegant implementation of equals() is given below:

```java
public boolean equals(StringHistory otherStringHistory) {
    if (size != otherStringHistory.size())
        return false;
    for (int index = 0; index < size; index++)
        if (!contents[index].equals(otherStringHistory.elementAt(index)))
            return false;
    return true;
}
```

This implementation does not have use the `instanceof` operation, which makes the code messy. However, this implementation does not really override the equals() method inherited from Object as its parameter type is different. Hence it overloads rather than overrides the inherited method. As a result, the method does not hide the inherited method; both methods are available for invocation. If we wish to invoke the overloaded method on instances of StringHistory, we must add the signature (header) of the method to the interface. Once we do that, the syntax for invoking it is exactly the same as the one for the inherited method:

```java
StringHistory stringHistory1 = new AStringHistory();
StringHistory stringHistory2 = new AStringHistory();
stringHistory1.equals(stringHistory2);
```

The equals that accepts the more specific argument type, StringHistory, is called, even though the equals() in objects would also be legal.

On the other hand, the call:

```java
stringHistory1.equals("Not an instance of StringHistory");
```

would call the equals() in Object.
It seems that overloading and overriding in this example give the same results. If the argument is an instance of StringHistory, then it seems the more specialized equals() is called. Otherwise, in the overriding case, the specialized method is called, which returns false, and in the overloaded case, the more general method is called, which also return false.

However, they can indeed give different results. The reason is that what matters, when overload resolution is done, is not the type of the actual object assigned to a variable but the type of the variable. The reason is that overload resolution is done at compile time, and at this time, it is not possible to determine the exact type of the objects that will be assigned to it. Thus if we type the second instance of StringHistory as Object:

```java
StringHistory stringHistory1 = new AStringHistory();
Object stringHistory2 = new AStringHistory();
stringHistory1.equals(stringHistory2);
```

then the more general equals() will be called and the result will be false. Even if we use StringHistory to type both variables, we can use a cast to call the equals() defined in Object:

```java
StringHistory stringHistory1 = new AStringHistory();
StringHistory stringHistory2 = new AStringHistory();
stringHistory1.equals((Object)stringHistory2);
```

### Annotations and Making Override Explicit

As overloaded and overriding can lead to different results, it is important to not accidentally overload when we intended to override, and vice versa. To prevent such mistakes, Java supports override annotations illustrated below:

```java
@Override
public boolean equals(Object otherObject) { ... }
```

Annotation is a typed “comment” about a method, class, or package that can be processed by some tool such as compiler, Eclipse or ObjectEditor at compile/execution time. It starts with the character, @, as illustrated below. Java supports several types of annotations. The override annotation type is associated with a method and tells Java that the method overrides an existing method. If the method does not in fact do so, the compiler will give an error. In the above example, since the method signature matches that of the equals() method of Object, no error will be given. Similarly, the following annotated method definition in AStringSet will give no error:

```java
@Override
public void addElement(String element) { ... }
```

On the other hand, the following method declaration in AStringHistory will give an error as it overloads rather than overrides the Object equals() method:

```java
@Override
```
public boolean equals(StringHistory otherStringHistory) { ... }

Interestingly, the following annotated method declaration in AStringHistory does not give an error:

```java
@Override
public String elementAt (int index) { return contents[index]; }
```

Recall that AStringHistory is a direct subclass of Object, which does not define an `elementAt()` method. In the definition of the override annotation, Java does not distinguish between IS-A and inheritance. An override annotation for method M in class C indicates that M is a(n) (re)implementation of some M declared in some type T, where C IS-A T. In other words, Java considers the implementation of an interface method also as overriding. This is inconsistent with all definitions of overriding in the literature.

Java allows programmers to define their own annotation types to be processed by tools written by them. This extensibility is used in the design of ObjectEditor. For example, the following annotated interface declaration tells ObjectEditor to not consider the interface as an atomic shape type even though it follows all the rules of a point type by having `int X` and `Y` properties and including the substring `Point` in its name:

```java
public interface NotAPoint {
    int getX();
    int getY();
}
```

As the declaration shows, an annotation can take arguments. In Java, an annotation is actually represented by an interface or class. The arguments to the annotation are passed to a special method defined by the interface or class.

**Overloading and IS-A**

As we saw above, the is-a relation is used in overload resolution: if a call matches multiple overloaded methods, then the one that declared more specific formal arguments is used. In the example above, the formal argument of the `equals()` in AStringHistory is `StringHistory`, which is more specific than `StringHistory`.

To better understand this scheme and show issues raised by, let us consider the following two variations of `equals`, which could be defined in arbitrary classes:

```java
public boolean equals (StringHistory stringHistory1,
                        Object stringHistory2) {
    ...
}
public boolean equals (Object stringHistory1,
                        StringHistory stringHistory2) {
    ...
}
```
In these implementations, the two instances that are to be compared are both passed as arguments. Hence it does not matter on what objects these methods are actually called (which means they should really be static methods).

Now suppose we make the call:

```
equals(new AStringSet(), new AStringSet());
```

Both overloaded methods can accept the two arguments – which one should be called? Neither class has formal parameters that are of more specific types. Thus, the correct call is ambiguous, and Java will say so at compile time. We can, of course, use casts to disambiguate:

```
equals(new AStringSet(), (Object) new AStringSet());
```

In this case the first equals() is called.

If we add a third equals method:

```java
public boolean equals (StringHistory stringHistory1,
                      StringHistory stringHistory2) {
```

Then the call:

```
equals(new AStringSet(), new AStringSet());
```

invokes this new overloaded method as each of its formal parameters is more specific than the corresponding formal parameter of the two other methods. Thus, interesting, adding an overloaded method reduces rather than increases the ambiguity of overload resolution!

Given these three definitions of equals, which one is invoked by the following call?

```
equals(null, null);
```

`null` can be typed as any object type, so it may seems that the call is still ambiguous. However, as the types of the formal parameters of the third equals() are most specific, there is really no ambiguity, and the this method is called.

Suppose we add a fourth equals(), one that takes Point arguments:

```java
public boolean equals (Point point1, Point point2) {
```

Now the call:
equals(null, null);

is indeed ambiguous as none of the matching equals() declared more specific types. We can resolve the ambiguity by explicitly casting null:

equals ((StringHistory)null, (StringHistory)null);

**Shallow Copy**

Let us finish our exploration of the class Object by looking finally at the method clone(). To motivate it, let us return to the Point example.

```java
p1 = new ACartesianPoint(200, 200);
p2 = p1;
p1.setX (100);
System.out.println(p2.getX() == p1.getX());
```

As we saw before, the code above will print true. The reason is that assignment simply copies pointers. What if did not want p1 and p2 to share the same object? Instead, we wanted the initial value of p2 to be a copy of the object referenced by p1; and later wanted to change the copy without affecting the original object? We might want to do so because we want a backup of p1 to which we would like to revert later.

Since assignment does not do the job for us, we can extract the information in ACartesianPoint and use this to create a new instance with the same state:

```java
p1 = new ACartesianPoint(200, 200);
p2 = new ACartesianPoint (p1.getX(), p1.getY());
p1.setX (100);
System.out.println(p2.getX() == p1.getX());
```

This time the output will be false. However, this approach requires the copier to do the copying, which may not seem much work in this case, but would be more if the object to be copied had several instance variables. Therefore, a better approach is to make the copied object do the work of copying. That is, the copier should simply invoke a method on the object to be copied that returns a copy. The clone() method in Object is such a method. As its name indicates, it returns a copy of the object on which it is invoked. Since the method can be invoked on an object of any type, its return type is Object:

```java
// defined in Object
public Object clone() { ... }
```

This return type can be cast to the actual type, as shown below:

```java
p1 = new ACartesianPoint(200, 200);
p2 = (Point) p1.clone();
p1.setX (100);
System.out.println(p2.getX() == p1.getX());
```

Again, the output is false, as clone() creates a new copy with separate instance variables.
To better understand the notion of copying, let us consider instances of a more complicated type. Consider the following code:

```java
        p1 = new ABoundedPoint(75, 75, new ACartesianPoint(50,50), new ACartesianPoint(100,100) );
p2 = (BoundedPoint) p1.clone();
p1.setX (100);
p1.getUpperLeftCorner().setX(200);
System.out.println(p2.getX() == p1.getX());
System.out.println (p1.getUpperLeftCorner().getX() == p2.getUpperLeftCorner().getX() );
```

The first output will print false but the second one will, in fact, print true. The reason is that Object implements clone() by simply making a copy of the memory block of the copied object, as shown in the figure below:

Here, the clone() method, when executed on the object, ABoundedPoint@48, creates a new object, ABoundedPoint@48, whose memory block is a copy of the memory block of the first object. The two primitive instance variables, x and y, of the new copy are assigned copies of the values of the x and y
variables of the original object. The two pointer variables in the copy, however are the assigned copies of the *pointers* to the (Point) objects referenced by the upperLeftCorner and lowerRightCorner of the original object. These subobjects of the original object are not themselves copied. This means that object variables in the copy point to the same objects as the corresponding variables in the original object. The second println() call returns true as p1.getUpperLeftCorner() and p2. getUpperLeftCorner() refer to the same object, ACartesianPoint@8. This is shown graphically in the figure below:

Each line in the picture represents an instance variable of the object. A copy that simply duplicates the memory block of the copied object is called a *shallow copy*. The word shallow indicates that it copies only the top level of the physical structure of the object.

**Deep Copy**

A copy that also copies memory blocks of components of the copied object is called a *deep copy*. The following figure illustrates deep copy of our example object:
Such a copy is not provided by Java, so we must override the clone() method to implement it:

```java
public Object clone() {
    return new ABoundedPoint(x, y, (Point) upperLeftCorner.clone(),
                              (Point) lowerRightCorner.clone());
}
```

Here we construct a new instance of ABoundedPoint that has a copy of the objects to which the upperLeftCorner and lowerRightCorner instance variables point.

This copy, however, does not always work. The problem is illustrated by the example below:

```java
p1 = new ABoundedPoint(75, 75, new ACartesianPoint(50, 50), new ACartesianPoint(100, 100));
p1.setUpperLeftCorner(p1);
p2 = (BoundedPoint) p1.clone();
```

Here, we have created a recursive structure, that is, a structure in which a child component points to its ancestor:

As a result, when the clone method is invoked on this object, the following call leads to an infinite recursion:
(Point) upperLeftCorner.clone()

Each time the call is made, a new copy of ABoundedPoint@48 is made, and the call is made again, which makes another copy of the object, and so on, leading to the creation of an infinite number of copies of the object, some of which are shown below:

Thus, we must be careful in either how we implement deep copy or the kind of objects on which we try to invoke this operation. Recursive structures cannot be banned as they are very useful. A class of problems called graph problems require such structures.

The Java Object class avoids this problem in its implementation of clone() by providing a shallow copy. However, another component of Java, called serialization, which makes object copies that are written to files or sent across the network, does support deep copy, and thus can lead to the problem above. In fact, I once spent a whole day trying to debug a student’s program that tried to serialize a recursive structure and send it across the network – the serialization implementation available at that time did not simply say that it had run out of free memory; instead it overwrote existing memory, thereby changing objects that were not even copied!

**Supporting Multiple Copy Operations**
One way to reduce the problem of copying recursive structures is to associate each object with both a shallow and deep copy, and allow the deep copy to determine if shallow or deep copies of components are made. This solution is illustrated below for our example class, `ABoundedPoint`:

```java
public Object shallowCopy() {
    return new ABoundedPoint(x, y, upperLeftCorner, lowerRightCorner);
}

public Object deepCopy() {
    return new ABoundedPoint(x, y, (Point) upperLeftCorner.shallowCopy(),
                              (Point) lowerRightCorner.shallowCopy());
}
```

The “deep” copy in this solution is not a full deep copy as it does not copy all levels in the physical structure of the copied object – it is simply a deeper copy than a shallow copy. However, a call to it will never lead to infinite recursion.

An extension of the approach of providing both a shallow and deep(er) copy is taken in Smalltalk. Each Smalltalk object provides three copy methods: shallow, deep, and regular copy. The shallow copy is like the Object clone() method: It creates a new object and assigns instance variables of the copied object to corresponding instance variables of new object. The deep copy creates a new object and assigns a regular copy of each instance variable of the copied object to corresponding instance variable of new object. The semantics of the regular copy of an object of some class C is defined by that class. It is expected to either be a shallow or deep copy – the programmer of each class defines which of these two choices is taken. By default, the regular copy is a shallow copy.

**Detecting Recursion**

The multiple copy solution does not, of course, work when a full deep copy is needed. In this situation, recursive structures can be handled by detecting recursion while performing the copy operation, that is, before copying a component, detecting if the component has already been copied in the operation. When recursion is detected, we can either not copy the component, or give an error, or create an identical or isomorphic structure. In the example above, we could create an isomorphic structure y creating another instance of a BoundedPoint whose upper left corner points to it.

ObjectEditor faces a similar problem when creating a widget structure for the logical structure of an object. The current version detects recursion and does not create a widget for an component for which it has already created a widget earlier.

**Other Object Methods**

The discussion above helps us better understand three operations provided by class Object: toString(), equals(), and clone(). This class provides several other methods:
Objects and Interfaces

As we have seen above, it is possible to invoke an Object method on any object variable. If the variable is typed by a class, this makes sense, as it is possible to invoke any method declared in the class or its super class chain; and Object is the last type in any superclass chain. However, if the variable is typed by an interface, this rule does not make sense, as Object is a class and not an interface, and thus cannot be on the supertype chain of any interface. Yet, if we type a variable by an interface:

```java
StringHistory stringHistory1, stringHistory2;
```

we can indeed call Object methods on that variable:

```java
stringHistory1.equals(stringHistory2);
```

The reason this is legal is that Java uses a special rule to allow all interfaces to “inherit” Object methods. The cleaner solution would have been to associate Object with an interface, and make this interface the last interface in the supertype chain of all interfaces. Unfortunately, the designers of the language and libraries have had a schizophrenic attitude towards interfaces, using them in some situation and not in others. Had they followed our rule of making every class implement one or more interfaces, we would not have this fundamental problem with Object methods.

Garbage Collection

Suppose we execute the following code:

```java
Point p1 = new ACartesianPoint(100,100);
Point p2 = new ACartesianPoint(150,75);
```

Two new variables are created, which point to different objects, as shown in the figures below:
What if we now execute the following code:

```plaintext
p2 = p1;
```

Now both variables point to the same object, and the object to which `p2` pointed cannot ever be accessed. This is shown below:
As no variable refers to the object, it is *garbage collected*. With each object, Java keeps a count, called a *reference count*, that tracks how many object variables store pointers to it. When this count goes to zero, it collects the object as garbage, since no other variable will ever be able to point to it again.

Automatic garbage collection is a really nice feature of Java since in most traditional languages such as C, the programmer is responsible for deleting objects. In such languages, the danger is that we may accidentally delete something that is being used, thereby creating *dangling pointers* to it, or forget to delete something that is not being used, thereby creating a *memory leak* that keeps wasting memory. On the other hand, garbage collection makes Java slower, since Java must interrupt our program execution to do garbage collection.

**Collection Inheritance in Java**

In this chapter, we have seen how inheritance can be used while defining collection classes. The Java API also addresses this issue. The hierarchy it creates has some similarity with the one we created. Part of the hierarchy is given below:
It defines two interfaces, Set and List, which are both subtypes of Collection. This is analogous to StringDatabase and StringSet being subtypes of StringHistory. Vector, ArrayList, and LinkedList provide three different implementations of the List interface. In earlier versions, Vector did not implement any interface. When Java programmers felt the need for additional implementations with the same functionality, they created the List interface. Both Vector and ArrayList use arrays to create a variable-sized collection — the main difference is in how they grow when new elements are added. LinkedList uses a different data structure, called a linked list, which you will study later. Similarly, there are a variety of classes that implement the Set interface, we see only HashSet in this figure.

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.
- 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.
- All variables of an object, including its inherited variables, are stored in a single memory block.
- Overriding != overloading
- If two overloaded method match a call, the one with more specific parameter types is chosen.
- Casts may be needed to disambiguate between overloaded methods.
- equals() != ==
- Copies can be shallow or deep depending on whether components of the copied object are themselves copied or not.
- Recursive structures interfere with deep copy.
- Unreferenced objects are garbage collected.
Exercises